

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

December 14, 2021

Contents

1	Introduction	7
2	Preliminaries	8
2.1	Prover configuration	9
2.2	Forward reasoning ("attributes")	9
2.3	General results	9
2.3.1	Maps	9
2.3.2	Set	10
2.3.3	Relations	10
2.3.4	Lists	10
2.3.5	Finite sets	10
2.4	Consensus: types	11
2.5	Quorums	11
2.6	Miscellaneous lemmas	13
2.7	Argmax	14
2.8	Function and map graphs	17
2.9	Constant maps	17
2.10	Votes with maximum timestamps.	18
2.11	Step definitions for 2-step algorithms	21
2.12	Step definitions for 3-step algorithms	22

3	Models, Invariants and Refinements	23
3.1	Specifications, reachability, and behaviours.	23
3.1.1	Finite behaviours	23
3.1.2	Specifications, observability, and implementation	25
3.2	Invariants	29
3.2.1	Hoare triples	29
3.2.2	Characterization of reachability	30
3.2.3	Invariant proof rules	31
3.3	Refinement	32
3.3.1	Relational Hoare tuples	32
3.3.2	Refinement proof obligations	35
3.3.3	Deriving invariants from refinements	36
3.3.4	Transferring abstract invariants to concrete systems	37
3.3.5	Refinement of specifications	39
3.4	Transition system semantics for HO models	42
4	The Voting Model	45
4.1	Model definition	45
4.2	Invariants	47
4.2.1	Proofs of invariants	48
4.3	Agreement and stability	51
5	The Optimized Voting Model	55
5.1	Model definition	56
5.2	Refinement	57
5.2.1	Guard strengthening	57
5.2.2	Action refinement	59
5.2.3	The complete refinement proof	59
6	The OneThirdRule Algorithm	60
6.1	Model of the algorithm	61
6.2	Communication predicate for <i>One-Third Rule</i>	63
6.3	The <i>One-Third Rule</i> Heard-Of machine	63

6.4	Proofs	64
6.4.1	Refinement	67
6.4.2	Termination	71
7	The $A_{T,E}$ Algorithm	71
7.1	Model of the algorithm	72
7.2	Communication predicate for $A_{T,E}$	73
7.3	The $A_{T,E}$ Heard-Of machine	74
7.4	Proofs	74
7.4.1	Refinement	76
7.4.2	Termination	81
8	The Same Vote Model	81
8.1	Model definition	81
8.2	Refinement	82
8.3	Invariants	82
8.3.1	Proof of invariants	83
8.3.2	Transfer of abstract invariants	83
8.3.3	Additional invariants	84
9	The Observing Quorums Model	85
9.1	Model definition	85
9.2	Invariants	86
9.2.1	Proofs of invariants	87
9.3	Refinement	88
9.4	Additional invariants	88
9.4.1	Proofs of additional invariants	89
10	The Optimized Observing Quorums Model	91
10.1	Model definition	91
10.2	Refinement	92
11	Two-Step Observing Quorums Model	95

11.1	Model definition	95
11.2	Refinement	96
11.3	Invariants	99
11.3.1	Proofs of invariants	100
12	The UniformVoting Algorithm	100
12.1	Model of the algorithm	100
12.2	The <i>UniformVoting</i> Heard-Of machine	103
12.3	Proofs	104
12.3.1	Invariants	105
12.3.2	Refinement	106
12.3.3	Termination	115
13	The Ben-Or Algorithm	115
13.1	The <i>Ben-Or</i> Heard-Of machine	117
13.2	Proofs	118
13.2.1	Refinement	120
13.2.2	Termination	128
14	The MRU Vote Model	130
15	Optimized MRU Vote Model	132
15.1	Model definition	132
15.2	Refinement	133
15.2.1	The concrete guard implies the abstract guard	134
15.2.2	The concrete action refines the abstract action	136
15.2.3	The complete refinement	136
15.3	Invariants	137
16	Three-step Optimized MRU Model	138
16.1	Model definition	138
16.2	Refinement	140
17	The New Algorithm	144

17.1	Model of the algorithm	145
17.2	The Heard-Of machine	148
17.3	Proofs	149
17.3.1	Refinement	150
17.3.2	Termination	157
18	The Paxos Algorithm	160
18.1	Model of the algorithm	160
18.2	The <i>Paxos</i> Heard-Of machine	164
18.3	Proofs	165
18.3.1	Refinement	166
18.3.2	Termination	172
19	Chandra-Toueg $\diamond S$ Algorithm	174
19.1	The <i>CT</i> Heard-Of machine	178
19.2	Proofs	179
19.2.1	Refinement	180
19.2.2	Termination	186

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-

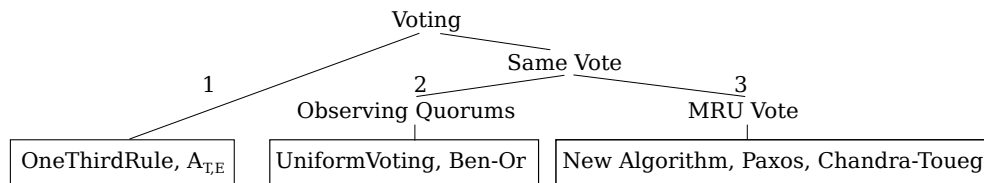


Figure 1: The consensus family tree. Boxes contain models of concrete algorithms.

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 claset 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*:

$f x = \text{Some } y \implies x \in \text{dom } f$

by (*simp add: domIff*)

lemma *map-definedness-contra*:

$\llbracket f x = \text{Some } y; z \notin \text{dom } f \rrbracket \implies x \neq z$

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*: $A \subseteq f^{-1}(f'A)$
by (*auto simp add: image-def vimage-def*)

2.3.3 Relations

lemma *Image-compose* [*simp*]:
 $(R1 \circ R2)^{-1}A = R2^{-1}(R1^{-1}A)$
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*: $map\ (g \circ f) = map\ g \circ map\ f$
by (*simp*)

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

lemma *take-prefix*: $\llbracket take\ n\ l = xs \rrbracket \implies \exists xs'. l = xs @ xs'$
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!*]:
 $\llbracket A \neq \{\}; finite\ A \rrbracket \implies 0 < card\ A$
by (*auto*)

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

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

lemma *finite-zero-cardD* [*dest!*]:
 $\llbracket \text{card } A = 0; \text{finite } A \rrbracket \implies A = \{\}$
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

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

typedecl *val* — Type of values to choose from

type-synonym *round* = *nat*

end

2.5 Quorums

locale *quorum* =

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

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

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

locale $quorum\text{-process} = quorum\ Quorum$
for $Quorum :: process \text{ set set}$

locale $mono\text{-quorum} = quorum\text{-process} +$
assumes $mono\text{-quorum}: \llbracket Q \in Quorum; Q \subseteq Q' \rrbracket \implies Q' \in Quorum$

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

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

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

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

qed

lemma *majorities-intersect*:

$\text{card } (Q :: \text{process set}) + \text{card } Q' > N \implies Q \cap Q' \neq \{\}$

by (*metis card-Un-disjoint card-mono finite not-le top-greatest*)

interpretation *majorities: mono-quorum majs*

proof

fix $Q Q'$ **assume** $Q \in \text{majs } Q' \in \text{majs}$

thus $Q \cap Q' \neq \{\}$

by (*intro majorities-intersect*) (*auto simp add: majs-def*)

next

show $\exists Q. Q \in \text{majs}$

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 $Q \in \text{majs } Q \subseteq Q'$

thus $Q' \in \text{majs}$ **using** *card-mono[OF - ⟨Q ⊆ 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

lemma *div-Suc*:

$\text{Suc } m \text{ div } n = (\text{if } \text{Suc } m \text{ mod } n = 0 \text{ then } \text{Suc } (m \text{ div } n) \text{ else } m \text{ div } n)$ (**is** - = *?rhs*)

proof (*cases Suc m mod n = 0*)

case *True*

then show *?thesis*

by *simp (metis Zero-not-Suc diff-Suc-Suc div-geq minus-mod-eq-mult-div mod-Suc*

mod-less neq0-conv nonzero-mult-div-cancel-left
next
 case *False*
 then show *?thesis*
 by *simp (metis diff-Suc-Suc div-eq-0-iff minus-mod-eq-mult-div mod-Suc neq0-conv nonzero-mult-div-cancel-left)*
qed

definition *flip* **where**

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

lemma *option-expand'*:

$\llbracket (option = None) = (option' = None); \bigwedge x\ y. \llbracket option = Some\ x; option' = Some\ y \rrbracket \implies x = y \rrbracket \implies$
 $option = option'$
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* ($f \text{ ' } A$) **by** *simp*
from *assms*(2) **have** *non-empty*: $A \neq \{\}$ **by** *auto*
have $f x \in f \text{ ' } A$ **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 ($\bigcup S$) $\implies \forall A \in S. \textit{finite } A$
by (*metis Union-upper finite-subset*)

lemma *Max-by-eqI*:
assumes
fn: *finite* A
and $\bigwedge y. y \in A \implies \textit{cmp-f } y \leq \textit{cmp-f } x$
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) $\leq \textit{cmp-f } x$ **using** *assms*
by *auto*
also have *cmp-f* $x \leq \textit{cmp-f}$ (*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*:
 $\llbracket \textit{finite } A; A = \bigcup S; S \neq \{\}; \{\} \notin S; \textit{inj-on } \textit{cmp-f } A \rrbracket \implies$
 $\textit{Max-by } \textit{cmp-f } A = \textit{Max-by } \textit{cmp-f} (\textit{Max-by } \textit{cmp-f} \text{ ' } S)$
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* $y \leq \textit{cmp-f}$ (*Max-by* *cmp-f* B) **using** *assms*
by (*metis Max-by-ge finite-UN-D*)
also have $\dots \leq \textit{cmp-f}$ (*Max-by* *cmp-f* (*Max-by* *cmp-f* \text{ ' } S)) **using** *B-def*(1)

assms

by (*metis Max-by-ge finite-UnionD finite-imageI imageI*)
finally show $\text{cmp-f } y \leq \text{cmp-f } (\text{Max-by cmp-f } (\text{Max-by cmp-f } 'S))$.
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: $\forall v0 v1. (\neg \text{finite } v0 \vee v0 = \{\}) \vee \text{Max-by } (v1::'a \Rightarrow 'b) v0 \in v0$*
using *Max-by-in by blast*
have $\forall v1. v1 \notin S \vee \text{finite } v1$ **using** *assms(1) assms(2) finite-UN-D by blast*
then obtain *v2-13 :: 'a set set \Rightarrow 'a \Rightarrow 'a set where Max-by cmp-f ' S $\subseteq \bigcup S$*
using *f1 assms(3) by blast*
thus *Max-by cmp-f ' S \subseteq A using assms(2) by presburger*
qed
finally show *Max-by cmp-f (Max-by cmp-f ' S) \in A* .
qed

lemma *Max-by-UNION-distrib:*

$\llbracket \text{finite } A; A = (\bigcup_{x \in S} f x); S \neq \{\}; \{\} \notin f ' S; \text{inj-on cmp-f } A \rrbracket \Longrightarrow$
 $\text{Max-by cmp-f } A = \text{Max-by cmp-f } (\text{Max-by cmp-f } ' (f ' S))$
by(*force intro!: Max-by-Union-distrib*)

lemma *Max-by-eta:*

$\text{Max-by } f = (\lambda S. (\text{SOME } x. x \in S \wedge f x = \text{Max } (f ' S)))$
by(*auto simp add: Max-by-def*)

lemma *Max-is-Max-by-id:*

$\llbracket \text{finite } S; S \neq \{\} \rrbracket \Longrightarrow \text{Max } S = \text{Max-by id } S$
apply(*clarsimp simp add: Max-by-def*)
by (*metis (mono-tags, lifting) Max-in someI-ex*)

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

2.8 Function and map graphs

definition *fun-graph* **where**

$$\text{fun-graph } f = \{(x, f\ x) \mid x. \text{ True}\}$$

definition *map-graph* $:: ('a, 'b)\text{map} \Rightarrow ('a \times 'b)\text{ set}$ **where**

$$\text{map-graph } f = \{(x, y) \mid x\ y. (x, \text{Some } y) \in \text{fun-graph } f\}$$

lemma *map-graph-mem*[*simp*]:

$$((x, y) \in \text{map-graph } f) = (f\ x = \text{Some } y)$$

by(*auto simp add: dom-def map-graph-def fun-graph-def*)

lemma *finite-fun-graph*:

$$\text{finite } A \Longrightarrow \text{finite } (\text{fun-graph } f \cap (A \times \text{UNIV}))$$

apply(*rule bij-betw-finite*[**where** $A=A$ **and** $f=\lambda x. (x, f\ x)$, *THEN iffD1*])

by(*auto simp add: fun-graph-def bij-betw-def inj-on-def*)

lemma *finite-map-graph*:

$$\text{finite } A \Longrightarrow \text{finite } (\text{map-graph } f \cap (A \times \text{UNIV}))$$

by(*force simp add: map-graph-def*

dest!: *finite-fun-graph*[**where** $f=f$]

intro!: *finite-surj*[**where** $A=\text{fun-graph } f \cap (A \times \text{UNIV})$ **and** $f=\text{apsnd the}$

)

lemma *finite-dom-finite-map-graph*:

$$\text{finite } (\text{dom } f) \Longrightarrow \text{finite } (\text{map-graph } f)$$

apply(*simp add: dom-def map-graph-def fun-graph-def*)

apply(*erule finite-surj*[**where** $f=\lambda x. (x, \text{the } (f\ x))$])

apply(*clarsimp simp add: image-def*)

by (*metis option.sel*)

lemma *ran-map-addD*:

$$x \in \text{ran } (m ++ f) \Longrightarrow x \in \text{ran } m \vee x \in \text{ran } f$$

by(*auto simp add: ran-def*)

2.9 Constant maps

definition *const-map* $:: 'v \Rightarrow 'k\ \text{set} \Rightarrow ('k, 'v)\text{map}$ **where**

$$\text{const-map } v\ S \equiv (\lambda-. \text{Some } v) \mid 'S$$

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

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

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

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

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

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

lemma *dom-const-map*:
 $dom\ (const\text{-}map\ v\ S) = S$
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**
 $vote\text{-}set\ vs\ Q \equiv \{(r, v) \mid a\ r\ v.\ ((r, a), v) \in map\text{-}graph\ (case\text{-}prod\ vs) \wedge a \in Q\}$

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

lemma *finite-vote-set*:
assumes $\forall r' \geq (r :: nat).\ v\text{-}hist\ r' = Map.empty$
 $finite\ S$

shows *finite (vote-set v-hist S)*
proof –
define *vs* **where** $vs = \{((r, a), v) \mid r \ a \ v. ((r, a), v) \in \text{map-graph } (\text{case-prod } v\text{-hist}) \wedge a \in S\}$
have *vs*
 $= (\bigcup_{p \in S}. ((\lambda r. v). ((r, p), v)) \text{ ‘ } (\text{map-graph } (\lambda r. v\text{-hist } r \ p))))$
by(*auto simp add: map-graph-def fun-graph-def vs-def*)
also have $\dots \leq (\bigcup_{p \in S}. (\lambda r. ((r, p), \text{the } (v\text{-hist } r \ p))) \text{ ‘ } \{0..<r\})$
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 $I = \text{finite-subset}[OF \text{ calculation}]$
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 = $\lambda((r, a), v). (r, v)$]*)
by(*force simp add: image-def*)
qed

definition *mru-of-set*
 $:: ('round :: \text{linorder} \Rightarrow ('process, 'val)\text{map}) \Rightarrow ('process \text{ set}, 'round \times 'val)\text{map}$
where
 $\text{mru-of-set } vs \equiv \lambda Q. \text{option-Max-by fst } (\text{vote-set } vs \ Q)$

definition *process-mru*
 $:: ('round :: \text{linorder} \Rightarrow ('process, 'val)\text{map}) \Rightarrow ('process, 'round \times 'val)\text{map}$
where
 $\text{process-mru } vs \equiv \lambda a. \text{mru-of-set } vs \ \{a\}$

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

lemma *process-mru-is-Some:*
 $(\text{process-mru } v\text{-f } a = \text{Some } rv) = (\text{vote-set } v\text{-f } \{a\} \neq \{\} \wedge rv = \text{Max-by fst } (\text{vote-set } v\text{-f } \{a\}))$

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}) ⇒
  finite (vote-set (v-hist(r := v-f)) {a})
by(simp add: vote-set-upd)
```

lemma *vote-setD*:

```
rv ∈ vote-set v-f {a} ⇒ v-f (fst rv) a = Some (snd rv)
by(auto simp add: vote-set-def)
```

lemma *process-mru-new-votes*:

assumes

$\forall r' \geq (r :: \text{nat}). v\text{-hist } r' = \text{Map.empty}$

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*:

$process\text{-}mru\ (v\text{-}hist(r := v\text{-}f))\ p = Some\ rv$
 $(process\text{-}mru\ v\text{-}hist\ ++\ (\lambda p. map\text{-}option\ (Pair\ r)\ (v\text{-}f\ p)))\ p = Some\ rv'$
moreover have $v\text{-}hist\ (r)\ p = None$ **using** $assms(1)$
by($auto$)

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

qed

end

2.11 Step definitions for 2-step algorithms

definition $two\text{-}phase$ **where** $two\text{-}phase\ (r::nat) \equiv r\ div\ 2$

definition $two\text{-}step$ **where** $two\text{-}step\ (r::nat) \equiv r\ mod\ 2$

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

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

lemma $two\text{-}phase\text{-}step$: $(two\text{-}phase\ r * 2) + two\text{-}step\ r = r$
by ($auto\ simp\ add: two\text{-}phase\text{-}def\ two\text{-}step\text{-}def$)

lemma $two\text{-}step\text{-}phase\text{-}Suc$:

$two\text{-}step\ r = 0 \implies two\text{-}phase\ (Suc\ r) = two\text{-}phase\ r$

$two\text{-}step\ r = 0 \implies two\text{-}step\ (Suc\ r) = 1$

$two\text{-}step\ r = 0 \implies two\text{-}phase\ (Suc\ (Suc\ r)) = Suc\ (two\text{-}phase\ r)$

$two\text{-}step\ r = (Suc\ 0) \implies two\text{-}phase\ (Suc\ r) = Suc\ (two\text{-}phase\ r)$

$two\text{-}step\ r = (Suc\ 0) \implies two\text{-}step\ (Suc\ r) = 0$

by($simp\text{-}all\ add: two\text{-}step\text{-}def\ two\text{-}phase\text{-}def\ mod\text{-}Suc\ div\text{-}Suc$)

end

2.12 Step definitions for 3-step algorithms

abbreviation $(input) nr\text{-steps} \equiv 3$

definition $three\text{-phase}$ **where** $three\text{-phase} (r::nat) \equiv r \text{ div } nr\text{-steps}$

definition $three\text{-step}$ **where** $three\text{-step} (r::nat) \equiv r \text{ mod } nr\text{-steps}$

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

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

lemma $three\text{-phase-step}$: $(three\text{-phase} r * nr\text{-steps}) + three\text{-step} r = r$
by $(auto \text{ simp add: } three\text{-phase-def } three\text{-step-def})$

lemma $three\text{-step-Suc}$:

$three\text{-step} r = 0 \implies three\text{-step} (Suc (Suc r)) = 2$
 $three\text{-step} r = 0 \implies three\text{-step} (Suc r) = 1$
 $three\text{-step} r = (Suc 0) \implies three\text{-step} (Suc r) = 2$
 $three\text{-step} r = (Suc 0) \implies three\text{-step} (Suc (Suc r)) = 0$
 $three\text{-step} r = (Suc (Suc 0)) \implies three\text{-step} ((Suc r)) = 0$
by $(unfold \text{ three-step-def, simp-all add: mod-Suc})$

lemma $three\text{-step-phase-Suc}$:

$three\text{-step} r = 0 \implies three\text{-phase} (Suc r) = three\text{-phase} r$
 $three\text{-step} r = 0 \implies three\text{-phase} (Suc (Suc r)) = three\text{-phase} r$
 $three\text{-step} r = 0 \implies three\text{-phase} (Suc (Suc (Suc r))) = Suc (three\text{-phase} r)$
 $three\text{-step} r = (Suc 0) \implies three\text{-phase} (Suc r) = three\text{-phase} r$
 $three\text{-step} r = (Suc 0) \implies three\text{-phase} (Suc (Suc r)) = Suc (three\text{-phase} r)$
 $three\text{-step} r = (Suc (Suc 0)) \implies three\text{-phase} (Suc r) = Suc (three\text{-phase} r)$
by $(simp\text{-all add: } three\text{-step-def } three\text{-phase-def mod-Suc div-Suc})$

lemma $three\text{-step2-phase-Suc}$:

$three\text{-step} r = 2 \implies (3 * (Suc (three\text{-phase} r)) - 1) = r$
apply $(simp \text{ add: } three\text{-step-def } three\text{-phase-def})$
by $(metis \text{ add-2-eq-Suc' mult-div-mod-eq})$

lemma $three\text{-stepE}$:

```

[[ three-step r = 0 ==> P; three-step r = 1 ==> P; three-step r = 2 ==> P ]] ==>
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*: $s \# b \in \text{beh } T$

Behaviours are prefix closed.

lemma *beh-immediate-prefix-closed*:

$s \# b \in \text{beh } T \implies b \in \text{beh } T$
by (*erule beh-non-empty, auto*)

lemma *beh-prefix-closed*:

$c @ b \in \text{beh } T \implies b \in \text{beh } T$
by (*induct c, auto dest!: beh-immediate-prefix-closed*)

States in behaviours are exactly reachable.

lemma *beh-in-reach* [*rule-format*]:

$b \in \text{beh } T \implies (\forall s \in \text{set } b. s \in \text{reach } T)$
by (*erule beh.induct*) (*auto*)

lemma *reach-in-beh*:

$s \in \text{reach } T \implies \exists b \in \text{beh } T. s \in \text{set } b$
proof (*induction rule: reach.induct*)
case (*r-init s*) **thus** *?case* **by** (*auto intro: behI [where x=[s]]*)
next
case (*r-trans s t*) **thus** *?case*
proof –
from *r-trans(?)* **obtain** $b \ b0 \ b1$ **where** $b \in \text{beh } T \ b = b1 @ s \# b0$ **by** (*auto dest: split-list*)
hence $s \# b0 \in \text{beh } T$ **by** (*auto intro: beh-prefix-closed*)
hence $t \# s \# b0 \in \text{beh } T$ **using** $\langle (s, t) \in \text{trans } T \rangle$ **by** *auto*
thus *?thesis* **by** – (*rule behI, auto*)
qed
qed

lemma *reach-equiv-beh-states*: $\text{reach } T = (\bigcup b \in \text{beh } T. \text{set } b)$

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 $b \in \text{beh } TS$
and $\text{Suc } i < \text{length } b$
and $s = b ! \text{Suc } i$
and $t = b ! i$

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

$observable :: ['s \Rightarrow 'o, 's\ set] \Rightarrow bool$

where

$observable\ ob\ P \equiv \forall s\ s'.\ ob\ s = ob\ s' \longrightarrow s' \in P \longrightarrow s \in P$

definition

$observable2 :: ['s \Rightarrow 'o, 's\ set] \Rightarrow bool$

where

$observable2\ ob\ P \equiv \exists Q.\ P = ob-'Q$

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*]:

$\llbracket observable\ ob\ P; ob\ s = ob\ s'; s' \in P \rrbracket \Longrightarrow s \in P$

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*: $\llbracket Q \subseteq \text{ob}'P; \text{observable } \text{ob } P \rrbracket \implies \text{ob}'Q \subseteq P$
by (*auto*)

lemma *vimage-l-image-r*: $\llbracket \text{ob}'Q \subseteq P; Q \subseteq \text{range } \text{ob} \rrbracket \implies Q \subseteq \text{ob}'P$
by (*drule image-mono [where f=ob]*, *auto*)

Internal and external invariants

lemma *external-from-internal-invariant*:
 $\llbracket \text{reach } S \subseteq P; (\text{obs } S)'P \subseteq Q \rrbracket$
 $\implies \text{oreach } S \subseteq Q$
by (*auto simp add: oreach-def*)

lemma *external-from-internal-invariant-vimage*:
 $\llbracket \text{reach } S \subseteq P; P \subseteq (\text{obs } S)'Q \rrbracket$
 $\implies \text{oreach } S \subseteq Q$
by (*erule external-from-internal-invariant*) (*auto*)

lemma *external-to-internal-invariant-vimage*:
 $\llbracket \text{oreach } S \subseteq Q; (\text{obs } S)'Q \subseteq P \rrbracket$
 $\implies \text{reach } S \subseteq P$
by (*auto simp add: oreach-def*)

lemma *external-to-internal-invariant*:
 $\llbracket \text{oreach } S \subseteq Q; Q \subseteq (\text{obs } S)'P; \text{observable } (\text{obs } S) P \rrbracket$
 $\implies \text{reach } S \subseteq P$
by (*erule external-to-internal-invariant-vimage*) (*auto*)

lemma *external-equiv-internal-invariant-vimage*:
 $\llbracket P = (\text{obs } S)'Q \rrbracket$
 $\implies (\text{oreach } S \subseteq Q) = (\text{reach } S \subseteq P)$
by (*fast intro: external-from-internal-invariant-vimage*
external-to-internal-invariant-vimage
del: subsetI)

lemma *external-equiv-internal-invariant*:
 $\llbracket (\text{obs } S) 'P = Q; \text{observable } (\text{obs } S) P \rrbracket$
 $\implies (\text{oreach } S \subseteq Q) = (\text{reach } S \subseteq P)$
by (*rule external-equiv-internal-invariant-vimage*) (*auto*)

Our notion of implementation is inclusion of observable behaviours.

definition

implements :: [*p* \implies *'o*, (*'s*, *'o*) *spec*, (*'t*, *'p*) *spec*] \implies *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*:
 $\llbracket \text{implements } pi1 S1 S2; \text{implements } pi2 S2 S3 \rrbracket$
 $\implies \text{implements } (pi1 \circ pi2) S1 S3$
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 $\implies pi$ '(*oreach Sc*) \subseteq *oreach Sa*
by (*auto simp add: implements-def oreach-equiv-obeh-states dest!: subsetD*)

lemma *external-invariant-preservation*:
 $\llbracket \text{oreach } Sa \subseteq Q; \text{implements } pi Sa Sc \rrbracket$
 $\implies pi$ '(*oreach Sc*) $\subseteq Q$
by (*rule subset-trans [OF implements-oreach]*) (*auto*)

lemma *external-invariant-translation*:
 $\llbracket \text{oreach } Sa \subseteq Q; pi - 'Q \subseteq P; \text{implements } pi Sa Sc \rrbracket$
 $\implies \text{oreach } Sc \subseteq P$
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*:

$\llbracket \text{reach } Sa \subseteq Pa; Pa \subseteq \text{obs } Sa - ' Qa; pi - ' Qa \subseteq Q; \text{obs } S - ' Q \subseteq P; \\ \text{implements } pi Sa S \rrbracket \\ \implies \text{reach } S \subseteq P$

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-hoare :: [*'s set, ('s × 's) set, 's set*] \Rightarrow *bool*
 $((\exists \{-\} - \{> -\}) [0, 0, 0] 90)$

where

$\{pre\} R \{> post\} \equiv R \text{'pre} \subseteq post$

lemmas *PO-hoare-defs = PO-hoare-def Image-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*:

$\llbracket \{I\} R \{> J\}; s \in I; (s, s') \in R \rrbracket \implies s' \in J$
by(*auto simp add: PO-hoare-def*)

Some essential facts about Hoare triples.

lemma *hoare-conseq-left* [*intro*]:

$\llbracket \{P'\} R \{> Q\}; P \subseteq P' \rrbracket \\ \implies \{P\} R \{> Q\}$

by (*auto simp add: PO-hoare-defs*)

lemma *hoare-conseq-right*:

$\llbracket \{P\} R \{> Q'\}; Q' \subseteq Q \rrbracket \\ \implies \{P\} R \{> Q\}$

