Abstract

Algorithms for solving the consensus problem are fundamental to distributed computing. Despite their brevity, their ability to operate in concurrent, asynchronous and failure-prone environments comes at the cost of complex and subtle behaviors. Accordingly, understanding how they work and proving their correctness is a non-trivial endeavor where abstraction is immensely helpful. Moreover, research on consensus has yielded a large number of algorithms, many of which appear to share common algorithmic ideas. A natural question is whether and how these similarities can be distilled and described in a precise, unified way. In this work, we combine stepwise refinement and lockstep models to provide an abstract and unified view of a sizeable family of consensus algorithms. Our models provide insights into the design choices underlying the different algorithms, and classify them based on those choices.

Consensus Refined

March 17, 2025

Contents

1 Introduction

Distributed consensus is a fundamental problem in distributed computing: a fixed set of processes must *agree* on a single value from a set of proposed ones. Algorithms that solve this problem provide building blocks for many higher-level tasks, such as distributed leases, group membership, atomic broadcast (also known as total-order broadcast or multi-consensus), and so forth. These in turn provide building blocks for yet higher-level tasks like system replication. In this work, however, our focus is on consensus algorithms "proper", rather than their applications. Namely, we consider consensus algorithms for the asynchronous message-passing setting with benign link and process failures.

Although the setting we consider explicitly excludes malicious behavior, the interplay of concurrency, asynchrony, and failures can still drive the execution of any consensus algorithm in many different ways. This makes the understanding of both the algorithms and their correctness non-trivial. Furthermore, many consensus algorithms have been proposed in the literature. Many of these algorithms appear to share similar underlying algorithmic ideas, although their presentation, structure and details differ. A natural question is whether these similarities can be distilled and captured in a uniform and generic way. In the same vein, one may ask whether the algorithms can be classified by some natural criteria.

This formalization, which accompanies our conference paper [5], is our contribution towards addressing these issues. Our primary tool in tackling them is *abstraction*. We describe consensus algorithms using *stepwise refinement*. In this method, an algorithm is derived through a sequence of models. The initial models in the sequence can describe the algorithms in arbitrarily abstract terms. In our abstractions, we remove message passing and describe the system using non-local steps that depend on the states of multiple processes. These abstractions allow us to focus on the main algorithmic ideas, without getting bogged down in details, thereby providing simplicity. We then gradually introduce details in successive, more concrete models that refine the abstract ones. In order to be implementable in a distributed setting, the final models must use strictly local steps, and communicate only by passing messages. The link between abstract and concrete models is precisely described and proved using *refinement relations*. Furthermore, the same abstract model can be implemented by different algorithms. This re-

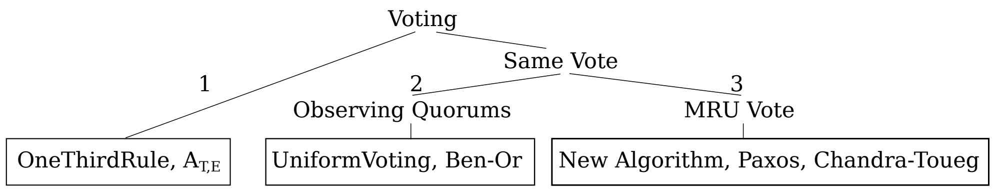

Figure 1: The consensus family tree. Boxes contain models of concrete algorithms.

sults in a *refinement tree* of models, where branching corresponds to different implementations.

Figure 1 shows the resulting refinement tree for our development. It captures the relationships between the different consensus algorithms found at its leaves: OneThirdRule, $A_{T,E}$, Ben-Or's algorithm, UniformVoting, Paxos, Chandra-Toueg algorithm and a new algorithm that we present. The refinement tree provides a natural classification of these algorithms. The new algorithm answers a question raised in [2], asking whether there exists a leaderless consensus algorithm that requires no waiting to provide safety, while tolerating up to $\frac{N}{2}$ process failures.

Our abstract (non-leaf) models are represented using unlabeled transition systems. For the models of the concrete algorithms, we adopt the Heard-Of model [2]) and reuse its Isabelle formalization by Debrat and Merz [3]. The Heard-Of model belongs to a class of models we refer to as *lockstep*, and which are applicable to algorithms which operate in communication-closed rounds. For this class of algorithms, the asynchronous setting is replaced by what is an essentially a synchronous model weakened by message loss (dual to strengthening the asynchronous model by failure detectors). This provides the illusion that all the processes operate in lockstep. Yet our results translate to the asynchronous setting of the real world, thanks to the preservation result established in [1] (and formalized in [3]).

2 Preliminaries

theory *Infra* **imports** *Main*
begin

2.1 Prover configuration

declare *if-split-asm* [*split*]

2.2 Forward reasoning ("attributes")

The following lemmas are used to produce intro/elim rules from set definitions and relation definitions.

lemmas *set-def-to-intro* = *eqset-imp-iff* [*THEN iffD2*]
lemmas *set-def-to-dest* = *eqset-imp-iff* [*THEN iffD1*]
lemmas *set-def-to-elim* = *set-def-to-dest* [*elim-format*]

lemmas *setc-def-to-intro* =
set-def-to-intro [**where** $B=\{x.\ P\ x\}$, *simplified*] **for** P

lemmas *setc-def-to-dest* =
set-def-to-dest [**where** $B=\{x.\ P\ x\}$, *simplified*] **for** P

lemmas *setc-def-to-elim* = *setc-def-to-dest* [*elim-format*]

lemmas *rel-def-to-intro* = *setc-def-to-intro* [**where** $x=(s,\ t)$] **for** $s\ t$
lemmas *rel-def-to-dest* = *setc-def-to-dest* [**where** $x=(s,\ t)$] **for** $s\ t$
lemmas *rel-def-to-elim* = *rel-def-to-dest* [*elim-format*]

2.3 General results

2.3.1 Maps

We usually remove *domIff* from the simpset and clasets due to annoying behavior. Sometimes the lemmas below are more well-behaved than *domIff*. Usually to be used as "dest: dom_lemmas". However, adding them as permanent dest rules slows down proofs too much, so we refrain from doing this.

lemma *map-definedness*:
$f\ x = Some\ y \Longrightarrow x \in dom\ f$
by (*simp add*: *domIff*)

lemma *map-definedness-contra*:
$[\![\ f\ x = Some\ y;\ z \notin dom\ f\ ]\!] \Longrightarrow x \neq z$

by (*auto simp add*: *domIff*)

lemmas *dom-lemmas* = *map-definedness map-definedness-contra*

2.3.2 Set

declare *image-comp*[*symmetric*, *simp*]

lemma *vimage-image-subset*: $A \subseteq f-`(f`A)$
by (*auto simp add*: *image-def vimage-def*)

2.3.3 Relations

lemma *Image-compose* [*simp*]:
  $(R1\ O\ R2)``A\ = R2``(R1``A)$
by (*auto*)

2.3.4 Lists

lemma *map-id*: *map id* = *id*
by (*simp*)

— Do NOT add the following equation to the simpset! (looping)
lemma *map-comp*: *map* (*g o f*) = *map g o map f*
by (*simp*)

declare *map-comp-map* [*simp del*]

lemma *take-prefix*: $[\![$ *take n l* = *xs* $]\!] \Longrightarrow \exists\, xs'.\ l = xs\ @\ xs'$
by (*induct l arbitrary*: *n xs*, *auto*)
  (*case-tac n*, *auto*)

2.3.5 Finite sets

Cardinality.

declare *arg-cong* [**where** *f*=*card*, *intro*]

lemma *finite-positive-cardI* [*intro*!]:
  $[\![\ A \neq \{\};\ finite\ A\ ]\!] \Longrightarrow 0 < card\ A$
by (*auto*)

lemma *finite-positive-cardD* [*dest*!]:
  $[\![\; 0 < card\ A;\ finite\ A \;]\!] \Longrightarrow A \neq \{\}$
by (*auto*)

lemma *finite-zero-cardI* [*intro*!]:
  $[\![\; A = \{\};\ finite\ A \;]\!] \Longrightarrow card\ A = 0$
by (*auto*)

lemma *finite-zero-cardD* [*dest*!]:
  $[\![\; card\ A = 0;\ finite\ A \;]\!] \Longrightarrow A = \{\}$
by (*auto*)

end

2.4 Consensus: types

typedecl *process*

Once we start taking maximums (e.g. in Last_Voting), we will need the process set to be finite

axiomatization where *process-finite*:

  OFCLASS(*process*, *finite-class*)

instance *process* :: *finite* **by** (*rule process-finite*)

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

typedecl *val* — Type of values to choose from

type-synonym *round* = *nat*

end

2.5 Quorums

locale *quorum* =

fixes *Quorum* :: *'a set set*
assumes
qintersect: $\llbracket$ $Q \in Quorum$; $Q' \in Quorum$ $\rrbracket \Longrightarrow Q \cap Q' \neq \{\}$
— Non-emptiness needed for some invariants of Coordinated Voting
and *Quorum-not-empty*: $\exists\, Q.\ Q \in Quorum$

lemma (**in** *quorum*) *quorum-non-empty*: $Q \in Quorum \Longrightarrow Q \neq \{\}$
by (*auto dest*: *qintersect*)

lemma (**in** *quorum*) *empty-not-quorum*: $\{\} \in Quorum \Longrightarrow False$
using *quorum-non-empty*
by *blast*

locale *quorum-process* = *quorum Quorum*
for *Quorum* :: *process set set*

locale *mono-quorum* = *quorum-process* +
assumes *mono-quorum*: $\llbracket$ $Q \in Quorum$; $Q \subseteq Q'$ $\rrbracket \Longrightarrow Q' \in Quorum$

lemma (**in** *mono-quorum*) *UNIV-quorum*:
UNIV $\in$ *Quorum*
using *Quorum-not-empty mono-quorum*
by(*blast*)

definition *majs* :: (*process set*) *set* **where**
majs $\equiv$ $\{S.\ card\ S > N\ div\ 2\}$

lemma *majsI*:
N div 2 < *card S* $\Longrightarrow$ $S \in majs$
by(*simp add*: *majs-def*)

lemma *card-Compl*:
fixes *S* :: (*'a* :: *finite*) *set*
shows *card* $(-S)$ = *card* (*UNIV* :: *'a set*) − *card S*
proof−
have *card S* + *card* $(-S)$ = *card* (*UNIV* :: *'a set*)
by(*rule card-Un-disjoint*[*of S* −*S*, *simplified Compl-partition*, *symmetric*])
(*auto*)
thus *?thesis*
by *simp*

qed

lemma *majorities-intersect*:
  $card\ (Q :: process\ set) + card\ Q' > N \Longrightarrow Q \cap Q' \neq \{\}$
  by (*metis card-Un-disjoint card-mono finite not-le top-greatest*)

interpretation *majorities*: *mono-quorum majs*
proof
  fix $Q\ Q'$ **assume** $Q \in majs\ Q' \in majs$
  thus $Q \cap Q' \neq \{\}$
    by (*intro majorities-intersect*) (*auto simp add*: *majs-def*)
next
  show $\exists\, Q.\ Q \in majs$
    apply(*rule-tac x=UNIV* **in** *exI*)
   apply(*auto simp add*: *majs-def intro*!: *div-less-dividend finite-UNIV-card-ge-0*)
    done
next
  fix $Q\ Q'$
  assume $Q \in majs\ Q \subseteq Q'$
  thus $Q' \in majs$ **using** *card-mono*[*OF* - ‹$Q \subseteq Q'$›]
    by(*auto simp add*: *majs-def*)
qed

end

2.6 Miscellaneous lemmas

method-setup *clarsimp-all* =
  ‹*Method.sections clasimp-modifiers* >>
    K (*SIMPLE-METHOD o CHANGED-PROP o PARALLEL-ALLGOALS o clarsimp-tac*)›
  clarify simplified, *all goals*

definition *flip* **where**
  flip-def: $flip\ f \equiv \lambda x\ y.\ f\ y\ x$

lemma *option-expand'*:
  $[\![(option = None) = (option' = None); \bigwedge x\ y.\ [\![option = Some\ x; option' = Some\ y]\!] \Longrightarrow x = y]\!] \Longrightarrow$
    $option = option'$

by(*rule option.expand*, *auto*)

2.7 Argmax

definition *Max-by* :: $('a \Rightarrow 'b :: linorder) \Rightarrow 'a\ set \Rightarrow 'a$ **where**
Max-by f S = (*SOME x*. $x \in S \wedge f\ x = Max\ (f\ `\ S)$)

lemma *Max-by-dest*:
assumes *finite A* **and** $A \neq \{\}$
shows *Max-by f A* $\in A \wedge f$ (*Max-by f A*) = *Max* (f ‘ A) (**is** *?P* (*Max-by f A*))
proof(*simp only*: *Max-by-def some-eq-ex*[**where** *P*=*?P*])
from *assms* **have** *finite* (f ‘ A) f ‘ $A \neq \{\}$ **by** *auto*
from *Max-in*[*OF this*] **show** $\exists\, x.\ x \in A \wedge f\ x = Max\ (f$ ‘ $A)$
by (*metis image-iff*)
qed

lemma *Max-by-in*:
assumes *finite A* **and** $A \neq \{\}$
shows *Max-by f A* $\in A$ **using** *assms*
by(*rule Max-by-dest*[*THEN conjunct1*])

lemma *Max-by-ge*:
assumes *finite A* $x \in A$
shows $f\ x \leq f$ (*Max-by f A*)
proof−
from *assms*(*1*) **have** *fin-image*: *finite* (f ‘ A) **by** *simp*
from *assms*(*2*) **have** *non-empty*: $A \neq \{\}$ **by** *auto*
have $f\ x \in f$ ‘ A **using** *assms*(*2*) **by** *simp*
from *Max-ge*[*OF fin-image this*] **and** *Max-by-dest*[*OF assms*(*1*) *non-empty*, *of f*] **show** *?thesis*
by(*simp*)
qed

lemma *finite-UN-D*:
finite ($\bigcup S$) $\Longrightarrow \forall\, A \in S$. *finite A*
by (*metis Union-upper finite-subset*)

lemma *Max-by-eqI*:
assumes
fin: *finite A*

and $\bigwedge y.\ y \in A \Longrightarrow$ *cmp-f y* $\leq$ *cmp-f x*
and *in-X*: $x \in A$
and *inj*: *inj-on cmp-f A*
shows *Max-by cmp-f A* $= x$
proof−
have *in-M*: *Max-by cmp-f A* $\in A$ **using** *assms*
by(*auto intro*!: *Max-by-in*)
hence *cmp-f* (*Max-by cmp-f A*) $\leq$ *cmp-f x* **using** *assms*
by *auto*
also have *cmp-f x* $\leq$ *cmp-f* (*Max-by cmp-f A*)
by (*intro Max-by-ge assms*)
finally show *?thesis* **using** *inj in-M in-X*
by(*auto simp add*: *inj-on-def*)
qed

lemma *Max-by-Union-distrib*:
⟦ *finite A*; $A = \bigcup S$; $S \neq \{\}$; $\{\} \notin S$; *inj-on cmp-f A* ⟧ $\Longrightarrow$
Max-by cmp-f A $=$ *Max-by cmp-f* (*Max-by cmp-f* ‘ *S*)
proof(*rule Max-by-eqI*, *assumption*)
fix *y*
assume *assms*: *finite A* $A = \bigcup S$ $\{\} \notin S$ $y \in A$
then obtain *B* **where** *B-def*: $B \in S$ $y \in B$ **by** *auto*
hence *cmp-f y* $\leq$ *cmp-f* (*Max-by cmp-f B*) **using** *assms*
by (*metis Max-by-ge finite-UN-D*)
also have ... $\leq$ *cmp-f* (*Max-by cmp-f* (*Max-by cmp-f* ‘ *S*)) **using** *B-def*(*1*) *assms*
by (*metis Max-by-ge finite-UnionD finite-imageI imageI*)
finally show *cmp-f y* $\leq$ *cmp-f* (*Max-by cmp-f* (*Max-by cmp-f* ‘ *S*)) **.**
next
assume *assms*: *finite A* $A = \bigcup S$ $\{\} \notin S$ $S \neq \{\}$
hence *Max-by cmp-f* ‘ $S \neq \{\}$ *finite* (*Max-by cmp-f* ‘ *S*)
apply (*metis image-is-empty*)
by (*metis assms*(*1*) *assms*(*2*) *finite-UnionD finite-imageI*)
hence *Max-by cmp-f* (*Max-by cmp-f* ‘ *S*) $\in$ (*Max-by cmp-f* ‘ *S*)
by(*blast intro*!: *Max-by-in*)
also have (*Max-by cmp-f* ‘ *S*) $\subseteq A$
proof −
have *f1*: $\forall$ *v0 v1*. ($\neg$ *finite v0* $\vee$ *v0* $= \{\}$) $\vee$ *Max-by* (*v1*::'*a* $\Rightarrow$ '*b*) *v0* $\in$ *v0* **using** *Max-by-in* **by** *blast*
have $\forall$ *v1*. *v1* $\notin S$ $\vee$ *finite v1* **using** *assms*(*1*) *assms*(*2*) *finite-UN-D* **by** *blast*

then obtain *v2-13* :: *'a set set* ⇒ *'a* ⇒ *'a set* **where** *Max-by cmp-f ‘ S* ⊆ ⋃ *S*
using *f1 assms*(*3*) **by** *blast*
thus *Max-by cmp-f ‘ S* ⊆ *A* **using** *assms*(*2*) **by** *presburger*
qed
finally show *Max-by cmp-f* (*Max-by cmp-f ‘ S*) ∈ *A* **.**
qed

lemma *Max-by-UNION-distrib*:
⟦*finite A*; *A* = (⋃*x*∈*S*. *f x*); *S* ≠ {}; {} ∉ *f ‘ S*; *inj-on cmp-f A*⟧ ⟹
Max-by cmp-f A = *Max-by cmp-f* (*Max-by cmp-f ‘* (*f ‘ S*))
by(*force intro*!: *Max-by-Union-distrib*)

lemma *Max-by-eta*:
Max-by f = (λ*S*. (*SOME x*. *x* ∈ *S* ∧ *f x* = *Max* (*f ‘ S*)))
by(*auto simp add*: *Max-by-def*)

lemma *Max-is-Max-by-id*:
⟦ *finite S*; *S* ≠ {} ⟧ ⟹ *Max S* = *Max-by id S*
apply(*clarsimp simp add*: *Max-by-def*)
by (*metis* (*mono-tags*, *lifting*) *Max-in someI-ex*)

definition *option-Max-by* :: (*'a* ⇒ *'b* :: *linorder*) ⇒ *'a set* ⇒ *'a option* **where**
option-Max-by cmp-f A ≡ *if A* = {} *then None else Some* (*Max-by cmp-f A*)

2.8 Function and map graphs

definition *fun-graph* **where**
fun-graph f = {(*x*, *f x*)|*x*. *True*}

definition *map-graph* :: (*'a*, *'b*)*map* ⇒ (*'a* × *'b*) *set* **where**
map-graph f = {(*x*, *y*)|*x y*. (*x*, *Some y*) ∈ *fun-graph f*}

lemma *map-graph-mem*[*simp*]:
((*x*, *y*) ∈ *map-graph f*) = (*f x* = *Some y*)
by(*auto simp add*: *dom-def map-graph-def fun-graph-def*)

lemma *finite-fun-graph*:
finite A ⟹ *finite* (*fun-graph f* ∩ (*A* × *UNIV*))
apply(*rule bij-betw-finite*[**where** *A*=*A* **and** *f*=λ*x*. (*x*, *f x*), *THEN iffD1*])
by(*auto simp add*: *fun-graph-def bij-betw-def inj-on-def*)

lemma *finite-map-graph*:
finite A $\Longrightarrow$ *finite* (*map-graph f* $\cap$ (*A* $\times$ *UNIV*))
by(*force simp add*: *map-graph-def*
dest!: *finite-fun-graph*[**where** *f*=*f*]
intro!: *finite-surj*[**where** *A*=*fun-graph f* $\cap$ (*A* $\times$ *UNIV*) **and** *f*=*apsnd the*]
)

lemma *finite-dom-finite-map-graph*:
finite (*dom f*) $\Longrightarrow$ *finite* (*map-graph f*)
apply(*simp add*: *dom-def map-graph-def fun-graph-def*)
apply(*erule finite-surj*[**where** *f*=λ*x*. (*x*, *the* (*f x*))])
apply(*clarsimp simp add*: *image-def*)
by (*metis option.sel*)

lemma *ran-map-addD*:
x $\in$ *ran* (*m* ++ *f*) $\Longrightarrow$ *x* $\in$ *ran m* $\vee$ *x* $\in$ *ran f*
by(*auto simp add*: *ran-def*)

2.9 Constant maps

definition *const-map* :: *'v* $\Rightarrow$ *'k set* $\Rightarrow$ (*'k*, *'v*)*map* **where**
const-map v S $\equiv$ (λ-. *Some v*) |\` *S*

lemma *const-map-empty*[*simp*]:
const-map v {} = *Map.empty*
by(*auto simp add*: *const-map-def*)

lemma *const-map-ran*[*simp*]: *x* $\in$ *ran* (*const-map v S*) = (*S* $\neq$ {} $\wedge$ *x* = *v*)
by(*auto simp add*: *const-map-def ran-def restrict-map-def*)

lemma *const-map-is-None*:
(*const-map y A x* = *None*) = (*x* $\notin$ *A*)
by(*auto simp add*: *const-map-def restrict-map-def*)

lemma *const-map-is-Some*:
(*const-map y A x* = *Some z*) = (*z* = *y* $\wedge$ *x* $\in$ *A*)
by(*auto simp add*: *const-map-def restrict-map-def*)

lemma *const-map-in-set*:
$x \in A \Longrightarrow$ *const-map v A x* $=$ *Some v*
by(*auto simp add*: *const-map-def*)

lemma *const-map-notin-set*:
$x \notin A \Longrightarrow$ *const-map v A x* $=$ *None*
by(*auto simp add*: *const-map-def*)

lemma *dom-const-map*:
dom (*const-map v S*) $= S$
by(*auto simp add*: *const-map-def*)

2.10 Votes with maximum timestamps.

definition *vote-set* :: ($'round \Rightarrow ('process, 'val)map) \Rightarrow 'process\ set \Rightarrow ('round \times 'val)set$ **where**
vote-set vs Q $\equiv \{(r, v)|a\ r\ v.\ ((r, a), v) \in$ *map-graph* (*case-prod vs*) $\wedge a \in Q\}$

lemma *inj-on-fst-vote-set*:
inj-on fst (*vote-set v-hist* $\{p\}$)
by(*clarsimp simp add*: *inj-on-def vote-set-def*)

lemma *finite-vote-set*:
assumes $\forall r' \geq (r :: nat).$ *v-hist* $r' =$ *Map.empty*
finite S
shows *finite* (*vote-set v-hist S*)
proof−
define *vs* **where** *vs* $= \{((r, a), v)|r\ a\ v.\ ((r, a), v) \in$ *map-graph* (*case-prod v-hist*) $\wedge a \in S\}$
have *vs*
$= (\bigcup p{\in}S.\ ((\lambda(r, v).\ ((r, p), v))$ ‘ (*map-graph* ($\lambda r.$ *v-hist r p*))))
by(*auto simp add*: *map-graph-def fun-graph-def vs-def*)
also have ... $\leq (\bigcup p{\in}S.\ (\lambda r.\ ((r, p),$ *the* (*v-hist r p*))) ‘ $\{0..<r\})$
using *assms*(*1*)
apply *auto*
apply (*auto simp add*: *map-graph-def fun-graph-def image-def*)
apply (*metis le-less-linear option.distinct*(*1*))
done
also note *I*=*finite-subset*[*OF calculation*]
have *finite vs*

by(*auto intro*: *I assms*(*2*) *nat-seg-image-imp-finite*[**where** *n*=*r*])

thus *?thesis*
apply(*clarsimp simp add*: *map-graph-def fun-graph-def vote-set-def vs-def*)
apply(*erule finite-surj*[**where** *f*=λ((*r*, *a*), *v*). (*r*, *v*)])
by(*force simp add*: *image-def*)
qed

definition *mru-of-set*
:: (′*round* :: *linorder* ⇒ (′*process*, ′*val*)*map*) ⇒ (′*process set*, ′*round* × ′*val*)*map*
where
mru-of-set vs ≡ λ*Q*. *option-Max-by fst* (*vote-set vs Q*)

definition *process-mru*
:: (′*round* :: *linorder* ⇒ (′*process*, ′*val*)*map*) ⇒ (′*process*, ′*round* × ′*val*)*map*
where
process-mru vs ≡ λ*a*. *mru-of-set vs* {*a*}

lemma *process-mru-is-None*:
(*process-mru v-f a* = *None*) = (*vote-set v-f* {*a*} = {})
by(*auto simp add*: *process-mru-def mru-of-set-def option-Max-by-def*)

lemma *process-mru-is-Some*:
(*process-mru v-f a* = *Some rv*) = (*vote-set v-f* {*a*} ≠ {} ∧ *rv* = *Max-by fst* (*vote-set v-f* {*a*}))
by(*auto simp add*: *process-mru-def mru-of-set-def option-Max-by-def*)

lemma *vote-set-upd*:
vote-set (*v-hist*(*r* := *v-f*)) {*p*} =
(*if p* ∈ *dom v-f*
then insert (*r*, *the* (*v-f p*))
else id
)
(*if v-hist r p* = *None*
then vote-set v-hist {*p*}
else vote-set v-hist {*p*} − {(*r*, *the* (*v-hist r p*))}
)

by(*auto simp add*: *vote-set-def const-map-is-Some split*: *if-split-asm*)

lemma *finite-vote-set-upd*:
  finite (*vote-set v-hist* {*a*}) $\Longrightarrow$
   finite (*vote-set* (*v-hist*(*r* := *v-f*)) {*a*})
 by(*simp add*: *vote-set-upd*)

lemma *vote-setD*:
 rv $\in$ *vote-set v-f* {*a*} $\Longrightarrow$ *v-f* (*fst rv*) *a* = *Some* (*snd rv*)
 by(*auto simp add*: *vote-set-def*)

lemma *process-mru-new-votes*:
 assumes
  $\forall r' \geq (r :: nat)$. *v-hist* r' = *Map.empty*
 shows
  process-mru (*v-hist*(*r* := *v-f*)) =
  (*process-mru v-hist* ++ (λp. *map-option* (*Pair r*) (*v-f p*)))
proof(*rule ext*, *rule option-expand'*)
 fix *p*
 show
  (*process-mru* (*v-hist*(*r* := *v-f*)) *p* = *None*) =
  ((*process-mru v-hist* ++ (λp. *map-option* (*Pair r*) (*v-f p*))) *p* = *None*) **using**
assms
  by(*force simp add*: *vote-set-def restrict-map-def const-map-is-None process-mru-is-None*)
next
 fix *p rv rv'*
 assume *eqs*:
  process-mru (*v-hist*(*r* := *v-f*)) *p* = *Some rv*
  (*process-mru v-hist* ++ (λp. *map-option* (*Pair r*) (*v-f p*))) *p* = *Some rv'*
 moreover have *v-hist* (*r*) *p* = *None* **using** *assms*(*1*)
  by(*auto*)

 ultimately show *rv* = *rv'* **using** *eqs assms*
  by(*auto simp add*: *map-add-Some-iff const-map-is-Some const-map-is-None*
   process-mru-is-Some vote-set-upd dest!: *vote-setD intro*!: *Max-by-eqI*
   finite-vote-set[*OF assms*]
   intro: *ccontr inj-on-fst-vote-set*)
qed

end

2.11 Step definitions for 2-step algorithms

definition *two-phase* **where** *two-phase* (*r*::*nat*) ≡ *r div 2*

definition *two-step* **where** *two-step* (*r*::*nat*) ≡ *r mod 2*

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

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

lemma *two-phase-step*: (*two-phase r* ∗ *2*) + *two-step r* = *r*
by (*auto simp add*: *two-phase-def two-step-def*)

lemma *two-step-phase-Suc*:
two-step r = *0* ⟹ *two-phase* (*Suc r*) = *two-phase r*
two-step r = *0* ⟹ *two-step* (*Suc r*) = *1*
two-step r = *0* ⟹ *two-phase* (*Suc* (*Suc r*)) = *Suc* (*two-phase r*)
two-step r = (*Suc 0*) ⟹ *two-phase* (*Suc r*) = *Suc* (*two-phase r*)
two-step r = (*Suc 0*) ⟹ *two-step* (*Suc r*) = *0*
by(*simp-all add*: *two-step-def two-phase-def mod-Suc div-Suc*)

end

2.12 Step definitions for 3-step algorithms

abbreviation (*input*) *nr-steps* ≡ *3*

definition *three-phase* **where** *three-phase* (*r*::*nat*) ≡ *r div nr-steps*

definition *three-step* **where** *three-step* (*r*::*nat*) ≡ *r mod nr-steps*

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

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

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

lemma *three-step-Suc*:
  three-step $r = 0 \Longrightarrow$ *three-step* (*Suc* (*Suc* r)) = *2*
  three-step $r = 0 \Longrightarrow$ *three-step* (*Suc* r) = *1*
  three-step r = (*Suc 0*) $\Longrightarrow$ *three-step* (*Suc* r) = *2*
  three-step r = (*Suc 0*) $\Longrightarrow$ *three-step* (*Suc* (*Suc* r)) = *0*
  three-step r = (*Suc* (*Suc 0*)) $\Longrightarrow$ *three-step* ((*Suc* r)) = *0*
  by(*unfold three-step-def*, *simp-all add*: *mod-Suc*)

lemma *three-step-phase-Suc*:
  three-step $r = 0 \Longrightarrow$ *three-phase* (*Suc* r) = *three-phase* r
  three-step $r = 0 \Longrightarrow$ *three-phase* (*Suc* (*Suc* r)) = *three-phase* r
  three-step $r = 0 \Longrightarrow$ *three-phase* (*Suc* (*Suc* (*Suc* r))) = *Suc* (*three-phase* r)
  three-step r = (*Suc 0*) $\Longrightarrow$ *three-phase* (*Suc* r) = *three-phase* r
  three-step r = (*Suc 0*) $\Longrightarrow$ *three-phase* (*Suc* (*Suc* r)) = *Suc* (*three-phase* r)
  three-step r = (*Suc* (*Suc 0*)) $\Longrightarrow$ *three-phase* (*Suc* r) = *Suc* (*three-phase* r)
  by(*simp-all add*: *three-step-def three-phase-def mod-Suc div-Suc*)

lemma *three-step2-phase-Suc*:
  three-step $r = 2 \Longrightarrow$ (*3* $*$ (*Suc* (*three-phase* r)) − *1*) = r
  apply(*simp add*: *three-step-def three-phase-def*)
  by (*metis add-2-eq-Suc′ mult-div-mod-eq*)

lemma *three-stepE*:
  ⟦ *three-step* $r = 0 \Longrightarrow P$; *three-step* $r = 1 \Longrightarrow P$; *three-step* $r = 2 \Longrightarrow P$ ⟧ $\Longrightarrow$
P
  by(*unfold three-step-def*, *arith*)

end

3 Models, Invariants and Refinements

theory *Refinement* **imports** *Infra*
begin

3.1 Specifications, reachability, and behaviours.

Transition systems are multi-pointed graphs.

record *′s TS* =

init :: *'s set*
trans :: (*'s* × *'s*) *set*

The inductive set of reachable states.

inductive-set
reach :: (*'s*, *'a*) *TS-scheme* ⇒ *'s set*
for *T* :: (*'s*, *'a*) *TS-scheme*
where
r-init [*intro*]: *s* ∈ *init T* ⟹ *s* ∈ *reach T*
| *r-trans* [*intro*]: ⟦ (*s*, *t*) ∈ *trans T*; *s* ∈ *reach T* ⟧ ⟹ *t* ∈ *reach T*

3.1.1 Finite behaviours

Note that behaviours grow at the head of the list, i.e., the initial state is at the end.

inductive-set
beh :: (*'s*, *'a*) *TS-scheme* ⇒ (*'s list*) *set*
for *T* :: (*'s*, *'a*) *TS-scheme*
where
b-empty [*iff*]: [] ∈ *beh T*
| *b-init* [*intro*]: *s* ∈ *init T* ⟹ [*s*] ∈ *beh T*
| *b-trans* [*intro*]: ⟦ *s* # *b* ∈ *beh T*; (*s*, *t*) ∈ *trans T* ⟧ ⟹ *t* # *s* # *b* ∈ *beh T*

inductive-cases *beh-non-empty*: *s* # *b* ∈ *beh T*

Behaviours are prefix closed.

lemma *beh-immediate-prefix-closed*:
s # *b* ∈ *beh T* ⟹ *b* ∈ *beh T*
by (*erule beh-non-empty*, *auto*)

lemma *beh-prefix-closed*:
c @ *b* ∈ *beh T* ⟹ *b* ∈ *beh T*
by (*induct c*, *auto dest*!: *beh-immediate-prefix-closed*)

States in behaviours are exactly reachable.

lemma *beh-in-reach* [*rule-format*]:
b ∈ *beh T* ⟹ (∀ *s* ∈ *set b*. *s* ∈ *reach T*)
by (*erule beh.induct*) (*auto*)

lemma *reach-in-beh*:

s ∈ *reach T* ⟹ ∃ *b* ∈ *beh T*. *s* ∈ *set b*
proof (*induction rule*: *reach.induct*)
case (*r-init s*) **thus** *?case* **by** (*auto intro*: *bexI* [**where** *x*=[*s*]])
next
case (*r-trans s t*) **thus** *?case*
proof −
from *r-trans*(*3*) **obtain** *b b0 b1* **where** *b* ∈ *beh T b* = *b1* @ *s* # *b0* **by** (*auto dest*: *split-list*)
hence *s* # *b0* ∈ *beh T* **by** (*auto intro*: *beh-prefix-closed*)
hence *t* # *s* # *b0* ∈ *beh T* **using** ‹(*s*, *t*) ∈ *trans T*› **by** *auto*
thus *?thesis* **by** − (*rule bexI*, *auto*)
qed
qed

lemma *reach-equiv-beh-states*: *reach T* = (⋃ *b*∈*beh T*. *set b*)
by (*auto intro*!: *reach-in-beh beh-in-reach*)

Consecutive states in a behavior are connected by the transition relation

lemma *beh-consecutive-in-trans*:
assumes *b* ∈ *beh TS*
and *Suc i* < *length b*
and *s* = *b* ! *Suc i*
and *t* = *b* ! *i*
shows (*s*, *t*) ∈ *trans TS*
proof−
from *assms* **have**
b = *take i b* @ *t* # *s* # *drop* (*Suc* (*Suc i*)) *b*
by(*auto simp add*: *id-take-nth-drop Cons-nth-drop-Suc*)
thus *?thesis*
by (*metis assms*(*1*) *beh-non-empty beh-prefix-closed list.distinct*(*1*) *list.inject*)
qed

3.1.2 Specifications, observability, and implementation

Specifications add an observer function to transition systems.

record (′*s*, ′*o*) *spec* = ′*s TS* +
obs :: ′*s* ⇒ ′*o*

lemma *beh-obs-upd* [*simp*]: *beh* (*S*(| *obs* := *x* |)) = *beh S*
by (*safe*) (*erule beh.induct*, *auto*)+

lemma *reach-obs-upd* [*simp*]: *reach* (*S*(| *obs* := *x* |)) = *reach S*
by (*safe*) (*erule reach.induct*, *auto*)+

Observable behaviour and reachability.

definition
  obeh :: ('*s*, '*o*) *spec* ⇒ ('*o list*) *set* **where**
  obeh S ≡ (*map* (*obs S*))'(*beh S*)

definition
  oreach :: ('*s*, '*o*) *spec* ⇒ '*o set* **where**
  oreach S ≡ (*obs S*)'(*reach S*)

lemma *oreach-equiv-obeh-states*: *oreach S* = (⋃ *b*∈*obeh S*. *set b*)
by (*auto simp add*: *reach-equiv-beh-states oreach-def obeh-def*)

lemma *obeh-pi-translation*:
  (*map pi*)'(*obeh S*) = *obeh* (*S*(| *obs* := *pi o* (*obs S*) |))
  by (*simp add*: *obeh-def image-comp*)

lemma *oreach-pi-translation*:
  pi'(*oreach S*) = *oreach* (*S*(| *obs* := *pi o* (*obs S*) |))
by (*auto simp add*: *oreach-def*)

A predicate P on the states of a specification is *observable* if it cannot distinguish between states yielding the same observation. Equivalently, P is observable if it is the inverse image under the observation function of a predicate on observations.

definition
  observable :: ['*s* ⇒ '*o*, '*s set*] ⇒ *bool*
where
  observable ob P ≡ ∀ *s s'*. *ob s* = *ob s'* ⟶ *s'* ∈ *P* ⟶ *s* ∈ *P*

definition
  observable2 :: ['*s* ⇒ '*o*, '*s set*] ⇒ *bool*
where
  observable2 ob P ≡ ∃ *Q*. *P* = *ob*−'*Q*

definition

$observable3 :: ['s \Rightarrow 'o, 's\ set] \Rightarrow bool$
where
$observable3\ ob\ P \equiv ob-`ob`P \subseteq P$ — other direction holds trivially

lemma *observableE* [*elim*]:
$[\![observable\ ob\ P;\ ob\ s = ob\ s';\ s' \in P]\!] \Longrightarrow s \in P$
by (*unfold observable-def*) (*fast*)

lemma *observable2-equiv-observable*: $observable2\ ob\ P = observable\ ob\ P$
by (*unfold observable-def observable2-def*) (*auto*)

lemma *observable3-equiv-observable2*: $observable3\ ob\ P = observable2\ ob\ P$
by (*unfold observable3-def observable2-def*) (*auto*)

lemma *observable-id* [*simp*]: *observable id P*
by (*simp add*: *observable-def*)

The set extension of a function *ob* is the left adjoint of a Galois connection on the powerset lattices over domain and range of *ob* where the right adjoint is the inverse image function.

lemma *image-vimage-adjoints*: $(ob`P \subseteq Q) = (P \subseteq ob-`Q)$
by *auto*

declare *image-vimage-subset* [*simp*, *intro*]
declare *vimage-image-subset* [*simp*, *intro*]

Similar but "reversed" (wrt to adjointness) relationships only hold under additional conditions.

lemma *image-r-vimage-l*: $[\![\ Q \subseteq ob`P;\ observable\ ob\ P\ ]\!] \Longrightarrow ob-`Q \subseteq P$
by (*auto*)

lemma *vimage-l-image-r*: $[\![\ ob-`Q \subseteq P;\ Q \subseteq range\ ob\ ]\!] \Longrightarrow Q \subseteq ob`P$
by (*drule image-mono* [**where** *f=ob*], *auto*)

Internal and external invariants

lemma *external-from-internal-invariant*:
$[\![\ reach\ S \subseteq P;\ (obs\ S)`P \subseteq Q\ ]\!]$
$\Longrightarrow oreach\ S \subseteq Q$
by (*auto simp add*: *oreach-def*)

lemma *external-from-internal-invariant-vimage*:
  $[\![$ *reach* $S \subseteq P$; $P \subseteq$ (*obs* S)$-$‘Q $]\!]$
  $\Longrightarrow$ *oreach* $S \subseteq Q$
by (*erule external-from-internal-invariant*) (*auto*)

lemma *external-to-internal-invariant-vimage*:
  $[\![$ *oreach* $S \subseteq Q$; (*obs* S)$-$‘$Q \subseteq P$ $]\!]$
  $\Longrightarrow$ *reach* $S \subseteq P$
by (*auto simp add*: *oreach-def*)

lemma *external-to-internal-invariant*:
  $[\![$ *oreach* $S \subseteq Q$; $Q \subseteq$ (*obs* S)‘P; *observable* (*obs* S) P $]\!]$
  $\Longrightarrow$ *reach* $S \subseteq P$
by (*erule external-to-internal-invariant-vimage*) (*auto*)

lemma *external-equiv-internal-invariant-vimage*:
  $[\![$ P = (*obs* S)$-$‘Q $]\!]$
  $\Longrightarrow$ (*oreach* $S \subseteq Q$) = (*reach* $S \subseteq P$)
by (*fast intro*: *external-from-internal-invariant-vimage*
          external-to-internal-invariant-vimage
    del: *subsetI*)

lemma *external-equiv-internal-invariant*:
  $[\![$ (*obs* S)‘P = Q; *observable* (*obs* S) P $]\!]$
  $\Longrightarrow$ (*oreach* $S \subseteq Q$) = (*reach* $S \subseteq P$)
by (*rule external-equiv-internal-invariant-vimage*) (*auto*)

Our notion of implementation is inclusion of observable behaviours.

definition
  implements :: [$'p \Rightarrow 'o$, ($'s$, $'o$) *spec*, ($'t$, $'p$) *spec*] $\Rightarrow$ *bool* **where**
  implements pi Sa Sc $\equiv$ (*map pi*)‘(*obeh Sc*) $\subseteq$ *obeh Sa*

Reflexivity and transitivity

lemma *implements-refl*: *implements id S S*
by (*auto simp add*: *implements-def*)

lemma *implements-trans*:
  $[\![$ *implements pi1 S1 S2*; *implements pi2 S2 S3* $]\!]$

$\Longrightarrow$ *implements* (*pi1 o pi2*) *S1 S3*
by (*auto simp add*: *implements-def image-comp del*: *subsetI*
dest: *image-mono* [**where** *f*=*map pi1*])

Preservation of external invariants

lemma *implements-oreach*:
implements pi Sa Sc $\Longrightarrow$ *pi*‘(*oreach Sc*) $\subseteq$ *oreach Sa*
by (*auto simp add*: *implements-def oreach-equiv-obeh-states dest*!: *subsetD*)

lemma *external-invariant-preservation*:
⟦ *oreach Sa* $\subseteq$ *Q*; *implements pi Sa Sc* ⟧
$\Longrightarrow$ *pi*‘(*oreach Sc*) $\subseteq$ *Q*
by (*rule subset-trans* [*OF implements-oreach*]) (*auto*)

lemma *external-invariant-translation*:
⟦ *oreach Sa* $\subseteq$ *Q*; *pi*−‘*Q* $\subseteq$ *P*; *implements pi Sa Sc* ⟧
$\Longrightarrow$ *oreach Sc* $\subseteq$ *P*
apply (*rule subset-trans* [*OF vimage-image-subset*, *of pi*])
apply (*rule subset-trans* [**where** *B*=*pi*−‘*Q*])
apply (*intro vimage-mono external-invariant-preservation*, *auto*)
done

Preservation of internal invariants

lemma *internal-invariant-translation*:
⟦ *reach Sa* $\subseteq$ *Pa*; *Pa* $\subseteq$ *obs Sa* −‘ *Qa*; *pi* −‘ *Qa* $\subseteq$ *Q*; *obs S* −‘ *Q* $\subseteq$ *P*;
implements pi Sa S ⟧
$\Longrightarrow$ *reach S* $\subseteq$ *P*
by (*rule external-from-internal-invariant-vimage* [
THEN external-invariant-translation,
THEN external-to-internal-invariant-vimage])
(*assumption*+)

3.2 Invariants

First we define Hoare triples over transition relations and then we derive proof rules to establish invariants.

