Verification of Heard-Of Algorithms in Isabelle

Stephan Merz

July 30, 2009

Contents

theory *CHO*
imports *Main*
begin

1 Heard-Of Algorithms

We propose a generic representation of (coordinated) HO algorithms [1] in Isabelle/HOL.

An HO algorithm executes a sequence of rounds. A concrete algorithm is described by the following parameters:

- a type *'proc* of processes whose extension is assumed to be finite,
- a type *'pst* of local process states,
- a type *'msg* of messages sent in the course of the algorithm,
- a predicate *initState* such that *initState p st* is true precisely of the initial states *st* of process *p*,
- a function *sendMsg* where *sendMsg r p q st crd* yields the message that process *p* sends to process *q* at round *r*, given its local state *st* and coordinator *crd*, and
- a predicate *nextState* where *nextState r p st msgs crd st'* characterizes the successor states *st'* of state *st* for process *p* at round *r*, where *crd* denotes the process that *p* believes to be the coordinator of round *r* and the function *msgs* :: *'proc* ⇒ *'msg option* represents the vector of messages that *p* received at round *r*,

- a communication predicate that constrains the heard-of and coordinator assignments (see below) that may occur during a run. For convenience, we split this predicate into a *safety* part that should hold at every round and a *liveness* part that should hold of the sequence of HO assignments.

An uncoordinated algorithm simply ignores the parameter *crd* of functions *nextState* and *sendMsg*. Similarly, the communication predicate does not refer to the coordinator assignment. The HO model assumes communication-closed rounds, that is, processes receive only messages sent for the round they are currently in. By a general result on the HO model, it can be assumed that each round is executed atomically. A snapshot of the system can therefore be represented by the local states of each process at the beginning of a round. The messages sent can be computed from the local state, so they do not have to be recorded explicitly.

We represent a system configuration as an array of process states. A system run is just an infinite sequence of configurations. At this generic level, process states are left parametric (represented by a type variable); they will be defined by particular algorithms. (For some reason type and record definitions cannot go inside locale definitions so we introduce them beforehand.)

```
types
  ('proc,'pst) run = nat ⇒ 'proc ⇒ 'pst
```

A *heard-of assignment* associates a set of processes with each process. The idea is that *HO p* designates the set of processes from which process *p* receives a message at the current round. A *coordinator assignment* associates a process (the coordinator) to each process.

```
types
  'proc HO = 'proc ⇒ 'proc set

types
  'proc coord = 'proc ⇒ 'proc

locale CHOAlgorithm =
  fixes
    initState :: 'proc ⇒ 'pst ⇒ bool
  and
    sendMsg :: nat ⇒ 'proc ⇒ 'proc ⇒ 'pst ⇒ 'proc ⇒ 'msg
  and
    nextState :: nat ⇒ 'proc ⇒ 'pst ⇒ ('proc ⇒ 'msg option) ⇒ 'proc ⇒ 'pst ⇒ bool
  and
    commSafe :: nat ⇒ 'proc HO ⇒ 'proc coord ⇒ bool
  and
    commLive :: (nat ⇒ 'proc HO) ⇒ (nat ⇒ 'proc coord) ⇒ bool
  assumes
    finiteProc: finite (UNIV::'proc set)
begin
```

By assumption *finiteProc*, any set of processes is finite.

```
lemma finiteProcset [simp,intro]: finite (P::'proc set)
using finiteProc by (blast intro:finite-subset)
```

Similarly, the range of any partial function from *Proc* is finite. (The Isabelle library contains a similar lemma for the range of a total function, a generalization of the following lemma could go to the standard library.)

```
lemma finite-ran: finite (ran (f :: 'proc ⇀ 'a))
```

proof −
  let *?g* = λ*y*. *case y of None => arbitrary* | *Some x => x*
  have *ran f* ⊆ *?g* ‘ (*range f*)
  proof
    fix *y*
    assume *y* ∈ *ran f*
    then obtain *x* **where** *f x* = *Some y* **by** (*auto simp add*: *ran-def*)
    hence *y* = *?g* (*f x*) **by** *simp*
    thus *y* ∈ *?g* ‘ (*range f*) **by** *blast*
  qed
  moreover
  have *finite* (*?g* ‘ *range f*) **by** *auto*
  ultimately
  show *?thesis* **by** (*rule finite-subset*)
qed

Any two sets S and T of processes such that the sum of their cardinalities exceeds the number of processes have a non-empty intersection.

lemma *majorities-intersect*:
  assumes *crd*: *card* (*UNIV*::'*proc set*) < *card* (*S*::'*proc set*) + *card* (*T*::'*proc set*)
  shows *S* ∩ *T* ≠ {}
proof (*clarify*)
  assume *contra*: *S* ∩ *T* = {}
  with *crd* **have** *card* (*UNIV*::'*proc set*) < *card* (*S* ∪ *T*)
    by (*auto simp add*: *card-Un-Int*)
  moreover have *card* (*S* ∪ *T*) ≤ *card* (*UNIV*::'*proc set*)
    by (*simp add*: *card-mono*)
  ultimately show *False*
    by *simp*
qed

lemma *majoritiesE*:
  assumes *crd*: *card* (*UNIV*::'*proc set*) < *card* (*S*::'*proc set*) + *card* (*T*::'*proc set*)
  obtains *p* **where** *p* ∈ *S* **and** *p* ∈ *T*
using *crd majorities-intersect* **by** *blast*

Frequent special case

lemma *majoritiesE'*:
  assumes *S*: *card* (*S*::'*proc set*) > (*card* (*UNIV*::'*proc set*)) *div 2*
  and *T*: *card* (*T*::'*proc set*) > (*card* (*UNIV*::'*proc set*)) *div 2*
  obtains *p* **where** *p* ∈ *S* **and** *p* ∈ *T*
proof (*rule majoritiesE*)
  from *S T* **show** *card* (*UNIV*::'*proc set*) < *card S* + *card T* **by** *auto*
qed

Because messages are not corrupted in the HO model and processes only react to messages sent at the current round, we need not explicitly represent the network state in the runs and use the following utility function to compute the messages that a process receives.

The function *rcvMsgs* computes the messages that process p receives at round r, given a Heard-Of set, the collections of coordinators and process states, and a message send function. (This last parameter is useful in applications because *rcvdMsgs* can be used with sub-functions of the overall message sending function used by the algorithm.)

definition
  rcvdMsgs **where**

$rcvdMsgs\ (p::'proc)\ (HO::'proc\ set)\ (coord::'proc\ coord)\ (cfg::'proc \Rightarrow 'pst)$
$\quad (send::'proc \Rightarrow 'proc \Rightarrow 'pst \Rightarrow 'proc \Rightarrow 'msg)$
$\equiv \lambda q.\ if\ q \in HO\ then\ Some\ (send\ q\ p\ (cfg\ q)\ (coord\ q))\ else\ None$

An initial configuration is one where all processes are in an initial state.

definition
initConfig **where**
$initConfig\ cfg \equiv \forall p.\ initState\ p\ (cfg\ p)$

The following definition characterizes successor configurations *cfg'* of a source configuration *cfg* at round *r*, given assignments *HO* of heard-of sets and *coord* of coordinators.

definition
nextConfig **where**
$nextConfig\ r\ cfg\ (HO :: 'proc\ HO)\ (coord :: 'proc\ coord)\ cfg' \equiv$
$\forall p.\ nextState\ r\ p\ (cfg\ p)\ (rcvdMsgs\ p\ (HO\ p)\ coord\ cfg\ (sendMsg\ r))\ (coord\ p)\ (cfg'\ p)$

Given heard-of and coordinator collections, i.e. a heard-of and coordinator assignment for each round, a run ρ of the algorithm is a sequence of configurations starting with an initial configuration and respecting the successor function *nextConfig*.

definition
CHORun **where**
$CHORun\ rho\ HOs\ coords \equiv$
$\quad (initConfig\ (rho\ 0))$
$\wedge (\forall r.\ commSafe\ r\ (HOs\ r)\ (coords\ r)$
$\quad\quad \wedge\ nextConfig\ r\ (rho\ r)\ (HOs\ r)\ (coords\ r)\ (rho\ (Suc\ r)))$
$\wedge\ commLive\ HOs\ coords$

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

lemma *CHORun-0*:
assumes *CHORun rho HOs coords* **and** $\bigwedge cfg.\ initConfig\ cfg \Longrightarrow P\ cfg$
shows $P\ (rho\ 0)$
using *prems* **unfolding** *CHORun-def* **by** *blast*

lemma *CHORun-Suc*:
assumes *CHORun rho HOs coords*
and $\bigwedge r.\ [\![\ commSafe\ r\ (HOs\ r)\ (coords\ r);$
$\quad nextConfig\ r\ (rho\ r)\ (HOs\ r)\ (coords\ r)\ (rho\ (Suc\ r))\ ]\!]$
$\quad \Longrightarrow P\ r$
shows $P\ n$
using *prems* **unfolding** *CHORun-def* **by** *blast*

lemma *CHORun-induct*:
assumes *run*: *CHORun rho HOs coords*
and *init*: $initConfig\ (rho\ 0) \Longrightarrow P\ 0$
and *step*: $\bigwedge r.\ [\![\ P\ r;\ commSafe\ r\ (HOs\ r)\ (coords\ r);$
$\quad nextConfig\ r\ (rho\ r)\ (HOs\ r)\ (coords\ r)\ (rho\ (Suc\ r))\ ]\!]$
$\quad \Longrightarrow P\ (Suc\ r)$
shows $P\ n$
using *run* **unfolding** *CHORun-def* **by** (*induct n*, *auto elim*: *init step*)

end — locale CHOAlgorithm

end — theory CHO

theory *LastVoting*
imports *CHO*
begin

2 Verification of the *LastVoting* Consensus Algorithm

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

The *LastVoting* algorithm can be considered as a version of Lamport's Paxos consensus algorithm [2] for the Heard-Of model. Following [1], we define the algorithm as an instance of the generic Heard-Of model.