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!*]:

$$\begin{aligned} & \llbracket \{P\} R \{> Q1\}; \{P\} R \{> Q2\} \rrbracket \\ & \implies \{P\} R \{> Q1 \cap Q2\} \end{aligned}$$

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!*]:

$$P \subseteq Q \implies \{P\} Id \{> Q\}$$

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*: $reach\ T \subseteq I \implies init\ T \subseteq I$

by (*auto dest: subsetD*)

lemma *reach-trans*: $reach\ T \subseteq I \implies \{reach\ T\} trans\ T \{> I\}$

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:*

$\llbracket init\ T \subseteq P; \{P\}\ (trans\ T)\ \{>\ P\} \rrbracket$
 $\implies reach\ T \subseteq P$

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:*

$\llbracket init\ T \subseteq I; I \subseteq P; \{I\}\ (trans\ T)\ \{>\ I\} \rrbracket$
 $\implies reach\ T \subseteq P$

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:*

$\llbracket init\ T \subseteq I; \{I \cap reach\ T\}\ (trans\ T)\ \{>\ I\} \rrbracket$
 $\implies reach\ T \subseteq I$

by (*safe, erule reach.induct, auto simp add: PO-hoare-defs*)

Proof of equivalence.

lemma *inv-rule-from-INV-rule:*

$\llbracket init\ T \subseteq I; I \subseteq P; \{I\}\ (trans\ T)\ \{>\ I\} \rrbracket$
 $\implies reach\ T \subseteq P$

apply (*rule subset-trans, auto del: subsetI*)

apply (*rule INV-rule, auto*)

done

lemma *INV-rule-from-inv-rule:*

$\llbracket init\ T \subseteq I; \{I \cap reach\ T\}\ (trans\ T)\ \{>\ I\} \rrbracket$
 $\implies reach\ T \subseteq I$

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*:

$$\llbracket \text{init } T \subseteq I; \{I \cap J\} (\text{trans } T) \{> I\}; \text{reach } T \subseteq J \rrbracket \\ \implies \text{reach } T \subseteq I$$

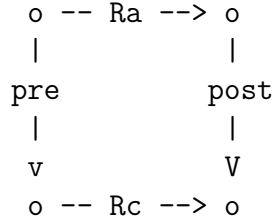
by (*rule INV-rule, auto*)

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:



Here, *Ra* and *Rc* are the abstract and concrete transition relations, and *pre* and *post* are the pre- and post-relations. (In the definition below, the operator (*O*) stands for relational composition, which is defined as follows: (*O*) $\equiv \lambda r s. \{(xa, x). ((\lambda x xa. (x, xa) \in r) O (\lambda x xa. (x, xa) \in s)) xa x\}$.)

definition

$$\text{PO-rhoare} :: \\ [(\text{'s} \times \text{'t}) \text{ set}, (\text{'s} \times \text{'s}) \text{ set}, (\text{'t} \times \text{'t}) \text{ set}, (\text{'s} \times \text{'t}) \text{ set}] \Rightarrow \text{bool} \\ ((\lambda \{-\} -, - \{> -\}) [0, 0, 0] 90)$$

where

$$\{\text{pre}\} Ra, Rc \{> \text{post}\} \equiv \text{pre } O Rc \subseteq Ra O \text{post}$$

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

Facts about relational Hoare tuples.

lemma *relhoare-conseq-left* [*intro*]:
 $\llbracket \{pre'\} Ra, Rc \{> post'\}; pre \subseteq pre' \rrbracket$
 $\implies \{pre\} Ra, Rc \{> post\}$
by (*auto simp add: PO-rhoare-defs dest!: subsetD*)

lemma *relhoare-conseq-right*: — do NOT declare [*intro*]
 $\llbracket \{pre\} Ra, Rc \{> post'\}; post' \subseteq post \rrbracket$
 $\implies \{pre\} Ra, Rc \{> post\}$
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*]:
 $Domain \ pre \subseteq R \implies Domain (pre \ O \ Rc) \subseteq R$
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*:
 $\llbracket \{R1\} T1, T2 \{> R1\}; \{R2\} T2, T3 \{> R2\} \rrbracket$
 $\implies \{R1 \ O \ R2\} T1, T3 \{> R1 \ O \ R2\}$
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 transition relation is deterministic and in the second case one conjunct is a cartesian product of two state predicates.

lemma *relhoare-conj-right-det*:

$$\begin{aligned} & \llbracket \{pre\} Ra, Rc \{> post1\}; \{pre\} Ra, Rc \{> post2\}; \\ & \quad \text{single-valued } Ra \rrbracket \quad \text{--- only for deterministic } Ra! \\ \implies & \{pre\} Ra, Rc \{> post1 \cap post2\} \end{aligned}$$

by (*auto simp add: PO-rhoare-defs dest: single-valuedD dest!: subsetD*)

lemma *relhoare-conj-right-cartesian* [*intro*]:

$$\begin{aligned} & \llbracket \{Domain\ pre\} Ra \{> I\}; \{Range\ pre\} Rc \{> J\}; \\ & \quad \{pre\} Ra, Rc \{> post\} \rrbracket \\ \implies & \{pre\} Ra, Rc \{> post \cap I \times J\} \end{aligned}$$

by (*force simp add: PO-rhoare-defs PO-hoare-defs Domain-def Range-def*)

Separate rule for cartesian products.

corollary *relhoare-cartesian*:

$$\begin{aligned} & \llbracket \{Domain\ pre\} Ra \{> I\}; \{Range\ pre\} Rc \{> J\}; \\ & \quad \{pre\} Ra, Rc \{> post\} \rrbracket \quad \text{--- any } post, \text{ including } UNIV! \\ \implies & \{pre\} Ra, Rc \{> I \times J\} \end{aligned}$$

by (*auto intro: relhoare-conseq-right*)

Unions of transition relations.

lemma *relhoare-concrete-Un* [*simp*]:

$$\begin{aligned} & \{pre\} Ra, Rc1 \cup Rc2 \{> post\} \\ & = (\{pre\} Ra, Rc1 \{> post\} \wedge \{pre\} Ra, Rc2 \{> post\}) \end{aligned}$$

apply (*auto simp add: PO-rhoare-defs*)

apply (*auto dest!: subsetD*)

done

lemma *relhoare-concrete-UN* [*simp*]:

$$\{pre\} Ra, \bigcup x. Rc\ x \{> post\} = (\forall x. \{pre\} Ra, Rc\ x \{> post\})$$

apply (*auto simp add: PO-rhoare-defs*)

apply (*auto dest!: subsetD*)

done

lemma *relhoare-abstract-Un-left* [*intro*]:

$$\begin{aligned} & \llbracket \{pre\} Ra1, Rc \{> post\} \rrbracket \\ \implies & \{pre\} Ra1 \cup Ra2, Rc \{> post\} \end{aligned}$$

by (*auto simp add: PO-rhoare-defs*)

lemma *relhoare-abstract-Un-right* [*intro*]:
 $\llbracket \{pre\} Ra2, Rc \{> post\} \rrbracket$
 $\implies \{pre\} Ra1 \cup Ra2, Rc \{> post\}$
by (*auto simp add: PO-rhoare-defs*)

lemma *relhoare-abstract-UN* [*intro!*]: — might be too aggressive?
 $\llbracket \{pre\} Ra x, Rc \{> post\} \rrbracket$
 $\implies \{pre\} \bigcup x. Ra x, Rc \{> post\}$
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 ::
 $[('s \times 't) \text{ set}, ('s, 'a) \text{ TS-scheme}, ('t, 'b) \text{ TS-scheme}] \Rightarrow \text{bool}$
where
 $PO\text{-refines } R \text{ } Ta \text{ } Tc \equiv ($
 $\quad \text{init } Tc \subseteq R^{\text{``}}(\text{init } Ta)$
 $\quad \wedge \{R\} (\text{trans } Ta), (\text{trans } Tc) \{> R\}$
 $\quad)$

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

lemma *refine-basic*:

$\llbracket \text{init } Tc \subseteq R^{\text{``}}(\text{init } Ta); \{R\} (\text{trans } Ta), (\text{trans } Tc) \{> R\} \rrbracket$
 $\implies PO\text{-refines } R \text{ } Ta \text{ } Tc$
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 $\text{Domain } (R \cap UNIV \times J)$ 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*:

$$\begin{aligned} & \llbracket \text{init } Tc \subseteq R \text{''}(\text{init } Ta); \text{init } Ta \subseteq I; \text{init } Tc \subseteq J \rrbracket \\ & \implies \text{init } Tc \subseteq (R \cap I \times J) \text{''}(\text{init } Ta) \end{aligned}$$

by (*auto simp add: Image-def dest!: bspec subsetD*)

lemma *refine-trans-using-invariants*:

$$\begin{aligned} & \llbracket \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R\}; \\ & \quad \{I \cap \text{Domain } (R \cap UNIV \times J)\} (\text{trans } Ta) \{> I\}; \\ & \quad \{J \cap \text{Range } (R \cap I \times UNIV)\} (\text{trans } Tc) \{> J\} \rrbracket \\ & \implies \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R \cap I \times J\} \end{aligned}$$

by (*rule relhoare-conj-right-cartesian*) (*auto*)

This is our main rule for refinements.

lemma *refine-using-invariants*:

$$\begin{aligned} & \llbracket \{R \cap I \times J\} (\text{trans } Ta), (\text{trans } Tc) \{> R\}; \\ & \quad \{I \cap \text{Domain } (R \cap UNIV \times J)\} (\text{trans } Ta) \{> I\}; \\ & \quad \{J \cap \text{Range } (R \cap I \times UNIV)\} (\text{trans } Tc) \{> J\}; \\ & \quad \text{init } Tc \subseteq R \text{''}(\text{init } Ta); \\ & \quad \text{init } Ta \subseteq I; \text{init } Tc \subseteq J \rrbracket \\ & \implies PO\text{-refines } (R \cap I \times J) \text{ } Ta \text{ } Tc \end{aligned}$$

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*:

$$PO\text{-refines } R \text{ } Ta \text{ } Tc \implies \text{init } Tc \subseteq \text{Range } R$$

by (*auto simp add: PO-refines-def*)

lemma *PO-refines-implies-Range-trans*:

$$PO\text{-refines } R \text{ } Ta \text{ } Tc \implies \{\text{Range } R\} \text{trans } Tc \{> \text{Range } R\}$$

by (*auto simp add: PO-refines-def PO-rhoare-def PO-hoare-def*)

lemma *PO-refines-implies-Range-invariant*:
 $PO\text{-refines } R \text{ } Ta \text{ } Tc \implies reach \ Tc \subseteq Range \ R$
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*:
 $\llbracket PO\text{-refines } R \text{ } Ta \text{ } Tc; Range \ R \subseteq I \rrbracket$
 $\implies init \ Tc \subseteq I$
by (drule *PO-refines-implies-Range-init*, *auto*)

corollary *INV-trans-from-refinement*:
 $\llbracket PO\text{-refines } R \text{ } Ta \text{ } Tc; K \subseteq Range \ R; Range \ R \subseteq I \rrbracket$
 $\implies \{K\} \ trans \ Tc \ \{> \ I\}$
apply (drule *PO-refines-implies-Range-trans*)
apply (auto intro: *hoare-conseq-right*)
done

corollary *INV-from-refinement*:
 $\llbracket PO\text{-refines } R \text{ } Ta \text{ } Tc; Range \ R \subseteq I \rrbracket$
 $\implies reach \ Tc \subseteq I$
by (drule *PO-refines-implies-Range-invariant*, *fast*)

3.3.4 Transferring 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*:
 $PO\text{-refines } R \text{ } Ta \text{ } Tc \implies init \ Tc \subseteq R \ \text{“} (init \ Ta)$
apply(rule *subset-trans*[**where** $B=R \ \text{“} \ init \ Ta$])
apply (auto simp add: *PO-refines-def*)
done

lemma *commute-dest*:
 $\llbracket R \ O \ Tc \subseteq Ta \ O \ R; (sa, sc) \in R; (sc, sc') \in Tc \rrbracket \implies \exists sa'. (sa, sa') \in Ta \wedge$
 $(sa', sc') \in R$
by(*auto*)

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

lemma *PO-refines-implies-R-image-invariant*:
assumes *PO-refines R Ta Tc*
shows $\text{reach } Tc \subseteq R \text{ “ reach } Ta$
proof(*rule INV-rule*)
show $\text{init } Tc \subseteq R \text{ “ reach } Ta$
by (*rule subset-trans[OF PO-refines-implies-R-image-init, OF assms]*) (*auto*)
next
show $\{R \text{ “ reach } Ta \cap \text{reach } Tc\} \text{ TS.trans } Tc \{> R \text{ “ reach } Ta\}$ **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 $\text{init } Tc \subseteq R \text{ “ } I$ **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 $\{R \text{ “ } I\} \text{ trans } Tc \{> R \text{ “ } J\}$
proof(*unfold PO-hoare-def Image-def, safe*)
fix *sc sc' sa*

assume *step*: $(sc, sc') \in \text{trans } Tc$ **and** *abs-inv*: $sa \in I$ **and** $R: (sa, sc) \in R$
from *ref step* **and** R **obtain** sa' **where**
abs-step: $(sa, sa') \in \text{trans } Ta$ **and** $R': (sa', sc') \in R$
by(*auto simp add: PO-refines-def PO-rhoare-def*)
with *hoareD*[*OF abs-hoare abs-inv abs-step*]
show $\exists sa' \in J. (sa', sc') \in R$
by(*blast*)
qed

lemma *abs-INV-transfer*:

assumes
PO-refines $R Ta Tc$
reach $Ta \subseteq I$
shows *reach* $Tc \subseteq R \text{ `` } I$ **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 $:: ('s \times 't) \text{ set} \Rightarrow ('s \text{ list} \times 't \text{ list}) \text{ set}$
for $R :: ('s \times 't) \text{ set}$

where

sl-nil [*iff*]: $([], []) \in \text{seq-lift } R$
| *sl-cons* [*intro*]:
 $\llbracket (xs, ys) \in \text{seq-lift } R; (x, y) \in R \rrbracket \Longrightarrow (x\#xs, y\#ys) \in \text{seq-lift } R$

inductive-cases *sl-cons-right-invert*: $(ba', t \# bc) \in \text{seq-lift } R$

For each concrete behaviour there is a related abstract one.

lemma *behaviour-refinement*:

assumes *PO-refines* $R Ta Tc$ $bc \in \text{beh } Tc$
shows $\exists ba \in \text{beh } Ta. (ba, bc) \in \text{seq-lift } R$
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 × t) set, 'p ⇒ 'o, ('s, 'o) spec, ('t, 'p) spec] ⇒ bool
where
  obs-consistent R pi Sa Sc ≡ (∀ s t. (s, t) ∈ R ⟶ 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]
 ⟹ *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 ⟹ *obs-consistent* (R ∩ R') pi Sa Sc
by (*auto simp add: obs-consistent-def*)

lemma *obs-consistent-conj2* [*intro*]:

$obs-consistent\ R\ pi\ Sa\ Sc \implies obs-consistent\ (R' \cap R)\ pi\ Sa\ Sc$
by (*auto simp add: obs-consistent-def*)

lemma *obs-consistent-behaviours:*

$\llbracket obs-consistent\ R\ pi\ Sa\ Sc; bc \in beh\ Sc; ba \in beh\ Sa; (ba, bc) \in seq-lift\ R \rrbracket$
 $\implies map\ pi\ (map\ (obs\ Sc)\ bc) = map\ (obs\ Sa)\ ba$
by (*erule seq-lift.induct*) (*auto simp add: obs-consistent-def*)

Definition of refinement proof obligations.

definition

refines ::
 $[('s \times 't)\ set, 'p \implies 'o, ('s, 'o)\ spec, ('t, 'p)\ spec] \implies bool$

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:*

$\llbracket PO-refines\ R\ Sa\ Sc; obs-consistent\ R\ pi\ Sa\ Sc \rrbracket$
 $\implies refines\ R\ pi\ Sa\ Sc$
by (*simp add: refines-def*)

lemma *PO-refines-from-refines:*

$refines\ R\ pi\ Sa\ Sc \implies PO-refines\ R\ Sa\ Sc$
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:*

$\llbracket refines\ R1\ pi1\ S1\ S2; refines\ R2\ pi2\ S2\ S3 \rrbracket$
 $\implies refines\ (R1\ O\ R2)\ (pi1\ o\ pi2)\ S1\ S3$

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:*

$\llbracket \text{refines } R \text{ pi } Sa \text{ Sc}; bc \in \text{obeh } Sc \rrbracket \implies \text{map pi } bc \in \text{obeh } Sa$
by (*auto simp add: refines-def obeh-def image-def*
dest!: behaviour-refinement obs-consistent-behaviours)

theorem refinement-soundness:
refines R pi Sa Sc \implies 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')) \mid r \ r' \ st \ st'. \$
 $\quad r' = \text{Suc } r$
 $\quad \wedge \text{CSHOnextConfig } A \ r \ st \ (\text{HOs } r) \ (\text{SHOs } r) \ (\text{coord } r) \ st'$
 $\}$

definition *CHO-to-TS* ::

$(\text{'proc}, \text{'pst}, \text{'msg}) \text{ CHOAlgorithm}$
 $\Rightarrow (\text{nat} \Rightarrow \text{'proc HO})$
 $\Rightarrow (\text{nat} \Rightarrow \text{'proc HO})$
 $\Rightarrow (\text{nat} \Rightarrow \text{'proc coord})$
 $\Rightarrow (\text{nat} \times (\text{'proc} \Rightarrow \text{'pst})) \text{ TS}$

where

$\text{CHO-to-TS } A \text{ HOs SHOs coord} \equiv \langle$
 $\text{init} = \{(0, \text{st}) \mid \text{st. CHOinitConfig } A \text{ st } (\text{coord } 0)\},$
 $\text{trans} = \text{CHO-trans } A \text{ HOs SHOs coord}$
 \rangle

definition *get-msgs* ::

$(\text{'proc} \Rightarrow \text{'proc} \Rightarrow \text{'pst} \Rightarrow \text{'msg})$
 $\Rightarrow (\text{'proc} \Rightarrow \text{'pst})$
 $\Rightarrow \text{'proc HO}$
 $\Rightarrow \text{'proc HO}$
 $\Rightarrow \text{'proc} \Rightarrow (\text{'proc} \multimap \text{'msg}) \text{set}$

where

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

definition *CSHO-trans-alt*

::
 $(\text{nat} \Rightarrow \text{'proc} \Rightarrow \text{'proc} \Rightarrow \text{'pst} \Rightarrow \text{'msg})$
 $\Rightarrow (\text{nat} \Rightarrow \text{'proc} \Rightarrow \text{'pst} \Rightarrow (\text{'proc} \multimap \text{'msg}) \Rightarrow \text{'proc} \Rightarrow \text{'pst} \Rightarrow \text{bool})$
 $\Rightarrow (\text{nat} \Rightarrow \text{'proc HO})$
 $\Rightarrow (\text{nat} \Rightarrow \text{'proc HO})$
 $\Rightarrow (\text{nat} \Rightarrow \text{'proc} \Rightarrow \text{'proc})$
 $\Rightarrow ((\text{nat} \times (\text{'proc} \Rightarrow \text{'pst})) \times (\text{nat} \times (\text{'proc} \Rightarrow \text{'pst}))) \text{set}$

where

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

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 ≡ λx. 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)
 $\wedge (\forall r. \exists \mu.$
 $(\forall p.$
 $\mu p \in \text{get-msgs (sendMsg A r) (rho r) (HOs r) (SHOs r) p}$
 $\wedge \text{CnextState A r p (rho r p) (\mu p) (coords (Suc r) p) (rho (Suc r) p))$
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 ∈ get-msgs send s HOs SHOs p ⇒ 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 (λq. (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* = λ*r 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**

$locked\text{-}in\text{-}vf\ v\text{-}f\ v \equiv \exists Q. quorum\text{-}for\ Q\ v\ v\text{-}f$

definition $locked\text{-}in :: v\text{-}state \Rightarrow round \Rightarrow val \Rightarrow bool$ **where**
 $locked\text{-}in\ s\ r\ v = locked\text{-}in\text{-}vf\ (votes\ s\ r)\ v$

definition $d\text{-}guard :: (process \Rightarrow val\ option) \Rightarrow (process \Rightarrow val\ option) \Rightarrow bool$
where

$d\text{-}guard\ r\text{-}decisions\ r\text{-}votes \equiv \forall p\ v.$
 $r\text{-}decisions\ p = Some\ v \longrightarrow locked\text{-}in\text{-}vf\ r\text{-}votes\ v$

definition $no\text{-}defection :: v\text{-}state \Rightarrow (process, val)map \Rightarrow round \Rightarrow bool$ **where**
 $no\text{-}defection\text{-}def'$:

$no\text{-}defection\ s\ r\text{-}votes\ r \equiv$
 $\forall r' < r. \forall Q \in Quorum. \forall v. (votes\ s\ r') \text{ ' } Q = \{Some\ v\} \longrightarrow r\text{-}votes \text{ ' } Q \subseteq$
 $\{None, Some\ v\}$

The following definition of $no\text{-}defection$ is easier to reason about in Isabelle.

lemma $no\text{-}defection\text{-}def$:

$no\text{-}defection\ s\ round\text{-}votes\ r =$
 $(\forall r' < r. \forall a\ Q\ v. quorum\text{-}for\ Q\ v\ (votes\ s\ r') \wedge a \in Q \longrightarrow round\text{-}votes\ a \in$
 $\{None, Some\ v\})$

apply($auto\ simp\ add: no\text{-}defection\text{-}def'\ Ball\text{-}def\ quorum\text{-}for\text{-}def'$)

apply($blast$)

by ($metis\ option.\text{discI}\ option.\text{inject}$)

definition $locked :: v\text{-}state \Rightarrow val\ set$ **where**

$locked\ s = \{v. \exists r. locked\text{-}in\ s\ r\ v\}$

The sole system event.

definition $v\text{-}round :: round \Rightarrow (process, val)map \Rightarrow (process, val)map \Rightarrow (v\text{-}state$
 $\times v\text{-}state)$ **set** **where**

$v\text{-}round\ r\ r\text{-}votes\ r\text{-}decisions = \{(s, s')\}.$

— guards

$r = next\text{-}round\ s$

$\wedge no\text{-}defection\ s\ r\text{-}votes\ r$

$\wedge d\text{-}guard\ r\text{-}decisions\ r\text{-}votes$

\wedge — actions

$s' = s|$

$next\text{-}round := Suc\ r,$

$votes := (votes\ s)(r := r\text{-}votes),$

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  $\subseteq$  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 \subseteq 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  $\in$  Vinv1  $\implies$  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}  $\times$  UNIV])
apply(auto simp add: Vinv1-def dom-def not-le[symmetric])
done
```

lemma *Vinv1-finite-vote-set*:

```
s  $\in$  Vinv1  $\implies$  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= $\lambda((r, a), v). (r, v)$ ])
by(force simp add: image-def)
```

lemma *process-mru-map-add*:

assumes

$s \in Vinv1$

shows

```
process-mru ((votes s)(next-round s := v-f)) =
(process-mru (votes s) ++ ( $\lambda p. \text{map-option (Pair (next-round s)) (v-f p)}$ ))
```

proof–

from *assms*[*THEN* *Vinv1D*] **have** *empty*: $\forall r' \geq \text{next-round } s. \text{votes } s \ r' =$

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 ∈ Vinv1
r = next-round s
no-defection s v-f r
no-defection s (votes s r') r'
votes s' = (votes s)(r := v-f)
shows
no-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-TS ⊆ Vinv2
 $\{Vinv2 \cap Vinv1\} \text{trans } v\text{-TS } \{> Vinv2\}$
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-TS \subseteq Vinv2*

by (*rule inv-rule-incr, auto intro: Vinv2-inductive Vinv1-invariant del: subsetI*)

lemma *locked-preserved:*

assumes

$s \in \text{Vinv1}$

$r = \text{next-round } s$

$\text{votes } s' = (\text{votes } s)(r := v\text{-f})$

shows

$\text{locked } s \subseteq \text{locked } s'$ **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:*

$\{\text{Vinv3} \cap \text{Vinv1}\} \text{ v-round } r \text{ v-f d-f } \{> \text{Vinv3}\}$

proof(*clarsimp simp add: PO-hoare-defs, intro Vinv3I, safe*)

fix $s \ s' \ v$

assume *step: $(s, s') \in \text{v-round } r \text{ v-f d-f}$ and inv3: $s \in \text{Vinv3}$ and inv1: $s \in \text{Vinv1}$*

and *dec: $v \in \text{ran (decisions } s')$*

have $\text{locked } s \subseteq \text{locked } s'$ **using** *step*

by(*intro locked-preserved[OF inv1, where $s'=s'$] (auto simp add: v-round-def)*)

with *Vinv3D[OF inv3] step dec*

show $v \in \text{locked } s'$

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 \subseteq Vinv3

$\{Vinv3 \cap Vinv1\} \text{ trans } v\text{-TS } \{> Vinv3\}$
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 \subseteq Vinv3*
by (rule inv-rule-incr, auto intro: Vinv3-inductive Vinv1-invariant del: subsetI)

4.3 Agreement and stability

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

lemma *locked-in-vf-same:*
 $\llbracket \text{locked-in-vf } v\text{-f } v; \text{locked-in-vf } v\text{-f } w \rrbracket \implies v = w$ **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

$s \in Vinv2$
 $\text{locked-in } s \ r1 \ v$
 $\text{locked-in } s \ r2 \ w$
 $r1 < r2$

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 $Q12$: $Q1 \in \text{Quorum } Q2 \in \text{Quorum } \text{quorum-for } Q1 \ v \ (\text{votes } s \ r1) \ \text{quorum-for } Q2 \ w \ (\text{votes } s \ r2)$

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** $a \in Q1 \ \text{votes } s \ r2 \ a = \text{Some } w$

using *qintersect[OF $\langle Q1 \in \text{Quorum} \rangle \langle Q2 \in \text{Quorum} \rangle$]*

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** $\langle s \in Vinv2 \rangle \langle \text{quorum-for } Q1 \ v \ (\text{votes } s \ r1) \rangle \langle r1 < r2 \rangle$

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

$s \in \text{Vinv2}$

$v \in \text{locked } s \ w \in \text{locked } s$

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 $r1 < r2 \vee r1 = r2 \vee r2 < r1$ **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:  $s \in \text{Vinv2}$  and invs3:  $s \in \text{Vinv3}$ 
  by(blast dest: Vinv2-invariant[THEN subsetD] Vinv3-invariant[THEN subsetD])+

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  $t' = \text{tr} ! (i - j)$ 
    — Both  $t$  and  $t'$  are reachable, thus respect the invariants, and they are related by the transition relation.
    hence  $t' \in \text{reach } v\text{-TS}$   $t \in \text{reach } v\text{-TS}$  using beh len Suc
      by (metis beh-in-reach less-imp-diff-less nth-mem)+
    hence invs:  $t' \in \text{Vinv1}$   $t' \in \text{Vinv3}$   $t \in \text{Vinv2}$   $t \in \text{Vinv3}$ 
      by(blast dest: Vinv1-invariant[THEN subsetD] Vinv2-invariant[THEN subsetD]
Vinv3-invariant[THEN subsetD])+
    hence locked-v:  $v \in \text{locked } t'$  using Suc
      by(auto simp add: t'-def intro: ranI)
    have  $i - j = \text{Suc } (i - (\text{Suc } j))$  using False
      by simp
    hence trans:  $(t', t) \in \text{trans } v\text{-TS}$  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:  $v \in \text{locked } t$  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 precedes s in the trace.

lemma *Voting-agreement*:

```

assumes beh:  $tr \in \text{beh } v\text{-TS}$ 
and len:  $i < \text{length } tr$ 
and s:  $s = \text{nth } tr \ i$ 
and t:  $t = \text{nth } tr \ (i - j)$ 
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:  $s \in \text{reach } v\text{-TS}$  using beh s t len
apply(simp-all add: reach-equiv-beh-states)
by (metis nth-mem)
hence invs2:  $s \in \text{Vinv2}$  and invs3:  $s \in \text{Vinv3}$ 
by(blast dest: Vinv2-invariant[THEN subsetD] Vinv3-invariant[THEN sub-
setD])+

```

— We now proceed to prove the thesis by induction.

