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

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

March 17, 2025

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

Indeed, there have been numerous misunderstandings on what exactly a given algorithm is supposed to realize in what kinds of environments. Moreover, the abundance of subtly different computational models complicates comparisons between different algorithms. Charron-Bost and Schiper introduced the Heard-Of model for representing algorithms and failure assumptions in a uniform framework, simplifying comparisons between algorithms.

In this contribution, we represent the Heard-Of model in Isabelle/HOL. We define two semantics of runs of algorithms with different unit of atomicity and relate these through a *reduction theorem* that allows us to verify algorithms in the coarse-grained semantics (where proofs are easier) and infer their correctness for the fine-grained one (which corresponds to actual executions). We instantiate the framework by verifying six Consensus algorithms that differ in the underlying algorithmic mechanisms and the kinds of faults they tolerate.

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

Contents

1 Introduction

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

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

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

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

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

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

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

theory *HOModel*
imports *Main*
begin

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

2 Heard-Of Algorithms

2.1 The Consensus Problem

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

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

Integrity. Processes can only decide initially proposed values.

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

Termination. Every process decides eventually.

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

type-synonym
$('proc,'pst)\ run = nat \Rightarrow 'proc \Rightarrow 'pst$

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

A variant of the Consensus problem replaces the Integrity requirement by

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

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

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

lemma *consensus-then-weak-consensus*:
assumes *consensus vals dec rho*
shows *weak-consensus vals dec rho*
$\langle proof \rangle$

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

lemma *binary-weak-consensus-then-consensus*:
assumes *bc*: $weak\text{-}consensus\ (vals::'proc \Rightarrow bool)\ dec\ rho$
shows *consensus vals dec rho*
$\langle proof \rangle$

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

2.2 A Generic Representation of Heard-Of Algorithms

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

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

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

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

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

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

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

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

We represent an algorithm by a record as follows.

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

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

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

definition *initState* **where**
$\quad initState\ alg\ p\ st \equiv CinitState\ alg\ p\ st\ undefined$

definition *nextState* **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.

type-synonym
$'proc\ HO = 'proc \Rightarrow 'proc\ set$

type-synonym
$'proc\ coord = 'proc \Rightarrow 'proc$

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

The following definitions formalize this idea. We define "coarse-grained" executions whose unit of atomicity is the round of execution. At each round, the entire collection of processes performs a transition according to the *CnextState* function of the algorithm. Consequently, a system state is simply described by a configuration, i.e. a function assigning a process state to every process. This definition of executions may appear surprising for an asynchronous distributed system, but it simplifies system verification, compared to a "fine-grained" execution model that records individual events such as message sending and reception or local transitions. We will justify later why the "coarse-grained" model is sufficient for verifying interesting correctness properties of HO algorithms.

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

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

Given the current configuration *cfg* and the HO and SHO sets *HOp* and *SHOp* for process p at round r, the function *SHOmsgVectors* computes the set of possible vectors of messages that process p may receive. For processes $q \notin HOp$, p receives no message (represented as value *None*). For processes $q \in SHOp$, p receives the message that q computed according to the *sendMsg* function of the algorithm. For the remaining processes $q \in HOp - SHOp$, p may receive some arbitrary value.

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

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

definition *CSHOnextConfig* **where**
$$\begin{aligned}
&\mathit{CSHOnextConfig}\ A\ r\ \mathit{cfg}\ \mathit{HO}\ \mathit{SHO}\ \mathit{coord}\ \mathit{cfg}' \equiv\\
&\quad \forall p.\ \exists \mu \in \mathit{SHOmsgVectors}\ A\ r\ p\ \mathit{cfg}\ (\mathit{HO}\ p)\ (\mathit{SHO}\ p).\\
&\qquad\quad \mathit{CnextState}\ A\ r\ p\ (\mathit{cfg}\ p)\ \mu\ (\mathit{coord}\ p)\ (\mathit{cfg}'\ p)
\end{aligned}$$

definition *CSHORun* **where**
$$\begin{aligned}
&\mathit{CSHORun}\ A\ \mathit{rho}\ \mathit{HOs}\ \mathit{SHOs}\ \mathit{coords} \equiv\\
&\qquad \mathit{CHOinitConfig}\ A\ (\mathit{rho}\ 0)\ (\mathit{coords}\ 0)\\
&\quad \wedge (\forall r.\ \mathit{CSHOnextConfig}\ A\ r\ (\mathit{rho}\ r)\ (\mathit{HOs}\ r)\ (\mathit{SHOs}\ r)\ (\mathit{coords}\ (\mathit{Suc}\ r))\\
&\qquad\qquad\qquad (\mathit{rho}\ (\mathit{Suc}\ r)))
\end{aligned}$$

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**
$$\mathit{HOinitConfig}\ A\ \mathit{cfg} \equiv \mathit{CHOinitConfig}\ A\ \mathit{cfg}\ (\lambda q.\ \mathit{undefined})$$

lemma *HOinitConfig-eq*:
$$\mathit{HOinitConfig}\ A\ \mathit{cfg} = (\forall p.\ \mathit{initState}\ A\ p\ (\mathit{cfg}\ p))$$
$\langle \mathit{proof} \rangle$

definition *SHOnextConfig* **where**
$$\begin{aligned}
&\mathit{SHOnextConfig}\ A\ r\ \mathit{cfg}\ \mathit{HO}\ \mathit{SHO}\ \mathit{cfg}' \equiv\\
&\quad \mathit{CSHOnextConfig}\ A\ r\ \mathit{cfg}\ \mathit{HO}\ \mathit{SHO}\ (\lambda q.\ \mathit{undefined})\ \mathit{cfg}'
\end{aligned}$$

lemma *SHOnextConfig-eq*:
$$\begin{aligned}
&\mathit{SHOnextConfig}\ A\ r\ \mathit{cfg}\ \mathit{HO}\ \mathit{SHO}\ \mathit{cfg}' =\\
&\quad (\forall p.\ \exists \mu \in \mathit{SHOmsgVectors}\ A\ r\ p\ \mathit{cfg}\ (\mathit{HO}\ p)\ (\mathit{SHO}\ p).\\
&\qquad\quad \mathit{nextState}\ A\ r\ p\ (\mathit{cfg}\ p)\ \mu\ (\mathit{cfg}'\ p))
\end{aligned}$$
$\langle \mathit{proof} \rangle$

definition *SHORun* **where**
$$\begin{aligned}
&\mathit{SHORun}\ A\ \mathit{rho}\ \mathit{HOs}\ \mathit{SHOs} \equiv\\
&\quad \mathit{CSHORun}\ A\ \mathit{rho}\ \mathit{HOs}\ \mathit{SHOs}\ (\lambda r\ q.\ \mathit{undefined})
\end{aligned}$$

lemma *SHORun-eq*:
$$\begin{aligned}
&\mathit{SHORun}\ A\ \mathit{rho}\ \mathit{HOs}\ \mathit{SHOs} =\\
&\qquad (\mathit{HOinitConfig}\ A\ (\mathit{rho}\ 0)\\
&\quad \wedge (\forall r.\ \mathit{SHOnextConfig}\ A\ r\ (\mathit{rho}\ r)\ (\mathit{HOs}\ r)\ (\mathit{SHOs}\ r)\ (\mathit{rho}\ (\mathit{Suc}\ r))))
\end{aligned}$$
$\langle \mathit{proof} \rangle$

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**
$HOrcvdMsgs\ A\ r\ p\ HO\ cfg \equiv$
$\lambda q.\ if\ q \in HO\ then\ Some\ (sendMsg\ A\ r\ q\ p\ (cfg\ q))\ else\ None$

lemma *SHOmsgVectors-HO*:
$SHOmsgVectors\ A\ r\ p\ cfg\ HO\ HO = \{HOrcvdMsgs\ A\ r\ p\ HO\ cfg\}$
$\langle proof \rangle$

With coordinators

definition *CHOnextConfig* **where**
$CHOnextConfig\ A\ r\ cfg\ HO\ coord\ cfg' \equiv$
$CSHOnextConfig\ A\ r\ cfg\ HO\ HO\ coord\ cfg'$

lemma *CHOnextConfig-eq*:
$CHOnextConfig\ A\ r\ cfg\ HO\ coord\ cfg' =$
$(\forall p.\ CnextState\ A\ r\ p\ (cfg\ p)\ (HOrcvdMsgs\ A\ r\ p\ (HO\ p)\ cfg)$
$(coord\ p)\ (cfg'\ p))$
$\langle proof \rangle$

definition *CHORun* **where**
$CHORun\ A\ rho\ HOs\ coords \equiv CSHORun\ A\ rho\ HOs\ HOs\ coords$

lemma *CHORun-eq*:
$CHORun\ A\ rho\ HOs\ coords =$
$(CHOinitConfig\ A\ (rho\ 0)\ (coords\ 0)$
$\wedge (\forall r.\ CHOnextConfig\ A\ r\ (rho\ r)\ (HOs\ r)\ (coords\ (Suc\ r))\ (rho\ (Suc\ r))))$
$\langle proof \rangle$

Without coordinators

definition *HOnextConfig* **where**
$HOnextConfig\ A\ r\ cfg\ HO\ cfg' \equiv SHOnextConfig\ A\ r\ cfg\ HO\ HO\ cfg'$

lemma *HOnextConfig-eq*:
$HOnextConfig\ A\ r\ cfg\ HO\ cfg' =$
$(\forall p.\ nextState\ A\ r\ p\ (cfg\ p)\ (HOrcvdMsgs\ A\ r\ p\ (HO\ p)\ cfg)\ (cfg'\ p))$
$\langle proof \rangle$

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

lemma *HORun-eq*:
$HORun\ A\ rho\ HOs =$
$(\ \ HOinitConfig\ A\ (rho\ 0)$
$\wedge (\forall r.\ HOnextConfig\ A\ r\ (rho\ r)\ (HOs\ r)\ (rho\ (Suc\ r))))$
$\langle proof \rangle$

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

lemma *CHORun-0*:
  assumes *CHORun A rho HOs coords*
    and $\bigwedge cfg.\ CHOinitConfig\ A\ cfg\ (coords\ 0) \Longrightarrow P\ cfg$
  shows $P\ (rho\ 0)$
$\langle proof \rangle$

lemma *CHORun-Suc*:
  assumes *CHORun A rho HOs coords*
  and $\bigwedge r.\ CHOnextConfig\ A\ r\ (rho\ r)\ (HOs\ r)\ (coords\ (Suc\ r))\ (rho\ (Suc\ r))$
    $\Longrightarrow P\ r$
  shows $P\ n$
$\langle proof \rangle$

lemma *CHORun-induct*:
  assumes *run*: *CHORun A rho HOs coords*
  and *init*: $CHOinitConfig\ A\ (rho\ 0)\ (coords\ 0) \Longrightarrow P\ 0$
  and *step*: $\bigwedge r.\ \llbracket\ P\ r;\ CHOnextConfig\ A\ r\ (rho\ r)\ (HOs\ r)\ (coords\ (Suc\ r))$
    $(rho\ (Suc\ r))\ \rrbracket \Longrightarrow P\ (Suc\ r)$
  shows $P\ n$
$\langle proof \rangle$

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

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

record $('proc,\ 'pst,\ 'msg)\ HOMachine = ('proc,\ 'pst,\ 'msg)\ CHOAlgorithm\ +$
  $HOcommPerRd::'proc\ HO \Rightarrow bool$
  $HOcommGlobal::(nat \Rightarrow 'proc\ HO) \Rightarrow bool$

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

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

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

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

3 Reduction Theorem

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

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

As a corollary, interesting properties remain valid in the fine-grained semantics if they hold in the coarse-grained semantics. It is therefore enough to verify such properties in the coarse-grained semantics, which is much easier to reason about. The essential restriction is that properties may not depend on states of different processes occurring simultaneously. (For example, the coarse-grained semantics ensures by definition that all processes execute the same round at any instant, which is obviously not true of the fine-grained semantics.) We claim that all "reasonable" properties of fault-tolerant distributed algorithms are preserved by our reduction. For example, the Consensus (and Weak Consensus) problems fall into this class.

The proofs follow Chaouch-Saad et al. [4], where the reduction theorem was proved for uncoordinated HO algorithms.