3.2.1 Hoare triples

definition

$PO\text{-}hoare :: [{'s}\ set,\ ({'s} \times {'s})\ set,\ {'s}\ set] \Rightarrow bool$
  $(\langle(3\{\text{-}\}\ \text{-}\ \{>\ \text{-}\})\rangle\ [0,\ 0,\ 0]\ 90)$
where
  $\{pre\}\ R\ \{>\ post\} \equiv R``pre \subseteq post$

lemmas $PO\text{-}hoare\text{-}defs = PO\text{-}hoare\text{-}def\ Image\text{-}def$

lemma $\{P\}\ R\ \{>\ Q\} = (\forall\, s\ t.\ s \in P \longrightarrow (s,\ t) \in R \longrightarrow t \in Q)$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoareD*:
  $[\![\ \{I\}\ R\ \{>J\};\ s \in I;\ (s,\ s') \in R\ ]\!] \Longrightarrow s' \in J$
  by(*auto simp add*: *PO-hoare-def*)

Some essential facts about Hoare triples.

lemma *hoare-conseq-left* [*intro*]:
  $[\![\ \{P'\}\ R\ \{>\ Q\};\ P \subseteq P'\ ]\!]$
  $\Longrightarrow \{P\}\ R\ \{>\ Q\}$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-conseq-right*:
  $[\![\ \{P\}\ R\ \{>\ Q'\};\ Q' \subseteq Q\ ]\!]$
  $\Longrightarrow \{P\}\ R\ \{>\ Q\}$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-false-left* [*simp*]:
  $\{\{\}\}\ R\ \{>\ Q\}$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-true-right* [*simp*]:
  $\{P\}\ R\ \{>\ UNIV\}$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-conj-right* [*intro*!]:
  $[\![\ \{P\}\ R\ \{>\ Q1\};\ \{P\}\ R\ \{>\ Q2\}\ ]\!]$
  $\Longrightarrow \{P\}\ R\ \{>\ Q1 \cap Q2\}$
by (*auto simp add*: *PO-hoare-defs*)

Special transition relations.

lemma *hoare-stop* [*simp*, *intro*!]:

$\{P\}$ $\{\}$ $\{> Q\}$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-skip* [*simp*, *intro*!]:
$P \subseteq Q \Longrightarrow \{P\}\ Id\ \{> Q\}$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-trans-Un* [*iff*]:
$\{P\}\ R1 \cup R2\ \{> Q\} = (\{P\}\ R1\ \{> Q\} \wedge \{P\}\ R2\ \{> Q\})$
by (*auto simp add*: *PO-hoare-defs*)

lemma *hoare-trans-UN* [*iff*]:
$\{P\} \bigcup\ x.\ R\ x\ \{> Q\} = (\forall x.\ \{P\}\ R\ x\ \{> Q\})$
by (*auto simp add*: *PO-hoare-defs*)

3.2.2 Characterization of reachability

lemma *reach-init*: $reach\ T \subseteq I \Longrightarrow init\ T \subseteq I$
by (*auto dest*: *subsetD*)

lemma *reach-trans*: $reach\ T \subseteq I \Longrightarrow \{reach\ T\}\ trans\ T\ \{> I\}$
by (*auto simp add*: *PO-hoare-defs*)

Useful consequences.

corollary *init-reach* [*iff*]: $init\ T \subseteq reach\ T$
by (*rule reach-init*, *simp*)

corollary *trans-reach* [*iff*]: $\{reach\ T\}\ trans\ T\ \{> reach\ T\}$
by (*rule reach-trans*, *simp*)

3.2.3 Invariant proof rules

Basic proof rule for invariants.

lemma *inv-rule-basic*:
$[\![\ init\ T \subseteq P; \{P\}\ (trans\ T)\ \{> P\}\ ]\!]$
$\Longrightarrow reach\ T \subseteq P$
by (*safe*, *erule reach.induct*, *auto simp add*: *PO-hoare-def*)

General invariant proof rule. This rule is complete (set $I = reach\ T$).

lemma *inv-rule*:

$[\![$ *init* $T \subseteq I$; $I \subseteq P$; $\{I\}$ (*trans* T) $\{> I\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq P$
apply (*rule subset-trans*, *auto*) — strengthen goal
apply (*erule reach.induct*, *auto simp add*: *PO-hoare-def*)
done

The following rule is equivalent to the previous one.

lemma *INV-rule*:
$[\![$ *init* $T \subseteq I$; $\{I \cap$ *reach* $T\}$ (*trans* T) $\{> I\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq I$
by (*safe*, *erule reach.induct*, *auto simp add*: *PO-hoare-defs*)

Proof of equivalence.

lemma *inv-rule-from-INV-rule*:
$[\![$ *init* $T \subseteq I$; $I \subseteq P$; $\{I\}$ (*trans* T) $\{> I\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq P$
apply (*rule subset-trans*, *auto del*: *subsetI*)
apply (*rule INV-rule*, *auto*)
done

lemma *INV-rule-from-inv-rule*:
$[\![$ *init* $T \subseteq I$; $\{I \cap$ *reach* $T\}$ (*trans* T) $\{> I\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq I$
by (*rule-tac I=I* $\cap$ *reach T* **in** *inv-rule*, *auto*)

Incremental proof rule for invariants using auxiliary invariant(s). This rule might have become obsolete by addition of INV_rule.

lemma *inv-rule-incr*:
$[\![$ *init* $T \subseteq I$; $\{I \cap J\}$ (*trans* T) $\{> I\}$; *reach* $T \subseteq J$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq I$
by (*rule INV-rule*, *auto*)

3.3 Refinement

Our notion of refinement is simulation. We first define a general notion of relational Hoare tuple, which we then use to define the refinement proof obligation. Finally, we show that observation-consistent refinement of specifications implies the implementation relation between them.

3.3.1 Relational Hoare tuples

Relational Hoare tuples formalize the following generalized simulation diagram:

```
o -- Ra --> o
|           |
pre        post
|           |
v           V
o -- Rc --> o
```

Here, *Ra* and *Rc* are the abstract and concrete transition relations, and *pre* and *post* are the pre- and post-relations. (In the definiton below, the operator (*O*) stands for relational composition, which is defined as follows: (*O*) ≡ λ*r* *s*. {(*xa*, *x*). ((λ*x* *xa*. (*x*, *xa*) ∈ *r*) *OO* (λ*x* *xa*. (*x*, *xa*) ∈ *s*)) *xa* *x*}.)

definition
PO-rhoare ::
[('*s* × '*t*) *set*, ('*s* × '*s*) *set*, ('*t* × '*t*) *set*, ('*s* × '*t*) *set*] ⇒ *bool*
(‹(*4*{-} -, - {> -})› [*0*, *0*, *0*] *90*)
where
{*pre*} *Ra*, *Rc* {> *post*} ≡ *pre* *O* *Rc* ⊆ *Ra* *O* *post*

lemmas *PO-rhoare-defs* = *PO-rhoare-def relcomp-unfold*

Facts about relational Hoare tuples.

lemma *relhoare-conseq-left* [*intro*]:
⟦ {*pre*′} *Ra*, *Rc* {> *post*}; *pre* ⊆ *pre*′ ⟧
⟹ {*pre*} *Ra*, *Rc* {> *post*}
by (*auto simp add*: *PO-rhoare-defs dest*!: *subsetD*)

lemma *relhoare-conseq-right*: — do NOT declare [intro]
⟦ {*pre*} *Ra*, *Rc* {> *post*′}; *post*′ ⊆ *post* ⟧
⟹ {*pre*} *Ra*, *Rc* {> *post*}
by (*auto simp add*: *PO-rhoare-defs*)

lemma *relhoare-false-left* [*simp*]: — do NOT declare [intro]
{ {} } *Ra*, *Rc* {> *post*}
by (*auto simp add*: *PO-rhoare-defs*)

lemma *relhoare-true-right* [*simp*]: — not true in general
$\{pre\}\ Ra,\ Rc\ \{>\ UNIV\} = (Domain\ (pre\ O\ Rc) \subseteq Domain\ Ra)$
by (*auto simp add*: *PO-rhoare-defs*)

lemma *Domain-rel-comp* [*intro*]:
$Domain\ pre \subseteq R \Longrightarrow Domain\ (pre\ O\ Rc) \subseteq R$
by (*auto simp add*: *Domain-def*)

lemma *rel-hoare-skip* [*iff*]: $\{R\}\ Id,\ Id\ \{>\ R\}$
by (*auto simp add*: *PO-rhoare-def*)

Reflexivity and transitivity.

lemma *relhoare-refl* [*simp*]: $\{Id\}\ R,\ R\ \{>\ Id\}$
by (*auto simp add*: *PO-rhoare-defs*)

lemma *rhoare-trans*:
$[\![\ \{R1\}\ T1,\ T2\ \{>\ R1\};\ \{R2\}\ T2,\ T3\ \{>\ R2\}\ ]\!]$
$\Longrightarrow \{R1\ O\ R2\}\ T1,\ T3\ \{>\ R1\ O\ R2\}$
apply (*auto simp add*: *PO-rhoare-def del*: *subsetI*)
apply (*drule subset-refl* [*THEN relcomp-mono*, **where** *r*=*R1*])
apply (*drule subset-refl* [*THEN* [*2*] *relcomp-mono*, **where** *s*=*R2*])
apply (*auto simp add*: *O-assoc del*: *subsetI*)
done

Conjunction in the post-relation cannot be split in general. However, here are two useful special cases. In the first case the abstract transtition relation is deterministic and in the second case one conjunct is a cartesian product of two state predicates.

lemma *relhoare-conj-right-det*:
$[\![\ \{pre\}\ Ra,\ Rc\ \{>\ post1\};\ \{pre\}\ Ra,\ Rc\ \{>\ post2\};$
$single\text{-}valued\ Ra\ ]\!]$ — only for deterministic *Ra*!
$\Longrightarrow \{pre\}\ Ra,\ Rc\ \{>\ post1 \cap post2\}$
by (*auto simp add*: *PO-rhoare-defs dest*: *single-valuedD dest*!: *subsetD*)

lemma *relhoare-conj-right-cartesian* [*intro*]:
$[\![\ \{Domain\ pre\}\ Ra\ \{>\ I\};\ \{Range\ pre\}\ Rc\ \{>\ J\};$
$\{pre\}\ Ra,\ Rc\ \{>\ post\}\ ]\!]$
$\Longrightarrow \{pre\}\ Ra,\ Rc\ \{>\ post \cap I \times J\}$
by (*force simp add*: *PO-rhoare-defs PO-hoare-defs Domain-def Range-def*)

Separate rule for cartesian products.

corollary *relhoare-cartesian*:
  ⟦ {*Domain pre*} *Ra* {> *I*}; {*Range pre*} *Rc* {> *J*};
    {*pre*} *Ra*, *Rc* {> *post*} ⟧ — any *post*, including *UNIV*!
  ⟹ {*pre*} *Ra*, *Rc* {> *I* × *J*}
by (*auto intro*: *relhoare-conseq-right*)

Unions of transition relations.

lemma *relhoare-concrete-Un* [*simp*]:
  {*pre*} *Ra*, *Rc1* ∪ *Rc2* {> *post*}
  = ({*pre*} *Ra*, *Rc1* {> *post*} ∧ {*pre*} *Ra*, *Rc2* {> *post*})
apply (*auto simp add*: *PO-rhoare-defs*)
apply (*auto dest*!: *subsetD*)
done

lemma *relhoare-concrete-UN* [*simp*]:
  {*pre*} *Ra*, ⋃*x*. *Rc x* {> *post*} = (∀ *x*. {*pre*} *Ra*, *Rc x* {> *post*})
apply (*auto simp add*: *PO-rhoare-defs*)
 apply (*auto dest*!: *subsetD*)
done

lemma *relhoare-abstract-Un-left* [*intro*]:
  ⟦ {*pre*} *Ra1*, *Rc* {> *post*} ⟧
  ⟹ {*pre*} *Ra1* ∪ *Ra2*, *Rc* {> *post*}
by (*auto simp add*: *PO-rhoare-defs*)

lemma *relhoare-abstract-Un-right* [*intro*]:
  ⟦ {*pre*} *Ra2*, *Rc* {> *post*} ⟧
  ⟹ {*pre*} *Ra1* ∪ *Ra2*, *Rc* {> *post*}
by (*auto simp add*: *PO-rhoare-defs*)

lemma *relhoare-abstract-UN* [*intro*!]: — might be too aggressive?
  ⟦ {*pre*} *Ra x*, *Rc* {> *post*} ⟧
  ⟹ {*pre*} ⋃*x*. *Ra x*, *Rc* {> *post*}
apply (*auto simp add*: *PO-rhoare-defs*)
apply (*auto dest*!: *subsetD*)
done

3.3.2 Refinement proof obligations

A transition system refines another one if the initial states and the transitions are refined. Initial state refinement means that for each concrete initial state there is a related abstract one. Transition refinement means that the simulation relation is preserved (as expressed by a relational Hoare tuple).

definition
  PO-refines ::
    $[('s \times 't)\ set,\ ('s,\ 'a)\ TS\text{-}scheme,\ ('t,\ 'b)\ TS\text{-}scheme] \Rightarrow bool$
where
  PO-refines R Ta Tc ≡ (
      $init\ Tc \subseteq R``(init\ Ta)$
    $\wedge \{R\}\ (trans\ Ta),\ (trans\ Tc)\ \{>\ R\}$
)

Basic refinement rule. This is just an introduction rule for the definition.

lemma *refine-basic*:
  $[\![\ init\ Tc \subseteq R``(init\ Ta);\ \{R\}\ (trans\ Ta),\ (trans\ Tc)\ \{>\ R\}\ ]\!]$
  $\Longrightarrow$ *PO-refines R Ta Tc*
by (*simp add*: *PO-refines-def*)

The following proof rule uses individual invariants I and J of the concrete and abstract systems to strengthen the simulation relation R.

The hypotheses state that these state predicates are indeed invariants. Note that the pre-condition of the invariant preservation hypotheses for I and J are strengthened by adding the predicates $Domain\ (R \cap UNIV \times J)$ and $Range\ (R \cap I \times UNIV)$, respectively. In particular, the latter predicate may be essential, if a concrete invariant depends on the simulation relation and an abstract invariant, i.e. to "transport" abstract invariants to the concrete system.

lemma *refine-init-using-invariants*:
  $[\![\ init\ Tc \subseteq R``(init\ Ta);\ init\ Ta \subseteq I;\ init\ Tc \subseteq J\ ]\!]$
  $\Longrightarrow init\ Tc \subseteq (R \cap I \times J)``(init\ Ta)$
by (*auto simp add*: *Image-def dest*!: *bspec subsetD*)

lemma *refine-trans-using-invariants*:
  $[\![\ \{R \cap I \times J\}\ (trans\ Ta),\ (trans\ Tc)\ \{>\ R\};$
    $\{I \cap Domain\ (R \cap UNIV \times J)\}\ (trans\ Ta)\ \{>\ I\};$
    $\{J \cap Range\ (R \cap I \times UNIV)\}\ (trans\ Tc)\ \{>\ J\}\ ]\!]$

$\Longrightarrow \{R \cap I \times J\}$ (*trans Ta*), (*trans Tc*) $\{> R \cap I \times J\}$
by (*rule relhoare-conj-right-cartesian*) (*auto*)

This is our main rule for refinements.

lemma *refine-using-invariants*:
$[\![$ $\{R \cap I \times J\}$ (*trans Ta*), (*trans Tc*) $\{> R\}$;
$\{I \cap$ *Domain* $(R \cap$ *UNIV* $\times J)\}$ (*trans Ta*) $\{> I\}$;
$\{J \cap$ *Range* $(R \cap I \times$ *UNIV*$)\}$ (*trans Tc*) $\{> J\}$;
init Tc $\subseteq R$‘‘(*init Ta*);
init Ta $\subseteq I$; *init Tc* $\subseteq J$ $]\!]$
$\Longrightarrow$ *PO-refines* $(R \cap I \times J)$ *Ta Tc*
by (*unfold PO-refines-def*)
(*intro refine-init-using-invariants refine-trans-using-invariants conjI*)

3.3.3 Deriving invariants from refinements

Some invariants can only be proved after the simulation has been established, because they depend on the simulation relation and some abstract invariants. Here is a rule to derive invariant theorems from the refinement.

lemma *PO-refines-implies-Range-init*:
PO-refines R Ta Tc $\Longrightarrow$ *init Tc* $\subseteq$ *Range R*
by (*auto simp add*: *PO-refines-def*)

lemma *PO-refines-implies-Range-trans*:
PO-refines R Ta Tc $\Longrightarrow$ {*Range R*} *trans Tc* {> *Range R*}
by (*auto simp add*: *PO-refines-def PO-rhoare-def PO-hoare-def*)

lemma *PO-refines-implies-Range-invariant*:
PO-refines R Ta Tc $\Longrightarrow$ *reach Tc* $\subseteq$ *Range R*
by (*rule INV-rule*)
(*auto intro*!: *PO-refines-implies-Range-init*
PO-refines-implies-Range-trans)

The following rules are more useful in proofs.

corollary *INV-init-from-refinement*:
$[\![$ *PO-refines R Ta Tc*; *Range R* $\subseteq I$ $]\!]$
$\Longrightarrow$ *init Tc* $\subseteq I$
by (*drule PO-refines-implies-Range-init*, *auto*)

corollary *INV-trans-from-refinement*:

$[\![$ *PO-refines R Ta Tc*; $K \subseteq$ *Range R*; *Range R* $\subseteq I$ $]\!]$
$\Longrightarrow \{K\}$ *trans Tc* $\{> I\}$
apply (*drule PO-refines-implies-Range-trans*)
apply (*auto intro*: *hoare-conseq-right*)
done

corollary *INV-from-refinement*:
$[\![$ *PO-refines R Ta Tc*; *Range R* $\subseteq I$ $]\!]$
$\Longrightarrow$ *reach Tc* $\subseteq I$
by (*drule PO-refines-implies-Range-invariant*, *fast*)

3.3.4 Transfering abstract invariants to concrete systems

lemmas *hoare-conseq* = *hoare-conseq-right*[*OF hoare-conseq-left*] **for** P' R Q'

lemma *PO-refines-implies-R-image-init*:
PO-refines R Ta Tc $\Longrightarrow$ *init Tc* $\subseteq$ *R* ‘‘ (*init Ta*)
apply(*rule subset-trans*[**where** *B*=*R* ‘‘ *init Ta*])
apply (*auto simp add*: *PO-refines-def*)
done

lemma *commute-dest*:
$[\![$ *R O Tc* $\subseteq$ *Ta O R*; (*sa*, *sc*) $\in$ *R*; (*sc*, *sc*$'$) $\in$ *Tc* $]\!]$ $\Longrightarrow \exists$ *sa*$'$. (*sa*, *sa*$'$) $\in$ *Ta* $\wedge$ (*sa*$'$, *sc*$'$) $\in$ *R*
by(*auto*)

lemma *PO-refines-implies-R-image-trans*:
assumes *PO-refines R Ta Tc*
shows {*R* ‘‘ *reach Ta*} *trans Tc* {> *R* ‘‘ *reach Ta*} **using** *assms*
proof(*unfold PO-hoare-def Image-def PO-refines-def PO-rhoare-def*, *safe*)
fix *sc sc*$'$ *sa*
assume *R*: (*sa*, *sc*) $\in$ *R*
and *step*: (*sc*, *sc*$'$) $\in$ *TS.trans Tc*
and *sa-reach*: *sa* $\in$ *reach Ta*
and *trans-ref*: *R O trans Tc* $\subseteq$ *trans Ta O R*
from *commute-dest*[*OF trans-ref R step*] *sa-reach*
show $\exists$ *sa*$'$$\in$*reach Ta*. (*sa*$'$, *sc*$'$) $\in$ *R*
by(*auto*)
qed

```
lemma PO-refines-implies-R-image-invariant:
  assumes PO-refines R Ta Tc
  shows reach Tc ⊆ R `` reach Ta
proof(rule INV-rule)
  show init Tc ⊆ R `` reach Ta
    by (rule subset-trans[OF PO-refines-implies-R-image-init, OF assms]) (auto)
next
  show {R `` reach Ta ∩ reach Tc} TS.trans Tc {> R `` reach Ta} using assms
    by (auto intro!: PO-refines-implies-R-image-trans)
qed

lemma abs-INV-init-transfer:
  assumes
    PO-refines R Ta Tc
    init Ta ⊆ I
   shows init Tc ⊆ R `` I using PO-refines-implies-R-image-init[OF assms(1)]
assms(2)
    by(blast elim!: subset-trans intro: Image-mono)

lemma abs-INV-trans-transfer:
  assumes
    ref: PO-refines R Ta Tc
    and abs-hoare: {I} trans Ta {> J}
  shows {R `` I} trans Tc {> R `` J}
proof(unfold PO-hoare-def Image-def, safe)
  fix sc sc' sa
  assume step: (sc, sc') ∈ trans Tc and abs-inv: sa ∈ I and R: (sa, sc) ∈ R
  from ref step and R obtain sa' where
    abs-step: (sa, sa') ∈ trans Ta and R': (sa', sc') ∈ R
    by(auto simp add: PO-refines-def PO-rhoare-def)
  with hoareD[OF abs-hoare abs-inv abs-step]
  show ∃ sa'∈J. (sa', sc') ∈ R
    by(blast)
qed

lemma abs-INV-transfer:
  assumes
    PO-refines R Ta Tc
    reach Ta ⊆ I
  shows reach Tc ⊆ R `` I using PO-refines-implies-R-image-invariant[OF assms(1)]
```

assms(*2*)
  by(*auto*)

3.3.5 Refinement of specifications

Lift relation membership to finite sequences

inductive-set
  seq-lift :: ($'s \times 't$) *set* ⇒ ($'s$ *list* × $'t$ *list*) *set*
  for *R* :: ($'s \times 't$) *set*
where
  sl-nil [*iff*]: ([], []) ∈ *seq-lift R*
| *sl-cons* [*intro*]:
    ⟦ (*xs*, *ys*) ∈ *seq-lift R*; (*x*, *y*) ∈ *R* ⟧ ⟹ (*x*#*xs*, *y*#*ys*) ∈ *seq-lift R*

inductive-cases *sl-cons-right-invert*: (*ba′*, *t* # *bc*) ∈ *seq-lift R*

For each concrete behaviour there is a related abstract one.

lemma *behaviour-refinement*:
assumes *PO-refines R Ta Tc bc* ∈ *beh Tc*
shows ∃ *ba* ∈ *beh Ta*. (*ba*, *bc*) ∈ *seq-lift R*
using *assms*(*2*)
proof (*induct rule*: *beh.induct*)
  case *b-empty* **thus** *?case* **by** *auto*
next
  case (*b-init s*) **thus** *?case* **using** *assms*(*1*) **by** (*auto simp add*: *PO-refines-def*)
next
  case (*b-trans s b s′*) **show** *?case*
  proof −
    from *b-trans*(*2*) **obtain** *t c* **where** *t* # *c* ∈ *beh Ta* (*t*, *s*) ∈ *R* (*t* # *c*, *s* # *b*) ∈ *seq-lift R*
      by (*auto elim*!: *sl-cons-right-invert*)
    moreover
    from ‹(*t*, *s*) ∈ *R*› ‹(*s*, *s′*) ∈ *TS.trans Tc*› *assms*(*1*)
    obtain *t′* **where** (*t*, *t′*) ∈ *trans Ta* (*t′*, *s′*) ∈ *R*
      by (*auto simp add*: *PO-refines-def PO-rhoare-def*)
    ultimately
    have *t′* # *t* # *c* ∈ *beh Ta* (*t′* # *t* # *c*, *s′* # *s* # *b*) ∈ *seq-lift R* **by** *auto*
    thus *?thesis* **by** (*auto*)
  qed
qed

Observation consistency of a relation is defined using a mediator function *pi* to abstract the concrete observation. This allows us to also refine the observables as we move down a refinement branch.

definition
obs-consistent ::
$[('s \times 't)\ set,\ 'p \Rightarrow 'o,\ ('s,\ 'o)\ spec,\ ('t,\ 'p)\ spec] \Rightarrow bool$
where
obs-consistent R pi Sa Sc $\equiv (\forall\, s\ t.\ (s,\ t) \in R \longrightarrow pi\ (obs\ Sc\ t) = obs\ Sa\ s)$

lemma *obs-consistent-refl* [*iff*]: *obs-consistent Id id S S*
by (*simp add*: *obs-consistent-def*)

lemma *obs-consistent-trans* [*intro*]:
$[\![$ *obs-consistent R1 pi1 S1 S2*; *obs-consistent R2 pi2 S2 S3* $]\!]$
$\Longrightarrow$ *obs-consistent* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
by (*auto simp add*: *obs-consistent-def*)

lemma *obs-consistent-empty*: *obs-consistent* {} *pi Sa Sc*
by (*auto simp add*: *obs-consistent-def*)

lemma *obs-consistent-conj1* [*intro*]:
obs-consistent R pi Sa Sc $\Longrightarrow$ *obs-consistent* $(R \cap R')$ *pi Sa Sc*
by (*auto simp add*: *obs-consistent-def*)

lemma *obs-consistent-conj2* [*intro*]:
obs-consistent R pi Sa Sc $\Longrightarrow$ *obs-consistent* $(R' \cap R)$ *pi Sa Sc*
by (*auto simp add*: *obs-consistent-def*)

lemma *obs-consistent-behaviours*:
$[\![$ *obs-consistent R pi Sa Sc*; $bc \in beh\ Sc$; $ba \in beh\ Sa$; $(ba,\ bc) \in seq\text{-}lift\ R$ $]\!]$
$\Longrightarrow map\ pi\ (map\ (obs\ Sc)\ bc) = map\ (obs\ Sa)\ ba$
by (*erule seq-lift.induct*) (*auto simp add*: *obs-consistent-def*)

Definition of refinement proof obligations.

definition
refines ::
$[('s \times 't)\ set,\ 'p \Rightarrow 'o,\ ('s,\ 'o)\ spec,\ ('t,\ 'p)\ spec] \Rightarrow bool$
where
refines R pi Sa Sc $\equiv$ *obs-consistent R pi Sa Sc* $\wedge$ *PO-refines R Sa Sc*

lemmas *refines-defs* =
refines-def PO-refines-def

lemma *refinesI*:
⟦ *PO-refines R Sa Sc*; *obs-consistent R pi Sa Sc* ⟧
⟹ *refines R pi Sa Sc*
by (*simp add*: *refines-def*)

lemma *PO-refines-from-refines*:
refines R pi Sa Sc ⟹ *PO-refines R Sa Sc*
by (*simp add*: *refines-def*)

Reflexivity and transitivity of refinement.

lemma *refinement-reflexive*: *refines Id id S S*
by (*auto simp add*: *refines-defs*)

lemma *refinement-transitive*:
⟦ *refines R1 pi1 S1 S2*; *refines R2 pi2 S2 S3* ⟧
⟹ *refines* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
apply (*auto simp add*: *refines-defs del*: *subsetI intro*: *rhoare-trans*)
apply (*fastforce dest*: *Image-mono*)
done

Soundness of refinement for proving implementation

lemma *observable-behaviour-refinement*:
⟦ *refines R pi Sa Sc*; *bc* ∈ *obeh Sc* ⟧ ⟹ *map pi bc* ∈ *obeh Sa*
by (*auto simp add*: *refines-def obeh-def image-def*
dest!: *behaviour-refinement obs-consistent-behaviours*)

theorem *refinement-soundness*:
refines R pi Sa Sc ⟹ *implements pi Sa Sc*
by (*auto simp add*: *implements-def*
elim!: *observable-behaviour-refinement*)

Extended versions of proof rules including observations

lemmas *Refinement-basic* = *refine-basic* [*THEN refinesI*]
lemmas *Refinement-using-invariants* = *refine-using-invariants* [*THEN refinesI*]

lemmas *INV-init-from-Refinement* =
INV-init-from-refinement [*OF PO-refines-from-refines*]

lemmas *INV-trans-from-Refinement* =
INV-trans-from-refinement [*OF PO-refines-from-refines*]

lemmas *INV-from-Refinement* =
INV-from-refinement [*OF PO-refines-from-refines*]

end

3.4 Transition system semantics for HO models

The HO development already defines two trace semantics for algorithms in this model, the coarse- and fine-grained ones. However, both of these are defined on infinite traces. Since the semantics of our transition systems are defined on finite traces, we also provide such a semantics for the HO model. Since we only use refinement for safety properties, the result also extend to infinite traces (although we do not prove this in Isabelle).

definition *CHO-trans* **where**
CHO-trans A HOs SHOs coord =
$\{((r, st), (r', st'))|r\ r'\ st\ st'.$
$r' = Suc\ r$
$\wedge\ CSHOnextConfig\ A\ r\ st\ (HOs\ r)\ (SHOs\ r)\ (coord\ r)\ st'$
$\}$

definition *CHO-to-TS* ::
$('proc, 'pst, 'msg)\ CHOAlgorithm$
$\Rightarrow (nat \Rightarrow 'proc\ HO)$
$\Rightarrow (nat \Rightarrow 'proc\ HO)$
$\Rightarrow (nat \Rightarrow 'proc\ coord)$
$\Rightarrow (nat \times ('proc \Rightarrow 'pst))\ TS$
where
CHO-to-TS A HOs SHOs coord ≡ (|
$init = \{(0, st)|st.\ CHOinitConfig\ A\ st\ (coord\ 0)\},$
trans = *CHO-trans A HOs SHOs coord*
|)

definition *get-msgs* ::

$('proc \Rightarrow 'proc \Rightarrow 'pst \Rightarrow 'msg)$
$\Rightarrow ('proc \Rightarrow 'pst)$
$\Rightarrow 'proc\ HO$
$\Rightarrow 'proc\ HO$
$\Rightarrow 'proc \Rightarrow ('proc \rightharpoonup 'msg)set$

where

get-msgs snd-f cfg HO SHO $\equiv \lambda p.$
$\{\mu.\ (\forall q.\ q \in HO\ p \longleftrightarrow \mu\ q \neq None)$
$\wedge (\forall q.\ q \in SHO\ p \cap HO\ p \longrightarrow \mu\ q = Some\ (snd\text{-}f\ q\ p\ (cfg\ q)))\}$

definition *CSHO-trans-alt*
::
$(nat \Rightarrow 'proc \Rightarrow 'proc \Rightarrow 'pst \Rightarrow 'msg)$
$\Rightarrow (nat \Rightarrow 'proc \Rightarrow 'pst \Rightarrow ('proc \rightharpoonup 'msg) \Rightarrow 'proc \Rightarrow 'pst \Rightarrow bool)$
$\Rightarrow (nat \Rightarrow 'proc\ HO)$
$\Rightarrow (nat \Rightarrow 'proc\ HO)$
$\Rightarrow (nat \Rightarrow 'proc \Rightarrow 'proc)$
$\Rightarrow ((nat \times ('proc \Rightarrow 'pst)) \times (nat \times ('proc \Rightarrow 'pst)))set$

where

CSHO-trans-alt snd-f nxt-st HOs SHOs coords $\equiv$
$\bigcup r\ \mu.\ \{((r,\ cfg),\ (Suc\ r,\ cfg')) | cfg\ cfg'.\ \forall p.$
$\mu\ p \in (get\text{-}msgs\ (snd\text{-}f\ r)\ cfg\ (HOs\ r)\ (SHOs\ r)\ p)$
$\wedge (\forall p.\ nxt\text{-}st\ r\ p\ (cfg\ p)\ (\mu\ p)\ (coords\ r\ p)\ (cfg'\ p))$
$\}$

lemma *CHO-trans-alt*:
CHO-trans A HOs SHOs coords = *CSHO-trans-alt* (*sendMsg A*) (*CnextState A*) *HOs SHOs coords*
apply(*rule equalityI*)
apply(*force simp add*: *CHO-trans-def CSHO-trans-alt-def CSHOnextConfig-def SHOmsgVectors-def*
get-msgs-def restrict-map-def map-add-def choice-iff)
apply(*force simp add*: *CHO-trans-def CSHO-trans-alt-def CSHOnextConfig-def SHOmsgVectors-def*
get-msgs-def restrict-map-def map-add-def)
done

definition *K* **where**
K y $\equiv \lambda x.\ y$

lemma *SHOmsgVectors-get-msgs*:
SHOmsgVectors A r p cfg HOp SHOp = *get-msgs* (*sendMsg A r*) *cfg* (*K HOp*) (*K SHOp*) *p*
by(*auto simp add*: *SHOmsgVectors-def get-msgs-def K-def*)

lemma *get-msgs-K*:
get-msgs snd-f cfg (*K* (*HOs r p*)) (*K* (*SHOs r p*)) *p*
= *get-msgs snd-f cfg* (*HOs r*) (*SHOs r*) *p*
by(*auto simp add*: *get-msgs-def K-def*)

lemma *CSHORun-get-msgs*:
CSHORun (*A* :: ('*proc*, '*pst*, '*msg*) *CHOAlgorithm*) *rho HOs SHOs coords* = (
CHOinitConfig A (*rho 0*) (*coords 0*)
$\wedge$ ($\forall$ *r*. $\exists$ μ.
($\forall$ *p*.
μ *p* $\in$ *get-msgs* (*sendMsg A r*) (*rho r*) (*HOs r*) (*SHOs r*) *p*
$\wedge$ *CnextState A r p* (*rho r p*) (μ *p*) (*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*))))
by(*auto simp add*: *CSHORun-def CSHOnextConfig-def SHOmsgVectors-get-msgs nextState-def get-msgs-K*
Bex-def choice-iff)

lemmas *CSHORun-step* = *CSHORun-get-msgs*[*THEN iffD1*, *THEN conjunct2*]

lemma *get-msgs-dom*:
msgs $\in$ *get-msgs send s HOs SHOs p* $\Longrightarrow$ *dom msgs* = *HOs p*
by(*auto simp add*: *get-msgs-def*)

lemma *get-msgs-benign*:
get-msgs snd-f cfg HOs HOs p = { (*Some o* (λ*q*. (*snd-f q p* (*cfg q*)))) |' (*HOs p*)}
by(*auto simp add*: *get-msgs-def restrict-map-def*)

end

4 The Voting Model

theory *Voting* **imports** *Refinement Consensus-Misc Quorums*
begin

4.1 Model definition

record *v-state* =

next-round :: *round*
votes :: *round* ⇒ (*process*, *val*) *map*
decisions :: (*process*, *val*)*map*

Initially, no rounds have been executed (the next round is 0), no votes have been cast, and no decisions have been made.

definition *v-init* :: *v-state set* **where**
v-init = { (| *next-round* = *0*, *votes* = λr *a*. *None*, *decisions* = *Map.empty* |) }

context *quorum-process* **begin**

definition *quorum-for* :: *process set* ⇒ *val* ⇒ (*process*, *val*)*map* ⇒ *bool* **where**
quorum-for-def′:
quorum-for Q v v-f ≡ *Q* ∈ *Quorum* ∧ *v-f* ‘ *Q* = {*Some v*}

The following definition of *quorum-for* is easier to reason about in Isabelle.

lemma *quorum-for-def*:
quorum-for Q v v-f = (*Q* ∈ *Quorum* ∧ (∀ *p* ∈ *Q*. *v-f p* = *Some v*))
by(*auto simp add*: *quorum-for-def′ image-def dest*: *quorum-non-empty*)

definition *locked-in-vf* :: (*process*, *val*)*map* ⇒ *val* ⇒ *bool* **where**
locked-in-vf v-f v ≡ ∃ *Q*. *quorum-for Q v v-f*

definition *locked-in* :: *v-state* ⇒ *round* ⇒ *val* ⇒ *bool* **where**
locked-in s r v = *locked-in-vf* (*votes s r*) *v*

definition *d-guard* :: (*process* ⇒ *val option*) ⇒ (*process* ⇒ *val option*) ⇒ *bool* **where**
d-guard r-decisions r-votes ≡ ∀ *p v*.
r-decisions p = *Some v* ⟶ *locked-in-vf r-votes v*

definition *no-defection* :: *v-state* ⇒ (*process*, *val*)*map* ⇒ *round* ⇒ *bool* **where**
no-defection-def′:
no-defection s r-votes r ≡
∀ *r′* < *r*. ∀ *Q* ∈ *Quorum*. ∀ *v*. (*votes s r′*) ‘ *Q* = {*Some v*} ⟶ *r-votes* ‘ *Q* ⊆ {*None*, *Some v*}

The following definition of *no-defection* is easier to reason about in Isabelle.

```
lemma no-defection-def:
  no-defection s round-votes r =
    (∀ r' < r. ∀ a Q v. quorum-for Q v (votes s r') ∧ a ∈ Q ⟶ round-votes a ∈
{None, Some v})
  apply(auto simp add: no-defection-def' Ball-def quorum-for-def')
   apply(blast)
  by (metis option.discI option.inject)

definition locked :: v-state ⇒ val set where
  locked s = {v. ∃ r. locked-in s r v}
```

The sole system event.

```
definition v-round :: round ⇒ (process, val)map ⇒ (process, val)map ⇒ (v-state
× v-state) set where
  v-round r r-votes r-decisions = {(s, s').
     — guards
     r = next-round s
     ∧ no-defection s r-votes r
     ∧ d-guard r-decisions r-votes
     ∧ — actions
     s' = s⦇
       next-round := Suc r,
       votes := (votes s)(r := r-votes),
       decisions := (decisions s) ++ r-decisions
     ⦈
  }

lemmas v-evt-defs = v-round-def

definition v-trans :: (v-state × v-state) set where
  v-trans = (⋃ r v-f d-f. v-round r v-f d-f) ∪ Id

definition v-TS :: v-state TS where
  v-TS = ⦇ init = v-init, trans = v-trans ⦈

lemmas v-TS-defs = v-TS-def v-init-def v-trans-def
```

4.2 Invariants

The only rounds where votes could have been cast are the ones preceding the next round.

definition *Vinv1* **where**
Vinv1 = {*s*. ∀ *r*. *next-round s* ≤ *r* ⟶ *votes s r* = *Map.empty* }

lemmas *Vinv1I* = *Vinv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *Vinv1E* [*elim*] = *Vinv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *Vinv1D* = *Vinv1-def* [*THEN setc-def-to-dest*, *rule-format*]

The votes cast must respect the *no-defection* property.

definition *Vinv2* **where**
Vinv2 = {*s*. ∀ *r*. *no-defection s* (*votes s r*) *r* }

lemmas *Vinv2I* = *Vinv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *Vinv2E* [*elim*] = *Vinv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *Vinv2D* = *Vinv2-def* [*THEN setc-def-to-dest*, *rule-format*]

definition *Vinv3* **where**
Vinv3 = {*s*. *ran* (*decisions s*) ⊆ *locked s*}

lemmas *Vinv3I* = *Vinv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *Vinv3E* [*elim*] = *Vinv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *Vinv3D* = *Vinv3-def* [*THEN setc-def-to-dest*, *rule-format*]

4.2.1 Proofs of invariants

lemma *Vinv1-v-round*:
{*Vinv1*} *v-round r v-f d-f* {> *Vinv1*}
by(*auto simp add*: *PO-hoare-defs v-round-def intro*!: *Vinv1I*)

lemmas *Vinv1-event-pres* = *Vinv1-v-round*

lemma *Vinv1-inductive*:
init v-TS ⊆ *Vinv1*
{*Vinv1*} *trans v-TS* {> *Vinv1*}
apply (*simp add*: *v-TS-defs Vinv1-def*)
by (*auto simp add*: *v-TS-defs Vinv1-event-pres*)

lemma *Vinv1-invariant*: *reach v-TS* ⊆ *Vinv1*
by (*rule inv-rule-basic*, *auto intro*!: *Vinv1-inductive*)

The following two lemmas will be useful later, when we start taking votes with the maximum timestamp.

lemma *Vinv1-finite-map-graph*:
s ∈ *Vinv1* ⟹ *finite* (*map-graph* (*case-prod* (*votes s*)))
apply(*rule finite-dom-finite-map-graph*)
apply(*rule finite-subset*[**where** *B*={*0*..< *v-state.next-round s*} × *UNIV*])
apply(*auto simp add*: *Vinv1-def dom-def not-le*[*symmetric*])
done

lemma *Vinv1-finite-vote-set*:
s ∈ *Vinv1* ⟹ *finite* (*vote-set* (*votes s*) *Q*)
apply(*drule Vinv1-finite-map-graph*)
apply(*clarsimp simp add*: *map-graph-def fun-graph-def vote-set-def*)
apply(*erule finite-surj*[**where** *f*=λ((*r*, *a*), *v*). (*r*, *v*)])
by(*force simp add*: *image-def*)

lemma *process-mru-map-add*:
assumes
s ∈ *Vinv1*
shows
process-mru ((*votes s*)(*next-round s* := *v-f*)) =
(*process-mru* (*votes s*) ++ (λ*p*. *map-option* (*Pair* (*next-round s*)) (*v-f p*)))
proof−
from *assms*[*THEN Vinv1D*] **have** *empty*: ∀ *r'* ≥ *next-round s*. *votes s r'* = *Map.empty*
by *simp*
show *?thesis*
by(*auto simp add*: *process-mru-new-votes*[*OF empty*] *map-add-def split*: *option.split*)
qed

lemma *no-defection-empty*:
no-defection s Map.empty r'
by(*auto simp add*: *no-defection-def*)

lemma *no-defection-preserved*:
assumes
  $s \in Vinv1$
  $r = next\text{-}round\ s$
  $no\text{-}defection\ s\ v\text{-}f\ r$
  $no\text{-}defection\ s\ (votes\ s\ r')\ r'$
  $votes\ s' = (votes\ s)(r := v\text{-}f)$
shows
  $no\text{-}defection\ s'\ (votes\ s'\ r')\ r'$ **using** *assms*
  by(*force simp add*: *no-defection-def*)

lemma *Vinv2-v-round*:
  $\{Vinv2 \cap Vinv1\}\ v\text{-}round\ r\ v\text{-}f\ d\text{-}f\ \{>\ Vinv2\}$
  apply(*auto simp add*: *PO-hoare-defs intro*!: *Vinv2I*)
  apply(*rename-tac s' r' s*)
  apply(*erule no-defection-preserved*)
    apply(*auto simp add*: *v-round-def intro*!: *v-state.equality*)
  done

lemmas *Vinv2-event-pres* = *Vinv2-v-round*

lemma *Vinv2-inductive*:
  $init\ v\text{-}TS \subseteq Vinv2$
  $\{Vinv2 \cap Vinv1\}\ trans\ v\text{-}TS\ \{>\ Vinv2\}$
  apply(*simp add*: *v-TS-defs Vinv2-def no-defection-def*)
  by (*auto simp add*: *v-TS-defs Vinv2-event-pres*)

lemma *Vinv2-invariant*: $reach\ v\text{-}TS \subseteq Vinv2$
  by (*rule inv-rule-incr*, *auto intro*: *Vinv2-inductive Vinv1-invariant del*: *subsetI*)

lemma *locked-preserved*:
assumes
  $s \in Vinv1$
  $r = next\text{-}round\ s$
  $votes\ s' = (votes\ s)(r := v\text{-}f)$
shows
  $locked\ s \subseteq locked\ s'$ **using** *assms*

apply(*auto simp add*: *locked-def locked-in-def locked-in-vf-def quorum-for-def dest*!: *Vinv1D*)
by (*metis option.distinct*(*1*))

lemma *Vinv3-v-round*:
{*Vinv3* ∩ *Vinv1*} *v-round r v-f d-f* {> *Vinv3*}
proof(*clarsimp simp add*: *PO-hoare-defs*, *intro Vinv3I*, *safe*)
fix *s s′ v*
assume *step*: (*s*, *s′*) ∈ *v-round r v-f d-f* **and** *inv3*: *s* ∈ *Vinv3* **and** *inv1*: *s* ∈ *Vinv1*
and *dec*: *v* ∈ *ran* (*decisions s′*)
have *locked s* ⊆ *locked s′* **using** *step*
by(*intro locked-preserved*[*OF inv1*, **where** *s′*=*s′*]) (*auto simp add*: *v-round-def*)
with *Vinv3D*[*OF inv3*] *step dec*
show *v* ∈ *locked s′*
apply(*auto simp add*: *v-round-def dest*!: *ran-map-addD*)
apply(*auto simp add*: *locked-def locked-in-def d-guard-def ran-def*)
done
qed

lemmas *Vinv3-event-pres* = *Vinv3-v-round*

lemma *Vinv3-inductive*:
init v-TS ⊆ *Vinv3*
{*Vinv3* ∩ *Vinv1*} *trans v-TS* {> *Vinv3*}
apply(*simp add*: *v-TS-defs Vinv3-def no-defection-def*)
by (*auto simp add*: *v-TS-defs Vinv3-event-pres*)

lemma *Vinv3-invariant*: *reach v-TS* ⊆ *Vinv3*
by (*rule inv-rule-incr*, *auto intro*: *Vinv3-inductive Vinv1-invariant del*: *subsetI*)

4.3 Agreement and stability

Only a singe value can be locked within the votes for one round.

lemma *locked-in-vf-same*:
⟦ *locked-in-vf v-f v*; *locked-in-vf v-f w* ⟧ ⟹ *v* = *w* **using** *qintersect*

apply(*auto simp add*: *locked-in-vf-def quorum-for-def image-iff*)
by (*metis Int-iff all-not-in-conv option.inject*)

In any reachable state, no two different values can be locked in different rounds.