2.1 Formal Model of *LastVoting*

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*

axioms
  procFinite: *finite* (*UNIV*::*Proc set*)

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*
by (*simp add*: *phase-def*)

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

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

The following record models the local state of a process.

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

Possible messages sent during 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 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 $initState$ **where**
  $initState\ p\ st \equiv$
   $(vote\ st = None) \wedge \neg\ (commt\ st) \wedge \neg(ready\ st) \wedge (timestamp\ st = 0) \wedge (decide\ st = None)$

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

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

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

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

A process that considers itself to be a coordinator updates its $vote$ and $commt$ fields if it has received messages from a majority of processes.

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

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

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**
  $send1\ r\ p\ q\ st\ crd \equiv$

$if\ p = crd \wedge commt\ st\ then\ Vote\ (the\ (vote\ st))\ else\ Null$

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

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**
$send2\ r\ p\ q\ st\ crd \equiv$
$if\ timestamp\ st = Suc(phase\ r) \wedge q = crd\ then\ (Ack{::}'val\ msg)\ else\ Null$

definition *acksRcvd* **where** — processes from which an acknowledgement was received
$acksRcvd\ (msgs :: Proc \rightharpoonup 'val\ msg) \equiv$
$\{\ q\ .\ msgs\ q \neq None \wedge isAck\ (the\ (msgs\ q))\ \}$

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

In step 3, coordinators that are ready send their vote to all processes.

Processes that received a vote from their coordinator decide on that value. Coordinators reset their *ready* and *commt* fields to false.

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

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

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

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

definition
$nextState :: nat \Rightarrow Proc \Rightarrow 'val\ pstate \Rightarrow (Proc \rightharpoonup 'val\ msg) \Rightarrow Proc \Rightarrow 'val\ pstate \Rightarrow bool$
where

nextState r ≡
if step r = 0 then next0 r
else if step r = 1 then next1 r
else if step r = 2 then next2 r
else next3 r

We now define the communication predicate for the LastVoting algorithm. The safety part is trivial: integrity and agreement are always ensured. However, coordinators are supposed to change only between phases. For the liveness part, Charron and Bost 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 [1], but obviously sufficient).

In fact, it is enough (as noted in the text of [1]) to require the existence of a single such phase.

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

definition
LV-commLive **where**
LV-commLive HOs coords ≡
(∀ *r*. *step r* ≠ *3* ⟶ *coords* (*Suc r*) = *coords r*)
∧ (∃ (*ph*::*nat*). ∃ (*c*::*Proc*).
(∀ *p*. *coords* (*4*∗*ph*) *p* = *c*)
∧ *card* (*HOs* (*4*∗*ph*) *c*) > *N div 2* ∧ *card* (*HOs* (*Suc* (*Suc* (*4*∗*ph*))) *c*) > *N div 2*
∧ (∀ *p*. *c* ∈ *HOs* (*Suc* (*4*∗*ph*)) *p* ∩ *HOs* (*Suc* (*Suc* (*Suc* (*4*∗*ph*)))) *p*))

We instantiate the generic definition of Heard-Of algorithms for the LastVoting algorithm.

interpretation *CHOAlgorithm initState sendMsg nextState LV-commSafe LV-commLive*
by (*unfold-locales*, *rule procFinite*)

2.2 Proof of *LastVoting*: Preliminary Lemmas

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

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

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

```
  let ?ts = {ts . ∃ q v. msgs q = Some (ValStamp v ts)}
  from nempty have ?ts ≠ {} by (auto simp add: valStampsRcvd-def)
  with timeStampsRcvdFinite
  have highestStampRcvd msgs ∈ ?ts unfolding highestStampRcvd-def by (rule Max-in)
  then obtain p v where msgs p = Some (ValStamp v (highestStampRcvd msgs))
    by (auto simp add: highestStampRcvd-def)
  with that show thesis .
qed

lemma highestStampRcvd-max:
  assumes msgs p = Some (ValStamp v ts)
  shows ts ≤ highestStampRcvd msgs
using prems unfolding highestStampRcvd-def
by (blast intro: Max-ge timeStampsRcvdFinite)
```