3.1 Fine-Grained Semantics

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

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

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

record $('pst, \ 'proc, \ 'msg) \ \mathit{config} =$
  $\mathit{round} :: \ 'proc \Rightarrow \mathit{nat}$
  $\mathit{state} :: \ 'proc \Rightarrow \ 'pst$
  $\mathit{sent} :: \ 'proc \Rightarrow \ 'proc \ \mathit{set}$
  $\mathit{rcvd} :: \ 'proc \Rightarrow \ 'proc \Rightarrow \ 'msg \ \mathit{option}$
  $\mathit{network} :: \ ('proc * \mathit{nat} * \ 'proc * \ 'msg) \ \mathit{set}$

type-synonym $('pst \ ,'proc \ , \ 'msg) \ \mathit{fgrun} = \mathit{nat} \Rightarrow ('pst, \ 'proc, \ 'msg) \ \mathit{config}$

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

definition *fg-init-config* **where**
  $\mathit{fg\text{-}init\text{-}config} \ A \ (\mathit{config}::('pst,'proc, \ 'msg) \ \mathit{config}) \ (\mathit{coord}::'proc \ \mathit{coord}) \equiv$
    $\mathit{round} \ \mathit{config} = (\lambda p. \ 0)$
  $\wedge \ (\forall p. \ \mathit{CinitState} \ A \ p \ (\mathit{state} \ \mathit{config} \ p) \ (\mathit{coord} \ p))$
  $\wedge \ \mathit{sent} \ \mathit{config} = (\lambda p. \ \{\})$
  $\wedge \ \mathit{rcvd} \ \mathit{config} = (\lambda p \ q. \ \mathit{None})$
  $\wedge \ \mathit{network} \ \mathit{config} = \{\}$

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

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

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

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

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

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

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

definition *fg-local* **where**

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

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

definition *fg-next-config* **where**
fg-next-config A p HO SHO crd config config' $\equiv$
  ($\exists$ *q. fg-send-msg A p q config config'*)
  $\vee$ ($\exists$ *q. fg-rcv-msg p q HO SHO config config'*)
  $\vee$ *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**
fg-run A rho HOs SHOs coords $\equiv$
  fg-init-config A (*rho 0*) (*coords 0*)
  $\wedge$ ($\forall$ *i.* $\exists$ *p. fg-next-config A p*
    (*HOs* (*round* (*rho i*) *p*) *p*)
    (*SHOs* (*round* (*rho i*) *p*) *p*)
    (*coords* (*round* (*rho* (*Suc i*)) *p*) *p*)
    (*rho i*) (*rho* (*Suc i*)))
  $\wedge$ ($\forall$ *p r.* $\exists$ *n. 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**
fg-start-round rho p r $\equiv$ *LEAST* (*n::nat*). *round* (*rho n*) *p* $=$ *r*

3.2 Properties of the Fine-Grained Semantics

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

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

lemma *fg-state-change*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
  and *rd*: *round* (*rho* (*Suc n*)) *p* $=$ *round* (*rho n*) *p*
shows *state* (*rho* (*Suc n*)) *p* $=$ *state* (*rho n*) *p*
$\langle$*proof*$\rangle$

Round numbers never decrease.

lemma *fg-round-numbers-increase*:
assumes *rho*: *fg-run A rho HOs SHOs coords* **and** *n*: $n \leq m$

shows *round* (*rho* *n*) *p* $\leq$ *round* (*rho* *m*) *p*
⟨*proof*⟩

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

lemma *fg-same-round-same-state*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
and *rd*: *round* (*rho m*) *p* = *round* (*rho n*) *p*
shows *state* (*rho m*) *p* = *state* (*rho n*) *p*
⟨*proof*⟩

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

lemma *fg-start-round*:
assumes *fg-run A rho HOs SHOs coords*
shows *round* (*rho* (*fg-start-round rho p r*)) *p* = *r*
⟨*proof*⟩

lemma *fg-start-round-smallest*:
assumes *round* (*rho k*) *p* = *r*
shows *fg-start-round rho p r* $\leq$ (*k*::*nat*)
⟨*proof*⟩

lemma *fg-start-round-later*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
and *r*: *round* (*rho n*) *p* = *r* **and** *r'*: *r* < *r'*
shows *n* < *fg-start-round rho p r'* (**is** - < *?start*)
⟨*proof*⟩

lemma *fg-start-round-0*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
shows *fg-start-round rho p 0* = *0*
⟨*proof*⟩

lemma *fg-start-round-strict-mono*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
shows *strict-mono* (*fg-start-round rho p*)
⟨*proof*⟩

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

lemma *fg-round-between-start-rounds*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
and *1*: *fg-start-round rho p r* $\leq$ *n*

  and *2*: $n <$ *fg-start-round rho p* (*Suc r*)
shows *round* (*rho n*) $p = r$ (**is** *?rd* $= r$)
⟨*proof*⟩

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

lemma *fg-local-transition-from-round*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
obtains n **where** *round* (*rho n*) $p = r$
  and *fg-start-round rho p* (*Suc r*) = *Suc n*
  and *fg-local A p* (*HOs r p*) (*coords* (*Suc r*) *p*) (*rho n*) (*rho* (*Suc n*))
⟨*proof*⟩

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

lemma *fg-invariant1*:
  assumes *rho*: *fg-run A rho HOs SHOs coords*
    and *m*: $(p,r,q,m) \in$ *network* (*rho n*) (**is** *?msg n*)
  shows $m =$ *sendMsg A r p q* (*state* (*rho* (*fg-start-round rho p r*)) *p*)
⟨*proof*⟩

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

lemma *fg-invariant2a*:
  assumes *rho*: *fg-run A rho HOs SHOs coords*
    and *m*: *rcvd* (*rho n*) *q p* = *Some m* (**is** *?rcvd n*)
  shows $p \in$ *HOs* (*round* (*rho n*) *q*) *q*
  (**is** $p \in$ *HOs* (*?rd n*) *q* **is** *?P n*)
⟨*proof*⟩

lemma *fg-invariant2b*:
  assumes *rho*: *fg-run A rho HOs SHOs coords*
    and *m*: *rcvd* (*rho n*) *q p* = *Some m* (**is** *?rcvd n*)
    and *sho*: $p \in$ *SHOs* (*round* (*rho n*) *q*) *q* (**is** $p \in$ *SHOs* (*?rd n*) *q*)
  shows $m =$ *sendMsg A* (*?rd n*) *p q*
      (*state* (*rho* (*fg-start-round rho p* (*?rd n*))) *p*)
    (**is** *?P n*)
⟨*proof*⟩

3.3 From Fine-Grained to Coarse-Grained Runs

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

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

definition *coarse-run* **where**
coarse-run rho r p $\equiv$ *state* (*rho* (*fg-start-round rho p r*)) *p*

theorem *reduction*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
shows *CSHORun A* (*coarse-run rho*) *HOs SHOs coords*
(**is** *CSHORun - ?cr - - -*)
⟨*proof*⟩

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

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

Local similarity is an equivalence relation.

lemma *locally-similar-refl*: *locally-similar* $\sigma\ \sigma$
⟨*proof*⟩

lemma *locally-similar-sym*: *locally-similar* $\sigma\ \tau \Longrightarrow$ *locally-similar* $\tau\ \sigma$
⟨*proof*⟩

lemma *locally-similar-trans* [*trans*]:
locally-similar $\varrho\ \sigma \Longrightarrow$ *locally-similar* $\sigma\ \tau \Longrightarrow$ *locally-similar* $\varrho\ \tau$
⟨*proof*⟩

lemma *local-property-eq*:
$\quad$ *local-property* $P = (\forall\, \sigma\ \tau.\ \textit{locally-similar}\ \sigma\ \tau \longrightarrow P\ \sigma = P\ \tau)$
$\quad$ $\langle proof \rangle$

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*:
$\quad$ **assumes** *rho*: *fg-run A rho HOs SHOs coords*
$\quad$ **shows** *locally-similar* (*state* $\circ$ *rho*) (*coarse-run rho*)
$\langle proof \rangle$

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*:
$\quad$ **assumes** *rho*: *fg-run A rho HOs SHOs coords*
$\quad\quad$ **and** *P*: *local-property P*
$\quad\quad$ **and** *coarse-correct*:
$$\bigwedge \textit{crho}.\ [\![\ \textit{CSHORun A crho HOs SHOs coords};\ \textit{crho}\ 0 = \textit{state}\ (\textit{rho}\ 0)]\!] \Longrightarrow P\ \textit{crho}$$
$\quad$ **shows** *P* (*state* $\circ$ *rho*)
$\langle proof \rangle$

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*:
$\quad$ **assumes** *sim*: *locally-similar* $\sigma\ \tau$
$\quad\quad$ **and** *val*: $\bigwedge n.\ \textit{dec}\ (\sigma\ n\ p) = \textit{Some}\ v \Longrightarrow v \in \textit{range vals}$
$\quad\quad$ **and** *dec*: $\textit{dec}\ (\tau\ n\ p) = \textit{Some}\ v$
$\quad$ **shows** $v \in \textit{range vals}$
$\langle proof \rangle$

lemma *validity-is-local*:
$\quad$ **assumes** *sim*: *locally-similar* $\sigma\ \tau$
$\quad\quad$ **and** *val*: $\bigwedge n.\ \textit{dec}\ (\sigma\ n\ p) = \textit{Some}\ w \Longrightarrow w = v$
$\quad\quad$ **and** *dec*: $\textit{dec}\ (\tau\ n\ p) = \textit{Some}\ w$
$\quad$ **shows** $w = v$
$\langle proof \rangle$

lemma *agreement-is-local*:
$\quad$ **assumes** *sim*: *locally-similar* $\sigma\ \tau$
$\quad$ **and** *agr*: $\bigwedge m\ n.\ [\![\textit{dec}\ (\sigma\ m\ p) = \textit{Some}\ v;\ \textit{dec}\ (\sigma\ n\ q) = \textit{Some}\ w]\!] \Longrightarrow v{=}w$

and v: $dec\ (\tau\ m\ p) = Some\ v$ **and** w: $dec\ (\tau\ n\ q) = Some\ w$
shows $v = w$
$\langle proof \rangle$

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

theorem *consensus-is-local*: *local-property* (*consensus vals dec*)
$\langle proof \rangle$

theorem *weak-consensus-is-local*: *local-property* (*weak-consensus vals dec*)
$\langle proof \rangle$

end
theory *Majorities*
imports *Main*
begin

4 Utility Lemmas About Majorities

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

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

lemma *abs-majorities-intersect*:
assumes *crd*: $card\ E < card\ S + card\ T$
and s: $S \subseteq E$ **and** t: $T \subseteq E$ **and** e: *finite* E
shows $S \cap T \neq \{\}$
$\langle proof \rangle$

lemma *abs-majoritiesE*:
assumes *crd*: $card\ E < card\ S + card\ T$
and s: $S \subseteq E$ **and** t: $T \subseteq E$ **and** e: *finite* E
obtains p **where** $p \in S$ **and** $p \in T$
$\langle proof \rangle$

Special case: both sets S and T are majorities.

lemma *abs-majoritiesE'*:
assumes *Smaj*: $card\ S > (card\ E)\ div\ 2$ **and** *Tmaj*: $card\ T > (card\ E)\ div\ 2$
and s: $S \subseteq E$ **and** t: $T \subseteq E$ **and** e: *finite* E
obtains p **where** $p \in S$ **and** $p \in T$
$\langle proof \rangle$

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

lemma *majorities-intersect*:
  assumes *crd*: $card\ (UNIV::('a::finite)\ set) < card\ (S::'a\ set) + card\ T$
  shows $S \cap T \neq \{\}$
  $\langle proof \rangle$

lemma *majoritiesE*:
  assumes *crd*: $card\ (UNIV::('a::finite)\ set) < card\ (S::'a\ set) + card\ (T::'a\ set)$
  obtains p **where** $p \in S$ **and** $p \in T$
$\langle proof \rangle$

lemma *majoritiesE'*:
  assumes S: $card\ (S::('a::finite)\ set) > (card\ (UNIV::'a\ set))\ div\ 2$
  and T: $card\ (T::'a\ set) > (card\ (UNIV::'a\ set))\ div\ 2$
  obtains p **where** $p \in S$ **and** $p \in T$
$\langle proof \rangle$

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

5 Verification of the *One-Third Rule* Consensus Algorithm

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

5.1 Model of the Algorithm

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

typedecl *Proc* — the set of processes
axiomatization where *Proc-finite*: *OFCLASS*(*Proc*, *finite-class*)
instance *Proc* :: *finite* $\langle proof \rangle$

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

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

record $'val\ pstate =$
$x :: 'val$
$decide :: 'val\ option$

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

definition *OTR-initState* **where**
$OTR\text{-}initState\ p\ st \equiv decide\ st = None$

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

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

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

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

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

lemma *MFR-exists*: $\exists v.\ MFR\ msgs\ v$
$\langle proof \rangle$

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

lemma *MFR-in-msgs*:
assumes $HO{:}HOs\ m\ p \neq \{\}$
and v: $MFR\ (HOrcvdMsgs\ OTR\text{-}M\ m\ p\ (HOs\ m\ p)\ (rho\ m))\ v$
(**is** $MFR\ ?msgs\ v$)
shows $\exists q \in HOs\ m\ p.\ v = the\ (?msgs\ q)$
$\langle proof \rangle$