```

thus ?thesis using assms
proof(induction j arbitrary: t)
  case 0
  hence
     $v \in \text{locked } (tr ! i)$ 
     $w \in \text{locked } (tr ! i)$ 
    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  $i \leq j$ )
    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  $\forall r' > r . \text{votes sa } r' p = \text{None}$ 
proof(safe)
  fix r'
  assume less:  $r < r'$ 
  hence  $\forall v. (r', v) \notin \text{vote-set (votes sa) \{p\}}$  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)  $\in$  flv-ref-rel
  and ainv: sa  $\in$  Vinv1 sa  $\in$  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  $\in$  Q
  have Q  $\in$  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  $q \in Q$ 
  with  $Q$  have  $q-r'$ : votes sa  $r' q = \text{Some } v$ 
    by(auto simp add: quorum-for-def)
  hence votes: vote-set (votes sa)  $\{q\} \neq \{\}$ 
    by(auto simp add: vote-set-def)
  then obtain  $w$  where  $w$ : last-vote sc  $q = \text{Some } w$  using  $R$ 
    by(clarsimp simp add: flv-ref-rel-def process-mru-def mru-of-set-def
      option-Max-by-def)
  with  $R$  obtain  $r\text{-max}$  where process-mru (votes sa)  $q = \text{Some } (r\text{-max}, w)$ 
    by(clarsimp simp add: flv-ref-rel-def)
  from process-mru-Max[OF ainv(1) this]  $q-r'$  have
    votes sa  $r\text{-max}$   $q = \text{Some } w$ 
     $r' \leq r\text{-max}$ 
    using  $q-r'$ 
    by(auto simp add: not-less[symmetric])
  thus last-vote sc  $q = \text{Some } v$  using ainv(2)  $Q \langle q \in Q \rangle$ 
    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 \in *Vinv1*

shows

map-option snd o (process-mru ((votes s)(v-state.next-round s := v-f)))
 $= ((\text{map-option snd } o \text{ (process-mru (votes s))}) ++ v\text{-f})$

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*:

$\{\text{flv-ref-rel} \cap (\text{Vinv1} \cap \text{Vinv2}) \times \text{UNIV}\}$

```

    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  $\cap$  (Vinv1  $\cap$  Vinv2)  $\times$  UNIV) v-TS flv-TS
proof(rule refine-using-invariants)
  show init flv-TS  $\subseteq$  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  $\cap$  (Vinv1  $\cap$  Vinv2)  $\times$  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  $\cap$  Vinv2  $\cap$  Domain (flv-ref-rel  $\cap$  UNIV  $\times$  UNIV)}
      trans v-TS
    {> Vinv1  $\cap$  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  $\equiv$  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  $\Rightarrow$  'val option)  $\Rightarrow$  'val  $\Rightarrow$  process set where
  HOV msgs v  $\equiv$  { q . msgs q = Some v }
```

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

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

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

```
lemma MFR-exists:  $\exists v$ . MFR msgs v
```

proof –

```
  let ?cards = { card (HOV msgs v) | v . True }
  let ?mfr = Max ?cards
  have  $\forall v$ . card (HOV msgs v)  $\leq$  N by (auto intro: card-mono)
  hence ?cards  $\subseteq$  { 0 .. N } by auto
  hence fin: finite ?cards by (metis atLeast0AtMost finite-atMost finite-subset)
  hence ?mfr  $\in$  ?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)  $\leq$  ?mfr by (rule Max-ge) auto
```

thus $\text{card } (HOV \text{ msgs } w) \leq \text{card } (HOV \text{ msgs } v)$ **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 $HO:HOs \ m \ p \neq \{\}$
and $v: MFR (HORcvdMsgs \ OTR-M \ m \ p \ (HOs \ m \ p) \ (\rho \ m)) \ v$
(is $MFR \ ?msgs \ v)$
shows $\exists q \in HOs \ m \ p. \ v = \text{the } (?msgs \ q)$
proof –
from HO **obtain** q **where** $q: q \in HOs \ m \ p$
by *auto*
with v **have** $HOV \ ?msgs \ (\text{the } (?msgs \ q)) \neq \{\}$
by (*auto simp: HOV-def HORcvdMsgs-def*)
hence $HOp: 0 < \text{card } (HOV \ ?msgs \ (\text{the } (?msgs \ q)))$
by *auto*
also from v **have** $\dots \leq \text{card } (HOV \ ?msgs \ v)$
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**

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

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**

$$\begin{aligned} \text{OTR-nextState } r \ p \ (st::('val::linorder) \ pstate) \ msgs \ st' \equiv \\ \text{if } (2*N) \ \text{div } 3 < \text{card } \{q. \ msgs \ q \neq \text{None}\} \\ \text{then } st' = (\text{last-vote} = \text{Min } \{v. \ \text{MFR } \ msgs \ v\}, \\ \quad \text{decision} = (\text{if } (\exists v. \ \text{TwoThirds } \ msgs \ v) \\ \quad \quad \text{then } \text{Some } (\epsilon v. \ \text{TwoThirds } \ msgs \ v) \\ \quad \quad \text{else } \text{decision } st) \) \\ \text{else } st' = st \end{aligned}$$

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**

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

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**

$$\text{OTR-commPerRd } HOs \equiv \text{True}$$

definition *OTR-commGlobal* **where**

$$\begin{aligned} \text{OTR-commGlobal } HOs \equiv \\ \forall r. \ \exists r0 \ \Pi. \ r0 \geq r \wedge (\forall p. \ HOs \ r0 \ p = \Pi) \wedge \text{card } \Pi > (2*N) \ \text{div } 3 \end{aligned}$$

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**

$$\begin{aligned} \text{OTR-HOMachine} = \\ (\text{CinitState} = (\lambda p \ st \ \text{crd}. \ \text{OTR-initState } p \ st), \end{aligned}$$

```

sendMsg = OTR-sendMsg,
CnextState = ( $\lambda$  r p st msgs crd st'. OTR-nextState r p st msgs st'),
HOcommPerRd = OTR-commPerRd,
HOcommGlobal = OTR-commGlobal )

```

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

end

6.4 Proofs

definition $majs :: (\text{process set}) \text{ set}$ **where**
 $majs \equiv \{S. \text{card } S > (2 * N) \text{ div } 3\}$

lemma card-Compl :

fixes $S :: ('a :: \text{finite}) \text{ set}$

shows $\text{card } (-S) = \text{card } (UNIV :: 'a \text{ set}) - \text{card } S$

proof –

have $\text{card } S + \text{card } (-S) = \text{card } (UNIV :: 'a \text{ set})$

by ($\text{rule } \text{card-Un-disjoint}[\text{of } S -S, \text{simplified Compl-partition, symmetric}]$)
 (auto)

thus $?thesis$

by simp

qed

lemma $m\text{-mult-div-Suc-}m$:

$n > 0 \implies m * n \text{ div } \text{Suc } m < n$

by ($\text{simp add: less-mult-imp-div-less}$)

interpretation $\text{majorities: quorum-process } majs$

proof

fix $Q Q'$ **assume** $Q \in majs Q' \in majs$

hence $(4 * N) \text{ div } 3 < \text{card } Q + \text{card } Q'$

by ($\text{auto simp add: majs-def}$)

thus $Q \cap Q' \neq \{\}$

apply ($\text{intro majorities-intersect}$)

apply (auto)

done

next


```

have  $N > 0$ 
  by auto
have  $2 * N \text{ div } 3 < N$ 
  by(simp only: eval-nat-numeral m-mult-div-Suc-m[OF <N > 0>])
thus  $\exists Q. Q \in \text{maj}s$ 
  apply(rule-tac x=UNIV in exI)
  apply(auto simp add: maj-s-def intro!: div-less-dividend)
  done
qed

lemma card-Un-le:
   $\llbracket \text{finite } A; \text{finite } B \rrbracket \implies \text{card } (A \cup B) \leq \text{card } A + \text{card } B$ 
  by(simp only: card-Un-Int)

lemma qintersect-card:
  assumes  $Q \in \text{maj}s \ Q' \in \text{maj}s$ 
  shows  $\text{card } (Q \cap Q') > \text{card } (Q \cap -Q')$ 
proof–
  have  $\text{card } (Q \cap -Q') \leq \text{card } (-Q')$ 
  by(auto intro!: card-mono)
  also have  $\dots < N - (\text{card } (-Q) + \text{card } (-Q'))$ 
proof–
  have sum:  $N < \text{card } Q + \text{card } Q'$  using assms
  by(auto simp add: maj-s-def)
  have le-N:  $\text{card } Q \leq N \ \text{card } Q' \leq N$  by (auto intro!: card-mono)
  show ?thesis using assms sum
  apply(simp add: card-Compl)
  apply(intro diff-less-mono2)
  apply(auto simp add: maj-s-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  $\dots \leq \text{card } (Q \cap Q')$ 
proof–
  have  $N - (\text{card } (-Q) + \text{card } (-Q')) \leq \text{card } (-(-Q \cup -Q'))$ 
  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 ≡ *λx. 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 ≡ $\bigcup r \mu.$
 $\{((r, cfg), Suc\ r, cfg') \mid cfg\ cfg'\}$
 $(\forall p. \mu\ p \in\ get\ msgs\ (OTR\ sendMsg\ r)\ cfg\ (HOs\ r)\ (HOs\ r)\ p) \wedge$

$(\forall p. \text{OTR-nextState } r \ p \ (cfg \ p) \ (\mu \ p) \ (cfg' \ p))\}$

definition *CSHOnextConfig* **where**

CSHOnextConfig $A \ r \ cfg \ HO \ SHO \ coord \ cfg' \equiv$
 $\forall p. \exists \mu \in \text{SHOmsgVectors } A \ r \ p \ cfg \ (HO \ p) \ (SHO \ p).$
 $CnextState \ A \ r \ p \ (cfg \ p) \ \mu \ (coord \ p) \ (cfg' \ p)$

type-synonym $rHO = nat \Rightarrow process \ HO$

6.4.1 Refinement

definition *otr-ref-rel* $:: (opt-v-state \times p-TS-state)set$ **where**

otr-ref-rel $= \{(sa, (r, sc)).$
 $r = next-round \ sa$
 $\wedge (\forall p. decisions \ sa \ p = decision \ (sc \ p))$
 $\wedge majorities.opt-no-defection \ sa \ (Some \ o \ last-vote \ o \ sc)$
 $\}$

lemma *decide-origin*:

assumes

send: $\mu \ p \in get-msgs \ (OTR-sendMsg \ r) \ sc \ (HOs \ r) \ (HOs \ r) \ p$
and *next*: $OTR-nextState \ r \ p \ (sc \ p) \ (\mu \ p) \ (sc' \ p)$
and *new-dec*: $decision \ (sc' \ p) \neq decision \ (sc \ p)$

shows

$\exists v. decision \ (sc' \ p) = Some \ v \wedge \{q. last-vote \ (sc \ q) = v\} \in majS$

proof–

from *new-dec* **and** *next* **obtain** v **where**

p-dec-v: $decision \ (sc' \ p) = Some \ v$
and *two-thirds-v*: $TwoThirds \ (\mu \ p) \ v$
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 $HO:dom \ msgs \neq \{\}$

and $v: MFR\ msgs\ v$
shows $\exists q \in dom\ msgs.\ v = the\ (msgs\ q)$
proof –
from HO **obtain** q **where** $q: q \in dom\ msgs$
by $auto$
with v **have** $HOV\ msgs\ (the\ (msgs\ q)) \neq \{\}$
by $(auto\ simp: HOV-def)$
hence $HO_p: 0 < card\ (HOV\ msgs\ (the\ (msgs\ q)))$
by $auto$
also from v **have** $\dots \leq card\ (HOV\ msgs\ v)$
by $(simp\ add: MFR-def)$
finally have $HOV\ msgs\ v \neq \{\}$
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 $sa\ r\ sc\ sc'\ \mu$
assume
 $R: (sa,\ r,\ sc) \in otr-ref-rel$
and $send: \forall p.\ \mu\ p \in get-msgs\ (OTR-sendMsg\ r)\ sc\ (HOs\ r)\ (HOs\ r)\ p$
and $next: \forall p.\ OTR-nextState\ r\ p\ (sc\ p)\ (\mu\ p)\ (sc'\ p)$
note $step=send\ next$
define $d-f$
where $d-f\ p = (if\ decision\ (sc'\ p) \neq decision\ (sc\ p)\ then\ decision\ (sc'\ p)\ else\ None)$ **for** p

define sa' **where** $sa' = (\langle$
 $opt-v-state.next-round = Suc\ r$
 $,\ opt-v-state.last-vote = opt-v-state.last-vote\ sa\ ++\ (Some\ o\ last-vote\ o\ sc)$
 $,\ opt-v-state.decisions = opt-v-state.decisions\ sa\ ++\ d-f$
 $\rangle)$

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 $v \neq \text{decision } (sc\ p)$
decision $(sc'\ p) = \text{Some } v$
from this and
decide-origin[where $\mu = \mu$ and $HOs = HOs$ and $sc' = sc'$, OF send[THEN spec,
of p] next[THEN spec, of p]]
show *quorum-process.locked-in-vf majs (Some \circ last-vote \circ sc) v*
by(*auto simp add: majorities.locked-in-vf-def majorities.quorum-for-def*)
qed
hence
(sa, sa') \in majorities.flv-round r (Some \circ last-vote \circ sc) d-f using R
by(*auto simp add: majorities.flv-round-def otr-ref-rel-def sa'-def*)
moreover have *($sa', Suc\ r, sc'$) \in otr-ref-rel*
proof(*unfold otr-ref-rel-def, safe*)
fix p
show *opt-v-state.decisions $sa'\ p = \text{decision } (sc'\ p)$ using R next[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 \circ last-vote \circ 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 : $Q \in \text{majs}$ **and** $Q-v$: $\forall q \in Q. \text{last-vote } (sc\ q) = v$ **and** $p-Q$: $p \in$
 Q
hence old: *last-vote $(sc\ p) = v$ by simp*
have v -maj: $\{q. \text{last-vote } (sc\ q) = v\} \in \text{majs}$ **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 $(sc'\ p) \neq v$*
let $?w = \text{last-vote } (sc'\ p)$
have
 w -MFR: $?w = \text{Min } \{z. \text{MFR } (\mu\ p)\ z\}$ (**is** $?w = \text{Min } ?\text{MFRs}$) **and** dom-maj :
 $\text{dom } (\mu\ p) \in \text{majs}$

```

    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 ( $\mu$  p)  $\neq$  {} by(elim majorities.quorum-non-empty)
from MFR-exists obtain mfr-v where mfr-v: mfr-v  $\in$  ?MFRs
    by fastforce
from not-empty obtain q z where  $\mu$  p q = Some z by(fastforce)
hence 0 < card (HOV ( $\mu$  p) (the ( $\mu$  p q)))
    by(auto simp add: HOV-def)
have ?w  $\in$  {z. MFR ( $\mu$  p) z}
proof(unfold w-MFR, rule Min-in)
    have ?MFRs  $\subseteq$  (the o ( $\mu$  p)) ‘ (dom ( $\mu$  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 ( $\mu$  p) v)  $\leq$  card (HOV ( $\mu$  p) ?w)
    by(auto simp add: MFR-def)
have dom ( $\mu$  p) = HOs r p using send[THEN spec, where x=p]
    by(auto simp add: get-msgs-def)
from this[symmetric] have  $\forall v'. \text{HOV } (\mu \text{ p}) \text{ } v' = \{q. \text{last-vote } (sc \text{ } q) = v'\}$ 
 $\cap$  dom ( $\mu$  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 ( $\mu$  p))  $\leq$  card ({q.
last-vote (sc q) = ?w}  $\cap$  dom ( $\mu$  p))
    using card-HOV
    by(simp)
have
    card ({q. last-vote (sc q) = v}  $\cap$  dom ( $\mu$  p))  $\leq$  card ({q. last-vote (sc q)
 $\neq$  v}  $\cap$  dom ( $\mu$  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

```

```

     $\exists sa'. (\exists ra \text{ v-f d-f. } (sa, sa') \in \text{quorum-process.flv-round maj}s \text{ ra v-f d-f})$ 
     $\wedge (sa', \text{Suc } r, sc') \in \text{otr-ref-rel}$ 
  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 CHOinitCon-
fig-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 maj)s, 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  $\equiv$  (decide st = None)
```

```
locale ate-parameters =
  fixes  $\alpha::nat$  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 Eqta:  $E > \alpha$ 
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  $\Rightarrow$  'val option)  $\equiv$ 
   $\{v. \forall w. \text{card } \{qq. \text{msgs } qq = \text{Some } w\} \leq \text{card } \{qq. \text{msgs } qq = \text{Some } v\}\}$ 
```


definition

$$Ate-sendMsg :: nat \Rightarrow process \Rightarrow process \Rightarrow 'val pstate \Rightarrow 'val$$
where

$$Ate-sendMsg r p q st \equiv x st$$
definition

$$Ate-nextState :: nat \Rightarrow process \Rightarrow ('val::linorder) pstate \Rightarrow (process \Rightarrow 'val option) \\ \Rightarrow 'val pstate \Rightarrow bool$$
where

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

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

definition *Ate-commPerRd* **where**

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

The global communication predicate stipulates the three following conditions:

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

definition

Ate-commGlobal **where**

$$Ate-commGlobal\ HOs\ SHOs \equiv \\ (\forall r\ p.\ \exists r' > r.\ card\ (HOs\ r'\ p) > T) \\ \wedge (\forall r\ p.\ \exists r' > r.\ card\ (SHOs\ r'\ p \cap HOs\ r'\ p) > E)$$

$$\begin{aligned} &\wedge (\forall r. \exists r' > r. \exists \pi1 \pi2. \\ &\quad \text{card } \pi1 > E - \alpha \\ &\quad \wedge \text{card } \pi2 > T \\ &\quad \wedge (\forall p \in \pi1. \text{HOS } r' p = \pi2 \wedge \text{SHOs } r' p \cap \text{HOS } r' p = \pi2)) \end{aligned}$$

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

```

abbreviation

```

Ate-M  $\equiv$  (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  $\equiv$  {S. card S > E}

```

interpretation *majorities*: *quorum-process majs*

```

proof
  fix  $Q Q'$  assume  $Q \in \text{maj}s$   $Q' \in \text{maj}s$ 
  hence  $N < \text{card } Q + \text{card } Q'$  using  $\text{maj}E$ 
    by( $\text{auto simp add: maj}s\text{-def}$ )
  thus  $Q \cap Q' \neq \{\}$ 
    apply ( $\text{intro majorities-intersect}$ )
    apply( $\text{auto}$ )
    done
next
  from  $\text{Elt}N$ 
  show  $\exists Q. Q \in \text{maj}s$ 
    apply( $\text{rule-tac } x = \text{UNIV in } \text{exI}$ )
    apply( $\text{auto simp add: maj}s\text{-def intro!: div-less-dividend}$ )
    done
qed

type-synonym  $p\text{-TS-state} = (\text{nat} \times (\text{process} \Rightarrow (\text{val } p\text{state})))$ 

```

definition K **where**

$K y \equiv \lambda x. y$

definition Ate-Alg **where**

$\text{Ate-Alg} =$
 \langle $\text{CinitState} = (\lambda p \text{ st } \text{crd}. \text{Ate-initState } p \text{ st}),$
 $\text{sendMsg} = \text{Ate-sendMsg},$
 $\text{CnextState} = (\lambda r p \text{ st } \text{msgs } \text{crd } \text{st}'. \text{Ate-nextState } r p \text{ st } \text{msgs } \text{st}')$
 \rangle

definition $\text{Ate-TS} ::$

$(\text{round} \Rightarrow \text{process } HO)$
 $\Rightarrow (\text{round} \Rightarrow \text{process } HO)$
 $\Rightarrow (\text{round} \Rightarrow \text{process})$
 $\Rightarrow p\text{-TS-state } TS$

where

$\text{Ate-TS } HOs \text{ SHOs } \text{crds} = \text{CHO-to-TS } \text{Ate-Alg } HOs \text{ SHOs } (K \text{ o } \text{crds})$

lemmas $\text{Ate-TS-defs} = \text{Ate-TS-def } \text{CHO-to-TS-def } \text{Ate-Alg-def } \text{CHOinitConfig-def}$
 Ate-initState-def

definition

Ate-trans-step $HOs \equiv \bigcup r \mu.$
 $\{((r, cfg), Suc\ r, cfg') \mid cfg\ cfg'\}.$
 $(\forall p. \mu\ p \in get\ msgs\ (Ate\ sendMsg\ r)\ cfg\ (HOs\ r)\ (HOs\ r)\ p) \wedge$
 $(\forall p. Ate\ nextState\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p))\}$

definition *CSHONextConfig* **where**

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

type-synonym $rHO = nat \Rightarrow process\ HO$

7.4.1 Refinement

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

$ate\ ref\ rel = \{(sa, (r, sc)).$
 $r = next\ round\ sa$
 $\wedge (\forall p. decisions\ sa\ p = Ate\ Defs.\ decide\ (sc\ p))$
 $\wedge majorities.\ opt\ no\ defection\ sa\ (Some\ o\ x\ o\ sc)$
 $\}$

lemma *decide-origin*:

assumes

$send: \mu\ p \in get\ msgs\ (Ate\ sendMsg\ r)\ sc\ (HOs\ r)\ (HOs\ r)\ p$
and $next: Ate\ nextState\ r\ p\ (sc\ p)\ (\mu\ p)\ (sc'\ p)$
and $new\ dec: decide\ (sc'\ p) \neq decide\ (sc\ p)$

shows

$\exists v. decide\ (sc'\ p) = Some\ v \wedge \{q. x\ (sc\ q) = v\} \in majs$ **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 $next: Ate\ nextState\ r\ q\ (sc\ q)\ \mu\ q\ ((sc')\ q)$
and $\mu\ q: \mu\ q \in get\ msgs\ (Ate\ sendMsg\ r)\ sc\ (HOs\ r)\ (HOs\ r)\ q$
and $vsent: card\ \{qq. sendMsg\ Ate\ M\ r\ qq\ q\ (sc\ qq) = v\} > E - \alpha$
(is $card\ ?vsent > -)$

shows $card\ (\{qq. \mu\ q\ qq \neq Some\ v\} \cap HOs\ r\ q) \leq N + 2*\alpha - E$

proof –

from $next\ \mu\ q$

have $(\{qq. \mu\ q\ qq \neq Some\ v\} \cap HOs\ r\ q) - (HOs\ r\ q - HOs\ r\ q)$

$\subseteq \{qq. \text{sendMsg Ate-M } r \text{ } qq \text{ } q \text{ } (sc \text{ } qq) \neq v\}$
 (is ?notrcvd - ?aho \subseteq ?notvsent)
unfolding get-msgs-def SHOMsgVectors-def Ate-SHOMachine-def **by** auto
hence card ?notvsent \geq card (?notrcvd - ?aho)
and card (?notrcvd - ?aho) \geq card ?notrcvd - card ?aho
by (auto simp: card-mono diff-card-le-card-Diff)
moreover
have 1: card ?notvsent + card ?vsent = card (?notvsent \cup ?vsent)
by (subst card-Un-Int) auto
have ?notvsent \cup ?vsent = (UNIV::process set) **by** auto
hence card (?notvsent \cup ?vsent) = N **by** simp
with 1 vsent **have** card ?notvsent $\leq N - (E + 1 - \alpha)$ **by** auto
ultimately
show ?thesis **using** EltN Egta **by** auto
qed

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

lemma mostOftenRcvd-v:

assumes nxt: Ate-nextState $r \text{ } q \text{ } (sc \text{ } q) \text{ } \mu q \text{ } ((sc') \text{ } q)$
and muq: $\mu q \in \text{get-msgs (Ate-sendMsg } r) \text{ } sc \text{ } (HOs \text{ } r) \text{ } (HOs \text{ } r) \text{ } q$
and threshold-T: card $\{qq. \mu q \text{ } qq \neq \text{None}\} > T$
and threshold-E: card $\{qq. \text{sendMsg Ate-M } r \text{ } qq \text{ } q \text{ } (sc \text{ } qq) = v\} > E - \alpha$
shows mostOftenRcvd $\mu q = \{v\}$

proof –

from muq **have** hodef: $HOs \text{ } r \text{ } q = \{qq. \mu q \text{ } qq \neq \text{None}\}$
unfolding get-msgs-def SHOMsgVectors-def **by** auto

from nxt muq threshold-E

have card $(\{qq. \mu q \text{ } qq \neq \text{Some } v\} \cap HOs \text{ } r \text{ } q) \leq N + 2*\alpha - E$
 (is card ?heardnotv \leq -)
by (rule other-values-received)

moreover

have card ?heardnotv $\geq T + 1 - \text{card } \{qq. \mu q \text{ } qq = \text{Some } v\}$

proof –

from muq

have ?heardnotv = $(HOs \text{ } r \text{ } q) - \{qq. \mu q \text{ } qq = \text{Some } v\}$

and $\{qq. \mu q \text{ } qq = \text{Some } v\} \subseteq HOs \text{ } r \text{ } q$

unfolding SHOMsgVectors-def get-msgs-def **by** auto

```

hence  $\text{card } ?\text{heardnotv} = \text{card } (HOs\ r\ q) - \text{card } \{qq.\ \mu q\ qq = \text{Some } v\}$ 
  by (auto simp: card-Diff-subset)
with hodef threshold-T show ?thesis by auto
qed
ultimately
have  $\text{card } \{qq.\ \mu q\ qq = \text{Some } v\} > \text{card } ?\text{heardnotv}$ 
  using TNaE by auto
moreover
{
  fix w
  assume w: w ≠ v
  with hodef have  $\{qq.\ \mu q\ qq = \text{Some } w\} \subseteq ?\text{heardnotv}$  by auto
  hence  $\text{card } \{qq.\ \mu q\ qq = \text{Some } w\} \leq \text{card } ?\text{heardnotv}$  by (auto simp: card-mono)
}
ultimately
have  $\{w.\ \text{card } \{qq.\ \mu q\ qq = \text{Some } w\} \geq \text{card } \{qq.\ \mu q\ qq = \text{Some } v\}\} = \{v\}$ 
  by force
thus ?thesis unfolding mostOftenRcvd-def by auto
qed

```

lemma *step-ref*:

```

{ate-ref-rel}
  ( $\bigcup v\text{-f } d\text{-f}.\ \text{majorities.flv-round } r\ v\text{-f } d\text{-f}$ ),
  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 next: ∀ p. Ate-nextState r p (sc p) (μ p) (sc' p)
note step=send next
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