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 rho HOs coords
  and init: ∀ p. initState p (rho 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)
                              (rcvdMsgs p (HOs r p) (coords r) (rho r) (send0 r))
                              (coords 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)
                              (rcvdMsgs p (HOs r p) (coords r) (rho r) (send1 r))
                              (coords r p)
                              (rho (Suc r) p) ⟧
                 ⟹ P (Suc r)
  and step2: ⋀r.
                 ⟦ step r = 2; P r; phase (Suc r) = phase r; step (Suc r) = 3;
                   ∀ p. next2 r p (rho r p)
                              (rcvdMsgs p (HOs r p) (coords r) (rho r) (send2 r))
                              (coords r p)
                              (rho (Suc r) p) ⟧
                 ⟹ P (Suc r)
  and step3: ⋀r.
                 ⟦ step r = 3; P r; phase (Suc r) = Suc (phase r); step (Suc r) = 0;
                   ∀ p. next3 r p (rho r p)
                              (rcvdMsgs p (HOs r p) (coords r) (rho r) (send3 r))
                              (coords r p)
                              (rho (Suc r) p) ⟧
                 ⟹ P (Suc r)
  shows P n
proof (rule CHORun-induct[OF run])
  assume initConfig (rho 0)
  thus P 0 by (auto simp add: initConfig-def init)
next
  fix r
  assume ih: P r and nxt: nextConfig r (rho r) (HOs r) (coords r) (rho (Suc r))
```

have *step r* ∈ {*0*,*1*,*2*,*3*} **by** (*auto simp add*: *step-def*)
thus *P* (*Suc r*)
proof *auto*
assume *stp*: *step r* = *0*
hence *stp'*: *step* (*Suc r*) = *1* **by** (*auto simp add*: *step-def mod-Suc*)
from *stp* **have** *ph*: *phase* (*Suc r*) = *phase r* **by** (*auto simp add*: *phase-def step-def*)
from *ih nxt stp stp' ph* **show** *?thesis*
by (*intro step0*, *auto simp add*: *nextConfig-def nextState-def sendMsg-def*)
next
assume *stp*: *step r* = *Suc 0*
hence *stp'*: *step* (*Suc r*) = *2* **by** (*auto simp add*: *step-def mod-Suc*)
from *stp* **have** *ph*: *phase* (*Suc r*) = *phase r*
unfolding *step-def phase-def* **by** *presburger*
from *ih nxt stp stp' ph* **show** *?thesis*
by (*intro step1*, *auto simp add*: *nextConfig-def nextState-def sendMsg-def*)
next
assume *stp*: *step r* = *2*
hence *stp'*: *step* (*Suc r*) = *3* **by** (*auto simp add*: *step-def mod-Suc*)
from *stp* **have** *ph*: *phase* (*Suc r*) = *phase r*
unfolding *step-def phase-def* **by** *presburger*
from *ih nxt stp stp' ph* **show** *?thesis*
by (*intro step2*, *auto simp add*: *nextConfig-def nextState-def sendMsg-def*)
next
assume *stp*: *step r* = *3*
hence *stp'*: *step* (*Suc r*) = *0* **by** (*auto simp add*: *step-def mod-Suc*)
from *stp* **have** *ph*: *phase* (*Suc r*) = *Suc* (*phase r*)
unfolding *step-def phase-def* **by** *presburger*
from *ih nxt stp stp' ph* **show** *?thesis*
by (*intro step3*, *auto simp add*: *nextConfig-def nextState-def sendMsg-def*)
qed
qed

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

lemma *LV-Suc*:
assumes *run*: *CHORun rho HOs coords*
and *step0*: ⟦ *step r* = *0*; *step* (*Suc r*) = *1*; *phase* (*Suc r*) = *phase r*;
∀ *p*. *next0 r p* (*rho r p*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send0 r*))
(*coords r p*) (*rho* (*Suc r*) *p*) ⟧
⟹ *P r*
and *step1*: ⟦ *step r* = *1*; *step* (*Suc r*) = *2*; *phase* (*Suc r*) = *phase r*;
∀ *p*. *next1 r p* (*rho r p*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send1 r*))
(*coords r p*) (*rho* (*Suc r*) *p*) ⟧
⟹ *P r*
and *step2*: ⟦ *step r* = *2*; *step* (*Suc r*) = *3*; *phase* (*Suc r*) = *phase r*;
∀ *p*. *next2 r p* (*rho r p*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send2 r*))
(*coords r p*) (*rho* (*Suc r*) *p*) ⟧
⟹ *P r*
and *step3*: ⟦ *step r* = *3*; *step* (*Suc r*) = *0*; *phase* (*Suc r*) = *Suc* (*phase r*);
∀ *p*. *next3 r p* (*rho r p*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send3 r*))
(*coords r p*) (*rho* (*Suc r*) *p*) ⟧

$\Longrightarrow$ *P r*
shows *P r*
proof −
from *run* **have** *nxt*: *nextConfig r* (*rho r*) (*HOs r*) (*coords r*) (*rho* (*Suc r*))
by (*auto simp add*: *CHORun-def*)
have *step r* ∈ {*0*,*1*,*2*,*3*} **by** (*auto simp add*: *step-def*)
thus *P r*
proof (*auto*)
assume *stp*: *step r* = *0*
hence *stp'*: *step* (*Suc r*) = *1* **by** (*auto simp add*: *step-def mod-Suc*)
from *stp* **have** *ph*: *phase* (*Suc r*) = *phase r* **by** (*auto simp add*: *phase-def step-def*)
from *nxt stp stp' ph* **show** *?thesis*
by (*intro step0*, *auto simp add*: *nextConfig-def nextState-def sendMsg-def*)
next
assume *stp*: *step r* = *Suc 0*
hence *stp'*: *step* (*Suc r*) = *2* **by** (*auto simp add*: *step-def mod-Suc*)
from *stp* **have** *ph*: *phase* (*Suc r*) = *phase r*
unfolding *step-def phase-def* **by** *presburger*
from *nxt stp stp' ph* **show** *?thesis*
by (*intro step1*, *auto simp add*: *nextConfig-def nextState-def sendMsg-def*)
next
assume *stp*: *step r* = *2*
hence *stp'*: *step* (*Suc r*) = *3* **by** (*auto simp add*: *step-def mod-Suc*)
from *stp* **have** *ph*: *phase* (*Suc r*) = *phase r*
unfolding *step-def phase-def* **by** *presburger*
from *nxt stp stp' ph* **show** *?thesis*
by (*intro step2*, *auto simp add*: *nextConfig-def nextState-def sendMsg-def*)
next
assume *stp*: *step r* = *3*
hence *stp'*: *step* (*Suc r*) = *0* **by** (*auto simp add*: *step-def mod-Suc*)
from *stp* **have** *ph*: *phase* (*Suc r*) = *Suc* (*phase r*)
unfolding *step-def phase-def* **by** *presburger*
from *nxt stp stp' ph* **show** *?thesis*
by (*intro step3*, *auto simp add*: *nextConfig-def nextState-def sendMsg-def*)
qed
qed

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

lemma *LV-induct'*:
assumes *run*: *CHORun rho HOs coords*
and *init*: *initState p* (*rho 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*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send0 r*))
(*coords 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*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send1 r*))
(*coords 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*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send2 r*))
(*coords 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*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send3 r*))
(*coords r p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p* (*Suc r*)
shows *P p n*
by (*rule LV-induct*[*OF run*], *auto intro*: *init step0 step1 step2 step3*)

lemma *LV-Suc′*:
assumes *run*: *CHORun rho HOs coords*
and *step0*: ⟦ *step r* = *0*; *step* (*Suc r*) = *1*; *phase* (*Suc r*) = *phase r*;
next0 r p (*rho r p*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send0 r*))
(*coords 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*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send1 r*))
(*coords 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*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send2 r*))
(*coords 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*)
(*rcvdMsgs p* (*HOs r p*) (*coords r*) (*rho r*) (*send3 r*))
(*coords r p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p r*
shows *P p r*
by (*rule LV-Suc*[*OF run*], *auto intro*: *step0 step1 step2 step3*)

2.3 Boundedness and monotonicity of timestamps

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

lemma *LV-timestamp-bounded*:
assumes *run*: *CHORun rho HOs coords*
shows *timestamp* (*rho n p*) $\le$ (*if step n* < *2 then phase n else Suc* (*phase n*))
(**is** *?P p n*)
by (*rule LV-induct′* [*OF run*, **where** *P*=*?P*],
auto simp add: *initState-def next0-def next1-def next2-def next3-def*)

Moreover, timestamps can only grow over time.

lemma *LV-timestamp-increasing*:
assumes *run*: *CHORun rho HOs coords*