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

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

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

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

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

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

5.2 Communication Predicate for *One-Third Rule*

We now define the communication predicate for the *One-Third Rule* algorithm to be correct. It requires that, infinitely often, there is a round where all processes receive messages from the same set Π of processes where Π contains more than two thirds of all processes. The "per-round" part of the communication predicate is trivial.

definition *OTR-commPerRd* **where**
$OTR\text{-}commPerRd\ HOrs \equiv True$

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

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

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

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

abbreviation *OTR-M* ≡ *OTR-HOMachine*::(*Proc*, ′*val*::*linorder pstate*, ′*val*) *HOMachine*

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

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

5.4 Proof of Integrity

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

definition *VInv* **where**
  VInv rho n ≡
   let xinit = (*range* (*x* ∘ (*rho 0*)))
   in range (*x* ∘ (*rho n*)) ⊆ *xinit*
    ∧ *range* (*decide* ∘ (*rho n*)) ⊆ {*None*} ∪ (*Some* ‘ *xinit*)

lemma *vinv-invariant*:
  assumes *run*:*HORun OTR-M rho HOs*
  shows *VInv rho n*
⟨*proof*⟩

Integrity is an immediate consequence.

theorem *OTR-integrity*:
  assumes *run*:*HORun OTR-M rho HOs* **and** *dec*: *decide* (*rho n p*) = *Some v*
  shows ∃ *q*. *v* = *x* (*rho 0 q*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *A1*:
assumes *run*:*HORun OTR-M rho HOs*
and *dec*: *decide* (*rho* (*Suc n*) *p*) = *Some v*
and *chg*: *decide* (*rho* (*Suc n*) *p*) ≠ *decide* (*rho n p*) (**is** *decide ?st′* ≠ *decide ?st*)
shows (*2*∗*N*) *div 3* < *card* { *q* . *x* (*rho n q*) = *v* }
⟨*proof*⟩

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

lemma *A2*:
assumes *run*: *HORun OTR-M rho HOs*
and *HO*: (*2*∗*N*) *div 3*
< *card* { *q* . *HOrcvdMsgs OTR-M n p* (*HOs n p*) (*rho n*) *q* ≠ *None* }
and *maj*: (*2*∗*N*) *div 3* < *card* { *q* . *x* (*rho n q*) = *v* }
shows *x* (*rho* (*Suc n*) *p*) = *v*
⟨*proof*⟩

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

lemma *A3*:
assumes *run*:*HORun OTR-M rho HOs*
and *n*: (*2*∗*N*) *div 3* < *card* { *q* . *x* (*rho n q*) = *v* } (**is** *?twothird n*)
shows *?twothird* (*n*+*k*)
⟨*proof*⟩

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

lemma *A4*:
assumes *run*: *HORun OTR-M rho HOs*
and *dec*: *decide* (*rho n p*) = *Some v* (**is** *?dec n*)
shows ∀ *k*. (*2*∗*N*) *div 3* < *card* { *q* . *x* (*rho* (*n*+*k*) *q*) = *v* }
(**is** ∀ *k*. *?twothird* (*n*+*k*))
⟨*proof*⟩

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

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

lemma *A5*:
  assumes *run*:*HORun OTR-M rho HOs*
  and *p*: *decide* (*rho n p*) = *Some v*
  and *p'*: *decide* (*rho* (*n*+*k*) *p'*) = *Some w*
  shows $v = w$
⟨*proof*⟩

theorem *OTR-agreement*:
  assumes *run*:*HORun OTR-M rho HOs*
  and *p*: *decide* (*rho n p*) = *Some v*
  and *p'*: *decide* (*rho m p'*) = *Some w*
  shows $v = w$
⟨*proof*⟩

5.6 Proof of Termination

We now show that every process must eventually decide.

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

Lemma *A2* is instrumental in this proof.

theorem *OTR-termination*:
  assumes *run*: *HORun OTR-M rho HOs*
    and *commG*: *HOcommGlobal OTR-M HOs*
  shows ∃ *r v*. *decide* (*rho r p*) = *Some v*
⟨*proof*⟩

5.7 *One-Third Rule* Solves Consensus

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

theorem *OTR-consensus*:
  assumes *run*: *HORun OTR-M rho HOs* **and** *commG*: *HOcommGlobal OTR-M HOs*
  shows *consensus* ($x \circ$ (*rho 0*)) *decide rho*
  ⟨*proof*⟩

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

theorem *OTR-consensus-fg*:
  assumes *run*: *fg-run OTR-M rho HOs HOs* (λ*r q*. *undefined*)

and *commG*: *HOcommGlobal OTR-M HOs*
shows *consensus* ($\lambda p.$ *x* (*state* (*rho 0*) *p*)) *decide* (*state* $\circ$ *rho*)
(**is** *consensus ?inits* - -)
⟨*proof*⟩

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

6 Verification of the *UniformVoting* Consensus Algorithm

Algorithm *UniformVoting* is presented in [7]. It can be considered as a deterministic version of Ben-Or's well-known probabilistic Consensus algorithm [2]. We formalize in Isabelle the correctness proof given in [7], using the framework of theory *HOModel*.

6.1 Model of the Algorithm

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

typedecl *Proc* — the set of processes
axiomatization where *Proc-finite*: *OFCLASS*(*Proc*, *finite-class*)
instance *Proc* :: *finite* ⟨*proof*⟩

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 $'val\ msg =$
$\quad Val\ 'val$
$|\ ValVote\ 'val\ 'val\ option$
$|\ Null$ — dummy message in case nothing needs to be sent

definition $isValVote$ **where** $isValVote\ m \equiv \exists z\ v.\ m = ValVote\ z\ v$

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

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

fun $getvote$ **where**
$\quad getvote\ (ValVote\ z\ v) = v$

fun $getval$ **where**
$\quad getval\ (ValVote\ z\ v) = z$
$|\ getval\ (Val\ z) = z$

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

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

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

definition $msgRcvd$ **where** — processes from which some message was received
$\quad msgRcvd\ (msgs{::}\ Proc \rightharpoonup 'val\ msg) = \{q\ .\ msgs\ q \neq None\}$

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

In step 0, each process sends its current x value.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2 Communication Predicate for *UniformVoting*

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

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

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

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

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

6.3 The *UniformVoting* Heard-Of Machine

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

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

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

end
theory *UvProof*
imports *UvDefs ../Reduction*
begin

6.4 Preliminary Lemmas

At any round, given two processes p and q, there is always some process which is heard by both of them, and from which p and q have received the same message.

lemma *some-common-msg*:
assumes *HOcommPerRd UV-M* (*HOs r*)
shows $\exists$ *pq*. *pq* $\in$ *msgRcvd* (*HOrcvdMsgs UV-M r p* (*HOs r p*) (*rho r*))
$\wedge$ *pq* $\in$ *msgRcvd* (*HOrcvdMsgs UV-M r q* (*HOs r q*) (*rho r*))
$\wedge$ (*HOrcvdMsgs UV-M r p* (*HOs r p*) (*rho r*)) *pq*
= (*HOrcvdMsgs UV-M r q* (*HOs r q*) (*rho r*)) *pq*
⟨*proof*⟩

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

lemma *minval-step0*:
assumes *com*: *HOcommPerRd UV-M* (*HOs r*) **and** *s0*: *step r* = *0*
shows *smallestValRcvd* (*HOrcvdMsgs UV-M r q* (*HOs r q*) (*rho r*))
$\in$ {*v*. $\exists$ *p*. (*HOrcvdMsgs UV-M r q* (*HOs r q*) (*rho r*)) *p* = *Some* (*Val v*)}
(**is** *smallestValRcvd ?msgs* $\in$ *?vals*)
⟨*proof*⟩

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*) (*rho r*)) = {}
shows *smallestValNoVoteRcvd* (*HOrcvdMsgs UV-M r q* (*HOs r q*) (*rho r*))
$\in$ {*v* . $\exists$ *p*. (*HOrcvdMsgs UV-M r q* (*HOs r q*) (*rho r*)) *p*
= *Some* (*ValVote v None*)}
(**is** *smallestValNoVoteRcvd ?msgs* $\in$ *?vals*)
⟨*proof*⟩

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*⟩

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*⟩

6.5 Proof of Irrevocability, Agreement and Integrity

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

lemma *decide-step*:
assumes *run*: *HORun UV-M rho HOs*
and *decide*: *decide* (*rho* (*Suc r*) *p*) $\neq$ *decide* (*rho r p*)
shows *step r* = *1*

⟨*proof*⟩

No process ever decides *None*.

lemma *decide-nonnull*:
  assumes *run*: *HORun UV-M rho HOs*
    and *decide*: $decide\ (rho\ (Suc\ r)\ p) \neq decide\ (rho\ r\ p)$
  shows $decide\ (rho\ (Suc\ r)\ p) \neq None$
⟨*proof*⟩

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

lemma *msgs-unanimity*:
  assumes *run*: *HORun UV-M rho HOs*
    and *vote*: $vote\ (rho\ (Suc\ r)\ p) = Some\ v$
    and *q*: $q \in msgRcvd\ (HOrcvdMsgs\ UV\text{-}M\ r\ p\ (HOs\ r\ p)\ (rho\ r))$
      (**is** - $\in$ *msgRcvd ?msgs*)
  shows $getval\ (the\ (?msgs\ q)) = v$
⟨*proof*⟩

Any two processes can only vote for the same value.

lemma *vote-agreement*:
  assumes *run*: *HORun UV-M rho HOs*
    and *com*: $\forall r.\ HOcommPerRd\ UV\text{-}M\ (HOs\ r)$
    and *p*: $vote\ (rho\ r\ p) = Some\ v$
    and *q*: $vote\ (rho\ r\ q) = Some\ w$
  shows $v = w$
⟨*proof*⟩

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

lemma *decide-equals-x*:
  assumes *run*: *HORun UV-M rho HOs*
    and *com*: $\forall r.\ HOcommPerRd\ UV\text{-}M\ (HOs\ r)$
    and *decide*: $decide\ (rho\ (Suc\ r)\ p) \neq decide\ (rho\ r\ p)$
    and *decval*: $decide\ (rho\ (Suc\ r)\ p) = Some\ v$
  shows $x\ (rho\ (Suc\ r)\ q) = v$
⟨*proof*⟩

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

lemma *same-x-stable*:
  assumes *run*: *HORun UV-M rho HOs*
    and *comm*: $\forall r.\ HOcommPerRd\ UV\text{-}M\ (HOs\ r)$
    and *x*: $\forall p.\ x\ (rho\ r\ p) = v$
  shows $x\ (rho\ (Suc\ r)\ q) = v$
⟨*proof*⟩

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*: $HORun\ UV\text{-}M\ rho\ HOs$
    and *com*: $\forall r.\ HOcommPerRd\ UV\text{-}M\ (HOs\ r)$
    and *decide*: $decide\ (rho\ (Suc\ r)\ p) \neq decide\ (rho\ r\ p)$
    and *decval*: $decide\ (rho\ (Suc\ r)\ p) = Some\ v$
  shows $x\ (rho\ (Suc\ r{+}k)\ q) = v$
$\langle proof \rangle$

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

lemma *decided-then-past-decision*:
  assumes *run*: $HORun\ UV\text{-}M\ rho\ HOs$
    and *dec*: $decide\ (rho\ n\ p) = Some\ v$
  shows $\exists m{<}n.\ decide\ (rho\ (Suc\ m)\ p) \neq decide\ (rho\ m\ p)$
      $\wedge\ decide\ (rho\ (Suc\ m)\ p) = Some\ v$
$\langle proof \rangle$

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

lemma *x-values-initial*:
  assumes *run*:$HORun\ UV\text{-}M\ rho\ HOs$
    and *com*:$\forall r.\ HOcommPerRd\ UV\text{-}M\ (HOs\ r)$
  shows $\exists q.\ x\ (rho\ r\ p) = x\ (rho\ 0\ q)$
$\langle proof \rangle$

theorem *uv-integrity*:
  assumes *run*: $HORun\ UV\text{-}M\ rho\ HOs$
    and *com*: $\forall r.\ HOcommPerRd\ UV\text{-}M\ (HOs\ r)$
    and *dec*: $decide\ (rho\ r\ p) = Some\ v$
  shows $\exists q.\ v = x\ (rho\ 0\ q)$
$\langle proof \rangle$

We now turn to Agreement.

lemma *two-decisions-agree*:
  assumes *run*: $HORun\ UV\text{-}M\ rho\ HOs$
    and *com*: $\forall r.\ HOcommPerRd\ UV\text{-}M\ (HOs\ r)$
    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 $v = w$
$\langle proof \rangle$

theorem *uv-agreement*:
  assumes *run*: $HORun\ UV\text{-}M\ rho\ HOs$
    and *com*: $\forall r.\ HOcommPerRd\ UV\text{-}M\ (HOs\ r)$
    and *p*: $decide\ (rho\ m\ p) = Some\ v$
    and *q*: $decide\ (rho\ n\ q) = Some\ w$

shows $v = w$
$\langle proof \rangle$

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

theorem *uv-irrevocability*:
assumes *run*: *HORun UV-M rho HOs*
and *com*: $\forall\, r.$ *HOcommPerRd UV-M* (*HOs r*)
and *p*: *decide* (*rho m p*) = *Some v*
shows *decide* (*rho* (*m+n*) *p*) = *Some v*
$\langle proof \rangle$

6.6 Proof of Termination

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

lemma *hoeq-xeq*:
assumes *run*: *HORun UV-M rho HOs*
and *com*: $\forall\, r.$ *HOcommPerRd UV-M* (*HOs r*)
and *hoeq*: *HOs r p* = *HOs r q*
shows *x* (*rho* (*Suc r*) *p*) = *x* (*rho* (*Suc r*) *q*)
$\langle proof \rangle$

We now prove that *UniformVoting* terminates.