```

\triangleright

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 \neq decide (sc p)
decide (sc' p) = Some v
from this and
decide-origin[where $\mu=\mu$ 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 o x o sc*) *v*
by(*auto simp add: majorities.locked-in-vf-def majorities.quorum-for-def*)
qed
hence
(sa, sa') \in 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'*) \in *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 o x o 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 \in majs and Q-v: $\forall q \in Q. x (sc q) = v$ and p-Q: $p \in Q$*
hence *old: x (sc p) = v* **by** *simp*

have *v-maj: {q. x (sc q) = v} \in 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. $\mu p qq \neq None$ }*)
case *True*
have

```

       $E - \alpha < \text{card } \{qq. \text{ sendMsg Ate-M } r \text{ } qq \text{ } p \text{ } (sc \text{ } qq) = v\}$  using v-maj
      by(auto simp add: Ate-SHOMachine-def Ate-sendMsg-def majjs-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
   $\exists sa'. (\exists ra \text{ } v\text{-f } d\text{-f}. (sa, sa') \in \text{quorum-process.flv-round majjs } ra \text{ } v\text{-f } d\text{-f})$ 
   $\wedge (sa', Suc \text{ } r, sc') \in \text{ate-ref-rel}$ 
  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
majjs)
    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
     $\{ate-ref-rel\} \text{ } TS.trans \text{ } (quorum-process.flv-TS \text{ } majjs), TS.trans \text{ } (Ate-TS \text{ } HOs$ 
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* :: $v\text{-state} \Rightarrow \text{round} \Rightarrow \text{val} \Rightarrow \text{bool}$ **where**
safe-def': $\text{safe } s \ r \ v \equiv$
 $\forall r' < r. \forall Q \in \text{Quorum}. \forall w. (\text{votes } s \ r') \wedge Q = \{\text{Some } w\} \longrightarrow v = w$

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

lemma *safe-def*:
 $\text{safe } s \ r \ v =$
 $(\forall r' < r. \forall Q \ w. \text{quorum-for } Q \ w \ (\text{votes } s \ r') \longrightarrow v = w)$
by(*auto simp add: safe-def' quorum-for-def' Ball-def*)

definition *sv-round* :: $\text{round} \Rightarrow \text{process set} \Rightarrow \text{val} \Rightarrow (\text{process}, \text{val})\text{map} \Rightarrow (v\text{-state} \times v\text{-state}) \text{ set}$ **where**

sv-round $r \ S \ v \ r\text{-decisions} = \{(s, s')\}.$
— guards
 $r = \text{next-round } s$
 $\wedge (S \neq \{\}) \longrightarrow \text{safe } s \ r \ v)$
 $\wedge \text{d-guard } r\text{-decisions } (\text{const-map } v \ S)$
— actions
 $s' = s|$
 $\text{next-round} := \text{Suc } r$

```

    , votes := (votes s)(r := const-map v S)
    , decisions := (decisions s ++ r-decisions)
  }
}

```

definition *sv-trans* :: (*v-state* × *v-state*) set **where**
sv-trans = ($\bigcup r S v D. sv-round\ r\ S\ v\ D$) \cup *Id*

definition *sv-TS* :: *v-state* *TS* **where**
sv-TS = (\langle *init* = *v-init*, *trans* = *sv-trans* \rangle)

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 \implies *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 \in *Quorum* \implies *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 = $\{s. \forall r a b v w.$
votes s r a = Some v \wedge *votes s r b = Some w* $\longrightarrow v = w$
 $\}$

lemmas $SV\text{-inv}3I = SV\text{-inv}3\text{-def}$ [*THEN setc-def-to-intro, rule-format*]
lemmas $SV\text{-inv}3E$ [*elim*] = $SV\text{-inv}3\text{-def}$ [*THEN setc-def-to-elim, rule-format*]
lemmas $SV\text{-inv}3D = SV\text{-inv}3\text{-def}$ [*THEN setc-def-to-dest, rule-format*]

8.3.1 Proof of invariants

lemma $SV\text{-inv}3\text{-v-round}$:
 $\{SV\text{-inv}3\}$ *sv-round* $r S v D \{> SV\text{-inv}3\}$
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 $SV\text{-inv}3\text{-event-pres} = SV\text{-inv}3\text{-v-round}$

lemma $SV\text{-inv}3\text{-inductive}$:
 $init\ sv\text{-}TS \subseteq SV\text{-inv}3$
 $\{SV\text{-inv}3\}$ *trans sv-TS* $\{> SV\text{-inv}3\}$
apply (*simp add: sv-TS-defs SV-inv3-def*)
by (*auto simp add: sv-TS-defs SV-inv3-event-pres*)

lemma $SV\text{-inv}3\text{-invariant}$: $reach\ sv\text{-}TS \subseteq SV\text{-inv}3$
by (*auto intro!: inv-rule-basic SV-inv3-inductive del: subsetI*)

This is a different characterization of *safe*, due to Lampton [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: $s \in SV\text{-inv}3 \implies safe' s = safe\ s$.

8.3.2 Transfer of abstract invariants

lemma $SV\text{-inv}1\text{-inductive}$:
 $init\ sv\text{-}TS \subseteq V\text{inv}1$
 $\{V\text{inv}1\}$ *trans sv-TS* $\{> V\text{inv}1\}$
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 \subseteq Vinv1

by(rule *abs-INV-transfer*[*OF Same-Vote-Refines Vinv1-invariant, simplified*])

lemma *SV-inv2-inductive:*

init sv-TS \subseteq Vinv2

{ Vinv2 \cap 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 \subseteq 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. $\forall v a r. votes s r a = Some v \longrightarrow safe s (Suc r) v$ }

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 \cap (Vinv1 \cap 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') \in sv-round r S v D

and *invs: s \in SV-inv4 s \in Vinv1 s \in 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  $\subseteq$  SV-inv4
  {SV-inv4  $\cap$  (Vinv1  $\cap$  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

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

definition *obsv-round*

$:: round \Rightarrow process\ set \Rightarrow val \Rightarrow (process, val)map \Rightarrow process\ set \Rightarrow (obsv\text{-}state \times obsv\text{-}state)\ set$

where

$obsv\text{-}round\ r\ S\ v\ r\text{-}decisions\ Os = \{(s, s')\}.$

— guards

$r = next\text{-}round\ s$

$\wedge (S \neq \{\}) \longrightarrow obs\text{-}safe\ r\ s\ v$

$\wedge d\text{-}guard\ r\text{-}decisions\ (const\text{-}map\ v\ S)$

$\wedge (S \in Quorum \longrightarrow Os = UNIV)$

$\wedge (Os \neq \{\}) \longrightarrow S \neq \{\}$

\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* $:: (obsv\text{-}state \times obsv\text{-}state)\ set$ **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* $:: obsv\text{-}state\ set$ **where**

$obsv\text{-}init = \{ \langle next\text{-}round = 0, votes = \lambda r\ a. None, decisions = Map.empty, obs = \lambda r\ a. None \rangle \}$

definition *obsv-TS* $:: obsv\text{-}state\ TS$ **where**

$obsv\text{-}TS = \langle init = obsv\text{-}init, trans = obsv\text{-}trans \rangle$

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

9.2 Invariants

definition *OV-inv1* **where**

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

lemmas $OV\text{-}inv1I = OV\text{-}inv1\text{-}def$ [*THEN setc-def-to-intro, rule-format*]

lemmas $OV\text{-}inv1E$ [*elim*] = $OV\text{-}inv1\text{-}def$ [*THEN setc-def-to-elim, rule-format*]

lemmas $OV\text{-}inv1D = OV\text{-}inv1\text{-}def$ [*THEN setc-def-to-dest, rule-format*]

9.2.1 Proofs of invariants

lemma $OV\text{-}inv1\text{-}obsv\text{-}round$:

$\{OV\text{-}inv1\}$ *obsv-round* $r S v d\text{-}f Ob \{> OV\text{-}inv1\}$

proof(*clarsimp simp add: PO-hoare-defs intro!: OV\text{-}inv1I*)

fix $v' s s' Q Q' r'$

assume

Q : *quorum-for* $Q v' (\text{votes } s' r')$

and inv : $s \in OV\text{-}inv1$

and $step$: $(s, s') \in \text{obsv-round } r S v d\text{-}f Ob$

and $quorum$: $Q' \in \text{Quorum}$

from $Q inv$ [*THEN OV\text{-}inv1D*] $step quorum$

show *quorum-for* $Q' v' (\text{obs } s' r')$

proof(*cases r'=r*)

case *True*

with $step$ **and** Q **have** $S \in \text{Quorum}$

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\text{-}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 $OV\text{-}inv1\text{-}inductive$:

init $obsv\text{-}TS \subseteq OV\text{-}inv1$

$\{OV\text{-}inv1\}$ *trans* $obsv\text{-}TS \{> OV\text{-}inv1\}$

apply (*simp add: obsv-TS-defs OV\text{-}inv1-def*)

apply (*auto simp add: obsv-TS-defs OV\text{-}inv1-obsv-round quorum-for-def dest: empty-not-quorum*)

done

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

9.3 Refinement

definition *obsv-ref-rel* **where**

$\text{obsv-ref-rel} \equiv \{(sa, sc).$
 $\quad sa = v\text{-state.truncate } sc$
 $\}$

lemma *obsv-round-refines*:

$\{\text{obsv-ref-rel} \cap \text{UNIV} \times \text{OV-inv1}\} \text{ sv-round } r \ S \ v \ \text{dec-f}, \text{ obsv-round } r \ S \ v \ \text{dec-f}$
 $\text{Ob } \{> \text{obsv-ref-rel}\}$

apply(*clarsimp simp add: PO-rhoare-defs sv-round-def obsv-round-def safe-def*
obsv-ref-rel-def

$v\text{-state.truncate-def obs-safe-def quorum-for-def OV-inv1-def}$)

by (*metis UNIV-I UNIV-quorum option.distinct(1) option.inject*)

lemma *Observable-Refines*:

$\text{PO-refines } (\text{obsv-ref-rel} \cap \text{UNIV} \times \text{OV-inv1}) \text{ sv-TS } \text{obsv-TS}$

proof(*rule refine-using-invariants*)

show $\text{init obsv-TS} \subseteq \text{obsv-ref-rel}$ “*init sv-TS*

by(*auto simp add: PO-refines-def sv-TS-defs obsv-TS-defs obsv-ref-rel-def*

$v\text{-state.truncate-def}$)

next

show $\{\text{obsv-ref-rel} \cap \text{UNIV} \times \text{OV-inv1}\} \text{ trans } \text{sv-TS}, \text{ trans } \text{obsv-TS} \{> \text{obsv-ref-rel}\}$

by(*auto simp add: PO-refines-def sv-TS-defs obsv-TS-defs intro!*

$\text{obsv-round-refines relhoare-refl}$)

qed(*auto intro: OV-inv1-inductive del: subsetI*)

9.4 Additional invariants

definition *OV-inv2* **where**

$\text{OV-inv2} = \{s. \forall r \geq \text{next-round } s. \text{obs } s \ r = \text{Map.empty} \}$

lemmas $\text{OV-inv2I} = \text{OV-inv2-def} \ [\text{THEN } \text{setc-def-to-intro}, \text{rule-format}]$

lemmas $\text{OV-inv2E} \ [\text{elim}] = \text{OV-inv2-def} \ [\text{THEN } \text{setc-def-to-elim}, \text{rule-format}]$

lemmas $\text{OV-inv2D} = \text{OV-inv2-def} \ [\text{THEN } \text{setc-def-to-dest}, \text{rule-format}]$

definition *OV-inv3* **where**

$$OV\text{-}inv3 = \{s. \forall r p v. \text{obs } s r p = \text{Some } v \longrightarrow \text{obs-safe } r s v\}$$

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**

$$OV\text{-}inv4 = \{s. \forall r p q v w. \text{obs } s r p = \text{Some } v \wedge \text{obs } s r q = \text{Some } w \longrightarrow w = v\}$$

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*:

$$\text{init } \text{obsv-TS} \subseteq OV\text{-}inv2$$

$$\{OV\text{-}inv2\} \text{ trans } \text{obsv-TS} \{> OV\text{-}inv2\}$$

by(*auto simp add: PO-hoare-defs OV-inv2-def obsv-TS-defs obsv-round-def const-map-is-Some*)

lemma *SVinv3-inductive*:

$$\text{init } \text{obsv-TS} \subseteq SV\text{-}inv3$$

$$\{SV\text{-}inv3\} \text{ trans } \text{obsv-TS} \{> SV\text{-}inv3\}$$

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*:

$$\{OV\text{-}inv3 \cap OV\text{-}inv2\} \text{ obsv-round } r S v D Ob \{> OV\text{-}inv3\}$$

proof(*clarsimp simp add: PO-hoare-defs intro!: OV-inv3I*)

fix *s s' r-w p w*

assume *Assms*:

$$\text{obs } s' r\text{-}w p = \text{Some } w$$

$$s \in OV\text{-}inv3$$

$$(s, s') \in \text{obsv-round } r S v D Ob$$

$$s \in OV\text{-}inv2$$

hence *s' ∈ OV-inv2*

by(*force simp add: obsv-TS-defs intro: OV-inv2-inductive(2)[THEN hoareD, OF ⟨s ∈ OV-inv2⟩*])

hence $r-w \leq \text{next-round } s'$ **using** *Assms*
by(*auto simp add: OV-inv2-def intro!: leI*)
hence $r-w \leq \text{next-round } s$ **using** *Assms*
by(*auto simp add: obsv-round-def le-Suc-eq*)
show *obs-safe* $r-w$ s' w
proof(*cases* $r-w = \text{next-round } s$)
case *True*
thus *?thesis* **using** *Assms*
by(*auto simp add: obsv-round-def const-map-is-Some obs-safe-def*)
next
case *False*
hence $r-w < \text{next-round } s$ **using** $r-w \leq$
by(*metis less-le*)
moreover **have** $\forall r'. r' \neq \text{next-round } s \longrightarrow \text{obs } s' r' = \text{obs } s r'$ **using** *Assms(3)*
by(*auto simp add: obsv-round-def*)
ultimately **have**
 $\forall r' \leq r-w. \text{obs } s' r' = \text{obs } s r'$
by(*auto*)
thus *?thesis* **using** *Assms*
by(*auto simp add: OV-inv3-def obs-safe-def*)
qed
qed

lemma *OV-inv3-inductive*:
 $\text{init } \text{obsv-TS} \subseteq \text{OV-inv3}$
 $\{\text{OV-inv3} \cap \text{OV-inv2}\} \text{ trans } \text{obsv-TS} \{> \text{OV-inv3}\}$
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*:
 $\text{init } \text{obsv-TS} \subseteq \text{OV-inv4}$
 $\{\text{OV-inv4}\} \text{ trans } \text{obsv-TS} \{> \text{OV-inv4}\}$
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-obs-safe obs-f v  $\equiv \exists p. \text{obs-f } p \in \{\text{None}, \text{Some } v\}$ 

```

```

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

```

```

definition olv-init where
  olv-init =  $\{ \langle \text{next-round} = 0, \text{decisions} = \text{Map.empty}, \text{last-obs} = \text{Map.empty}$ 
   $\rangle \}$ 

```

```

definition olv-trans :: (opt-obsv-state  $\times$  opt-obsv-state) set where

```

$olv-trans = (\bigcup r S v D Ob. olv-round r S v D Ob) \cup Id$

definition *olv-TS* :: *opt-obsv-state TS* **where**
olv-TS = $\langle init = olv-init, trans = olv-trans \rangle$

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

10.2 Refinement

definition *olv-ref-rel* **where**

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

lemma *OV-inv2-finite-map-graph*:

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

lemma *OV-inv2-finite-obs-set*:

$s \in OV-inv2 \implies finite\ (vote-set\ (obsv-state.obs\ s)\ Q)$
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*:

$\{olv-ref-rel \cap (OV-inv2 \cap OV-inv3 \cap OV-inv4) \times UNIV\} obsv-round\ r\ S\ v\ D$
 $Ob, olv-round\ r\ S\ v\ D\ Ob \{>olv-ref-rel\}$

proof(*clarsimp simp add: PO-rhoare-defs*)

fix *sa* :: *obsv-state* **and** *sc sc'*

assume

ainv: $sa \in OV-inv2\ sa \in OV-inv3\ sa \in OV-inv4$
and *step*: $(sc, sc') \in olv-round\ r\ S\ v\ D\ Ob$
and *R*: $(sa, sc) \in olv-ref-rel$

— Abstract guard.

have $S \neq \{\}$ \longrightarrow *obs-safe* (*v-state.next-round sa*) *sa v*

proof(*rule impI*, *rule ccontr*)

assume *S-nonempty*: $S \neq \{\}$ **and** *no-Q*: \neg *obs-safe* (*v-state.next-round sa*) *sa v*

from *no-Q* **obtain** *r-w w* **where**

r-w: $r-w < v\text{-state.next-round sa}$

and *all-obs*: $\forall p. \text{obsv-state.obs sa } r-w \ p = \text{Some } w$

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-option snd* \circ *process-mru* (*obsv-state.obs sa*)) $p \neq \text{Some } w$

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*) $\{p\} \neq \{\}$

by(*auto simp add: vote-set-def*)

then obtain *r-w' w'* **where** *w'*: *process-mru* (*obsv-state.obs sa*) $p = \text{Some } (r-w', w')$

by (*simp add: process-mru-def mru-of-set-def*)

 (*metis option-Max-by-def surj-pair*)

hence *max*: $(r-w', w') = \text{Max-by fst } (\text{vote-set } (\text{obsv-state.obs sa}) \{p\})$

by(*auto simp add: process-mru-def mru-of-set-def option-Max-by-def*)

hence *w'-obs*: $(r-w', w') \in \text{vote-set } (\text{obsv-state.obs sa}) \{p\}$

using *Max-by-in[OF OV-inv2-finite-obs-set[OF ainv(1), of {p}] nempty]*

by *fastforce*

have $r-w \leq r-w'$ **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 $w' \neq w$ **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

$(\text{map-option } \text{snd} \circ \text{process-mru } (\text{obsv-state.obs } sa)) \text{ ++ } \text{const-map } v \text{ Ob} =$
 $\text{map-option } \text{snd} \circ \text{process-mru } ((\text{obsv-state.obs } sa)(v\text{-state.next-round } sa :=$
 $\text{const-map } v \text{ Ob}))$

proof—

from $\langle sa \in \text{OV-inv2} \rangle [\text{THEN OV-inv2D}]$

have *empty*: $\forall r' \geq v\text{-state.next-round } sa. \text{obsv-state.obs } sa \ r' = \text{Map.empty}$

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 $\exists sa'. (sa, sa') \in \text{obsv-round } r \ S \ v \ D \ \text{Ob} \wedge (sa', sc') \in \text{olv-ref-rel}$

using *R step*

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

qed

lemma *OLV-Refines*:

PO-refines $(\text{olv-ref-rel} \cap (\text{OV-inv2} \cap \text{OV-inv3} \cap \text{OV-inv4}) \times \text{UNIV}) \text{ obsv-TS}$
 olv-TS

proof(*rule refine-using-invariants*)

show *init olv-TS* $\subseteq \text{olv-ref-rel}$ “*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

$\{\text{olv-ref-rel} \cap (\text{OV-inv2} \cap \text{OV-inv3} \cap \text{OV-inv4}) \times \text{UNIV}\} \text{TS.trans } \text{obsv-TS},$
 $\text{TS.trans } \text{olv-TS} \ \{> \ \text{olv-ref-rel}\}$

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  $\Rightarrow$  val option

```

```

context mono-quorum
begin

```

```

definition tso-round0
  :: round  $\Rightarrow$  process set  $\Rightarrow$  val  $\Rightarrow$  (tso-state  $\times$  tso-state)set
  where
    tso-round0 r S v  $\equiv$  {(s, s')}.
    — guards
    r = next-round s
     $\wedge$  two-step r = 0
     $\wedge$  (S  $\neq$  {})  $\longrightarrow$  opt-obs-safe (last-obs s) v
    — actions
     $\wedge$  s' = s(
      next-round := Suc r
      , r-votes := const-map v S
    )
  }

```

definition *obs-guard* :: (process, val)map \Rightarrow (process, val)map \Rightarrow bool **where**
obs-guard *r-obs* *r-v* $\equiv \forall p.$
 $(\forall v. r\text{-obs } p = \text{Some } v \longrightarrow (\exists q. r\text{-v } q = \text{Some } v))$
 $\wedge (\text{dom } r\text{-v} \in \text{Quorum} \longrightarrow (\exists q \in \text{dom } r\text{-v}. r\text{-obs } p = r\text{-v } q))$

definition *tso-round1*
:: round \Rightarrow (process, val)map \Rightarrow (process, val)map \Rightarrow (tso-state \times tso-state)set
where
tso-round1 *r* *r-decisions* *r-obs* $\equiv \{(s, s').$
— guards
r = *next-round* *s*
 \wedge *two-step* *r* = 1
 \wedge *d-guard* *r-decisions* (*r-votes* *s*)
 \wedge *obs-guard* *r-obs* (*r-votes* *s*)
— actions
 \wedge *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 \times tso-state) set **where**
tso-trans = ($\bigcup r S v. \text{tso-round0 } r S v$) \cup ($\bigcup r d\text{-f } o\text{-f}. \text{tso-round1 } r d\text{-f } o\text{-f}$) \cup *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 \times tso-state)set **where**
basic-rel = {(*sa*, *sc*).
 next-round *sa* = *two-phase* (*next-round* *sc*)

$$\begin{aligned} & \wedge \text{last-obs } sc = \text{last-obs } sa \\ & \wedge \text{decisions } sc = \text{decisions } sa \\ & \} \end{aligned}$$

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*:

$$(\text{const-map } v \ S = \text{const-map } v' \ S') = (S = S' \wedge (S = \{\} \vee v = v'))$$

apply(*simp add: const-map-def restrict-map-def*)

by (*metis equals0D option.distinct(2) option.inject subsetI subset-antisym*)

lemma *rhoare-skipI*:

$$\llbracket \bigwedge sa \ sc \ sc'. \llbracket (sa, sc) \in Pre; (sc, sc') \in Tc \rrbracket \implies (sa, sc') \in Post \rrbracket \implies \{Pre\}$$

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*:

$\{tso\text{-ref-rel}\} \cup r\ S\ v\ dec\text{-}f\ Ob.\ olv\text{-}round\ r\ S\ v\ dec\text{-}f\ Ob,\ tso\text{-}round1\ r\ dec\text{-}f\ o\text{-}f$
 $\{>tso\text{-ref-rel}\}$

proof(*clarsimp simp add: PO-rhoare-defs*)

fix *sa sc and sc'*

assume

R: $(sa, sc) \in tso\text{-ref-rel}$

and *step1*: $(sc, sc') \in tso\text{-round1}\ r\ dec\text{-}f\ o\text{-}f$

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: $(sa, sc0) \in step0\text{-rel}$ **and** *step0*: $(sc0, sc) \in tso\text{-round0}\ r0\ S0\ v0$

using *R*

by(*auto simp add: tso-ref-rel-def*)

from *step-r r-next R* **obtain** *S v* **where**

v: *r-votes* $sc = const\text{-}map\ v\ S$

and *safe*: $S \neq \{\}$ \longrightarrow *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** $sa' = sa \langle$

next-round := *Suc* (*next-round* sa)

, *decisions* := *decisions* $sa\ ++\ dec\text{-}f$

, *last-obs* := *last-obs* $sa\ ++\ const\text{-}map\ v\ (dom\ o\text{-}f)$

\rangle

have $S \in Quorum \longrightarrow dom\ o\text{-}f = UNIV$ **using** *step1 v*

by(*auto simp add: tso-round1-def obs-guard-def const-map-def*)

moreover **have** $o\text{-}f \neq Map.empty \longrightarrow S \neq \{\}$ **using** *step1 v*

by(*auto simp add: tso-round1-def obs-guard-def dom-const-map*)

ultimately **have**

abs-step:

$(sa, sa') \in olv\text{-}round\ (next\text{-}round\ sa)\ S\ v\ dec\text{-}f\ (dom\ o\text{-}f)$ **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** $post\text{-}rel: (sa', sc') \in tso\text{-}ref\text{-}rel$ **using** R $step1$
apply($clarsimp$ $simp$ $add: tso\text{-}round0\text{-}def$ $tso\text{-}round1\text{-}def$
 $step0\text{-}rel\text{-}def$ $basic\text{-}rel\text{-}def$ $sa'\text{-}def$ $tso\text{-}ref\text{-}rel\text{-}def$ $two\text{-}step\text{-}phase\text{-}Suc$ $o\text{-}def$
 $const\text{-}map\text{-}is\text{-}Some$ $const\text{-}map\text{-}is\text{-}None$ $const\text{-}map\text{-}equality$ $obs\text{-}guard\text{-}def$
 $intro!$: $arg\text{-}cong2$ [**where** $f=map\text{-}add$, OF $refl$])
apply($auto$ $simp$ $add: const\text{-}map\text{-}def$ $restrict\text{-}map\text{-}def$ $intro!$: ext)
done

from $abs\text{-}step$ $post\text{-}rel$ **show**
 $\exists sa'. (\exists r' S' w dec\text{-}f' Ob'. (sa, sa') \in olv\text{-}round$ $r' S' w dec\text{-}f' Ob') \wedge (sa', sc')$
 $\in tso\text{-}ref\text{-}rel$
by $blast$
qed

lemma $TS\text{-}Observing\text{-}Refines$:
 $PO\text{-}refines$ $tso\text{-}ref\text{-}rel$ $olv\text{-}TS$ $tso\text{-}TS$
apply($auto$ $simp$ $add: PO\text{-}refines\text{-}def$ $olv\text{-}TS\text{-}defs$ $tso\text{-}TS\text{-}defs$
 $intro!$: $tso\text{-}round0\text{-}refines$ $tso\text{-}round1\text{-}refines$)
apply($auto$ $simp$ $add: tso\text{-}ref\text{-}rel\text{-}def$ $step0\text{-}rel\text{-}def$ $basic\text{-}rel\text{-}def$ $tso\text{-}init\text{-}def$ $quorum\text{-}for\text{-}def$
 $dest: empty\text{-}not\text{-}quorum$)
done

11.3 Invariants

definition $TSO\text{-}inv1$ **where**

$$\begin{aligned}
TSO\text{-}inv1 = \{ & s. two\text{-}step (next\text{-}round\ s) = Suc\ 0 \longrightarrow \\
& (\exists v. \forall p\ w. r\text{-}votes\ s\ p = Some\ w \longrightarrow w = v) \}
\end{aligned}$$

lemmas $TSO\text{-}inv1I = TSO\text{-}inv1\text{-}def$ [$THEN$ $setc\text{-}def\text{-}to\text{-}intro$, $rule\text{-}format$]

lemmas $TSO\text{-}inv1E$ [$elim$] = $TSO\text{-}inv1\text{-}def$ [$THEN$ $setc\text{-}def\text{-}to\text{-}elim$, $rule\text{-}format$]

lemmas $TSO\text{-}inv1D = TSO\text{-}inv1\text{-}def$ [$THEN$ $setc\text{-}def\text{-}to\text{-}dest$, $rule\text{-}format$]

definition $TSO\text{-}inv2$ **where**

$$\begin{aligned}
TSO\text{-}inv2 = \{ & s. two\text{-}step (next\text{-}round\ s) = Suc\ 0 \longrightarrow \\
& (\forall p\ v. (r\text{-}votes\ s\ p = Some\ v \longrightarrow (\exists q. last\text{-}obs\ s\ q \in \{None, Some\ v\}))) \}
\end{aligned}$$

lemmas $TSO\text{-}inv2I = TSO\text{-}inv2\text{-}def$ [$THEN$ $setc\text{-}def\text{-}to\text{-}intro$, $rule\text{-}format$]

lemmas $TSO\text{-}inv2E$ [$elim$] = $TSO\text{-}inv2\text{-}def$ [$THEN$ $setc\text{-}def\text{-}to\text{-}elim$, $rule\text{-}format$]

lemmas $TSO\text{-}inv2D = TSO\text{-}inv2\text{-}def$ [$THEN$ $setc\text{-}def\text{-}to\text{-}dest$, $rule\text{-}format$]

11.3.1 Proofs of invariants

lemma *TSO-inv1-inductive:*

init tso-TS \subseteq 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 \subseteq TSO-inv1

by(*intro inv-rule-basic TSO-inv1-inductive*)

lemma *TSO-inv2-inductive:*

init tso-TS \subseteq 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 \subseteq 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 \equiv 2*

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

definition *step* **where** $step (r::nat) \equiv r \text{ mod } nSteps$

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

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

definition *smallestValRcvd* **where**
 $smallestValRcvd (msgs::process \rightarrow ('val::linorder) msg) \equiv$

$$\text{Min } \{v. \exists q. \text{msgs } q = \text{Some } (\text{Val } v)\}$$

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**

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

definition *next0* **where**

$$\begin{aligned} \text{next0 } r \ p \ st \ (\text{msgs}::\text{process} \rightarrow ('val::\text{linorder}) \ \text{msg}) \ st' \equiv \\ & (\exists v. (\forall q \in \text{msgRcvd } \text{msgs}. \text{msgs } q = \text{Some } (\text{Val } v)) \\ & \quad \wedge \ st' = st \ (\text{agreed-vote} := \text{Some } v, \text{last-obs} := \text{smallestValRcvd } \text{msgs} \)) \\ & \vee \neg(\exists v. \forall q \in \text{msgRcvd } \text{msgs}. \text{msgs } q = \text{Some } (\text{Val } v)) \\ & \quad \wedge \ st' = st \ (\text{last-obs} := \text{smallestValRcvd } \text{msgs} \) \end{aligned}$$

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

definition *send1* **where**

$$\text{send1 } r \ p \ q \ st \equiv \text{ValVote } (\text{last-obs } st) \ (\text{agreed-vote } st)$$

definition *valVoteRcvd* **where**

$$\begin{aligned} & \text{— processes from which values and votes were received} \\ \text{valVoteRcvd } (\text{msgs}::\text{process} \rightarrow 'val \ \text{msg}) \equiv \\ & \{q . \exists z \ v. \text{msgs } q = \text{Some } (\text{ValVote } z \ v)\} \end{aligned}$$

definition *smallestValNoVoteRcvd* **where**

$$\begin{aligned} \text{smallestValNoVoteRcvd } (\text{msgs}::\text{process} \rightarrow ('val::\text{linorder}) \ \text{msg}) \equiv \\ \text{Min } \{v. \exists q. \text{msgs } q = \text{Some } (\text{ValVote } v \ \text{None})\} \end{aligned}$$

definition *someVoteRcvd* **where**

$$\begin{aligned} & \text{— set of processes from which some vote was received} \\ \text{someVoteRcvd } (\text{msgs}::\text{process} \rightarrow 'val \ \text{msg}) \equiv \\ & \{q . q \in \text{msgRcvd } \text{msgs} \wedge \text{isValVote } (\text{the } (\text{msgs } q)) \wedge \text{getvote } (\text{the } (\text{msgs } q)) \neq \\ & \text{None } \} \end{aligned}$$

definition *identicalVoteRcvd* **where**

$$\begin{aligned} \text{identicalVoteRcvd } (\text{msgs}::\text{process} \rightarrow 'val \ \text{msg}) \ v \equiv \\ \forall q \in \text{msgRcvd } \text{msgs}. \text{isValVote } (\text{the } (\text{msgs } q)) \wedge \text{getvote } (\text{the } (\text{msgs } q)) = \text{Some } \\ v \end{aligned}$$

definition *x-update* **where**

$$\begin{aligned} x\text{-update } st \text{ msgs } st' &\equiv \\ &(\exists q \in \text{someVoteRcvd } msgs . \text{last-obs } st' = \text{the } (\text{getvote } (\text{the } (msgs \ q)))) \\ \vee \text{someVoteRcvd } msgs = \{\} &\wedge \text{last-obs } st' = \text{smallestValNoVoteRcvd } msgs \end{aligned}$$

definition *dec-update* **where**

$$\begin{aligned} dec\text{-update } st \text{ msgs } st' &\equiv \\ &(\exists v . \text{identicalVoteRcvd } msgs \ v \wedge \text{decide } st' = \text{Some } v) \\ \vee \neg(\exists v . \text{identicalVoteRcvd } msgs \ v) &\wedge \text{decide } st' = \text{decide } st \end{aligned}$$

definition *next1* **where**

$$\begin{aligned} next1 \ r \ p \ st \ msgs \ st' &\equiv \\ &x\text{-update } st \ msgs \ st' \\ &\wedge dec\text{-update } st \ msgs \ st' \\ &\wedge \text{agreed-vote } st' = \text{None} \end{aligned}$$

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

definition *UV-sendMsg* **where**

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

definition *UV-nextState* **where**

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

definition (**in** *quorum-process*) *UV-commPerRd* **where**

$$UV\text{-commPerRd } HOs \equiv \forall p . HOs \ p \in \text{Quorum}$$

definition *UV-commGlobal* **where**

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

12.2 The *Uniform Voting* Heard-Of machine

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

definition (**in** *quorum-process*) *UV-HOMachine* **where**

$$UV\text{-HOMachine} = \{\}$$