shows *timestamp* (*rho n p*) $\le$ *timestamp* (*rho* (*Suc n*) *p*)
(**is** *?P p n* **is** *?ts* $\le$ -)
proof (*rule LV-Suc′*[*OF run*, **where** *P*=*?P*])

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

fix *HO*
assume *stp*: *step n = 1*
and *nxt*: *next1 n p* (*rho n p*)
(*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send1 n*))
(*coords n p*) (*rho* (*Suc n*) *p*)
from *stp* **have** *?ts* ≤ *phase n*
using *LV-timestamp-bounded*[*OF run*, **where** *n=n*, **where** *p=p*] **by** *auto*
with *nxt* **show** *?thesis* **by** (*auto simp add*: *next1-def*)
qed (*auto simp add*: *next0-def next2-def next3-def*)

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

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

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

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

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

2.4 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. This is an immediate consequence of the communication/coordinator predicate.

lemma *notStep3EqualCoord*:
  assumes *CHORun rho HOs coords* **and** *step r* ≠ *3*
  shows *coords* (*Suc r*) *p* = *coords r p*
using *assms* **by** (*auto simp add*: *CHORun-def LV-commLive-def*)

Votes only change at step 0.

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

Commit status only changes at steps 0 and 3.

lemma *notStep03EqualCommit* [*rule-format*]:
  assumes *run*: *CHORun rho HOs coords*
  shows *step r* ≠ *0* ∧ *step r* ≠ *3* ⟶ *commt* (*rho* (*Suc r*) *p*) = *commt* (*rho r p*)
      (**is** *?P p r*)
by (*rule LV-Suc'*[*OF run*, **where** *P*=*?P*],
    auto simp add: *next0-def next1-def next2-def next3-def*)

Timestamps only change at step 1.

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

The x field only changes at step 1.

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

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

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

lemma *commitE*:
  assumes *run*: *CHORun rho HOs coords* **and** *cmt*: *commt* (*rho r p*)
  and *conds*: ⟦ *1* ≤ *step r*; *coords r p* = *p*; *vote* (*rho r p*) ≠ *None*;
          card {*q* . *coords r q* = *p*} > *N div 2*
        ⟧ ⟹ *A*
  shows *A*
proof –

have *commt* (*rho r p*) ⟶
1 ≤ *step r* ∧ *coords r p* = *p* ∧ *vote* (*rho r p*) ≠ *None* ∧ *card* {*q* . *coords r q* = *p*} > *N div 2*
(**is** *?P p r* **is** - ⟶ *?R r*)
proof (*rule LV-induct'*[*OF run*, **where** *P*=*?P*])
— the only interesting step is step 0
fix *n*
assume *nxt*: *next0 n p* (*rho n p*) (*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send0 n*)) (*coords n p*) (*rho* (*Suc n*) *p*)
and *ph*: *phase* (*Suc n*) = *phase n*
and *stp*: *step n* = *0* **and** *stp'*: *step* (*Suc n*) = *1*
and *ih*: *?P p n*
show *?P p* (*Suc n*)
proof
assume *cm'*: *commt* (*rho* (*Suc n*) *p*)
from *stp ih* **have** *cm*: ¬ *commt* (*rho n p*) **by** *simp*
with *nxt cm'*
have *coords n p* = *p* ∧ *vote* (*rho* (*Suc n*) *p*) ≠ *None*
∧ *card* (*valStampsRcvd* (*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send0 n*))) > *N div 2*
by (*auto simp add*: *next0-def*)
moreover
have *valStampsRcvd* (*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send0 n*)) ⊆ {*q* . *coords n q* = *p*}
by (*auto simp add*: *valStampsRcvd-def rcvdMsgs-def send0-def*)
hence *card* (*valStampsRcvd* (*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send0 n*))) ≤ *card* {*q* . *coords n q* = *p*}
by (*auto intro*: *card-mono*)
moreover
note *stp stp' run*
ultimately
show *?R* (*Suc n*)
by (*auto simp add*: *notStep3EqualCoord*)
qed
— the remaining cases are all solved by expanding the definitions
qed (*auto simp add*: *initState-def next1-def next2-def next3-def notStep3EqualCoord*[*OF run*])
with *cmt* **show** *?thesis* **by** (*intro conds*, *auto*)
qed

A process has a current timestamp only if:

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

lemma *currentTimestampE*:
assumes *run*: *CHORun rho HOs coords*
and *ts*: *timestamp* (*rho r p*) = *Suc* (*phase r*)
and *conds*: ⟦ *2* ≤ *step r*;
commt (*rho r* (*coords r p*));
x (*rho r p*) = *the* (*vote* (*rho r* (*coords r p*)))
⟧ ⟹ *A*
shows *A*
proof −
let *?ts n* = *timestamp* (*rho n p*)
let *?crd n* = *coords n p*
have *?ts r* = *Suc* (*phase r*) ⟶ *2* ≤ *step r* ∧ *commt* (*rho r* (*?crd r*)) ∧ *x* (*rho r p*) = *the* (*vote* (*rho r* (*?crd r*)))

(**is** *?Q p r* **is** - ⟶ *?R r*)
proof (*rule LV-induct′*[*OF run*, **where** *P=?Q*])
— The assertion is trivially true initially because the timestamp is 0.
assume *initState p* (*rho 0 p*) **thus** *?Q p 0*
by (*auto simp add*: *initState-def*)
next
— The assertion is trivially preserved by step 0 because the timestamp in the post-state cannot be current (cf. lemma *LV-timestamp-bounded*).
fix *n*
assume *stp′*: *step* (*Suc n*) = *1*
with *run LV-timestamp-bounded*[**where** *n=Suc n*] **have** *?ts* (*Suc n*) ≤ *phase* (*Suc n*)
by *auto*
thus *?Q p* (*Suc n*) **by** *simp*
next
— Step 1 establishes the assertion by definition of the transition relation.
fix *n*
assume *stp*: *step n* = *1* **and** *stp′*: *step* (*Suc n*) = *2*
and *ph*: *phase* (*Suc n*) = *phase n*
and *nxt*: *next1 n p* (*rho n p*) (*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send1 n*)) (*?crd n*) (*rho* (*Suc n*) *p*)
show *?Q p* (*Suc n*)
proof
assume *ts*: *?ts* (*Suc n*) = *Suc* (*phase* (*Suc n*))
from *run stp LV-timestamp-bounded*[**where** *n=n*] **have** *?ts n* ≤ *phase n* **by** *auto*
moreover
from *run stp* **have** *vote* (*rho* (*Suc n*) (*?crd* (*Suc n*))) = *vote* (*rho n* (*?crd n*))
by (*auto simp add*: *notStep3EqualCoord notStep0EqualVote*)
moreover
from *run stp* **have** *commt* (*rho* (*Suc n*) (*?crd* (*Suc n*))) = *commt* (*rho n* (*?crd n*))
by (*auto simp add*: *notStep3EqualCoord notStep03EqualCommit*)
moreover
note *ts nxt stp′ ph*
ultimately
show *?R* (*Suc n*)
by (*auto simp add*: *next1-def send1-def rcvdMsgs-def isVote-def*)
qed
next
— For step 2, the assertion follows from the induction hypothesis, observing that none of the relevant state components change.
fix *n*
assume *stp*: *step n* = *2* **and** *stp′*: *step* (*Suc n*) = *3*
and *ph*: *phase* (*Suc n*) = *phase n*
and *ih*: *?Q p n*
and *nxt*: *next2 n p* (*rho n p*) (*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send2 n*)) (*?crd n*) (*rho* (*Suc n*) *p*)
show *?Q p* (*Suc n*)
proof
assume *ts*: *?ts* (*Suc n*) = *Suc* (*phase* (*Suc n*))
from *run stp*
have *vt*: *vote* (*rho* (*Suc n*) (*?crd* (*Suc n*))) = *vote* (*rho n* (*?crd n*))
by (*auto simp add*: *notStep3EqualCoord notStep0EqualVote*)
from *run stp*
have *cmt*: *commt* (*rho* (*Suc n*) (*?crd* (*Suc n*))) = *commt* (*rho n* (*?crd n*))
by (*auto simp add*: *notStep3EqualCoord notStep03EqualCommit*)
with *vt ts ph stp stp′ ih nxt*

show *?R* (*Suc n*)
by (*auto simp add*: *next2-def*)
qed
next
— The assertion is trivially preserved by step 3 because the timestamp in the post-state cannot be current (cf. lemma *LV-timestamp-bounded*).
fix *n*
assume *stp'*: *step* (*Suc n*) = *0*
with *run LV-timestamp-bounded*[**where** *n*=*Suc n*] **have** *?ts* (*Suc n*) ≤ *phase* (*Suc n*)
by *auto*
thus *?Q p* (*Suc n*) **by** *simp*
qed
with *ts* **show** *?thesis* **by** (*intro conds*, *auto*)
qed

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

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

lemma *readyE*:
assumes *run*: *CHORun rho HOs coords* **and** *rdy*: *ready* (*rho r p*)
and *conds*: ⟦ *step r* = *3*; *coords r p* = *p*;
card { *q* . *coords r q* = *p* ∧ *timestamp* (*rho r q*) = *Suc* (*phase r*) } > *N div 2*
⟧ ⟹ *P*
shows *P*
proof −
let *?qs n* = { *q* . *coords n q* = *p* ∧ *timestamp* (*rho n q*) = *Suc* (*phase n*) }