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

6.7 *UniformVoting* Solves Consensus

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

theorem *uv-consensus*:
assumes *run*: *HORun UV-M rho HOs*
and *commR*: $\forall\, r.$ *HOcommPerRd UV-M* (*HOs r*)
and *commG*: *HOcommGlobal UV-M HOs*
shows *consensus* ($x \circ$ (*rho 0*)) *decide rho*
$\langle proof \rangle$

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* (λp. *x* (*state* (*rho 0*) *p*)) *decide* (*state* $\circ$ *rho*)
(**is** *consensus ?inits* - -)
⟨*proof*⟩

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

7 Verification of the *LastVoting* Consensus Algorithm

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

7.1 Model of the Algorithm

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

typedecl *Proc* — the set of processes
axiomatization where *Proc-finite*: *OFCLASS*(*Proc*, *finite-class*)
instance *Proc* :: *finite* ⟨*proof*⟩

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

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

definition *phase* **where** *phase* (*r*::*nat*) ≡ *r div 4*

definition *step* **where** *step* (*r*::*nat*) ≡ *r mod 4*

lemma *phase-zero* [*simp*]: *phase 0* = *0*
⟨*proof*⟩

lemma *step-zero* [*simp*]: *step 0* = *0*
⟨*proof*⟩

lemma *phase-step*: (*phase r* ∗ *4*) + *step r* = *r*
⟨*proof*⟩

The following record models the local state of a process.

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

Possible messages sent during the execution of the algorithm.

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

Characteristic predicates on messages.

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

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

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

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

fun val **where**
  $val\ (ValStamp\ v\ ts) = v$
| $val\ (Vote\ v) = v$

fun $stamp$ **where**
  $stamp\ (ValStamp\ v\ ts) = ts$

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

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

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

definition $valStampsRcvd$ **where**

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

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

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

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

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

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

In step 1, coordinators that have committed send their vote to all processes. Processes update their x and *timestamp* fields if they have received a vote from their coordinator.

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

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

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

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

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

— processes from which an acknowledgement was received
definition *acksRcvd* **where**
$acksRcvd\ (msgs :: Proc \rightharpoonup {'}val\ msg) \equiv$

$\{ \ q \ . \ msgs \ q \neq None \wedge isAck \ (the \ (msgs \ q)) \ \}$

definition *next2* **where**

$$
\begin{array}{l}
next2 \ r \ p \ st \ msgs \ crd \ st' \equiv \\
\quad if \ p = coord\Phi \ st \wedge card \ (acksRcvd \ msgs) > N \ div \ 2 \\
\quad then \ st' = st \ (\!| \ ready := True \ |\!) \\
\quad else \ st' = st
\end{array}
$$

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**

$$
\begin{array}{l}
send3 \ r \ p \ q \ st \equiv \\
\quad if \ p = coord\Phi \ st \wedge ready \ st \ then \ Vote \ (the \ (vote \ st)) \ else \ Null
\end{array}
$$

definition *next3* **where**

$$
\begin{array}{l}
next3 \ r \ p \ st \ msgs \ crd \ st' \equiv \\
\qquad (if \ msgs \ (coord\Phi \ st) \neq None \wedge isVote \ (the \ (msgs \ (coord\Phi \ st))) \\
\qquad \ then \ decide \ st' = Some \ (val \ (the \ (msgs \ (coord\Phi \ st)))) \\
\qquad \ else \ decide \ st' = decide \ st) \\
\quad \wedge \ (if \ p = coord\Phi \ st \\
\qquad then \ \neg(ready \ st') \wedge \neg(commt \ st') \\
\qquad else \ ready \ st' = ready \ st \wedge commt \ st' = commt \ st) \\
\quad \wedge \ x \ st' = x \ st \\
\quad \wedge \ vote \ st' = vote \ st \\
\quad \wedge \ timestamp \ st' = timestamp \ st \\
\quad \wedge \ coord\Phi \ st' = crd
\end{array}
$$

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

definition *LV-sendMsg* $:: nat \Rightarrow Proc \Rightarrow Proc \Rightarrow {}'val \ pstate \Rightarrow {}'val \ msg$ **where**

$$
\begin{array}{l}
LV\text{-}sendMsg \ (r::nat) \equiv \\
\quad if \ step \ r = 0 \ then \ send0 \ r \\
\quad else \ if \ step \ r = 1 \ then \ send1 \ r \\
\quad else \ if \ step \ r = 2 \ then \ send2 \ r \\
\quad else \ send3 \ r
\end{array}
$$

definition

$$
\begin{array}{l}
LV\text{-}nextState :: nat \Rightarrow Proc \Rightarrow {}'val \ pstate \Rightarrow (Proc \rightharpoonup {}'val \ msg) \\
\qquad\qquad\qquad \Rightarrow Proc \Rightarrow {}'val \ pstate \Rightarrow bool
\end{array}
$$

where

$$
\begin{array}{l}
LV\text{-}nextState \ r \equiv \\
\quad if \ step \ r = 0 \ then \ next0 \ r \\
\quad else \ if \ step \ r = 1 \ then \ next1 \ r \\
\quad else \ if \ step \ r = 2 \ then \ next2 \ r \\
\quad else \ next3 \ r
\end{array}
$$

7.2 Communication Predicate for *LastVoting*

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

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

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

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

definition
LV-commPerRd **where**
$LV\text{-}commPerRd\ r\ (HO{::}Proc\ HO)\ (coord{::}Proc\ coord) \equiv True$

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

7.3 The *LastVoting* Heard-Of Machine

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

definition *LV-CHOMachine* **where**
$LV\text{-}CHOMachine \equiv$
$\quad (\!|\ CinitState = LV\text{-}initState,$
$\quad\ sendMsg = LV\text{-}sendMsg,$
$\quad\ CnextState = LV\text{-}nextState,$
$\quad\ CHOcommPerRd = LV\text{-}commPerRd,$
$\quad\ CHOcommGlobal = LV\text{-}commGlobal\ |\!)$

abbreviation
$LV\text{-}M \equiv (LV\text{-}CHOMachine{::}(Proc,\ 'val\ pstate,\ 'val\ msg)\ CHOMachine)$

end

theory *LastVotingProof*
imports *LastVotingDefs ../Majorities ../Reduction*
begin

7.4 Preliminary Lemmas

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

lemma *timeStampsRcvdFinite*:
  finite {*ts* . ∃ *q v*. (*msgs*::*Proc* ⇀ *'val msg*) *q* = *Some* (*ValStamp v ts*)}
  (**is** *finite ?ts*)
⟨*proof*⟩

lemma *highestStampRcvd-exists*:
  assumes *nempty*: *valStampsRcvd msgs* ≠ {}
  obtains *p v* **where** *msgs p* = *Some* (*ValStamp v* (*highestStampRcvd msgs*))
⟨*proof*⟩

lemma *highestStampRcvd-max*:
  assumes *msgs p* = *Some* (*ValStamp v ts*)
  shows *ts* ≤ *highestStampRcvd msgs*
  ⟨*proof*⟩

lemma *phase-Suc*:
  phase (*Suc r*) = (*if step r* = *3 then Suc* (*phase r*)
                  else phase r)
  ⟨*proof*⟩

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*: ⋀*r*.
        ⟦ *step r* = *0*; *P r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *1*;
          ∀ *p*. *next0 r p* (*rho r p*)
                (*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
                (*coords* (*Suc r*) *p*)
                (*rho* (*Suc r*) *p*) ⟧
        ⟹ *P* (*Suc r*)
  and *step1*: ⋀*r*.
        ⟦ *step r* = *1*; *P r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *2*;
          ∀ *p*. *next1 r p* (*rho r p*)
                (*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
                (*coords* (*Suc r*) *p*)
                (*rho* (*Suc r*) *p*) ⟧
        ⟹ *P* (*Suc r*)

and *step2*: $\bigwedge r.$
$[\![$ *step r = 2*; *P r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *3*;
∀ *p*. *next2 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*)
(*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P* (*Suc r*)
and *step3*: $\bigwedge r.$
$[\![$ *step r = 3*; *P r*; *phase* (*Suc r*) = *Suc* (*phase r*); *step* (*Suc r*) = *0*;
∀ *p*. *next3 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*)
(*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P* (*Suc r*)
shows *P n*
⟨*proof*⟩

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

lemma *LV-Suc*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *step0*: $[\![$ *step r = 0*; *step* (*Suc r*) = *1*; *phase* (*Suc r*) = *phase r*;
∀ *p*. *next0 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P r*
and *step1*: $[\![$ *step r = 1*; *step* (*Suc r*) = *2*; *phase* (*Suc r*) = *phase r*;
∀ *p*. *next1 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P r*
and *step2*: $[\![$ *step r = 2*; *step* (*Suc r*) = *3*; *phase* (*Suc r*) = *phase r*;
∀ *p*. *next2 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P r*
and *step3*: $[\![$ *step r = 3*; *step* (*Suc r*) = *0*; *phase* (*Suc r*) = *Suc* (*phase r*);
∀ *p*. *next3 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P r*
shows *P r*
⟨*proof*⟩

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

lemma *LV-induct′*:

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

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

⟨*proof*⟩

7.5 Boundedness and Monotonicity of Timestamps

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

lemma *LV-timestamp-bounded*:
assumes *run*: *CHORun LV-M rho HOs coords*
shows *timestamp* (*rho n p*) $\leq$ (*if step n* $<$ *2 then phase n else Suc* (*phase n*))
(**is** *?P p n*)
⟨*proof*⟩

Moreover, timestamps can only grow over time.

lemma *LV-timestamp-increasing*:
assumes *run*: *CHORun LV-M rho HOs coords*
shows *timestamp* (*rho n p*) $\leq$ *timestamp* (*rho* (*Suc n*) *p*)
(**is** *?P p n* **is** *?ts* $\leq$ -)
⟨*proof*⟩

lemma *LV-timestamp-monotonic*:
assumes *run*: *CHORun LV-M rho HOs coords* **and** *le*: *m* $\leq$ *n*
shows *timestamp* (*rho m p*) $\leq$ *timestamp* (*rho n p*)
(**is** *?ts m* $\leq$ -)
⟨*proof*⟩

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

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

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

lemma *procsBeyondTS-monotonic*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *p*: *p* $\in$ *procsBeyondTS ts* (*rho m*) **and** *le*: *m* $\leq$ *n*
shows *p* $\in$ *procsBeyondTS ts* (*rho n*)
⟨*proof*⟩

7.6 Obvious Facts About the Algorithm

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

Coordinators change only at step 3.

lemma *notStep3EqualCoord*:
assumes *run*: *CHORun LV-M rho HOs coords* **and** *stp*:*step r* $\neq$ *3*

shows $coord\Phi$ $(rho\ (Suc\ r)\ p) = coord\Phi\ (rho\ r\ p)$ (**is** $?P\ p\ r$)
⟨*proof*⟩

lemma *coordinators*:
assumes *run*: *CHORun LV-M rho HOs coords*
shows $coord\Phi\ (rho\ r\ p) = coords\ (4*(phase\ r))\ p$
⟨*proof*⟩

Votes only change at step 0.

lemma *notStep0EqualVote* [*rule-format*]:
assumes *run*: *CHORun LV-M rho HOs coords*
shows $step\ r \neq 0 \longrightarrow vote\ (rho\ (Suc\ r)\ p) = vote\ (rho\ r\ p)$ (**is** $?P\ p\ r$)
⟨*proof*⟩

Commit status only changes at steps 0 and 3.

lemma *notStep03EqualCommit* [*rule-format*]:
assumes *run*: *CHORun LV-M rho HOs coords*
shows $step\ r \neq 0 \wedge step\ r \neq 3 \longrightarrow commt\ (rho\ (Suc\ r)\ p) = commt\ (rho\ r\ p)$
(**is** $?P\ p\ r$)
⟨*proof*⟩