```

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

```

abbreviation (in *quorum-process*)

```
UV-M  $\equiv$  (UV-HOMachine::(process, 'val::linorder pstate, 'val msg) HOMachine)
```

end

12.3 Proofs

type-synonym *uv-TS-state* = ($\text{nat} \times (\text{process} \Rightarrow (\text{val } pstate))$)

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  $\Rightarrow$  process HO)  $\Rightarrow$  (round  $\Rightarrow$  process HO)  $\Rightarrow$  (round  $\Rightarrow$  process)  $\Rightarrow$ 
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 $rHO = nat \Rightarrow process\ HO$

definition $UV-trans-step$

where

$$UV-trans-step\ HOs\ SHOs\ next-f\ snd-f\ stp \equiv \bigcup r\ \mu. \\ \{((r, cfg), (Suc\ r, cfg')) \mid cfg\ cfg'.\ step\ r = stp \wedge (\forall p. \\ \mu\ p \in get-msgs\ (snd-f\ r)\ cfg\ (HOs\ r)\ (SHOs\ r)\ p \\ \wedge next-f\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p) \\)\}$$

lemma $step-less-D$:

$$0 < step\ r \implies step\ r = Suc\ 0$$

by($auto\ simp\ add$: $step-def$)

lemma $UV-trans$:

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

proof

show $CSHO-trans-alt\ UV-sendMsg\ (\lambda r\ p\ st\ msgs\ crd.\ UV-nextState\ r\ p\ st\ msgs)$
 $HOs\ SHOs\ crds$

$$\subseteq UV-trans-step\ HOs\ SHOs\ next0\ send0\ 0 \cup UV-trans-step\ HOs\ SHOs\ next1 \\ send1\ 1$$

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 $UV-trans-step\ HOs\ SHOs\ next0\ send0\ 0 \cup \\ UV-trans-step\ HOs\ SHOs\ next1\ send1\ 1$

$$\subseteq CSHO-trans-alt\ UV-sendMsg$$

$$(\lambda r\ p\ st\ msgs\ crd.\ UV-nextState\ r\ p\ st\ msgs)\ HOs\ SHOs\ crds$$

by($force\ simp\ add$: $CSHO-trans-alt-def\ UV-sendMsg-def\ UV-nextState-def\ UV-trans-step-def$)

qed

12.3.1 Invariants

definition $UV-inv1$

$:: uv\text{-}TS\text{-}state\ set$
where
 $UV\text{-}inv1 = \{(r, s).$
 $\quad two\text{-}step\ r = 0 \longrightarrow (\forall p. agreed\text{-}vote\ (s\ p) = None)$
 $\}$

lemmas $UV\text{-}inv1I = UV\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}intro,\ rule\text{-}format]$
lemmas $UV\text{-}inv1E\ [elim] = UV\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}elim,\ rule\text{-}format]$
lemmas $UV\text{-}inv1D = UV\text{-}inv1\text{-}def\ [THEN\ setc\text{-}def\text{-}to\text{-}dest,\ rule\text{-}format]$

lemma $UV\text{-}inv1\text{-}inductive:$
 $init\ (UV\text{-}TS\ HOs\ SHOs\ crds) \subseteq UV\text{-}inv1$
 $\{UV\text{-}inv1\}\ TS.trans\ (UV\text{-}TS\ HOs\ SHOs\ crds) \{>\ UV\text{-}inv1\}$
by($auto\ simp\ add: UV\text{-}inv1\text{-}def\ UV\text{-}TS\text{-}defs\ CHO\text{-}trans\text{-}alt\ UV\text{-}trans\ PO\text{-}hoare\text{-}def$
 $UV\text{-}HOMachine\text{-}def\ CHOAlgorithm.truncate\text{-}def\ UV\text{-}trans\text{-}step\text{-}def$
 $all\text{-}conj\text{-}distrib\ two\text{-}step\text{-}phase\text{-}Suc\ two\text{-}step\text{-}step\ next1\text{-}def$)

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

12.3.2 Refinement

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

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 $global: UV\text{-}commGlobal\ HOs$
and $per\text{-}rd: \forall r. UV\text{-}commPerRd\ (HOs\ r)$

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: $\forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ } \text{cfg } (\text{HOs } r) (\text{HOs } r) p$
and *step*: $\forall p. \text{next0 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$
and *inv*: $(r, \text{cfg}) \in \text{UV-inv1}$
and *step-r*: *two-step r = 0*
shows
 $\text{agreed-vote } (\text{cfg}' p) = \text{Some } v \longleftrightarrow (\forall q \in \text{HOs } r p. \text{last-obs } (\text{cfg } q) = v)$
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: $\forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ } \text{cfg } (\text{HOs } r) (\text{HOs } r) p$
and *step*: $\forall p. \text{next0 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$
and *inv*: $(r, \text{cfg}) \in \text{UV-inv1}$
and *step-r*: *two-step r = 0*
obtains *v* **where** $\forall p w. \text{agreed-vote } (\text{cfg}' p) = \text{Some } w \longrightarrow w = v$
proof(*cases* $\exists p v. \text{agreed-vote } (\text{cfg}' p) = \text{Some } v$)
case *True*
assume *asm*: $\bigwedge v. \forall p w. \text{agreed-vote } (\text{cfg}' p) = \text{Some } w \longrightarrow w = v \implies \text{thesis}$
from *True* **obtain** *p v* **where** $\text{agreed-vote } (\text{cfg}' p) = \text{Some } v$ **by** *auto*

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

from $\text{asm}[OF \text{ this}]$ **show** $?thesis$.
qed(auto)

lemma $x\text{-origin1}$:

assumes

$\text{send}: \forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ cfg } (\text{HOs } r) (\text{HOs } r) p$

and $\text{step}: \forall p. \text{next0 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$

and $\text{step-r}: \text{two-step } r = 0$

and $\text{last-obs}: \text{last-obs } (\text{cfg}' p) = v$

shows

$\exists q. \text{last-obs } (\text{cfg } q) = v$

proof–

have $\text{smallestValRcvd } (\mu p) \in \{v. \exists q. \mu p q = \text{Some } (\text{Val } v)\}$ (**is** smallestValRcvd
 $?msgs \in ?vals$)

unfolding $\text{smallestValRcvd-def}$

proof(rule Min-in)

have $?vals \subseteq \text{getval } ' ((\text{the } \circ ?msgs) ' (\text{HOs } r p))$

using $\text{send}[THEN \text{ spec}, \text{ where } x=p]$

by ($\text{auto simp: image-def get-msgs-benign restrict-map-def send0-def}$)

thus $\text{finite } ?vals$ **by** ($\text{auto simp: finite-subset}$)

next

from $\text{send}[THEN \text{ spec}, \text{ where } x=p]$

show $?vals \neq \{\}$ **using** $\text{HOs-nonempty}[of r p]$

by ($\text{auto simp: image-def get-msgs-benign restrict-map-def send0-def}$)

qed

hence $v \in ?vals$ **using** $\text{last-obs step}[THEN \text{ spec}, of p]$

by($\text{auto simp add: next0-def all-conj-distrib}$)

thus $?thesis$ **using** $\text{send}[THEN \text{ spec}, of p]$

by($\text{auto simp add: get-msgs-benign send0-def restrict-map-def}$)

qed

lemma step0-ref :

$\{\text{ref-rel} \cap \text{UNIV} \times \text{UV-inv1}\} \cup r S v. \text{tso-round0 } r S v,$
 $\text{UV-trans-step HOs HOs next0 send0 0 } \{> \text{ref-rel}\}$

proof($\text{clarsimp simp add: PO-rhoare-defs UV-trans-step-def two-step-step all-conj-distrib}$)

fix $sa r \text{cfg } \mu \text{cfg}'$

assume
 $R: (sa, (r, cfg)) \in \text{ref-rel}$
and $\text{step-r}: \text{two-step } r = 0$
and $\text{send}: \forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ } cfg \text{ } (HOs \ r) \text{ } (HOs \ r) \text{ } p$
and $\text{step}: \forall p. \text{next0 } r \text{ } p \text{ } (cfg \text{ } p) \text{ } (\mu \text{ } p) \text{ } (cfg' \text{ } p)$
and $\text{inv}: (r, cfg) \in UV\text{-inv1}$

from R **have** $\text{next-r}: \text{next-round } sa = r$
by $(\text{simp add}: \text{ref-rel-def})$

from $HOs\text{-nonempty}$ **send** **have** $\forall p. \exists q. q \in \text{msgRcvd } (\mu \text{ } p)$
by $(\text{fastforce simp add}: \text{get-msgs-benign send0-def msgRcvd-def restrict-map-def})$
with step **have** $\text{same-dec}: \text{decide } o \text{ } cfg' = \text{decide } o \text{ } cfg$
apply $(\text{simp add}: \text{next0-def } o\text{-def})$
by $(\text{metis } pstate.\text{select-convs}(3) \text{ } pstate.\text{surjective } pstate.\text{update-convs}(1) \text{ } pstate.\text{update-convs}(2))$

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

note $x\text{-origin} = x\text{-origin1}[OF \text{ send step step-r}]$

define sa' **where** $sa' = sa \lfloor \text{next-round} := \text{Suc } r, r\text{-votes} := \text{const-map } v \text{ } S \rfloor$

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

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

moreover **have** $(sa', \text{Suc } r, cfg') \in \text{ref-rel}$ **using** $\text{step send } v \text{ } R \text{ } \text{same-dec step-r}$
 next-r
apply $(\text{clarsimp simp add}: \text{ref-rel-def } sa'\text{-def two-step-step two-step-phase-Suc}$
 $\text{vote-const-map})$

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

lemma *x-origin2*:

assumes
send: $\forall p. \mu\ p \in get_msgs\ (send1\ r)\ cfg\ (HOs\ r)\ (HOs\ r)\ p$
and *step*: $\forall p. next1\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p)$
and *step-r*: $two_step\ r = Suc\ 0$
and *last-obs*: $last_obs\ (cfg'\ p) = v$
shows
 $(\exists q. last_obs\ (cfg\ q) = v) \vee (\exists q. agreed_vote\ (cfg\ q) = Some\ v)$
proof(*cases* $\forall q \in HOs\ r\ p. \exists w. \mu\ p\ q = Some\ (ValVote\ w\ None)$)
case *True*
hence *empty*: $someVoteRcvd\ (\mu\ p) = \{\}$ **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* $(\mu\ p) \in \{v. \exists q. \mu\ p\ q = Some\ (ValVote\ v\ None)\}$

(is smallestValNoVoteRcvd ?msgs ∈ ?vals)
unfolding *smallestValNoVoteRcvd-def*
proof(*rule Min-in*)
have $?vals \subseteq getval\ '((the\ \circ\ ?msgs)\ '(HOs\ r\ p))$
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 $?vals \neq \{\}$
by (*auto simp: image-def get-msgs-benign restrict-map-def send1-def*)
qed
hence $v \in ?vals$ **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*) $\neq \{\}$ **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 $\exists q \in \text{someVoteRcvd } (\mu \text{ } p)$. *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'* $\equiv \{p. \text{decide } (cfg' \text{ } p) \neq \text{decide } (cfg \text{ } p) \}$

lemma *decide-origin*:

assumes

send: $\forall p. \mu \text{ } p \in \text{get-msgs } (send1 \text{ } r) \text{ } cfg \text{ } (HOs \text{ } r) \text{ } (HOs \text{ } r) \text{ } p$

and *step*: $\forall p. \text{next1 } r \text{ } p \text{ } (cfg \text{ } p) \text{ } (\mu \text{ } p) \text{ } (cfg' \text{ } p)$

and *step-r*: *two-step* *r* = *Suc 0*

shows

D *cfg* *cfg'* $\subseteq \{p. \exists v. \text{decide } (cfg' \text{ } p) = \text{Some } v \wedge (\forall q \in HOs \text{ } r \text{ } p. \text{agreed-vote } (cfg \text{ } q) = \text{Some } v)\}$

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*:

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

proof(*clarsimp simp add: PO-rhoare-defs UV-trans-step-def two-step-step all-conj-distrib*)

fix *sa* *r* *cfg* μ **and** *cfg'* $:: \text{process} \Rightarrow \text{val } pstate$

assume

R: (*sa*, (*r*, *cfg*)) $\in \text{ref-rel}$

and *step-r*: *two-step* *r* = *Suc 0*

and *send*: $\forall p. \mu \text{ } p \in \text{get-msgs } (send1 \text{ } r) \text{ } cfg \text{ } (HOs \text{ } r) \text{ } (HOs \text{ } r) \text{ } p$

```

and step:  $\forall p. \text{next1 } r \ p \ (cfg \ p) \ (\mu \ p) \ (cfg' \ p)$ 
and ainv:  $sa \in TSO\text{-inv1}$ 
and ainv2:  $sa \in TSO\text{-inv2}$ 

from R have next-r:  $\text{next-round } sa = r$ 
  by(simp add: ref-rel-def)

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

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

define dec-f where  $dec-f \ p = (\text{if } p \in D \text{ cfg } cfg' \text{ then } \text{decide } (cfg' \ p) \text{ else } \text{None})$ 
for  $p$ 

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

have d-guard:  $d\text{-guard } dec-f \ (\text{agreed-vote} \circ \text{cfg})$  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  $\text{dom } (\text{agreed-vote} \circ \text{cfg}) \in \text{Quorum} \longrightarrow Ob = \text{UNIV}$ 
proof(auto simp add: Ob-def)
  fix  $p$ 
  assume  $Q: \text{dom } (\text{agreed-vote} \circ \text{cfg}) \in \text{Quorum} \ (\text{is } ?Q \in \text{Quorum})$ 
  hence  $?Q \cap \text{HOs } r \ p \neq \{\}$  using per-rd[THEN spec, of r]
    by(auto simp add: UV-commPerRd-def dest: qintersect)
  hence  $\text{someVoteRcvd } (\mu \ p) \neq \{\}$  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 $\forall q \in \text{someVoteRcvd } (\mu \ p). \exists x'. \mu \ p \ q = \text{Some } (\text{ValVote } x'$
(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 \circ 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[^]) \in 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 $\forall p. ((\text{decide} \circ \text{cfg}) ++ \text{dec-f}) \ p = \text{decide } (\text{cfg}' \ p)$
proof
fix *p*
show $((\text{decide} \circ \text{cfg}) ++ \text{dec-f}) \ p = \text{decide } (\text{cfg}' \ p)$ **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 $\forall p. (\exists q. \text{o-f } q = \text{None} \wedge \text{opt-obsv-state.last-obs } sa \ q = \text{None}$
 $\vee (\text{opt-obsv-state.last-obs } sa ++ \text{o-f}) \ q = \text{Some } (\text{last-obs } (\text{cfg}' \ p)))$

```

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 ( $\exists q. o-f\ q = None \wedge opt-obsv-state.last-obs\ sa\ q = None$ 
     $\vee (opt-obsv-state.last-obs\ sa\ ++\ o-f)\ q = Some\ (last-obs\ (cfg'\ p))$ )
    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 ( $\exists q. o-f\ q = None \wedge opt-obsv-state.last-obs\ sa\ q = None$ 
     $\vee (opt-obsv-state.last-obs\ sa\ ++\ o-f)\ q = Some\ (last-obs\ (cfg'\ p))$ )
  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')  $\in$  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
   $\exists sa'. (\exists r\ d-f\ o-f. (sa, sa') \in tso-round1\ r\ d-f\ o-f) \wedge (sa', Suc\ r, cfg') \in ref-rel$ 
  by blast

```

qed

lemma *UV-Refines-votes:*

PO-refines (ref-rel \cap (TSO-inv1 \cap TSO-inv2) \times UV-inv1)
tso-TS (UV-TS HOs HOs crds)

proof(*rule refine-using-invariants*)

show *init (UV-TS HOs HOs crds) \subseteq 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 \cap (TSO-inv1 \cap TSO-inv2) \times 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 \Rightarrow process \Rightarrow 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 $\equiv \exists v. m = \text{Vote } v$

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

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

fun *getval* **where**
getval (Val z) = z

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

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

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

definition *next0* **where**
next0 r p st (msgs::process \rightarrow 'val msg) st' \equiv
 ($\exists v. (\forall q \in \text{msgRcvd } \text{msgs}. \text{msgs } q = \text{Some } (\text{Val } v))$
 $\wedge \text{st}' = \text{st } (\downarrow \text{vote } := \text{Some } v \downarrow)$)
 $\vee \neg(\exists v. \forall q \in \text{msgRcvd } \text{msgs}. \text{msgs } q = \text{Some } (\text{Val } v))$
 $\wedge \text{st}' = \text{st } (\downarrow \text{vote } := \text{None } \downarrow)$

definition *send1* **where**
send1 r p q st $\equiv \text{Vote } (\text{vote } st)$

definition *someVoteRcvd* **where**
 — set of processes from which some vote was received
someVoteRcvd (msgs :: process \rightarrow 'val msg) \equiv

$$\{ q . q \in \text{msgRcvd } \text{msgs} \wedge \text{isVote } (\text{the } (\text{msgs } q)) \wedge \text{getvote } (\text{the } (\text{msgs } q)) \neq \text{None} \}$$

definition *identicalVoteRcvd* **where**

$$\begin{aligned} \text{identicalVoteRcvd } (\text{msgs} :: \text{process} \rightarrow \text{'val } \text{msg}) \text{ } v \equiv \\ \forall q \in \text{msgRcvd } \text{msgs}. \text{isVote } (\text{the } (\text{msgs } q)) \wedge \text{getvote } (\text{the } (\text{msgs } q)) = \text{Some } v \end{aligned}$$

definition *x-update* **where**

$$\begin{aligned} \text{x-update } r \text{ } p \text{ } \text{msgs } \text{st}' \equiv \\ (\exists q \in \text{someVoteRcvd } \text{msgs} . \text{x } \text{st}' = \text{the } (\text{getvote } (\text{the } (\text{msgs } q)))) \\ \vee \text{someVoteRcvd } \text{msgs} = \{\} \wedge \text{x } \text{st}' = \text{coin } r \text{ } p \end{aligned}$$

definition *dec-update* **where**

$$\begin{aligned} \text{dec-update } \text{st } \text{msgs } \text{st}' \equiv \\ (\exists v. \text{identicalVoteRcvd } \text{msgs } v \wedge \text{decide } \text{st}' = \text{Some } v) \\ \vee \neg(\exists v. \text{identicalVoteRcvd } \text{msgs } v) \wedge \text{decide } \text{st}' = \text{decide } \text{st} \end{aligned}$$

definition *next1* **where**

$$\begin{aligned} \text{next1 } r \text{ } p \text{ } \text{st } \text{msgs } \text{st}' \equiv \\ \text{x-update } r \text{ } p \text{ } \text{msgs } \text{st}' \\ \wedge \text{dec-update } \text{st } \text{msgs } \text{st}' \\ \wedge \text{vote } \text{st}' = \text{None} \end{aligned}$$

definition *BenOr-sendMsg* **where**

$$\text{BenOr-sendMsg } (r :: \text{nat}) \equiv \text{if two-step } r = 0 \text{ then send0 } r \text{ else send1 } r$$

definition *BenOr-nextState* **where**

$$\text{BenOr-nextState } r \equiv \text{if two-step } r = 0 \text{ then next0 } r \text{ else next1 } r$$

13.1 The *Ben-Or* Heard-Of machine

definition (in *quorum-process*) *BenOr-commPerRd* **where**

$$\text{BenOr-commPerRd } \text{HOs} \equiv \forall p. \text{HOs } p \in \text{Quorum}$$

definition *BenOr-commGlobal* **where**

$$\begin{aligned} \text{BenOr-commGlobal } \text{HOs} \equiv \exists r. \text{two-step } r = 1 \\ \wedge (\forall p \text{ } q. \text{HOs } r \text{ } p = \text{HOs } r \text{ } q \wedge (\text{coin } r \text{ } p :: \text{val}) = \text{coin } r \text{ } q) \end{aligned}$$

definition (in *quorum-process*) *BenOr-HOMachine* **where**

```
BenOr-HOMachine = (  
  CinitState = ( $\lambda p\ st\ crd.$  BenOr-initState  $p\ st$ ),  
  sendMsg = BenOr-sendMsg,  
  CnextState = ( $\lambda r\ p\ st\ msgs\ crd\ st'.$  BenOr-nextState  $r\ p\ st\ msgs\ st'$ ),  
  HOcommPerRd = BenOr-commPerRd,  
  HOcommGlobal = BenOr-commGlobal  
)
```

abbreviation (in *quorum-process*)

```
BenOr-M  $\equiv$  (BenOr-HOMachine::(process, val pstate, val msg) HOMachine)
```

end

13.2 Proofs

type-synonym *ben-or-TS-state* = ($nat \times (process \Rightarrow (val\ pstate))$)

consts

```
val0 :: val
```

```
val1 :: val
```

Ben-Or works only on binary values.