have *ready* (*rho r p*) ⟶ *step r* = *3* ∧ *coords r p* = *p* ∧ *card* (*?qs r*) > *N div 2*
(**is** *?Q p r* **is** - ⟶ *?R p r*)
proof (*rule LV-induct'*[*OF run*, **where** *P*=*?Q*])
— the interesting case is step 2
fix *n*
assume *stp*: *step n* = *2* **and** *stp'*: *step* (*Suc n*) = *3*
and *ih*: *?Q p n* **and** *ph*: *phase* (*Suc n*) = *phase n*
and *nxt*: *next2 n p* (*rho n p*) (*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send2 n*)) (*coords n p*) (*rho* (*Suc n*) *p*)
show *?Q p* (*Suc n*)
proof
assume *rdy*: *ready* (*rho* (*Suc n*) *p*)
from *stp ih* **have** *nrdy*: ¬ *ready* (*rho n p*) **by** *simp*
with *rdy nxt* **have** *coords n p* = *p*
by (*auto simp add*: *next2-def*)
with *run stp* **have** *coord*: *coords* (*Suc n*) *p* = *p*
by (*simp add*: *notStep3EqualCoord*)
let *?acks* = *acksRcvd* (*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send2 n*))
from *nrdy rdy nxt* **have** *aRcvd*: *card ?acks* > *N div 2*
by (*auto simp add*: *next2-def*)
have *?acks* ⊆ *?qs* (*Suc n*)
proof (*clarify*)
fix *q*
assume *q*: *q* ∈ *?acks*
hence *n*: *coords n q* = *p* ∧ *timestamp* (*rho n q*) = *Suc* (*phase n*)

by (*auto simp add*: *acksRcvd-def rcvdMsgs-def send2-def isAck-def*)
with *run stp ph*
show *coords* (*Suc n*) *q* = *p* ∧ *timestamp* (*rho* (*Suc n*) *q*) = *Suc* (*phase* (*Suc n*))
by (*simp add*: *notStep3EqualCoord notStep1EqualTimestamp*)
qed
hence *card ?acks* ≤ *card* (*?qs* (*Suc n*))
by (*intro card-mono*, *auto*)
with *stp' coord aRcvd* **show** *?R p* (*Suc n*)
by *auto*
qed
— the remaining steps are all solved trivially
qed (*auto simp add*: *initState-def next0-def next1-def next3-def*)
with *rdy* **show** *?thesis* **by** (*blast intro*: *conds*)
qed

A process decides only if the following conditions hold:

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

This is (essentially) Bernadette's Lemma 3.

lemma *decisionE*:
assumes *run*: *CHORun rho HOs coords*
and *dec*: *decide* (*rho* (*Suc r*) *p*) ≠ *decide* (*rho r p*)
and *conds*: ⟦ *step r* = *3*;
decide (*rho* (*Suc r*) *p*) = *Some* (*the* (*vote* (*rho r* (*coords r p*))));
ready (*rho r* (*coords r p*)); *commt* (*rho r* (*coords r p*))
⟧ ⟹ *P*
shows *P*
proof −
let *?cfg* = *rho r*
let *?cfg'* = *rho* (*Suc r*)
let *?crd* = *coords r*
let *?dec'* = *decide* (*?cfg' p*)
— Except for the assertion about the *commt* field, the assertion can be proved directly from the next-state relation.
have *1*: *step r* = *3* ∧ *?dec'* = *Some* (*the* (*vote* (*?cfg* (*?crd p*)))) ∧ *ready* (*?cfg* (*?crd p*))
(**is** *?Q p r*)
proof (*rule LV-Suc'*[*OF run*, **where** *P*=*?Q*])
— for step 3, we prove the thesis by expanding the relevant definitions
assume *next3 r p* (*?cfg p*) (*rcvdMsgs p* (*HOs r p*) *?crd ?cfg* (*send3 r*)) (*?crd p*) (*?cfg' p*)
and *step r* = *3*
with *dec* **show** *?thesis*
by (*auto simp add*: *next3-def send3-def isVote-def rcvdMsgs-def*)
next
— for the other steps, the proof is by contradiction because they don't change the decision
assume *next0 r p* (*?cfg p*) (*rcvdMsgs p* (*HOs r p*) *?crd ?cfg* (*send0 r*)) (*?crd p*) (*?cfg' p*)
with *dec* **show** *?thesis* **by** (*auto simp add*: *next0-def*)
next
assume *next1 r p* (*?cfg p*) (*rcvdMsgs p* (*HOs r p*) *?crd ?cfg* (*send1 r*)) (*?crd p*) (*?cfg' p*)
with *dec* **show** *?thesis* **by** (*auto simp add*: *next1-def*)
next
assume *next2 r p* (*?cfg p*) (*rcvdMsgs p* (*HOs r p*) *?crd ?cfg* (*send2 r*)) (*?crd p*) (*?cfg' p*)

with *dec* **show** *?thesis* **by** (*auto simp add*: *next2-def*)
qed
hence *ready* (*?cfg* (*?crd p*)) **by** *blast*
— Because the coordinator is ready, there is a majority of processes that consider it to be the coordinator and that have a current timestamp.
with *run*
have *card* {*q* . *?crd q* = *?crd p* ∧ *timestamp* (*?cfg q*) = *Suc* (*phase r*)} > *N div 2*
by (*rule readyE*)
— Hence there is at least one such process . . .
hence *card* {*q* . *?crd q* = *?crd p* ∧ *timestamp* (*?cfg q*) = *Suc* (*phase r*)} ≠ *0*
by *arith*
then obtain *q* **where** *?crd q* = *?crd p* **and** *timestamp* (*?cfg q*) = *Suc* (*phase r*)
by *auto*
— . . . and by a previous lemma the coordinator must have committed.
with *run* **have** *commt* (*?cfg* (*?crd p*))
by (*auto elim*: *currentTimestampE*)
with *1* **show** *?thesis* **by** (*blast intro*: *conds*)
qed

2.5 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 *integrityInvariant*:
assumes *run*: *CHORun rho HOs coords*
and *inv*: ⟦ *range* (*x* ∘ (*rho n*)) ⊆ *range* (*x* ∘ (*rho 0*));
range (*vote* ∘ (*rho n*)) ⊆ {*None*} ∪ *Some* ‘ *range* (*x* ∘ (*rho 0*));
range (*decide* ∘ (*rho n*)) ⊆ {*None*} ∪ *Some* ‘ *range* (*x* ∘ (*rho 0*))
⟧ ⟹ *A*
shows *A*
proof −
let *?x0* = *range* (*x* ∘ *rho 0*)
let *?x0opt* = {*None*} ∪ *Some* ‘ *?x0*
have *range* (*x* ∘ *rho n*) ⊆ *?x0* ∧
range (*vote* ∘ *rho n*) ⊆ *?x0opt* ∧
range (*decide* ∘ *rho n*) ⊆ *?x0opt* (**is** *?Inv n* **is** *?X n* ∧ *?Vote n* ∧ *?Decide n*)
proof (*induct n*)
from *run* **show** *?Inv 0*
by (*auto simp add*: *CHORun-def initConfig-def initState-def*)
next
fix *n*
assume *ih*: *?Inv n* **thus** *?Inv* (*Suc n*)
proof (*clarify*)
assume *x*: *?X n* **and** *vt*: *?Vote n* **and** *dec*: *?Decide n*

Proof of first conjunct

have *x'*: *?X* (*Suc n*)
proof (*clarsimp*)
fix *p*
from *run* **show** *x* (*rho* (*Suc n*) *p*) ∈ *range* (λ*q*. *x* (*rho 0 q*)) (**is** *?P p n*)
proof (*rule LV-Suc'*[**where** *P*=*?P*])
— only *step1* is of interest
assume *nxt*: *next1 n p* (*rho n p*)
(*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send1 n*))
(*coords n p*) (*rho* (*Suc n*) *p*)

show *?thesis*
proof (*cases rho* (*Suc n*) *p* = *rho n p*)
case *True*
with *x* **show** *?thesis* **by** *auto*
next
case *False*
with *nxt* **have** *cmt*: *commt* (*rho n* (*coords n p*))
and *xp*: *x* (*rho* (*Suc n*) *p*) = *the* (*vote* (*rho n* (*coords n p*)))
by (*auto simp add*: *next1-def send1-def rcvdMsgs-def isVote-def*)
from *run cmt* **have** *vote* (*rho n* (*coords n p*)) ≠ *None*
by (*rule commitE*)
moreover
from *vt* **have** *vote* (*rho n* (*coords n p*)) ∈ *?x0opt*
by (*auto simp add*: *image-def*)
moreover
note *xp*
ultimately
show *?thesis* **by** (*force simp add*: *image-def*)
qed
— the other steps don't change *x* and therefore follow from the induction hypothesis
next
assume *step n* = *0*
with *run* **have** *x* (*rho* (*Suc n*) *p*) = *x* (*rho n p*) **by** (*simp add*: *notStep1EqualX*)
with *x* **show** *?thesis* **by** *auto*
next
assume *step n* = *2*
with *run* **have** *x* (*rho* (*Suc n*) *p*) = *x* (*rho n p*) **by** (*simp add*: *notStep1EqualX*)
with *x* **show** *?thesis* **by** *auto*
next
assume *step n* = *3*
with *run* **have** *x* (*rho* (*Suc n*) *p*) = *x* (*rho n p*) **by** (*simp add*: *notStep1EqualX*)
with *x* **show** *?thesis* **by** *auto*
qed
qed

Proof of second conjunct

have *vt′*: *?Vote* (*Suc n*)
proof (*clarsimp simp add*: *image-def*)
fix *p v*
assume *v*: *vote* (*rho* (*Suc n*) *p*) = *Some v*
from *run* **have** *vote* (*rho* (*Suc n*) *p*) = *Some v* ⟶ *v* ∈ *?x0* (**is** *?P p n*)
proof (*rule LV-Suc′*[**where** *P*=*?P*])
— here only *step0* is of interest
assume *nxt*: *next0 n p* (*rho n p*)
(*rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send0 n*))
(*coords n p*) (*rho* (*Suc n*) *p*)
show *?thesis*
proof (*cases rho* (*Suc n*) *p* = *rho n p*)
case *True*
from *vt* **have** *vote* (*rho n p*) ∈ *?x0opt* **by** (*auto simp add*: *image-def*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
from *nxt False v* **obtain** *q* **where** *v* = *x* (*rho n q*)
by (*auto simp add*: *next0-def send0-def rcvdMsgs-def*)
with *x* **show** *?thesis* **by** (*auto simp add*: *image-def*)