theorem *locked-in-different*:
assumes
s ∈ *Vinv2*
locked-in s r1 v
locked-in s r2 w
r1 < *r2*
shows
v = *w*
proof−
— To be locked, *v* and *w* must each have received votes from a quorum.
from *assms*(*2*−*3*) **obtain** *Q1 Q2*
where *Q12*: *Q1* ∈ *Quorum Q2* ∈ *Quorum quorum-for Q1 v* (*votes s r1*) *quorum-for Q2 w* (*votes s r2*)
by(*auto simp add*: *locked-in-def locked-in-vf-def quorum-for-def*)
— By the quorum intersection property, some process from *Q1* voted for *w*:
then obtain *a* **where** *a* ∈ *Q1 votes s r2 a* = *Some w*
using *qintersect*[*OF* ‹*Q1* ∈ *Quorum*› ‹*Q2* ∈ *Quorum*›]
by(*auto simp add*: *quorum-for-def*)
— But from *Vinv2* we conclude that *a* could not have defected by voting *w*, so *v* = *w*:
thus *?thesis* **using** ‹*s* ∈ *Vinv2*› ‹*quorum-for Q1 v* (*votes s r1*)› ‹*r1* < *r2*›
by(*fastforce simp add*: *Vinv2-def no-defection-def quorum-for-def′*)
qed

It is simple to extend the previous theorem to any two (not necessarily different) rounds.

theorem *locked-unique*:
assumes
s ∈ *Vinv2*
v ∈ *locked s w* ∈ *locked s*
shows
v = *w*
proof −
from *assms*(*2*−*3*) **obtain** *r1 r2* **where** *quoIn*: *locked-in s r1 v locked-in s r2 w*
by (*auto simp add*: *locked-def*)

```
  have r1 < r2 ∨ r1 = r2 ∨ r2 < r1 by (rule linorder-less-linear)
  thus ?thesis
  proof (elim disjE)
    assume r1 = r2
    with quoIn show ?thesis
      by(simp add: locked-in-def locked-in-vf-same)
  qed(auto intro: locked-in-different[OF ‹s ∈ Vinv2›] quoIn sym)
qed
```

We now prove that decisions are stable; once a process makes a decision, it never changes it, and it does not go back to an undecided state. Note that behaviors grow at the front; hence *tr* ! $(i - j)$ is later in the trace than *tr* ! i.

```
lemma stable-decision:
  assumes beh: tr ∈ beh v-TS
  and len: i < length tr
  and s: s = nth tr i
  and t: t = nth tr (i − j)
  and dec:
    decisions s p = Some v
  shows
    decisions t p = Some v
proof−
  — First, we show that the both s and t respect the invariants.
  have reach: s ∈ reach v-TS t ∈ reach v-TS using beh s t len
     apply(simp-all add: reach-equiv-beh-states)
     apply (metis len nth-mem)
    apply (metis less-imp-diff-less nth-mem)
    done
  hence invs2: s ∈ Vinv2 and invs3: s ∈ Vinv3
     by(blast dest: Vinv2-invariant[THEN subsetD] Vinv3-invariant[THEN sub-
setD])+

  show ?thesis using t
  proof(induction j arbitrary: t)
    case (Suc j)
    hence dec-j: decisions (tr ! (i − j)) p = Some v
      by simp
    thus decisions t p = Some v using Suc
    — As (−) is a total function on naturals, we perform a case distinction; if i <
```

j, the induction step is trivial.

proof(*cases* $i \leq j$)

— The non-trivial case.

case *False*

define t' **where** $t' = tr \; ! \; (i - j)$

— Both t and t' are reachable, thus respect the invariants, and they are related by the transition relation.

hence $t' \in$ *reach v-TS* $t \in$ *reach v-TS* **using** *beh len Suc*

by (*metis beh-in-reach less-imp-diff-less nth-mem*)+

hence *invs*: $t' \in$ *Vinv1* $t' \in$ *Vinv3* $t \in$ *Vinv2* $t \in$ *Vinv3*

by(*blast dest*: *Vinv1-invariant*[*THEN subsetD*] *Vinv2-invariant*[*THEN subsetD*]
Vinv3-invariant[*THEN subsetD*])+

hence *locked-v*: $v \in$ *locked* t' **using** *Suc*

by(*auto simp add*: t'*-def intro*: *ranI*)

have $i - j =$ *Suc* $(i - ($*Suc* $j))$ **using** *False*

by *simp*

hence *trans*: $(t', t) \in$ *trans v-TS* **using** *beh len Suc*

by(*auto simp add*: t'*-def intro*!: *beh-consecutive-in-trans*)

— Thus v also remains locked in t, and p does not revoke, nor change its decision.

hence *locked-v-t*: $v \in$ *locked* t **using** *locked-v*

by(*auto simp add*: *v-TS-defs v-round-def*
intro: *locked-preserved*[*OF invs*(*1*), *THEN subsetD*, *OF - - locked-v*])

from *trans* **obtain** w **where** *decisions* t p = *Some* w **using** *dec-j*

by(*fastforce simp add*: t'*-def v-TS-defs v-round-def*
split: *option.split option.split-asm*)

thus *?thesis* **using** *invs*(*4*)[*THEN Vinv3D*] *locked-v-t locked-unique*[*OF invs*(*3*)]

by (*metis contra-subsetD ranI*)

qed(*auto*)

next

case *0*

thus *decisions* t p = *Some* v **using** *assms*

by *auto*

qed

qed

Finally, we prove that the Voting model ensures agreement. Without a loss of generality, we assume that t preceeds s in the trace.

lemma *Voting-agreement*:
  assumes *beh*: *tr* ∈ *beh v-TS*
  and *len*: *i* < *length tr*
  and *s*: *s* = *nth tr i*
  and *t*: *t* = *nth tr* (*i* − *j*)
  and *dec*:
    decisions s p = *Some v*
    decisions t q = *Some w*
  shows *w* = *v*
proof−
  — Again, we first prove that the invariants hold for *s*.
  have *reach*: *s* ∈ *reach v-TS* **using** *beh s t len*
    apply(*simp-all add*: *reach-equiv-beh-states*)
    by (*metis nth-mem*)
  hence *invs2*: *s* ∈ *Vinv2* **and** *invs3*: *s* ∈ *Vinv3*
    by(*blast dest*: *Vinv2-invariant*[*THEN subsetD*] *Vinv3-invariant*[*THEN subsetD*])+

  — We now proceed to prove the thesis by induction.
  thus *?thesis* **using** *assms*
  proof(*induction j arbitrary*: *t*)
    case *0*
    hence
      v ∈ *locked* (*tr* ! *i*)
      w ∈ *locked* (*tr* ! *i*)
      by(*auto intro*: *ranI*)
    thus *?thesis* **using** *invs2* **using** *assms 0*
      by(*auto dest*: *locked-unique*)
  next
    case (*Suc j*)
    thus *?thesis*
    — Again, the totality of (−) makes the claim trivial if *i* < *j*.
    proof(*cases i* ≤ *j*)
      case *False*
      — In the non-trivial case, the proof follows from the decision stability theorem and the uniqueness of locked values.
      have *dec-t*: *decisions t p* = *Some v* **using** *Suc*
        by(*auto intro*: *stable-decision*[*OF beh len s*])
      have *t* ∈ *reach v-TS* **using** *beh len Suc*
        by (*metis beh-in-reach less-imp-diff-less nth-mem*)

hence *invs*: *t* ∈ *Vinv2* *t* ∈ *Vinv3*
by(*blast dest*: *Vinv2-invariant*[*THEN subsetD*] *Vinv3-invariant*[*THEN subsetD*])+
from *dec-t* **have** *v* ∈ *locked t* **using** *invs*(*2*)
by(*auto intro*: *ranI*)
moreover have *locked-w-t*: *w* ∈ *locked t* **using** *Suc* ‹*t* ∈ *Vinv3*›[*THEN Vinv3D*]
by(*auto intro*: *ranI*)
ultimately show *?thesis* **using** *locked-unique*[*OF* ‹*t* ∈ *Vinv2*›]
by *blast*
qed(*auto*)
qed
qed

end

end

5 The Optimized Voting Model

theory *Voting-Opt*
imports *Voting*
begin

5.1 Model definition

record *opt-v-state* =
next-round :: *round*
last-vote :: (*process*, *val*) *map*
decisions :: (*process*, *val*)*map*

definition *flv-init* **where**
flv-init = { ⦇ *next-round* = *0*, *last-vote* = *Map.empty*, *decisions* = *Map.empty* ⦈ }

context *quorum-process* **begin**

definition *fmru-lv* :: (*process*, *round* × *val*)*map* ⇒ (*process set*, *round* × *val*)*map* **where**

fmru-lv lvs Q = *option-Max-by fst* (*ran* (*lvs* |‘ *Q*))

definition *flv-guard* :: (*process*, *round* × *val*)*map* ⇒ *process set* ⇒ *val* ⇒ *bool*
where
flv-guard lvs Q v ≡ *Q* ∈ *Quorum* ∧
(*let alv* = *fmru-lv lvs Q in alv* = *None* ∨ (∃ *r*. *alv* = *Some* (*r*, *v*)))

definition *opt-no-defection* :: *opt-v-state* ⇒ (*process*, *val*)*map* ⇒ *bool* **where**
opt-no-defection-def′:
opt-no-defection s round-votes ≡
∀ *v*. ∀ *Q*. *quorum-for Q v* (*last-vote s*) ⟶ *round-votes* ‘ *Q* ⊆ {*None*, *Some v*}

lemma *opt-no-defection-def*:
opt-no-defection s round-votes =
(∀ *a Q v*. *quorum-for Q v* (*last-vote s*) ∧ *a* ∈ *Q* ⟶ *round-votes a* ∈ {*None*,
Some v})
apply(*auto simp add*: *opt-no-defection-def*′)
by (*metis option.distinct*(*1*) *option.sel*)

definition *flv-round* :: *round* ⇒ (*process*, *val*)*map* ⇒ (*process*, *val*)*map* ⇒ (*opt-v-state*
× *opt-v-state*) *set* **where**
flv-round r r-votes r-decisions = {(*s*, *s*′).
— guards
r = *next-round s*
∧ *opt-no-defection s r-votes*
∧ *d-guard r-decisions r-votes*
∧ — actions
s′ = *s*⦇
next-round := *Suc r*
, *last-vote* := *last-vote s* ++ *r-votes*
, *decisions* := (*decisions s*) ++ *r-decisions*
⦈
}

lemmas *flv-evt-defs* = *flv-round-def flv-guard-def*

definition *flv-trans* :: (*opt-v-state* × *opt-v-state*) *set* **where**
flv-trans = (⋃ *r v-f d-f*. *flv-round r v-f d-f*)

definition *flv-TS* :: *opt-v-state TS* **where**

flv-TS = ⦇ *init* = *flv-init*, *trans* = *flv-trans* ⦈

lemmas *flv-TS-defs* = *flv-TS-def flv-init-def flv-trans-def*

5.2 Refinement

definition *flv-ref-rel* :: (*v-state* × *opt-v-state*)*set* **where**
  flv-ref-rel = {(*sa*, *sc*).
    sc = ⦇
      next-round = *v-state.next-round sa*
      , *last-vote* = *map-option snd o* (*process-mru* (*votes sa*))
      , *decisions* = *v-state.decisions sa*
    ⦈
  }

5.2.1 Guard strengthening

lemma *process-mru-Max*:
  assumes
    inv: *sa* ∈ *Vinv1*
    and *process-mru*: *process-mru* (*votes sa*) *p* = *Some* (*r*, *v*)
  shows
    votes sa r p = *Some v* ∧ (∀ *r′* > *r*. *votes sa r′ p* = *None*)
proof−
  from *process-mru* **have** *not-empty*: *vote-set* (*votes sa*) {*p*} ≠ {}
    by(*auto simp add*: *process-mru-def mru-of-set-def option-Max-by-def*)
  note *Max-by-conds* = *Vinv1-finite-vote-set*[*OF inv*] *this*
  from *Max-by-dest*[*OF Max-by-conds*, **where** *f*=*fst*]
  have
    r:
      (*r*, *v*) = *Max-by fst* (*vote-set* (*votes sa*) {*p*})
      votes sa r p = *Some v*
      using *process-mru*
    by(*auto simp add*: *process-mru-def mru-of-set-def option-Max-by-def vote-set-def*)
  have ∀ *r′* >*r* . *votes sa r′ p* = *None*
  proof(*safe*)
    fix *r′*
    assume *less*: *r* < *r′*
    hence ∀ *v*. (*r′*, *v*) ∉ *vote-set* (*votes sa*) {*p*} **using** *process-mru*
      by(*auto dest*!: *Max-by-ge*[**where** *f*=*fst*, *OF Vinv1-finite-vote-set*[*OF inv*]]

simp add: *process-mru-def mru-of-set-def option-Max-by-def*)
thus *votes sa r′ p* = *None*
by(*auto simp add*: *vote-set-def*)
qed
thus *?thesis* **using** *r*(*2*)
by(*auto*)
qed

lemma *opt-no-defection-imp-no-defection*:
assumes
conc-guard: *opt-no-defection sc round-votes*
and *R*: (*sa*, *sc*) ∈ *flv-ref-rel*
and *ainv*: *sa* ∈ *Vinv1 sa* ∈ *Vinv2*
shows
no-defection sa round-votes r
proof(*unfold no-defection-def*, *safe*)
fix *r′ v a Q w*
assume
r′-less: *r′* < *r*
and *a-votes-w*: *round-votes a* = *Some w*
and *Q*: *quorum-for Q v* (*votes sa r′*)
and *a-in-Q*: *a* ∈ *Q*
have *Q* ∈ *Quorum* **using** *Q*
by(*auto simp add*: *quorum-for-def*)
hence *quorum-for Q v* (*last-vote sc*)
proof(*clarsimp simp add*: *quorum-for-def*)
fix *q*
assume *q* ∈ *Q*
with *Q* **have** *q-r′*: *votes sa r′ q* = *Some v*
by(*auto simp add*: *quorum-for-def*)
hence *votes*: *vote-set* (*votes sa*) {*q*} ≠ {}
by(*auto simp add*: *vote-set-def*)
then obtain *w* **where** *w*: *last-vote sc q* = *Some w* **using** *R*
by(*clarsimp simp add*: *flv-ref-rel-def process-mru-def mru-of-set-def*
option-Max-by-def)
with *R* **obtain** *r-max* **where** *process-mru* (*votes sa*) *q* = *Some* (*r-max*, *w*)
by(*clarsimp simp add*: *flv-ref-rel-def*)
from *process-mru-Max*[*OF ainv*(*1*) *this*] *q-r′* **have**
votes sa r-max q = *Some w*
r′ ≤ *r-max*

using *q-r′*
by(*auto simp add*: *not-less*[*symmetric*])
thus *last-vote sc q* = *Some v* **using** *ainv*(*2*) *Q* ‹*q* ∈ *Q*›
apply(*case-tac r-max* = *r′*)
apply(*clarsimp simp add*: *w Vinv2-def no-defection-def q-r′ dest*: *le-neq-implies-less*)
apply(*fastforce simp add*: *w Vinv2-def no-defection-def q-r′ dest*!: *le-neq-implies-less*)
done
qed
thus *round-votes a* = *Some v* **using** *conc-guard a-in-Q a-votes-w r′-less*
by(*fastforce simp add*: *opt-no-defection-def*)
qed

5.2.2 Action refinement

lemma *act-ref*:
assumes
inv: *s* ∈ *Vinv1*
shows
map-option snd o (*process-mru* ((*votes s*)(*v-state.next-round s* := *v-f*)))
= ((*map-option snd o* (*process-mru* (*votes s*))) ++ *v-f*)
by(*auto simp add*: *process-mru-map-add*[*OF inv*] *map-add-def split*: *option.split*)

5.2.3 The complete refinement proof

lemma *flv-round-refines*:
{*flv-ref-rel* ∩ (*Vinv1* ∩ *Vinv2*) × *UNIV*}
v-round r v-f d-f, *flv-round r v-f d-f*
{> *flv-ref-rel*}
by(*auto simp add*: *PO-rhoare-defs flv-round-def v-round-def*
flv-ref-rel-def act-ref
intro: *opt-no-defection-imp-no-defection*)

lemma *Last-Voting-Refines*:
PO-refines (*flv-ref-rel* ∩ (*Vinv1* ∩ *Vinv2*) × *UNIV*) *v-TS flv-TS*
proof(*rule refine-using-invariants*)
show *init flv-TS* ⊆ *flv-ref-rel* `` *init v-TS*
by(*auto simp add*: *flv-TS-defs v-TS-defs flv-ref-rel-def*
process-mru-def mru-of-set-def vote-set-def option-Max-by-def)
next
show
{*flv-ref-rel* ∩ (*Vinv1* ∩ *Vinv2*) × *UNIV*} *trans v-TS*, *trans flv-TS* {> *flv-ref-rel*}

by(*auto simp add*: *v-TS-defs flv-TS-defs intro*!: *flv-round-refines*)
next
show
{*Vinv1* ∩ *Vinv2* ∩ *Domain* (*flv-ref-rel* ∩ *UNIV* × *UNIV*)}
trans v-TS
{> *Vinv1* ∩ *Vinv2*}
using *Vinv1-inductive*(*2*) *Vinv2-inductive*(*2*)
by *blast*
qed(*auto intro*!: *Vinv1-inductive*(*1*) *Vinv2-inductive*(*1*))

end

end

6 The OneThirdRule Algorithm

theory *OneThirdRule-Defs*
imports *Heard-Of.HOModel* ../*Consensus-Types*
begin

The contents of this file have been taken almost verbatim from the Heard Of Model AFP entry. The only difference is that the types have been changed.

6.1 Model of the algorithm

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

record *'val pstate* =
last-vote :: *'val*
decision :: *'val option*

The initial value of field *last-vote* is unconstrained, but no decision has been taken initially.

definition *OTR-initState* **where**
OTR-initState p st ≡ *decision 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* :: (*process* ⇒ *'val option*) ⇒ *'val* ⇒ *process set* **where**
HOV msgs v ≡ { *q* . *msgs q* = *Some v* }

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

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

definition *MFR* :: (*process* ⇒ *'val option*) ⇒ *'val* ⇒ *bool* **where**
MFR msgs v ≡ ∀ *w*. *card* (*HOV msgs w*) ≤ *card* (*HOV msgs v*)

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

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

lemma *MFR-in-msgs*:
assumes *HO*:*HOs m p* ≠ {}
and *v*: *MFR* (*HOrcvdMsgs OTR-M m p* (*HOs m p*) (*rho m*)) *v*
(**is** *MFR ?msgs v*)
shows ∃ *q* ∈ *HOs m p*. *v* = *the* (*?msgs q*)
proof −
from *HO* **obtain** *q* **where** *q*: *q* ∈ *HOs m p*
by *auto*
with *v* **have** *HOV ?msgs* (*the* (*?msgs q*)) ≠ {}
by (*auto simp*: *HOV-def HOrcvdMsgs-def*)

hence *HOp*: $0 < card\ (HOV\ ?msgs\ (the\ (?msgs\ q)))$
  by *auto*
also from *v* **have** $\ldots \leq card\ (HOV\ ?msgs\ v)$
  by (*simp add*: *MFR-def*)
finally have $HOV\ ?msgs\ v \neq \{\}$
  by *auto*
thus *?thesis*
  by (*auto simp*: *HOV-def HOrcvdMsgs-def*)
qed

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

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

The next-state relation of algorithm *One-Third Rule* for every process is defined as follows: if the process has received values from more than 2/3 of all processes, the *last-vote* 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**
  $OTR\text{-}nextState\ r\ p\ (st::('val::linorder)\ pstate)\ msgs\ st' \equiv$
   $if\ (2*N)\ div\ 3 < card\ \{q.\ msgs\ q \neq None\}$
   $then\ st' = (\!|\ last\text{-}vote = Min\ \{v\ .\ MFR\ msgs\ v\},$
     $decision = (if\ (\exists\, v.\ TwoThirds\ msgs\ v)$
       $then\ Some\ (\epsilon\, v.\ TwoThirds\ msgs\ v)$
       $else\ decision\ st)\ |\!)$
   $else\ st' = st$

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

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

6.2 Communication predicate for *One-Third Rule*

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

definition *OTR-commPerRd* **where**
OTR-commPerRd HOrs ≡ *True*

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

6.3 The *One-Third Rule* Heard-Of machine

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

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

abbreviation *OTR-M* ≡ *OTR-HOMachine*::(*process*, *'val*::*linorder pstate*, *'val*) *HOMachine*

end

6.4 Proofs

definition *majs* :: (*process set*) *set* **where**
majs ≡ {*S*. *card S* > (*2* ∗ *N*) *div 3*}

lemma *card-Compl*:

fixes S :: ('*a* :: *finite*) *set*
shows *card* $(-S)$ = *card* (*UNIV* :: '*a set*) − *card* S
proof−
have *card* S + *card* $(-S)$ = *card* (*UNIV* :: '*a set*)
by(*rule card-Un-disjoint*[*of* S $-S$, *simplified Compl-partition*, *symmetric*])
(*auto*)
thus *?thesis*
by *simp*
qed

lemma *m-mult-div-Suc-m*:
$n > 0 \Longrightarrow m * n$ *div Suc* $m < n$
by (*simp add*: *less-mult-imp-div-less*)

interpretation *majorities*: *quorum-process majs*
proof
fix Q Q' **assume** $Q \in$ *majs* $Q' \in$ *majs*
hence $(4 * N)$ *div* $3 <$ *card* Q + *card* Q'
by(*auto simp add*: *majs-def*)
thus $Q \cap Q' \neq \{\}$
apply (*intro majorities-intersect*)
apply(*auto*)
done
next
have $N > 0$
by *auto*
have $2 * N$ *div* $3 < N$
by(*simp only*: *eval-nat-numeral m-mult-div-Suc-m*[*OF* ‹$N > 0$›])
thus $\exists\, Q.\ Q \in$ *majs*
apply(*rule-tac x*=*UNIV* **in** *exI*)
apply(*auto simp add*: *majs-def intro*!: *div-less-dividend*)
done
qed

lemma *card-Un-le*:
⟦ *finite* A; *finite* B ⟧ $\Longrightarrow$ *card* $(A \cup B) \leq$ *card* A + *card* B
by(*simp only*: *card-Un-Int*)

lemma *qintersect-card*:

assumes $Q \in majs\ Q' \in majs$
shows $card\ (Q \cap Q') > card\ (Q \cap -Q')$
proof−
have $card\ (Q \cap -Q') \leq card\ (-Q')$
by(*auto intro*!: *card-mono*)
also have ... $< N - (card\ (-Q) + card\ (-Q'))$
proof−
have *sum*: $N < card\ Q + card\ Q'$ **using** *assms*
by(*auto simp add*: *majs-def*)
have *le-N*: $card\ Q \leq N\ card\ Q' \leq N$ **by** (*auto intro*!: *card-mono*)
show *?thesis* **using** *assms sum*
apply(*simp add*: *card-Compl*)
apply(*intro diff-less-mono2*)
apply(*auto simp add*: *majs-def card-Compl*)
apply(*simp add*: *diff-add-assoc2*[*symmetric*, *OF le-N*(*1*)] *add-diff-assoc*[*OF le-N*(*2*)])
by (*metis add-mono le-N*(*1*) *le-N*(*2*) *less-diff-conv2 nat-add-left-cancel-less*)
qed
also have ... $\leq card\ (Q \cap Q')$
proof−
have $N - (card\ (-Q) + card\ (-Q')) \leq card\ (-(-Q \cup -Q'))$
apply(*simp only*: *card-Compl*[**where** $S{=}{-Q} \cup -Q'$])
apply(*auto intro*!: *diff-le-mono2 card-Un-le*)
done
thus *?thesis*
by(*auto*)
qed
finally show *?thesis* **.**
qed

axiomatization where *val-linorder*:

OFCLASS(*val*, *linorder-class*)

instance *val* :: *linorder* **by** (*rule val-linorder*)

type-synonym *p-TS-state* = (*nat* × (*process* ⇒ (*val pstate*)))

definition K **where**
$K\ y \equiv \lambda x.\ y$

definition *OTR-Alg* **where**
OTR-Alg =
(| *CinitState* = (λ *p st crd*. *OTR-initState p st*),
sendMsg = *OTR-sendMsg*,
CnextState = (λ *r p st msgs crd st'*. *OTR-nextState r p st msgs st'*)
|)

definition *OTR-TS* ::
(*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process*)
⇒ *p-TS-state TS*
where
OTR-TS HOs SHOs crds = *CHO-to-TS OTR-Alg HOs SHOs* (*K o crds*)

lemmas *OTR-TS-defs* = *OTR-TS-def CHO-to-TS-def OTR-Alg-def CHOinit-Config-def*
OTR-initState-def

definition
OTR-trans-step HOs ≡ ⋃ *r μ*.
{((*r*, *cfg*), *Suc r*, *cfg'*)|*cfg cfg'*.
(∀ *p*. *μ p* ∈ *get-msgs* (*OTR-sendMsg r*) *cfg* (*HOs r*) (*HOs r*) *p*) ∧
(∀ *p*. *OTR-nextState r p* (*cfg p*) (*μ p*) (*cfg' p*))}

definition *CSHOnextConfig* **where**
CSHOnextConfig A r cfg HO SHO coord cfg' ≡
∀ *p*. ∃ *μ* ∈ *SHOmsgVectors A r p cfg* (*HO p*) (*SHO p*).
CnextState A r p (*cfg p*) *μ* (*coord p*) (*cfg' p*)

type-synonym *rHO* = *nat* ⇒ *process HO*

6.4.1 Refinement

definition *otr-ref-rel* :: (*opt-v-state* × *p-TS-state*)*set* **where**
otr-ref-rel = {(*sa*, (*r*, *sc*)).
r = *next-round sa*
∧ (∀ *p*. *decisions sa p* = *decision* (*sc p*))
∧ *majorities.opt-no-defection sa* (*Some o last-vote o sc*)

}

lemma *decide-origin*:
assumes
send: μ *p* ∈ *get-msgs* (*OTR-sendMsg r*) *sc* (*HOs r*) (*HOs r*) *p*
and *nxt*: *OTR-nextState r p* (*sc p*) (μ *p*) (*sc′ p*)
and *new-dec*: *decision* (*sc′ p*) ≠ *decision* (*sc p*)
shows
∃ *v*. *decision* (*sc′ p*) = *Some v* ∧ {*q*. *last-vote* (*sc q*) = *v*} ∈ *majs*
proof−
from *new-dec* **and** *nxt* **obtain** *v* **where**
p-dec-v: *decision* (*sc′ p*) = *Some v*
and *two-thirds-v*: *TwoThirds* (μ *p*) *v*
apply(*auto simp add*: *OTR-nextState-def split*: *if-split-asm*)
by (*metis exE-some*)
then have *2* ∗ *N div 3* < *card* {*q*. *last-vote* (*sc q*) = *v*} **using** *send*
by(*auto simp add*: *get-msgs-benign OTR-sendMsg-def TwoThirds-def HOV-def*

restrict-map-def elim!: *less-le-trans intro*!: *card-mono*)
with *p-dec-v* **show** *?thesis* **by** (*auto simp add*: *majs-def*)
qed

lemma *MFR-in-msgs*:
assumes *HO*:*dom msgs* ≠ {}
and *v*: *MFR msgs v*
shows ∃ *q* ∈ *dom msgs*. *v* = *the* (*msgs q*)
proof −
from *HO* **obtain** *q* **where** *q*: *q* ∈ *dom msgs*
by *auto*
with *v* **have** *HOV msgs* (*the* (*msgs q*)) ≠ {}
by (*auto simp*: *HOV-def*)
hence *HOp*: *0* < *card* (*HOV msgs* (*the* (*msgs q*)))
by *auto*
also from *v* **have** ... ≤ *card* (*HOV msgs v*)
by (*simp add*: *MFR-def*)
finally have *HOV msgs v* ≠ {}
by *auto*
thus *?thesis*
by (*force simp*: *HOV-def*)
qed

lemma *step-ref*:
{*otr-ref-rel*}
($\bigcup$ *r v-f d-f*. *majorities.flv-round r v-f d-f*),
OTR-trans-step HOs
{> *otr-ref-rel*}
proof(*simp add*: *PO-rhoare-defs OTR-trans-step-def*, *safe*)
fix *sa r sc sc′ μ*
assume
R: (*sa*, *r*, *sc*) ∈ *otr-ref-rel*
and *send*: ∀ *p*. *μ p* ∈ *get-msgs* (*OTR-sendMsg r*) *sc* (*HOs r*) (*HOs r*) *p*
and *nxt*: ∀ *p*. *OTR-nextState r p* (*sc p*) (*μ p*) (*sc′ p*)
note *step*=*send nxt*
define *d-f*
where *d-f p* = (*if decision* (*sc′ p*) ≠ *decision* (*sc p*) *then decision* (*sc′ p*) *else None*) **for** *p*

define *sa′* **where** *sa′* = (|
opt-v-state.next-round = *Suc r*
, *opt-v-state.last-vote* = *opt-v-state.last-vote sa* ++ (*Some o last-vote o sc*)
, *opt-v-state.decisions* = *opt-v-state.decisions sa* ++ *d-f*
|)

have *majorities.d-guard d-f* (*Some o last-vote o sc*)
proof(*clarsimp simp add*: *majorities.d-guard-def d-f-def*)
fix *p v*
assume
Some v ≠ *decision* (*sc p*)
decision (*sc′ p*) = *Some v*
from *this* **and**
decide-origin[**where** *μ*=*μ* **and** *HOs*=*HOs* **and** *sc′*=*sc′*, *OF send*[*THEN spec*, *of p*] *nxt*[*THEN spec*, *of p*]]
show *quorum-process.locked-in-vf majs* (*Some* ∘ *last-vote* ∘ *sc*) *v*
by(*auto simp add*: *majorities.locked-in-vf-def majorities.quorum-for-def*)
qed
hence
(*sa*, *sa′*) ∈ *majorities.flv-round r* (*Some o last-vote o sc*) *d-f* **using** *R*
by(*auto simp add*: *majorities.flv-round-def otr-ref-rel-def sa′-def*)
moreover have (*sa′*, *Suc r*, *sc′*) ∈ *otr-ref-rel*
proof(*unfold otr-ref-rel-def*, *safe*)

```
    fix p
    show opt-v-state.decisions sa′ p = decision (sc′ p) using R nxt[THEN spec,
of p]
    by(auto simp add: otr-ref-rel-def sa′-def map-add-def d-f-def OTR-nextState-def
        split: option.split)
  next
    show quorum-process.opt-no-defection majs sa′ (Some ∘ last-vote ∘ sc′)
    proof(clarsimp simp add: sa′-def majorities.opt-no-defection-def map-add-def
majorities.quorum-for-def)
      fix Q p v
      assume Q: Q ∈ majs and Q-v: ∀ q ∈ Q. last-vote (sc q) = v and p-Q: p ∈
Q
      hence old: last-vote (sc p) = v by simp
      have v-maj: {q. last-vote (sc q) = v} ∈ majs using Q Q-v
        apply(simp add: majs-def)
        apply(erule less-le-trans, rule card-mono, auto)
        done
      show last-vote (sc′ p) = v
      proof(rule ccontr)
        assume new: last-vote (sc′ p) ≠ v
        let ?w = last-vote (sc′ p)
        have
        w-MFR: ?w = Min {z. MFR (μ p) z} (is ?w = Min ?MFRs) and dom-maj:
dom (μ p) ∈ majs
          using old new nxt[THEN spec, where x=p]
         by(auto simp add: OTR-nextState-def majs-def dom-def split: if-split-asm)
      from dom-maj have not-empty: dom (μ p) ≠ {} by(elim majorities.quorum-non-empty)
        from MFR-exists obtain mfr-v where mfr-v: mfr-v ∈ ?MFRs
          by fastforce
        from not-empty obtain q z where μ p q = Some z by(fastforce)
        hence 0 < card (HOV (μ p) (the (μ p q)))
          by(auto simp add: HOV-def)
        have ?w ∈ {z. MFR (μ p) z}
        proof(unfold w-MFR, rule Min-in)
          have ?MFRs ⊆ (the o (μ p)) ‘ (dom (μ p)) using not-empty
            by(auto simp: image-def intro: MFR-in-msgs)
          thus finite ?MFRs by (auto elim: finite-subset)
        qed(auto simp add: MFR-exists)
        hence card-HOV: card (HOV (μ p) v) ≤ card (HOV (μ p) ?w)
          by(auto simp add: MFR-def)
```

have *dom* (μ *p*) = *HOs r p* **using** *send*[*THEN spec*, **where** *x*=*p*]
by(*auto simp add*: *get-msgs-def*)
from *this*[*symmetric*] **have** $\forall$ *v'*. *HOV* (μ *p*) *v'* = {*q*. *last-vote* (*sc q*) = *v'*} $\cap$ *dom* (μ *p*)
using *send*[*THEN spec*, **where** *x*=*p*]
by(*fastforce simp add*: *HOV-def get-msgs-benign OTR-sendMsg-def restrict-map-def*)
hence *card-le1*: *card* ({*q*. *last-vote* (*sc q*) = *v*} $\cap$ *dom* (μ *p*)) $\leq$ *card* ({*q*. *last-vote* (*sc q*) = *?w*} $\cap$ *dom* (μ *p*))
using *card-HOV*
by(*simp*)
have
card ({*q*. *last-vote* (*sc q*) = *v*} $\cap$ *dom* (μ *p*)) $\leq$ *card* ({*q*. *last-vote* (*sc q*) $\neq$ *v*} $\cap$ *dom* (μ *p*))
apply(*rule le-trans*[*OF card-le1*], *rule card-mono*)
apply(*auto simp add*: *new*[*symmetric*])
done
thus *False* **using** *qintersect-card*[*OF dom-maj v-maj*]
by(*simp add*: *Int-commute Collect-neg-eq*)
qed
qed
qed(*auto simp add*: *sa'-def*)

ultimately show
$\exists$ *sa'*. ($\exists$ *ra v-f d-f*. (*sa*, *sa'*) $\in$ *quorum-process.flv-round majs ra v-f d-f*)
$\wedge$ (*sa'*, *Suc r*, *sc'*) $\in$ *otr-ref-rel*
by *blast*
qed

lemma *OTR-Refines-LV-VOting*:
PO-refines (*otr-ref-rel*)
majorities.flv-TS (*OTR-TS HOs HOs crds*)
proof(*rule refine-basic*)
show *init* (*OTR-TS HOs HOs crds*) $\subseteq$ *otr-ref-rel* `` *init* (*quorum-process.flv-TS majs*)
by(*auto simp add*: *OTR-TS-def CHO-to-TS-def OTR-Alg-def CHOinitConfig-def OTR-initState-def*
majorities.flv-TS-def flv-init-def majorities.opt-no-defection-def majorities.quorum-for-def'
otr-ref-rel-def)
next

show
{*otr-ref-rel*} *TS.trans* (*quorum-process.flv-TS majs*), *TS.trans* (*OTR-TS HOs HOs crds*) {> *otr-ref-rel*}
using *step-ref*
by(*simp add*: *majorities.flv-TS-defs OTR-TS-def CHO-to-TS-def OTR-Alg-def*

CSHO-trans-alt-def CHO-trans-alt OTR-trans-step-def)
qed

6.4.2 Termination

The termination proof for the algorithm is already given in the Heard-Of Model AFP entry, and we do not repeat it here.

end

7 The $A_{T,E}$ Algorithm

theory *Ate-Defs*
imports *Heard-Of.HOModel ../Consensus-Types*
begin

The contents of this file have been taken almost verbatim from the Heard Of Model AFP entry. The only difference is that the types have been changed.

7.1 Model of the algorithm

The following record models the local state of a process.

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

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

definition *Ate-initState* **where**
Ate-initState p st ≡ (*decide st* = *None*)

locale *ate-parameters* =
fixes α::*nat* **and** T::*nat* **and** E::*nat*

assumes *TNaE*: $T \geq 2*(N + 2*\alpha - E)$
and *TltN*: $T < N$
and *EltN*: $E < N$

begin

The following are consequences of the assumptions on the parameters.

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

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

lemma *Tge2a*: $T \geq 2 * \alpha$
using *TNaE EltN* **by** *auto*

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

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

definition *mostOftenRcvd* **where**
mostOftenRcvd (*msgs*::*process* ⇒ '*val option*) ≡
{*v*. ∀ *w*. *card* {*qq*. *msgs qq* = *Some w*} ≤ *card* {*qq*. *msgs qq* = *Some v*}}

definition
Ate-sendMsg :: *nat* ⇒ *process* ⇒ *process* ⇒ '*val pstate* ⇒ '*val*
where
Ate-sendMsg r p q st ≡ *x st*

definition
Ate-nextState :: *nat* ⇒ *process* ⇒ ('*val*::*linorder*) *pstate* ⇒ (*process* ⇒ '*val option*)
⇒ '*val pstate* ⇒ *bool*
where
Ate-nextState r p st msgs st' ≡
(*if card* {*q*. *msgs q* ≠ *None*} > *T*
then x st' = *Min* (*mostOftenRcvd msgs*)
else x st' = *x st*)
∧ ((∃ *v*. *card* {*q*. *msgs q* = *Some v*} > *E* ∧ *decide st'* = *Some v*)

$\vee \neg (\exists v.\ card\ \{q.\ msgs\ q = Some\ v\} > E)$
$\wedge\ decide\ st' = decide\ st)$

7.2 Communication predicate for $A_{T,E}$

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

7.3 The $A_{T,E}$ Heard-Of machine

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

definition *Ate-SHOMachine* **where**
$Ate\text{-}SHOMachine = (\!|$
$CinitState = (\lambda\ p\ st\ crd.\ Ate\text{-}initState\ p\ (st::('val::linorder)\ pstate)),$
$sendMsg = Ate\text{-}sendMsg,$
$CnextState = (\lambda\ r\ p\ st\ msgs\ crd\ st'.\ Ate\text{-}nextState\ r\ p\ st\ msgs\ st'),$

SHOcommPerRd = (*Ate-commPerRd*:: *process HO* ⇒ *process HO* ⇒ *bool*),
SHOcommGlobal = *Ate-commGlobal*
⦈

abbreviation
Ate-M ≡ (*Ate-SHOMachine*::(*process*, ′*val*::*linorder pstate*, ′*val*) *SHOMachine*)

end — locale *ate-parameters*

end

7.4 Proofs

axiomatization where *val-linorder*:
OFCLASS(*val*, *linorder-class*)

instance *val* :: *linorder* **by** (*rule val-linorder*)

context *ate-parameters*
begin

definition *majs* :: (*process set*) *set* **where**
majs ≡ {*S*. *card S* > *E*}

interpretation *majorities*: *quorum-process majs*
proof
fix *Q Q*′ **assume** *Q* ∈ *majs Q*′ ∈ *majs*
hence *N* < *card Q* + *card Q*′ **using** *majE*
by(*auto simp add*: *majs-def*)
thus *Q* ∩ *Q*′ ≠ {}
apply (*intro majorities-intersect*)
apply(*auto*)
done
next
from *EltN*
show ∃ *Q*. *Q* ∈ *majs*
apply(*rule-tac x*=*UNIV* **in** *exI*)
apply(*auto simp add*: *majs-def intro*!: *div-less-dividend*)
done
qed

type-synonym *p-TS-state* = (*nat* × (*process* ⇒ (*val pstate*)))

definition *K* **where**
$K\ y \equiv \lambda x.\ y$

definition *Ate-Alg* **where**
Ate-Alg =
⦇ *CinitState* = (λ *p st crd*. *Ate-initState p st*),
sendMsg = *Ate-sendMsg*,
CnextState = (λ *r p st msgs crd st'*. *Ate-nextState r p st msgs st'*)
⦈

definition *Ate-TS* ::
(*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process*)
⇒ *p-TS-state TS*
where
Ate-TS HOs SHOs crds = *CHO-to-TS Ate-Alg HOs SHOs* (*K o crds*)

lemmas *Ate-TS-defs* = *Ate-TS-def CHO-to-TS-def Ate-Alg-def CHOinitConfig-def Ate-initState-def*

definition
Ate-trans-step HOs ≡ $\bigcup$ *r μ*.
{((*r*, *cfg*), *Suc r*, *cfg'*)|*cfg cfg'*.
(∀ *p*. *μ p* ∈ *get-msgs* (*Ate-sendMsg r*) *cfg* (*HOs r*) (*HOs r*) *p*) ∧
(∀ *p*. *Ate-nextState r p* (*cfg p*) (*μ p*) (*cfg' p*))}

definition *CSHOnextConfig* **where**
CSHOnextConfig A r cfg HO SHO coord cfg' ≡
∀ *p*. ∃ *μ* ∈ *SHOmsgVectors A r p cfg* (*HO p*) (*SHO p*).
CnextState A r p (*cfg p*) *μ* (*coord p*) (*cfg' p*)

type-synonym *rHO* = *nat* ⇒ *process HO*

7.4.1 Refinement

definition *ate-ref-rel* :: (*opt-v-state* × *p-TS-state*)*set* **where**

```
ate-ref-rel = {(sa, (r, sc)).
  r = next-round sa
  ∧ (∀ p. decisions sa p = Ate-Defs.decide (sc p))
  ∧ majorities.opt-no-defection sa (Some o x o sc)
}

lemma decide-origin:
  assumes
    send: μ p ∈ get-msgs (Ate-sendMsg r) sc (HOs r) (HOs r) p
    and nxt: Ate-nextState r p (sc p) (μ p) (sc′ p)
    and new-dec: decide (sc′ p) ≠ decide (sc p)
  shows
    ∃ v. decide (sc′ p) = Some v ∧ {q. x (sc q) = v} ∈ majs using assms
  by(auto simp add: get-msgs-benign Ate-sendMsg-def Ate-nextState-def
    majs-def restrict-map-def elim!: order.strict-trans2 intro!: card-mono)

lemma other-values-received:
  assumes nxt: Ate-nextState r q (sc q) μq ((sc′) q)
  and muq: μq ∈ get-msgs (Ate-sendMsg r) sc (HOs r) (HOs r) q
  and vsent: card {qq. sendMsg Ate-M r qq q (sc qq) = v} > E − α
          (is card ?vsent > -)
  shows card ({qq. μq qq ≠ Some v} ∩ HOs r q) ≤ N + 2*α − E
proof −
  from nxt muq
  have ({qq. μq qq ≠ Some v} ∩ HOs r q) − (HOs r q − HOs r q)
      ⊆ {qq. sendMsg Ate-M r qq q (sc qq) ≠ v}
    (is ?notvrcvd − ?aho ⊆ ?notvsent)
    unfolding get-msgs-def SHOmsgVectors-def Ate-SHOMachine-def by auto
  hence card ?notvsent ≥ card (?notvrcvd − ?aho)
    and card (?notvrcvd − ?aho) ≥ card ?notvrcvd − card ?aho
    by (auto simp: card-mono diff-card-le-card-Diff)
  moreover
  have 1: card ?notvsent + card ?vsent = card (?notvsent ∪ ?vsent)
    by (subst card-Un-Int) auto
  have ?notvsent ∪ ?vsent = (UNIV::process set) by auto
  hence card (?notvsent ∪ ?vsent) = N by simp
  with 1 vsent have card ?notvsent ≤ N − (E + 1 − α) by auto
  ultimately
  show ?thesis using EltN Egta by auto
qed
```

If more than $E - \alpha$ processes send a value v to some process q at some round r, and if q receives more than T messages in r, then v is the most frequently received value by q in r.