axiomatization where

```
val-exhaust:  $v = val0 \vee v = val1$ 
```

```
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  $\Rightarrow$  process HO)  $\Rightarrow$  (round  $\Rightarrow$  process HO)  $\Rightarrow$  (round  $\Rightarrow$  process)  $\Rightarrow$   
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 $rHO = nat \Rightarrow process\ HO$

definition *BenOr-trans-step*

where

$BenOr-trans-step\ HOs\ SHOs\ next-f\ snd-f\ stp \equiv \bigcup r\ \mu.$
 $\{((r, cfg), (Suc\ r, cfg')) \mid cfg\ cfg'.\ two-step\ r = stp \wedge (\forall p.$
 $\mu\ p \in get-msgs\ (snd-f\ r)\ cfg\ (HOs\ r)\ (SHOs\ r)\ p$
 $\wedge\ next-f\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p)$
 $\})\}$

lemma *two-step-less-D*:

$0 < two-step\ r \implies two-step\ r = Suc\ 0$

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

lemma *BenOr-trans*:

$CSHO-trans-alt\ BenOr-sendMsg\ (\lambda r\ p\ st\ msgs\ crd\ st'.\ BenOr-nextState\ r\ p\ st\ msgs\ st')$ $HOs\ SHOs\ crds =$

$BenOr-trans-step\ HOs\ SHOs\ next0\ send0\ 0$

$\cup\ BenOr-trans-step\ HOs\ SHOs\ next1\ send1\ 1$

proof

show $CSHO-trans-alt\ BenOr-sendMsg\ (\lambda r\ p\ st\ msgs\ crd.\ BenOr-nextState\ r\ p\ st\ msgs)$ $HOs\ SHOs\ crds$

$\subseteq BenOr-trans-step\ HOs\ SHOs\ next0\ send0\ 0 \cup BenOr-trans-step\ HOs\ SHOs\ next1\ send1\ 1$

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 $BenOr-trans-step\ HOs\ SHOs\ next0\ send0\ 0 \cup$

$BenOr-trans-step\ HOs\ SHOs\ next1\ send1\ 1$

$\subseteq CSHO-trans-alt\ BenOr-sendMsg$

$(\lambda r\ p\ st\ msgs\ crd.\ BenOr-nextState\ r\ p\ st\ msgs)\ HOs\ SHOs\ crds$

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

$HOs :: nat \Rightarrow process \Rightarrow process\ set$

and $\rho :: nat \Rightarrow process \Rightarrow val\ pstate$

assumes $comm\text{-}global: BenOr\text{-}commGlobal\ HOs$

and $per\text{-}rd: \forall r. BenOr\text{-}commPerRd\ (HOs\ r)$

and $run: HORun\ BenOr\text{-}A\ \rho\ HOs$

begin

definition $no\text{-}vote\text{-}diff$ **where**

$no\text{-}vote\text{-}diff\ sc\ p \equiv vote\ (sc\ p) = None \longrightarrow$
 $(\exists q\ q'. x\ (sc\ q) \neq x\ (sc\ q'))$

definition $ref\text{-}rel :: (tso\text{-}state \times ben\text{-}or\text{-}TS\text{-}state)set$ **where**

$ref\text{-}rel \equiv \{(sa, (r, sc)).$

$r = next\text{-}round\ sa$

$\wedge (two\text{-}step\ r = 1 \longrightarrow r\text{-}votes\ sa = vote\ o\ sc)$

$\wedge (two\text{-}step\ r = 1 \longrightarrow (\forall p. no\text{-}vote\text{-}diff\ sc\ p))$

$\wedge (\forall p\ v. x\ (sc\ p) = v \longrightarrow (\exists q. last\text{-}obs\ sa\ q \in \{None, Some\ v\}))$

$\wedge decisions\ sa = decide\ o\ sc$

$\}$

lemma $HOs\text{-}intersect:$

$HOs\ r\ p \cap HOs\ r'\ q \neq \{\}$ **using** $per\text{-}rd$

apply($simp\ add: BenOr\text{-}commPerRd\text{-}def$)

apply($blast\ dest: qintersect$)

done

lemma $HOs\text{-}nonempty:$

$HOs\ r\ p \neq \{\}$

using $HOs\text{-}intersect$

by $blast$

lemma $vote\text{-}origin:$

assumes

send: $\forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ cfg } (\text{HOs } r) (\text{HOs } r) p$

and *step*: $\forall p. \text{next0 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$

and *step-r*: $\text{two-step } r = 0$

shows

$\text{vote } (\text{cfg}' p) = \text{Some } v \longleftrightarrow (\forall q \in \text{HOs } r p. x (\text{cfg } q) = v)$

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: $\forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ cfg } (\text{HOs } r) (\text{HOs } r) p$

and *step*: $\forall p. \text{next0 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$

and *step-r*: $\text{two-step } r = 0$

obtains *v* **where** $\forall p w. \text{vote } (\text{cfg}' p) = \text{Some } w \longrightarrow w = v$

proof(*cases* $\exists p v. \text{vote } (\text{cfg}' p) = \text{Some } v$)

case *True*

assume *asm*: $\bigwedge v. \forall p w. \text{vote } (\text{cfg}' p) = \text{Some } w \longrightarrow w = v \implies \text{thesis}$

from *True* **obtain** *p v* **where** $\text{vote } (\text{cfg}' p) = \text{Some } v$ **by** *auto*

hence $\forall p w. \text{vote } (\text{cfg}' p) = \text{Some } w \longrightarrow w = v$ (**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: $\forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ cfg } (\text{HOs } r) (\text{HOs } r) p$

and *step*: $\forall p. \text{next0 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$

and *step-r*: $\text{two-step } r = 0$

shows

$x (\text{cfg}' p) = x (\text{cfg } p)$

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: $\forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ cfg } (\text{HOs } r) (\text{HOs } r) p$

and *step*: $\forall p. \text{next0 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$

and *step-r*: $\text{two-step } r = 0$

shows

no-vote-diff *cfg'* *p*

unfolding *no-vote-diff-def*

proof

assume

vote (*cfg'* *p*) = *None*

hence $(\exists q q'. x (\text{cfg } q) \neq x (\text{cfg}' q'))$

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 $(\exists q q'. x (\text{cfg}' q) \neq x (\text{cfg}' q'))$

using *no-x-change*[*OF send step step-r*]

by(*simp*)

qed

lemma *step0-ref*:

$\{\text{ref-rel}\} \cup r S v. \text{tso-round0 } r S v,$

BenOr-trans-step HOs HOs next0 send0 0 $\{> \text{ref-rel}\}$

proof(*clarsimp simp add: PO-rhoare-defs BenOr-trans-step-def all-conj-distrib*)

fix *sa r cfg μ cfg'*

assume

R: $(sa, (r, \text{cfg})) \in \text{ref-rel}$

and *step-r*: $\text{two-step } r = 0$

and *send*: $\forall p. \mu p \in \text{get-msgs } (\text{send0 } r) \text{ cfg } (\text{HOs } r) (\text{HOs } r) p$

and *step*: $\forall p. \text{next0 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$

from *R* **have** *next-r*: $\text{next-round } sa = r$

by(*simp add: ref-rel-def*)

from *HOs-nonempty send* **have** $\forall p. \exists q. q \in \text{msgRcvd } (\mu p)$

by(*fastforce simp add: get-msgs-benign send0-def msgRcvd-def restrict-map-def*)

with *step* **have** *same-dec*: $\text{decide } o \text{ } cfg' = \text{decide } o \text{ } cfg$
apply(*simp add*: *next0-def o-def*)
by (*metis pstate.select-convs(3) pstate.surjective pstate.update-convs(2)*)

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

define *sa'* **where** $sa' = sa \lfloor \text{next-round} := \text{Suc } r, r\text{-votes} := \text{const-map } v \ S \rfloor$

have $\forall p. p \in S \longrightarrow \text{opt-obs-safe } (\text{last-obs } sa) \ v$
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 $(sa, sa') \in \text{tso-round0 } r \ S \ v$ **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** $(sa', \text{Suc } r, cfg') \in \text{ref-rel}$ **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
 $\exists sa'. (\exists r \ S \ v. (sa, sa') \in \text{tso-round0 } r \ S \ v) \wedge (sa', \text{Suc } r, cfg') \in \text{ref-rel}$
by *blast*

qed

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

lemma *decide-origin*:
assumes

send: $\forall p. \mu p \in \text{get-msgs } (\text{send1 } r) \text{ cfg } (\text{HOs } r) (\text{HOs } r) p$
and *step*: $\forall p. \text{next1 } r p (\text{cfg } p) (\mu p) (\text{cfg}' p)$
and *step-r*: *two-step* $r = \text{Suc } 0$
shows
 $D \text{ cfg } \text{cfg}' \subseteq \{p. \exists v. \text{decide } (\text{cfg}' p) = \text{Some } v \wedge (\forall q \in \text{HOs } r p. \text{vote } (\text{cfg } q) = \text{Some } v)\}$
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*:

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

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

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

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

define *dec-f* **where** $\text{dec-f } p = (\text{if } p \in D \text{ cfg } \text{cfg}' \text{ then } \text{decide } (\text{cfg}' p) \text{ else } \text{None})$
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  $d-guard: d-guard\ dec-f\ (vote\ \circ\ cfg)$  using  $per-rd[THEN\ spec, of\ r]$ 
  by( $fastforce\ simp\ add: d-guard-def\ locked-in-vf-def\ quorum-for-def\ dec-f-def$ 
     $BenOr-commPerRd-def\ dest!: decide-origin[OF\ send\ step\ step-r, THEN\ sub-$ 
 $setD]$ )

have  $dom\ (vote\ \circ\ cfg) \in Quorum \longrightarrow Ob = UNIV$ 
proof( $auto\ simp\ add: Ob-def$ )
  fix  $p$ 
  assume  $Q: dom\ (vote\ \circ\ cfg) \in Quorum$  (is  $?Q \in Quorum$ )
  hence  $?Q \cap HOs\ r\ p \neq \{\}$  using  $per-rd[THEN\ spec, of\ r]$ 
    by( $auto\ simp\ add: BenOr-commPerRd-def\ dest: qintersect$ )
  hence  $someVoteRcvd\ (\mu\ p) \neq \{\}$  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  $\forall q \in someVoteRcvd\ (\mu\ p). \exists x'. \mu\ p\ q = Some\ (Vote\ (Some$ 
 $v))$ 
    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  $obs-guard: obs-guard\ o-f\ (vote\ \circ\ cfg)$ 
  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 ∀ p. (∃ q. o-f q = None ∧ opt-obsv-state.last-obs sa q = None
  ∨ (opt-obsv-state.last-obs sa ++ o-f) q = Some (x (cfg' p)))
proof(intro allI impI, cases S ∈ Quorum)
  fix p
  case True
  hence x (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 (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 ≠ 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) ≠ 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  $\forall v. \exists q. \text{opt-obsv-state.last-obs sa } q \in \{\text{None}, \text{Some } v\}$  using  $R$  step-r
  apply(auto simp add: ref-rel-def)
  by (metis (poly-guards-query) val-exhaust)

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

have post-rel:
  ( $sa', \text{Suc } r, \text{cfg}'$ )  $\in$  ref-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
   $\exists sa'. (\exists r \text{ d-f o-f. } (sa, sa') \in \text{tso-round1 } r \text{ d-f o-f}) \wedge (sa', \text{Suc } r, \text{cfg}'$ )  $\in$  ref-rel
  by blast
qed

lemma BenOr-Refines-Two-Step-Obs:
  PO-refines (ref-rel  $\cap$  (TSO-inv1  $\cap$  TSO-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-rel  $\cap$  (TSO-inv1  $\cap$  TSO-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 CHOAL-
    gorithm.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 $\exists r v. \text{decide } (rho \ r \ p) = \text{Some } v$

proof –

from *comm-global* **obtain** *r1* **where** *r1*:

$\forall q. \text{HOs } r1 \ p = \text{HOs } r1 \ q$

$\forall q. (\text{coin } r1 \ p :: \text{val}) = \text{coin } r1 \ q$

two-step *r1* = 1

by(*simp add: BenOr-commGlobal-def all-conj-distrib, blast*)

from *r1* **obtain** *r0* **where** *r1-def*: $r1 = \text{Suc } r0$ **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  $\mu0 \mu1 \mu2 \mu3$  where
    send0:  $\forall p. \mu0 p \in \text{get-msgs } (\text{send0 } r0) \text{ cfg0 } (\text{HOs } r0) (\text{HOs } r0) p$ 
    and step0:  $\forall p. \text{next0 } r0 p (\text{cfg0 } p) (\mu0 p) (\text{cfg1 } p)$ 
    and send1:  $\forall p. \mu1 p \in \text{get-msgs } (\text{send1 } r1) \text{ cfg1 } (\text{HOs } r1) (\text{HOs } r1) p$ 
    and step1:  $\forall p. \text{next1 } r1 p (\text{cfg1 } p) (\mu1 p) (\text{cfg2 } p)$ 
    and send2:  $\forall p. \mu2 p \in \text{get-msgs } (\text{send0 } r2) \text{ cfg2 } (\text{HOs } r2) (\text{HOs } r2) p$ 
    and step2:  $\forall p. \text{next0 } r2 p (\text{cfg2 } p) (\mu2 p) (\text{cfg3 } p)$ 
    and send3:  $\forall p. \mu3 p \in \text{get-msgs } (\text{send1 } r3) \text{ cfg3 } (\text{HOs } r3) (\text{HOs } r3) p$ 
    and step3:  $\forall p. \text{next1 } r3 p (\text{cfg3 } p) (\mu3 p) (\text{cfg4 } p)$ 
  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  $?v = x (\text{cfg2 } p)$ 
from per-rd r1 have  $xs: \forall q. x (\text{cfg2 } q) = ?v$ 
proof(cases  $\exists q w. q \in \text{HOs } r1 p \wedge \text{vote } (\text{cfg1 } q) = \text{Some } w$ )
  case True
    then obtain  $q w$  where  $q-w: q \in \text{HOs } r1 p \wedge \text{vote } (\text{cfg1 } q) = \text{Some } w$ 
    by auto
    then have  $\forall q'. \text{vote } (\text{cfg1 } q') \in \{\text{None}, \text{Some } w\}$  using same-new-vote[OF
send0 step0 step-r0]
    by (metis insert-iff not-None-eq)
    hence  $\forall q'. x (\text{cfg2 } q') = w$  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  $\forall q'. x (\text{cfg2 } q') = \text{coin } r1 q'$  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  $\forall q. \text{vote } (cfg3\ q) = \text{Some } ?v$ 
  by(simp add: vote-origin[OF send2 step2 step-r2])

hence  $\text{decide } (cfg4\ p) = \text{Some } ?v$  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 ::  $v\text{-state} \Rightarrow \text{process set} \Rightarrow \text{val} \Rightarrow \text{bool}$  where
  mru-guard  $s\ Q\ v \equiv Q \in \text{Quorum} \wedge (\text{let } mru = \text{mru-of-set } (\text{votes } s)\ Q \text{ in}$ 

```

$$mru = None \vee (\exists r. mru = Some (r, v))$$

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: $s \in SV\text{-}inv4$
and *inv1*: $s \in Vinv1$
and *mru-vote*: $mru\text{-}guard\ s\ Q\ v$
and *is-Quorum*: $Q \in Quorum$
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
empty: $vote\text{-}set\ (votes\ s)\ Q \neq \{\}$
and *max*: $Max\text{-}by\ fst\ (vote\text{-}set\ (votes\ s)\ Q) = (r, v)$

from *Max-by-in[OF Vinv1-finite-vote-set[OF inv1] empty] max*
have *in-votes*: $(r, v) \in vote\text{-}set\ (votes\ s)\ Q$ **by** *metis*

have *no-larger*: $\forall a' \in Q. \forall r' > r. votes\ s\ r'\ a' = None$
proof(*safe, rule ccontr, clarsimp*)
fix *a' r' w*
assume $a' \in Q\ votes\ s\ r'\ a' = Some\ w$ **and** *gt*: $r' > r$
hence $(r', w) \in vote\text{-}set\ (votes\ s)\ Q$
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 \vee ($\exists r$. mru = Some (r, v)))

definition *opt-mru-round*

:: round \Rightarrow process set \Rightarrow process set \Rightarrow val \Rightarrow (process, val)map \Rightarrow (opt-mru-state \times opt-mru-state) set

where

opt-mru-round r Q S v r-decisions = {(s, s[^]).

— guards

r = next-round s

\wedge (S \neq {} \longrightarrow opt-mru-guard (mru-vote s) Q v)

\wedge d-guard r-decisions (const-map v S)

\wedge — 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 \times opt-mru-state) set **where**

mru-opt-trans = (\bigcup r Q S v D. *opt-mru-round* r Q S v D) \cup 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 \times 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 \Rightarrow (process, val)map) \Rightarrow process set **where**
voters *vs* = {*a* | *a* *r* *v*. ((*r*, *a*), *v*) \in map-graph (case-prod *vs*)}

lemma *vote-set-as-Union*:

vote-set *vs* *Q* = (\bigcup *a* \in (*Q* \cap *voters* *vs*)). *vote-set* *vs* {*a*}
by(*auto simp add: vote-set-def voters-def*)

lemma *empty-ran*:

(*ran* *f* = { }) = (\forall *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*:

$\llbracket (S = \{\}) \longleftrightarrow (S' = \{\}); S \neq \{\} \wedge S' \neq \{\} \implies \text{Max-by } f \ S = \text{Max-by } g \ S' \rrbracket$
 $\implies \text{option-Max-by } f \ S = \text{option-Max-by } g \ S'$
by(*auto simp add: option-Max-by-def*)

lemma *ran-process-mru-only-voters*:

ran (*process-mru* *vs* |' *Q*) = *ran* (*process-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-inv3* \implies *inj-on* *fst* (*vote-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-inv3*
defines *vs* \equiv *votes* *s*
shows

opt-mru-vote (*process-mru* *vs*) *Q* = *mru-of-set* *vs* *Q*

proof(*simp add: opt-mru-vote-def mru-of-set-def, intro option-Max-by-eqI, clar-*

simp-all)
show ($\text{ran } (\text{process-mru } vs \mid ' Q) = \{\}$) = ($\text{vote-set } vs \ Q = \{\}$)
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*: $\text{ran } (\text{process-mru } vs \mid ' Q) \neq \{\}$ $\text{vote-set } vs \ Q \neq \{\}$

hence *nempty'*: $Q \cap \text{voters } vs \neq \{\}$
by(*auto simp add: vote-set-def voters-def*)

have *nempty''*: $\{\} \notin (\lambda a. \text{vote-set } vs \ \{a\}) \ ' (Q \cap \text{voters } vs)$
by(*auto simp add: vote-set-def voters-def*)

note *fin=Vinv1-finite-vote-set[OF inv1]*

have *ran-eq*:
 $\text{ran } (\text{process-mru } vs \mid ' Q) = \text{Max-by fst } ' (\lambda a. \bigcup_{a \in \{a\}} \text{voters } vs. \text{vote-set } vs \ \{a\}) \ ' (Q \cap \text{voters } vs)$
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* ($\text{ran } (\text{process-mru } vs \mid ' Q)$) = *Max-by fst* ($\text{vote-set } vs \ Q$)
apply(*subst vs-def*)
 $\text{Max-by-UNION-distrib}[OF \text{fin } \text{vote-set-as-Union } \text{nempty}'[\text{simplified } \text{vs-def}] \text{nempty}''[\text{simplified } \text{vs-def}] \text{inj}])+$
apply(*subst vote-set-as-Union*)
by(*metis ran-eq vs-def*)
qed

lemma *opt-mru-guard-imp-mru-guard*:
assumes *invs*:
 $s \in \text{Vinv1 } s \in \text{SV-inv3}$
and *c-guard*: *opt-mru-guard* ($\text{process-mru } (\text{votes } s)$) $Q \ v$

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 \implies *Q* \in *Quorum*

by (*simp add: opt-mru-guard-def Let-def*)

lemma *opt-mru-round-refines*:

$\{lv\text{-ref-rel} \cap (Vinv1 \cap SV\text{-inv3} \cap SV\text{-inv4}) \times UNIV\}$

sv-round r S v d-f, opt-mru-round r Q S v d-f

$\{> lv\text{-ref-rel}\}$

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* \cap (*Vinv1* \cap *Vinv2* \cap *SV-inv3* \cap *SV-inv4*) \times *UNIV*) *sv-TS*
mru-opt-TS

proof(*rule refine-using-invariants*)

show *init mru-opt-TS* \subseteq *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  $\cap$  (Vinv1  $\cap$  Vinv2  $\cap$  SV-inv3  $\cap$  SV-inv4)  $\times$  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  $\cap$  Vinv2  $\cap$  SV-inv3  $\cap$  SV-inv4  $\cap$  Domain (lv-ref-rel  $\cap$  UNIV  $\times$ 
    UNIV)}
    trans sv-TS
    {> Vinv1  $\cap$  Vinv2  $\cap$  SV-inv3  $\cap$  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.  $\forall p$ . (case mru-vote s p of
    Some (r, -)  $\Rightarrow$  r < next-round s
    | None  $\Rightarrow$  True)
  }

```

lemma *OMRU-inv1-inductive:*

```

  init mru-opt-TS  $\subseteq$  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  $\Rightarrow$  val set  $\Rightarrow$  (three-step-mru-state  $\times$  three-step-mru-state)
set where

```

```

  opt-mru-step0 r C = {(s, s').
    — guards
    r = next-round s  $\wedge$  three-step r = 0
     $\wedge$  ( $\forall$  cand  $\in$  C.  $\exists$  Q. opt-mru-guard (mru-vote s) Q cand)
     $\wedge$  — actions
    s' = s[
      candidates := C
      , next-round := Suc r
    ]
  }

```

```

definition opt-mru-step1 :: round  $\Rightarrow$  process set  $\Rightarrow$  val  $\Rightarrow$ 
  (three-step-mru-state  $\times$  three-step-mru-state) set where

```

```

  opt-mru-step1 r S v = {(s, s').
    — guards
    r = next-round s  $\wedge$  three-step r = 1
     $\wedge$  (S  $\neq$  {}  $\longrightarrow$  v  $\in$  candidates s)
     $\wedge$  — 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 \Rightarrow (process, val)map \Rightarrow bool **where**
step2-d-guard *r-decisions* *r-votes* $\equiv \forall p v. r\text{-decisions } p = \text{Some } v \longrightarrow$
 $v \in \text{ran } r\text{-votes} \wedge \text{dom } r\text{-votes} \in \text{Quorum}$

definition *r-votes* :: three-step-mru-state \Rightarrow round \Rightarrow (process, val)map **where**
r-votes *s* $\equiv \lambda p. \text{if } (\exists v. \text{mru-vote } s p = \text{Some } (\text{three-phase } r, v))$
then map-option snd (mru-vote s p)
else None

definition *opt-mru-step2* :: round \Rightarrow (process, val)map \Rightarrow (three-step-mru-state
 \times three-step-mru-state) set **where**

opt-mru-step2 *r* *r-decisions* = $\{(s, s')\}$.
 — guards
 $r = \text{next-round } s \wedge \text{three-step } r = 2$
 $\wedge \text{step2-d-guard } r\text{-decisions } (r\text{-votes } s r)$
 — actions
 $s' = s[$
 $\text{next-round} := \text{Suc } r$
 $\text{decisions} := \text{decisions } s ++ r\text{-decisions}$
 $\left. \right]$
 $\}$

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 \times three-step-mru-state) set **where**
ts-mru-trans = $(\bigcup r C. \text{opt-mru-step0 } r C)$
 $\cup (\bigcup r S v. \text{opt-mru-step1 } r S v)$
 $\cup (\bigcup r \text{dec-f. } \text{opt-mru-step2 } r \text{dec-f}) \cup \text{Id}$

definition *ts-mru-init* **where**

ts-mru-init = $\{ (\mid \text{next-round} = 0, \text{mru-vote} = \text{Map.empty}, \text{decisions} = \text{Map.empty},$
candidates = $\{\}$ $\mid \}$

definition *ts-mru-TS* :: three-step-mru-state *TS* **where**

ts-mru-TS = $(\mid \text{init} = \text{ts-mru-init}, \text{trans} = \text{ts-mru-trans} \mid)$

lemmas *ts-mru-TS-defs* = *ts-mru-TS-def* *ts-mru-init-def* *ts-mru-trans-def*

16.2 Refinement

definition *basic-rel* **where**

$$\begin{aligned} \text{basic-rel} &\equiv \{(sa, sc). \\ &\quad \text{decisions } sc = \text{decisions } sa \\ &\quad \wedge \text{next-round } sa = \text{three-phase } (\text{next-round } sc) \\ &\} \end{aligned}$$

definition *three-step0-rel* $:: (\text{opt-mru-state} \times \text{three-step-mru-state})\text{set}$ **where**

$$\begin{aligned} \text{three-step0-rel} &\equiv \text{basic-rel} \cap \{(sa, sc). \\ &\quad \text{three-step } (\text{next-round } sc) = 0 \\ &\quad \wedge \text{mru-vote } sc = \text{mru-vote } sa \\ &\} \end{aligned}$$

definition *three-step1-rel* $:: (\text{opt-mru-state} \times \text{three-step-mru-state})\text{set}$ **where**

$$\begin{aligned} \text{three-step1-rel} &\equiv \text{basic-rel} \cap \{(sa, sc). \\ &\quad (\exists sc' r C. (sa, sc') \in \text{three-step0-rel} \wedge (sc', sc) \in \text{opt-mru-step0 } r C) \\ &\quad \wedge \text{mru-vote } sc = \text{mru-vote } sa \\ &\} \end{aligned}$$

definition *three-step2-rel* $:: (\text{opt-mru-state} \times \text{three-step-mru-state})\text{set}$ **where**

$$\begin{aligned} \text{three-step2-rel} &\equiv \text{basic-rel} \cap \{(sa, sc). \\ &\quad (\exists sc' r S v. (sa, sc') \in \text{three-step1-rel} \wedge (sc', sc) \in \text{opt-mru-step1 } r S v) \\ &\} \end{aligned}$$

definition *ts-ref-rel* **where**

$$\begin{aligned} \text{ts-ref-rel} &= \{(sa, sc). \\ &\quad (\text{three-step } (\text{next-round } sc) = 0 \longrightarrow (sa, sc) \in \text{three-step0-rel}) \\ &\quad \wedge (\text{three-step } (\text{next-round } sc) = 1 \longrightarrow (sa, sc) \in \text{three-step1-rel}) \\ &\quad \wedge (\text{three-step } (\text{next-round } sc) = 2 \longrightarrow (sa, sc) \in \text{three-step2-rel}) \\ &\} \end{aligned}$$

lemmas *ts-ref-rel-defs* =

$$\begin{aligned} &\text{basic-rel-def} \\ &\text{ts-ref-rel-def} \\ &\text{three-step0-rel-def} \\ &\text{three-step1-rel-def} \\ &\text{three-step2-rel-def} \end{aligned}$$

lemma *step0-ref*:

$\{ts\text{-ref-rel}\} Id, \text{opt-mru-step0 } r \ C \ \{> \ ts\text{-ref-rel}\}$

proof(*simp only: PO-rhoare-defs, safe*)

fix *sa sc sc'*

assume *R*: $(sa, sc) \in ts\text{-ref-rel}$ **and** *step*: $(sc, sc') \in \text{opt-mru-step0 } r \ C$

hence *r-step*: $\text{three-step } (\text{next-round } sc) = 0 \ \text{three-step } (\text{next-round } sc') = 1$

by(*auto simp add: ts-ref-rel-def opt-mru-step0-def three-step-Suc*)

hence $(sa, sc') \in ts\text{-ref-rel}$ **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 $\exists sa'. (sa, sa') \in Id \wedge (sa', sc') \in ts\text{-ref-rel}$

by *blast*

qed

lemma *step1-ref*:

$\{ts\text{-ref-rel}\} Id, \text{opt-mru-step1 } r \ S \ v \ \{> \ ts\text{-ref-rel}\}$

proof(*simp only: PO-rhoare-defs, safe*)

fix *sa sc sc'*

assume *R*: $(sa, sc) \in ts\text{-ref-rel}$ **and** *step*: $(sc, sc') \in \text{opt-mru-step1 } r \ S \ v$

hence *r-step*: $\text{three-step } (\text{next-round } sc) = 1 \ \text{three-step } (\text{next-round } sc') = 2$

by(*auto simp add: ts-ref-rel-def opt-mru-step1-def three-step-Suc*)

hence $(sa, sc') \in ts\text{-ref-rel}$ **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 $\exists sa'. (sa, sa') \in Id \wedge (sa', sc') \in ts\text{-ref-rel}$

by *blast*

qed

lemma *step2-ref*:

$\{ts\text{-ref-rel} \cap \text{OMRU-inv1} \times \text{UNIV}\}$

$\bigcup r' \ Q \ S' \ v \ \text{dec-f}'. \ \text{opt-mru-round } r' \ Q \ S' \ v \ \text{dec-f}'$,

$\text{opt-mru-step2 } r \ \text{dec-f} \ \{> \ ts\text{-ref-rel}\}$

proof(*auto simp only: PO-rhoare-defs*)

fix *sa sc2 sc3*

assume
ainv: $sa \in \text{OMRU-inv1}$
and R : $(sa, sc2) \in \text{ts-ref-rel}$
and step2 : $(sc2, sc3) \in \text{opt-mru-step2 } r \text{ dec-f}$

from R step2 **obtain** $sc0$ $r0$ C $sc1$ $r1$ S v **where**
 $R0$: $(sa, sc0) \in \text{three-step0-rel}$ **and** step0 : $(sc0, sc1) \in \text{opt-mru-step0 } r0$ C
and $R1$: $(sa, sc1) \in \text{three-step1-rel}$ **and** step1 : $(sc1, sc2) \in \text{opt-mru-step1 } r1$
 S v
by(*fastforce simp add: ts-ref-rel-def three-step2-rel-def opt-mru-step2-def three-step1-rel-def*)

have $R2$: $(sa, sc2) \in \text{three-step2-rel}$
and $r2\text{-step}$: $\text{three-step } (\text{next-round } sc2) = \text{Suc } (\text{Suc } 0)$
and $r3\text{-step}$: $\text{three-step } (\text{next-round } sc3) = 0$
using R step2
by(*auto simp add: ts-ref-rel-def opt-mru-step2-def three-step-phase-Suc three-step-Suc*)

have r : $r = \text{Suc } r1$ **and** $r1$: $r1 = \text{Suc } r0$ $r1 = \text{next-round } sc1$ **and**
 $r0$: $r0 = \text{next-round } sc0$ **and** $r0\text{-step}$: $\text{three-step } r0 = 0$ **and**
 $r2$: $r = \text{next-round } sc2$
using step0 step1 step2
by(*auto simp add: opt-mru-step0-def opt-mru-step1-def opt-mru-step2-def*)

have abs-round2 : $\text{next-round } sa = \text{three-phase } r$ **using** $R2$ $r2$
by(*auto simp add: three-step2-rel-def basic-rel-def*)
have abs-round1 : $\text{next-round } sa = \text{three-phase } r1$ **using** $R1$ $r1$
by(*auto simp add: three-step1-rel-def basic-rel-def*)
have abs-round0 : $\text{next-round } sa = \text{three-phase } r0$ **using** $R0$ $r0$ r
by(*auto simp add: three-step0-rel-def basic-rel-def three-step-phase-Suc*)

have $r\text{-votes}$: $r\text{-votes } sc2$ $r = \text{const-map } v$ S
proof(*rule ext*)
fix p
show $r\text{-votes } sc2$ r $p = \text{const-map } v$ 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 ≠ {} → 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) → 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')  $\in$  opt-mru-round (three-phase r) Q S v dec-f  $\wedge$  (sa', sc3)  $\in$ 
ts-ref-rel

```