```
        qed
        — the other cases don't change the vote and therefore follow from the induction hypothesis
      next
        assume step n = 1
        with run have vote (rho (Suc n) p) = vote (rho n p)
          by (simp add: notStep0EqualVote)
        moreover
        from vt have vote (rho n p) ∈ ?x0opt by (auto simp add: image-def)
        ultimately
        show ?thesis by auto
      next
        assume step n = 2
        with run have vote (rho (Suc n) p) = vote (rho n p)
          by (simp add: notStep0EqualVote)
        moreover
        from vt have vote (rho n p) ∈ ?x0opt by (auto simp add: image-def)
        ultimately
        show ?thesis by auto
      next
        assume step n = 3
        with run have vote (rho (Suc n) p) = vote (rho n p)
          by (simp add: notStep0EqualVote)
        moreover
        from vt have vote (rho n p) ∈ ?x0opt by (auto simp add: image-def)
        ultimately
        show ?thesis by auto
      qed
      with v show ∃ q. v = x (rho 0 q) by auto
    qed
```

Proof of third conjunct

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

qed

The Integrity theorem follows as an easy consequence.

theorem *integrity*:
  assumes *run*: *CHORun rho HOs coords* **and** *dec*: *decide* (*rho n p*) = *Some v*
  shows $\exists$ *q*. *v* = *x* (*rho 0 q*)
proof −
  from *run* **have** *decide* (*rho n p*) $\in$ {*None*} $\cup$ *Some* ‘ (*range* (*x* $\circ$ (*rho 0*)))
    by (*rule integrityInvariant*, *auto simp add*: *image-def*)
  with *dec* **show** *?thesis* **by** (*auto simp add*: *image-def*)
qed

2.6 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 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*
  using *run dec* **proof** (*rule decisionE*)
  — Lemma *decisionE* tells us that we are at step 3 and that the coordinator is ready.
  let *?crd* = *coords r p*
  let *?qs* = { *q* . *coords r q* = *?crd* $\wedge$ *timestamp* (*rho r q*) = *Suc* (*phase r*) }
  assume *stp*: *step r* = *3* **and** *rdy*: *ready* (*rho r ?crd*)
  — Now, lemma *readyE* implies that a majority of processes have a recent timestamp.
  from *run rdy* **have** *card ?qs* > *N div 2* **by** (*rule readyE*)
  moreover
  from *stp LV-timestamp-bounded*[*OF run*, **where** *n*=*r*]
  have $\forall$ *q*. *timestamp* (*rho r q*) $\leq$ *Suc* (*phase r*) **by** *auto*
  hence *?qs* $\subseteq$ *procsBeyondTS* (*Suc* (*phase r*)) (*rho r*)
    by (*auto simp add*: *procsBeyondTS-def*)
  hence *card ?qs* $\leq$ *card* (*procsBeyondTS* (*Suc* (*phase r*)) (*rho r*))
    by (*intro card-mono*, *auto*)
  ultimately show *?thesis* **by** *simp*
qed

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

lemma *committedProcsEqual*:
  assumes *run*: *CHORun rho HOs coords*
  and *cmt*: *commt* (*rho r p*) **and** *cmt'*: *commt* (*rho r p'*)
  shows *p* = *p'*
proof −
  from *run cmt* **have** *card* {*q* . *coords r q* = *p*} > *N div 2* **by** (*blast elim*: *commitE*)
  moreover
  from *run cmt'* **have** *card* {*q* . *coords r q* = *p'*} > *N div 2* **by** (*blast elim*: *commitE*)
  ultimately
  obtain *q* **where** *coords r q* = *p* **and** *p'* = *coords r q* **by** (*auto elim*: *majoritiesE'*)
  thus *?thesis* **by** *simp*
qed

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

lemma *readyProcsEqual*:

assumes *run*: *CHORun rho HOs coords*
and *rdy*: *ready* (*rho r p*) **and** *rdy'*: *ready* (*rho r p'*)
shows $p = p'$
proof −
let *?C p* = {*q* . *coords r q* = *p* ∧ *timestamp* (*rho r q*) = *Suc* (*phase r*)}
from *run rdy* **have** *card* (*?C p*) > *N div 2* **by** (*blast elim*: *readyE*)
moreover
from *run rdy'* **have** *card* (*?C p'*) > *N div 2* **by** (*blast elim*: *readyE*)
ultimately
obtain *q* **where** *coords r q* = *p* **and** *p'* = *coords r q* **by** (*auto elim*: *majoritiesE'*)
thus *?thesis* **by** *simp*
qed

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

lemma *commitThenVoteRecent*:
assumes *run*: *CHORun rho HOs coords*
and *maj*: *card* (*procsBeyondTS ts* (*rho r*)) > *N div 2* **and** *cmt*: *commt* (*rho r p*)
shows ∃ *q* ∈ *procsBeyondTS ts* (*rho r*). *vote* (*rho r p*) = *Some* (*x* (*rho r q*))
(**is** *?Q r*)
proof −
let *?bynd n* = *procsBeyondTS ts* (*rho n*)
have *card* (*?bynd r*) > *N div 2* ∧ *commt* (*rho r p*) ⟶ *?Q r*
(**is** *?P p r*)
proof (*rule LV-induct*[*OF run*])

next0 establishes the property

fix *n*
assume *stp*: *step n* = *0*
and *nxt*: ∀ *q*. *next0 n q* (*rho n q*) (*rcvdMsgs q* (*HOs n q*) (*coords n*) (*rho n*) (*send0 n*)) (*coords n q*) (*rho* (*Suc n*) *q*) (**is** ∀ *q*. *?nxt q*)
from *nxt* **have** *nxp*: *?nxt p* **..**
show *?P p* (*Suc n*)
proof (*clarify*)
assume *mj*: *card* (*?bynd* (*Suc n*)) > *N div 2* **and** *ct*: *commt* (*rho* (*Suc n*) *p*)
show *?Q* (*Suc n*)
proof −
let *?msgs* = *rcvdMsgs p* (*HOs n p*) (*coords n*) (*rho n*) (*send0 n*)
from *stp run* **have** ¬ *commt* (*rho n p*) **by** (*auto elim*: *commitE*)
with *nxp ct* **obtain** *q v* **where**
v: *?msgs q* = *Some* (*ValStamp v* (*highestStampRcvd ?msgs*)) **and**
vote: *vote* (*rho* (*Suc n*) *p*) = *Some v* **and**
rcvd: *card* (*valStampsRcvd ?msgs*) > *N div 2*
by (*auto simp add*: *next0-def*)
from *mj rcvd* **obtain** *q'* **where**
q1': *q'* ∈ *?bynd* (*Suc n*) **and** *q2'*: *q'* ∈ *valStampsRcvd ?msgs*
by (*rule majoritiesE'*)
have *timestamp* (*rho n q'*) ≤ *timestamp* (*rho n q*)
proof −
from *q2'* **obtain** *v' ts'* **where** *ts'*: *?msgs q'* = *Some* (*ValStamp v' ts'*)
by (*auto simp add*: *valStampsRcvd-def*)
hence *ts'* ≤ *highestStampRcvd ?msgs*
by (*rule highestStampRcvd-max*)
moreover

from *ts'* **have** *timestamp* (*rho n q'*) = *ts'*
by (*auto simp add*: *rcvdMsgs-def send0-def*)
moreover
from *v* **have** *timestamp* (*rho n q*) = *highestStampRcvd ?msgs*
by (*auto simp add*: *rcvdMsgs-def send0-def*)
ultimately
show *?thesis*
by *simp*
qed
moreover
from *run stp* **have** *timestamp* (*rho* (*Suc n*) *q'*) = *timestamp* (*rho n q'*)
by (*simp add*: *notStep1EqualTimestamp*)
moreover
from *run stp* **have** *timestamp* (*rho* (*Suc n*) *q*) = *timestamp* (*rho n q*)
by (*simp add*: *notStep1EqualTimestamp*)
moreover
note *q1'*
ultimately
have *q* ∈ *?bynd* (*Suc n*)
by (*simp add*: *procsBeyondTS-def*)
moreover
from *v vote* **have** *vote* (*rho* (*Suc n*) *p*) = *Some* (*x* (*rho n q*))
by (*auto simp add*: *rcvdMsgs-def send0-def split*: *split-if-asm*)
moreover
from *run stp* **have** *x* (*rho* (*Suc n*) *q*) = *x* (*rho n q*)
by (*simp add*: *notStep1EqualX*)
ultimately
show *?thesis* **by** *force*
qed
qed

next

We now prove that *next1* preserves the property. Observe that *next1* may establish a majority of processes with current timestamps, so we cannot just refer to the induction hypothesis. However, if that happens, there is at least one process with a fresh timestamp that copies the vote of the (only) committed coordinator, thus establishing the property.

fix *n*
assume *stp*: *step n* = *1*
and *nxt*: ∀ *q*. *next1 n q* (*rho n q*) (*rcvdMsgs q* (*HOs n q*) (*coords n*) (*rho n*) (*send1 n*)) (*coords n q*) (*rho* (*Suc n*) *q*) (**is** ∀ *q*. *?nxt q*)
and *ih*: *?P p n*
from *nxt* **have** *nxp*: *?nxt p* **..**
show *?P p* (*Suc n*)
proof (*clarify*)
assume *mj'*: *card* (*?bynd* (*Suc n*)) > *N div 2* **and** *ct'*: *commt* (*rho* (*Suc n*) *p*)
from *run stp ct'* **have** *ct*: *commt* (*rho n p*)
by (*simp add*: *notStep03EqualCommit*)
from *run stp* **have** *vote'*: *vote* (*rho* (*Suc n*) *p*) = *vote* (*rho n p*)
by (*simp add*: *notStep0EqualVote*)
show *?Q* (*Suc n*)
proof (*cases* ∃ *q* ∈ *?bynd* (*Suc n*). *rho* (*Suc n*) *q* ≠ *rho n q*)
case *True*
— in this case the property holds because *q* updates its *x* field to the vote
then obtain *q* **where** *q1*: *q* ∈ *?bynd* (*Suc n*) **and** *q2*: *rho* (*Suc n*) *q* ≠ *rho n q* **..**
from *nxt* **have** *?nxt q* **..**

with *q2*
have *x'*: *x* (*rho* (*Suc n*) *q*) = *the* (*vote* (*rho n* (*coords n q*)))
and *coord*: *commt* (*rho n* (*coords n q*))
by (*auto simp add*: *next1-def send1-def rcvdMsgs-def isVote-def*)
from *run ct* **have** *vote*: *vote* (*rho n p*) ≠ *None* **by** (*rule commitE*)
from *run coord ct* **have** *coords n q* = *p* **by** (*rule committedProcsEqual*)
with *q1 x' vote vote'* **show** *?thesis* **by** *auto*
next
case *False*
— if no relevant process moves then *procsBeyondTS* doesn't change and we invoke the induction hypothesis
hence *bynd*: *?bynd* (*Suc n*) = *?bynd n*
proof (*auto simp add*: *procsBeyondTS-def*)
fix *r*
assume *ts*: *ts* ≤ *timestamp* (*rho n r*)
from *run* **have** *timestamp* (*rho n r*) ≤ *timestamp* (*rho* (*Suc n*) *r*)
by (*simp add*: *LV-timestamp-monotonic*)
with *ts* **show** *ts* ≤ *timestamp* (*rho* (*Suc n*) *r*) **by** *simp*
qed
with *mj'* **have** *mj*: *card* (*?bynd n*) > *N div 2* **by** *simp*
with *ct ih* **obtain** *q* **where**
q ∈ *?bynd n* **and** *vote* (*rho n p*) = *Some* (*x* (*rho n q*))
by *blast*
with *vote' bynd False* **show** *?thesis* **by** *auto*
qed
qed

next

step2 preserves the property, via the induction hypothesis.

fix *n*
assume *stp*: *step n* = *2*
and *nxt*: ∀ *q*. *next2 n q* (*rho n q*) (*rcvdMsgs q* (*HOs n q*) (*coords n*) (*rho n*) (*send2 n*)) (*coords n q*) (*rho* (*Suc n*) *q*) (**is** ∀ *q*. *?nxt q*)
and *ih*: *?P p n*
from *nxt* **have** *nxp*: *?nxt p* **..**
show *?P p* (*Suc n*)
proof (*clarify*)
assume *mj'*: *card* (*?bynd* (*Suc n*)) > *N div 2* **and** *ct'*: *commt* (*rho* (*Suc n*) *p*)
from *run stp ct'* **have** *ct*: *commt* (*rho n p*)
by (*simp add*: *notStep03EqualCommit*)
from *run stp* **have** *vote'*: *vote* (*rho* (*Suc n*) *p*) = *vote* (*rho n p*)
by (*simp add*: *notStep0EqualVote*)
from *run stp* **have** ∀ *q*. *timestamp* (*rho* (*Suc n*) *q*) = *timestamp* (*rho n q*)
by (*simp add*: *notStep1EqualTimestamp*)
hence *bynd'*: *?bynd* (*Suc n*) = *?bynd n*
by (*auto simp add*: *procsBeyondTS-def*)
from *run stp* **have** ∀ *q*. *x* (*rho* (*Suc n*) *q*) = *x* (*rho n q*)
by (*simp add*: *notStep1EqualX*)
with *bynd' vote' ct mj' ih* **show** *?Q* (*Suc n*)
by *auto*
qed

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

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

qed

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

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

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