lemma *mostOftenRcvd-v*:
  assumes *nxt*: *Ate-nextState r q (sc q) μq ((sc′) q)*
  and *muq*: *μq ∈ get-msgs (Ate-sendMsg r) sc (HOs r) (HOs r) q*
  and *threshold-T*: *card {qq. μq qq ≠ None} > T*
  and *threshold-E*: *card {qq. sendMsg Ate-M r qq q (sc qq) = v} > E − α*
  shows *mostOftenRcvd μq = {v}*
proof −
  from *muq* **have** *hodef:HOs r q = {qq. μq qq ≠ None}*
    unfolding *get-msgs-def SHOmsgVectors-def* **by** *auto*

  from *nxt muq threshold-E*
  have *card ({qq. μq qq ≠ Some v} ∩ HOs r q) ≤ N + 2∗α − E*
    (**is** *card ?heardnotv ≤ -*)
    by (*rule other-values-received*)
  moreover
  have *card ?heardnotv ≥ T + 1 − card {qq. μq qq = Some v}*
  proof −
    from *muq*
    have *?heardnotv = (HOs r q) − {qq. μq qq = Some v}*
      and *{qq. μq qq = Some v} ⊆ HOs r q*
      unfolding *SHOmsgVectors-def get-msgs-def* **by** *auto*
    hence *card ?heardnotv = card (HOs r q) − card {qq. μq qq = Some v}*
      by (*auto simp*: *card-Diff-subset*)
    with *hodef threshold-T* **show** *?thesis* **by** *auto*
  qed
  ultimately
  have *card {qq. μq qq = Some v} > card ?heardnotv*
    using *TNaE* **by** *auto*
  moreover
  {
    fix *w*
    assume *w*: *w ≠ v*
    with *hodef* **have** *{qq. μq qq = Some w} ⊆ ?heardnotv* **by** *auto*
   hence *card {qq. μq qq = Some w} ≤ card ?heardnotv* **by** (*auto simp*: *card-mono*)
  }
  ultimately
  have *{w. card {qq. μq qq = Some w} ≥ card {qq. μq qq = Some v}} = {v}*

```
    by force
  thus ?thesis unfolding mostOftenRcvd-def by auto
qed

lemma step-ref:
  {ate-ref-rel}
      (⋃ r v-f d-f. majorities.flv-round r v-f d-f),
      Ate-trans-step HOs
    {> ate-ref-rel}
proof(simp add: PO-rhoare-defs Ate-trans-step-def, safe)
  fix sa r sc sc′ μ
  assume
    R: (sa, r, sc) ∈ ate-ref-rel
    and send: ∀ p. μ p ∈ get-msgs (Ate-sendMsg r) sc (HOs r) (HOs r) p
    and nxt: ∀ p. Ate-nextState r p (sc p) (μ p) (sc′ p)
  note step=send nxt
  define d-f
    where d-f p = (if decide (sc′ p) ≠ decide (sc p) then decide (sc′ p) else None)
for p

  define sa′ where sa′ = (|
    opt-v-state.next-round = Suc r
    , opt-v-state.last-vote = opt-v-state.last-vote sa ++ (Some o x o sc)
    , opt-v-state.decisions = opt-v-state.decisions sa ++ d-f
    |)

  have majorities.d-guard d-f (Some o x o sc)
  proof(clarsimp simp add: majorities.d-guard-def d-f-def)
    fix p v
    assume
      Some v ≠ decide (sc p)
      decide (sc′ p) = Some v
    from this and
     decide-origin[where μ=μ and HOs=HOs and sc′=sc′, OF send[THEN spec,
of p] nxt[THEN spec, of p]]
    show quorum-process.locked-in-vf majs (Some ∘ x ∘ sc) v
      by(auto simp add: majorities.locked-in-vf-def majorities.quorum-for-def)
  qed
  hence
    (sa, sa′) ∈ majorities.flv-round r (Some o x o sc) d-f using R
```

by(*auto simp add*: *majorities.flv-round-def ate-ref-rel-def sa′-def*)
moreover have (*sa′*, *Suc r*, *sc′*) ∈ *ate-ref-rel*
proof(*unfold ate-ref-rel-def*, *safe*)
fix *p*
show *opt-v-state.decisions sa′ p* = *decide* (*sc′ p*) **using** *R nxt*[*THEN spec*, *of p*]
by(*auto simp add*: *ate-ref-rel-def sa′-def map-add-def d-f-def Ate-nextState-def split*: *option.split*)
next
show *quorum-process.opt-no-defection majs sa′* (*Some* ∘ *x* ∘ *sc′*)
proof(*clarsimp simp add*: *sa′-def majorities.opt-no-defection-def map-add-def majorities.quorum-for-def*)
fix *Q p v*
assume *Q*: *Q* ∈ *majs* **and** *Q-v*: ∀ *q* ∈ *Q*. *x* (*sc q*) = *v* **and** *p-Q*: *p* ∈ *Q*
hence *old*: *x* (*sc p*) = *v* **by** *simp*

have *v-maj*: {*q*. *x* (*sc q*) = *v*} ∈ *majs* **using** *Q Q-v*
apply(*simp add*: *majs-def*)
apply(*erule less-le-trans*, *rule card-mono*, *auto*)
done
show *x* (*sc′ p*) = *v*
proof(*cases T* < *card* {*qq*. *μ p qq* ≠ *None*})
case *True*
have
E − *α* < *card* {*qq*. *sendMsg Ate-M r qq p* (*sc qq*) = *v*} **using** *v-maj*
by(*auto simp add*: *Ate-SHOMachine-def Ate-sendMsg-def majs-def*)
from *mostOftenRcvd-v*[**where** *HOs*=*HOs* **and** *sc*=*sc* **and** *sc′*=*sc′*,
OF nxt[*THEN spec*, *of p*] *send*[*THEN spec*, *of p*] *True this*]
show *?thesis* **using** *nxt*[*THEN spec*, *of p*] *old*
by(*clarsimp simp add*: *Ate-nextState-def*)
next
case *False*
thus *?thesis* **using** *nxt*[*THEN spec*, *of p*] *old*
by(*clarsimp simp add*: *Ate-nextState-def*)
qed
qed
qed(*auto simp add*: *sa′-def*)

ultimately show
∃ *sa′*. (∃ *ra v-f d-f*. (*sa*, *sa′*) ∈ *quorum-process.flv-round majs ra v-f d-f*)

$\wedge$ (*sa'*, *Suc r*, *sc'*) $\in$ *ate-ref-rel*
by *blast*
qed

lemma *Ate-Refines-LV-VOting*:
PO-refines (*ate-ref-rel*)
majorities.flv-TS (*Ate-TS HOs HOs crds*)
proof(*rule refine-basic*)
show *init* (*Ate-TS HOs HOs crds*) $\subseteq$ *ate-ref-rel* `` *init* (*quorum-process.flv-TS majs*)
by(*auto simp add*: *Ate-TS-def CHO-to-TS-def Ate-Alg-def CHOinitConfig-def Ate-initState-def*
majorities.flv-TS-def flv-init-def majorities.opt-no-defection-def majorities.quorum-for-def'
ate-ref-rel-def)
next
show
{*ate-ref-rel*} *TS.trans* (*quorum-process.flv-TS majs*), *TS.trans* (*Ate-TS HOs HOs crds*) {> *ate-ref-rel*}
using *step-ref*
by(*simp add*: *majorities.flv-TS-defs Ate-TS-def CHO-to-TS-def Ate-Alg-def*
CSHO-trans-alt-def CHO-trans-alt Ate-trans-step-def)
qed

end — context *ate-parameters*

7.4.2 Termination

The termination proof for the algorithm is already given in the Heard-Of Model AFP entry, and we do not repeat it here.

end

8 The Same Vote Model

theory *Same-Vote*
imports *Voting*
begin

context *quorum-process* **begin**

8.1 Model definition

The system state remains the same as in the Voting model, but the voting event is changed.

definition *safe* :: *v-state* ⇒ *round* ⇒ *val* ⇒ *bool* **where**
  safe-def': *safe s r v* ≡
    ∀ *r'* < *r*. ∀ *Q* ∈ *Quorum*. ∀ *w*. (*votes s r'*) ‘ *Q* = {*Some w*} ⟶ *v* = *w*

This definition of *safe* is easier to reason about in Isabelle.

lemma *safe-def*:
  safe s r v =
    (∀ *r'* < *r*. ∀ *Q w*. *quorum-for Q w* (*votes s r'*) ⟶ *v* = *w*)
  by(*auto simp add*: *safe-def' quorum-for-def' Ball-def*)

definition *sv-round* :: *round* ⇒ *process set* ⇒ *val* ⇒ (*process*, *val*)*map* ⇒ (*v-state* × *v-state*) *set* **where**
  sv-round r S v r-decisions = {(*s*, *s'*).
    — guards
    r = *next-round s*
    ∧ (*S* ≠ {} ⟶ *safe s r v*)
    ∧ *d-guard r-decisions* (*const-map v S*)
    ∧ — actions
    s' = *s*⦇
      next-round := *Suc r*
      , *votes* := (*votes s*)(*r* := *const-map v S*)
      , *decisions* := (*decisions s* ++ *r-decisions*)
    ⦈
  }

definition *sv-trans* :: (*v-state* × *v-state*) *set* **where**
  sv-trans = (⋃ *r S v D*. *sv-round r S v D*) ∪ *Id*

definition *sv-TS* :: *v-state TS* **where**
  sv-TS = ⦇ *init* = *v-init*, *trans* = *sv-trans* ⦈

lemmas *sv-TS-defs* = *sv-TS-def v-init-def sv-trans-def*

8.2 Refinement

lemma *safe-imp-no-defection*:

safe s (*next-round s*) *v* ⟹ *no-defection s* (*const-map v S*) (*next-round s*)
by(*auto simp add*: *safe-def no-defection-def restrict-map-def const-map-def*)

lemma *const-map-quorum-locked*:
S ∈ *Quorum* ⟹ *locked-in-vf* (*const-map v S*) *v*
by(*auto simp add*: *locked-in-vf-def const-map-def quorum-for-def*)

lemma *sv-round-refines*:
{*Id*} *v-round r* (*const-map v S*) *r-decisions*, *sv-round r S v r-decisions* {> *Id*}
by(*auto simp add*: *PO-rhoare-defs sv-round-def v-round-def no-defection-empty*
dest!: *safe-imp-no-defection const-map-quorum-locked*)

lemma *Same-Vote-Refines*:
PO-refines Id v-TS sv-TS
by(*auto simp add*: *PO-refines-def sv-TS-def sv-trans-def v-TS-defs intro*!:
sv-round-refines relhoare-refl)

8.3 Invariants

definition *SV-inv3* **where**
SV-inv3 = {*s*. ∀ *r a b v w*.
votes s r a = *Some v* ∧ *votes s r b* = *Some w* ⟶ *v* = *w*
}

lemmas *SV-inv3I* = *SV-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *SV-inv3E* [*elim*] = *SV-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *SV-inv3D* = *SV-inv3-def* [*THEN setc-def-to-dest*, *rule-format*]

8.3.1 Proof of invariants

lemma *SV-inv3-v-round*:
{*SV-inv3*} *sv-round r S v D* {> *SV-inv3*}
apply(*clarsimp simp add*: *PO-hoare-defs intro*!: *SV-inv3I*)
apply(*force simp add*: *sv-round-def const-map-def restrict-map-def SV-inv3-def*)
done

lemmas *SV-inv3-event-pres* = *SV-inv3-v-round*

lemma *SV-inv3-inductive*:
init sv-TS ⊆ *SV-inv3*

{*SV-inv3*} *trans sv-TS* {> *SV-inv3*}
apply (*simp add*: *sv-TS-defs SV-inv3-def*)
by (*auto simp add*: *sv-TS-defs SV-inv3-event-pres*)

lemma *SV-inv3-invariant*: *reach sv-TS* ⊆ *SV-inv3*
by (*auto intro*!: *inv-rule-basic SV-inv3-inductive del*: *subsetI*)

This is a different characterization of *safe*, due to Lampson [4]: $safe'\ s\ r\ v = (\forall\, r'<r.\ \exists\, Q{\in}Quorum.\ \forall\, a{\in}Q.\ \forall\, w.\ votes\ s\ r'\ a = Some\ w \longrightarrow w = v)$
It is, however, strictly stronger than our characterization, since we do not at this point assume the "completeness" of our quorum system (for any set S, either S or the complement of S is a quorum), and the following is thus not provable: $s \in majorities.SV\text{-}inv3 \Longrightarrow safe'\ s = safe\ s$.

8.3.2 Transfer of abstract invariants

lemma *SV-inv1-inductive*:
init sv-TS ⊆ *Vinv1*
{*Vinv1*} *trans sv-TS* {> *Vinv1*}
apply(*rule abs-INV-init-transfer*[*OF Same-Vote-Refines Vinv1-inductive*(*1*), *simplified*])
apply(*rule abs-INV-trans-transfer*[*OF Same-Vote-Refines Vinv1-inductive*(*2*), *simplified*])
done

lemma *SV-inv1-invariant*:
reach sv-TS ⊆ *Vinv1*
by(*rule abs-INV-transfer*[*OF Same-Vote-Refines Vinv1-invariant*, *simplified*])

lemma *SV-inv2-inductive*:
init sv-TS ⊆ *Vinv2*
{*Vinv2* ∩ *Vinv1*} *trans sv-TS* {> *Vinv2*}
apply(*rule abs-INV-init-transfer*[*OF Same-Vote-Refines Vinv2-inductive*(*1*), *simplified*])
apply(*rule abs-INV-trans-transfer*[*OF Same-Vote-Refines Vinv2-inductive*(*2*), *simplified*])
done

lemma *SV-inv2-invariant*:
reach sv-TS ⊆ *Vinv2*

by(*rule abs-INV-transfer*[*OF Same-Vote-Refines Vinv2-invariant*, *simplified*])

8.3.3 Additional invariants

With Same Voting, the voted values are safe in the next round.

definition *SV-inv4* :: *v-state set* **where**
SV-inv4 = {*s*. ∀ *v a r*. *votes s r a* = *Some v* ⟶ *safe s* (*Suc r*) *v* }

lemmas *SV-inv4I* = *SV-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *SV-inv4E* [*elim*] = *SV-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *SV-inv4D* = *SV-inv4-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *SV-inv4-sv-round*:
{*SV-inv4* ∩ (*Vinv1* ∩ *Vinv2*)} *sv-round r S v D* {> *SV-inv4*}
proof(*clarsimp simp add*: *PO-hoare-defs intro*!: *SV-inv4I*)
fix *s v′ a r′ s′*
assume
step: (*s*, *s′*) ∈ *sv-round r S v D*
and *invs*: *s* ∈ *SV-inv4 s* ∈ *Vinv1 s* ∈ *Vinv2*
and *vote*: *votes s′ r′ a* = *Some v′*
thus *safe s′* (*Suc r′*) *v′*
proof(*cases r*=*r′*)
case *True*
moreover hence *safe*: *safe s′ r′ v′* **using** *step vote*
by(*force simp add*: *sv-round-def const-map-is-Some safe-def quorum-for-def*)
ultimately show *?thesis* **using** *step vote*
by(*force simp add*: *safe-def less-Suc-eq sv-round-def quorum-for-def const-map-is-Some*

dest: *quorum-non-empty*)
qed(*clarsimp simp add*: *sv-round-def safe-def Vinv2-def Vinv1-def SV-inv4-def*
intro: *Quorum-not-empty*)
qed

lemmas *SV-inv4-event-pres* = *SV-inv4-sv-round*

lemma *SV-inv4-inductive*:
init sv-TS ⊆ *SV-inv4*
{*SV-inv4* ∩ (*Vinv1* ∩ *Vinv2*)} *trans sv-TS* {> *SV-inv4*}
apply(*simp add*: *sv-TS-defs SV-inv4-def*)
by (*auto simp add*: *sv-TS-defs SV-inv4-event-pres*)

lemma *SV-inv4-invariant*: *reach sv-TS* $\subseteq$ *SV-inv4*
by (*rule inv-rule-incr*)
(*auto intro*: *SV-inv4-inductive SV-inv2-invariant SV-inv1-invariant del*: *subsetI*)

end

end

9 The Observing Quorums Model

theory *Observing-Quorums*
imports *Same-Vote*
begin

9.1 Model definition

The state adds one field to the Voting model state:

record *obsv-state* = *v-state* +
obs :: *round* $\Rightarrow$ (*process*, *val*) *map*

For the observation mechanism to work, we need monotonicity of quorums.

context *mono-quorum* **begin**

definition *obs-safe*
where
obs-safe r s v $\equiv$ ($\forall r' < r.\ \exists p.$ *obs s r' p* $\in$ {*None*, *Some v*})

definition *obsv-round*
:: *round* $\Rightarrow$ *process set* $\Rightarrow$ *val* $\Rightarrow$ (*process*, *val*)*map* $\Rightarrow$ *process set* $\Rightarrow$ (*obsv-state* $\times$ *obsv-state*) *set*
where
obsv-round r S v r-decisions Os = {(*s*, *s'*).
— guards
r = *next-round s*
$\wedge$ ($S \neq$ {} $\longrightarrow$ *obs-safe r s v*)
$\wedge$ *d-guard r-decisions* (*const-map v S*)
$\wedge$ ($S \in$ *Quorum* $\longrightarrow$ *Os* = *UNIV*)
$\wedge$ (*Os* $\neq$ {} $\longrightarrow$ $S \neq$ {})

$\wedge$ — actions
$s' = s(|$
$next\text{-}round := Suc\ r$
$, votes := (votes\ s)(r := const\text{-}map\ v\ S)$
$, decisions := decisions\ s ++ r\text{-}decisions$
$, obs := (obs\ s)(r := const\text{-}map\ v\ Os)$
$|)$
}

definition *obsv-trans* :: (*obsv-state* × *obsv-state*) *set* **where**
$obsv\text{-}trans = (\bigcup r\ S\ v\ d\text{-}f\ Os.\ obsv\text{-}round\ r\ S\ v\ d\text{-}f\ Os) \cup Id$

definition *obsv-init* :: *obsv-state set* **where**
$obsv\text{-}init = \{ (|\ next\text{-}round = 0,\ votes = \lambda r\ a.\ None,\ decisions = Map.empty,\ obs = \lambda r\ a.\ None\ |) \}$

definition *obsv-TS* :: *obsv-state TS* **where**
$obsv\text{-}TS = (|\ init = obsv\text{-}init,\ trans = obsv\text{-}trans\ |)$

lemmas *obsv-TS-defs* = *obsv-TS-def obsv-init-def obsv-trans-def*

9.2 Invariants

definition *OV-inv1* **where**
$OV\text{-}inv1 = \{s.\ \forall r\ Q\ v.\ quorum\text{-}for\ Q\ v\ (votes\ s\ r) \longrightarrow (\forall Q' \in Quorum.\ quorum\text{-}for\ Q'\ v\ (obs\ s\ r))\}$

lemmas *OV-inv1I* = *OV-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *OV-inv1E* [*elim*] = *OV-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *OV-inv1D* = *OV-inv1-def* [*THEN setc-def-to-dest*, *rule-format*]

9.2.1 Proofs of invariants

lemma *OV-inv1-obsv-round*:
$\{OV\text{-}inv1\}\ obsv\text{-}round\ r\ S\ v\ d\text{-}f\ Ob\ \{> OV\text{-}inv1\}$
proof(*clarsimp simp add*: *PO-hoare-defs intro*!: *OV-inv1I*)
fix $v'\ s\ s'\ Q\ Q'\ r'$
assume
Q: $quorum\text{-}for\ Q\ v'\ (votes\ s'\ r')$
and *inv*: $s \in OV\text{-}inv1$

and *step*: $(s, s') \in$ *obsv-round r S v d-f Ob*
and *quorum*: $Q' \in$ *Quorum*
from *Q inv*[*THEN OV-inv1D*] *step quorum*
show *quorum-for* Q' v' (*obs* s' r')
proof(*cases* $r'=r$)
case *True*
with *step* **and** *Q* **have** $S \in$ *Quorum*
by(*fastforce simp add*: *obsv-round-def obs-safe-def quorum-for-def const-map-is-Some ball-conj-distrib subset-eq*[*symmetric*] *intro*: *mono-quorum*[**where** $Q'=S$])

thus *?thesis* **using** *step inv*[*THEN OV-inv1D*] *Q quorum*
by(*clarsimp simp add*: *obsv-round-def obs-safe-def quorum-for-def const-map-is-Some ball-conj-distrib subset-eq*[*symmetric*] *dest*!: *quorum-non-empty*)
qed(*clarsimp simp add*: *obsv-round-def obs-safe-def quorum-for-def*)
qed

lemma *OV-inv1-inductive*:
init obsv-TS $\subseteq$ *OV-inv1*
{*OV-inv1*} *trans obsv-TS* {> *OV-inv1*}
apply (*simp add*: *obsv-TS-defs OV-inv1-def*)
apply (*auto simp add*: *obsv-TS-defs OV-inv1-obsv-round quorum-for-def dest*: *empty-not-quorum*)
done

lemma *quorum-for-const-map*:
(*quorum-for Q w* (*const-map v S*)) = ($Q \in$ *Quorum* $\wedge$ $Q \subseteq S$ $\wedge$ $w = v$)
by(*auto simp add*: *quorum-for-def const-map-is-Some dest*: *quorum-non-empty*)

9.3 Refinement

definition *obsv-ref-rel* **where**
obsv-ref-rel $\equiv$ {(*sa*, *sc*).
sa = *v-state.truncate sc*
}

lemma *obsv-round-refines*:
{*obsv-ref-rel* $\cap$ *UNIV* $\times$ *OV-inv1*} *sv-round r S v dec-f*, *obsv-round r S v dec-f Ob* {> *obsv-ref-rel*}
apply(*clarsimp simp add*: *PO-rhoare-defs sv-round-def obsv-round-def safe-def obsv-ref-rel-def*

v-state.truncate-def obs-safe-def quorum-for-def OV-inv1-def)
by (*metis UNIV-I UNIV-quorum option.distinct*(*1*) *option.inject*)

lemma *Observable-Refines*:
PO-refines (*obsv-ref-rel* ∩ *UNIV* × *OV-inv1*) *sv-TS obsv-TS*
proof(*rule refine-using-invariants*)
show *init obsv-TS* ⊆ *obsv-ref-rel* `` *init sv-TS*
by(*auto simp add*: *PO-refines-def sv-TS-defs obsv-TS-defs obsv-ref-rel-def*
v-state.truncate-def)
next
show {*obsv-ref-rel* ∩ *UNIV* × *OV-inv1*} *trans sv-TS*, *trans obsv-TS* {> *obsv-ref-rel*}
by(*auto simp add*: *PO-refines-def sv-TS-defs obsv-TS-defs intro*!:
obsv-round-refines relhoare-refl)
qed(*auto intro*: *OV-inv1-inductive del*: *subsetI*)

9.4 Additional invariants

definition *OV-inv2* **where**
OV-inv2 = {*s*. ∀ *r* ≥ *next-round s*. *obs s r* = *Map.empty* }

lemmas *OV-inv2I* = *OV-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *OV-inv2E* [*elim*] = *OV-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *OV-inv2D* = *OV-inv2-def* [*THEN setc-def-to-dest*, *rule-format*]

definition *OV-inv3* **where**
OV-inv3 = {*s*. ∀ *r p v*. *obs s r p* = *Some v* ⟶
obs-safe r s v}

lemmas *OV-inv3I* = *OV-inv3-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *OV-inv3E* [*elim*] = *OV-inv3-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *OV-inv3D* = *OV-inv3-def* [*THEN setc-def-to-dest*, *rule-format*]

definition *OV-inv4* **where**
OV-inv4 = {*s*. ∀ *r p q v w*. *obs s r p* = *Some v* ∧ *obs s r q* = *Some w* ⟶
w = *v*}

lemmas *OV-inv4I* = *OV-inv4-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *OV-inv4E* [*elim*] = *OV-inv4-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *OV-inv4D* = *OV-inv4-def* [*THEN setc-def-to-dest*, *rule-format*]

9.4.1 Proofs of additional invariants

lemma *OV-inv2-inductive*:
init obsv-TS $\subseteq$ *OV-inv2*
{*OV-inv2*} *trans obsv-TS* {> *OV-inv2*}
by(*auto simp add*: *PO-hoare-defs OV-inv2-def obsv-TS-defs obsv-round-def const-map-is-Some*)

lemma *SVinv3-inductive*:
init obsv-TS $\subseteq$ *SV-inv3*
{*SV-inv3*} *trans obsv-TS* {> *SV-inv3*}
by(*auto simp add*: *PO-hoare-defs SV-inv3-def obsv-TS-defs obsv-round-def const-map-is-Some*)

lemma *OV-inv3-obsv-round*:
{*OV-inv3* $\cap$ *OV-inv2*} *obsv-round r S v D Ob* {> *OV-inv3*}
proof(*clarsimp simp add*: *PO-hoare-defs intro*!: *OV-inv3I*)
fix *s s′ r-w p w*
assume *Assms*:
obs s′ r-w p = Some w
s $\in$ *OV-inv3*
(*s*, *s′*) $\in$ *obsv-round r S v D Ob*
s $\in$ *OV-inv2*
hence *s′* $\in$ *OV-inv2*
by(*force simp add*: *obsv-TS-defs intro*: *OV-inv2-inductive*(*2*)[*THEN hoareD*,
OF ‹*s* $\in$ *OV-inv2*›])
hence *r-w* $\leq$ *next-round s′* **using** *Assms*
by(*auto simp add*: *OV-inv2-def intro*!: *leI*)
hence *r-w-le*: *r-w* $\leq$ *next-round s* **using** *Assms*
by(*auto simp add*: *obsv-round-def le-Suc-eq*)
show *obs-safe r-w s′ w*
proof(*cases r-w = next-round s*)
case *True*
thus *?thesis* **using** *Assms*
by(*auto simp add*: *obsv-round-def const-map-is-Some obs-safe-def*)
next
case *False*
hence *r-w* < *next-round s* **using** *r-w-le*
by(*metis less-le*)
moreover have $\forall$ *r′*. *r′* $\neq$ *next-round s* $\longrightarrow$ *obs s′ r′ = obs s r′* **using** *Assms*(*3*)
by(*auto simp add*: *obsv-round-def*)
ultimately have

$\forall r' \leq r\text{-}w.\ obs\ s'\ r' = obs\ s\ r'$
by(*auto*)
thus *?thesis* **using** *Assms*
by(*auto simp add*: *OV-inv3-def obs-safe-def*)
qed
qed

lemma *OV-inv3-inductive*:
init obsv-TS $\subseteq$ *OV-inv3*
$\{$*OV-inv3* $\cap$ *OV-inv2*$\}$ *trans obsv-TS* $\{>$ *OV-inv3*$\}$
apply(*auto simp add*: *obsv-TS-def obsv-trans-def intro*: *OV-inv3-obsv-round*)
apply(*auto simp add*: *obsv-init-def OV-inv3-def*)
done

lemma *OV-inv4-inductive*:
init obsv-TS $\subseteq$ *OV-inv4*
$\{$*OV-inv4*$\}$ *trans obsv-TS* $\{>$ *OV-inv4*$\}$
by(*auto simp add*: *PO-hoare-defs OV-inv4-def obsv-TS-defs obsv-round-def const-map-is-Some*)

end

end

10 The Optimized Observing Quorums Model

theory *Observing-Quorums-Opt*
imports *Observing-Quorums*
begin

10.1 Model definition

record *opt-obsv-state* =
next-round :: *round*
decisions :: (*process*, *val*)*map*
last-obs :: (*process*, *val*)*map*

context *mono-quorum*
begin

definition *opt-obs-safe* **where**
$opt\text{-}obs\text{-}safe\ obs\text{-}f\ v \equiv \exists p.\ obs\text{-}f\ p \in \{None,\ Some\ v\}$

definition *olv-round* **where**
$olv\text{-}round\ r\ S\ v\ r\text{-}decisions\ Ob \equiv \{(s,\ s').$
— guards
$r = next\text{-}round\ s$
$\wedge\ (S \neq \{\} \longrightarrow opt\text{-}obs\text{-}safe\ (last\text{-}obs\ s)\ v)$
$\wedge\ (S \in Quorum \longrightarrow Ob = UNIV)$
$\wedge\ d\text{-}guard\ r\text{-}decisions\ (const\text{-}map\ v\ S)$
$\wedge\ (Ob \neq \{\} \longrightarrow S \neq \{\})$
$\wedge$ — actions
$s' = s(\!|$
$next\text{-}round := Suc\ r$
$,\ decisions := decisions\ s\ {+}{+}\ r\text{-}decisions$
$,\ last\text{-}obs := last\text{-}obs\ s\ {+}{+}\ const\text{-}map\ v\ Ob$
$|\!)$
$\}$

definition *olv-init* **where**
$olv\text{-}init = \{\ (\!|\ next\text{-}round = 0,\ decisions = Map.empty,\ last\text{-}obs = Map.empty\ |\!)\ \}$

definition *olv-trans* :: (*opt-obsv-state* × *opt-obsv-state*) *set* **where**
$olv\text{-}trans = (\bigcup r\ S\ v\ D\ Ob.\ olv\text{-}round\ r\ S\ v\ D\ Ob) \cup Id$

definition *olv-TS* :: *opt-obsv-state TS* **where**
$olv\text{-}TS = (\!|\ init = olv\text{-}init,\ trans = olv\text{-}trans\ |\!)$

lemmas *olv-TS-defs* = *olv-TS-def olv-init-def olv-trans-def*

10.2 Refinement

definition *olv-ref-rel* **where**
$olv\text{-}ref\text{-}rel \equiv \{(sa,\ sc).$
$next\text{-}round\ sc = v\text{-}state.next\text{-}round\ sa$
$\wedge\ decisions\ sc = v\text{-}state.decisions\ sa$
$\wedge\ last\text{-}obs\ sc = map\text{-}option\ snd\ o\ process\text{-}mru\ (obsv\text{-}state.obs\ sa)$
$\}$

lemma *OV-inv2-finite-map-graph*:
  $s \in OV\text{-}inv2 \Longrightarrow finite\ (map\text{-}graph\ (case\text{-}prod\ (obsv\text{-}state.obs\ s)))$
  apply(*rule finite-dom-finite-map-graph*)
  apply(*rule finite-subset*[**where** $B=\{0..< v\text{-}state.next\text{-}round\ s\} \times UNIV$])
   apply(*auto simp add*: *OV-inv2-def dom-def not-le*[*symmetric*])
  done

lemma *OV-inv2-finite-obs-set*:
  $s \in OV\text{-}inv2 \Longrightarrow finite\ (vote\text{-}set\ (obsv\text{-}state.obs\ s)\ Q)$
  apply(*drule OV-inv2-finite-map-graph*)
  apply(*clarsimp simp add*: *map-graph-def fun-graph-def vote-set-def*)
  apply(*erule finite-surj*[**where** $f=\lambda((r, a), v). (r, v)$])
  by(*force simp add*: *image-def*)

lemma *olv-round-refines*:
  $\{olv\text{-}ref\text{-}rel \cap (OV\text{-}inv2 \cap OV\text{-}inv3 \cap OV\text{-}inv4) \times UNIV\}\ obsv\text{-}round\ r\ S\ v\ D\ Ob,\ olv\text{-}round\ r\ S\ v\ D\ Ob\ \{>olv\text{-}ref\text{-}rel\}$
proof(*clarsimp simp add*: *PO-rhoare-defs*)
  fix *sa* :: *obsv-state* **and** *sc* sc'
  assume
    ainv: $sa \in OV\text{-}inv2\ sa \in OV\text{-}inv3\ sa \in OV\text{-}inv4$
    and *step*: $(sc, sc') \in olv\text{-}round\ r\ S\ v\ D\ Ob$
    and *R*: $(sa, sc) \in olv\text{-}ref\text{-}rel$

  — Abstract guard.
  have $S \neq \{\} \longrightarrow obs\text{-}safe\ (v\text{-}state.next\text{-}round\ sa)\ sa\ v$
  proof(*rule impI*, *rule ccontr*)
    assume *S-nonempty*: $S \neq \{\}$ **and** *no-Q*: $\neg\ obs\text{-}safe\ (v\text{-}state.next\text{-}round\ sa)\ sa\ v$
    from *no-Q* **obtain** *r-w w* **where**
      r-w: $r\text{-}w < v\text{-}state.next\text{-}round\ sa$
      and *all-obs*: $\forall p.\ obsv\text{-}state.obs\ sa\ r\text{-}w\ p = Some\ w$
      and *diff*: $w \neq v$ **using** *ainv*(*3*)[*THEN OV-inv4D*]
      by(*simp add*: *obs-safe-def*) (*metis*)
    from *diff step R* **obtain** *p* **where**
       p-w: $S \neq \{\} \longrightarrow (map\text{-}option\ snd \circ process\text{-}mru\ (obsv\text{-}state.obs\ sa))\ p \neq Some\ w$
      by (*simp add*: *opt-obs-safe-def quorum-for-def olv-round-def olv-ref-rel-def*)

```
    (metis option.distinct(1) option.sel snd-conv)
  from all-obs have nempty: vote-set (obsv-state.obs sa) {p} ≠ {}
    by(auto simp add:  vote-set-def)
   then obtain r-w' w' where w': process-mru (obsv-state.obs sa) p = Some
(r-w', w')
    by (simp add: process-mru-def mru-of-set-def)
      (metis option-Max-by-def surj-pair)
  hence max: (r-w', w') = Max-by fst (vote-set (obsv-state.obs sa) {p})
    by(auto simp add: process-mru-def mru-of-set-def option-Max-by-def)
  hence w'-obs: (r-w', w') ∈ vote-set (obsv-state.obs sa) {p}
    using Max-by-in[OF OV-inv2-finite-obs-set[OF ainv(1), of {p}] nempty]
    by fastforce
  have r-w ≤ r-w' using all-obs
    apply −
      apply(rule Max-by-ge[OF OV-inv2-finite-obs-set[OF ainv(1), of {p}], of
(r-w, w) fst,
                  simplified max[symmetric], simplified])
    apply(auto simp add: quorum-for-def vote-set-def)
    done
  moreover have w' ≠ w using p-w w' S-nonempty
    by(auto)
  ultimately have r-w < r-w' using all-obs w'-obs
    apply(elim le-neq-implies-less)
    apply(auto simp add: quorum-for-def vote-set-def)
    done
  thus False using ainv(2)[THEN OV-inv3D] w'-obs all-obs ‹w' ≠ w›
    by(fastforce simp add: vote-set-def obs-safe-def)
qed

— Action refinement.
moreover have
  (map-option snd ∘ process-mru (obsv-state.obs sa)) ++ const-map v Ob =
    map-option snd ∘ process-mru ((obsv-state.obs sa)(v-state.next-round sa :=
const-map v Ob))
proof−
  from ‹sa ∈ OV-inv2›[THEN OV-inv2D]
  have empty: ∀ r'≥v-state.next-round sa. obsv-state.obs sa r' = Map.empty
    by simp
  show ?thesis
   by(auto simp add: map-add-def const-map-def restrict-map-def process-mru-new-votes[OF
```

empty])
  qed

 ultimately show $\exists$ *sa'*. (*sa*, *sa'*) $\in$ *obsv-round r S v D Ob* $\wedge$ (*sa'*, *sc'*) $\in$ *olv-ref-rel*

    using *R step*
    by(*clarsimp simp add*: *obsv-round-def olv-round-def olv-ref-rel-def*)

qed

lemma *OLV-Refines*:
  PO-refines (*olv-ref-rel* $\cap$ (*OV-inv2* $\cap$ *OV-inv3* $\cap$ *OV-inv4*) $\times$ *UNIV*) *obsv-TS olv-TS*
proof(*rule refine-using-invariants*)
  show *init olv-TS* $\subseteq$ *olv-ref-rel* '' *init obsv-TS*
      by(*auto simp add*: *obsv-TS-defs olv-TS-defs olv-ref-rel-def process-mru-def mru-of-set-def*
      vote-set-def option-Max-by-def intro!: *ext*)
next
  show
    {*olv-ref-rel* $\cap$ (*OV-inv2* $\cap$ *OV-inv3* $\cap$ *OV-inv4*) $\times$ *UNIV*} *TS.trans obsv-TS*, *TS.trans olv-TS* {> *olv-ref-rel*}
    by(*auto simp add*: *PO-refines-def obsv-TS-defs olv-TS-defs*
      intro!: *olv-round-refines*)
qed (*auto intro*: *OV-inv2-inductive OV-inv3-inductive OV-inv4-inductive*
  OV-inv2-inductive(*1*)[*THEN subsetD*] *OV-inv3-inductive*(*1*)[*THEN subsetD*]
  OV-inv4-inductive(*1*)[*THEN subsetD*])

end

end

11 Two-Step Observing Quorums Model

theory *Two-Step-Observing*
imports ../*Observing-Quorums-Opt* ../*Two-Steps*
begin

To make the coming proofs of concrete algorithms easier, in this model we split the *olv-round* into two steps.

11.1 Model definition

record *tso-state* = *opt-obsv-state* +
  r-votes :: *process* ⇒ *val option*

context *mono-quorum*
begin

definition *tso-round0*
  :: *round* ⇒ *process set* ⇒ *val* ⇒ (*tso-state* × *tso-state*)*set*
  where
  tso-round0 r S v ≡ {(*s*, *s*′).
    — guards
    r = *next-round s*
    ∧ *two-step r* = *0*
    ∧ (*S* ≠ {} ⟶ *opt-obs-safe* (*last-obs s*) *v*)
    — actions
    ∧ *s*′ = *s*⦇
      next-round := *Suc r*
      , *r-votes* := *const-map v S*
    ⦈
  }

definition *obs-guard* :: (*process*, *val*)*map* ⇒ (*process*, *val*)*map* ⇒ *bool* **where**
  obs-guard r-obs r-v ≡ ∀ *p*.
    (∀ *v*. *r-obs p* = *Some v* ⟶ (∃ *q*. *r-v q* = *Some v*))
    ∧ (*dom r-v* ∈ *Quorum* ⟶ (∃ *q* ∈ *dom r-v*. *r-obs p* = *r-v q*))

definition *tso-round1*
  :: *round* ⇒ (*process*, *val*)*map* ⇒ (*process*, *val*)*map* ⇒ (*tso-state* × *tso-state*)*set*
  where
  tso-round1 r r-decisions r-obs ≡ {(*s*, *s*′).
    — guards
    r = *next-round s*
    ∧ *two-step r* = *1*
    ∧ *d-guard r-decisions* (*r-votes s*)
    ∧ *obs-guard r-obs* (*r-votes s*)
    — actions
    ∧ *s*′ = *s*⦇
      next-round := *Suc r*

, *decisions* := *decisions s* ++ *r-decisions*
, *last-obs* := *last-obs s* ++ *r-obs*
⦈
}

definition *tso-init* **where**
tso-init = { ⦇ *next-round* = *0*, *decisions* = *Map.empty*, *last-obs* = *Map.empty*, *r-votes* = *Map.empty* ⦈ }

definition *tso-trans* :: (*tso-state* × *tso-state*) *set* **where**
tso-trans = ($\bigcup$ *r S v*. *tso-round0 r S v*) ∪ ($\bigcup$ *r d-f o-f*. *tso-round1 r d-f o-f*) ∪ *Id*

definition *tso-TS* :: *tso-state TS* **where**
tso-TS = ⦇ *init* = *tso-init*, *trans* = *tso-trans* ⦈

lemmas *tso-TS-defs* = *tso-TS-def tso-init-def tso-trans-def*

11.2 Refinement

definition *basic-rel* :: (*opt-obsv-state* × *tso-state*)*set* **where**
basic-rel = {(*sa*, *sc*).
next-round sa = *two-phase* (*next-round sc*)
∧ *last-obs sc* = *last-obs sa*
∧ *decisions sc* = *decisions sa*
}

definition *step0-rel* :: (*opt-obsv-state* × *tso-state*)*set* **where**
step0-rel = *basic-rel*

definition *step1-add-rel* :: (*opt-obsv-state* × *tso-state*)*set* **where**
step1-add-rel = {(*sa*, *sc*). ∃ *S v*.
r-votes sc = *const-map v S*
∧ (*S* ≠ {} ⟶ *opt-obs-safe* (*last-obs sc*) *v*)
}

definition *step1-rel* :: (*opt-obsv-state* × *tso-state*)*set* **where**
step1-rel = *basic-rel* ∩ *step1-add-rel*

definition *tso-ref-rel* :: (*opt-obsv-state* × *tso-state*)*set* **where**
tso-ref-rel ≡ {(*sa*, *sc*).

(*two-step* (*next-round sc*) = *0* ⟶ (*sa*, *sc*) ∈ *step0-rel*)
∧ (*two-step* (*next-round sc*) = *1* ⟶
(*sa*, *sc*) ∈ *step1-rel*
∧ (∃ *sc′ r S v*. (*sc′*, *sc*) ∈ *tso-round0 r S v* ∧ (*sa*, *sc′*) ∈ *step0-rel*))
}

lemma *const-map-equality*:
(*const-map v S* = *const-map v′ S′*) = (*S* = *S′* ∧ (*S* = {} ∨ *v* = *v′*))
apply(*simp add*: *const-map-def restrict-map-def*)
by (*metis equals0D option.distinct*(*2*) *option.inject subsetI subset-antisym*)

lemma *rhoare-skipI*:
⟦ ⋀*sa sc sc′*. ⟦ (*sa*, *sc*) ∈ *Pre*; (*sc*, *sc′*) ∈ *Tc* ⟧ ⟹ (*sa*, *sc′*) ∈ *Post* ⟧ ⟹ {*Pre*} *Id*, *Tc* {>*Post*}
by(*auto simp add*: *PO-rhoare-defs*)

lemma *tso-round0-refines*:
{*tso-ref-rel*} *Id*, *tso-round0 r S v* {>*tso-ref-rel*}
apply(*rule rhoare-skipI*)
apply(*auto simp add*: *tso-ref-rel-def basic-rel-def step1-rel-def*
step1-add-rel-def step0-rel-def tso-round0-def const-map-equality conj-disj-distribR
ex-disj-distrib two-step-phase-Suc)
done

lemma *tso-round1-refines*:
{*tso-ref-rel*} ⋃*r S v dec-f Ob*. *olv-round r S v dec-f Ob*, *tso-round1 r dec-f o-f* {>*tso-ref-rel*}
proof(*clarsimp simp add*: *PO-rhoare-defs*)
fix *sa sc* **and** *sc′*
assume
R: (*sa*, *sc*) ∈ *tso-ref-rel*
and *step1*: (*sc*, *sc′*) ∈ *tso-round1 r dec-f o-f*

hence *step-r*: *two-step r* = *1* **and** *r-next*: *next-round sc* = *r* **by** (*auto simp add*: *tso-round1-def*)
then obtain *r0 sc0 S0 v0* **where**
R0: (*sa*, *sc0*) ∈ *step0-rel* **and** *step0*: (*sc0*, *sc*) ∈ *tso-round0 r0 S0 v0*
using *R*
by(*auto simp add*: *tso-ref-rel-def*)

```
  from step-r r-next R obtain S v where
    v: r-votes sc = const-map v S
    and safe: S ≠ {} ⟶ opt-obs-safe (last-obs sc) v
    by(auto simp add: tso-ref-rel-def step1-rel-def step1-add-rel-def)

  define sa' where sa' = sa⦇
      next-round := Suc (next-round sa)
      , decisions := decisions sa ++ dec-f
      , last-obs := last-obs sa ++ const-map v (dom o-f)
    ⦈

  have S ∈ Quorum ⟶ dom o-f = UNIV using step1 v
    by(auto simp add: tso-round1-def obs-guard-def const-map-def)
  moreover have o-f ≠ Map.empty ⟶ S ≠ {} using step1 v
    by(auto simp add: tso-round1-def obs-guard-def dom-const-map)
  ultimately have
    abs-step:
   (sa, sa') ∈ olv-round (next-round sa) S v dec-f (dom o-f) using R safe v step-r
r-next step1
    by(clarsimp simp add: tso-ref-rel-def step1-rel-def basic-rel-def sa'-def
      olv-round-def tso-round1-def)

  from v have post-rel: (sa', sc') ∈ tso-ref-rel using R step1
    apply(clarsimp simp add: tso-round0-def tso-round1-def
      step0-rel-def basic-rel-def  sa'-def tso-ref-rel-def two-step-phase-Suc o-def
      const-map-is-Some const-map-is-None const-map-equality obs-guard-def
      intro!: arg-cong2[where f=map-add, OF refl])
    apply(auto simp add: const-map-def restrict-map-def intro!: ext)
    done

  from abs-step post-rel show
   ∃ sa'. (∃ r' S' w dec-f' Ob'. (sa, sa') ∈ olv-round r' S' w dec-f' Ob') ∧ (sa', sc')
∈ tso-ref-rel
    by blast
qed

lemma TS-Observing-Refines:
  PO-refines tso-ref-rel olv-TS tso-TS
  apply(auto simp add: PO-refines-def olv-TS-defs tso-TS-defs
    intro!: tso-round0-refines tso-round1-refines)
```

apply(*auto simp add*: *tso-ref-rel-def step0-rel-def basic-rel-def tso-init-def quorum-for-def*
dest: *empty-not-quorum*)
done

11.3 Invariants

definition *TSO-inv1* **where**
TSO-inv1 = {*s. two-step* (*next-round s*) = *Suc 0* ⟶
(∃ *v.* ∀ *p w. r-votes s p* = *Some w* ⟶ *w* = *v*)}

lemmas *TSO-inv1I* = *TSO-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *TSO-inv1E* [*elim*] = *TSO-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *TSO-inv1D* = *TSO-inv1-def* [*THEN setc-def-to-dest*, *rule-format*]

definition *TSO-inv2* **where**
TSO-inv2 = {*s. two-step* (*next-round s*) = *Suc 0* ⟶
(∀ *p v.* (*r-votes s p* = *Some v* ⟶ (∃ *q. last-obs s q* ∈ {*None*, *Some v*})))}

lemmas *TSO-inv2I* = *TSO-inv2-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *TSO-inv2E* [*elim*] = *TSO-inv2-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *TSO-inv2D* = *TSO-inv2-def* [*THEN setc-def-to-dest*, *rule-format*]

11.3.1 Proofs of invariants

lemma *TSO-inv1-inductive*:
init tso-TS ⊆ *TSO-inv1*
{*TSO-inv1*} *TS.trans tso-TS* {> *TSO-inv1*}
by(*auto simp add*: *TSO-inv1-def tso-TS-defs PO-hoare-def*
tso-round0-def tso-round1-def const-map-is-Some two-step-phase-Suc)

lemma *TSO-inv1-invariant*:
reach tso-TS ⊆ *TSO-inv1*
by(*intro inv-rule-basic TSO-inv1-inductive*)

lemma *TSO-inv2-inductive*:
init tso-TS ⊆ *TSO-inv2*
{*TSO-inv2*} *TS.trans tso-TS* {> *TSO-inv2*}
by(*auto simp add*: *TSO-inv2-def tso-TS-defs PO-hoare-def*
opt-obs-safe-def tso-round0-def tso-round1-def const-map-is-Some two-step-phase-Suc)

lemma *TSO-inv2-invariant*:
  reach tso-TS ⊆ *TSO-inv2*
  by(*intro inv-rule-basic TSO-inv2-inductive*)

end

end

12 The UniformVoting Algorithm

theory *Uv-Defs*
imports *Heard-Of.HOModel ../Consensus-Types ../Quorums*
begin

The contents of this file have been taken almost verbatim from the Heard Of Model AFP entry. The only difference is that the types have been changed.