Timestamps only change at step 1.

lemma *notStep1EqualTimestamp* [*rule-format*]:
assumes *run*: *CHORun LV-M rho HOs coords*
shows $step\ r \neq 1 \longrightarrow timestamp\ (rho\ (Suc\ r)\ p) = timestamp\ (rho\ r\ p)$
(**is** $?P\ p\ r$)
⟨*proof*⟩

The x field only changes at step 1.

lemma *notStep1EqualX* [*rule-format*]:
assumes *run*: *CHORun LV-M rho HOs coords*
shows $step\ r \neq 1 \longrightarrow x\ (rho\ (Suc\ r)\ p) = x\ (rho\ r\ p)$ (**is** $?P\ p\ r$)
⟨*proof*⟩

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

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

lemma *commitE*:
assumes *run*: *CHORun LV-M rho HOs coords* **and** *cmt*: $commt\ (rho\ r\ p)$
and *conds*: $[\![\ 1 \leq step\ r;\ coord\Phi\ (rho\ r\ p) = p;\ vote\ (rho\ r\ p) \neq None;$
$card\ \{q\ .\ coord\Phi\ (rho\ r\ q) = p\} > N\ div\ 2$
$]\!] \Longrightarrow A$

shows A
$\langle proof \rangle$

A process has a current timestamp only if:

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

lemma *currentTimestampE*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *ts*: *timestamp* (*rho r p*) = *Suc* (*phase r*)
and *conds*: $[\![$ *2* $\leq$ *step r*;
commt (*rho r* (*coord*Φ (*rho r p*)));
x (*rho r p*) = *the* (*vote* (*rho r* (*coord*Φ (*rho r p*))))
$]\!] \Longrightarrow A$
shows A
$\langle proof \rangle$

If a process p has its *ready* bit set then:

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

lemma *readyE*:
assumes *run*: *CHORun LV-M rho HOs coords* **and** *rdy*: *ready* (*rho r p*)
and *conds*: $[\![$ *step r* = *3*; *coord*Φ (*rho r p*) = *p*;
card { *q* . *coord*Φ (*rho r q*) = *p*
$\wedge$ *timestamp* (*rho r q*) = *Suc* (*phase r*) } > *N div 2*
$]\!] \Longrightarrow P$
shows P
$\langle proof \rangle$

A process decides only if the following conditions hold:

- it is at step 3,
- its coordinator votes for the value the process decides on,
- the coordinator has its *ready* and *commt* bits set.

lemma *decisionE*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *dec*: *decide* (*rho* (*Suc r*) *p*) $\neq$ *decide* (*rho r p*)

and *conds*: $\llbracket$
$\quad$ $step\ r = 3$;
$\quad$ $decide\ (rho\ (Suc\ r)\ p) = Some\ (the\ (vote\ (rho\ r\ (coord\Phi\ (rho\ r\ p)))))$;
$\quad$ $ready\ (rho\ r\ (coord\Phi\ (rho\ r\ p)))$; $commt\ (rho\ r\ (coord\Phi\ (rho\ r\ p)))$
$\rrbracket \Longrightarrow P$
shows P
$\langle proof \rangle$

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*: $\llbracket\ range\ (x \circ (rho\ n)) \subseteq range\ (x \circ (rho\ 0))$;
$range\ (vote \circ (rho\ n)) \subseteq \{None\} \cup Some\ `\ range\ (x \circ (rho\ 0))$;
$range\ (decide \circ (rho\ n)) \subseteq \{None\} \cup Some\ `\ range\ (x \circ (rho\ 0))$
$\rrbracket \Longrightarrow A$
shows A
$\langle proof \rangle$

Integrity now follows immediately.

theorem *lv-integrity*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *dec*: $decide\ (rho\ n\ p) = Some\ v$
shows $\exists\, q.\ v = x\ (rho\ 0\ q)$
$\langle proof \rangle$

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*: $decide\ (rho\ (Suc\ r)\ p) \neq decide\ (rho\ r\ p)$
shows $card\ (procsBeyondTS\ (Suc\ (phase\ r))\ (rho\ r)) > N\ div\ 2$
$\langle proof \rangle$

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 $p = p'$
$\langle proof \rangle$

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

lemma *readyProcsEqual*:
  assumes *run*: *CHORun LV-M rho HOs coords*
  and *rdy*: *ready* (*rho r p*) **and** *rdy'*: *ready* (*rho r p'*)
  shows $p = p'$
⟨*proof*⟩

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

lemma *commitThenVoteRecent*:
  assumes *run*: *CHORun LV-M rho HOs coords*
  and *maj*: *card* (*procsBeyondTS ts* (*rho r*)) > *N div 2*
  and *cmt*: *commt* (*rho r p*)
  shows ∃ *q* ∈ *procsBeyondTS ts* (*rho r*). *vote* (*rho r p*) = *Some* (*x* (*rho r q*))
    (**is** *?Q r*)
⟨*proof*⟩

The following lemma gives the crucial argument for agreement: after some process p has decided, all processes whose timestamp is beyond the timestamp at the point of decision contain the decision value in their x field.

lemma *XOfTimestampBeyondDecision*:
  assumes *run*: *CHORun LV-M rho HOs coords*
    and *dec*: *decide* (*rho* (*Suc r*) *p*) ≠ *decide* (*rho r p*)
  shows ∀ *q* ∈ *procsBeyondTS* (*Suc* (*phase r*)) (*rho* (*r*+*k*)).
      x (*rho* (*r*+*k*) *q*) = *the* (*decide* (*rho* (*Suc r*) *p*))
  (**is** ∀ *q* ∈ *?bynd k*. - = *?v* **is** *?P p k*)
⟨*proof*⟩

We are now in position to prove Agreement: if some process decides at step r and another (or possibly the same) process decides at step $r+k$ then they decide the same value.

lemma *laterProcessDecidesSameValue*:
  assumes *run*: *CHORun LV-M rho HOs coords*
  and *p*: *decide* (*rho* (*Suc r*) *p*) ≠ *decide* (*rho r p*)
  and *q*: *decide* (*rho* (*Suc* (*r*+*k*)) *q*) ≠ *decide* (*rho* (*r*+*k*) *q*)
  shows *decide* (*rho* (*Suc* (*r*+*k*)) *q*) = *decide* (*rho* (*Suc r*) *p*)
⟨*proof*⟩

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

lemma *decisionNonNullThenDecided*:
  assumes *run*: *CHORun LV-M rho HOs coords*
    and *dec*: *decide* (*rho n p*) = *Some v*
  shows ∃ *m*<*n*. *decide* (*rho* (*Suc m*) *p*) ≠ *decide* (*rho m p*)
      ∧ *decide* (*rho* (*Suc m*) *p*) = *Some v*
⟨*proof*⟩

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

theorem *lv-irrevocability*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *p*: *decide* (*rho m p*) = *Some v*
shows *decide* (*rho* (*m*+*k*) *p*) = *Some v*
⟨*proof*⟩

theorem *lv-agreement*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *p*: *decide* (*rho m p*) = *Some v*
and *q*: *decide* (*rho n q*) = *Some w*
shows *v* = *w*
⟨*proof*⟩

7.9 Proof of Termination

The proof of termination relies on the communication predicate, which stipulates the existence of some phase during which there is a single coordinator that (a) receives a majority of messages and (b) is heard by everybody. Therefore, all processes successfully execute the protocol, deciding at step 3 of that phase.

theorem *lv-termination*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *commG*:*CHOcommGlobal LV-M HOs coords*
shows $\exists$ *r*. $\forall$ *p*. *decide* (*rho r p*) $\neq$ *None*
⟨*proof*⟩

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*⟩

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

theorem *lv-consensus-fg*:
assumes *run*: *fg-run LV-M rho HOs HOs coords*
and *commG*: *CHOcommGlobal LV-M HOs coords*
shows *consensus* (λ*p*. *x* (*state* (*rho 0*) *p*)) *decide* (*state* ∘ *rho*)
(**is** *consensus ?inits* - -)
⟨*proof*⟩

end
theory *UteDefs*

imports *../HOModel*
begin

8 Verification of the $\mathcal{U}_{T,E,\alpha}$ Consensus Algorithm

Algorithm $\mathcal{U}_{T,E,\alpha}$ is presented in [3]. It is an uncoordinated algorithm that tolerates value (a.k.a. Byzantine) faults, and can be understood as a variant of *UniformVoting*. The parameters T, E, and α appear as thresholds of the algorithm and in the communication predicates. Their values can be chosen within certain bounds in order to adapt the algorithm to the characteristics of different systems.

We formalize in Isabelle the correctness proof of the algorithm that appears in [3], using the framework of theory *HOModel*.

8.1 Model of the Algorithm

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

typedecl *Proc* — the set of processes
axiomatization where *Proc-finite*: *OFCLASS*(*Proc*, *finite-class*)
instance *Proc* :: *finite* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *step-zero* [*simp*]: *step 0* = *0*
⟨*proof*⟩

lemma *phase-step*: (*phase r* ∗ *nSteps*) + *step r* = *r*
  ⟨*proof*⟩

The following record models the local state of a process.

record *'val pstate* =
  x :: *'val*   — current value held by process

vote :: *'val option* — value the process voted for, if any
decide :: *'val option* — value the process has decided on, if any

Possible messages sent during the execution of the algorithm.

datatype *'val msg* =
  Val 'val
| *Vote 'val option*

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

definition *Ute-initState* **where**
  Ute-initState p st ≡
  (*vote st* = *None*) ∧ (*decide st* = *None*)

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

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

Simple consequences of the above parameter constraints.

lemma *alpha-lt-N*: $\alpha < N$
⟨*proof*⟩

lemma *alpha-lt-T*: $\alpha < T$
⟨*proof*⟩

lemma *alpha-lt-E*: $\alpha < E$
⟨*proof*⟩

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

In step 0, each process sends its current x. If it receives the value v more than T times, it votes for v, otherwise it doesn't vote.

definition
  send0 :: *nat* ⇒ *Proc* ⇒ *Proc* ⇒ *'val pstate* ⇒ *'val msg*
where
  send0 r p q st ≡ *Val* (*x st*)

definition
  next0 :: *nat* ⇒ *Proc* ⇒ *'val pstate* ⇒ (*Proc* ⇒ *'val msg option*)
    ⇒ *'val pstate* ⇒ *bool*
where

$$
\begin{aligned}
&\mathit{next0}\ r\ p\ st\ msgs\ st' \equiv\\
&\quad (\exists\, v.\ card\ \{q.\ msgs\ q = Some\ (Val\ v)\} > T \wedge st' = st\ (\!|\ vote := Some\ v\ |\!))\\
&\ \vee \neg(\exists\, v.\ card\ \{q.\ msgs\ q = Some\ (Val\ v)\} > T) \wedge st' = st\ (\!|\ vote := None\ |\!)
\end{aligned}
$$

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

If it receives more than α votes for a given value v, it sets its x field to v, else it sets x to a default value.

If the process receives more than E votes for v, it decides v, otherwise it leaves its decision unchanged.

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

definition
$\mathit{next1} :: nat \Rightarrow Proc \Rightarrow {}'val\ pstate \Rightarrow (Proc \Rightarrow {}'val\ msg\ option) \Rightarrow {}'val\ pstate \Rightarrow bool$
where

$$
\begin{aligned}
&\mathit{next1}\ r\ p\ st\ msgs\ st' \equiv\\
&\quad (\ (\exists\, v.\ card\ \{q.\ msgs\ q = Some\ (Vote\ (Some\ v))\} > \alpha \wedge x\ st' = v)\\
&\quad\ \vee \neg(\exists\, v.\ card\ \{q.\ msgs\ q = Some\ (Vote\ (Some\ v))\} > \alpha)\\
&\qquad \wedge x\ st' = undefined\ \ )\\
&\ \wedge (\ (\exists\, v.\ card\ \{q.\ msgs\ q = Some\ (Vote\ (Some\ v))\} > E \wedge decide\ st' = Some\ v)\\
&\quad\ \vee \neg(\exists\, v.\ card\ \{q.\ msgs\ q = Some\ (Vote\ (Some\ v))\} > E)\\
&\qquad \wedge decide\ st' = decide\ st\ )\\
&\ \wedge vote\ st' = None
\end{aligned}
$$

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

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

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

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

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

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

- no process may receive more than α corrupted messages, and

- every process should receive more than $max(T,\ N + 2{*}\alpha - E - 1)$ correct messages.