```
    from run dec
    have stp: step r = 3
      and v: decide (rho (Suc r) p) = Some (the (vote (rho r (coords r p))))
      and cmt: commt (rho r (coords r p))
      by (auto elim: decisionE)
    from stp LV-timestamp-bounded[OF run, where n=r]
    have timestamp (rho r q) ≤ Suc (phase r) by simp
    with q have timestamp (rho r q) = Suc (phase r)
      by (simp add: procsBeyondTS-def)
    with run
    have x: x (rho r q) = the (vote (rho r (coords r q)))
      and cmt': commt (rho r (coords r q))
      by (auto elim: currentTimestampE)
    from run cmt cmt' have coords r p = coords r q by (rule committedProcsEqual)
    with x v show x (rho (r+0) q) = ?v by simp
  qed
  next
  — induction step
  fix k
  assume ih: ?P p k
  show ?P p (Suc k)
  proof (clarify)
    fix q
    assume q: q ∈ ?bynd (Suc k)
    — distinguish the kind of transition—only step1 is interesting
    have x (rho (Suc (r + k)) q) = ?v (is ?X q (r+k))
    proof (rule LV-Suc'[OF run, where P=?X])
      fix HO
      assume stp: step (r + k) = 1
      and nxt: next1 (r+k) q (rho (r+k) q)
                  (rcvdMsgs q (HOs (r+k) q) (coords (r+k)) (rho (r+k)) (send1 (r+k)))
                  (coords (r+k) q) (rho (Suc (r+k)) q)
      show ?thesis
      proof (cases rho (Suc (r+k)) q = rho (r+k) q)
        case True
        with q ih show ?thesis by (auto simp add: procsBeyondTS-def)
```

next
case *False*
from *run dec* **have** *card (?bynd 0) > N div 2*
by (*simp add*: *decisionThenMajorityBeyondTS*)
moreover
have *?bynd 0 ⊆ ?bynd k*
by (*auto elim*: *procsBeyondTS-monotonic*[*OF run*])
hence *card (?bynd 0) ≤ card (?bynd k)*
by (*auto intro*: *card-mono*)
ultimately
have *maj*: *card (?bynd k) > N div 2* **by** *simp*
let *?crd = coords (r+k) q*
from *False nxt* **have**
cmt: *commt (rho (r+k) ?crd)* **and**
x: *x (rho (Suc (r+k)) q) = the (vote (rho (r+k) ?crd))*
by (*auto simp add*: *next1-def rcvdMsgs-def send1-def isVote-def*)
from *run maj cmt stp* **obtain** *q'*
where *q1'*: *q' ∈ ?bynd k* **and** *q2'*: *vote (rho (r+k) ?crd) = Some (x (rho (r+k) q'))*
by (*blast dest*: *commitThenVoteRecent*)
with *x ih* **show** *?thesis* **by** *auto*
qed
next
— all other steps hold by induction hypothesis
assume *step (r+k) = 0*
with *run* **have** *x*: *x (rho (Suc (r+k)) q) = x (rho (r+k) q)*
and *ts*: *timestamp (rho (Suc (r+k)) q) = timestamp (rho (r+k) q)*
by (*auto simp add*: *notStep1EqualX notStep1EqualTimestamp*)
from *ts q* **have** *q ∈ ?bynd k*
by (*auto simp add*: *procsBeyondTS-def*)
with *x ih* **show** *?thesis* **by** *auto*
next
assume *step (r+k) = 2*
with *run* **have** *x*: *x (rho (Suc (r+k)) q) = x (rho (r+k) q)*
and *ts*: *timestamp (rho (Suc (r+k)) q) = timestamp (rho (r+k) q)*
by (*auto simp add*: *notStep1EqualX notStep1EqualTimestamp*)
from *ts q* **have** *q ∈ ?bynd k*
by (*auto simp add*: *procsBeyondTS-def*)
with *x ih* **show** *?thesis* **by** *auto*
next
assume *step (r+k) = 3*
with *run* **have** *x*: *x (rho (Suc (r+k)) q) = x (rho (r+k) q)*
and *ts*: *timestamp (rho (Suc (r+k)) q) = timestamp (rho (r+k) q)*
by (*auto simp add*: *notStep1EqualX notStep1EqualTimestamp*)
from *ts q* **have** *q ∈ ?bynd k*
by (*auto simp add*: *procsBeyondTS-def*)
with *x ih* **show** *?thesis* **by** *auto*
qed
thus *x (rho (r + Suc k) q) = ?v* **by** *simp*
qed
qed

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

lemma *laterProcessDecidesSameValue*:
assumes *run*: *CHORun rho HOs coords*

and p: *decide* (*rho* (*Suc* r) p) $\neq$ *decide* (*rho* r p)
and q: *decide* (*rho* (*Suc* ($r+k$)) q) $\neq$ *decide* (*rho* ($r+k$) q)
shows *decide* (*rho* (*Suc* ($r+k$)) q) = *decide* (*rho* (*Suc* r) p)
proof −
let *?bynd k* = *procsBeyondTS* (*Suc* (*phase* r)) (*rho* ($r+k$))
let *?qcrd* = *coords* ($r+k$) q
from *run p* **have** *notNone*: *decide* (*rho* (*Suc* r) p) $\neq$ *None*
by (*auto elim*: *decisionE*)
— process q decides on the vote of its coordinator
from *run q* **have** *dec*: *decide* (*rho* (*Suc* ($r+k$)) q) = *Some* (*the* (*vote* (*rho* ($r+k$) *?qcrd*)))
and *cmt*: *commt* (*rho* ($r+k$) *?qcrd*)
by (*auto elim*: *decisionE*)
— that vote is the x field of some process q' with a recent timestamp
from *run p* **have** *card* (*?bynd 0*) > *N div 2*
by (*simp add*: *decisionThenMajorityBeyondTS*)
moreover
from *run* **have** *?bynd 0* $\subseteq$ *?bynd k* **by** (*auto elim*: *procsBeyondTS-monotonic*)
hence *card* (*?bynd 0*) $\leq$ *card* (*?bynd k*) **by** (*auto intro*: *card-mono*)
ultimately
have *maj*: *card* (*?bynd k*) > *N div 2* **by** *simp*
from *run maj cmt* **obtain** q' **where**
q'1: $q' \in$ *?bynd k* **and** *q'2*: *vote* (*rho* ($r+k$) *?qcrd*) = *Some* (x (*rho* ($r+k$) q'))
by (*auto dest*: *commitThenVoteRecent*)
— the x field of process q' is the value p decided on
from *run p q'1* **have** x (*rho* ($r+k$) q') = *the* (*decide* (*rho* (*Suc* r) p))
by (*auto dest*: *XOfTimestampBeyondDecision*)
— which proves the assertion
with *dec q'2 notNone* **show** *?thesis* **by** *auto*
qed

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

lemma *decisionNonNullThenDecided*:
assumes *run*: *CHORun rho HOs coords* **and** *dec*: *decide* (*rho n p*) = *Some v*
shows $\exists\, m<n$. *decide* (*rho* (*Suc m*) *p*) $\neq$ *decide* (*rho m p*)
$\wedge$ *decide* (*rho* (*Suc m*) *p*) = *Some v*
proof −
let *?dec k* = *decide* (*rho k p*)
have ($\forall\, m<n$. *?dec* (*Suc m*) $\neq$ *?dec m* $\longrightarrow$ *?dec* (*Suc m*) $\neq$ *Some v*) $\longrightarrow$ *?dec n* $\neq$ *Some v*
(**is** *?P n* **is** *?A n* $\longrightarrow$ -)
proof (*induct n*)
from *run* **show** *?P 0* **by** (*auto simp add*: *CHORun-def initConfig-def initState-def*)
next
fix *n*
assume *ih*: *?P n*
show *?P* (*Suc n*)
proof (*clarify*)
assume *p*: *?A* (*Suc n*) **and** *v*: *?dec* (*Suc n*) = *Some v*
from *p* **have** *?A n* **by** *simp*
with *ih* **have** *?dec n* $\neq$ *Some v* **by** *simp*
moreover
from *p* **have** *?dec* (*Suc n*) $\neq$ *?dec n* $\longrightarrow$ *?dec* (*Suc n*) $\neq$ *Some v* **by** *simp*
ultimately
have *?dec* (*Suc n*) $\neq$ *Some v* **by** *auto*
with *v* **show** *False* **by** *simp*
qed