12.1 Model of the algorithm

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* =
  last-obs :: *'val* — current value held by process
  agreed-vote :: *'val option* — value the process voted for, if any
  decide :: *'val option* — value the process has decided on, if any

Possible messages sent during the execution of the algorithm, and characteristic predicates to distinguish types of messages.

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

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

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

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

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

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

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

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

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

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

In step 0, each process sends its current *last-obs* value.

It updates its *last-obs* 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 *agreed-vote* field to v.

definition *send0* **where**
$send0\ r\ p\ q\ st \equiv Val\ (last\text{-}obs\ st)$

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

In step 1, each process sends its current *last-obs* and *agreed-vote* values.

definition *send1* **where**
send1 r p q st ≡ *ValVote* (*last-obs st*) (*agreed-vote st*)

definition *valVoteRcvd* **where**
— processes from which values and votes were received
valVoteRcvd (*msgs* :: *process* ⇀ *'val msg*) ≡
{*q* . ∃ *z v*. *msgs q* = *Some* (*ValVote z v*)}

definition *smallestValNoVoteRcvd* **where**
smallestValNoVoteRcvd (*msgs*::*process* ⇀ (*'val*::*linorder*) *msg*) ≡
Min {*v*. ∃ *q*. *msgs q* = *Some* (*ValVote v None*)}

definition *someVoteRcvd* **where**
— set of processes from which some vote was received
someVoteRcvd (*msgs* :: *process* ⇀ *'val msg*) ≡
{ *q* . *q* ∈ *msgRcvd msgs* ∧ *isValVote* (*the* (*msgs q*)) ∧ *getvote* (*the* (*msgs q*)) ≠ *None* }

definition *identicalVoteRcvd* **where**
identicalVoteRcvd (*msgs* :: *process* ⇀ *'val msg*) *v* ≡
∀ *q* ∈ *msgRcvd msgs*. *isValVote* (*the* (*msgs q*)) ∧ *getvote* (*the* (*msgs q*)) = *Some v*

definition *x-update* **where**
x-update st msgs st' ≡
(∃ *q* ∈ *someVoteRcvd msgs* . *last-obs st'* = *the* (*getvote* (*the* (*msgs q*))))
∨ *someVoteRcvd msgs* = {} ∧ *last-obs st'* = *smallestValNoVoteRcvd msgs*

definition *dec-update* **where**
dec-update st msgs st' ≡
(∃ *v*. *identicalVoteRcvd msgs v* ∧ *decide st'* = *Some v*)
∨ ¬(∃ *v*. *identicalVoteRcvd msgs v*) ∧ *decide st'* = *decide st*

definition *next1* **where**
next1 r p st msgs st' ≡
x-update st msgs st'
∧ *dec-update st msgs st'*
∧ *agreed-vote st'* = *None*

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

definition *UV-sendMsg* **where**
UV-sendMsg (*r*::*nat*) ≡ *if step r = 0 then send0 r else send1 r*

definition *UV-nextState* **where**
UV-nextState r ≡ *if step r = 0 then next0 r else next1 r*

definition (**in** *quorum-process*) *UV-commPerRd* **where**
UV-commPerRd HOrs ≡ ∀ *p*. *HOrs p* ∈ *Quorum*

definition *UV-commGlobal* **where**
UV-commGlobal HOs ≡ ∃ *r*. ∀ *p q*. *HOs r p* = *HOs r q*

12.2 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 (**in** *quorum-process*) *UV-HOMachine* **where**
UV-HOMachine = (|
CinitState = (λ*p st crd*. *UV-initState p st*),
sendMsg = *UV-sendMsg*,
CnextState = (λ*r p st msgs crd st*′. *UV-nextState r p st msgs st*′),
HOcommPerRd = *UV-commPerRd*,
HOcommGlobal = *UV-commGlobal*
|)

abbreviation (**in** *quorum-process*)
UV-M ≡ (*UV-HOMachine*::(*process*, ′*val*::*linorder pstate*, ′*val msg*) *HOMachine*)

end

12.3 Proofs

type-synonym *uv-TS-state* = (*nat* × (*process* ⇒ (*val pstate*)))

axiomatization where *val-linorder*:
OFCLASS(*val*, *linorder-class*)

instance *val* :: *linorder* **by** (*rule val-linorder*)

lemma *two-step-step*:
step = *two-step*
phase = *two-phase*
by(*auto simp add*: *step-def two-step-def phase-def two-phase-def*)

context *mono-quorum*
begin

definition *UV-Alg* :: (*process*, *val pstate*, *val msg*)*CHOAlgorithm* **where**
UV-Alg = *CHOAlgorithm.truncate UV-M*

definition *UV-TS* ::
(*round* ⇒ *process HO*) ⇒ (*round* ⇒ *process HO*) ⇒ (*round* ⇒ *process*) ⇒ *uv-TS-state TS*
where
UV-TS HOs SHOs crds = *CHO-to-TS UV-Alg HOs SHOs* (*K o crds*)

lemmas *UV-TS-defs* = *UV-TS-def CHO-to-TS-def UV-Alg-def CHOinitConfig-def UV-initState-def*

type-synonym *rHO* = *nat* ⇒ *process HO*

definition *UV-trans-step*
where
UV-trans-step HOs SHOs nxt-f snd-f stp ≡ ⋃ *r μ*.
{((*r*, *cfg*), (*Suc r*, *cfg'*))|*cfg cfg'*. *step r* = *stp* ∧ (∀ *p*.
μ p ∈ *get-msgs* (*snd-f r*) *cfg* (*HOs r*) (*SHOs r*) *p*
∧ *nxt-f r p* (*cfg p*) (*μ p*) (*cfg' p*)
)}

lemma *step-less-D*:
0 < *step r* ⟹ *step r* = *Suc 0*
by(*auto simp add*: *step-def*)

lemma *UV-trans*:

CSHO-trans-alt UV-sendMsg (λr *p st msgs crd st'*. *UV-nextState r p st msgs st'*)
HOs SHOs crds =
UV-trans-step HOs SHOs next0 send0 0
∪ *UV-trans-step HOs SHOs next1 send1 1*

proof
show *CSHO-trans-alt UV-sendMsg* (λr *p st msgs crd*. *UV-nextState r p st msgs*)
HOs SHOs crds
⊆ *UV-trans-step HOs SHOs next0 send0 0* ∪ *UV-trans-step HOs SHOs next1*
send1 1
by(*force simp add*: *CSHO-trans-alt-def UV-sendMsg-def UV-nextState-def UV-trans-step-def*

K-def dest!: *step-less-D*)
next
show *UV-trans-step HOs SHOs next0 send0 0* ∪
UV-trans-step HOs SHOs next1 send1 1
⊆ *CSHO-trans-alt UV-sendMsg*
(λr *p st msgs crd*. *UV-nextState r p st msgs*) *HOs SHOs crds*
by(*force simp add*: *CSHO-trans-alt-def UV-sendMsg-def UV-nextState-def UV-trans-step-def*)
qed

12.3.1 Invariants

definition *UV-inv1*
:: *uv-TS-state set*
where
UV-inv1 = {(*r*, *s*).
two-step r = *0* ⟶ (∀ *p*. *agreed-vote* (*s p*) = *None*)
}

lemmas *UV-inv1I* = *UV-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *UV-inv1E* [*elim*] = *UV-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *UV-inv1D* = *UV-inv1-def* [*THEN setc-def-to-dest*, *rule-format*]

lemma *UV-inv1-inductive*:
init (*UV-TS HOs SHOs crds*) ⊆ *UV-inv1*
{*UV-inv1*} *TS.trans* (*UV-TS HOs SHOs crds*) {> *UV-inv1*}
by(*auto simp add*: *UV-inv1-def UV-TS-defs CHO-trans-alt UV-trans PO-hoare-def*

UV-HOMachine-def CHOAlgorithm.truncate-def UV-trans-step-def *all-conj-distrib two-step-phase-Suc two-step-step next1-def*)

lemma *UV-inv1-invariant*:
reach (*UV-TS HOs SHOs crds*) $\subseteq$ *UV-inv1*
by(*intro inv-rule-basic UV-inv1-inductive*)

12.3.2 Refinement

definition *ref-rel* :: (*tso-state* $\times$ *uv-TS-state*)*set* **where**
ref-rel $\equiv$ {(*sa*, (*r*, *sc*)).
r = *next-round sa*
$\wedge$ (*step r* = *1* $\longrightarrow$ *r-votes sa* = *agreed-vote o sc*)
$\wedge$ ($\forall$ *p v*. *last-obs* (*sc p*) = *v* $\longrightarrow$ ($\exists$ *q*. *opt-obsv-state.last-obs sa q* $\in$ {*None*, *Some v*}))
$\wedge$ *decisions sa* = *decide o sc*
}

Agreement for UV only holds if the communication predicates hold

context
fixes
HOs :: *nat* $\Rightarrow$ *process* $\Rightarrow$ *process set*
and *rho* :: *nat* $\Rightarrow$ *process* $\Rightarrow$ *'val pstate*
assumes *global*: *UV-commGlobal HOs*
and *per-rd*: $\forall$ *r*. *UV-commPerRd* (*HOs r*)
and *run*: *HORun fA rho HOs*
begin

lemma *HOs-intersect*:
HOs r p $\cap$ *HOs r' q* $\neq$ {} **using** *per-rd*
apply(*simp add*: *UV-commPerRd-def*)
apply(*blast dest*: *qintersect*)
done

lemma *HOs-nonempty*:
HOs r p $\neq$ {}
using *HOs-intersect*
by *blast*

lemma *vote-origin*:

assumes
send: $\forall p.\ \mu\ p \in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall p.$ *next0 r p* (*cfg p*) (μ *p*) (*cfg′ p*)
and *inv*: (*r*, *cfg*) $\in$ *UV-inv1*
and *step-r*: *two-step r* = *0*
shows
agreed-vote (*cfg′ p*) = *Some v* $\longleftrightarrow$ ($\forall q \in$ *HOs r p*. *last-obs* (*cfg q*) = *v*)
using *send*[*THEN spec*, **where** *x*=*p*] *step*[*THEN spec*, **where** *x*=*p*] *inv step-r HOs-nonempty*
by(*auto simp add*: *next0-def get-msgs-benign send0-def msgRcvd-def o-def restrict-map-def*)

lemma *same-new-vote*:
assumes
send: $\forall p.\ \mu\ p \in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall p.$ *next0 r p* (*cfg p*) (μ *p*) (*cfg′ p*)
and *inv*: (*r*, *cfg*) $\in$ *UV-inv1*
and *step-r*: *two-step r* = *0*
obtains *v* **where** $\forall p\ w.$ *agreed-vote* (*cfg′ p*) = *Some w* $\longrightarrow$ *w* = *v*
proof(*cases* $\exists p\ v.$ *agreed-vote* (*cfg′ p*) = *Some v*)
case *True*
assume *asm*: $\bigwedge v.\ \forall p\ w.$ *agreed-vote* (*cfg′ p*) = *Some w* $\longrightarrow$ *w* = *v* $\Longrightarrow$ *thesis*
from *True* **obtain** *p v* **where** *agreed-vote* (*cfg′ p*) = *Some v* **by** *auto*

hence $\forall p\ w.$ *agreed-vote* (*cfg′ p*) = *Some w* $\longrightarrow$ *w* = *v* (**is** *?LV*(*v*))
using *vote-origin*[*OF send step inv step-r*] *HOs-intersect*
by(*force*)

from *asm*[*OF this*] **show** *?thesis* **.**
qed(*auto*)

lemma *x-origin1*:
assumes
send: $\forall p.\ \mu\ p \in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall p.$ *next0 r p* (*cfg p*) (μ *p*) (*cfg′ p*)
and *step-r*: *two-step r* = *0*
and *last-obs*: *last-obs* (*cfg′ p*) = *v*
shows
$\exists q.$ *last-obs* (*cfg q*) = *v*
proof−

```
 have smallestValRcvd (μ p) ∈ {v. ∃ q. μ p q = Some (Val v)} (is smallestValRcvd
?msgs ∈ ?vals)
  unfolding smallestValRcvd-def
  proof(rule Min-in)
    have ?vals ⊆ getval ` ((the ∘ ?msgs) ` (HOs r p))
      using send[THEN spec, where x=p]
      by (auto simp: image-def get-msgs-benign restrict-map-def send0-def)
    thus finite ?vals by (auto simp: finite-subset)
  next
    from send[THEN spec, where x=p]
    show ?vals ≠ {} using HOs-nonempty[of r p]
      by (auto simp: image-def get-msgs-benign restrict-map-def send0-def)
  qed
  hence v ∈ ?vals using last-obs step[THEN spec, of p]
    by(auto simp add: next0-def all-conj-distrib)
  thus ?thesis using send[THEN spec, of p]
    by(auto simp add: get-msgs-benign send0-def restrict-map-def)
qed

lemma step0-ref:
  {ref-rel ∩ UNIV × UV-inv1} ⋃r S v. tso-round0 r S v,
    UV-trans-step HOs HOs next0 send0 0 {> ref-rel}
proof(clarsimp simp add: PO-rhoare-defs UV-trans-step-def two-step-step all-conj-distrib)
  fix sa r cfg μ cfg′
  assume
    R: (sa, (r, cfg)) ∈ ref-rel
    and step-r: two-step r = 0
    and send: ∀ p. μ p ∈ get-msgs (send0 r) cfg (HOs r) (HOs r) p
    and step: ∀ p. next0 r p (cfg p) (μ p) (cfg′ p)
    and inv: (r, cfg) ∈ UV-inv1

  from R have next-r: next-round sa = r
    by(simp add: ref-rel-def)

  from HOs-nonempty send have ∀ p. ∃ q. q ∈ msgRcvd (μ p)
   by(fastforce simp add: get-msgs-benign send0-def msgRcvd-def restrict-map-def)
  with step have same-dec: decide o cfg′ = decide o cfg
    apply(simp add: next0-def o-def)
   by (metis pstate.select-convs(3) pstate.surjective pstate.update-convs(1) pstate.update-convs(2))
```

define *S* **where** $S = \{p.\ \exists\, v.\ \textit{agreed-vote}\ (\textit{cfg}'\ p) = \textit{Some}\ v\}$
from *same-new-vote*[*OF send step inv step-r*]
obtain *v* **where** v: $\forall\, p \in S.\ \textit{agreed-vote}\ (\textit{cfg}'\ p) = \textit{Some}\ v$
by(*simp add*: *S-def*) (*metis*)
hence *vote-const-map*: *agreed-vote o cfg′* = *const-map v S*
by(*auto simp add*: *S-def const-map-def restrict-map-def intro*!: *ext*)

note *x-origin* = *x-origin1*[*OF send step step-r*]

define *sa′* **where** *sa′* = *sa*⦇ *next-round* := *Suc r*, *r-votes* := *const-map v S* ⦈

have $\forall\, p.\ p \in S \longrightarrow$ *opt-obs-safe* (*opt-obsv-state.last-obs sa*) *v*
using *vote-origin*[*OF send step inv step-r*] *R per-rd*[*THEN spec*, *of r*] *v*
apply(*clarsimp simp add*: *UV-commPerRd-def opt-obs-safe-def ref-rel-def*)
by *metis*

hence (*sa*, *sa′*) ∈ *tso-round0 r S v* **using** *next-r step-r v R*
vote-origin[*OF send step inv step-r*]
by(*auto simp add*: *tso-round0-def sa′-def all-conj-distrib*)

moreover have (*sa′*, *Suc r*, *cfg′*) ∈ *ref-rel* **using** *step send v R same-dec step-r next-r*
apply(*clarsimp simp add*: *ref-rel-def sa′-def two-step-step two-step-phase-Suc vote-const-map*)
by (*metis x-origin*)
ultimately show
∃ *sa′*. (∃ *r S v*. (*sa*, *sa′*) ∈ *tso-round0 r S v*) ∧ (*sa′*, *Suc r*, *cfg′*) ∈ *ref-rel*
by *blast*
qed

lemma *x-origin2*:
assumes
send: ∀ *p*. *μ p* ∈ *get-msgs* (*send1 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: ∀ *p*. *next1 r p* (*cfg p*) (*μ p*) (*cfg′ p*)
and *step-r*: *two-step r* = *Suc 0*
and *last-obs*: *last-obs* (*cfg′ p*) = *v*
shows
(∃ *q*. *last-obs* (*cfg q*) = *v*) ∨ (∃ *q*. *agreed-vote* (*cfg q*) = *Some v*)
proof(*cases* ∀ *q* ∈ *HOs r p*. ∃ *w*. *μ p q* = *Some* (*ValVote w None*))

```
  case True
 hence empty: someVoteRcvd (μ p) = {} using send[THEN spec, of p] HOs-nonempty[of
r p]
    by(auto simp add: someVoteRcvd-def msgRcvd-def isValVote-def
      get-msgs-benign send1-def restrict-map-def)
  have smallestValNoVoteRcvd (μ p) ∈ {v. ∃ q. μ p q = Some (ValVote v None)}

    (is smallestValNoVoteRcvd ?msgs ∈ ?vals)
  unfolding smallestValNoVoteRcvd-def
  proof(rule Min-in)
    have ?vals ⊆ getval ` ((the ∘ ?msgs) ` (HOs r p))
      using send[THEN spec, where x=p]
      by (auto simp: image-def get-msgs-benign restrict-map-def send0-def)
    thus finite ?vals by (auto simp: finite-subset)
  next
    from send[THEN spec, where x=p] True HOs-nonempty[of r p]
    show ?vals ≠ {}
      by (auto simp: image-def get-msgs-benign restrict-map-def send1-def)
  qed
  hence v ∈ ?vals using empty step[THEN spec, of p] last-obs
    by(auto simp add: next1-def x-update-def)
  thus ?thesis using send[THEN spec, of p]
    by(auto simp add: get-msgs-benign restrict-map-def send1-def)
next
  case False
 hence someVoteRcvd (μ p) ≠ {} using send[THEN spec, of p] HOs-nonempty[of
r p]
    by(auto simp add: someVoteRcvd-def msgRcvd-def isValVote-def
      get-msgs-benign send1-def restrict-map-def)
   hence ∃ q ∈ someVoteRcvd (μ p). v = the (getvote (the (μ p q))) using
step[THEN spec, of p] last-obs
    by(auto simp add: next1-def x-update-def)
  thus ?thesis using send[THEN spec, of p]
    by(auto simp add: next1-def x-update-def someVoteRcvd-def isValVote-def
      send1-def get-msgs-benign msgRcvd-def restrict-map-def)
qed

definition D where
  D cfg cfg' ≡ {p. decide (cfg' p) ≠ decide (cfg p) }
```

lemma *decide-origin*:
assumes
send: $\forall p.\ \mu\ p \in$ *get-msgs* (*send1 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall p.$ *next1 r p* (*cfg p*) (μ *p*) (*cfg' p*)
and *step-r*: *two-step r* = *Suc 0*
shows
D cfg cfg' $\subseteq$ {*p*. $\exists v.$ *decide* (*cfg' p*) = *Some v* $\wedge$ ($\forall q \in$ *HOs r p*. *agreed-vote* (*cfg q*) = *Some v*)}
using *assms*
by(*fastforce simp add*: *D-def next1-def get-msgs-benign send1-def msgRcvd-def o-def restrict-map-def*
x-update-def dec-update-def identicalVoteRcvd-def all-conj-distrib someVoteRcvd-def isValVote-def
smallestValNoVoteRcvd-def)

lemma *step1-ref*:
{*ref-rel* $\cap$ (*TSO-inv1* $\cap$ *TSO-inv2*) $\times$ *UNIV*} $\bigcup$ *r d-f o-f*. *tso-round1 r d-f o-f*,
UV-trans-step HOs HOs next1 send1 (*Suc 0*) {> *ref-rel*}
proof(*clarsimp simp add*: *PO-rhoare-defs UV-trans-step-def two-step-step all-conj-distrib*)
fix *sa r cfg* μ **and** *cfg'* :: *process* $\Rightarrow$ *val pstate*
assume
R: (*sa*, (*r*, *cfg*)) $\in$ *ref-rel*
and *step-r*: *two-step r* = *Suc 0*
and *send*: $\forall p.\ \mu\ p \in$ *get-msgs* (*send1 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall p.$ *next1 r p* (*cfg p*) (μ *p*) (*cfg' p*)
and *ainv*: *sa* $\in$ *TSO-inv1*
and *ainv2*: *sa* $\in$ *TSO-inv2*

from *R* **have** *next-r*: *next-round sa* = *r*
by(*simp add*: *ref-rel-def*)

define *S* **where** *S* = {*p*. $\exists v.$ *agreed-vote* (*cfg p*) = *Some v*}
from *R* **obtain** *v* **where** *v*: $\forall p \in S.$ *agreed-vote* (*cfg p*) = *Some v* **using** *ainv step-r*
by(*auto simp add*: *ref-rel-def TSO-inv1-def S-def two-step-step*)

define *Ob* **where** *Ob* = {*p*. *last-obs* (*cfg' p*) = *v*}
define *o-f* **where** *o-f p* = (*if S* $\in$ *Quorum then Some v else None*) **for** *p* :: *process*

define *dec-f* **where** *dec-f p* = (*if p* ∈ *D cfg cfg′ then decide* (*cfg′ p*) *else None*)
for *p*

{
fix *p w*
assume *agreed-vote* (*cfg p*) = *Some w*
hence *w* = *v* **using** *v*
by(*unfold S-def*, *auto*)
} **note** *v′*=*this*

have *d-guard*: *d-guard dec-f* (*agreed-vote* ∘ *cfg*) **using** *per-rd*[*THEN spec*, *of r*]
by(*fastforce simp add*: *d-guard-def locked-in-vf-def quorum-for-def dec-f-def UV-commPerRd-def dest*!: *decide-origin*[*OF send step step-r*, *THEN subsetD*])

have *dom* (*agreed-vote* ∘ *cfg*) ∈ *Quorum* ⟶ *Ob* = *UNIV*
proof(*auto simp add*: *Ob-def*)
fix *p*
assume *Q*: *dom* (*agreed-vote* ∘ *cfg*) ∈ *Quorum* (**is** *?Q* ∈ *Quorum*)
hence *?Q* ∩ *HOs r p* ≠ {} **using** *per-rd*[*THEN spec*, *of r*]
by(*auto simp add*: *UV-commPerRd-def dest*: *qintersect*)
hence *someVoteRcvd* (*μ p*) ≠ {} **using** *send*[*THEN spec*, *of p*]
by(*force simp add*: *someVoteRcvd-def get-msgs-benign msgRcvd-def restrict-map-def*

isValVote-def send1-def)
moreover have ∀ *q* ∈ *someVoteRcvd* (*μ p*). ∃ *x′*. *μ p q* = *Some* (*ValVote x′* (*Some v*))
using *send*[*THEN spec*, *of p*]
by(*auto simp add*: *someVoteRcvd-def get-msgs-benign msgRcvd-def restrict-map-def isValVote-def send1-def dest*: *v′*)
ultimately show *last-obs* (*cfg′ p*) = *v* **using** *step*[*THEN spec*, *of p*]
by(*auto simp add*: *next1-def x-update-def*)
qed
note *Ob-UNIV*=*this*[*rule-format*]

have *obs-guard*: *obs-guard o-f* (*agreed-vote* ∘ *cfg*)
apply(*auto simp add*: *obs-guard-def o-f-def S-def dom-def dest*: *v′ Ob-UNIV quorum-non-empty*)
apply (*metis S-def all-not-in-conv empty-not-quorum v*)
done

define *sa′* **where** *sa′* = *sa*⦇
next-round := *Suc* (*next-round sa*)
, *decisions* := *decisions sa* ++ *dec-f*
, *opt-obsv-state.last-obs* := *opt-obsv-state.last-obs sa* ++ *o-f*
⦈

— Abstract step
have *abs-step*: (*sa*, *sa′*) ∈ *tso-round1 r dec-f o-f* **using** *next-r step-r R d-guard obs-guard*
by(*auto simp add*: *tso-round1-def sa′-def ref-rel-def two-step-step*)

— Relation preserved
have ∀ *p*. ((*decide* ∘ *cfg*) ++ *dec-f*) *p* = *decide* (*cfg′ p*)
proof
fix *p*
show ((*decide* ∘ *cfg*) ++ *dec-f*) *p* = *decide* (*cfg′ p*) **using** *step*[*THEN spec*, *of p*]
by(*auto simp add*: *dec-f-def D-def next1-def dec-update-def map-add-def*)
qed
note *dec-rel*=*this*[*rule-format*]

have ∀ *p*. (∃ *q*. *o-f q* = *None* ∧ *opt-obsv-state.last-obs sa q* = *None*
∨ (*opt-obsv-state.last-obs sa* ++ *o-f*) *q* = *Some* (*last-obs* (*cfg′ p*)))
proof(*intro allI impI*, *cases S* ∈ *Quorum*)
fix *p*
case *True*
hence *last-obs* (*cfg′ p*) = *v* **using** *Ob-UNIV*
by(*auto simp add*: *S-def Ob-def dom-def*)
thus (∃ *q*. *o-f q* = *None* ∧ *opt-obsv-state.last-obs sa q* = *None*
∨ (*opt-obsv-state.last-obs sa* ++ *o-f*) *q* = *Some* (*last-obs* (*cfg′ p*)))
using *True*
by(*auto simp add*: *o-f-def*)
next
fix *p*
case *False*
hence *empty*: *o-f* = *Map.empty*
by(*auto simp add*: *o-f-def*)
note *last-obs* = *x-origin2*[*OF send step step-r refl*, *of p*]
thus (∃ *q*. *o-f q* = *None* ∧ *opt-obsv-state.last-obs sa q* = *None*

```
      ∨ (opt-obsv-state.last-obs sa ++ o-f) q = Some (last-obs (cfg′ p)))
    proof(elim disjE exE)
      fix q
      assume last-obs (cfg q) = last-obs (cfg′ p)
      from this[symmetric] show ?thesis using R step-r empty
        by(simp add: ref-rel-def two-step-step)
    next
      fix q
      assume agreed-vote (cfg q) = Some (last-obs (cfg′ p))
      from this[symmetric] show ?thesis using R ainv2 step-r empty
        apply(auto simp add: ref-rel-def two-step-step TSO-inv2-def)
        by metis
    qed
  qed
  note obs-rel=this[rule-format]

  have post-rel:
    (sa′, Suc r, cfg′) ∈ ref-rel using step send next-r R step-r
    by(auto simp add: sa′-def ref-rel-def two-step-step
      two-step-phase-Suc dec-rel obs-rel)

  from abs-step post-rel show
    ∃ sa′. (∃ r d-f o-f. (sa, sa′) ∈ tso-round1 r d-f o-f) ∧ (sa′, Suc r, cfg′) ∈ ref-rel
    by blast
qed

lemma UV-Refines-votes:
  PO-refines (ref-rel ∩ (TSO-inv1 ∩ TSO-inv2) × UV-inv1)
    tso-TS (UV-TS HOs HOs crds)
proof(rule refine-using-invariants)
  show init (UV-TS HOs HOs crds) ⊆ ref-rel ‘‘ init tso-TS
   by(auto simp add: UV-TS-defs UV-HOMachine-def CHOAlgorithm.truncate-def

     tso-TS-defs ref-rel-def tso-init-def Let-def o-def)
next
  show
    {ref-rel ∩ (TSO-inv1 ∩ TSO-inv2) × UV-inv1} TS.trans tso-TS,
      TS.trans (UV-TS HOs HOs crds) {> ref-rel}
   apply(simp add: tso-TS-defs UV-TS-defs UV-HOMachine-def CHOAlgorithm.truncate-def)
    apply(auto simp add: CHO-trans-alt UV-trans intro!: step0-ref step1-ref)
```

done
qed(*auto intro*!: *TSO-inv1-inductive TSO-inv2-inductive UV-inv1-inductive*)

end

end

12.3.3 Termination

As the model of the algorithm is taken verbatim from the HO Model AFP, we do not repeat the termination proof here and refer to that AFP entry.

end

13 The Ben-Or Algorithm

theory *BenOr-Defs*
imports *Heard-Of.HOModel ../Consensus-Types ../Quorums ../Two-Steps*
begin

consts *coin* :: *round* ⇒ *process* ⇒ *val*

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

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

definition *isVote* **where** *isVote m* ≡ ∃ *v*. *m* = *Vote v*

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

fun *getvote* **where**
getvote (*Vote v*) = *v*

fun *getval* **where**

getval (*Val z*) = *z*

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

definition *msgRcvd* **where** — processes from which some message was received
msgRcvd (*msgs*:: *process* ⇀ ′*val msg*) = {*q* . *msgs q* ≠ *None*}

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

definition *next0* **where**
next0 r p st (*msgs*::*process* ⇀ ′*val msg*) *st′* ≡
    (∃ *v*. (∀ *q* ∈ *msgRcvd msgs*. *msgs q* = *Some* (*Val v*))
        ∧ *st′* = *st* (| *vote* := *Some v* |))
  ∨ ¬(∃ *v*. ∀ *q* ∈ *msgRcvd msgs*. *msgs q* = *Some* (*Val v*))
    ∧ *st′* = *st* (| *vote* := *None* |)

definition *send1* **where**
send1 r p q st ≡ *Vote* (*vote st*)

definition *someVoteRcvd* **where**
— set of processes from which some vote was received
someVoteRcvd (*msgs* :: *process* ⇀ ′*val msg*) ≡
  { *q* . *q* ∈ *msgRcvd msgs* ∧ *isVote* (*the* (*msgs q*)) ∧ *getvote* (*the* (*msgs q*)) ≠ *None* }

definition *identicalVoteRcvd* **where**
identicalVoteRcvd (*msgs* :: *process* ⇀ ′*val msg*) *v* ≡
  ∀ *q* ∈ *msgRcvd msgs*. *isVote* (*the* (*msgs q*)) ∧ *getvote* (*the* (*msgs q*)) = *Some v*

definition *x-update* **where**
x-update r p msgs st′ ≡
  (∃ *q* ∈ *someVoteRcvd msgs* . *x st′* = *the* (*getvote* (*the* (*msgs q*))))
∨ *someVoteRcvd msgs* = {} ∧ *x st′* = *coin r p*

definition *dec-update* **where**
dec-update st msgs st′ ≡
  (∃ *v*. *identicalVoteRcvd msgs v* ∧ *decide st′* = *Some v*)
∨ ¬(∃ *v*. *identicalVoteRcvd msgs v*) ∧ *decide st′* = *decide st*

definition *next1* **where**
  next1 r p st msgs st′ ≡
    x-update r p msgs st′
  ∧ *dec-update st msgs st′*
  ∧ *vote st′* = *None*

definition *BenOr-sendMsg* **where**
  BenOr-sendMsg (*r::nat*) ≡ *if two-step r* = *0 then send0 r else send1 r*

definition *BenOr-nextState* **where**
  BenOr-nextState r ≡ *if two-step r* = *0 then next0 r else next1 r*

13.1 The *Ben-Or* Heard-Of machine

definition (**in** *quorum-process*) *BenOr-commPerRd* **where**
  BenOr-commPerRd HOrs ≡ ∀ *p. HOrs p* ∈ *Quorum*

definition *BenOr-commGlobal* **where**
  BenOr-commGlobal HOs ≡ ∃ *r. two-step r* = *1*
   ∧ (∀ *p q. HOs r p* = *HOs r q* ∧ (*coin r p* :: *val*) = *coin r q*)

definition (**in** *quorum-process*) *BenOr-HOMachine* **where**
  BenOr-HOMachine = (|
   CinitState = (*λp st crd. BenOr-initState p st*),
   sendMsg = *BenOr-sendMsg*,
   CnextState = (*λr p st msgs crd st′. BenOr-nextState r p st msgs st′*),
   HOcommPerRd = *BenOr-commPerRd*,
   HOcommGlobal = *BenOr-commGlobal*
  |)

abbreviation (**in** *quorum-process*)
  BenOr-M ≡ (*BenOr-HOMachine*::(*process*, *val pstate*, *val msg*) *HOMachine*)

end

13.2 Proofs

type-synonym *ben-or-TS-state* = (*nat* × (*process* ⇒ (*val pstate*)))

consts
val0 :: *val*
val1 :: *val*

Ben-Or works only on binary values.

axiomatization where
val-exhaust: $v = val0 \lor v = val1$
and *val-diff*: $val0 \neq val1$

context *mono-quorum*
begin

definition *BenOr-Alg* :: (*process*, *val pstate*, *val msg*)*CHOAlgorithm* **where**
BenOr-Alg = *CHOAlgorithm.truncate BenOr-M*

definition *BenOr-TS* ::
(*round* ⇒ *process HO*) ⇒ (*round* ⇒ *process HO*) ⇒ (*round* ⇒ *process*) ⇒ *ben-or-TS-state TS*
where
BenOr-TS HOs SHOs crds = *CHO-to-TS BenOr-Alg HOs SHOs* (*K o crds*)

lemmas *BenOr-TS-defs* = *BenOr-TS-def CHO-to-TS-def BenOr-Alg-def CHOinit-Config-def*
BenOr-initState-def

type-synonym *rHO* = *nat* ⇒ *process HO*

definition *BenOr-trans-step*
where
BenOr-trans-step HOs SHOs nxt-f snd-f stp ≡ ⋃ *r* μ.
{((*r*, *cfg*), (*Suc r*, *cfg'*))|*cfg cfg'*. *two-step r* = *stp* ∧ (∀ *p*.
μ *p* ∈ *get-msgs* (*snd-f r*) *cfg* (*HOs r*) (*SHOs r*) *p*
∧ *nxt-f r p* (*cfg p*) (μ *p*) (*cfg' p*)
)}

lemma *two-step-less-D*:
0 < *two-step r* ⟹ *two-step r* = *Suc 0*

by(*auto simp add*: *two-step-def*)

lemma *BenOr-trans*:
CSHO-trans-alt BenOr-sendMsg ($\lambda r\ p\ st\ msgs\ crd\ st'$. *BenOr-nextState r p st msgs st'*) *HOs SHOs crds* =
BenOr-trans-step HOs SHOs next0 send0 0
∪ *BenOr-trans-step HOs SHOs next1 send1 1*

proof
show *CSHO-trans-alt BenOr-sendMsg* ($\lambda r\ p\ st\ msgs\ crd$. *BenOr-nextState r p st msgs*) *HOs SHOs crds*
⊆ *BenOr-trans-step HOs SHOs next0 send0 0* ∪ *BenOr-trans-step HOs SHOs next1 send1 1*
by(*force simp add*: *CSHO-trans-alt-def BenOr-sendMsg-def BenOr-nextState-def BenOr-trans-step-def*
K-def dest!: *two-step-less-D*)
next
show *BenOr-trans-step HOs SHOs next0 send0 0* ∪
BenOr-trans-step HOs SHOs next1 send1 1
⊆ *CSHO-trans-alt BenOr-sendMsg*
($\lambda r\ p\ st\ msgs\ crd$. *BenOr-nextState r p st msgs*) *HOs SHOs crds*
by(*force simp add*: *CSHO-trans-alt-def BenOr-sendMsg-def BenOr-nextState-def BenOr-trans-step-def*)
qed

definition *BenOr-A* = *CHOAlgorithm.truncate BenOr-M*

13.2.1 Refinement

Agreement for BenOr only holds if the communication predicates hold

context
fixes
HOs :: *nat* ⇒ *process* ⇒ *process set*
and *rho* :: *nat* ⇒ *process* ⇒ *val pstate*
assumes *comm-global*: *BenOr-commGlobal HOs*
and *per-rd*: ∀ *r*. *BenOr-commPerRd* (*HOs r*)
and *run*: *HORun BenOr-A rho HOs*
begin

definition *no-vote-diff* **where**
no-vote-diff sc p $\equiv$ *vote* (*sc p*) = *None* $\longrightarrow$
($\exists$ *q q'*. *x* (*sc q*) $\neq$ *x* (*sc q'*))

definition *ref-rel* :: (*tso-state* $\times$ *ben-or-TS-state*)*set* **where**
ref-rel $\equiv$ {(*sa*, (*r*, *sc*)).
r = *next-round sa*
$\wedge$ (*two-step r* = *1* $\longrightarrow$ *r-votes sa* = *vote o sc*)
$\wedge$ (*two-step r* = *1* $\longrightarrow$ ($\forall$ *p*. *no-vote-diff sc p*))
$\wedge$ ($\forall$ *p v*. *x* (*sc p*) = *v* $\longrightarrow$ ($\exists$ *q*. *last-obs sa q* $\in$ {*None*, *Some v*}))
$\wedge$ *decisions sa* = *decide o sc*
}

lemma *HOs-intersect*:
HOs r p $\cap$ *HOs r' q* $\neq$ {} **using** *per-rd*
apply(*simp add*: *BenOr-commPerRd-def*)
apply(*blast dest*: *qintersect*)
done

lemma *HOs-nonempty*:
HOs r p $\neq$ {}
using *HOs-intersect*
by *blast*

lemma *vote-origin*:
assumes
send: $\forall$ *p*. μ *p* $\in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall$ *p*. *next0 r p* (*cfg p*) (μ *p*) (*cfg' p*)
and *step-r*: *two-step r* = *0*
shows
vote (*cfg' p*) = *Some v* $\longleftrightarrow$ ($\forall$ *q* $\in$ *HOs r p*. *x* (*cfg q*) = *v*)
using *send*[*THEN spec*, **where** *x*=*p*] *step*[*THEN spec*, **where** *x*=*p*] *step-r* *HOs-nonempty*
by(*auto simp add*: *next0-def get-msgs-benign send0-def msgRcvd-def o-def restrict-map-def*)

lemma *same-new-vote*:
assumes
send: $\forall$ *p*. μ *p* $\in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall$ *p*. *next0 r p* (*cfg p*) (μ *p*) (*cfg' p*)

and *step-r*: *two-step r* = *0*
obtains *v* **where** $\forall$ *p w*. *vote* (*cfg'* *p*) = *Some w* $\longrightarrow$ *w* = *v*
proof(*cases* $\exists$ *p v*. *vote* (*cfg'* *p*) = *Some v*)
case *True*
assume *asm*: $\bigwedge$*v*. $\forall$ *p w*. *vote* (*cfg'* *p*) = *Some w* $\longrightarrow$ *w* = *v* $\Longrightarrow$ *thesis*
from *True* **obtain** *p v* **where** *vote* (*cfg'* *p*) = *Some v* **by** *auto*

hence $\forall$ *p w*. *vote* (*cfg'* *p*) = *Some w* $\longrightarrow$ *w* = *v* (**is** *?LV*(*v*))
using *vote-origin*[*OF send step step-r*] *HOs-intersect*
by(*force*)

from *asm*[*OF this*] **show** *?thesis* **.**
qed(*auto*)

lemma *no-x-change*:
assumes
send: $\forall$ *p*. μ *p* $\in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall$ *p*. *next0 r p* (*cfg p*) (μ *p*) (*cfg'* *p*)
and *step-r*: *two-step r* = *0*
shows
x (*cfg'* *p*) = *x* (*cfg p*)
using *send*[*THEN spec*, **where** *x*=*p*] *step*[*THEN spec*, **where** *x*=*p*] *step-r* *HOs-nonempty*
by(*auto simp add*: *next0-def get-msgs-benign send0-def msgRcvd-def o-def restrict-map-def*)

lemma *no-vote*:
assumes
send: $\forall$ *p*. μ *p* $\in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall$ *p*. *next0 r p* (*cfg p*) (μ *p*) (*cfg'* *p*)
and *step-r*: *two-step r* = *0*
shows
no-vote-diff cfg' *p*
unfolding *no-vote-diff-def*
proof
assume
vote (*cfg'* *p*) = *None*
hence ($\exists$ *q q'*. *x* (*cfg q*) $\neq$ *x* (*cfg q'*))
using *send*[*THEN spec*, **where** *x*=*p*] *step*[*THEN spec*, **where** *x*=*p*] *step-r* *HOs-nonempty*

apply(*clarsimp simp add*: *next0-def get-msgs-benign send0-def msgRcvd-def o-def restrict-map-def*)
by *metis*
thus ($\exists$ *q q'*. *x* (*cfg' q*) $\neq$ *x* (*cfg' q'*))
using *no-x-change*[*OF send step step-r*]
by(*simp*)
qed

lemma *step0-ref*:
{*ref-rel*} $\bigcup$ *r S v*. *tso-round0 r S v*,
BenOr-trans-step HOs HOs next0 send0 0 {> *ref-rel*}
proof(*clarsimp simp add*: *PO-rhoare-defs BenOr-trans-step-def all-conj-distrib*)
fix *sa r cfg* μ *cfg'*
assume
R: (*sa*, (*r*, *cfg*)) $\in$ *ref-rel*
and *step-r*: *two-step r* = *0*
and *send*: $\forall$ *p*. μ *p* $\in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: $\forall$ *p*. *next0 r p* (*cfg p*) (μ *p*) (*cfg' p*)

from *R* **have** *next-r*: *next-round sa* = *r*
by(*simp add*: *ref-rel-def*)

from *HOs-nonempty send* **have** $\forall$ *p*. $\exists$ *q*. *q* $\in$ *msgRcvd* (μ *p*)
by(*fastforce simp add*: *get-msgs-benign send0-def msgRcvd-def restrict-map-def*)
with *step* **have** *same-dec*: *decide o cfg'* = *decide o cfg*
apply(*simp add*: *next0-def o-def*)
by (*metis pstate.select-convs*(*3*) *pstate.surjective pstate.update-convs*(*2*))

define *S* **where** *S* = {*p*. $\exists$ *v*. *vote* (*cfg' p*) = *Some v*}
from *same-new-vote*[*OF send step step-r*]
obtain *v* **where** *v*: $\forall$ *p* $\in$ *S*. *vote* (*cfg' p*) = *Some v*
by(*simp add*: *S-def*) (*metis*)
hence *vote-const-map*: *vote o cfg'* = *const-map v S*
by(*auto simp add*: *S-def const-map-def restrict-map-def intro*!: *ext*)

define *sa'* **where** *sa'* = *sa*⦇ *next-round* := *Suc r*, *r-votes* := *const-map v S* ⦈

have $\forall$ *p*. *p* $\in$ *S* $\longrightarrow$ *opt-obs-safe* (*last-obs sa*) *v*
using *vote-origin*[*OF send step step-r*] *R per-rd*[*THEN spec*, *of r*] *v*

apply(*clarsimp simp add*: *BenOr-commPerRd-def opt-obs-safe-def ref-rel-def*)
by *metis*

hence $(sa, sa') \in$ *tso-round0 r S v* **using** *next-r step-r v R vote-origin*[*OF send step step-r*]
by(*auto simp add*: *tso-round0-def sa'-def all-conj-distrib*)

moreover have $(sa', Suc\ r, cfg') \in$ *ref-rel* **using** *step send v R same-dec step-r next-r*
apply(*auto simp add*: *ref-rel-def sa'-def two-step-phase-Suc vote-const-map next0-def*
all-conj-distrib no-vote[*OF send step step-r*])
by (*metis pstate.select-convs*(*1*) *pstate.surjective pstate.update-convs*(*2*))

ultimately show
$\exists sa'. (\exists r\ S\ v. (sa, sa') \in tso\text{-}round0\ r\ S\ v) \wedge (sa', Suc\ r, cfg') \in ref\text{-}rel$
by *blast*
qed

definition *D* **where**
$D\ cfg\ cfg' \equiv \{p.\ decide\ (cfg'\ p) \neq decide\ (cfg\ p)\ \}$

lemma *decide-origin*:
assumes
send: $\forall p.\ \mu\ p \in get\text{-}msgs\ (send1\ r)\ cfg\ (HOs\ r)\ (HOs\ r)\ p$
and *step*: $\forall p.\ next1\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p)$
and *step-r*: $two\text{-}step\ r = Suc\ 0$
shows
$D\ cfg\ cfg' \subseteq \{p.\ \exists v.\ decide\ (cfg'\ p) = Some\ v \wedge (\forall q \in HOs\ r\ p.\ vote\ (cfg\ q) = Some\ v)\}$
using *assms*
by(*fastforce simp add*: *D-def next1-def get-msgs-benign send1-def msgRcvd-def o-def restrict-map-def*
x-update-def dec-update-def identicalVoteRcvd-def all-conj-distrib someVoteRcvd-def isVote-def)

lemma *step1-ref*:
$\{ref\text{-}rel \cap (TSO\text{-}inv1 \cap TSO\text{-}inv2) \times UNIV\} \bigcup r\ d\text{-}f\ o\text{-}f.\ tso\text{-}round1\ r\ d\text{-}f\ o\text{-}f,$
BenOr-trans-step HOs HOs next1 send1 (*Suc 0*) {> *ref-rel*}

proof(*clarsimp simp add*: *PO-rhoare-defs BenOr-trans-step-def all-conj-distrib*)
fix *sa r cfg* μ **and** *cfg′* :: *process* ⇒ *val pstate*
assume
R: (*sa*, (*r*, *cfg*)) ∈ *ref-rel*
and *step-r*: *two-step r* = *Suc 0*
and *send*: ∀ *p*. μ *p* ∈ *get-msgs* (*send1 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: ∀ *p*. *next1 r p* (*cfg p*) (μ *p*) (*cfg′ p*)
and *ainv*: *sa* ∈ *TSO-inv1*
and *ainv2*: *sa* ∈ *TSO-inv2*

from *R* **have** *next-r*: *next-round sa* = *r*
by(*simp add*: *ref-rel-def*)

define *S* **where** *S* = {*p*. ∃ *v*. *vote* (*cfg p*) = *Some v*}
from *R* **obtain** *v* **where** *v*: ∀ *p* ∈ *S*. *vote* (*cfg p*) = *Some v* **using** *ainv step-r*
by(*auto simp add*: *ref-rel-def TSO-inv1-def S-def*)

define *Ob* **where** *Ob* = {*p*. *x* (*cfg′ p*) = *v*}
define *o-f* **where** *o-f p* = (*if S* ∈ *Quorum then Some v else None*) **for** *p* :: *process*

define *dec-f* **where** *dec-f p* = (*if p* ∈ *D cfg cfg′ then decide* (*cfg′ p*) *else None*) **for** *p*