```

proof
  show (sa', sc3)  $\in$  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')  $\in$  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  $\exists y. (sa, y) \in (\bigcup r' Q S' v D'. \text{opt-mru-round } r' Q S' v D') \wedge (y, sc\beta) \in$ 
ts-ref-rel
    by simp blast
qed

lemma ThreeStep-Coordinated-Refines:
    PO-refines (ts-ref-rel  $\cap$  OMRU-inv1  $\times$  UNIV)
    mru-opt-TS ts-mru-TS
proof(rule refine-using-invariants)
    show init ts-mru-TS  $\subseteq$  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  $\cap$  OMRU-inv1  $\times$  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* \rightarrow *msg*)
 \Rightarrow (*process*, *round* \times *val*) *map*

where

msgs-to-lvs msgs \equiv *msg-to-val-stamp* \circ_m *msgs*

definition *smallest-proposal* **where**

smallest-proposal (msgs::process \rightarrow *msg*) \equiv
 $\text{Min } \{v. \exists q \text{ mv. } \text{msgs } q = \text{Some } (\text{MruVote } \text{mv } v)\}$

definition *next0*

$::$ *nat*
 \Rightarrow *process*
 \Rightarrow *pstate*
 \Rightarrow (*process* \rightarrow *msg*)
 \Rightarrow *process*
 \Rightarrow *pstate*
 \Rightarrow *bool*

where

next0 r p st msgs crd st' \equiv *let*
 $Q = \text{dom } \text{msgs};$
 $\text{lvs} = \text{msgs-to-lvs } \text{msgs};$
 $\text{smallest} = \text{if } Q = \{\} \text{ then } x \text{ st else } \text{smallest-proposal } \text{msgs}$
in
 $\text{st}' = \text{st } ($
 $\text{prop-vote} := \text{if } \text{card } Q > N \text{ div } 2$
 $\text{then } \text{Some } (\text{case-option } \text{smallest } \text{snd } (\text{option-Max-by } \text{fst } (\text{ran } (\text{lvs } |' Q))))$
 $\text{else } \text{None}$
 $)$

definition *send1* **where**

send1 r p q st \equiv *case prop-vote st of*
 $\text{None} \Rightarrow \text{Null}$
 $| \text{Some } v \Rightarrow \text{PreVote } v$

definition *Q-prevotes-v* **where**

Q-prevotes-v msgs Q v \equiv *let* $D = \text{dom } \text{msgs}$ *in*
 $Q \subseteq D \wedge \text{card } Q > N \text{ div } 2 \wedge (\forall q \in Q. \text{msgs } q = \text{Some } (\text{PreVote } v))$

definition *next1*

$::$ *nat*

\Rightarrow *process*
 \Rightarrow *pstate*
 \Rightarrow (*process* \rightarrow *msg*)
 \Rightarrow *process*
 \Rightarrow *pstate*
 \Rightarrow *bool*

where

$next1\ r\ p\ st\ msgs\ crd\ st' \equiv$
 $decide\ st' = decide\ st$
 $\wedge\ x\ st' = x\ st$
 $\wedge\ (\forall Q\ v.\ Q\text{-prevotes-}v\ msgs\ Q\ v$
 $\longrightarrow\ mru\text{-vote}\ st' = Some\ (three\text{-phase}\ r,\ v))$
 $\wedge\ (\neg\ (\exists Q\ v.\ Q\text{-prevotes-}v\ msgs\ Q\ v)$
 $\longrightarrow\ mru\text{-vote}\ st' = mru\text{-vote}\ st)$

definition *send2* **where**

$send2\ r\ p\ q\ st \equiv case\ mru\text{-vote}\ st\ of$
 $None \Rightarrow Null$
 $| Some\ (\Phi,\ v) \Rightarrow if\ \Phi = three\text{-phase}\ r\ then\ Vote\ v\ else\ Null$

definition *Q'-votes-v* **where**

$Q'\text{-votes-}v\ r\ msgs\ Q\ Q'\ v \equiv$
 $Q' \subseteq Q \wedge card\ Q' > N\ div\ 2 \wedge (\forall q \in Q'.\ msgs\ q = Some\ (Vote\ v))$

definition *next2*

$::\ nat$
 \Rightarrow *process*
 \Rightarrow *pstate*
 \Rightarrow (*process* \rightarrow *msg*)
 \Rightarrow *process*
 \Rightarrow *pstate*
 \Rightarrow *bool*

where

$next2\ r\ p\ st\ msgs\ crd\ st' \equiv let\ Q = dom\ msgs;\ lvs = msgs\text{-to-}lvs\ msgs\ in$
 $x\ st' = x\ st$
 $\wedge\ mru\text{-vote}\ st' = mru\text{-vote}\ st$
 $\wedge\ (\forall Q'\ v.\ Q'\text{-votes-}v\ r\ msgs\ Q\ Q'\ v \longrightarrow decide\ st' = Some\ v)$
 $\wedge\ (\neg\ (\exists Q'\ v.\ Q'\text{-votes-}v\ r\ msgs\ Q\ Q'\ v \longrightarrow decide\ st' = decide\ st))$

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* \rightarrow *msg*)
 \Rightarrow *process* \Rightarrow *pstate* \Rightarrow *bool*
where
NA-nextState *r* \equiv
 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*) \equiv *True*

definition
NA-commGlobal **where**
NA-commGlobal *HOrs* \equiv
 \exists *ph*::*nat*. \forall *i* \in {0..2}.
 $(\forall p$. *card* (*HOrs* (*nr-steps***ph*+*i*) *p*) > *N div 2*)
 \wedge ($\forall p$ *q*. *HOrs* (*nr-steps***ph*+*i*) *p* = *HOrs* (*nr-steps***ph*) *q*)

definition *New-Algo-Alg* **where**
New-Algo-Alg \equiv
 \langle *CinitState* = *NA-initState*,
sendMsg = *NA-sendMsg*,
CnextState = *NA-nextState* \rangle

definition *New-Algo-HOMachine* **where**
New-Algo-HOMachine \equiv
 \langle *CinitState* = *NA-initState*,
sendMsg = *NA-sendMsg*,

$CnextState = NA-nextState,$
 $HOcommPerRd = NA-commPerRd,$
 $HOcommGlobal = NA-commGlobal$)

abbreviation

$New-Algo-M \equiv (New-Algo-HOMachine::(process, pstate, msg) HOMachine)$

end

17.3 Proofs

type-synonym $p-TS-state = (nat \times (process \Rightarrow pstate))$

definition $New-Algo-TS ::$

$(round \Rightarrow process HO) \Rightarrow (round \Rightarrow process HO) \Rightarrow (round \Rightarrow process) \Rightarrow$
 $p-TS-state TS$

where

$New-Algo-TS HOs SHOs crds = CHO-to-TS New-Algo-Alg HOs SHOs (K \circ$
 $crds)$

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**

$New-Algo-trans-step HOs SHOs crds next-f snd-f stp \equiv \bigcup r \mu.$
 $\{((r, cfg), (Suc r, cfg')) | cfg \text{ cfg}'. \text{three-step } r = stp \wedge (\forall p.$
 $\mu p \in \text{get-msgs } (snd-f r) \text{ cfg } (HOs r) (SHOs r) p$
 $\wedge \text{next-f } r p \text{ (cfg } p) (\mu p) (crds r) (cfg' p)$
 $)\}$

lemma $three-step-less-D:$

$0 < \text{three-step } r \implies \text{three-step } r = 1 \vee \text{three-step } r = 2$
by($\text{unfold three-step-def, arith}$)

lemma $New-Algo-trans:$

$C SHO-trans-alt NA-sendMsg NA-nextState HOs SHOs (K \circ crds) =$
 $New-Algo-trans-step HOs SHOs crds next0 send0 0$
 $\cup New-Algo-trans-step HOs SHOs crds next1 send1 1$
 $\cup New-Algo-trans-step HOs SHOs crds next2 send2 2$

proof(*rule equalityI*)
show *CSHO-trans-alt NA-sendMsg NA-nextState HOs SHOs (K o crds)*
 \subseteq *New-Algo-trans-step HOs SHOs crds next0 send0 0* \cup
New-Algo-trans-step HOs SHOs crds next1 send1 1 \cup
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* \cup
New-Algo-trans-step HOs SHOs crds next1 send1 1 \cup
New-Algo-trans-step HOs SHOs crds next2 send2 2
 \subseteq *CSHO-trans-alt NA-sendMsg NA-nextState HOs SHOs (K o 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 $rHO = nat \Rightarrow process\ HO$

17.3.1 Refinement

definition *new-algo-ref-rel* :: (*three-step-mru-state* \times *p-TS-state*)*set* **where**
new-algo-ref-rel = {(*sa*, (*r*, *sc*)).
opt-mru-state.next-round sa = r
 \wedge *opt-mru-state.decisions sa = pstate.decide o sc*
 \wedge *opt-mru-state.mru-vote sa = pstate.mru-vote o sc*
 \wedge (*three-step r = Suc 0* \longrightarrow *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*:
 $\forall p. next0\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow mru-vote\ o\ s' = mru-vote\ o\ s$
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*:
 $\forall p. next2\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow mru-vote\ o\ s' = mru-vote\ o\ s$
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:*

$\forall p. \text{next0 } r \ p \ (s \ p) \ (\text{msgs } p) \ (\text{crd } p) \ (s' \ p) \implies \text{decide } o \ s = \text{decide } o \ s'$

$\forall p. \text{next1 } r \ p \ (s \ p) \ (\text{msgs } p) \ (\text{crd } p) \ (s' \ p) \implies \text{decide } o \ s = \text{decide } o \ s'$

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

$\mu \ p \in \text{get-msgs } (\text{send0 } r) \ \text{cfg} \ (\text{HOs } r) \ (\text{HOs } r) \ p$

shows $((\text{msgs-to-lvs } (\mu \ p)) \mid' \ \text{HOs } r \ p) = (\text{mru-vote } o \ \text{cfg}) \mid' \ \text{HOs } r \ p$ **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:*

$\{\text{new-algo-ref-rel}\}$

$(\bigcup r \ C. \text{majorities.opt-mru-step0 } r \ C),$

$\text{New-Algo-trans-step } \text{HOs } \text{HOs } \text{crds } \text{next0 } \text{send0 } 0 \ \{> \ \text{new-algo-ref-rel}\}$

proof(*clarsimp simp add: PO-rhoare-defs New-Algo-trans-step-def all-conj-distrib*)

fix $r \ sa \ sc \ sc' \ \mu$

assume $R: (sa, (r, sc)) \in \text{new-algo-ref-rel}$

and $r: \text{three-step } r = 0$

and $\mu: \forall p. \mu \ p \in \text{get-msgs } (\text{send0 } r) \ sc \ (\text{HOs } r) \ (\text{HOs } r) \ p$

and $\text{next}: \forall p. \text{next0 } r \ p \ (sc \ p) \ (\mu \ p) \ (\text{crds } r) \ (sc' \ p)$

note $\mu \text{next} = \mu \ \text{next}$

have $r\text{-phase-step}: nr\text{-steps} * \text{three-phase } r = r$ **using** $r \ \text{three-phase-step}[of \ r]$

by(*auto*)

define C **where** $C = \text{ran } (\text{prop-vote } o \ sc')$

have $\text{guard}: \forall \text{cand} \in C. \exists Q. \text{majorities.opt-mru-guard } (\text{mru-vote } o \ sc) \ Q \ \text{cand}$

proof(*simp add: C-def ran-def, safe*)

fix $p \ \text{cand}$

assume $\text{Some}: \text{prop-vote } (sc' \ p) = \text{Some } \text{cand}$

let $?Q = \text{HOs } r \ p$

let $?lvs0 = \text{mru-vote } o \ sc$

have $?Q \in \text{majS}$ **using** $\text{Some } \mu \text{next}[THEN \ \text{spec}, \ \text{where } x=p]$

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))) $\in \{None, Some\ cand\}$
using *Some nxt[THEN spec, where $x=p$]*
msgs-mru-vote[where $HOs=HOs$ and $\mu=\mu$, OF $\mu[THEN spec, of p]$]
get-msgs-dom[OF $\mu[THEN spec, of p]$]
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 $\exists Q.$ *majorities.opt-mru-guard ?lvs0 Q cand*
by *blast*

qed

define *sa'* **where** *sa' = sa*(
next-round := Suc r,
candidates := C
)
have *(sa, sa') \in 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')) \in 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
 $\exists sa'. (\exists r C. (sa, sa') \in majorities.opt-mru-step0 r C)$
 $\wedge (sa', Suc r, sc') \in new-algo-ref-rel$
by *blast*

qed

lemma *step1-ref:*
{new-algo-ref-rel}
 $(\bigcup r S v. majorities.opt-mru-step1 r S v),$
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)) \in new-algo-ref-rel*
and *r: three-step r = Suc 0*


```

and  $\mu$ :  $\forall p. \mu p \in \text{get-msgs } (\text{send1 } r) \text{ } sc \text{ } (HOs \ r) \text{ } (HOs \ r) \ p$ 
and  $next$ :  $\forall p. \text{next1 } r \ p \ (sc \ p) \ (\mu \ p) \ (crds \ r) \ (sc' \ p)$ 
note  $\mu next = \mu \ next$ 

define  $S$  where  $S = \{p. \text{mru-vote } (sc' \ p) \neq \text{mru-vote } (sc \ p)\}$ 

have  $S$ :  $S \subseteq \{p. \exists Q \ v. Q \subseteq HOs \ r \ p$ 
   $\wedge (\forall q \in Q. \text{prop-vote } (sc \ q) = \text{Some } v)$ 
   $\wedge Q \in \text{maj}s$ 
   $\wedge (\text{mru-vote } (sc' \ p) = \text{Some } (\text{three-phase } r, \ v))$ 
   $\}$ 
proof(safe)
fix  $p$ 
assume  $p \in S$ 
then obtain  $Q \ v$ 
where
   $\forall q \in Q. \mu \ p \ q = \text{Some } (\text{PreVote } v)$ 
and  $\text{maj-Q}$ :  $Q \in \text{maj}s$ 
and  $Q\text{-HOs}$ :  $Q \subseteq \text{dom } (\mu \ p)$ 
and  $lv$ :  $\text{mru-vote } (sc' \ p) = \text{Some } (\text{three-phase } r, \ v)$  (is  $?LV \ v$ )
using  $next$ [THEN spec, where x=p]
by(clarsimp simp add: next1-def Let-def S-def maj-s-def Q-prevotes-v-def)

then have
   $\forall q \in Q. \text{prop-vote } (sc \ q) = \text{Some } v$  (is  $?P \ Q \ v$ )
   $Q \subseteq HOs \ r \ p$ 
using  $\mu$ [THEN spec, where x=p]
  by(auto simp add: get-msgs-benign send1-def restrict-map-def split: option.split-asm)

with  $\text{maj-Q}$  and  $lv$  show  $\exists Q \ v. Q \subseteq HOs \ r \ p \wedge ?P \ Q \ v \wedge Q \in \text{maj}s \wedge ?LV$ 
 $v$  by blast
qed

obtain  $v$  where
   $v: \forall p \in S. \text{mru-vote } (sc' \ p) = \text{Some } (\text{three-phase } r, \ v) \wedge v \in \text{ran } (\text{prop-vote } o$ 
 $sc)$ 
proof(cases S = {})
case False
assume asm:

```

$\bigwedge v. \forall p \in S. \text{pstate.mru-vote } (sc' p) = \text{Some } (\text{three-phase } r, v) \wedge v \in \text{ran } (\text{prop-vote } o \text{ } sc)$
 $\implies \text{thesis}$
from *False* **obtain** $p \in S$ **by** *auto*
with $S \text{ next}$
obtain $Q v$
where $\text{prop-vote}: (\forall q \in Q. \text{prop-vote } (sc q) = \text{Some } v)$ **and** $\text{maj-Q}: Q \in \text{majS}$
by *auto*

hence $\forall p \in S. \text{pstate.mru-vote } (sc' p) = \text{Some } (\text{three-phase } r, v)$ (**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** $sa' = sa \lfloor \text{next-round} := \text{Suc } r,$
 $\text{opt-mru-state.mru-vote} := \text{opt-mru-state.mru-vote } sa ++ \text{const-map } (\text{three-phase } r, v) S$
 \rfloor

have $(sa, sa') \in \text{majorities.opt-mru-step1 } r S v$ **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** $(sa', (\text{Suc } r, sc')) \in \text{new-algo-ref-rel}$ **using** $r R$
proof–

have $\text{mru-vote } o \text{ } sc' = ((\text{mru-vote } o \text{ } sc) ++ \text{const-map } (\text{three-phase } r, v) S)$
proof(*rule ext, simp*)
fix p
show $\text{mru-vote } (sc' p) = ((\text{mru-vote } o \text{ } sc) ++ \text{const-map } (\text{three-phase } r, v) S) p$
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** $R r \text{ next}$
by(*force simp add: new-algo-ref-rel-def sa'-def three-step-Suc intro: de-*)

```

cide-evolution)
qed
ultimately show
   $\exists sa'. (\exists r S v. (sa, sa') \in \text{majorities.opt-mru-step1 } r S v)$ 
     $\wedge (sa', \text{Suc } r, sc') \in \text{new-algo-ref-rel}$ 
  by blast
qed

lemma step2-ref:
  {new-algo-ref-rel}
  ( $\bigcup r \text{dec-f. majorities.opt-mru-step2 } r \text{dec-f}$ ),
  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'  $\mu$ 
  assume R: (sa, (r, sc))  $\in$  new-algo-ref-rel
  and r: three-step r = 2
  and  $\mu: \forall p. \mu p \in \text{get-msgs (send2 } r) \text{sc (HOs } r) \text{(HOs } r) p$ 
    and next:  $\forall p. \text{next2 } r p (\text{sc } p) (\mu p) (\text{crds } r) (\text{sc}' p)$ 
  note  $\mu \text{next} = \mu \text{next}$ 

  define dec-f
    where dec-f p = (if decide (sc' p)  $\neq$  decide (sc p) then decide (sc' p) else
None) for p

  have dec-f: (decide  $\circ$  sc) ++ dec-f = decide  $\circ$  sc'
  proof
    fix p
    show ((decide  $\circ$  sc) ++ dec-f) p = (decide  $\circ$  sc') p using next[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'))  $\in$  new-algo-ref-rel using R r next
  by (auto simp add: new-algo-ref-rel-def sa'-def dec-f three-step-Suc

```

$mru\text{-vote}\text{-evolution2}[OF\ next]$
moreover have $(sa, sa') \in majorities.opt\text{-}mru\text{-}step2\ r\ dec\text{-}f$ **using** $r\ R$
proof–
define $sc\text{-}r\text{-}votes$ **where** $sc\text{-}r\text{-}votes\ p = (if\ (\exists v. mru\text{-}vote\ (sc\ p) = Some\ (three\text{-}phase\ r, v))$
 $then\ map\ option\ snd\ (mru\text{-}vote\ (sc\ p))$
 $else\ None)$ **for** p
have $sc\text{-}r\text{-}votes: sc\text{-}r\text{-}votes = majorities.r\text{-}votes\ sa\ r$ **using** $R\ r$
by $(auto\ simp\ add: new\text{-}algo\text{-}ref\text{-}rel\text{-}def\ sc\text{-}r\text{-}votes\text{-}def\ majorities.r\text{-}votes\text{-}def$
 $intro!: ext)$
have $majorities.step2\text{-}d\text{-}guard\ dec\text{-}f\ sc\text{-}r\text{-}votes$
proof $(clarsimp\ simp\ add: majorities.step2\text{-}d\text{-}guard\text{-}def)$
fix $p\ v$
assume $d\text{-}f\text{-}p: dec\text{-}f\ p = Some\ v$
then obtain Q **where** $Q:$
 $Q \in maj_s$
and $vote: Q \subseteq HOs\ r\ p\ \forall q \in Q. \mu\ p\ q = Some\ (Vote\ v)$
using $next[THEN\ spec, of\ p]\ d\text{-}f\text{-}p$
by $(auto\ simp\ add: next2\text{-}def\ dec\text{-}f\text{-}def\ Q'\text{-}votes\text{-}v\text{-}def\ Let\text{-}def\ maj_s\text{-}def)$
have $mru\text{-}vote: \forall q \in Q. mru\text{-}vote\ (sc\ q) = Some\ (three\text{-}phase\ r, v)$ **using**
 $vote\ \mu[THEN\ spec, of\ p]$
by $(fastforce\ simp\ add: get\text{-}msgs\text{-}benign\ send2\text{-}def\ sc\text{-}r\text{-}votes\text{-}def\ restrict\text{-}map\text{-}def$
 $split: option.split\text{-}asm\ if\text{-}split\text{-}asm)$
hence $dom\ sc\text{-}r\text{-}votes \in maj_s$
by $(auto\ intro!: majorities.mono\text{-}quorum[OF\ Q]\ simp\ add: sc\text{-}r\text{-}votes\text{-}def)$
moreover have $v \in ran\ sc\text{-}r\text{-}votes$ **using** $Q[THEN\ majorities.quorum\text{-}non\text{-}empty]$
 $mru\text{-}vote$
by $(force\ simp\ add: sc\text{-}r\text{-}votes\text{-}def\ ex\text{-}in\text{-}conv[symmetric]\ intro: ranI)$
ultimately show $v \in ran\ sc\text{-}r\text{-}votes \wedge dom\ sc\text{-}r\text{-}votes \in maj_s$
by $blast$
qed

thus $?thesis$ **using** $r\ R$
by $(auto\ simp\ add: majorities.opt\text{-}mru\text{-}step2\text{-}def\ sa'\text{-}def\ new\text{-}algo\text{-}ref\text{-}rel\text{-}def$
 $sc\text{-}r\text{-}votes)$
qed

ultimately show
 $\exists sa'. (\exists r\ dec\text{-}f. (sa, sa') \in majorities.opt\text{-}mru\text{-}step2\ r\ dec\text{-}f)$

$\wedge (sa', Suc\ r, sc') \in new_algo_ref_rel$
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) \subseteq 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: $\forall r. HOcommPerRd\ New_Algo_M\ (HOs\ r)$*

and *commG: HOcommGlobal New-Algo-M HOs*

shows $\exists r\ v. decide\ (rho\ r\ p) = Some\ v$

proof —

from *commG* **obtain** *ph* **where**

HOs: $\forall i \in \{0..2\}$.

$(\forall p. card\ (HOs\ (nr_steps*ph+i)\ p) > N\ div\ 2)$

$\wedge (\forall p\ q. HOs\ (nr_steps*ph+i)\ p = HOs\ (nr_steps*ph)\ q)$

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  $\mu0 \mu1 \mu2$  where
  send0:  $\forall p. \mu0 p \in \text{get-msgs (send0 r0) cfg0 (HOs r0) (HOs r0) p}$ 
and three-step0:  $\forall p. \text{next0 r0 p (cfg0 p) (\mu0 p) undefined (cfg1 p)}$ 
and send1:  $\forall p. \mu1 p \in \text{get-msgs (send1 r1) cfg1 (HOs r1) (HOs r1) p}$ 
and three-step1:  $\forall p. \text{next1 r1 p (cfg1 p) (\mu1 p) undefined (cfg2 p)}$ 
and send2:  $\forall p. \mu2 p \in \text{get-msgs (send2 r2) cfg2 (HOs r2) (HOs r2) p}$ 
and three-step2:  $\forall p. \text{next2 r2 p (cfg2 p) (\mu2 p) undefined (cfg3 p)}$ 
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 $r0\dots$

```

from HOs[THEN bspec, where x=0, simplified] send0
have
   $\forall p q. \mu0 p = \mu0 q \forall p. N \text{ div } 2 < \text{card (dom (\mu0 p))}$ 
apply(auto simp add: get-msgs-benign send0-def r1-def r0-def dom-def re-
strict-map-def intro!: ext)
apply(blast)+
done

```

— ...hence everybody sets *prop-vote* to the same value...

hence *same-prevote*:

$\forall p. \text{prop-vote (cfg1 p)} \neq \text{None}$

$\forall p q. \text{prop-vote (cfg1 p)} = \text{prop-vote (cfg1 q)}$ **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:  $\forall p. \text{prop-vote } (cfg1\ p) = \text{Some } dec-v$ 
by (metis option.collapse)

```

— ...and since everybody hears from majority in r1...