```
  qed
  with dec show ?thesis by auto
qed
```

Irrevocability and Agreement follow as easy consequences.

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

```
theorem agreement:
  assumes run: CHORun rho HOs coords
  and p: decide (rho m p) = Some v and q: decide (rho n q) = Some w
  shows v = w
proof –
  from run p obtain k where
    k1: decide (rho (Suc k) p) ≠ decide (rho k p) and k2: decide (rho (Suc k) p) = Some v
    by (auto dest: decisionNonNullThenDecided)
  from run q obtain l where
    l1: decide (rho (Suc l) q) ≠ decide (rho l q) and l2: decide (rho (Suc l) q) = Some w
    by (auto dest: decisionNonNullThenDecided)
  show ?thesis
```

proof (*cases* $k \le l$)
  case *True*
  then obtain *m* **where** *m*: $l = k+m$ **by** (*auto simp add*: *le-iff-add*)
  from *run k1 l1 m* **have** *decide* (*rho* (*Suc l*) *q*) = *decide* (*rho* (*Suc k*) *p*)
    by (*auto elim*: *laterProcessDecidesSameValue*)
  with *k2 l2* **show** *?thesis* **by** *simp*
next
  case *False*
  hence $l \le k$ **by** *simp*
  then obtain *m* **where** *m*: $k = l+m$ **by** (*auto simp add*: *le-iff-add*)
  from *run l1 k1 m* **have** *decide* (*rho* (*Suc k*) *p*) = *decide* (*rho* (*Suc l*) *q*)
    by (*auto elim*: *laterProcessDecidesSameValue*)
  with *l2 k2* **show** *?thesis* **by** *simp*
qed
qed

2.7 Proof of liveness

We now show that the communication predicate ensures termination of the algorithm: there exists some round r at which all processes have decided. In fact, the assumption ensures the existence of some phase during which there is a single coordinator that receives a majority of messages. Moreover, all processes receive the messages sent by the coordinator and therefore successfully execute the protocol, deciding at step 3 of that phase.

theorem *decision*:
  assumes *run*: *CHORun rho HOs coords*
  shows ∃ *r*. ∀ *p*. *decide* (*rho r p*) ≠ *None*
proof −

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

  from *run* **obtain** *ph c*
    where *c*: ∀ *p*. *coords* (*4∗ph*) *p* = *c*
    and *maj0*: *card* (*HOs* (*4∗ph*) *c*) > *N div 2*
    and *maj2*: *card* (*HOs* (*Suc* (*Suc* (*4∗ph*))) *c*) > *N div 2*
    and *rcv1*: ∀ *p*. *c* ∈ (*HOs* (*Suc* (*4∗ph*)) *p*)
    and *rcv3*: ∀ *p*. *c* ∈ (*HOs* (*Suc* (*Suc* (*Suc* (*4∗ph*)))) *p*)
    by (*auto simp add*: *CHORun-def LV-commLive-def*)
  let *?r* = *4∗ph*
  let *?r1* = *Suc ?r*
  let *?r2* = *Suc* (*Suc ?r*)
  let *?r3* = *Suc* (*Suc* (*Suc ?r*))
  let *?r4* = *Suc* (*Suc* (*Suc* (*Suc ?r*)))

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

  from *run c* **have** *c1*: ∀ *p*. *coords ?r1 p* = *c*
    by (*auto simp add*: *step-def notStep3EqualCoord*)
  with *run* **have** *c2*: ∀ *p*. *coords ?r2 p* = *c*
    by (*auto simp add*: *step-def mod-Suc notStep3EqualCoord*)
  with *run* **have** *c3*: ∀ *p*. *coords ?r3 p* = *c*
    by (*auto simp add*: *step-def mod-Suc notStep3EqualCoord*)

The coordinator receives *ValStamp* messages from a majority of processes at step 0 of phase *ph* and therefore commits during the transition at the end of step 0.

  have *1*: *commt* (*rho ?r1 c*) (**is** *?P c* (*4∗ph*))

proof (*rule LV-Suc'*[*OF run*, **where** *P*=*?P*], *auto simp add*: *step-def*)
assume *next0 ?r c* (*rho ?r c*) (*rcvdMsgs c* (*HOs ?r c*) (*coords ?r*) (*rho ?r*) (*send0 ?r*))
(*coords ?r c*) (*rho* (*Suc ?r*) *c*)
with *c maj0* **show** *commt* (*rho* (*Suc ?r*) *c*)
by (*auto simp add*: *next0-def send0-def valStampsRcvd-def rcvdMsgs-def*)
qed

All processes receive the vote of *c* at step 1 and therefore update their time stamps during the transition at the end of step 1.

have *2*: ∀ *p*. *timestamp* (*rho ?r2 p*) = *Suc ph*
proof
fix *p*
let *?msgs* = *rcvdMsgs p* (*HOs ?r1 p*) (*coords ?r1*) (*rho ?r1*) (*send1 ?r1*)
let *?crd* = *coords ?r1 p*
from *run 1 c1 rcv1* **have** *cnd*: *?msgs ?crd* ≠ *None* ∧ *isVote* (*the* (*?msgs ?crd*))
by (*auto elim*: *commitE simp add*: *rcvdMsgs-def send1-def isVote-def*)
show *timestamp* (*rho ?r2 p*) = *Suc ph* (**is** *?P p* (*Suc* (*4∗ph*)))
proof (*rule LV-Suc'*[*OF run*, **where** *P*=*?P*], *auto simp add*: *step-def mod-Suc*)
assume *next1 ?r1 p* (*rho ?r1 p*) *?msgs ?crd* (*rho ?r2 p*)
with *cnd* **show** *?thesis*
by (*auto simp add*: *next1-def phase-def*)
qed
qed

The coordinator receives acknowledgements from a majority of processes at step 2 and sets its *ready* flag during the transition at the end of step 2.

have *3*: *ready* (*rho ?r3 c*) (**is** *?P c* (*Suc* (*Suc* (*4∗ph*))))
proof (*rule LV-Suc'*[*OF run*, **where** *P*=*?P*], *auto simp add*: *step-def mod-Suc*)
assume *next2 ?r2 c* (*rho ?r2 c*) (*rcvdMsgs c* (*HOs ?r2 c*) (*coords ?r2*) (*rho ?r2*) (*send2 ?r2*))
(*coords ?r2 c*) (*rho ?r3 c*)
with *2 c2 maj2* **show** *?thesis*
by (*auto simp add*: *next2-def send2-def rcvdMsgs-def acksRcvd-def isAck-def phase-def*)
qed

All processes receive the vote of the coordinator during step 3 and decide during the transition at the end of that step.

have *4*: ∀ *p*. *decide* (*rho ?r4 p*) ≠ *None*
proof
fix *p*
let *?msgs* = *rcvdMsgs p* (*HOs ?r3 p*) (*coords ?r3*) (*rho ?r3*) (*send3 ?r3*)
let *?crd* = *coords ?r3 p*
from *run 3 c3 rcv3* **have** *cnd*: *?msgs ?crd* ≠ *None* ∧ *isVote* (*the* (*?msgs ?crd*))
by (*auto elim*: *readyE simp add*: *rcvdMsgs-def send3-def isVote-def*)
show *decide* (*rho ?r4 p*) ≠ *None* (**is** *?P p* (*Suc* (*Suc* (*Suc* (*4∗ph*)))))
proof (*rule LV-Suc'*[*OF run*, **where** *P*=*?P*], *auto simp add*: *step-def mod-Suc*)
assume *next3 ?r3 p* (*rho ?r3 p*) *?msgs ?crd* (*rho ?r4 p*)
with *cnd* **show** ∃ *v*. *decide* (*rho ?r4 p*) = *Some v*
by (*auto simp add*: *next3-def*)
qed
qed

This immediately proves the assertion.

from *4* **show** *?thesis* **..**
qed

end

References

[1] B. Charron-Bost and A. Schiper: *The Heard-Of Model: Computing in Distributed Systems with Benign Failures*. LSR-Report 2007-001, EPFL, Lausanne, 2007.

[2] L. Lamport: *The Part-Time Parliament*. ACM Trans. Comput. Syst. 16(2):133–169, 1998.