{
fix *p w*
assume *vote* (*cfg p*) = *Some w*
hence *w* = *v* **using** *v*
by(*unfold S-def*, *auto*)
} **note** *v′*=*this*

have *d-guard*: *d-guard dec-f* (*vote* ∘ *cfg*) **using** *per-rd*[*THEN spec*, *of r*]
by(*fastforce simp add*: *d-guard-def locked-in-vf-def quorum-for-def dec-f-def*
BenOr-commPerRd-def dest!: *decide-origin*[*OF send step step-r*, *THEN subsetD*])

have *dom* (*vote* ∘ *cfg*) ∈ *Quorum* ⟶ *Ob* = *UNIV*
proof(*auto simp add*: *Ob-def*)
fix *p*
assume *Q*: *dom* (*vote* ∘ *cfg*) ∈ *Quorum* (**is** *?Q* ∈ *Quorum*)

```
    hence ?Q ∩ HOs r p ≠ {}  using per-rd[THEN spec, of r]
      by(auto simp add: BenOr-commPerRd-def dest: qintersect)
    hence someVoteRcvd (μ p) ≠ {} using send[THEN spec, of p]
     by(force simp add: someVoteRcvd-def get-msgs-benign msgRcvd-def restrict-map-def

        isVote-def send1-def)
    moreover have ∀ q ∈ someVoteRcvd (μ p). ∃ x'. μ p q = Some (Vote (Some
v))
       using send[THEN spec, of p]
     by(auto simp add: someVoteRcvd-def get-msgs-benign msgRcvd-def restrict-map-def
        isVote-def send1-def dest: v')
    ultimately show x (cfg' p) = v using step[THEN spec, of p]
      by(auto simp add: next1-def x-update-def)
  qed
  note Ob-UNIV=this[rule-format]

  have obs-guard: obs-guard o-f (vote ∘ cfg)
    apply(auto simp add: obs-guard-def o-f-def S-def dom-def
      dest: v' Ob-UNIV quorum-non-empty)
    apply (metis S-def all-not-in-conv  empty-not-quorum v)
    done

  define sa' where sa' = sa⦇
    next-round := Suc (next-round sa)
    , decisions := decisions sa ++ dec-f
    , last-obs := last-obs sa ++ o-f
    ⦈

  — Abstract step
  have abs-step: (sa, sa') ∈ tso-round1 r dec-f o-f  using next-r step-r R d-guard
obs-guard
    by(auto simp add: tso-round1-def sa'-def ref-rel-def)

  — Relation preserved
  have ∀ p. ((decide ∘ cfg) ++ dec-f) p = decide (cfg' p)
  proof
    fix p
    show ((decide ∘ cfg) ++ dec-f) p = decide (cfg' p) using step[THEN spec, of
p]
      by(auto simp add: dec-f-def D-def next1-def dec-update-def map-add-def)
```

qed
note *dec-rel=this*[*rule-format*]

have $\forall$ *p*. ($\exists$ *q*. *o-f q* = *None* $\wedge$ *opt-obsv-state.last-obs sa q* = *None*
  $\vee$ (*opt-obsv-state.last-obs sa* ++ *o-f*) *q* = *Some* (*x* (*cfg′ p*)))
proof(*intro allI impI*, *cases S* $\in$ *Quorum*)
 fix *p*
 case *True*
 hence *x* (*cfg′ p*) = *v* **using** *Ob-UNIV*
  by(*auto simp add*: *S-def Ob-def dom-def*)
 thus ($\exists$ *q*. *o-f q* = *None* $\wedge$ *opt-obsv-state.last-obs sa q* = *None*
  $\vee$ (*opt-obsv-state.last-obs sa* ++ *o-f*) *q* = *Some* (*x* (*cfg′ p*)))
  using *True*
  by(*auto simp add*: *o-f-def*)
next
 fix *p*
 case *False*
 hence *empty*: *o-f* = *Map.empty*
  by(*auto simp add*: *o-f-def*)
 from *False* **have** *S* $\neq$ *UNIV* **using** *UNIV-quorum*
  by *auto*
 then obtain *q* **where** *q*: *vote* (*cfg q*) = *None* **using** *False*
  by(*auto simp add*: *o-f-def S-def*)
 then obtain *q1 q2* **where**
  x (*cfg q1*) $\neq$ *x* (*cfg q2*) **using** *R step-r*
  by(*auto simp add*: *ref-rel-def no-vote-diff-def*)
 then obtain *q1′ q2′* **where**
  x (*cfg q1′*) = *val0*
  x (*cfg q2′*) = *val1*
  by (*metis* (*poly-guards-query*) *val-exhaust*)
 hence $\forall$ *v*. $\exists$ *q*. *opt-obsv-state.last-obs sa q* $\in$ {*None*, *Some v*} **using** *R step-r*
  apply(*auto simp add*: *ref-rel-def*)
  by (*metis* (*poly-guards-query*) *val-exhaust*)

 thus ($\exists$ *q*. *o-f q* = *None* $\wedge$ *opt-obsv-state.last-obs sa q* = *None*
  $\vee$ (*opt-obsv-state.last-obs sa* ++ *o-f*) *q* = *Some* (*x* (*cfg′ p*))) **using** *empty*
  by(*auto*)
qed
note *obs-rel=this*[*rule-format*]

have *post-rel*:
  $(sa', Suc\ r, cfg') \in ref\text{-}rel$ **using** *step send next-r R step-r*
  by(*auto simp add*: *sa′-def ref-rel-def*
    two-step-phase-Suc dec-rel obs-rel)

from *abs-step post-rel* **show**
  $\exists\, sa'. (\exists\, r\ d\text{-}f\ o\text{-}f. (sa, sa') \in tso\text{-}round1\ r\ d\text{-}f\ o\text{-}f) \wedge (sa', Suc\ r, cfg') \in ref\text{-}rel$
  by *blast*

qed

lemma *BenOr-Refines-Two-Step-Obs*:
  $PO\text{-}refines\ (ref\text{-}rel \cap (TSO\text{-}inv1 \cap TSO\text{-}inv2) \times UNIV)$
    tso-TS (*BenOr-TS HOs HOs crds*)
proof(*rule refine-using-invariants*)
  show *init* (*BenOr-TS HOs HOs crds*) $\subseteq$ *ref-rel* ‘‘ *init tso-TS*
  by(*auto simp add*: *BenOr-TS-defs BenOr-HOMachine-def CHOAlgorithm.truncate-def*

    tso-TS-defs ref-rel-def tso-init-def Let-def o-def)
next
  show
    $\{ref\text{-}rel \cap (TSO\text{-}inv1 \cap TSO\text{-}inv2) \times UNIV\}$ *TS.trans tso-TS*,
      TS.trans (*BenOr-TS HOs HOs crds*) {> *ref-rel*}
    apply(*simp add*: *tso-TS-defs BenOr-TS-defs BenOr-HOMachine-def CHOAlgorithm.truncate-def*)
    apply(*auto simp add*: *CHO-trans-alt BenOr-trans intro*!: *step0-ref step1-ref*)
    done
qed(*auto intro*!: *TSO-inv1-inductive TSO-inv2-inductive*)

13.2.2 Termination

The full termination proof for Ben-Or is probabilistic, and depends on the state of the processes, and a "favorable" coin toss, where "favorable" is relative to this state. As this termination pre-condition is state-dependent, we cannot capture it in an HO predicate.

Instead, we prove a variant of the argument, where we assume that there exists a round where all the processes hear from the same set of other processes, and all toss the same coin.

theorem *BenOr-termination*:
  shows $\exists\, r\ v.\ decide\ (rho\ r\ p) = Some\ v$

proof −
  from *comm-global* **obtain** *r1* **where** *r1*:
    ∀ *q*. *HOs r1 p* = *HOs r1 q*
    ∀ *q*. (*coin r1 p* :: *val*) = *coin r1 q*
    two-step r1 = *1*
    by(*simp add*: *BenOr-commGlobal-def all-conj-distrib*, *blast*)

  from *r1* **obtain** *r0* **where** *r1-def*: *r1* = *Suc r0* **and** *step-r0*: *two-step r0* = *0*
    by (*cases r1*) (*auto simp add*: *two-step-phase-Suc two-step-def mod-Suc*)

  define *cfg0* **where** *cfg0* = *rho r0*
  define *cfg1* **where** *cfg1* = *rho r1*
  define *r2* **where** *r2* = *Suc r1*
  define *cfg2* **where** *cfg2* = *rho r2*
  define *r3* **where** *r3* = *Suc r2*
  define *cfg3* **where** *cfg3* = *rho r3*
  define *cfg4* **where** *cfg4* = *rho* (*Suc r3*)

  have *step-r2*: *two-step r2* = *0* **using** *r1*
    by(*auto simp add*: *r2-def two-step-phase-Suc*)

  from
    run[*simplified HORun-def SHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r0*]
    run[*simplified HORun-def SHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r1*]
    run[*simplified HORun-def SHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r2*]
    run[*simplified HORun-def SHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r3*]
    obtain *μ0 μ1 μ2 μ3* **where**
    send0: ∀ *p*. *μ0 p* ∈ *get-msgs* (*send0 r0*) *cfg0* (*HOs r0*) (*HOs r0*) *p*
    and *step0*: ∀ *p*. *next0 r0 p* (*cfg0 p*) (*μ0 p*) (*cfg1 p*)
    and *send1*: ∀ *p*. *μ1 p* ∈ *get-msgs* (*send1 r1*) *cfg1* (*HOs r1*) (*HOs r1*) *p*
    and *step1*: ∀ *p*. *next1 r1 p* (*cfg1 p*) (*μ1 p*) (*cfg2 p*)
    and *send2*: ∀ *p*. *μ2 p* ∈ *get-msgs* (*send0 r2*) *cfg2* (*HOs r2*) (*HOs r2*) *p*
    and *step2*: ∀ *p*. *next0 r2 p* (*cfg2 p*) (*μ2 p*) (*cfg3 p*)
    and *send3*: ∀ *p*. *μ3 p* ∈ *get-msgs* (*send1 r3*) *cfg3* (*HOs r3*) (*HOs r3*) *p*
    and *step3*: ∀ *p*. *next1 r3 p* (*cfg3 p*) (*μ3 p*) (*cfg4 p*)
    by(*auto simp add*: *BenOr-A-def BenOr-HOMachine-def*

```
    two-step-phase-Suc BenOr-nextState-def BenOr-sendMsg-def all-conj-distrib
    CHOAlgorithm.truncate-def step-r0 r1-def r2-def r3-def
    cfg0-def cfg1-def cfg2-def cfg3-def cfg4-def
    )

  let ?v = x (cfg2 p)
  from per-rd r1 have xs: ∀ q. x (cfg2 q) = ?v
  proof(cases ∃ q w. q ∈ HOs r1 p ∧ vote (cfg1 q) = Some w)
    case True
    then obtain q w where q-w: q ∈ HOs r1 p ∧ vote (cfg1 q) = Some w
      by auto
    then have ∀ q'. vote (cfg1 q') ∈ {None, Some w} using same-new-vote[OF
send0 step0 step-r0]
      by (metis insert-iff not-None-eq)
    hence ∀ q'. x (cfg2 q') = w using step1 send1 q-w
      apply(auto simp add: next1-def all-conj-distrib dec-update-def x-update-def
        get-msgs-benign send1-def isVote-def msgRcvd-def identicalVoteRcvd-def
        someVoteRcvd-def restrict-map-def)
      by (metis (erased, opaque-lifting) option.distinct(2) option.sel r1(1))
    thus ?thesis
      by auto
  next
    case False
    hence ∀ q'. x (cfg2 q') = coin r1 q' using r1 step1 send1
      apply(auto simp add: next1-def all-conj-distrib dec-update-def x-update-def
        get-msgs-benign send1-def isVote-def msgRcvd-def identicalVoteRcvd-def
        someVoteRcvd-def restrict-map-def)
      by (metis False)
    thus ?thesis using r1
      by(metis)
  qed

  hence ∀ q. vote (cfg3 q) = Some ?v
    by(simp add: vote-origin[OF send2 step2 step-r2])

  hence decide (cfg4 p) = Some ?v using send3[THEN spec, of p] step3[THEN
spec, of p] HOs-nonempty
    by(auto simp add: next1-def send1-def get-msgs-benign dec-update-def
      restrict-map-def identicalVoteRcvd-def msgRcvd-def isVote-def)
```

thus *?thesis*
by(*auto simp add*: *cfg4-def*)
qed

end

end

end

14 The MRU Vote Model

theory *MRU-Vote*
imports *Same-Vote*
begin

context *quorum-process*
begin

This model is identical to Same Vote, except that it replaces the *safe* guard with the following one, which says that v is the most recently used (MRU) vote of a quorum:

definition *mru-guard* :: *v-state* ⇒ *process set* ⇒ *val* ⇒ *bool* **where**
mru-guard s Q v ≡ *Q* ∈ *Quorum* ∧ (*let mru* = *mru-of-set* (*votes s*) *Q in*
mru = *None* ∨ (∃ *r*. *mru* = *Some* (*r*, *v*)))

The concrete algorithms will not refine the MRU Voting model directly, but its optimized version instead. For simplicity, we thus do not create the model explicitly, but just prove guard strengthening. We will show later that the optimized model refines the Same Vote model.

lemma *mru-vote-implies-safe*:
assumes
inv4: *s* ∈ *SV-inv4*
and *inv1*: *s* ∈ *Vinv1*
and *mru-vote*: *mru-guard s Q v*
and *is-Quorum*: *Q* ∈ *Quorum*
shows *safe s* (*v-state.next-round s*) *v* **using** *mru-vote*
proof(*clarsimp simp add*: *mru-guard-def mru-of-set-def option-Max-by-def*)

— The first case: some votes have been cast. We prove that the most recently used one is safe.

```
fix r
assume
  nempty: vote-set (votes s) Q ≠ {}
  and max: Max-by fst (vote-set (votes s) Q) = (r, v)

from Max-by-in[OF Vinv1-finite-vote-set[OF inv1] nempty] max
have in-votes: (r, v) ∈ vote-set (votes s) Q by metis

have no-larger: ∀ a'∈Q. ∀ r'>r. votes s r' a' = None
proof(safe, rule ccontr, clarsimp)
  fix a' r' w
  assume a' ∈ Q  votes s r' a' = Some w and gt: r' > r
  hence (r', w) ∈ vote-set (votes s) Q
    by(auto simp add: vote-set-def)
  thus False
     using Max-by-ge[where f=fst, OF Vinv1-finite-vote-set[where Q=Q, OF
inv1]] max gt
    by(clarsimp simp add: not-le[symmetric])
qed

have safe s (Suc r) v using inv4 in-votes and SV-inv4-def
  by(clarsimp simp add: vote-set-def)

thus safe s (v-state.next-round s) v using no-larger is-Quorum[THEN qintersect]
  apply(clarsimp simp add: safe-def quorum-for-def)
  by (metis IntE all-not-in-conv not-less-eq option.distinct(1))

next
  assume vote-set (votes s) Q = {}
  thus safe s (v-state.next-round s) v using is-Quorum[THEN qintersect]
    by(force simp add: vote-set-def safe-def quorum-for-def)
qed

end

end
```

15 Optimized MRU Vote Model

theory *MRU-Vote-Opt*
imports *MRU-Vote*
begin

15.1 Model definition

record *opt-mru-state* =
  next-round :: *round*
  mru-vote :: (*process*, *round* × *val*) *map*
  decisions :: (*process*, *val*)*map*

definition *opt-mru-init* **where**
  opt-mru-init = { ⦇ *next-round* = *0*, *mru-vote* = *Map.empty*, *decisions* = *Map.empty* ⦈ }

context *quorum-process* **begin**

definition *opt-mru-vote* :: (*process*, *round* × *val*)*map* ⇒ (*process set*, *round* × *val*)*map* **where**
  opt-mru-vote lvs Q = *option-Max-by fst* (*ran* (*lvs* |' *Q*))

definition *opt-mru-guard* :: (*process*, *round* × *val*)*map* ⇒ *process set* ⇒ *val* ⇒ *bool* **where**
  opt-mru-guard mru-votes Q v ≡ *Q* ∈ *Quorum* ∧
    (*let mru* = *opt-mru-vote mru-votes Q in mru* = *None* ∨ (∃ *r*. *mru* = *Some* (*r*, *v*)))

definition *opt-mru-round*
  :: *round* ⇒ *process set* ⇒ *process set* ⇒ *val* ⇒ (*process*, *val*)*map* ⇒ (*opt-mru-state* × *opt-mru-state*) *set*
  where
  opt-mru-round r Q S v r-decisions = {(*s*, *s'*).
    — guards
    r = *next-round s*
    ∧ (*S* ≠ {} ⟶ *opt-mru-guard* (*mru-vote s*) *Q v*)
    ∧ *d-guard r-decisions* (*const-map v S*)
    ∧ — actions
    s' = *s*⦇

mru-vote := *mru-vote s* ++ *const-map* (*r*, *v*) *S*
, *next-round* := *Suc r*
, *decisions* := *decisions s* ++ *r-decisions*
⦈
}

lemmas *lv-evt-defs* = *opt-mru-round-def opt-mru-guard-def*

definition *mru-opt-trans* :: (*opt-mru-state* × *opt-mru-state*) *set* **where**
mru-opt-trans = (⋃ *r Q S v D*. *opt-mru-round r Q S v D*) ∪ *Id*

definition *mru-opt-TS* :: *opt-mru-state TS* **where**
mru-opt-TS = ⦇ *init* = *opt-mru-init*, *trans* = *mru-opt-trans* ⦈

lemmas *mru-opt-TS-defs* = *mru-opt-TS-def opt-mru-init-def mru-opt-trans-def*

15.2 Refinement

definition *lv-ref-rel* :: (*v-state* × *opt-mru-state*)*set* **where**
lv-ref-rel = {(*sa*, *sc*).
sc = ⦇
next-round = *v-state.next-round sa*
, *mru-vote* = *process-mru* (*votes sa*)
, *decisions* = *v-state.decisions sa*
⦈
}

15.2.1 The concrete guard implies the abstract guard

definition *voters* :: (*round* ⇒ (*process*, *val*)*map*) ⇒ *process set* **where**
voters vs = {*a*|*a r v*. ((*r*, *a*), *v*) ∈ *map-graph* (*case-prod vs*)}

lemma *vote-set-as-Union*:
vote-set vs Q = (⋃ *a*∈(*Q* ∩ *voters vs*). *vote-set vs* {*a*})
by(*auto simp add*: *vote-set-def voters-def*)

lemma *empty-ran*:
(*ran f* = {}) = (∀ *x*. *f x* = *None*)
apply(*auto simp add*: *ran-def*)
by (*metis option.collapse*)

lemma *empty-ran-restrict*:
$(ran\ (f\ |`\ A) = \{\}) = (\forall\, x \in A.\ f\ x = None)$
by(*auto simp add*: *empty-ran restrict-map-def*)

lemma *option-Max-by-eqI*:
$[\![\ (S = \{\}) \longleftrightarrow (S' = \{\});\ S \neq \{\} \wedge S' \neq \{\} \Longrightarrow Max\text{-}by\ f\ S = Max\text{-}by\ g\ S'\ ]\!]$
$\Longrightarrow\ option\text{-}Max\text{-}by\ f\ S = option\text{-}Max\text{-}by\ g\ S'$
by(*auto simp add*: *option-Max-by-def*)

lemma *ran-process-mru-only-voters*:
$ran\ (process\text{-}mru\ vs\ |`\ Q) = ran\ (process\text{-}mru\ vs\ |`\ (Q \cap voters\ vs))$
by(*auto simp add*: *ran-def restrict-map-def voters-def process-mru-def*
mru-of-set-def option-Max-by-def vote-set-def)

lemma *SV-inv3-inj-on-fst-vote-set*:
$s \in SV\text{-}inv3 \Longrightarrow inj\text{-}on\ fst\ (vote\text{-}set\ (votes\ s)\ Q)$
by(*clarsimp simp add*: *SV-inv3-def inj-on-def vote-set-def*)

lemma *opt-mru-vote-mru-of-set*:
assumes
inv1: $s \in Vinv1$
and *inv3*: $s \in SV\text{-}inv3$
defines $vs \equiv votes\ s$
shows
$opt\text{-}mru\text{-}vote\ (process\text{-}mru\ vs)\ Q = mru\text{-}of\text{-}set\ vs\ Q$
proof(*simp add*: *opt-mru-vote-def mru-of-set-def*, *intro option-Max-by-eqI*, *clar-simp-all*)
show $(ran\ (process\text{-}mru\ vs\ |`\ Q) = \{\}) = (vote\text{-}set\ vs\ Q = \{\})$
apply(*clarsimp simp add*: *empty-ran-restrict process-mru-def mru-of-set-def option-Max-by-def*
vote-set-def)
by (*metis option.collapse option.distinct*(*1*))
next
assume *nempty*: $ran\ (process\text{-}mru\ vs\ |`\ Q) \neq \{\}\ vote\text{-}set\ vs\ Q \neq \{\}$

hence *nempty'*: $Q \cap voters\ vs \neq \{\}$
by(*auto simp add*: *vote-set-def voters-def*)

have *nempty''*: $\{\} \notin (\lambda a.\ vote\text{-}set\ vs\ \{a\})\ `\ (Q \cap voters\ vs)$

by(*auto simp add*: *vote-set-def voters-def*)

note *fin*=*Vinv1-finite-vote-set*[*OF inv1*]

have *ran-eq*:
ran (*process-mru vs* |‘ *Q*) = *Max-by fst* ‘ (λa. $\bigcup a \in \{a\} \cap$ *voters vs*. *vote-set vs* $\{a\}$) ‘ ($Q \cap$ *voters vs*)
apply(*subst ran-process-mru-only-voters*)
apply(*auto simp add*: *image-def process-mru-def ran-def restrict-map-def mru-of-set-def option-Max-by-def*)
by (*metis* (*erased*, *lifting*) *Set.set-insert image-eqI insertI1 insert-inter-insert nempty''*)

note *inj*=*inv3*[*THEN SV-inv3-inj-on-fst-vote-set*]

show *Max-by fst* (*ran* (*process-mru vs* |‘ *Q*)) = *Max-by fst* (*vote-set vs Q*)
apply(*subst vs-def*
Max-by-UNION-distrib[*OF fin vote-set-as-Union nempty'*[*simplified vs-def*]
nempty''[*simplified vs-def*] *inj*])+
apply(*subst vote-set-as-Union*)
by(*metis ran-eq vs-def*)
qed

lemma *opt-mru-guard-imp-mru-guard*:
assumes *invs*:
$s \in$ *Vinv1* $s \in$ *SV-inv3*
and *c-guard*: *opt-mru-guard* (*process-mru* (*votes s*)) *Q v*
shows *mru-guard s Q v* **using** *c-guard*
by(*simp add*: *opt-mru-vote-mru-of-set*[*OF invs*] *opt-mru-guard-def mru-guard-def Let-def*)

15.2.2 The concrete action refines the abstract action

lemma *act-ref*:
assumes
$s \in$ *Vinv1*
shows
process-mru (*votes s*) ++ *const-map* (*v-state.next-round s*, *v*) *S*
= *process-mru* ((*votes s*)(*v-state.next-round s* := *const-map v S*))
by(*auto simp add*: *process-mru-map-add*[*OF assms*(*1*)] *map-add-def const-map-def*

restrict-map-def
split:*option.split*)

15.2.3 The complete refinement

lemma *opt-mru-guard-imp-Quorum*:
opt-mru-guard vs Q v ⟹ *Q* ∈ *Quorum*
by (*simp add*: *opt-mru-guard-def Let-def*)

lemma *opt-mru-round-refines*:
{*lv-ref-rel* ∩ (*Vinv1* ∩ *SV-inv3* ∩ *SV-inv4*) × *UNIV*}
sv-round r S v d-f, *opt-mru-round r Q S v d-f*
{> *lv-ref-rel*}
apply(*clarsimp simp add*: *PO-rhoare-defs lv-ref-rel-def opt-mru-round-def sv-round-def*

act-ref del: *disjCI*)
apply(*auto intro*!: *opt-mru-guard-imp-mru-guard*[**where** *Q*=*Q*] *mru-vote-implies-safe*[**where** *Q*=*Q*]
dest: *opt-mru-guard-imp-Quorum*)
done

lemma *Opt-MRU-Vote-Refines*:
PO-refines (*lv-ref-rel* ∩ (*Vinv1* ∩ *Vinv2* ∩ *SV-inv3* ∩ *SV-inv4*) × *UNIV*) *sv-TS* *mru-opt-TS*
proof(*rule refine-using-invariants*)
show *init mru-opt-TS* ⊆ *lv-ref-rel* '' *init sv-TS*
by(*auto simp add*: *mru-opt-TS-defs sv-TS-defs lv-ref-rel-def*
process-mru-def mru-of-set-def vote-set-def option-Max-by-def)
next
show
{*lv-ref-rel* ∩ (*Vinv1* ∩ *Vinv2* ∩ *SV-inv3* ∩ *SV-inv4*) × *UNIV*} *trans sv-TS*, *trans mru-opt-TS* {> *lv-ref-rel*}
by(*auto simp add*: *sv-TS-defs mru-opt-TS-defs intro*!: *opt-mru-round-refines*)
next
show
{*Vinv1* ∩ *Vinv2* ∩ *SV-inv3* ∩ *SV-inv4* ∩ *Domain* (*lv-ref-rel* ∩ *UNIV* × *UNIV*)}
trans sv-TS
{> *Vinv1* ∩ *Vinv2* ∩ *SV-inv3* ∩ *SV-inv4*}

using *SV-inv1-inductive*(*2*) *SV-inv2-inductive*(*2*) *SV-inv3-inductive*(*2*) *SV-inv4-inductive*(*2*)
by *blast*
qed(*auto intro*!: *SV-inv1-inductive*(*1*) *SV-inv2-inductive*(*1*) *SV-inv3-inductive*(*1*) *SV-inv4-inductive*(*1*))

15.3 Invariants

definition *OMRU-inv1* :: *opt-mru-state set* **where**
OMRU-inv1 = {*s*. ∀ *p*. (*case mru-vote s p of*
Some (*r*, -) ⇒ *r* < *next-round s*
| *None* ⇒ *True*)
}

lemma *OMRU-inv1-inductive*:
init mru-opt-TS ⊆ *OMRU-inv1*
{*OMRU-inv1*} *trans mru-opt-TS* {> *OMRU-inv1*}
by(*fastforce simp add*: *mru-opt-TS-def opt-mru-init-def PO-hoare-def OMRU-inv1-def mru-opt-trans-def*
opt-mru-round-def const-map-is-Some less-Suc-eq
split: *option.split-asm option.split*)+

lemmas *OMRU-inv1I* = *OMRU-inv1-def* [*THEN setc-def-to-intro*, *rule-format*]
lemmas *OMRU-inv1E* [*elim*] = *OMRU-inv1-def* [*THEN setc-def-to-elim*, *rule-format*]
lemmas *OMRU-inv1D* = *OMRU-inv1-def* [*THEN setc-def-to-dest*, *rule-format*]

end

end

16 Three-step Optimized MRU Model

theory *Three-Step-MRU*
imports ../*MRU-Vote-Opt Three-Steps*
begin

To make the coming proofs of concrete algorithms easier, in this model we split the *opt-mru-round* into three steps

16.1 Model definition

record *three-step-mru-state* = *opt-mru-state* +
  candidates :: *val set*

context *mono-quorum*
begin

definition *opt-mru-step0* :: *round* ⇒ *val set* ⇒ (*three-step-mru-state* × *three-step-mru-state*) *set* **where**
  opt-mru-step0 r C = {(*s*, *s'*).
    — guards
    r = *next-round s* ∧ *three-step r* = *0*
    ∧ (∀ *cand* ∈ *C*. ∃ *Q*. *opt-mru-guard* (*mru-vote s*) *Q cand*)
    ∧ — actions
    s' = *s*⦇
      candidates := *C*
      , *next-round* := *Suc r*
    ⦈
  }

definition *opt-mru-step1* :: *round* ⇒ *process set* ⇒ *val* ⇒
  (*three-step-mru-state* × *three-step-mru-state*) *set* **where**
  opt-mru-step1 r S v = {(*s*, *s'*).
    — guards
    r = *next-round s* ∧ *three-step r* = *1*
    ∧ (*S* ≠ {} ⟶ *v* ∈ *candidates s*)
    ∧ — actions
    s' = *s*⦇
      mru-vote := *mru-vote s* ++ *const-map* (*three-phase r*, *v*) *S*
      , *next-round* := *Suc r*
    ⦈
  }

definition *step2-d-guard* :: (*process*, *val*)*map* ⇒ (*process*, *val*)*map* ⇒ *bool* **where**
  step2-d-guard r-decisions r-votes ≡ ∀ *p v*. *r-decisions p* = *Some v* ⟶
    v ∈ *ran r-votes* ∧ *dom r-votes* ∈ *Quorum*

definition *r-votes* :: *three-step-mru-state* ⇒ *round* ⇒ (*process*, *val*)*map* **where**
  r-votes s r ≡ λ*p*. *if* (∃ *v*. *mru-vote s p* = *Some* (*three-phase r*, *v*))

then map-option snd (mru-vote s p)
else None

definition *opt-mru-step2* :: *round* ⇒ (*process*, *val*)*map* ⇒ (*three-step-mru-state* × *three-step-mru-state*) *set* **where**
opt-mru-step2 r r-decisions = {(*s*, *s'*).
— guards
r = *next-round s* ∧ *three-step r* = *2*
∧ *step2-d-guard r-decisions* (*r-votes s r*)
∧ — actions
s' = *s*⦇
next-round := *Suc r*
, *decisions* := *decisions s* ++ *r-decisions*
⦈
}

lemmas *ts-mru-evt-defs* = *opt-mru-step0-def opt-mru-step1-def opt-mru-guard-def*

definition *ts-mru-trans* :: (*three-step-mru-state* × *three-step-mru-state*) *set* **where**
ts-mru-trans = ($\bigcup$ *r C*. *opt-mru-step0 r C*)
∪ ($\bigcup$ *r S v*. *opt-mru-step1 r S v*)
∪ ($\bigcup$ *r dec-f*. *opt-mru-step2 r dec-f*) ∪ *Id*

definition *ts-mru-init* **where**
ts-mru-init = { ⦇ *next-round* = *0*, *mru-vote* = *Map.empty*, *decisions* = *Map.empty*, *candidates* = {} ⦈ }

definition *ts-mru-TS* :: *three-step-mru-state TS* **where**
ts-mru-TS = ⦇ *init* = *ts-mru-init*, *trans* = *ts-mru-trans* ⦈

lemmas *ts-mru-TS-defs* = *ts-mru-TS-def ts-mru-init-def ts-mru-trans-def*

16.2 Refinement

definition *basic-rel* **where**
basic-rel ≡ {(*sa*, *sc*).
decisions sc = *decisions sa*
∧ *next-round sa* = *three-phase* (*next-round sc*)
}

definition *three-step0-rel* :: (*opt-mru-state* × *three-step-mru-state*)*set* **where**
three-step0-rel ≡ *basic-rel* ∩ {(*sa*, *sc*).
three-step (*next-round sc*) = *0*
∧ *mru-vote sc* = *mru-vote sa*
}

definition *three-step1-rel* :: (*opt-mru-state* × *three-step-mru-state*)*set* **where**
three-step1-rel ≡ *basic-rel* ∩ {(*sa*, *sc*).
(∃ *sc′ r C*. (*sa*, *sc′*) ∈ *three-step0-rel* ∧ (*sc′*, *sc*) ∈ *opt-mru-step0 r C*)
∧ *mru-vote sc* = *mru-vote sa*
}

definition *three-step2-rel* :: (*opt-mru-state* × *three-step-mru-state*)*set* **where**
three-step2-rel ≡ *basic-rel* ∩ {(*sa*, *sc*).
(∃ *sc′ r S v*. (*sa*, *sc′*) ∈ *three-step1-rel* ∧ (*sc′*, *sc*) ∈ *opt-mru-step1 r S v*)
}

definition *ts-ref-rel* **where**
ts-ref-rel = {(*sa*, *sc*).
(*three-step* (*next-round sc*) = *0* ⟶ (*sa*, *sc*) ∈ *three-step0-rel*)
∧ (*three-step* (*next-round sc*) = *1* ⟶ (*sa*, *sc*) ∈ *three-step1-rel*)
∧ (*three-step* (*next-round sc*) = *2* ⟶ (*sa*, *sc*) ∈ *three-step2-rel*)
}

lemmas *ts-ref-rel-defs* =
basic-rel-def
ts-ref-rel-def
three-step0-rel-def
three-step1-rel-def
three-step2-rel-def

lemma *step0-ref*:
{*ts-ref-rel*} *Id*, *opt-mru-step0 r C* {> *ts-ref-rel*}
proof(*simp only*: *PO-rhoare-defs*, *safe*)
fix *sa sc sc′*
assume *R*: (*sa*, *sc*) ∈ *ts-ref-rel* **and** *step*: (*sc*, *sc′*) ∈ *opt-mru-step0 r C*
hence *r-step*: *three-step* (*next-round sc*) = *0 three-step* (*next-round sc′*) = *1*
by(*auto simp add*: *ts-ref-rel-def opt-mru-step0-def three-step-Suc*)
hence (*sa*, *sc′*) ∈ *ts-ref-rel* **using** *R step*

apply(*auto simp add*: *ts-ref-rel-def three-step-phase-Suc three-step1-rel-def three-step2-rel-def intro*!: *exI*[**where** *x*=*sc*] *step R*)
apply(*auto simp add*: *three-step0-rel-def basic-rel-def three-step-phase-Suc opt-mru-step0-def*)
done
thus $\exists$ *sa'*. (*sa*, *sa'*) $\in$ *Id* $\wedge$ (*sa'*, *sc'*) $\in$ *ts-ref-rel*
by *blast*
qed

lemma *step1-ref*:
{*ts-ref-rel*} *Id*, *opt-mru-step1 r S v* {> *ts-ref-rel*}
proof(*simp only*: *PO-rhoare-defs*, *safe*)
fix *sa sc sc'*
assume *R*: (*sa*, *sc*) $\in$ *ts-ref-rel* **and** *step*: (*sc*, *sc'*) $\in$ *opt-mru-step1 r S v*
hence *r-step*: *three-step* (*next-round sc*) = *1 three-step* (*next-round sc'*) = *2*
by(*auto simp add*: *ts-ref-rel-def opt-mru-step1-def three-step-Suc*)
hence (*sa*, *sc'*) $\in$ *ts-ref-rel* **using** *R step*
apply(*auto simp add*: *ts-ref-rel-def three-step-phase-Suc three-step0-rel-def three-step2-rel-def intro*!: *exI*[**where** *x*=*sc*] *step R*)
apply(*auto simp add*: *three-step1-rel-def basic-rel-def three-step-phase-Suc opt-mru-step1-def*)
done
thus $\exists$ *sa'*. (*sa*, *sa'*) $\in$ *Id* $\wedge$ (*sa'*, *sc'*) $\in$ *ts-ref-rel*
by *blast*
qed

lemma *step2-ref*:
{*ts-ref-rel* $\cap$ *OMRU-inv1* $\times$ *UNIV*}
$\bigcup$ *r' Q S' v dec-f'*. *opt-mru-round r' Q S' v dec-f'*,
opt-mru-step2 r dec-f {> *ts-ref-rel*}
proof(*auto simp only*: *PO-rhoare-defs*)
fix *sa sc2 sc3*
assume
ainv: *sa* $\in$ *OMRU-inv1*
and *R*: (*sa*, *sc2*) $\in$ *ts-ref-rel*
and *step2*: (*sc2*, *sc3*) $\in$ *opt-mru-step2 r dec-f*

from *R step2* **obtain** *sc0 r0 C sc1 r1 S v* **where**
R0: (*sa*, *sc0*) $\in$ *three-step0-rel* **and** *step0*: (*sc0*, *sc1*) $\in$ *opt-mru-step0 r0 C*
and *R1*: (*sa*, *sc1*) $\in$ *three-step1-rel* **and** *step1*: (*sc1*, *sc2*) $\in$ *opt-mru-step1 r1*

```
S v
    by(fastforce simp add: ts-ref-rel-def three-step2-rel-def opt-mru-step2-def
      three-step1-rel-def)

  have R2: (sa, sc2) ∈ three-step2-rel
    and r2-step: three-step (next-round sc2) = Suc (Suc 0)
    and r3-step: three-step (next-round sc3) = 0
    using R step2
   by(auto simp add: ts-ref-rel-def opt-mru-step2-def three-step-phase-Suc three-step-Suc)

  have r: r = Suc r1 and r1: r1 = Suc r0 r1 = next-round sc1 and
    r0: r0 = next-round sc0 and r0-step: three-step r0 = 0 and
    r2: r = next-round sc2
    using step0 step1 step2
    by(auto simp add: opt-mru-step0-def opt-mru-step1-def opt-mru-step2-def)

  have abs-round2: next-round sa = three-phase r using R2 r2
    by(auto simp add: three-step2-rel-def basic-rel-def)
  have abs-round1: next-round sa = three-phase r1 using R1 r1
    by(auto simp add: three-step1-rel-def basic-rel-def)
  have abs-round0: next-round sa = three-phase r0 using R0 r0 r
    by(auto simp add: three-step0-rel-def basic-rel-def three-step-phase-Suc)

  have r-votes: r-votes sc2 r = const-map v S
  proof(rule ext)
    fix p
    show r-votes sc2 r p = const-map v S p
    proof(cases r-votes sc2 r p)
      case None
      thus ?thesis using step0 step1 abs-round0 abs-round1 abs-round2
        by(auto simp add: r-votes-def opt-mru-step1-def
          const-map-def restrict-map-def map-add-def)
    next
      case (Some w)
       hence in-S: mru-vote sc1 p ≠ mru-vote sc2 p using R0 step0 ainv r1 r
abs-round0
        by(auto simp add: r-votes-def ts-ref-rel-defs
          three-step-phase-Suc opt-mru-step0-def dest: OMRU-inv1D[where p=p])
      hence p ∈ S  using step1
        by(auto simp add: opt-mru-step1-def map-add-def const-map-is-Some
```

split: *option.split-asm*)
moreover have *w*=*v* **using** *R0 R1 step1 ainv r abs-round1 Some*
by(*auto simp add*: *r-votes-def ts-ref-rel-defs const-map-is-Some three-step-phase-Suc opt-mru-step1-def dest*: *OMRU-inv1D*[**where** *p*=*p*])
ultimately show *?thesis* **using** *Some*
by(*auto simp add*: *const-map-def*)
qed
qed

from *step0 step1* **obtain** *Q* **where** *Q*: $S \neq \{\} \longrightarrow$ *opt-mru-guard* (*mru-vote sc0*) *Q v*
by(*auto simp add*: *opt-mru-step0-def opt-mru-step1-def*)

define *sa′* **where** *sa′* = *sa*⦇
mru-vote := *mru-vote sa* ++ *const-map* (*three-phase r*, *v*) *S*
, *next-round* := *Suc* (*three-phase r*)
, *decisions* := *decisions sa* ++ *dec-f*
⦈

have *guard-strengthening*:
step2-d-guard dec-f (*r-votes sc2 r*) $\longrightarrow$ *d-guard dec-f* (*const-map v S*)
by(*auto simp add*: *r-votes d-guard-def step2-d-guard-def locked-in-vf-def quorum-for-def const-map-is-Some dom-const-map*)
have (*sa*, *sa′*) ∈ *opt-mru-round* (*three-phase r*) *Q S v dec-f* ∧ (*sa′*, *sc3*) ∈ *ts-ref-rel*
proof
show (*sa′*, *sc3*) ∈ *ts-ref-rel* **using** *r3-step R0 R1 R2 step0 step1 step2*
by(*auto simp add*: *ts-ref-rel-def three-step0-rel-def basic-rel-def opt-mru-step0-def*

opt-mru-step1-def opt-mru-step2-def sa′-def three-step-phase-Suc)
next
show (*sa*, *sa′*) ∈ *opt-mru-round* (*three-phase r*) *Q S v dec-f*
using *R0 r0-step step0 step1 step2 r0 r1 r2 r r-votes Q guard-strengthening*
by(*auto simp add*: *ts-ref-rel-defs opt-mru-round-def three-step-phase-Suc opt-mru-step0-def opt-mru-step1-def opt-mru-step2-def sa′-def*)
qed
thus ∃ *y*. (*sa*, *y*) ∈ (⋃ *r′ Q S′ v D′*. *opt-mru-round r′ Q S′ v D′*) ∧ (*y*, *sc3*) ∈ *ts-ref-rel*
by *simp blast*
qed

lemma *ThreeStep-Coordinated-Refines*:
  PO-refines (*ts-ref-rel* ∩ *OMRU-inv1* × *UNIV*)
    mru-opt-TS ts-mru-TS
proof(*rule refine-using-invariants*)
  show *init ts-mru-TS* ⊆ *ts-ref-rel* ‘‘ *init mru-opt-TS*
   by(*auto simp add*: *ts-mru-TS-defs mru-opt-TS-defs ts-ref-rel-def three-step0-rel-def*

      three-step1-rel-def three-step2-rel-def basic-rel-def)
next
  show
    {*ts-ref-rel* ∩ *OMRU-inv1* × *UNIV*} *TS.trans mru-opt-TS*,
      TS.trans (*ts-mru-TS*) {> *ts-ref-rel*}
    apply(*simp add*: *mru-opt-TS-defs ts-mru-TS-defs*)
    apply(*auto simp add*: *ts-mru-trans-def intro*!: *step0-ref step1-ref step2-ref*)
    done
qed(*auto intro*!: *OMRU-inv1-inductive*)

end

end

17 The New Algorithm

theory *New-Algorithm-Defs*
imports *Heard-Of.HOModel ../Consensus-Types ../Consensus-Misc Three-Steps*
begin