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

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

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

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

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

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

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

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

definition *Ute-SHOMachine* **where**
  Ute-SHOMachine = (|
    CinitState = (λ *p st crd. Ute-initState p st*),
    sendMsg = *Ute-sendMsg*,
    CnextState = (λ *r p st msgs crd st'. Ute-nextState r p st msgs st'*),
    SHOcommPerRd = *Ute-commPerRd*,
    SHOcommGlobal = *Ute-commGlobal*
  |)

abbreviation

Ute-M $\equiv$ (*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.

lemma *vote-step*:
  assumes *nxt*: *nextState Ute-M r p* (*rho r p*) μ (*rho* (*Suc r*) *p*)
  and *vote* (*rho* (*Suc r*) *p*) $\neq$ *None*
  shows *step r* = *0*
⟨*proof*⟩

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

lemma *decide-step*:
  assumes *run*: *SHORun Ute-M rho HOs SHOs*
  and *d1*: *decide* (*rho r p*) $\neq$ *Some v*
  and *d2*: *decide* (*rho* (*Suc r*) *p*) = *Some v*
  shows *step r* $\neq$ *0*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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

lemma *common-vote*:

assumes *usafe*: *SHOcommPerRd Ute-M HO SHO*
and *nxtp*: *nextState Ute-M r p (rho r p) μp (rho (Suc r) p)*
and *mup*: *μp ∈ SHOmsgVectors Ute-M r p (rho r) (HO p) (SHO p)*
and *nxtq*: *nextState Ute-M r q (rho r q) μq (rho (Suc r) q)*
and *muq*: *μq ∈ SHOmsgVectors Ute-M r q (rho r) (HO q) (SHO q)*
and *vp*: *vote (rho (Suc r) p) = Some vp*
and *vq*: *vote (rho (Suc r) q) = Some vq*
shows *vp = vq*
⟨*proof*⟩

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*⟩

8.5 Proof of Agreement and Validity

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

lemma *common-x-argument-1*:
assumes *usafe*:*SHOcommPerRd Ute-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 − α
(**is** *card (?msgs p v) > -*)
shows *card (?msgs pp v ∩ (SHOs (Suc r) pp ∩ HOs (Suc r) pp)) > α*
⟨*proof*⟩

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

lemma *common-x-argument-2*:
assumes *run*: *SHORun Ute-M rho HOs SHOs*
and *usafe*: *∀ r. SHOcommPerRd Ute-M (HOs r) (SHOs r)*
and *nxtpp*: *nextState Ute-M (Suc r) pp (rho (Suc r) pp)*
μpp (rho (Suc (Suc r)) pp)
and *mupp*: *μpp ∈ SHOmsgVectors Ute-M (Suc r) pp (rho (Suc r))*
(HOs (Suc r) pp) (SHOs (Suc r) pp)
and *threshold*: *card {q. sendMsg Ute-M (Suc r) q p (rho (Suc r) q)*
= Vote (Some v)} > E − α
(**is** *card (?sent p v) > -*)
shows *x (rho (Suc (Suc r)) pp) = v*
⟨*proof*⟩

Inductive argument for the agreement and validity theorems.

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

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*)
  $\wedge$ *decide* (*rho* (*Suc m*) *p*) = *Some v*
⟨*proof*⟩

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*⟩

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

theorem *ute-agreement*:
 assumes *run*: *SHORun Ute-M rho HOs SHOs*
 and *comm*: $\forall r.$ *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
 and *p*: *decide* (*rho m p*) = *Some v*
 and *q*: *decide* (*rho n q*) = *Some w*
 shows *v* = *w*
⟨*proof*⟩

Main lemma for the proof of the Validity property.

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

⟨*proof*⟩

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

theorem *ute-validity*:
  assumes *run*: *SHORun Ute-M rho HOs SHOs*
  and *comm*: $\forall$ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
  and *init*: $\forall$ *p*. *x* (*rho 0 p*) = *v*
  and *dw*: *decide* (*rho r p*) = *Some w*
  shows *v* = *w*
⟨*proof*⟩

8.6 Proof of Termination

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

lemma *set-x-from-vote*:
  assumes *run*: *SHORun Ute-M rho HOs SHOs*
  and *comm*: *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
  and *stp*: *step* (*Suc r*) = *Suc 0*
  and π: $\forall$ *p*. *HOs* (*Suc r*) *p* = *SHOs* (*Suc r*) *p*
  and *nxt*: *nextState Ute-M* (*Suc r*) *p* (*rho* (*Suc r*) *p*) μ (*rho* (*Suc* (*Suc r*)) *p*)
  and *mu*: $\mu \in$ *SHOmsgVectors Ute-M* (*Suc r*) *p* (*rho* (*Suc r*))
      (*HOs* (*Suc r*) *p*) (*SHOs* (*Suc r*) *p*)
  and *vp*: $\alpha <$ *card* {*qq*. μ *qq* = *Some* (*Vote* (*Some v*))}
  shows *x* ((*rho* (*Suc* (*Suc r*))) *p*) = *v*
⟨*proof*⟩

Assume that HO and SHO sets are uniform at the second step of some phase. Then at the subsequent round there exists some value v such that any received message which is not corrupted holds v.

lemma *termination-argument-1*:
  assumes *run*: *SHORun Ute-M rho HOs SHOs*
  and *comm*: *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
  and *stp*: *step* (*Suc r*) = *Suc 0*
  and π: $\forall$ *p*. $\pi 0$ = *HOs* (*Suc r*) *p* $\wedge$ $\pi 0$ = *SHOs* (*Suc r*) *p*
  obtains *v* **where**
    $\bigwedge$*p* $\mu p'$ *q*.
      ⟦ *q* $\in$ *SHOs* (*Suc* (*Suc r*)) *p* $\cap$ *HOs* (*Suc* (*Suc r*)) *p*;
        $\mu p' \in$ *SHOmsgVectors Ute-M* (*Suc* (*Suc r*)) *p* (*rho* (*Suc* (*Suc r*)))
          (*HOs* (*Suc* (*Suc r*)) *p*) (*SHOs* (*Suc* (*Suc r*)) *p*)
      ⟧ $\Longrightarrow$ $\mu p'$ *q* = (*Some* (*Val v*))
⟨*proof*⟩

If a process p votes v at some round r, then all messages received by p in r that are not corrupted hold v.

lemma *termination-argument-2*:

assumes *mup*: $\mu p \in$ *SHOmsgVectors Ute-M* (*Suc r*) *p* (*rho* (*Suc r*))
(*HOs* (*Suc r*) *p*) (*SHOs* (*Suc r*) *p*)
and *nxtq*: *nextState Ute-M r q* (*rho r q*) μq (*rho* (*Suc r*) *q*)
and *vq*: *vote* (*rho* (*Suc r*) *q*) = *Some v*
and *qsho*: $q \in$ *SHOs* (*Suc r*) $p \cap$ *HOs* (*Suc r*) *p*
shows μp *q* = *Some* (*Vote* (*Some v*))
⟨*proof*⟩

We now prove the Termination property.

theorem *ute-termination*:
assumes *run*: *SHORun Ute-M rho HOs SHOs*
and *commR*: $\forall$ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ute-M HOs SHOs*
shows $\exists$ *r v*. *decide* (*rho r p*) = *Some v*
⟨*proof*⟩

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

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

theorem *ute-weak-consensus*:
assumes *run*: *SHORun Ute-M rho HOs SHOs*
and *commR*: $\forall$ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ute-M HOs SHOs*
shows *weak-consensus* ($x \circ$ (*rho 0*)) *decide rho*
⟨*proof*⟩

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

theorem *ute-weak-consensus-fg*:
assumes *run*: *fg-run Ute-M rho HOs SHOs* (λ*r q*. *undefined*)
and *commR*: $\forall$ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ute-M HOs SHOs*
shows *weak-consensus* (λ*p*. *x* (*state* (*rho 0*) *p*)) *decide* (*state* $\circ$ *rho*)
(**is** *weak-consensus ?inits* - -)
⟨*proof*⟩

end — context *ute-parameters*

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

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

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

We formalize in Isabelle the correctness proof of the algorithm that appears in [3], using the framework of theory *HOModel*.

9.1 Model of the Algorithm

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

typedecl *Proc* — the set of processes
axiomatization where *Proc-finite*: *OFCLASS*(*Proc*, *finite-class*)
instance *Proc* :: *finite* ⟨*proof*⟩

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

The following record models the local state of a process.

record *'val pstate* =
  x :: *'val* — current value held by process
  decide :: *'val option* — value the process has decided on, if any

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

definition *Ate-initState* **where**
  Ate-initState p st $\equiv$ (*decide st* = *None*)

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

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

begin

The following are consequences of the assumptions on the parameters.

lemma *majE*: $2 * (E - \alpha) \geq N$
⟨*proof*⟩

lemma *Egta*: $E > \alpha$

⟨*proof*⟩

lemma *Tge2a*: $T \geq 2 * \alpha$
⟨*proof*⟩

At every round, each process sends its current x. If it received more than T messages, it selects the smallest value and store it in x. As in algorithm *OneThirdRule*, we therefore require values to be linearly ordered.

If more than E messages holding the same value are received, the process decides that value.

definition *mostOftenRcvd* **where**
$mostOftenRcvd\ (msgs{::}Proc \Rightarrow {'}val\ option) \equiv$
$\{v.\ \forall w.\ card\ \{qq.\ msgs\ qq = Some\ w\} \leq card\ \{qq.\ msgs\ qq = Some\ v\}\}$

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

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

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

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

definition *Ate-commPerRd* **where**
$Ate\text{-}commPerRd\ HOrs\ SHOrs \equiv$
$\forall p.\ card\ (HOrs\ p - SHOrs\ p) \leq \alpha$

The global communication predicate stipulates the three following conditions:

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

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

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

9.3 The $\mathcal{A}_{T,E,\alpha}$ Heard-Of Machine

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

definition *Ate-SHOMachine* **where**
Ate-SHOMachine = (|
CinitState = (λ *p st crd*. *Ate-initState p* (*st*::('*val*::*linorder*) *pstate*)),
sendMsg = *Ate-sendMsg*,
CnextState = (λ *r p st msgs crd st*'. *Ate-nextState r p st msgs st*'),
SHOcommPerRd = (*Ate-commPerRd*:: *Proc HO* ⇒ *Proc HO* ⇒ *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 $card\ \{qq.\ sendMsg\ Ate\text{-}M\ r\ qq\ p\ (rho\ r\ qq) = v\} > E - \alpha$
⟨*proof*⟩

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*: $SHOcommPerRd\ Ate\text{-}M\ (HOs\ r)\ (SHOs\ r)$
and *nxt*: $nextState\ Ate\text{-}M\ r\ q\ (rho\ r\ q)\ \mu q\ ((rho\ (Suc\ r))\ q)$
and *muq*: $\mu q \in SHOmsgVectors\ Ate\text{-}M\ r\ q\ (rho\ r)\ (HOs\ r\ q)\ (SHOs\ r\ q)$
and *vsent*: $card\ \{qq.\ sendMsg\ Ate\text{-}M\ r\ qq\ q\ (rho\ r\ qq) = v\} > E - \alpha$
(**is** $card\ ?vsent > \text{-}$)
shows $card\ (\{qq.\ \mu q\ qq \neq Some\ v\} \cap HOs\ r\ q) \leq N + 2{*}\alpha - E$
⟨*proof*⟩

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*: $SHOcommPerRd\ Ate\text{-}M\ (HOs\ r)\ (SHOs\ r)$
and *nxt*: $nextState\ Ate\text{-}M\ r\ q\ (rho\ r\ q)\ \mu q\ ((rho\ (Suc\ r))\ q)$
and *muq*: $\mu q \in SHOmsgVectors\ Ate\text{-}M\ r\ q\ (rho\ r)\ (HOs\ r\ q)\ (SHOs\ r\ q)$
and *threshold-T*: $card\ \{qq.\ \mu q\ qq \neq None\} > T$
and *threshold-E*: $card\ \{qq.\ sendMsg\ Ate\text{-}M\ r\ qq\ q\ (rho\ r\ qq) = v\} > E - \alpha$
shows $mostOftenRcvd\ \mu q = \{v\}$
⟨*proof*⟩

If at some round more than $E - \alpha$ processes have their x variable set to v, then this is also true at next round.

lemma *common-x-induct*:
assumes *run*: $SHORun\ Ate\text{-}M\ rho\ HOs\ SHOs$
and *comm*: $SHOcommPerRd\ Ate\text{-}M\ (HOs\ (r{+}k))\ (SHOs\ (r{+}k))$
and *ih*: $card\ \{qq.\ x\ (rho\ (r + k)\ qq) = v\} > E - \alpha$
shows $card\ \{qq.\ x\ (rho\ (r + Suc\ k)\ qq) = v\} > E - \alpha$
⟨*proof*⟩

Whenever some process newly decides value v, then any process that updates its x variable will set it to v.

lemma *common-x*:
assumes *run*: $SHORun\ Ate\text{-}M\ rho\ HOs\ SHOs$
and *comm*: $\forall r.\ SHOcommPerRd\ (Ate\text{-}M{::}(Proc, 'val{::}linorder\ pstate, 'val)\ SHOMachine)$
$(HOs\ r)\ (SHOs\ r)$
and *d1*: $decide\ (rho\ r\ p) \neq Some\ v$
and *d2*: $decide\ (rho\ (Suc\ r)\ p) = Some\ v$
and *qupdatex*: $x\ (rho\ (r + Suc\ k)\ q) \neq x\ (rho\ (r + k)\ q)$

shows $x\ (rho\ (r + Suc\ k)\ q) = v$
⟨*proof*⟩

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

lemma *decisionNonNullThenDecided*:
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *dec*: *decide* $(rho\ n\ p) = Some\ v$
obtains m **where** $m < n$
and *decide* $(rho\ m\ p) \neq Some\ v$
and *decide* $(rho\ (Suc\ m)\ p) = Some\ v$
⟨*proof*⟩

9.5 Proof of Validity

Validity asserts that if all processes were initialized with the same value, then no other value may ever be decided.

theorem *ate-validity*:
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *comm*: $\forall r.$ *SHOcommPerRd Ate-M* $(HOs\ r)\ (SHOs\ r)$
and *initv*: $\forall q.\ x\ (rho\ 0\ q) = v$
and *dp*: *decide* $(rho\ r\ p) = Some\ w$
shows $w = v$
⟨*proof*⟩

9.6 Proof of Agreement

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

lemma *common-decision*:
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *comm*: *SHOcommPerRd Ate-M* $(HOs\ r)\ (SHOs\ r)$
and *nvp*: *decide* $(rho\ r\ p) \neq Some\ v$
and *vp*: *decide* $(rho\ (Suc\ r)\ p) = Some\ v$
and *nwq*: *decide* $(rho\ r\ q) \neq Some\ w$
and *wq*: *decide* $(rho\ (Suc\ r)\ q) = Some\ w$
shows $w = v$
⟨*proof*⟩

If process p decides at step r and process q decides at some later step $r+k$ then p and q decide the same value.

lemma *laterProcessDecidesSameValue* :
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *comm*: $\forall r.$ *SHOcommPerRd Ate-M* $(HOs\ r)\ (SHOs\ r)$
and *nd1*: *decide* $(rho\ r\ p) \neq Some\ v$
and *d1*: *decide* $(rho\ (Suc\ r)\ p) = Some\ v$
and *nd2*: *decide* $(rho\ (r+k)\ q) \neq Some\ w$
and *d2*: *decide* $(rho\ (Suc\ (r+k))\ q) = Some\ w$
shows $w = v$

⟨*proof*⟩

The Agreement property is now an immediate consequence.

theorem *ate-agreement*:
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *comm*: $\forall r$. *SHOcommPerRd Ate-M* (*HOs r*) (*SHOs r*)
and *p*: *decide* (*rho m p*) = *Some v*
and *q*: *decide* (*rho n q*) = *Some w*
shows *w* = *v*
⟨*proof*⟩

9.7 Proof of Termination

We now prove that every process must eventually decide, given the global and round-by-round communication predicates.

theorem *ate-termination*:
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *commR*: $\forall r$. (*SHOcommPerRd*::((*Proc*, *'val*::*linorder pstate*, *'val*) *SHOMachine*)
$\Rightarrow$ (*Proc HO*) $\Rightarrow$ (*Proc HO*) $\Rightarrow$ *bool*)
Ate-M (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ate-M HOs SHOs*
shows $\exists r\ v$. *decide* (*rho r p*) = *Some v*
⟨*proof*⟩

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

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

theorem *ate-weak-consensus*:
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *commR*: $\forall r$. *SHOcommPerRd Ate-M* (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ate-M HOs SHOs*
shows *weak-consensus* ($x \circ$ (*rho 0*)) *decide rho*
⟨*proof*⟩

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

theorem *ate-weak-consensus-fg*:
assumes *run*: *fg-run Ate-M rho HOs SHOs* ($\lambda r\ q$. *undefined*)
and *commR*: $\forall r$. *SHOcommPerRd Ate-M* (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ate-M HOs SHOs*
shows *weak-consensus* (λp. *x* (*state* (*rho 0*) *p*)) *decide* (*state* $\circ$ *rho*)
(**is** *weak-consensus ?inits* - -)
⟨*proof*⟩

end — context *ate-parameters*

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

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

Lynch [12] presents $\textit{EIGByz}_f$, a version of the *exponential information gathering* algorithm tolerating Byzantine faults, that works in f rounds, and that was originally introduced in [1].

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

typedecl *Proc* — the set of processes
axiomatization where *Proc-finite*: *OFCLASS*(*Proc*, *finite-class*)
instance *Proc* :: *finite* ⟨*proof*⟩

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

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

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

10.1 Tree Data Structure

The algorithm relies on propagating information about the initially proposed values among all the processes. This information is stored in trees whose branches are labeled by lists of (distinct) processes. For example, the interpretation of an entry $[p,q] \mapsto Some\ v$ is that the current process heard from process q that it had heard from process p that its proposed value is v. The value initially proposed by the process itself is stored at the root of the tree.

We introduce the type of *labels*, which encapsulate lists of distinct process identifiers and whose length is at most $f+1$.

definition $Label = \{xs::Proc\ list.\ length\ xs \le Suc\ f \wedge distinct\ xs\}$
typedef $Label = Label$
⟨*proof*⟩

There is a finite number of different labels.

lemma *finite-Label*: *finite Label*

⟨*proof*⟩

lemma *finite-UNIV-Label*: *finite* (*UNIV*::*Label set*)
⟨*proof*⟩

lemma *finite-Label-set* [*iff*]: *finite* (*S* :: *Label set*)
⟨*proof*⟩

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*
⟨*proof*⟩

definition *is-leaf* **where**
is-leaf l ≡ *length-lbl l* = *Suc f*

definition *last-lbl* **where**
last-lbl l ≡ *last* (*Rep-Label l*)

definition *butlast-lbl* **where**
butlast-lbl l ≡ *Abs-Label* (*butlast* (*Rep-Label l*))

definition *set-lbl* **where**
set-lbl l = *set* (*Rep-Label l*)

The children of a non-leaf label are all possible extensions of that label.

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

10.2 Model of the Algorithm

The following record models the local state of a process.

record *'val pstate* =
vals :: *Label* ⇒ *'val option*
newvals :: *Label* ⇒ *'val*
decide :: *'val option*

Initially, no values are assigned to non-root labels, and an arbitrary value is assigned to the root: that value is interpreted as the initial proposal of the process. No decision has yet been taken, and the *newvals* field is unconstrained.

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

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

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

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

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

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

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

In the final round, in addition to extending the tree as described previously, processes construct the tree *newvals*, starting at the leaves. The values at the leaves are copied from *vals*, except that missing values *None* are replaced by the default value *undefined*. Moving up, if there exists a majority value among the children, it is assigned to the parent node, otherwise the parent node receives the default value *undefined*. The decision is set to the value computed for the root of the tree.

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

definition *has-majority* :: *'val* ⇒ (*'a* ⇒ *'val*) ⇒ *'a set* ⇒ *bool* **where**
  has-majority v g S ≡ *card* {*e* ∈ *S*. *g e* = *v*} > (*card S*) *div 2*

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

definition *next-end* **where**
  next-end r p st msgs st' ≡
    extend-vals r p st msgs st'
  ∧ *check-newvals st'*
  ∧ *decide st'* = *Some* (*newvals st' root-node*)

The overall next-state relation is defined such that every process applies *nextMain* during rounds *0*, ..., *f−1*, and applies *nextEnd* during round *f*. After that, the algorithm terminates and nothing changes anymore.

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

10.3 Communication Predicate for $EIGByz_f$

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

definition *SKr* :: *Proc HO* ⇒ *Proc HO* ⇒ *Proc set* **where**
  SKr HO SHO ≡ { *q* . ∀ *p*. *q* ∈ *HO p* ∩ *SHO p*}

The secure kernel *SK* of an entire execution (i.e., for sequences of HO and SHO collections) is the intersection of the secure kernels for all rounds. Obviously, only the first *f* rounds really matter, since the algorithm terminates after that.

definition *SK* :: (*nat* ⇒ *Proc HO*) ⇒ (*nat* ⇒ *Proc HO*) ⇒ *Proc set* **where**
  SK HOs SHOs ≡ {*q*. ∀ *r*. *q* ∈ *SKr* (*HOs r*) (*SHOs r*)}

The round-by-round predicate requires that the secure kernel at every round contains more than $(N+f)$ *div 2* processes.

definition *EIG-commPerRd* **where**
  EIG-commPerRd HO SHO ≡ *card* (*SKr HO SHO*) > $(N + f)$ *div 2*

The global predicate requires that the secure kernel for the entire execution contains at least $N-f$ processes. Messages from these processes are always correctly received by all processes.

definition *EIG-commGlobal* **where**
  EIG-commGlobal HOs SHOs ≡ *card* (*SK HOs SHOs*) ≥ $N - f$

The above communication predicates differ from Lynch's presentation of $EIGByz_f$. In fact, the algorithm was originally designed for synchronous systems with reliable links and at most *f* faulty processes. In such a system, every process receives the correct message from at least the non-faulty processes at every round, and therefore the global predicate *EIG-commGlobal* is satisfied. The standard correctness proof assumes that $N > 3f$, and therefore $N - f > (N + f) \div 2$. Since moreover, for any r, we obviously have

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

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

However, our proof shows that *EIGByz*$_f$ can indeed tolerate more transient faults than the standard bound can express. For example, consider the case where $N = 5$ and $f = 2$. Our predicates are satisfied in executions where two processes exhibit transient faults, but never fail simultaneously. Indeed, in such an execution, every process receives four correct messages at every round, hence *EIG-commPerRd* always holds. Also, *EIG-commGlobal* is satisfied because there are three processes from which every process receives the correct messages at all rounds. By our correctness proof, it follows that $EIGByz_f$ then achieves Consensus, unlike what one could expect from the standard correctness predicate. This observation underlines the interest of expressing assumptions about transient faults, as in the HO model.

10.4 The *EIGByz*$_f$ Heard-Of Machine

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

definition *EIG-SHOMachine* **where**
  EIG-SHOMachine = (|
    CinitState = (λ *p st crd*. *EIG-initState p st*),
    sendMsg = *EIG-sendMsg*,
    CnextState = (λ *r p st msgs crd st'*. *EIG-nextState r p st msgs st'*),
    SHOcommPerRd = *EIG-commPerRd*,
    SHOcommGlobal = *EIG-commGlobal*
  |)

abbreviation *EIG-M* ≡ (*EIG-SHOMachine*::(*Proc*, *'val pstate*, *'val Msg*) *SHOMachine*)

end
theory *EigbyzProof*
imports *EigbyzDefs ../Majorities ../Reduction*
begin

10.5 Preliminary Lemmas

Some technical lemmas about labels and trees.

lemma *not-leaf-length*:
assumes *l*: $\neg$(*is-leaf l*)
shows *length-lbl l* $\leq$ *f*
⟨*proof*⟩

lemma *nil-is-Label*: [] $\in$ *Label*
⟨*proof*⟩

lemma *card-set-lbl*: *card* (*set-lbl l*) = *length-lbl l*
⟨*proof*⟩

lemma *Rep-Label-root-node* [*simp*]: *Rep-Label root-node* = []
⟨*proof*⟩

lemma *root-node-length* [*simp*]: *length-lbl root-node* = *0*
⟨*proof*⟩

lemma *root-node-not-leaf*: $\neg$(*is-leaf root-node*)
⟨*proof*⟩

Removing the last element of a non-root label gives a label.

lemma *butlast-rep-in-label*:
assumes *l*:*l* $\neq$ *root-node*
shows *butlast* (*Rep-Label l*) $\in$ *Label*
⟨*proof*⟩

The label of a child is well-formed.

lemma *Rep-Label-append*:
assumes *l*: $\neg$(*is-leaf l*)
shows (*Rep-Label l* @ [*p*] $\in$ *Label*) = (*p* $\notin$ *set-lbl l*)
(**is** *?lhs* = *?rhs* **is** (*?l'* $\in$ -) = -)
⟨*proof*⟩

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

lemma *label-children*:
assumes *c*: *c* $\in$ *children l*
shows $\exists$ *p*. *p* $\notin$ *set-lbl l* $\wedge$ *Rep-Label c* = *Rep-Label l* @ [*p*]
⟨*proof*⟩

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

lemma *children-length*:
assumes *l* $\in$ *children h*
shows *length-lbl l* = *Suc* (*length-lbl h*)
⟨*proof*⟩

The root node is never a child.