17.1 Model of the algorithm

We assume that the values are linearly ordered, to be able to have each process select the smallest value.

axiomatization where *val-linorder*:
  OFCLASS(*val*, *linorder-class*)

instance *val* :: *linorder* **by** (*rule val-linorder*)

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

prop-vote :: *val option*
mru-vote :: (*nat* × *val*) *option*
decide :: *val option* — value the process has decided on, if any

datatype *msg* =
MruVote (*nat* × *val*) *option val*
| *PreVote val*
| *Vote val*
| *Null* — dummy message in case nothing needs to be sent

Characteristic predicates on messages.

definition *isLV* **where** *isLV m* ≡ ∃ *rv*. *m* = *Vote rv*

definition *isPreVote* **where** *isPreVote m* ≡ ∃ *px*. *m* = *PreVote px*

definition *NA-initState* **where**
NA-initState p st - ≡
mru-vote st = *None*
∧ *prop-vote st* = *None*
∧ *decide st* = *None*

definition *send0* **where**
send0 r p q st ≡ *MruVote* (*mru-vote st*) (*x st*)

fun *msg-to-val-stamp* :: *msg* ⇒ (*round* × *val*)*option* **where**
msg-to-val-stamp (*MruVote rv* -) = *rv*

definition *msgs-to-lvs* ::
(*process* ⇀ *msg*)
⇒ (*process*, *round* × *val*) *map*
where
msgs-to-lvs msgs ≡ *msg-to-val-stamp* $\circ_m$ *msgs*

definition *smallest-proposal* **where**
smallest-proposal (*msgs*::*process* ⇀ *msg*) ≡
Min {*v*. ∃ *q mv*. *msgs q* = *Some* (*MruVote mv v*)}

definition *next0*
:: *nat*

```
  ⇒ process
  ⇒ pstate
  ⇒ (process ⇀ msg)
  ⇒ process
  ⇒ pstate
  ⇒ bool
where
  next0 r p st msgs crd st′ ≡ let
      Q = dom msgs;
      lvs = msgs-to-lvs msgs;
      smallest = if Q = {} then x st else smallest-proposal msgs
    in
    st′ = st ⦇
      prop-vote := if card Q > N div 2
        then Some (case-option smallest snd (option-Max-by fst (ran (lvs |‘ Q))))
        else None
    ⦈

definition send1 where
  send1 r p q st ≡ case prop-vote st of
    None ⇒ Null
    | Some v ⇒ PreVote v

definition Q-prevotes-v where
  Q-prevotes-v msgs Q v ≡ let D = dom msgs in
    Q ⊆ D ∧ card Q > N div 2 ∧ (∀ q ∈ Q. msgs q = Some (PreVote v))

definition next1
  :: nat
  ⇒ process
  ⇒ pstate
  ⇒ (process ⇀ msg)
  ⇒ process
  ⇒ pstate
  ⇒ bool
where
  next1 r p st msgs crd st′ ≡
      decide st′ = decide st
      ∧ x st′ = x st
      ∧ (∀ Q v. Q-prevotes-v msgs Q v
```

$\longrightarrow$ *mru-vote st′ = Some (three-phase r, v))*
$\wedge$ $(\neg$ $(\exists$ *Q v. Q-prevotes-v msgs Q v)*
$\longrightarrow$ *mru-vote st′ = mru-vote st)*

definition *send2* **where**
send2 r p q st $\equiv$ *case mru-vote st of*
None $\Rightarrow$ *Null*
| *Some* $(\Phi, v) \Rightarrow$ *if* Φ = *three-phase r then Vote v else Null*

definition *Q′-votes-v* **where**
Q′-votes-v r msgs Q Q′ v $\equiv$
$Q' \subseteq Q \wedge$ *card* $Q' > N$ *div 2* $\wedge$ $(\forall q \in Q'.$ *msgs q = Some (Vote v))*

definition *next2*
:: *nat*
$\Rightarrow$ *process*
$\Rightarrow$ *pstate*
$\Rightarrow$ *(process* $\rightharpoonup$ *msg)*
$\Rightarrow$ *process*
$\Rightarrow$ *pstate*
$\Rightarrow$ *bool*
where
next2 r p st msgs crd st′ $\equiv$ *let Q = dom msgs; lvs = msgs-to-lvs msgs in*
x st′ = x st
$\wedge$ *mru-vote st′ = mru-vote st*
$\wedge$ $(\forall$ *Q′ v. Q′-votes-v r msgs Q Q′ v* $\longrightarrow$ *decide st′ = Some v)*
$\wedge$ $(\neg(\exists$ *Q′ v. Q′-votes-v r msgs Q Q′ v* $\longrightarrow$ *decide st′ = decide st))*

definition *NA-sendMsg* :: *nat* $\Rightarrow$ *process* $\Rightarrow$ *process* $\Rightarrow$ *pstate* $\Rightarrow$ *msg* **where**
NA-sendMsg (r::nat) $\equiv$
if three-step r = 0 then send0 r
else if three-step r = 1 then send1 r
else send2 r

definition
NA-nextState :: *nat* $\Rightarrow$ *process* $\Rightarrow$ *pstate* $\Rightarrow$ *(process* $\rightharpoonup$ *msg)*
$\Rightarrow$ *process* $\Rightarrow$ *pstate* $\Rightarrow$ *bool*
where

NA-nextState r ≡
if three-step r = *0 then next0 r*
else if three-step r = *1 then next1 r*
else next2 r

17.2 The Heard-Of machine

definition
NA-commPerRd **where**
NA-commPerRd (*HOrs*::*process HO*) ≡ *True*

definition
NA-commGlobal **where**
NA-commGlobal HOs ≡
∃ *ph*::*nat*. ∀ *i* ∈ {*0*..*2*}.
(∀ *p*. *card* (*HOs* (*nr-steps*∗*ph*+*i*) *p*) > *N div 2*)
∧ (∀ *p q*. *HOs* (*nr-steps*∗*ph*+*i*) *p* = *HOs* (*nr-steps*∗*ph*) *q*)

definition *New-Algo-Alg* **where**
New-Algo-Alg ≡
(| *CinitState* = *NA-initState*,
sendMsg = *NA-sendMsg*,
CnextState = *NA-nextState* |)

definition *New-Algo-HOMachine* **where**
New-Algo-HOMachine ≡
(| *CinitState* = *NA-initState*,
sendMsg = *NA-sendMsg*,
CnextState = *NA-nextState*,
HOcommPerRd = *NA-commPerRd*,
HOcommGlobal = *NA-commGlobal* |)

abbreviation
New-Algo-M ≡ (*New-Algo-HOMachine*::(*process*, *pstate*, *msg*) *HOMachine*)

end

17.3 Proofs

type-synonym *p-TS-state* = (*nat* × (*process* ⇒ *pstate*))

definition *New-Algo-TS* ::
(*round* ⇒ *process HO*) ⇒ (*round* ⇒ *process HO*) ⇒ (*round* ⇒ *process*) ⇒ *p-TS-state TS*
where
New-Algo-TS HOs SHOs crds = *CHO-to-TS New-Algo-Alg HOs SHOs* (*K o crds*)

lemmas *New-Algo-TS-defs* = *New-Algo-TS-def CHO-to-TS-def New-Algo-Alg-def CHOinitConfig-def*
NA-initState-def

definition *New-Algo-trans-step* **where**
New-Algo-trans-step HOs SHOs crds nxt-f snd-f stp ≡ ⋃ *r μ*.
{((*r*, *cfg*), (*Suc r*, *cfg′*))|*cfg cfg′*. *three-step r* = *stp* ∧ (∀ *p*.
μ p ∈ *get-msgs* (*snd-f r*) *cfg* (*HOs r*) (*SHOs r*) *p*
∧ *nxt-f r p* (*cfg p*) (*μ p*) (*crds r*) (*cfg′ p*)
)}

lemma *three-step-less-D*:
0 < *three-step r* ⟹ *three-step r* = *1* ∨ *three-step r* = *2*
by(*unfold three-step-def*, *arith*)

lemma *New-Algo-trans*:
CSHO-trans-alt NA-sendMsg NA-nextState HOs SHOs (*K* ∘ *crds*) =
New-Algo-trans-step HOs SHOs crds next0 send0 0
∪ *New-Algo-trans-step HOs SHOs crds next1 send1 1*
∪ *New-Algo-trans-step HOs SHOs crds next2 send2 2*

proof(*rule equalityI*)
show *CSHO-trans-alt NA-sendMsg NA-nextState HOs SHOs* (*K* ∘ *crds*)
⊆ *New-Algo-trans-step HOs SHOs crds next0 send0 0* ∪
New-Algo-trans-step HOs SHOs crds next1 send1 1 ∪
New-Algo-trans-step HOs SHOs crds next2 send2 2
by(*force simp add*: *CSHO-trans-alt-def NA-sendMsg-def NA-nextState-def*
New-Algo-trans-step-def K-def dest!: *three-step-less-D*)
next

show *New-Algo-trans-step HOs SHOs crds next0 send0 0* ∪
New-Algo-trans-step HOs SHOs crds next1 send1 1 ∪
New-Algo-trans-step HOs SHOs crds next2 send2 2
⊆ *CSHO-trans-alt NA-sendMsg NA-nextState HOs SHOs* (*K* ∘ *crds*)
by(*force simp add*: *CSHO-trans-alt-def NA-sendMsg-def NA-nextState-def*
New-Algo-trans-step-def K-def)
qed

type-synonym *rHO* = *nat* ⇒ *process HO*

17.3.1 Refinement

definition *new-algo-ref-rel* :: (*three-step-mru-state* × *p-TS-state*)*set* **where**
new-algo-ref-rel = {(*sa*, (*r*, *sc*)).
opt-mru-state.next-round sa = *r*
∧ *opt-mru-state.decisions sa* = *pstate.decide o sc*
∧ *opt-mru-state.mru-vote sa* = *pstate.mru-vote o sc*
∧ (*three-step r* = *Suc 0* ⟶ *three-step-mru-state.candidates sa* = *ran* (*prop-vote*
o sc))
}

Different types seem to be derived for the two *mru-vote-evolution* lemmas, so we state them separately.

lemma *mru-vote-evolution0*:
∀ *p. next0 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *mru-vote o s′* = *mru-vote o s*
apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x=x* **in** *allE*)
by(*auto simp add*: *next0-def next2-def Let-def*)

lemma *mru-vote-evolution2*:
∀ *p. next2 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *mru-vote o s′* = *mru-vote o s*
apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x=x* **in** *allE*)
by(*auto simp add*: *next0-def next2-def Let-def*)

lemma *decide-evolution*:
∀ *p. next0 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *decide o s* = *decide o s′*
∀ *p. next1 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *decide o s* = *decide o s′*
apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x=x* **in** *allE*)
by(*auto simp add*: *next0-def next1-def Let-def*)

lemma *msgs-mru-vote*:

assumes
μ p ∈ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
shows ((*msgs-to-lvs* (*μ p*)) |' *HOs r p*) = (*mru-vote o cfg*) |' *HOs r p* **using** *assms*
by(*auto simp add*: *get-msgs-benign send0-def restrict-map-def msgs-to-lvs-def map-comp-def intro*!: *ext split*: *option.split*)

lemma *step0-ref*:
{*new-algo-ref-rel*}
(⋃ *r C*. *majorities.opt-mru-step0 r C*),
New-Algo-trans-step HOs HOs crds next0 send0 0 {> *new-algo-ref-rel*}
proof(*clarsimp simp add*: *PO-rhoare-defs New-Algo-trans-step-def all-conj-distrib*)
fix *r sa sc sc′ μ*
assume *R*: (*sa*, (*r*, *sc*)) ∈ *new-algo-ref-rel*
and *r*: *three-step r* = *0*
and *μ*: ∀ *p*. *μ p* ∈ *get-msgs* (*send0 r*) *sc* (*HOs r*) (*HOs r*) *p*
and *nxt*: ∀ *p*. *next0 r p* (*sc p*) (*μ p*) (*crds r*) (*sc′ p*)
note *μnxt* = *μ nxt*
have *r-phase-step*: *nr-steps* ∗ *three-phase r* = *r* **using** *r three-phase-step*[*of r*]
by(*auto*)
define *C* **where** *C* = *ran* (*prop-vote o sc′*)
have *guard*: ∀ *cand*∈*C*. ∃ *Q*. *majorities.opt-mru-guard* (*mru-vote* ∘ *sc*) *Q cand*
proof(*simp add*: *C-def ran-def*, *safe*)
fix *p cand*
assume *Some*: *prop-vote* (*sc′ p*) = *Some cand*

let *?Q* = *HOs r p*
let *?lvs0* = *mru-vote o sc*

have *?Q* ∈ *majs* **using** *Some μnxt*[*THEN spec*, **where** *x*=*p*]
by(*auto simp add*: *Let-def majs-def next0-def get-msgs-dom*)

moreover have
map-option snd (*option-Max-by fst* (*ran* (*?lvs* |' *?Q*))) ∈ {*None*, *Some cand*}
using *Some nxt*[*THEN spec*, **where** *x*=*p*]
msgs-mru-vote[**where** *HOs*=*HOs* **and** *μ*=*μ*, *OF μ*[*THEN spec*, *of p*]]
get-msgs-dom[*OF μ*[*THEN spec*, *of p*]]
by(*auto simp add*: *next0-def Let-def split*: *option.split-asm*)

ultimately have *majorities.opt-mru-guard ?lvs0 ?Q cand*

```
    by(auto simp add: majorities.opt-mru-guard-def Let-def majorities.opt-mru-vote-def)
    thus ∃ Q. majorities.opt-mru-guard ?lvs0 Q cand
      by blast
  qed

  define sa' where sa' = sa(|
      next-round := Suc r,
      candidates := C
    |)
  have (sa, sa') ∈ majorities.opt-mru-step0 r C using R r nxt guard
    by(auto simp add: majorities.opt-mru-step0-def sa'-def new-algo-ref-rel-def)
  moreover have (sa', (Suc r, sc')) ∈ new-algo-ref-rel using R nxt
    apply(auto simp add: sa'-def new-algo-ref-rel-def intro!:
      mru-vote-evolution0[OF nxt, symmetric] decide-evolution(1)[OF nxt]
    )
       apply(auto simp add: Let-def C-def o-def intro!: ext)
    done
  ultimately show
    ∃ sa'. (∃ r C. (sa, sa') ∈ majorities.opt-mru-step0 r C)
        ∧ (sa', Suc r, sc') ∈ new-algo-ref-rel
    by blast
qed

lemma step1-ref:
  {new-algo-ref-rel}
    (⋃ r S v. majorities.opt-mru-step1 r S v),
    New-Algo-trans-step HOs HOs crds next1 send1 (Suc 0) {> new-algo-ref-rel}
proof(clarsimp simp add: PO-rhoare-defs New-Algo-trans-step-def all-conj-distrib)
  fix r sa sc sc' μ
  assume R: (sa, (r, sc)) ∈ new-algo-ref-rel
    and r: three-step r = Suc 0
    and μ: ∀ p. μ p ∈ get-msgs (send1 r) sc (HOs r) (HOs r) p
        and nxt: ∀ p. next1 r p (sc p) (μ p) (crds r) (sc' p)
  note μnxt = μ nxt

  define S where S = {p. mru-vote (sc' p) ≠ mru-vote (sc p)}

  have S: S ⊆ {p. ∃ Q v. Q ⊆ HOs r p
      ∧ (∀ q ∈ Q. prop-vote (sc q) = Some v)
      ∧ Q ∈ majs
```

$\wedge$ (*mru-vote* (sc' *p*) = *Some* (*three-phase r*, *v*))
}
proof(*safe*)
fix *p*
assume $p \in S$
then obtain *Q v*
where
$\forall\, q \in Q.\ \mu\ p\ q$ = *Some* (*PreVote v*)
and *maj-Q*: $Q \in$ *majs*
and *Q-HOs*: $Q \subseteq$ *dom* (μ *p*)
and *lv*: *mru-vote* (sc' *p*) = *Some* (*three-phase r*, *v*) (**is** *?LV v*)
using *nxt*[*THEN spec*, **where** *x*=*p*]
by(*clarsimp simp add*: *next1-def Let-def S-def majs-def Q-prevotes-v-def*)

then have
$\forall\, q \in Q$. *prop-vote* (*sc q*) = *Some v* (**is** *?P Q v*)
$Q \subseteq$ *HOs r p*
using μ[*THEN spec*, **where** *x*=*p*]
by(*auto simp add*: *get-msgs-benign send1-def restrict-map-def split*: *option.split-asm*)

with *maj-Q* **and** *lv* **show** $\exists\, Q\ v.\ Q \subseteq$ *HOs r p* $\wedge$ *?P Q v* $\wedge$ $Q \in$ *majs* $\wedge$ *?LV v* **by** *blast*
qed

obtain *v* **where**
v: $\forall\, p \in S$. *mru-vote* (sc' *p*) = *Some* (*three-phase r*, *v*) $\wedge$ $v \in$ *ran* (*prop-vote o sc*)
proof(*cases* $S = \{\}$)
case *False*
assume *asm*:
$\bigwedge v.\ \forall\, p{\in}S$. *pstate.mru-vote* ($sc'$ *p*) = *Some* (*three-phase r*, *v*) $\wedge$ $v \in$ *ran* (*prop-vote o sc*)
$\Longrightarrow$ *thesis*
from *False* **obtain** *p* **where** $p \in S$ **by** *auto*
with *S nxt*
obtain *Q v*
where *prop-vote*: ($\forall\, q \in Q$. *prop-vote* (*sc q*) = *Some v*) **and** *maj-Q*: $Q \in$ *majs*
by *auto*

hence ∀ *p*∈*S*. *pstate.mru-vote* (*sc′ p*) = *Some* (*three-phase r*, *v*) (**is** *?LV*(*v*))
using *S*
by(*fastforce dest*!: *subsetD dest*: *majorities.qintersect*)

with *asm prop-vote maj-Q* **show** *thesis*
by (*metis all-not-in-conv comp-eq-dest-lhs majorities.empty-not-quorum ranI*)
qed(*auto*)

define *sa′* **where** *sa′* = *sa*⦇ *next-round* := *Suc r*,
opt-mru-state.mru-vote := *opt-mru-state.mru-vote sa* ++ *const-map* (*three-phase r*, *v*) *S*
⦈

have (*sa*, *sa′*) ∈ *majorities.opt-mru-step1 r S v* **using** *r R v*
by(*clarsimp simp add*: *majorities.opt-mru-step1-def sa′-def*
new-algo-ref-rel-def ball-conj-distrib)

moreover have (*sa′*, (*Suc r*, *sc′*)) ∈ *new-algo-ref-rel* **using** *r R*
proof−
have *mru-vote o sc′* = ((*mru-vote* ∘ *sc*) ++ *const-map* (*three-phase r*, *v*) *S*)
proof(*rule ext*, *simp*)
fix *p*
show *mru-vote* (*sc′ p*) = ((*mru-vote* ∘ *sc*) ++ *const-map* (*three-phase r*, *v*) *S*) *p*
using *v*
by(*auto simp add*: *S-def map-add-def const-map-is-None const-map-is-Some split*: *option.split*)
qed
thus *?thesis* **using** *R r nxt*
by(*force simp add*: *new-algo-ref-rel-def sa′-def three-step-Suc intro*: *decide-evolution*)
qed
ultimately show
∃ *sa′*. (∃ *r S v*. (*sa*, *sa′*) ∈ *majorities.opt-mru-step1 r S v*)
∧ (*sa′*, *Suc r*, *sc′*) ∈ *new-algo-ref-rel*
by *blast*
qed

lemma *step2-ref*:

{*new-algo-ref-rel*}
(⋃ *r dec-f. majorities.opt-mru-step2 r dec-f*),
New-Algo-trans-step HOs HOs crds next2 send2 2 {> *new-algo-ref-rel*}
proof(*clarsimp simp add*: *PO-rhoare-defs New-Algo-trans-step-def all-conj-distrib*)
fix *r sa sc sc′ μ*
assume *R*: (*sa*, (*r*, *sc*)) ∈ *new-algo-ref-rel*
and *r*: *three-step r* = *2*
and *μ*: ∀ *p*. *μ p* ∈ *get-msgs* (*send2 r*) *sc* (*HOs r*) (*HOs r*) *p*
and *nxt*: ∀ *p*. *next2 r p* (*sc p*) (*μ p*) (*crds r*) (*sc′ p*)
note *μnxt* = *μ nxt*

define *dec-f*
where *dec-f p* = (*if decide* (*sc′ p*) ≠ *decide* (*sc p*) *then decide* (*sc′ p*) *else None*) **for** *p*

have *dec-f*: (*decide* ∘ *sc*) ++ *dec-f* = *decide* ∘ *sc′*
proof
fix *p*
show ((*decide* ∘ *sc*) ++ *dec-f*) *p* = (*decide* ∘ *sc′*) *p* **using** *nxt*[*THEN spec*, *of p*]
by(*auto simp add*: *map-add-def dec-f-def next2-def Let-def split*: *option.split intro*!: *ext*)
qed

define *sa′* **where** *sa′* = *sa*⦇
next-round := *Suc r*,
decisions := *decisions sa* ++ *dec-f*
⦈

have (*sa′*, (*Suc r*, *sc′*)) ∈ *new-algo-ref-rel* **using** *R r nxt*
by(*auto simp add*: *new-algo-ref-rel-def sa′-def dec-f three-step-Suc mru-vote-evolution2*[*OF nxt*])
moreover have (*sa*, *sa′*) ∈ *majorities.opt-mru-step2 r dec-f* **using** *r R*
proof−
define *sc-r-votes* **where** *sc-r-votes p* = (*if* (∃ *v*. *mru-vote* (*sc p*) = *Some* (*three-phase r*, *v*))
then map-option snd (*mru-vote* (*sc p*))
else None) **for** *p*
have *sc-r-votes*: *sc-r-votes* = *majorities.r-votes sa r* **using** *R r*
by(*auto simp add*: *new-algo-ref-rel-def sc-r-votes-def majorities.r-votes-def*

intro!: *ext*)
  have *majorities.step2-d-guard dec-f sc-r-votes*
  proof(*clarsimp simp add*: *majorities.step2-d-guard-def*)
   fix *p v*
   assume *d-f-p*: *dec-f p* = *Some v*
   then obtain *Q* **where** *Q*:
    Q ∈ *majs*
    and *vote*: *Q* ⊆ *HOs r p* ∀ *q*∈*Q*. *μ p q* = *Some* (*Vote v*)
    using *nxt*[*THEN spec*, *of p*] *d-f-p*
    by(*auto simp add*: *next2-def dec-f-def Q′-votes-v-def Let-def majs-def*)
   have *mru-vote*: ∀ *q*∈*Q*. *mru-vote* (*sc q*) = *Some* (*three-phase r*, *v*) **using**
vote μ[*THEN spec*, *of p*]
   by(*fastforce simp add*: *get-msgs-benign send2-def sc-r-votes-def restrict-map-def*

    split: *option.split-asm if-split-asm*)
   hence *dom sc-r-votes* ∈ *majs*
    by(*auto intro*!: *majorities.mono-quorum*[*OF Q*] *simp add*: *sc-r-votes-def*)
  moreover have *v* ∈ *ran sc-r-votes* **using** *Q*[*THEN majorities.quorum-non-empty*]
mru-vote
    by(*force simp add*: *sc-r-votes-def ex-in-conv*[*symmetric*] *intro*: *ranI*)
   ultimately show *v* ∈ *ran sc-r-votes* ∧ *dom sc-r-votes* ∈ *majs*
    by *blast*
  qed

  thus *?thesis* **using** *r R*
   by(*auto simp add*: *majorities.opt-mru-step2-def sa′-def new-algo-ref-rel-def*
sc-r-votes)
 qed

 ultimately show
  ∃ *sa′*. (∃ *r dec-f*. (*sa*, *sa′*) ∈ *majorities.opt-mru-step2 r dec-f*)
   ∧ (*sa′*, *Suc r*, *sc′*) ∈ *new-algo-ref-rel*
  by *blast*

qed

lemma *New-Algo-Refines-votes*:
 PO-refines new-algo-ref-rel
  majorities.ts-mru-TS (*New-Algo-TS HOs HOs crds*)
proof(*rule refine-basic*)

show *init* (*New-Algo-TS HOs HOs crds*) ⊆ *new-algo-ref-rel* `` *init majorities.ts-mru-TS*
by(*auto simp add*: *New-Algo-TS-defs majorities.ts-mru-TS-defs new-algo-ref-rel-def*)
next
show
{*new-algo-ref-rel*} *TS.trans majorities.ts-mru-TS*,
TS.trans (*New-Algo-TS HOs HOs crds*) {> *new-algo-ref-rel*}
apply(*simp add*: *majorities.ts-mru-TS-defs New-Algo-TS-defs*)
apply(*auto simp add*: *CHO-trans-alt New-Algo-trans intro*!: *step0-ref step1-ref step2-ref*)
done
qed

17.3.2 Termination

theorem *New-Algo-termination*:
assumes *run*: *HORun New-Algo-Alg rho HOs*
and *commR*: ∀ *r*. *HOcommPerRd New-Algo-M* (*HOs r*)
and *commG*: *HOcommGlobal New-Algo-M HOs*
shows ∃ *r v*. *decide* (*rho r p*) = *Some v*
proof −
from *commG* **obtain** *ph* **where**
HOs: ∀ *i* ∈ {*0..2*}.
(∀ *p*. *card* (*HOs* (*nr-steps*∗*ph*+*i*) *p*) > *N div 2*)
∧ (∀ *p q*. *HOs* (*nr-steps*∗*ph*+*i*) *p* = *HOs* (*nr-steps*∗*ph*) *q*)
by(*auto simp add*: *New-Algo-HOMachine-def NA-commGlobal-def*)

— The tedious bit: obtain four consecutive rounds linked by send/next functions
define *r0* **where** *r0* = *nr-steps* ∗ *ph*
define *cfg0* **where** *cfg0* = *rho r0*
define *r1* **where** *r1* = *Suc r0*
define *cfg1* **where** *cfg1* = *rho r1*
define *r2* **where** *r2* = *Suc r1*
define *cfg2* **where** *cfg2* = *rho r2*
define *cfg3* **where** *cfg3* = *rho* (*Suc r2*)

from
run[*simplified HORun-def SHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r0*]
run[*simplified HORun-def SHORun-def*, *THEN CSHORun-step*, *THEN spec*,

where *x=r1*]
run[*simplified HORun-def SHORun-def*, *THEN CSHORun-step*, *THEN spec*,
where *x=r2*]
obtain $\mu 0$ $\mu 1$ $\mu 2$ **where**
send0: $\forall p.\ \mu 0\ p \in$ *get-msgs* (*send0 r0*) *cfg0* (*HOs r0*) (*HOs r0*) *p*
and *three-step0*: $\forall p.$ *next0 r0 p* (*cfg0 p*) ($\mu 0$ *p*) *undefined* (*cfg1 p*)
and *send1*: $\forall p.\ \mu 1\ p \in$ *get-msgs* (*send1 r1*) *cfg1* (*HOs r1*) (*HOs r1*) *p*
and *three-step1*: $\forall p.$ *next1 r1 p* (*cfg1 p*) ($\mu 1$ *p*) *undefined* (*cfg2 p*)
and *send2*: $\forall p.\ \mu 2\ p \in$ *get-msgs* (*send2 r2*) *cfg2* (*HOs r2*) (*HOs r2*) *p*
and *three-step2*: $\forall p.$ *next2 r2 p* (*cfg2 p*) ($\mu 2$ *p*) *undefined* (*cfg3 p*)
apply(*auto simp add*: *New-Algo-Alg-def three-step-def NA-nextState-def NA-sendMsg-def all-conj-distrib*
r0-def r1-def r2-def
cfg0-def cfg1-def cfg2-def cfg3-def mod-Suc
)
done

— The proof: everybody hears the same messages (non-empty!) in r0...

from *HOs*[*THEN bspec*, **where** *x=0*, *simplified*] *send0*
have
$\forall p\ q.\ \mu 0\ p = \mu 0\ q\ \forall p.\ N\ div\ 2 < card\ (dom\ (\mu 0\ p))$
apply(*auto simp add*: *get-msgs-benign send0-def r1-def r0-def dom-def restrict-map-def intro*!: *ext*)
apply(*blast*)+
done

— ...hence everybody sets *prop-vote* to the same value...
hence *same-prevote*:
$\forall p.$ *prop-vote* (*cfg1 p*) $\neq$ *None*
$\forall p\ q.$ *prop-vote* (*cfg1 p*) = *prop-vote* (*cfg1 q*) **using** *three-step0*
apply(*auto simp add*: *next1-def Let-def all-conj-distrib intro*!: *ext*)
apply(*clarsimp simp add*: *next0-def all-conj-distrib Let-def*)
apply(*clarsimp simp add*: *next0-def all-conj-distrib Let-def*)
by (*metis* (*full-types*) *dom-eq-empty-conv empty-iff majoritiesE'*)

— ...which will become our decision value.
then obtain *dec-v* **where** *dec-v*: $\forall p.$ *prop-vote* (*cfg1 p*) = *Some dec-v*
by (*metis option.collapse*)

— ...and since everybody hears from majority in r1...
from *HOs*[*THEN bspec*, **where** *x=Suc 0*, *simplified*] *send1*
have ∀ *p q*. *μ1 p* = *μ1 q* ∀ *p*. *N div 2* < *card* (*dom* (*μ1 p*))
apply(*auto simp add*: *get-msgs-benign send1-def r1-def r0-def dom-def restrict-map-def intro*!: *ext*)
apply(*blast*)+
done

— and since everybody casts a pre-vote for *dec-v*, everybody will vote *dec-v*
have *all-vote*: ∀ *p*. *mru-vote* (*cfg2 p*) = *Some* (*three-phase r2*, *dec-v*)
proof
fix *p*
have *r0-step*: *three-step r0* = *0*
by(*auto simp add*: *r0-def three-step-def*)
from *HOs*[*THEN bspec*, **where** *x=Suc 0*, *simplified*]
obtain *Q* **where** *Q*: *N div 2* < *card Q Q* ⊆ *HOs r1 p*
by(*auto simp add*: *r1-def r0-def*)
hence *Q-prevotes-v* (*μ1 p*) *Q dec-v* **using** *dec-v send1*[*THEN spec*, **where** *x=p*]
by(*auto simp add*: *Q-prevotes-v-def get-msgs-benign restrict-map-def send1-def*)
thus *mru-vote* (*cfg2 p*) = *Some* (*three-phase r2*, *dec-v*) **using** *three-step1*[*THEN spec*, **where** *x=p*] *r0-step*
by(*auto simp add*: *next1-def r2-def r1-def three-step-phase-Suc*)
qed

— And finally, everybody will also decide *dec-v*
have *all-decide*: ∀ *p*. *decide* (*cfg3 p*) = *Some dec-v*
proof
fix *p*
from *HOs*[*THEN bspec*, **where** *x=Suc* (*Suc 0*), *simplified*]
obtain *Q* **where** *Q*: *N div 2* < *card Q Q* ⊆ *HOs r2 p*
by(*auto simp add*: *r2-def r1-def r0-def*)
thus *decide* (*cfg3 p*) = *Some dec-v*
using *three-step2*[*THEN spec*, **where** *x=p*] *send2*[*THEN spec*, **where** *x=p*]
by(*auto simp add*: *next2-def send2-def Let-def*)
qed

thus *?thesis*
by(*auto simp add*: *cfg3-def*)
qed

end

18 The Paxos Algorithm

theory *Paxos-Defs*
imports *Heard-Of.HOModel ../Consensus-Types ../Consensus-Misc Three-Steps*
begin

This is a modified version (closer to the original Paxos) of PaxosDefs from the Heard Of entry in the AFP.

18.1 Model of the algorithm

The following record models the local state of a process.

record *'val pstate* =
  x :: *'val* — current value held by process
  mru-vote :: (*nat* × *'val*) *option*
  commt :: *'val option* — for coordinators: the value processes are asked to commit to
  decide :: *'val option* — value the process has decided on, if any

The algorithm relies on a coordinator for each phase of the algorithm. A phase lasts three rounds. The HO model formalization already provides the infrastructure for this, but unfortunately the coordinator is not passed to the *sendMsg* function. Using the infrastructure would thus require additional invariants and proofs; for simplicity, we use a global constant instead.

consts *coord* :: *nat* ⇒ *process*
specification (*coord*)
  coord-phase[*rule-format*]: ∀ *r r'*. *three-phase r* = *three-phase r'* ⟶ *coord r* = *coord r'*
  by(*auto*)

Possible messages sent during the execution of the algorithm.

datatype *'val msg* =
  ValStamp 'val nat
| *NeverVoted*
| *Vote 'val*

| *Null* — dummy message in case nothing needs to be sent

Characteristic predicates on messages.

definition *isValStamp* **where** *isValStamp m* ≡ ∃ *v ts*. *m* = *ValStamp v ts*

definition *isVote* **where** *isVote m* ≡ ∃ *v*. *m* = *Vote v*

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*

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

definition *Paxos-initState* **where**
  Paxos-initState p st crd ≡
   mru-vote st = *None*
   ∧ *commt st* = *None*
   ∧ *decide st* = *None*

definition *mru-vote-to-msg* :: *'val pstate* ⇒ *'val msg* **where**
  mru-vote-to-msg st ≡ *case mru-vote st of*
    Some (*ts*, *v*) ⇒ *ValStamp v ts*
    | *None* ⇒ *NeverVoted*

fun *msg-to-val-stamp* :: *'val msg* ⇒ (*round* × *'val*)*option* **where**
  msg-to-val-stamp (*ValStamp v ts*) = *Some* (*ts*, *v*)
  | *msg-to-val-stamp* - = *None*

definition *msgs-to-lvs* ::
  (*process* ⇀ *'val msg*)
  ⇒ (*process*, *round* × *'val*) *map*
where
  msgs-to-lvs msgs ≡ *msg-to-val-stamp* $\circ_m$ *msgs*

definition *send0* **where**
  send0 r p q st ≡
   if q = *coord r then mru-vote-to-msg st else Null*

definition *next0*
:: *nat*
⇒ *process*
⇒ *'val pstate*
⇒ (*process* ⇀ *'val msg*)
⇒ *process*
⇒ *'val pstate*
⇒ *bool*
where
next0 r p st msgs crd st' ≡ *let Q* = *dom msgs*; *lvs* = *msgs-to-lvs msgs in*
if p = *coord r* ∧ *card Q* > *N div 2*
then (*st'* = *st* ⦇ *commt* := *Some* (*case-option* (*x st*) *snd* (*option-Max-by fst* (*ran* (*lvs* |' *Q*)))) ⦈)
else st' = *st*⦇ *commt* := *None* ⦈

definition *send1* **where**
send1 r p q st ≡
if p = *coord r* ∧ *commt st* ≠ *None then Vote* (*the* (*commt st*)) *else Null*

definition *next1*
:: *nat*
⇒ *process*
⇒ *'val pstate*
⇒ (*process* ⇀ *'val msg*)
⇒ *process*
⇒ *'val pstate*
⇒ *bool*
where
next1 r p st msgs crd st' ≡
if msgs (*coord r*) ≠ *None* ∧ *isVote* (*the* (*msgs* (*coord r*)))
then st' = *st* ⦇ *mru-vote* := *Some* (*three-phase r*, *val* (*the* (*msgs* (*coord r*)))) ⦈
else st' = *st*

definition *send2* **where**
send2 r p q st ≡ (*case mru-vote st of*
Some (*phs*, *v*) ⇒ (*if phs* = *three-phase r then Vote v else Null*)
| - ⇒ *Null*
)

— processes from which a vote was received

definition *votes-rcvd* **where**
  $votes\text{-}rcvd\ (msgs :: process \rightharpoonup {}'val\ msg) \equiv$
  $\{\ (q,\ v)\ .\ msgs\ q = Some\ (Vote\ v)\ \}$

definition *the-rcvd-vote* **where**
  $the\text{-}rcvd\text{-}vote\ (msgs :: process \rightharpoonup {}'val\ msg) \equiv SOME\ v.\ v \in snd\ `\ votes\text{-}rcvd\ msgs$

definition *next2* **where**
  $next2\ r\ p\ st\ msgs\ crd\ st' \equiv$
  $if\ card\ (votes\text{-}rcvd\ msgs) > N\ div\ 2$
  $then\ st' = st\ (\!|\ decide := Some\ (the\text{-}rcvd\text{-}vote\ msgs)\ |\!)$
  $else\ st' = st$

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

definition $Paxos\text{-}sendMsg :: nat \Rightarrow process \Rightarrow process \Rightarrow {}'val\ pstate \Rightarrow {}'val\ msg$
where
  $Paxos\text{-}sendMsg\ (r{::}nat) \equiv$
  $if\ three\text{-}step\ r = 0\ then\ send0\ r$
  $else\ if\ three\text{-}step\ r = 1\ then\ send1\ r$
  $else\ send2\ r$

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

definition
  Paxos-commPerRd **where**
  $Paxos\text{-}commPerRd\ r\ (HO{::}process\ HO)\ (crd{::}process\ coord) \equiv True$

definition
  Paxos-commGlobal **where**
  $Paxos\text{-}commGlobal\ HOs\ coords \equiv$

$\exists\, ph{::}nat.\ \exists\, c{::}process.$
$\quad coord\ (nr\text{-}steps*ph) = c$
$\quad \wedge\ card\ (HOs\ (nr\text{-}steps*ph)\ c) > N\ div\ 2$
$\quad \wedge\ (\forall\, p.\ c \in HOs\ (nr\text{-}steps*ph+1)\ p)$
$\quad \wedge\ (\forall\, p.\ card\ (HOs\ (nr\text{-}steps*ph+2)\ p) > N\ div\ 2)$

18.2 The *Paxos* Heard-Of machine

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

definition *Paxos-Alg* **where**
Paxos-Alg ≡
(| *CinitState* = *Paxos-initState*,
sendMsg = *Paxos-sendMsg*,
CnextState = *Paxos-nextState* |)

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

abbreviation
Paxos-M ≡ (*Paxos-CHOMachine*::(*process*, *'val pstate*, *'val msg*) *CHOMachine*)

end

18.3 Proofs

type-synonym *p-TS-state* = (*nat* × (*process* ⇒ (*val pstate*)))

definition *Paxos-TS* ::
(*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process*)
⇒ *p-TS-state TS*
where