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

shows *P*
⟨*proof*⟩

The label of a child with the last element removed is the label of the parent.

lemma *children-butlast-lbl*:
assumes $c \in$ *children l*
shows *butlast-lbl c* = *l*
⟨*proof*⟩

The root node is not a child, and it is the only such node.

lemma *root-iff-no-child*: (l = *root-node*) = ($\forall l'$. $l \notin$ *children* l')
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

Suppose a non-root label l' of length $r+1$ ending in q, and suppose that q is well heard by process p in round r. Then the value with which p decorates l is the one that q associates to the parent of l.

lemma *sho-correct-vals*:
assumes *run*: *SHORun EIG-M rho HOs SHOs*
and l': $l' \in$ *children l*
and *shop*: *last-lbl* $l' \in$ *SHOs* (*length-lbl l*) *p* ∩ *HOs* (*length-lbl l*) *p*
(**is** *?q* ∈ *SHOs* (*?len l*) *p* ∩ -)
shows *vals* (*rho* (*?len* l') *p*) l' = *vals* (*rho* (*?len l*) *?q*) *l*
⟨*proof*⟩

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

lemma *keep-vals*:
assumes *run*: *SHORun EIG-M rho HOs SHOs*

shows *vals* (*rho* (*length-lbl l* + *n*) *p*) *l* = *vals* (*rho* (*length-lbl l*) *p*) *l*
(**is** *?v n* = *?vl*)
⟨*proof*⟩

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': $l' \in$ *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' = *vals* (*rho* (*length-lbl* l') *q*) l'
⟨*proof*⟩

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*: $l \in$ *children t*
and *skrl*: *last-lbl l* $\in$ *SKr* (*HOs* (*length-lbl t*)) (*SHOs* (*length-lbl t*))
and l': $l' \in$ *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' = *vals* (*rho* (*length-lbl l*) *p*) *l*
⟨*proof*⟩

For any non-leaf label l, more than half of its children end with a process that is well heard by everyone at round *length-lbl l*.

lemma *lynch-6-16-c*:
assumes *commR*: *EIG-commPerRd* (*HOs* (*length-lbl l*)) (*SHOs* (*length-lbl l*))
(**is** *EIG-commPerRd* (*HOs ?r*) -)
and *l*: ¬(*is-leaf l*)
shows *card* {$l' \in$ *children l*. *last-lbl* $l' \in$ *SKr* (*HOs ?r*) (*SHOs ?r*)}
> *card* (*children l*) *div 2*
(**is** *card ?lhs* > -)
⟨*proof*⟩

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

lemma *newvals-skr-uniform*:
assumes *run*: *SHORun EIG-M rho HOs SHOs*
and *commR*: *EIG-commPerRd* (*HOs* (*length-lbl l*)) (*SHOs* (*length-lbl l*))
(**is** *EIG-commPerRd* (*HOs ?r*) -)
and *notleaf*: ¬(*is-leaf l*)

and *unif*: $\bigwedge l'. [\![l' \in children\ l;$
$last\text{-}lbl\ l' \in SKr\ (HOs\ (length\text{-}lbl\ l))\ (SHOs\ (length\text{-}lbl\ l))$
$]\!] \Longrightarrow newvals\ (rho\ (Suc\ f)\ p)\ l' = v$
shows $newvals\ (rho\ (Suc\ f)\ p)\ l = v$
⟨*proof*⟩

A node whose label l ends with a process which is well heard at round *length-lbl* l will have its *newvals* field set (at round $f+1$) to the "fixed-up" value given by *vals*.

lemma *lynch-6-16-d*:
assumes *run*: *SHORun EIG-M rho HOs SHOs*
and *commR*: $\forall r.\ EIG\text{-}commPerRd\ (HOs\ r)\ (SHOs\ r)$
and *notroot*: $l \in children\ t$
and *skr*: $last\text{-}lbl\ l \in SKr\ (HOs\ (length\text{-}lbl\ t))\ (SHOs\ (length\text{-}lbl\ t))$
(**is** - $\in SKr\ (HOs\ (?len\ t))$ -)
shows $newvals\ (rho\ (Suc\ f)\ p)\ l = fixupval\ (vals\ (rho\ (?len\ l)\ p)\ l)$
(**is** *?P l*)
⟨*proof*⟩

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**
$common\ rho\ l \equiv$
$\forall p\ q.\ newvals\ (rho\ (Suc\ f)\ p)\ l = newvals\ (rho\ (Suc\ f)\ q)\ l$

The subtrees of a given label are all its possible extensions.

definition *subtrees* **where**
$subtrees\ h \equiv \{\ l\ .\ \exists t.\ Rep\text{-}Label\ l = (Rep\text{-}Label\ h)\ @\ t\ \}$

lemma *children-in-subtree*:
assumes $l \in children\ h$
shows $l \in subtrees\ h$
⟨*proof*⟩

lemma *subtrees-refl* [*iff*]: $l \in subtrees\ l$
⟨*proof*⟩

lemma *subtrees-root* [*iff*]: $l \in subtrees\ root\text{-}node$
⟨*proof*⟩

lemma *subtrees-trans*:
assumes $l'' \in subtrees\ l'$ **and** $l' \in subtrees\ l$
shows $l'' \in subtrees\ l$
⟨*proof*⟩

lemma *subtrees-antisym*:
  assumes $l \in$ *subtrees* l' **and** $l' \in$ *subtrees* l
  shows $l' = l$
  ⟨*proof*⟩

lemma *subtrees-tree*:
  assumes l': $l \in$ *subtrees* l' **and** l'': $l \in$ *subtrees* l''
  shows $l' \in$ *subtrees* $l'' \lor l'' \in$ *subtrees* l'
⟨*proof*⟩

lemma *subtrees-cases*:
  assumes l': $l' \in$ *subtrees* l
    and *self*: $l' = l \Longrightarrow P$
    and *child*: $\bigwedge c.$ $\llbracket$ $c \in$ *children* l; $l' \in$ *subtrees* c $\rrbracket \Longrightarrow P$
  shows P
⟨*proof*⟩

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

lemma *children-subtrees-equal*:
  assumes c: $c \in$ *children* l **and** c': $c' \in$ *children* l
    and *sub*: $c' \in$ *subtrees* c
  shows $c' = c$
⟨*proof*⟩

A set C of labels is a *subcovering* w.r.t. label l if for all leaf subtrees s of l there exists some label $h \in C$ such that s is a subtree of h and h is a subtree of l.

definition *subcovering* **where**
subcovering C $l \equiv$
  $\forall s \in$ *subtrees* l. *is-leaf* $s \longrightarrow (\exists h \in C.\ h \in$ *subtrees* $l \land s \in$ *subtrees* $h)$

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

abbreviation *covering* **where**
  covering $C \equiv$ *subcovering* C *root-node*

The set of labels whose last element is well heard by all processes throughout the execution forms a covering, and all these labels are common.

lemma *lynch-6-18-a*:
  assumes *SHORun EIG-M rho HOs SHOs*
    and $\forall r.$ *EIG-commPerRd* (*HOs* r) (*SHOs* r)
    and $l \in$ *children* t
    and *last-lbl* $l \in$ *SKr* (*HOs* (*length-lbl* t)) (*SHOs* (*length-lbl* t))
  shows *common rho* l
  ⟨*proof*⟩

lemma *lynch-6-18-b*:
  assumes *run*: *SHORun EIG-M rho HOs SHOs*
    and *commG*: *EIG-commGlobal HOs SHOs*
    and *commR*: $\forall$ *r*. *EIG-commPerRd* (*HOs r*) (*SHOs r*)
  shows *covering* {*l*. $\exists$ *t*. *l* $\in$ *children t* $\wedge$ *last-lbl l* $\in$ (*SK HOs SHOs*)}
⟨*proof*⟩

If C covers the subtree rooted at label l and if $l \notin C$ then C also covers subtrees rooted at l's children.

lemma *lynch-6-19-a*:
  assumes *cov*: *subcovering C l*
    and *l*: *l* $\notin$ *C*
    and *e*: *e* $\in$ *children l*
  shows *subcovering C e*
⟨*proof*⟩

If there is a subcovering C for a label l such that all labels in C are common, then l itself is common as well.

lemma *lynch-6-19-b*:
  assumes *run*: *SHORun EIG-M rho HOs SHOs*
    and *cov*: *subcovering C l*
    and *com*: $\forall$ *l'* $\in$ *C*. *common rho l'*
  shows *common rho l*
⟨*proof*⟩

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*: $\forall$ *r*. *EIG-commPerRd* (*HOs r*) (*SHOs r*)
  shows *common rho root-node*
⟨*proof*⟩

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*⟩

10.7 Proof of Agreement, Validity, and Termination

The Agreement property is an immediate consequence of lemma *lynch-6-20*.

theorem *Agreement*:

assumes *run*: *SHORun EIG-M rho HOs SHOs*
and *commG*: *EIG-commGlobal HOs SHOs*
and *commR*: $\forall$ *r. EIG-commPerRd* (*HOs r*) (*SHOs r*)
and *p*: *decide* (*rho m p*) = *Some v*
and *q*: *decide* (*rho n q*) = *Some w*
shows *v* = *w*
⟨*proof*⟩

We now show the Validity property: if all processes initially propose the same value *v*, then no other value may be decided.

By lemma *sho-correct-vals*, value *v* must propagate to all children of the root that are well heard at round *0*, and lemma *lynch-6-16-d* implies that *v* is the value assigned to all these children by *newvals*. Finally, lemma *newvals-skr-uniform* lets us conclude.

theorem *Validity*:
assumes *run*: *SHORun EIG-M rho HOs SHOs*
and *commR*: $\forall$ *r. EIG-commPerRd* (*HOs r*) (*SHOs r*)
and *initv*: $\forall$ *q. the* (*vals* (*rho 0 q*) *root-node*) = *v*
and *dp*: *decide* (*rho r p*) = *Some w*
shows *v* = *w*
⟨*proof*⟩

Termination is trivial for $EIGByz_f$.

theorem *Termination*:
assumes *SHORun EIG-M rho HOs SHOs*
shows $\exists$ *r v. decide* (*rho r p*) = *Some v*
⟨*proof*⟩

10.8 $EIGByz_f$ Solves Weak Consensus

Summing up, all (coarse-grained) runs of $EIGByz_f$ for HO and SHO collections that satisfy the communication predicate satisfy the Weak Consensus property.

theorem *eig-weak-consensus*:
assumes *run*: *SHORun EIG-M rho HOs SHOs*
and *commR*: $\forall$ *r. EIG-commPerRd* (*HOs r*) (*SHOs r*)
and *commG*: *EIG-commGlobal HOs SHOs*
shows *weak-consensus* (λ*p. the* (*vals* (*rho 0 p*) *root-node*)) *decide rho*
⟨*proof*⟩

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*: $\forall$ *r. EIG-commPerRd* (*HOs r*) (*SHOs r*)
and *commG*: *EIG-commGlobal HOs SHOs*
shows *weak-consensus* (λ*p. the* (*vals* (*state* (*rho 0*) *p*) *root-node*))

decide (*state* ∘ *rho*)
(**is** *weak-consensus ?inits* - -)
⟨*proof*⟩

end

11 Conclusion

In this contribution we have formalized the Heard-Of model in the proof assistant Isabelle/HOL. We have established a formal framework, in which fault-tolerant distributed algorithms can be represented, and that caters for different variants (benign or malicious faults, coordinated and uncoordinated algorithms). We have formally proved a reduction theorem that relates fine-grained (asynchronous) interleaving executions and coarse-grained executions, in which an entire round constitutes the unit of atomicity. As a corollary, many correctness properties, including Consensus, can be transferred from the coarse-grained to the fine-grained representation.

We have applied this framework to give formal proofs in Isabelle/HOL for six different Consensus algorithms known from the literature. Thanks to the reduction theorem, it is enough to verify the algorithms over coarse-grained runs, and this keeps the effort manageable. For example, our *LastVoting* algorithm is similar to the DiskPaxos algorithm verified in [10], but our proof here is an order of magnitude shorter, although we prove safety and liveness properties, whereas only safety was considered in [10].

We also emphasize that the uniform characterization of fault assumptions via communication predicates in the HO model lets us consider the effects of transient failures, contrary to standard models that consider only permanent failures. For example, our correctness proof for the $EIGByz_f$ algorithm establishes a stronger result than that claimed by the designers of the algorithm. The uniform presentation also paves the way towards comparing assumptions of different algorithms.

The encoding of the HO model as Isabelle/HOL theories is quite straightforward, and we find our Isar proofs quite readable, although they necessarily contain the full details that are often glossed over in textbook presentations. We believe that our framework allows algorithm designers to study different fault-tolerant distributed algorithms, their assumptions, and their proofs, in a clear, rigorous and uniform way.

References

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

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

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

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

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

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

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

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

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

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

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

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