Paxos-TS HOs SHOs crds = *CHO-to-TS Paxos-Alg HOs SHOs* (*K o crds*)

lemmas *Paxos-TS-defs* = *Paxos-TS-def CHO-to-TS-def Paxos-Alg-def CHOinit-Config-def*
Paxos-initState-def

definition *Paxos-trans-step* **where**
Paxos-trans-step HOs SHOs crds nxt-f snd-f stp ≡ ⋃ *r μ*.
{((*r*, *cfg*), (*Suc r*, *cfg'*))|*cfg cfg'*. *three-step r* = *stp* ∧ (∀ *p*.
μ p ∈ *get-msgs* (*snd-f r*) *cfg* (*HOs r*) (*SHOs r*) *p*
∧ *nxt-f r p* (*cfg p*) (*μ p*) (*crds r*) (*cfg' p*)
)}

lemma *three-step-less-D*:
0 < *three-step r* ⟹ *three-step r* = *1* ∨ *three-step r* = *2*
by(*unfold three-step-def*, *arith*)

lemma *Paxos-trans*:
CSHO-trans-alt Paxos-sendMsg Paxos-nextState HOs SHOs (*K* ∘ *crds*) =
Paxos-trans-step HOs SHOs crds next0 send0 0
∪ *Paxos-trans-step HOs SHOs crds next1 send1 1*
∪ *Paxos-trans-step HOs SHOs crds next2 send2 2*

proof(*rule equalityI*)
show *CSHO-trans-alt Paxos-sendMsg Paxos-nextState HOs SHOs* (*K* ∘ *crds*)
⊆ *Paxos-trans-step HOs SHOs crds next0 send0 0* ∪
Paxos-trans-step HOs SHOs crds next1 send1 1 ∪
Paxos-trans-step HOs SHOs crds next2 send2 2
by(*force simp add*: *CSHO-trans-alt-def Paxos-sendMsg-def Paxos-nextState-def*
Paxos-trans-step-def K-def dest!: *three-step-less-D*)
next
show *Paxos-trans-step HOs SHOs crds next0 send0 0* ∪
Paxos-trans-step HOs SHOs crds next1 send1 1 ∪
Paxos-trans-step HOs SHOs crds next2 send2 2
⊆ *CSHO-trans-alt Paxos-sendMsg Paxos-nextState HOs SHOs* (*K* ∘ *crds*)
by(*force simp add*: *CSHO-trans-alt-def Paxos-sendMsg-def Paxos-nextState-def*
Paxos-trans-step-def K-def)
qed

type-synonym *rHO* = *nat* ⇒ *process HO*

18.3.1 Refinement

definition *coord-vote-to-set* :: *nat* ⇒ (*process* ⇒ (*val pstate*)) ⇒ *val set* **where**
  coord-vote-to-set r sc ≡ (*let v* = *pstate.commt* (*sc* (*coord r*)) *in*
    if v = *None*
      then {}
      else {*the v*})

definition *paxos-ref-rel* :: (*three-step-mru-state* × *p-TS-state*)*set* **where**
  paxos-ref-rel = {(*sa*, (*r*, *sc*)).
    opt-mru-state.next-round sa = *r*
    ∧ *opt-mru-state.decisions sa* = *pstate.decide o sc*
    ∧ *opt-mru-state.mru-vote sa* = *pstate.mru-vote o sc*
   ∧ (*three-step r* = *Suc 0* ⟶ *three-step-mru-state.candidates sa* = *coord-vote-to-set r sc*)
  }

lemma *mru-vote-evolution0*:
  ∀ *p. next0 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *mru-vote o s′* = *mru-vote o s*
  apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x*=*x* **in** *allE*)
  by(*auto simp add*: *next0-def next2-def Let-def*)

lemma *mru-vote-evolution2*:
  ∀ *p. next2 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *mru-vote o s′* = *mru-vote o s*
  apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x*=*x* **in** *allE*)
  by(*auto simp add*: *next0-def next2-def Let-def*)

lemma *decide-evolution*:
  ∀ *p. next0 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *decide o s* = *decide o s′*
  ∀ *p. next1 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *decide o s* = *decide o s′*
  apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x*=*x* **in** *allE*)
  by(*auto simp add*: *next0-def next1-def Let-def*)

lemma *msgs-mru-vote*:
  assumes
  μ (*coord r*) ∈ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) (*coord r*) (**is** *μ ?p* ∈ -)
  shows ((*msgs-to-lvs* (*μ ?p*)) |‘ *HOs r ?p*) = (*mru-vote o cfg*) |‘ *HOs r ?p* **using** *assms*
  by(*auto simp add*: *get-msgs-benign send0-def restrict-map-def msgs-to-lvs-def*
    mru-vote-to-msg-def map-comp-def intro!: *ext split*: *option.split*)

```
lemma step0-ref:
  {paxos-ref-rel}
    (⋃ r C. majorities.opt-mru-step0 r C),
    Paxos-trans-step HOs HOs crds next0 send0 0 {> paxos-ref-rel}
proof(clarsimp simp add: PO-rhoare-defs Paxos-trans-step-def all-conj-distrib)
  fix r sa sc sc' μ
  assume R: (sa, (r, sc)) ∈ paxos-ref-rel
    and r: three-step r = 0
    and μ: ∀ p. μ p ∈ get-msgs (send0 r) sc (HOs r) (HOs r) p
        and nxt: ∀ p. next0 r p (sc p) (μ p) (crds r) (sc' p)
  note μnxt = μ nxt
  from r have same-coord: coord (Suc r) = coord r
    by(auto simp add: three-step-phase-Suc intro: coord-phase)
  define C where C = coord-vote-to-set (Suc r) sc'
  have guard: ∀ cand∈C. ∃ Q. majorities.opt-mru-guard (mru-vote ∘ sc) Q cand
  proof
    fix cand
    assume cand: cand ∈ C
     hence Some: commt (sc' (coord r)) = Some cand using nxt[THEN spec,
where x=coord r]
      by(auto simp add: C-def coord-vote-to-set-def Let-def same-coord)

    let ?Q = HOs r (coord r)
    let ?lvs0 = mru-vote o sc

    have ?Q ∈ majs using Some μnxt[THEN spec, where x=coord r]
      by(auto simp add: Let-def majs-def next0-def same-coord get-msgs-dom)
    moreover have
     map-option snd (option-Max-by fst (ran (?lvs |` ?Q))) ∈ {None, Some cand}
      using Some nxt[THEN spec, where x=coord r]
        msgs-mru-vote[where HOs=HOs and μ=μ, OF μ[THEN spec, where
x=coord r]]
      get-msgs-dom[OF μ[THEN spec, of coord r]]
      by(auto simp add: next0-def Let-def split: option.split-asm)
    ultimately have majorities.opt-mru-guard ?lvs0 ?Q cand
    by(auto simp add: majorities.opt-mru-guard-def Let-def majorities.opt-mru-vote-def)
    thus ∃ Q. majorities.opt-mru-guard ?lvs0 Q cand
      by blast
  qed
```

```
  define sa' where sa' = sa⦇
      next-round := Suc r,
      candidates := C
    ⦈
  have (sa, sa') ∈ majorities.opt-mru-step0 r C using R r nxt guard
    by(auto simp add: majorities.opt-mru-step0-def sa'-def paxos-ref-rel-def)
  moreover have (sa', (Suc r, sc')) ∈ paxos-ref-rel using R nxt
    apply(auto simp add: sa'-def paxos-ref-rel-def intro!:
      mru-vote-evolution0[OF nxt, symmetric] decide-evolution(1)[OF nxt])
       apply(auto simp add: Let-def C-def o-def intro!: ext)
    done
  ultimately show
    ∃ sa'. (∃ r C. (sa, sa') ∈ majorities.opt-mru-step0 r C)
        ∧ (sa', Suc r, sc') ∈ paxos-ref-rel
    by blast
qed

lemma step1-ref:
  {paxos-ref-rel}
    (⋃ r S v. majorities.opt-mru-step1 r S v),
    Paxos-trans-step HOs HOs crds next1 send1 (Suc 0) {> paxos-ref-rel}
proof(clarsimp  simp add: PO-rhoare-defs Paxos-trans-step-def all-conj-distrib)
  fix r sa sc sc' μ
  assume R: (sa, (r, sc)) ∈ paxos-ref-rel
    and r: three-step r = Suc 0
    and μ: ∀ p. μ p ∈ get-msgs (send1 r) sc (HOs r) (HOs r) p
        and nxt: ∀ p. next1 r p (sc p) (μ p) (crds r) (sc' p)
  note μnxt = μ nxt

  define v where v = the (commt (sc (coord r)))
  define S where S = {p. coord r ∈ HOs r p ∧ commt (sc (coord r)) ≠ None}
  define sa' where sa' = sa⦇ next-round := Suc r,
   opt-mru-state.mru-vote := opt-mru-state.mru-vote sa ++ const-map (three-phase
r, v) S
  ⦈
  have (sa, sa') ∈ majorities.opt-mru-step1 r S v using r R
    by(clarsimp simp add: majorities.opt-mru-step1-def sa'-def  S-def v-def
      coord-vote-to-set-def paxos-ref-rel-def)
  moreover have (sa', (Suc r, sc')) ∈ paxos-ref-rel using r R
```

proof−
have *mru-vote o sc′* = ((*mru-vote* ∘ *sc*) ++ *const-map* (*three-phase r*, *v*) *S*)
proof(*rule ext*, *simp*)
fix *p*
show *mru-vote* (*sc′ p*) = ((*mru-vote* ∘ *sc*) ++ *const-map* (*three-phase r*, *v*) *S*) *p*
using *μnxt*[*THEN spec*, *of p*]
by(*auto simp add*: *get-msgs-benign next1-def send1-def S-def v-def map-add-def*

const-map-is-None const-map-is-Some restrict-map-def isVote-def split: *option.split*)
qed
thus *?thesis* **using** *R r nxt*
by(*force simp add*: *paxos-ref-rel-def sa′-def three-step-Suc intro*: *decide-evolution*)
qed
ultimately show
∃ *sa′*. (∃ *r S v*. (*sa*, *sa′*) ∈ *majorities.opt-mru-step1 r S v*)
∧ (*sa′*, *Suc r*, *sc′*) ∈ *paxos-ref-rel*
by *blast*
qed

lemma *step2-ref*:
{*paxos-ref-rel*}
(⋃ *r dec-f*. *majorities.opt-mru-step2 r dec-f*),
Paxos-trans-step HOs HOs crds next2 send2 2 {> *paxos-ref-rel*}
proof(*clarsimp simp add*: *PO-rhoare-defs Paxos-trans-step-def all-conj-distrib*)
fix *r sa sc sc′ μ*
assume *R*: (*sa*, (*r*, *sc*)) ∈ *paxos-ref-rel*
and *r*: *three-step r* = *2*
and *μ*: ∀ *p*. *μ p* ∈ *get-msgs* (*send2 r*) *sc* (*HOs r*) (*HOs r*) *p*
and *nxt*: ∀ *p*. *next2 r p* (*sc p*) (*μ p*) (*crds r*) (*sc′ p*)
note *μnxt* = *μ nxt*

define *dec-f*
where *dec-f p* = (*if decide* (*sc′ p*) ≠ *decide* (*sc p*) *then decide* (*sc′ p*) *else None*) **for** *p*

have *dec-f*: (*decide* ∘ *sc*) ++ *dec-f* = *decide* ∘ *sc′*
proof
fix *p*

show ((*decide* ∘ *sc*) ++ *dec-f*) *p* = (*decide* ∘ *sc'*) *p* **using** *nxt*[*THEN spec*, *of p*]
by(*auto simp add*: *map-add-def dec-f-def next2-def split*: *option.split intro*!: *ext*)
qed

define *sa'* **where** *sa'* = *sa*⦇
next-round := *Suc r*,
decisions := *decisions sa* ++ *dec-f*
⦈

have (*sa'*, (*Suc r*, *sc'*)) ∈ *paxos-ref-rel* **using** *R r nxt*
by(*auto simp add*: *paxos-ref-rel-def sa'-def dec-f three-step-Suc mru-vote-evolution2*[*OF nxt*])
moreover have (*sa*, *sa'*) ∈ *majorities.opt-mru-step2 r dec-f* **using** *r R*
proof−
define *sc-r-votes* **where** *sc-r-votes p* = (*if* (∃ *v*. *mru-vote* (*sc p*) = *Some* (*three-phase r*, *v*))
then map-option snd (*mru-vote* (*sc p*))
else None) **for** *p*
have *sc-r-votes*: *sc-r-votes* = *majorities.r-votes sa r* **using** *R r*
by(*auto simp add*: *paxos-ref-rel-def sc-r-votes-def majorities.r-votes-def intro*!: *ext*)
have *majorities.step2-d-guard dec-f sc-r-votes*
proof(*clarsimp simp add*: *majorities.step2-d-guard-def*)
fix *p v*
assume *d-f-p*: *dec-f p* = *Some v*
let *?Qv* = *votes-rcvd* (*μ p*)
have *Qv*: *card ?Qv* > *N div 2*
v = *the-rcvd-vote* (*μ p*) **using** *nxt*[*THEN spec*, *of p*] *d-f-p*
by(*auto simp add*: *next2-def dec-f-def*)

hence *v* ∈ *snd* ‘ *votes-rcvd* (*μ p*)
by(*fastforce simp add*: *the-rcvd-vote-def ex-in-conv*[*symmetric*]
dest!: *card-gt-0-iff*[*THEN iffD1*, *OF le-less-trans*[*OF le0*]] *elim*!: *imageI intro*: *someI*)
moreover have *?Qv* = *map-graph* (*sc-r-votes*) ∩ (*HOs r p* × *UNIV*) **using** *μ*[*THEN spec*, *of p*]
by(*auto simp add*: *get-msgs-benign send2-def restrict-map-def votes-rcvd-def sc-r-votes-def image-def split*: *option.split-asm*)

ultimately show *v* ∈ *ran sc-r-votes* ∧ *dom sc-r-votes* ∈ *majs* **using** *Qv*(*1*)
by(*auto simp add*: *majs-def inj-on-def map-graph-def fun-graph-def sc-r-votes-def the-rcvd-vote-def majs-def intro*: *ranI*
elim!: *less-le-trans intro*!: *card-inj-on-le*[**where** *f*=*fst*])
qed

thus *?thesis* **using** *r R*
by(*auto simp add*: *majorities.opt-mru-step2-def sa′-def paxos-ref-rel-def sc-r-votes*)
qed

ultimately show
∃ *sa′*. (∃ *r dec-f*. (*sa*, *sa′*) ∈ *majorities.opt-mru-step2 r dec-f*)
∧ (*sa′*, *Suc r*, *sc′*) ∈ *paxos-ref-rel*
by *blast*

qed

lemma *Paxos-Refines-ThreeStep-MRU*:
PO-refines paxos-ref-rel
majorities.ts-mru-TS (*Paxos-TS HOs HOs crds*)
proof(*rule refine-basic*)
show *init* (*Paxos-TS HOs HOs crds*) ⊆ *paxos-ref-rel* ‘‘ *init majorities.ts-mru-TS*
by(*auto simp add*: *Paxos-TS-defs majorities.ts-mru-TS-def paxos-ref-rel-def majorities.ts-mru-init-def*)
next
show
{*paxos-ref-rel*} *TS.trans majorities.ts-mru-TS*,
TS.trans (*Paxos-TS HOs HOs crds*) {> *paxos-ref-rel*}
apply(*simp add*: *majorities.ts-mru-TS-defs Paxos-TS-defs*)
apply(*auto simp add*: *CHO-trans-alt Paxos-trans intro*!: *step0-ref step1-ref step2-ref*)
done
qed

18.3.2 Termination

theorem *Paxos-termination*:
assumes *run*: *CHORun Paxos-Alg rho HOs crds*
and *commR*: ∀ *r*. *CHOcommPerRd Paxos-M r* (*HOs r*) (*crds r*)

and *commG*: *CHOcommGlobal Paxos-M HOs crds*
shows $\exists$ *r v. decide* (*rho r p*) = *Some v*
proof –
from *commG* **obtain** *ph c* **where**
HOs:
coord (*nr-steps*∗*ph*) = *c*
∧ *card* (*HOs* (*nr-steps*∗*ph*) *c*) > *N div 2*
∧ (∀ *p*. *c* ∈ *HOs* (*nr-steps*∗*ph*+*1*) *p*)
∧ (∀ *p*. *card* (*HOs* (*nr-steps*∗*ph*+*2*) *p*) > *N div 2*)
by(*auto simp add*: *Paxos-CHOMachine-def Paxos-commGlobal-def*)

— The tedious bit: obtain three consecutive rounds linked by send/next functions
define *r0* **where** *r0* = *nr-steps* ∗ *ph*
define *cfg0* **where** *cfg0* = *rho r0*
define *r1* **where** *r1* = *Suc r0*
define *cfg1* **where** *cfg1* = *rho r1*
define *r2* **where** *r2* = *Suc r1*
define *cfg2* **where** *cfg2* = *rho r2*
define *cfg3* **where** *cfg3* = *rho* (*Suc r2*)

from
run[*simplified CHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r0*]

run[*simplified CHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r1*]
run[*simplified CHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r2*]

obtain *μ0 μ1 μ2* **where**
send0: ∀ *p*. *μ0 p* ∈ *get-msgs* (*send0 r0*) *cfg0* (*HOs r0*) (*HOs r0*) *p*
and *three-step0*: ∀ *p*. *next0 r0 p* (*cfg0 p*) (*μ0 p*) (*crds* (*Suc r0*) *p*) (*cfg1 p*)
and *send1*: ∀ *p*. *μ1 p* ∈ *get-msgs* (*send1 r1*) *cfg1* (*HOs r1*) (*HOs r1*) *p*
and *three-step1*: ∀ *p*. *next1 r1 p* (*cfg1 p*) (*μ1 p*) (*crds* (*Suc r1*) *p*) (*cfg2 p*)
and *send2*: ∀ *p*. *μ2 p* ∈ *get-msgs* (*send2 r2*) *cfg2* (*HOs r2*) (*HOs r2*) *p*
and *three-step2*: ∀ *p*. *next2 r2 p* (*cfg2 p*) (*μ2 p*) (*crds* (*Suc r2*) *p*) (*cfg3 p*)
apply(*auto simp add*: *Paxos-Alg-def three-step-def Paxos-nextState-def Paxos-sendMsg-def all-conj-distrib*
r0-def r1-def r2-def
cfg0-def cfg1-def cfg2-def cfg3-def mod-Suc
)
done

— The proof: the coordinator hears enough messages in r0 and thus selects a value.

from *HOs three-step0*[*THEN spec*, *of c*] *send0*[*THEN spec*, *of c*]
have
commt (*cfg1 c*) $\neq$ *None*
by(*auto simp add*: *next0-def Let-def r0-def get-msgs-dom*)

then obtain *dec-v* **where** *dec-v*: *commt* (*cfg1 c*) = *Some dec-v*
by (*metis option.collapse*)

have *step-r0*: *three-step r0* = *0*
by(*auto simp add*: *r0-def three-step-def*)
hence *same-coord*:
coord r1 = *coord r0*
coord r2 = *coord r0*
by(*auto simp add*: *three-step-phase-Suc r2-def r1-def r0-def intro*!: *coord-phase*)

— All processes hear from the coordinator, and thus set their vote to *dec-v*.
hence *all-vote*: $\forall$ *p*. *mru-vote* (*cfg2 p*) = *Some* (*three-phase r2*, *dec-v*)
using *HOs three-step1 send1 step-r0 dec-v*
by(*auto simp add*: *next1-def Let-def get-msgs-benign send1-def restrict-map-def isVote-def*
r2-def r1-def r0-def[*symmetric*] *same-coord*[*simplified r2-def r1-def*] *three-step-phase-Suc*)

— And finally, everybody will also decide *dec-v*.
have *all-decide*: $\forall$ *p*. *decide* (*cfg3 p*) = *Some dec-v*
proof
fix *p*
have *votes-rcvd* (μ*2 p*) = *HOs r2 p* $\times$ {*dec-v*} **using** *send2*[*THEN spec*, **where** *x*=*p*] *all-vote*
by(*auto simp add*: *send2-def get-msgs-benign votes-rcvd-def restrict-map-def image-def o-def*)

moreover from *HOs* **have** *N div 2* < *card* (*HOs r2 p*)
by(*auto simp add*: *r2-def r1-def r0-def*)

moreover then have *HOs r2 p* $\neq$ {}
by (*metis card.empty less-nat-zero-code*)
ultimately show *decide* (*cfg3 p*) = *Some dec-v*

using *three-step2*[*THEN spec*, **where** *x*=*p*] *send2*[*THEN spec*, **where** *x*=*p*] *all-vote*
by(*auto simp add*: *next2-def send2-def Let-def get-msgs-benign the-rcvd-vote-def restrict-map-def image-def o-def*)
qed

thus *?thesis*
by(*auto simp add*: *cfg3-def*)

qed

end

19 Chandra-Toueg $\diamond S$ Algorithm

theory *CT-Defs*
imports *Heard-Of.HOModel ../Consensus-Types ../Consensus-Misc Three-Steps*
begin

The following record models the local state of a process.

record *′val pstate* =
x :: *′val* — current value held by process
mru-vote :: (*nat* × *′val*) *option*
commt :: *′val* — for coordinators: the value processes are asked to commit to
decide :: *′val option* — value the process has decided on, if any

The algorithm relies on a coordinator for each phase of the algorithm. A phase lasts three rounds. The HO model formalization already provides the infrastructure for this, but unfortunately the coordinator is not passed to the *sendMsg* function. Using the infrastructure would thus require additional invariants and proofs; for simplicity, we use a global constant instead.

consts *coord* :: *nat* ⇒ *process*
specification (*coord*)
coord-phase[*rule-format*]: ∀ *r r′*. *three-phase r* = *three-phase r′* ⟶ *coord r* = *coord r′*
by(*auto*)

Possible messages sent during the execution of the algorithm.

datatype *′val msg* =

ValStamp 'val nat
| *NeverVoted*
| *Vote 'val*
| *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** $isVote\ m \equiv \exists\, v.\ m = Vote\ v$

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

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

definition *CT-initState* **where**
  $CT\text{-}initState\ p\ st\ crd \equiv$
   $mru\text{-}vote\ st = None$
   $\wedge\ decide\ st = None$

definition $mru\text{-}vote\text{-}to\text{-}msg :: 'val\ pstate \Rightarrow 'val\ msg$ **where**
  $mru\text{-}vote\text{-}to\text{-}msg\ st \equiv case\ mru\text{-}vote\ st\ of$
    $Some\ (ts, v) \Rightarrow ValStamp\ v\ ts$
   | $None \Rightarrow NeverVoted$

fun $msg\text{-}to\text{-}val\text{-}stamp :: 'val\ msg \Rightarrow (round \times 'val) option$ **where**
  $msg\text{-}to\text{-}val\text{-}stamp\ (ValStamp\ v\ ts) = Some\ (ts, v)$
  | $msg\text{-}to\text{-}val\text{-}stamp\ \text{-} = None$

definition *msgs-to-lvs* ::
  $(process \rightharpoonup 'val\ msg)$
  $\Rightarrow (process, round \times 'val)\ map$
where
  $msgs\text{-}to\text{-}lvs\ msgs \equiv msg\text{-}to\text{-}val\text{-}stamp \circ_m msgs$

definition *send0* **where**

send0 r p q st ≡
if q = *coord r then mru-vote-to-msg st else Null*

definition *next0*
:: *nat*
⇒ *process*
⇒ *'val pstate*
⇒ (*process* ⇀ *'val msg*)
⇒ *process*
⇒ *'val pstate*
⇒ *bool*
where
next0 r p st msgs crd st' ≡ *let Q* = *dom msgs*; *lvs* = *msgs-to-lvs msgs in*
if p = *coord r*
then (*st'* = *st* ⦇ *commt* := (*case-option* (*x st*) *snd* (*option-Max-by fst* (*ran* (*lvs* |' *Q*)))) ⦈)
else st' = *st*

definition *send1* **where**
send1 r p q st ≡
if p = *coord r then Vote* (*commt st*) *else Null*

definition *next1*
:: *nat*
⇒ *process*
⇒ *'val pstate*
⇒ (*process* ⇀ *'val msg*)
⇒ *process*
⇒ *'val pstate*
⇒ *bool*
where
next1 r p st msgs crd st' ≡
if msgs (*coord r*) ≠ *None*
then st' = *st* ⦇ *mru-vote* := *Some* (*three-phase r*, *val* (*the* (*msgs* (*coord r*)))) ⦈
else st' = *st*

definition *send2* **where**
send2 r p q st ≡ (*case mru-vote st of*
Some (*phs*, *v*) ⇒ (*if phs* = *three-phase r then Vote v else Null*)
| - ⇒ *Null*

)

— processes from which a vote was received

definition *votes-rcvd* **where**

votes-rcvd (*msgs* :: *process* ⇀ *'val msg*) ≡
{ (*q*, *v*) . *msgs q* = *Some* (*Vote v*) }

definition *the-rcvd-vote* **where**

the-rcvd-vote (*msgs* :: *process* ⇀ *'val msg*) ≡ *SOME v*. *v* ∈ *snd* ' *votes-rcvd msgs*

definition *next2* **where**

next2 r p st msgs crd st' ≡
if card (*votes-rcvd msgs*) > *N div 2*
then st' = *st* ⦇ *decide* := *Some* (*the-rcvd-vote msgs*) ⦈
else st' = *st*

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

definition *CT-sendMsg* :: *nat* ⇒ *process* ⇒ *process* ⇒ *'val pstate* ⇒ *'val msg* **where**

CT-sendMsg (*r*::*nat*) ≡
if three-step r = *0 then send0 r*
else if three-step r = *1 then send1 r*
else send2 r

definition

CT-nextState :: *nat* ⇒ *process* ⇒ *'val pstate* ⇒ (*process* ⇀ *'val msg*)
⇒ *process* ⇒ *'val pstate* ⇒ *bool*

where

CT-nextState r ≡
if three-step r = *0 then next0 r*
else if three-step r = *1 then next1 r*
else next2 r

19.1 The *CT* Heard-Of machine

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

definition *CT-Alg* **where**

$CT\text{-}Alg \equiv$
$(\!|\ CinitState = CT\text{-}initState,$
$sendMsg = CT\text{-}sendMsg,$
$CnextState = CT\text{-}nextState\ |\!)$

The CT algorithm relies on *waiting*: in each round, the coordinator waits until it hears from $\frac{N}{2}$ processes. This it reflected in the following per-round predicate.

definition
$CT\text{-}commPerRd :: nat \Rightarrow process\ HO \Rightarrow process\ coord \Rightarrow bool$
where
$CT\text{-}commPerRd\ r\ HOs\ crds \equiv$
$three\text{-}step\ r = 0 \longrightarrow card\ (HOs\ (coord\ r)) > N\ div\ 2$

definition
$CT\text{-}commGlobal$ **where**
$CT\text{-}commGlobal\ HOs\ coords \equiv$
$\exists\, ph{::}nat.\ \exists\, c{::}process.$
$coord\ (nr\text{-}steps{*}ph) = c$
$\wedge\ (\forall\, p.\ c \in HOs\ (nr\text{-}steps{*}ph{+}1)\ p)$
$\wedge\ (\forall\, p.\ card\ (HOs\ (nr\text{-}steps{*}ph{+}2)\ p) > N\ div\ 2)$

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

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

end

19.2 Proofs

type-synonym $ct\text{-}TS\text{-}state = (nat \times (process \Rightarrow (val\ pstate)))$

definition $CT\text{-}TS$ ::

(*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process* ⇒ *process*)
⇒ *ct-TS-state TS*
where
CT-TS HOs SHOs crds = *CHO-to-TS CT-Alg HOs SHOs crds*

lemmas *CT-TS-defs* = *CT-TS-def CHO-to-TS-def CT-Alg-def CHOinitConfig-def CT-initState-def*

definition *CT-trans-step* **where**
CT-trans-step HOs SHOs crds nxt-f snd-f stp ≡ ⋃ *r* *μ*.
{((*r*, *cfg*), (*Suc r*, *cfg*′))|*cfg cfg*′. *three-step r* = *stp* ∧ (∀ *p*.
μ p ∈ *get-msgs* (*snd-f r*) *cfg* (*HOs r*) (*SHOs r*) *p*
∧ *nxt-f r p* (*cfg p*) (*μ p*) (*crds r p*) (*cfg*′ *p*)
)}

lemma *three-step-less-D*:
0 < *three-step r* ⟹ *three-step r* = *1* ∨ *three-step r* = *2*
by(*unfold three-step-def*, *arith*)

lemma *CT-trans*:
CSHO-trans-alt CT-sendMsg CT-nextState HOs SHOs crds =
CT-trans-step HOs SHOs crds next0 send0 0
∪ *CT-trans-step HOs SHOs crds next1 send1 1*
∪ *CT-trans-step HOs SHOs crds next2 send2 2*

proof(*rule equalityI*)
show *CSHO-trans-alt CT-sendMsg CT-nextState HOs SHOs crds*
⊆ *CT-trans-step HOs SHOs crds next0 send0 0* ∪
CT-trans-step HOs SHOs crds next1 send1 1 ∪
CT-trans-step HOs SHOs crds next2 send2 2
by(*force simp add*: *CSHO-trans-alt-def CT-sendMsg-def CT-nextState-def CT-trans-step-def K-def dest*!: *three-step-less-D*)
next
show *CT-trans-step HOs SHOs crds next0 send0 0* ∪
CT-trans-step HOs SHOs crds next1 send1 1 ∪
CT-trans-step HOs SHOs crds next2 send2 2
⊆ *CSHO-trans-alt CT-sendMsg CT-nextState HOs SHOs crds*
by(*force simp add*: *CSHO-trans-alt-def CT-sendMsg-def CT-nextState-def*

CT-trans-step-def K-def)
qed

type-synonym $rHO = nat \Rightarrow process\ HO$

19.2.1 Refinement

definition *ct-ref-rel* :: (*three-step-mru-state* × *ct-TS-state*)*set* **where**
ct-ref-rel = {(*sa*, (*r*, *sc*)).
opt-mru-state.next-round sa = *r*
∧ *opt-mru-state.decisions sa* = *pstate.decide o sc*
∧ *opt-mru-state.mru-vote sa* = *pstate.mru-vote o sc*
∧ (*three-step r* = *Suc 0* ⟶ *three-step-mru-state.candidates sa* = {*commt* (*sc* (*coord r*))})
}

Now we need to use the fact that SHOs = HOs (i.e. the setting is non-Byzantine), and also the fact that the coordinator receives enough messages in each round

lemma *mru-vote-evolution0*:
∀ *p*. *next0 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *mru-vote o s′* = *mru-vote o s*
apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x*=*x* **in** *allE*)
by(*auto simp add*: *next0-def next2-def Let-def*)

lemma *mru-vote-evolution2*:
∀ *p*. *next2 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *mru-vote o s′* = *mru-vote o s*
apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x*=*x* **in** *allE*)
by(*auto simp add*: *next0-def next2-def Let-def*)

lemma *decide-evolution*:
∀ *p*. *next0 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *decide o s* = *decide o s′*
∀ *p*. *next1 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *decide o s* = *decide o s′*
apply(*rule-tac*[!] *ext*, *rename-tac x*, *erule-tac*[!] *x*=*x* **in** *allE*)
by(*auto simp add*: *next0-def next1-def Let-def*)

lemma *msgs-mru-vote*:
assumes
μ (*coord r*) ∈ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) (*coord r*) (**is** μ *?p* ∈ -)
shows ((*msgs-to-lvs* (μ *?p*)) |‘ *HOs r ?p*) = (*mru-vote o cfg*) |‘ *HOs r ?p* **using** *assms*

by(*auto simp add*: *get-msgs-benign send0-def restrict-map-def msgs-to-lvs-def mru-vote-to-msg-def map-comp-def intro*!: *ext split*: *option.split*)

context
fixes
HOs :: *nat* ⇒ *process* ⇒ *process set*
and *crds* :: *nat* ⇒ *process* ⇒*process*
assumes
per-rd: ∀ *r*. *CT-commPerRd r* (*HOs r*) (*crds r*)
begin

lemma *step0-ref*:
{*ct-ref-rel*}
(⋃ *r C*. *majorities.opt-mru-step0 r C*),
CT-trans-step HOs HOs crds next0 send0 0 {> *ct-ref-rel*}
proof(*clarsimp simp add*: *PO-rhoare-defs CT-trans-step-def all-conj-distrib*)
fix *r sa sc sc′ μ*
assume *R*: (*sa*, (*r*, *sc*)) ∈ *ct-ref-rel*
and *r*: *three-step r* = *0*
and *μ*: ∀ *p*. *μ p* ∈ *get-msgs* (*send0 r*) *sc* (*HOs r*) (*HOs r*) *p*
and *nxt*: ∀ *p*. *next0 r p* (*sc p*) (*μ p*) (*crds r p*) (*sc′ p*)
note *μnxt* = *μ nxt*
have *r-phase-step*: *nr-steps* ∗ *three-phase r* = *r* **using** *r three-phase-step*[*of r*]
by(*auto*)
from *r* **have** *same-coord*: *coord* (*Suc r*) = *coord r*
by(*auto simp add*: *three-step-phase-Suc intro*: *coord-phase*)
define *cand* **where** *cand* = *commt* (*sc′* (*coord* (*Suc r*)))
define *C* **where** *C* = {*cand*}
have *guard*: ∀ *cand*∈*C*. ∃ *Q*. *majorities.opt-mru-guard* (*mru-vote* ∘ *sc*) *Q cand*
proof(*simp add*: *C-def*)
let *?Q* = *HOs r* (*coord r*)
let *?lvs0* = *mru-vote o sc*
have *?Q* ∈ *majs* **using** *per-rd μnxt*[*THEN spec*, **where** *x*=*coord r*]
per-rd[*simplified CT-commPerRd-def*, *rule-format*, *OF r*]
by(*auto simp add*: *Let-def majs-def next0-def same-coord get-msgs-dom r-phase-step*)
moreover have
map-option snd (*option-Max-by fst* (*ran* (*?lvs* |‘ *?Q*))) ∈ {*None*, *Some cand*}
using *nxt*[*THEN spec*, **where** *x*=*coord r*]
msgs-mru-vote[**where** *HOs*=*HOs* **and** *μ*=*μ*, *OF μ*[*THEN spec*, **where**

```
x=coord r]]
      get-msgs-dom[OF μ[THEN spec, of coord r]]
    by(auto simp add: next0-def Let-def cand-def same-coord split: option.split-asm)
    ultimately have majorities.opt-mru-guard ?lvs0 ?Q cand
    by(auto simp add: majorities.opt-mru-guard-def Let-def majorities.opt-mru-vote-def)
    thus ∃ Q. majorities.opt-mru-guard ?lvs0 Q cand
      by blast
  qed

  define sa' where sa' = sa(|
      next-round := Suc r,
      candidates := C
    |)
  have (sa, sa') ∈ majorities.opt-mru-step0 r C using R r nxt guard
    by(auto simp add: majorities.opt-mru-step0-def sa'-def ct-ref-rel-def)
  moreover have (sa', (Suc r, sc')) ∈ ct-ref-rel using R nxt
    apply(auto simp add: sa'-def ct-ref-rel-def intro!:
      mru-vote-evolution0[OF nxt, symmetric] decide-evolution(1)[OF nxt])
      apply(auto simp add: Let-def C-def cand-def o-def intro!: ext)
    done
  ultimately show
    ∃ sa'. (∃ r C. (sa, sa') ∈ majorities.opt-mru-step0 r C)
        ∧ (sa', Suc r, sc') ∈ ct-ref-rel
    by blast
qed

lemma step1-ref:
  {ct-ref-rel}
    (⋃ r S v. majorities.opt-mru-step1 r S v),
    CT-trans-step HOs HOs crds next1 send1 (Suc 0) {> ct-ref-rel}
proof(clarsimp  simp add: PO-rhoare-defs CT-trans-step-def all-conj-distrib)
  fix r sa sc sc' μ
  assume R: (sa, (r, sc)) ∈ ct-ref-rel
    and r: three-step r = Suc 0
    and μ: ∀ p. μ p ∈ get-msgs (send1 r) sc (HOs r) (HOs r) p
        and nxt: ∀ p. next1 r p (sc p) (μ p) (crds r p) (sc' p)
  note μnxt = μ nxt

  define v where v = commt (sc (coord r))
  define S where S = {p. coord r ∈ HOs r p}
```

define *sa′* **where** *sa′* = *sa*⦇ *next-round* := *Suc r*,
opt-mru-state.mru-vote := *opt-mru-state.mru-vote sa* ++ *const-map* (*three-phase r*, *v*) *S*
⦈
have (*sa*, *sa′*) ∈ *majorities.opt-mru-step1 r S v* **using** *r R*
by(*clarsimp simp add*: *majorities.opt-mru-step1-def sa′-def S-def v-def ct-ref-rel-def*)
moreover have (*sa′*, (*Suc r*, *sc′*)) ∈ *ct-ref-rel* **using** *r R*
proof−
have *mru-vote o sc′* = ((*mru-vote* ∘ *sc*) ++ *const-map* (*three-phase r*, *v*) *S*)
proof(*rule ext*, *simp*)
fix *p*
show *mru-vote* (*sc′ p*) = ((*mru-vote* ∘ *sc*) ++ *const-map* (*three-phase r*, *v*) *S*) *p*
using *μnxt*[*THEN spec*, *of p*]
by(*auto simp add*: *get-msgs-benign next1-def send1-def S-def v-def map-add-def*

const-map-is-None const-map-is-Some restrict-map-def isVote-def split: *option.split*)
qed
thus *?thesis* **using** *R r nxt*
by(*force simp add*: *ct-ref-rel-def sa′-def three-step-Suc intro*: *decide-evolution*)
qed
ultimately show
∃ *sa′*. (∃ *r S v*. (*sa*, *sa′*) ∈ *majorities.opt-mru-step1 r S v*)
∧ (*sa′*, *Suc r*, *sc′*) ∈ *ct-ref-rel*
by *blast*
qed

lemma *step2-ref*:
{*ct-ref-rel*}
(⋃ *r dec-f*. *majorities.opt-mru-step2 r dec-f*),
CT-trans-step HOs HOs crds next2 send2 2 {> *ct-ref-rel*}
proof(*clarsimp simp add*: *PO-rhoare-defs CT-trans-step-def all-conj-distrib*)
fix *r sa sc sc′ μ*
assume *R*: (*sa*, (*r*, *sc*)) ∈ *ct-ref-rel*
and *r*: *three-step r* = *2*
and *μ*: ∀ *p*. *μ p* ∈ *get-msgs* (*send2 r*) *sc* (*HOs r*) (*HOs r*) *p*
and *nxt*: ∀ *p*. *next2 r p* (*sc p*) (*μ p*) (*crds r p*) (*sc′ p*)
note *μnxt* = *μ nxt*

define *dec-f*
  where *dec-f p* = (*if decide* (*sc′ p*) ≠ *decide* (*sc p*) *then decide* (*sc′ p*) *else None*) **for** *p*

have *dec-f*: (*decide* ∘ *sc*) ++ *dec-f* = *decide* ∘ *sc′*
proof
  fix *p*
  show ((*decide* ∘ *sc*) ++ *dec-f*) *p* = (*decide* ∘ *sc′*) *p* **using** *nxt*[*THEN spec*, *of p*]
    by(*auto simp add*: *map-add-def dec-f-def next2-def split*: *option.split intro*!: *ext*)
qed

define *sa′* **where** *sa′* = *sa*⦇
  next-round := *Suc r*,
  decisions := *decisions sa* ++ *dec-f*
⦈

have (*sa′*, (*Suc r*, *sc′*)) ∈ *ct-ref-rel* **using** *R r nxt*
  by(*auto simp add*: *ct-ref-rel-def sa′-def dec-f three-step-Suc*
    mru-vote-evolution2[*OF nxt*])
moreover have (*sa*, *sa′*) ∈ *majorities.opt-mru-step2 r dec-f* **using** *r R*
proof−
  define *sc-r-votes* **where** *sc-r-votes p* = (*if* (∃ *v*. *mru-vote* (*sc p*) = *Some* (*three-phase r*, *v*))
      then map-option snd (*mru-vote* (*sc p*))
      else None) **for** *p*
  have *sc-r-votes*: *sc-r-votes* = *majorities.r-votes sa r* **using** *R r*
    by(*auto simp add*: *ct-ref-rel-def sc-r-votes-def majorities.r-votes-def intro*!: *ext*)
  have *majorities.step2-d-guard dec-f sc-r-votes*
  proof(*clarsimp simp add*: *majorities.step2-d-guard-def*)
    fix *p v*
    assume *d-f-p*: *dec-f p* = *Some v*
    let *?Qv* = *votes-rcvd* (*μ p*)
    have *Qv*: *card ?Qv* > *N div 2*
      v = *the-rcvd-vote* (*μ p*) **using** *nxt*[*THEN spec*, *of p*] *d-f-p*
      by(*auto simp add*: *next2-def dec-f-def*)

hence *v* ∈ *snd* ' *votes-rcvd* (*μ p*)
by(*fastforce simp add*: *the-rcvd-vote-def ex-in-conv*[*symmetric*]
dest!: *card-gt-0-iff*[*THEN iffD1*, *OF le-less-trans*[*OF le0*]] *elim*!: *imageI*
intro: *someI*)
moreover have *?Qv* = *map-graph* (*sc-r-votes*) ∩ (*HOs r p* × *UNIV*) **using**
μ[*THEN spec*, *of p*]
by(*auto simp add*: *get-msgs-benign send2-def restrict-map-def votes-rcvd-def*
sc-r-votes-def image-def split: *option.split-asm*)
ultimately show *v* ∈ *ran sc-r-votes* ∧ *dom sc-r-votes* ∈ *majs* **using** *Qv*(*1*)
by(*auto simp add*: *majs-def inj-on-def map-graph-def fun-graph-def sc-r-votes-def*
the-rcvd-vote-def majs-def intro: *ranI*
elim!: *less-le-trans intro*!: *card-inj-on-le*[**where** *f*=*fst*])
qed

thus *?thesis* **using** *r R*
by(*auto simp add*: *majorities.opt-mru-step2-def sa′-def ct-ref-rel-def sc-r-votes*)
qed

ultimately show
∃ *sa′*. (∃ *r dec-f*. (*sa*, *sa′*) ∈ *majorities.opt-mru-step2 r dec-f*)
∧ (*sa′*, *Suc r*, *sc′*) ∈ *ct-ref-rel*
by *blast*

qed

lemma *CT-Refines-ThreeStep-MRU*:
PO-refines ct-ref-rel majorities.ts-mru-TS (*CT-TS HOs HOs crds*)
proof(*rule refine-basic*)
show *init* (*CT-TS HOs HOs crds*) ⊆ *ct-ref-rel* '' *init majorities.ts-mru-TS*
by(*auto simp add*: *CT-TS-defs majorities.ts-mru-TS-defs ct-ref-rel-def*)
next
show
{*ct-ref-rel*} *TS.trans majorities.ts-mru-TS*,
TS.trans (*CT-TS HOs HOs crds*) {> *ct-ref-rel*}
apply(*simp add*: *majorities.ts-mru-TS-defs CT-TS-defs*)
apply(*auto simp add*: *CHO-trans-alt CT-trans intro*!: *step0-ref step1-ref step2-ref*)
done
qed

end

19.2.2 Termination

theorem *CT-termination*:
  assumes *run*: *CHORun CT-Alg rho HOs crds*
    and *commR*: $\forall$ *r. CHOcommPerRd CT-M r* (*HOs r*) (*crds r*)
    and *commG*: *CHOcommGlobal CT-M HOs crds*
  shows $\exists$ *r v. decide* (*rho r p*) = *Some v*
proof −
  from *commG commR* **obtain** *ph c* **where**
   HOs:
    coord (*nr-steps*∗*ph*) = *c*
     ∧ (∀ *p. c* ∈ *HOs* (*nr-steps*∗*ph*+*1*) *p*)
     ∧ (∀ *p. card* (*HOs* (*nr-steps*∗*ph*+*2*) *p*) > *N div 2*)
  by(*auto simp add*: *CT-CHOMachine-def CT-commGlobal-def CT-commPerRd-def three-step-def*)

— The tedious bit: obtain three consecutive rounds linked by send/next functions
  define *r0* **where** *r0* = *nr-steps* ∗ *ph*
  define *cfg0* **where** *cfg0* = *rho r0*
  define *r1* **where** *r1* = *Suc r0*
  define *cfg1* **where** *cfg1* = *rho r1*
  define *r2* **where** *r2* = *Suc r1*
  define *cfg2* **where** *cfg2* = *rho r2*
  define *cfg3* **where** *cfg3* = *rho* (*Suc r2*)

  from
   run[*simplified CHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r0*]

   run[*simplified CHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r1*]
   run[*simplified CHORun-def*, *THEN CSHORun-step*, *THEN spec*, **where** *x*=*r2*]

   obtain *μ0 μ1 μ2* **where**
   send0: ∀ *p. μ0 p* ∈ *get-msgs* (*send0 r0*) *cfg0* (*HOs r0*) (*HOs r0*) *p*
   and *three-step0*: ∀ *p. next0 r0 p* (*cfg0 p*) (*μ0 p*) (*crds* (*Suc r0*) *p*) (*cfg1 p*)
   and *send1*: ∀ *p. μ1 p* ∈ *get-msgs* (*send1 r1*) *cfg1* (*HOs r1*) (*HOs r1*) *p*
   and *three-step1*: ∀ *p. next1 r1 p* (*cfg1 p*) (*μ1 p*) (*crds* (*Suc r1*) *p*) (*cfg2 p*)
   and *send2*: ∀ *p. μ2 p* ∈ *get-msgs* (*send2 r2*) *cfg2* (*HOs r2*) (*HOs r2*) *p*
   and *three-step2*: ∀ *p. next2 r2 p* (*cfg2 p*) (*μ2 p*) (*crds* (*Suc r2*) *p*) (*cfg3 p*)
  apply(*auto simp add*: *CT-Alg-def three-step-def CT-nextState-def CT-sendMsg-def all-conj-distrib*

r0-def r1-def r2-def
cfg0-def cfg1-def cfg2-def cfg3-def mod-Suc
)
done

— The proof: the coordinator hears enough messages in r0 and thus selects a value.

obtain *dec-v* **where** *dec-v*: *commt* (*cfg1 c*) = *dec-v*
by *simp*

have *step-r0*: *three-step r0* = *0*
by(*auto simp add*: *r0-def three-step-def*)
hence *same-coord*:
coord r1 = *coord r0*
coord r2 = *coord r0*
by(*auto simp add*: *three-step-phase-Suc r2-def r1-def r0-def intro*!: *coord-phase*)

— All processes hear from the coordinator, and thus set their vote to *dec-v*.
hence *all-vote*: $\forall$ *p*. *mru-vote* (*cfg2 p*) = *Some* (*three-phase r2*, *dec-v*)
using *HOs three-step1 send1 step-r0 dec-v*
by(*auto simp add*: *next1-def Let-def get-msgs-benign send1-def restrict-map-def isVote-def*
r2-def r1-def r0-def[*symmetric*] *same-coord*[*simplified r2-def r1-def*] *three-step-phase-Suc*)

— And finally, everybody will also decide *dec-v*.
have *all-decide*: $\forall$ *p*. *decide* (*cfg3 p*) = *Some dec-v*
proof
fix *p*
have *votes-rcvd* (*μ2 p*) = *HOs r2 p* × {*dec-v*} **using** *send2*[*THEN spec*, **where** *x*=*p*] *all-vote*
by(*auto simp add*: *send2-def get-msgs-benign votes-rcvd-def restrict-map-def image-def o-def*)

moreover from *HOs* **have** *N div 2* < *card* (*HOs r2 p*)
by(*auto simp add*: *r2-def r1-def r0-def*)

moreover then have *HOs r2 p* ≠ {}
by (*metis card.empty less-nat-zero-code*)
ultimately show *decide* (*cfg3 p*) = *Some dec-v*

using *three-step2*[*THEN spec*, **where** $x=p$] *send2*[*THEN spec*, **where** $x=p$] *all-vote*
by(*auto simp add*: *next2-def send2-def Let-def get-msgs-benign the-rcvd-vote-def restrict-map-def image-def o-def*)
qed

thus *?thesis*
by(*auto simp add*: *cfg3-def*)

qed

end

References

[1] M. Chaouch-Saad, B. Charron-Bost, and S. Merz. A reduction theorem for the verification of round-based distributed algorithms. In *Reachability Problems*, pages 93–106. 2009.

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

[3] H. Debrat and S. Merz. Verifying fault-tolerant distributed algorithms in the heard-of model. *Archive of Formal Proofs*, 2012.

[4] B. Lampson. The ABCD's of Paxos. In *PODC*, volume 1, page 13, 2001.

[5] O. Marić, C. Sprenger, and D. Basin. Consensus refined. In *Proc. of DSN*, 2015. to appear.