```

from HOs[THEN bspec, where x=Suc 0, simplified] send1
have  $\forall p\ q. \mu1\ p = \mu1\ q \ \forall p. N\ \text{div}\ 2 < \text{card } (dom\ (\mu1\ p))$ 
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:  $\forall p. \text{mru-vote } (cfg2\ p) = \text{Some } (three-phase\ r2, dec-v)$ 

```

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:  $N\ \text{div}\ 2 < \text{card } Q \ Q \subseteq HOs\ r1\ p$ 
by(auto simp add: r1-def r0-def)
hence Q-prevotes-v ( $\mu1\ p$ ) 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:  $\forall p. \text{decide } (cfg3\ p) = \text{Some } dec-v$ 

```

proof

```

fix p
from HOs[THEN bspec, where x=Suc (Suc 0), simplified]
obtain Q where Q:  $N\ \text{div}\ 2 < \text{card } Q \ Q \subseteq HOs\ r2\ p$ 

```

```

    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]: $\forall r r'. \text{three-phase } r = \text{three-phase } r' \longrightarrow \text{coord } r = \text{coord } r'$
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* $\equiv \exists v ts. m = \text{ValStamp } v \ ts$

definition *isVote* **where** *isVote m* $\equiv \exists v. m = \text{Vote } v$

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 \equiv
mru-vote st = None
 \wedge *commt st = None*
 \wedge *decide st = None*

definition *mru-vote-to-msg* :: *'val pstate* \Rightarrow *'val msg* **where**
mru-vote-to-msg st \equiv *case mru-vote st of*
Some (ts, v) \Rightarrow ValStamp v ts
| *None \Rightarrow NeverVoted*

fun *msg-to-val-stamp* :: *'val msg* \Rightarrow (*round* \times *'val*)*option* **where**
msg-to-val-stamp (ValStamp v ts) = Some (ts, v)
| *msg-to-val-stamp - = None*

definition *msgs-to-lvs* ::

(*process* \rightarrow 'val *msg*)
 \Rightarrow (*process*, *round* \times 'val) *map*

where

msgs-to-lvs msgs \equiv *msg-to-val-stamp* \circ_m *msgs*

definition *send0* **where**

send0 r p q st \equiv
if *q* = *coord r* then *mru-vote-to-msg st* else *Null*

definition *next0*

:: *nat*
 \Rightarrow *process*
 \Rightarrow 'val *pstate*
 \Rightarrow (*process* \rightarrow 'val *msg*)
 \Rightarrow *process*
 \Rightarrow 'val *pstate*
 \Rightarrow *bool*

where

next0 r p st msgs crd st' \equiv let *Q* = *dom msgs*; *lvs* = *msgs-to-lvs msgs* in
if *p* = *coord r* \wedge *card Q* > *N div 2*
then (*st'* = *st* (\downarrow *commt* := *Some* (*case-option* (*x st*) *snd* (*option-Max-by fst*
(*ran* (*lvs* | ' *Q*)))))) (\downarrow)
else *st'* = *st* (\downarrow *commt* := *None*)

definition *send1* **where**

send1 r p q st \equiv
if *p* = *coord r* \wedge *commt st* \neq *None* then *Vote* (*the* (*commt st*)) else *Null*

definition *next1*

:: *nat*
 \Rightarrow *process*
 \Rightarrow 'val *pstate*
 \Rightarrow (*process* \rightarrow 'val *msg*)
 \Rightarrow *process*
 \Rightarrow 'val *pstate*
 \Rightarrow *bool*

where

next1 r p st msgs crd st' \equiv
if *msgs* (*coord r*) \neq *None* \wedge *isVote* (*the* (*msgs* (*coord r*)))

then $st' = st \ () \ mru\text{-vote} := \text{Some} (\text{three-phase } r, \text{val } (\text{the } (\text{msgs } (\text{coord } r)))) \ ()$
 else $st' = st$

definition *send2* **where**

$send2 \ r \ p \ q \ st \equiv (\text{case } mru\text{-vote } st \text{ of}$
 $\quad \text{Some } (phs, v) \Rightarrow (\text{if } phs = \text{three-phase } r \text{ then } \text{Vote } v \text{ else } \text{Null})$
 $\quad | _ \Rightarrow \text{Null}$
 $\quad)$

— processes from which a vote was received

definition *votes-rcvd* **where**

$votes\text{-rcvd} \ (msgs :: \text{process} \rightarrow 'val \ \text{msg}) \equiv$
 $\{ (q, v) . msgs \ q = \text{Some} (\text{Vote } v) \}$

definition *the-rcvd-vote* **where**

$the\text{-rcvd-vote} \ (msgs :: \text{process} \rightarrow 'val \ \text{msg}) \equiv \text{SOME } v. v \in \text{snd } ' \ votes\text{-rcvd } msgs$

definition *next2* **where**

$next2 \ r \ p \ st \ msgs \ crd \ st' \equiv$
 $\quad \text{if } \text{card } (\text{votes-rcvd } msgs) > N \ \text{div } 2$
 $\quad \text{then } st' = st \ () \ \text{decide} := \text{Some} (\text{the-rcvd-vote } msgs) \ ()$
 $\quad \text{else } st' = st$

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

definition *Paxos-sendMsg* $:: \text{nat} \Rightarrow \text{process} \Rightarrow \text{process} \Rightarrow 'val \ \text{pstate} \Rightarrow 'val \ \text{msg}$
where

$Paxos\text{-sendMsg} \ (r :: \text{nat}) \equiv$
 $\quad \text{if } \text{three-step } r = 0 \text{ then } send0 \ r$
 $\quad \text{else if } \text{three-step } r = 1 \text{ then } send1 \ r$
 $\quad \text{else } send2 \ r$

definition

$Paxos\text{-nextState} :: \text{nat} \Rightarrow \text{process} \Rightarrow 'val \ \text{pstate} \Rightarrow (\text{process} \rightarrow 'val \ \text{msg})$
 $\quad \Rightarrow \text{process} \Rightarrow 'val \ \text{pstate} \Rightarrow \text{bool}$

where

$Paxos\text{-nextState} \ r \equiv$
 $\quad \text{if } \text{three-step } r = 0 \text{ then } next0 \ r$
 $\quad \text{else if } \text{three-step } r = 1 \text{ then } next1 \ r$

else next2 r

definition

Paxos-commPerRd **where**

Paxos-commPerRd *r* (*HO::process HO*) (*crd::process coord*) \equiv *True*

definition

Paxos-commGlobal **where**

Paxos-commGlobal *HOs coords* \equiv

\exists *ph::nat*. \exists *c::process*.

coord (*nr-steps*ph*) = *c*

\wedge *card* (*HOs* (*nr-steps*ph*) *c*) > *N div 2*

\wedge (\forall *p*. *c* \in *HOs* (*nr-steps*ph+1*) *p*)

\wedge (\forall *p*. *card* (*HOs* (*nr-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 \equiv

(*CinitState* = *Paxos-initState*,

sendMsg = *Paxos-sendMsg*,

CnextState = *Paxos-nextState*)

definition *Paxos-CHOMachine* **where**

Paxos-CHOMachine \equiv

(*CinitState* = *Paxos-initState*,

sendMsg = *Paxos-sendMsg*,

CnextState = *Paxos-nextState*,

CHOcommPerRd = *Paxos-commPerRd*,

CHOcommGlobal = *Paxos-commGlobal*)

abbreviation

Paxos-M \equiv (*Paxos-CHOMachine::(process, 'val pstate, 'val msg)* *CHOMachine*)

end

18.3 Proofs

type-synonym $p\text{-TS}\text{-state} = (\text{nat} \times (\text{process} \Rightarrow (\text{val } p\text{state})))$

definition $Paxos\text{-TS} ::$

$(\text{round} \Rightarrow \text{process } HO)$
 $\Rightarrow (\text{round} \Rightarrow \text{process } HO)$
 $\Rightarrow (\text{round} \Rightarrow \text{process})$
 $\Rightarrow p\text{-TS}\text{-state } TS$

where

$Paxos\text{-TS } HOs \ SHOs \ crds = CHO\text{-to}\text{-TS } Paxos\text{-Alg } HOs \ SHOs \ (K \circ crds)$

lemmas $Paxos\text{-TS}\text{-defs} = Paxos\text{-TS}\text{-def } CHO\text{-to}\text{-TS}\text{-def } Paxos\text{-Alg}\text{-def } CHO\text{init}\text{-Config}\text{-def}$

$Paxos\text{-initState}\text{-def}$

definition $Paxos\text{-trans}\text{-step}$ **where**

$Paxos\text{-trans}\text{-step } HOs \ SHOs \ crds \ next\text{-f} \ snd\text{-f} \ stp \equiv \bigcup r \ \mu.$
 $\{((r, \text{cfg}), (\text{Suc } r, \text{cfg}')) \mid \text{cfg } \text{cfg}'. \text{three}\text{-step } r = stp \wedge (\forall p.$
 $\mu \ p \in \text{get}\text{-msgs } (\text{snd}\text{-f } r) \ \text{cfg} \ (HOs \ r) \ (SHOs \ r) \ p$
 $\wedge \text{next}\text{-f } r \ p \ (\text{cfg } p) \ (\mu \ p) \ (\text{crds } r) \ (\text{cfg}' \ p)$
 $)\}$

lemma $\text{three}\text{-step}\text{-less}\text{-D}$:

$0 < \text{three}\text{-step } r \implies \text{three}\text{-step } r = 1 \vee \text{three}\text{-step } r = 2$

by($\text{unfold } \text{three}\text{-step}\text{-def}, \text{arith}$)

lemma $Paxos\text{-trans}$:

$CSHO\text{-trans}\text{-alt } Paxos\text{-sendMsg } Paxos\text{-nextState } HOs \ SHOs \ (K \circ crds) =$

$Paxos\text{-trans}\text{-step } HOs \ SHOs \ crds \ next0 \ send0 \ 0$

$\cup Paxos\text{-trans}\text{-step } HOs \ SHOs \ crds \ next1 \ send1 \ 1$

$\cup Paxos\text{-trans}\text{-step } HOs \ SHOs \ crds \ next2 \ send2 \ 2$

proof($\text{rule } \text{equalityI}$)

show $CSHO\text{-trans}\text{-alt } Paxos\text{-sendMsg } Paxos\text{-nextState } HOs \ SHOs \ (K \circ crds)$

$\subseteq Paxos\text{-trans}\text{-step } HOs \ SHOs \ crds \ next0 \ send0 \ 0 \cup$

$Paxos\text{-trans}\text{-step } HOs \ SHOs \ crds \ next1 \ send1 \ 1 \cup$

$Paxos\text{-trans}\text{-step } HOs \ SHOs \ crds \ next2 \ send2 \ 2$

by($\text{force } \text{simp } \text{add}: CSHO\text{-trans}\text{-alt}\text{-def } Paxos\text{-sendMsg}\text{-def } Paxos\text{-nextState}\text{-def}$

$Paxos\text{-trans}\text{-step}\text{-def } K\text{-def } \text{dest}!: \text{three}\text{-step}\text{-less}\text{-D}$)

next

show *Paxos-trans-step HOs SHOs crds next0 send0 0* \cup
Paxos-trans-step HOs SHOs crds next1 send1 1 \cup
Paxos-trans-step HOs SHOs crds next2 send2 2
 \subseteq *CSHO-trans-alt Paxos-sendMsg Paxos-nextState HOs SHOs* ($K \circ crds$)
by(*force simp add: CSHO-trans-alt-def Paxos-sendMsg-def Paxos-nextState-def*
Paxos-trans-step-def K-def)

qed

type-synonym $rHO = nat \Rightarrow process\ HO$

18.3.1 Refinement

definition *coord-vote-to-set* $:: nat \Rightarrow (process \Rightarrow (val\ pstate)) \Rightarrow val\ set$ **where**
coord-vote-to-set r sc $\equiv (let\ v = pstate.commt\ (sc\ (coord\ r))\ in$
if $v = None$
then $\{\}$
else $\{the\ v\}$)

definition *paxos-ref-rel* $:: (three-step-mru-state \times p-TS-state)set$ **where**
paxos-ref-rel $= \{(sa, (r, sc))$.
opt-mru-state.next-round sa = r
 \wedge *opt-mru-state.decisions sa = pstate.decide o sc*
 \wedge *opt-mru-state.mru-vote sa = pstate.mru-vote o sc*
 \wedge (*three-step r = Suc 0* \longrightarrow *three-step-mru-state.candidates sa = coord-vote-to-set*
r sc)
 $\}$

lemma *mru-vote-evolution0*:

$\forall p. next0\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow mru-vote\ o\ s' = mru-vote\ o\ s$
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*:

$\forall p. next2\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow mru-vote\ o\ s' = mru-vote\ o\ s$
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*:

$\forall p. next0\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow decide\ o\ s = decide\ o\ s'$

$\forall p. \text{next1 } r \ p \ (s \ p) \ (\text{msgs } p) \ (\text{crd } p) \ (s' \ p) \implies \text{decide } o \ s = \text{decide } o \ s'$
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
 $\mu \ (\text{coord } r) \in \text{get-msgs } (\text{send0 } r) \ \text{cfg} \ (\text{HOs } r) \ (\text{HOs } r) \ (\text{coord } r) \ (\text{is } \mu \ ?p \in -)$
shows $((\text{msgs-to-lvs } (\mu \ ?p)) \mid \text{HOs } r \ ?p) = (\text{mru-vote } o \ \text{cfg}) \mid \text{HOs } r \ ?p$ **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*:

$\{\text{paxos-ref-rel}\}$
 $(\bigcup r \ C. \text{majorities.opt-mru-step0 } r \ C),$
 $\text{Paxos-trans-step } \text{HOs } \text{HOs } \text{crds } \text{next0 } \text{send0 } 0 \ \{> \ \text{paxos-ref-rel}\}$
proof(*clarsimp simp add: PO-rhoare-defs Paxos-trans-step-def all-conj-distrib*)
fix *r sa sc sc' μ*
assume *R*: $(sa, (r, sc)) \in \text{paxos-ref-rel}$
and *r*: *three-step* *r* = 0
and μ : $\forall p. \mu \ p \in \text{get-msgs } (\text{send0 } r) \ sc \ (\text{HOs } r) \ (\text{HOs } r) \ p$
and *next*: $\forall p. \text{next0 } r \ p \ (sc \ p) \ (\mu \ p) \ (\text{crds } r) \ (sc' \ p)$
note $\mu \ \text{next} = \mu \ \text{next}$
from *r* **have** *same-coord*: $\text{coord } (\text{Suc } r) = \text{coord } r$
by(*auto simp add: three-step-phase-Suc intro: coord-phase*)
define *C* **where** *C* = *coord-vote-to-set* $(\text{Suc } r) \ sc'$
have *guard*: $\forall \text{cand} \in C. \exists Q. \text{majorities.opt-mru-guard } (\text{mru-vote } o \ sc) \ Q \ \text{cand}$
proof
fix *cand*
assume *cand*: *cand* $\in C$
hence *Some*: *commt* $(sc' \ (\text{coord } r)) = \text{Some } \text{cand}$ **using** *next*[*THEN spec*],
where $x = \text{coord } r$
by(*auto simp add: C-def coord-vote-to-set-def Let-def same-coord*)

let $?Q = \text{HOs } r \ (\text{coord } r)$
let $?lvs0 = \text{mru-vote } o \ sc$

have $?Q \in \text{maj}$ **using** *Some* $\mu \ \text{next}$ [*THEN spec*], **where** $x = \text{coord } r$
by(*auto simp add: Let-def maj-def next0-def same-coord get-msgs-dom*)
moreover have

$map-option\ snd\ (option-Max-by\ fst\ (ran\ (?lvs\ |\ ' ?Q))) \in \{None, Some\ cand\}$
using $Some\ nxt[THEN\ spec, \mathbf{where}\ x=coord\ r]$
 $msgs-mru-vote[\mathbf{where}\ HOs=HOs\ \mathbf{and}\ \mu=\mu, OF\ \mu[THEN\ spec, \mathbf{where}\ x=coord\ r]]$
 $get-msgs-dom[OF\ \mu[THEN\ spec, of\ coord\ r]]$
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 $\exists Q. majorities.opt-mru-guard\ ?lvs0\ Q\ cand$
by blast
qed

define sa' **where** $sa' = sa[$
 $next-round := Suc\ r,$
 $candidates := C$
 $]$
have $(sa, sa') \in majorities.opt-mru-step0\ r\ C$ **using** $R\ r\ nxt\ guard$
by($auto\ simp\ add: majorities.opt-mru-step0-def\ sa'-def\ paxos-ref-rel-def$)
moreover have $(sa', (Suc\ r, sc')) \in paxos-ref-rel$ **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
 $\exists sa'. (\exists r\ C. (sa, sa') \in majorities.opt-mru-step0\ r\ C)$
 $\wedge (sa', Suc\ r, sc') \in paxos-ref-rel$
by blast
qed

lemma $step1-ref$:
 $\{paxos-ref-rel\}$
 $(\bigcup r\ S\ v. majorities.opt-mru-step1\ r\ S\ v),$
 $Paxos-trans-step\ HOs\ HOs\ crds\ next1\ send1\ (Suc\ 0)\ \{>\ paxos-ref-rel\}$
proof($clarsimp\ simp\ add: PO-rhoare-defs\ Paxos-trans-step-def\ all-conj-distrib$)
fix $r\ sa\ sc\ sc'\ \mu$
assume $R: (sa, (r, sc)) \in paxos-ref-rel$
and $r: three-step\ r = Suc\ 0$
and $\mu: \forall p. \mu\ p \in get-msgs\ (send1\ r)\ sc\ (HOs\ r)\ (HOs\ r)\ p$
and $nxt: \forall p. next1\ r\ p\ (sc\ p)\ (\mu\ p)\ (crds\ r)\ (sc'\ p)$
note $\mu\ next = \mu\ nxt$


```

define  $v$  where  $v = \text{the } (\text{commt } (\text{sc } (\text{coord } r)))$ 
define  $S$  where  $S = \{p. \text{coord } r \in \text{HOs } r \ p \wedge \text{commt } (\text{sc } (\text{coord } r)) \neq \text{None}\}$ 
define  $sa'$  where  $sa' = sa \lfloor \text{next-round} := \text{Suc } r,$ 
   $\text{opt-mru-state.mru-vote} := \text{opt-mru-state.mru-vote } sa \ ++ \ \text{const-map } (\text{three-phase}$ 
 $r, v) \ S$ 
   $\rfloor$ 
have  $(sa, sa') \in \text{majorities.opt-mru-step1 } r \ S \ v$  using  $r \ R$ 
  by( $\text{clarsimp simp add: majorities.opt-mru-step1-def } sa'\text{-def } S\text{-def } v\text{-def}$ 
   $\text{coord-vote-to-set-def paxos-ref-rel-def}$ )
moreover have  $(sa', (\text{Suc } r, sc')) \in \text{paxos-ref-rel}$  using  $r \ R$ 
proof–
  have  $\text{mru-vote } o \ sc' = ((\text{mru-vote } \circ \ sc) \ ++ \ \text{const-map } (\text{three-phase } r, v) \ S)$ 
  proof( $\text{rule ext, simp}$ )
    fix  $p$ 
    show  $\text{mru-vote } (sc' \ p) = ((\text{mru-vote } \circ \ sc) \ ++ \ \text{const-map } (\text{three-phase } r, v)$ 
 $S) \ p$ 
    using  $\mu\text{next}[\text{THEN } \text{spec, of } p]$ 
    by( $\text{auto simp add: get-msgs-benign next1-def send1-def } S\text{-def } v\text{-def map-add-def}$ 
   $\text{const-map-is-None const-map-is-Some restrict-map-def isVote-def split:}$ 
 $\text{option.split}$ )
    qed
  thus  $?thesis$  using  $R \ r \ \text{next}$ 
  by( $\text{force simp add: paxos-ref-rel-def } sa'\text{-def three-step-Suc intro: decide-evolution}$ )
  qed
ultimately show
   $\exists sa'. (\exists r \ S \ v. (sa, sa') \in \text{majorities.opt-mru-step1 } r \ S \ v)$ 
   $\wedge (sa', \text{Suc } r, sc') \in \text{paxos-ref-rel}$ 
  by blast
qed

lemma step2-ref:
   $\{\text{paxos-ref-rel}\}$ 
   $(\bigcup r \ \text{dec-f. majorities.opt-mru-step2 } r \ \text{dec-f}),$ 
   $\text{Paxos-trans-step HOs HOs crds next2 send2 } 2 \ \{> \ \text{paxos-ref-rel}\}$ 
proof( $\text{clarsimp simp add: PO-rhoare-defs Paxos-trans-step-def all-conj-distrib}$ )
  fix  $r \ sa \ sc \ sc' \ \mu$ 
  assume  $R: (sa, (r, sc)) \in \text{paxos-ref-rel}$ 
  and  $r: \text{three-step } r = 2$ 

```

```

and  $\mu$ :  $\forall p. \mu p \in \text{get-msgs } (\text{send2 } r) \text{ } sc \text{ } (HOs \ r) \text{ } (HOs \ r) \ p$ 
and  $\text{next}$ :  $\forall p. \text{next2 } r \ p \ (sc \ p) \ (\mu \ p) \ (\text{crds } r) \ (sc' \ p)$ 
note  $\mu\text{next} = \mu \ \text{next}$ 

define  $\text{dec-f}$ 
  where  $\text{dec-f } p = (\text{if } \text{decide } (sc' \ p) \neq \text{decide } (sc \ p) \ \text{then } \text{decide } (sc' \ p) \ \text{else } \text{None})$  for  $p$ 

have  $\text{dec-f}: (\text{decide} \circ sc) ++ \text{dec-f} = \text{decide} \circ sc'$ 
proof
  fix  $p$ 
  show  $((\text{decide} \circ sc) ++ \text{dec-f}) \ p = (\text{decide} \circ sc') \ p$  using  $\text{next}[\text{THEN } \text{spec}, \text{ of } p]$ 
  by( $\text{auto simp add: map-add-def dec-f-def next2-def split: option.split intro!:$ 
 $\text{ext}$ )
qed

define  $sa'$  where  $sa' = sa[$ 
   $\text{next-round} := \text{Suc } r,$ 
   $\text{decisions} := \text{decisions } sa ++ \text{dec-f}$ 
 $]$ 

have  $(sa', (\text{Suc } r, sc')) \in \text{paxos-ref-rel}$  using  $R \ r \ \text{next}$ 
by( $\text{auto simp add: paxos-ref-rel-def sa'-def dec-f three-step-Suc}$ 
 $\text{mru-vote-evolution2}[OF \ \text{next}]$ )
moreover have  $(sa, sa') \in \text{majorities.opt-mru-step2 } r \ \text{dec-f}$  using  $r \ R$ 
proof–
  define  $sc\text{-}r\text{-votes}$  where  $sc\text{-}r\text{-votes } p = (\text{if } (\exists v. \text{mru-vote } (sc \ p) = \text{Some}$ 
 $(\text{three-phase } r, v))$ 
   $\text{then } \text{map-option } \text{snd } (\text{mru-vote } (sc \ p))$ 
   $\text{else } \text{None})$  for  $p$ 
  have  $sc\text{-}r\text{-votes}: sc\text{-}r\text{-votes} = \text{majorities.r-votes } sa \ r$  using  $R \ r$ 
  by( $\text{auto simp add: paxos-ref-rel-def sc-r-votes-def majorities.r-votes-def intro!:$ 
 $\text{ext}$ )
  have  $\text{majorities.step2-d-guard } \text{dec-f } sc\text{-}r\text{-votes}$ 
proof( $\text{clarsimp simp add: majorities.step2-d-guard-def}$ )
  fix  $p \ v$ 
  assume  $d\text{-}f\text{-}p: \text{dec-f } p = \text{Some } v$ 
  let  $?Qv = \text{votes-rcvd } (\mu \ p)$ 
  have  $Qv: \text{card } ?Qv > N \ \text{div } 2$ 

```

$v = \text{the-rcvd-vote } (\mu p) \text{ using } \text{nxt}[\text{THEN spec, of } p] \text{ d-f-p}$
by(*auto simp add: next2-def dec-f-def*)

hence $v \in \text{snd } \text{' votes-rcvd } (\mu p)$

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 $?Qv = \text{map-graph } (\text{sc-r-votes}) \cap (\text{HOs } r p \times \text{UNIV})$ **using**
 $\mu[\text{THEN spec, of } p]$

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 $v \in \text{ran } \text{sc-r-votes} \wedge \text{dom } \text{sc-r-votes} \in \text{majS}$ **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** $f = \text{fst}$])

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

$\exists sa'. (\exists r \text{ dec-f. } (sa, sa') \in \text{majorities.opt-mru-step2 } r \text{ dec-f})$

$\wedge (sa', \text{Suc } r, sc') \in \text{paxos-ref-rel}$

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 $\text{init } (\text{Paxos-TS HOs HOs crds}) \subseteq \text{paxos-ref-rel}$ “ *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

$\{\text{paxos-ref-rel}\} \text{TS.trans } \text{majorities.ts-mru-TS},$

$\text{TS.trans } (\text{Paxos-TS HOs HOs crds}) \{> \text{paxos-ref-rel}\}$

```

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:  $\forall r. CHOcommPerRd Paxos-M r (HOs r) (crds r)$ 
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
 $\wedge$  card (HOs (nr-steps*ph) c) > N div 2
 $\wedge$  ( $\forall p. c \in HOs (nr-steps*ph+1) p$ )
 $\wedge$  ( $\forall p. card (HOs (nr-steps*ph+2) p) > N div 2$ )

```

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 $\mu0 \mu1 \mu2$ **where**

send0: $\forall p. \mu0 p \in get-msgs (send0 r0) cfg0 (HOs r0) (HOs r0) p$

and *three-step0*: $\forall p. next0 r0 p (cfg0 p) (\mu0 p) (crds (Suc r0) p) (cfg1 p)$

and *send1*: $\forall p. \mu1\ p \in \text{get-msgs } (\text{send1 } r1)\ \text{cfg1 } (\text{HOs } r1)\ (\text{HOs } r1)\ p$
and *three-step1*: $\forall p. \text{next1 } r1\ p\ (\text{cfg1 } p)\ (\mu1\ p)\ (\text{crds } (\text{Suc } r1)\ p)\ (\text{cfg2 } p)$
and *send2*: $\forall p. \mu2\ p \in \text{get-msgs } (\text{send2 } r2)\ \text{cfg2 } (\text{HOs } r2)\ (\text{HOs } r2)\ p$
and *three-step2*: $\forall p. \text{next2 } r2\ p\ (\text{cfg2 } p)\ (\mu2\ p)\ (\text{crds } (\text{Suc } r2)\ p)\ (\text{cfg3 } p)$
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: $\forall p. \text{mru-vote } (\text{cfg2 } p) = \text{Some } (\text{three-phase } r2, \text{dec-v})$*
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: $\forall p. \text{decide } (\text{cfg3 } p) = \text{Some } \text{dec-v}$*
proof
fix *p*

```

have votes-rcvd ( $\mu 2 p$ ) = 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

$mru-vote :: (nat \times 'val)$ 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  $\Rightarrow$  process
specification (coord)
  coord-phase[rule-format]:  $\forall r r'. \text{three-phase } r = \text{three-phase } r' \longrightarrow \text{coord } r = \text{coord } r'$ 
  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  $\equiv \exists v ts. m = \text{ValStamp } v ts$ 
```

```
definition isVote where isVote m  $\equiv \exists v. m = \text{Vote } v$ 
```

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-initState p st crd  $\equiv$ 
  mru-vote st = None
   $\wedge$  decide st = None
```

```
definition mru-vote-to-msg :: 'val pstate  $\Rightarrow$  'val msg where
  mru-vote-to-msg st  $\equiv$  case mru-vote st of
  Some (ts, v)  $\Rightarrow$  ValStamp v ts
| None  $\Rightarrow$  NeverVoted
```

fun *msg-to-val-stamp* :: 'val msg \Rightarrow (round \times 'val)option **where**
msg-to-val-stamp (ValStamp v ts) = Some (ts, v)
| *msg-to-val-stamp* - = None

definition *msgs-to-lvs* ::
(process \rightarrow 'val msg)
 \Rightarrow (process, round \times 'val) map

where
msgs-to-lvs msgs \equiv *msg-to-val-stamp* \circ_m msgs

definition *send0* **where**
send0 r p q st \equiv
if q = coord r then *mru-vote-to-msg* st else Null

definition *next0*
:: nat
 \Rightarrow process
 \Rightarrow 'val pstate
 \Rightarrow (process \rightarrow 'val msg)
 \Rightarrow process
 \Rightarrow 'val pstate
 \Rightarrow bool

where
next0 r p st msgs crd st' \equiv let Q = dom msgs; lvs = *msgs-to-lvs* msgs in
if p = coord r
then (st' = st \parallel commt := (case-option (x st) snd (option-Max-by fst (ran
(lvs | ' Q)))) \parallel)
else st' = st

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

definition *next1*
:: nat
 \Rightarrow process
 \Rightarrow 'val pstate
 \Rightarrow (process \rightarrow 'val msg)
 \Rightarrow process
 \Rightarrow 'val pstate

$\Rightarrow \text{bool}$

where

$\text{next1 } r \ p \ st \ msgs \ \text{crd} \ st' \equiv$
 $\text{if } msgs \ (\text{coord } r) \neq \text{None}$
 $\text{then } st' = st \ (\mid \ \text{mru-vote} := \text{Some } (\text{three-phase } r, \ \text{val } (\text{the } (msgs \ (\text{coord } r)))) \ \mid)$
 $\text{else } st' = st$

definition send2 where

$\text{send2 } r \ p \ q \ st \equiv$ (case $\text{mru-vote } st$ of
 $\text{Some } (phs, v) \Rightarrow$ (if $phs = \text{three-phase } r$ then $\text{Vote } v$ else Null)
 $\mid - \Rightarrow \text{Null}$
)

— processes from which a vote was received

definition votes-rcvd where

$\text{votes-rcvd } (msgs :: \text{process} \rightarrow 'val \ \text{msg}) \equiv$
 $\{ (q, v) . msgs \ q = \text{Some } (\text{Vote } v) \}$

definition the-rcvd-vote where

$\text{the-rcvd-vote } (msgs :: \text{process} \rightarrow 'val \ \text{msg}) \equiv \text{SOME } v. v \in \text{snd } ' \ \text{votes-rcvd } msgs$

definition next2 where

$\text{next2 } r \ p \ st \ msgs \ \text{crd} \ st' \equiv$
 $\text{if } \text{card } (\text{votes-rcvd } msgs) > N \ \text{div } 2$
 $\text{then } st' = st \ (\mid \ \text{decide} := \text{Some } (\text{the-rcvd-vote } msgs) \ \mid)$
 $\text{else } st' = st$

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

definition CT-sendMsg :: nat \Rightarrow process \Rightarrow process \Rightarrow 'val pstate \Rightarrow 'val msg where

$\text{CT-sendMsg } (r :: \text{nat}) \equiv$
 $\text{if three-step } r = 0 \text{ then } \text{send0 } r$
 $\text{else if three-step } r = 1 \text{ then } \text{send1 } r$
 $\text{else } \text{send2 } r$

definition

$\text{CT-nextState} :: \text{nat} \Rightarrow \text{process} \Rightarrow 'val \ \text{pstate} \Rightarrow (\text{process} \rightarrow 'val \ \text{msg})$
 $\Rightarrow \text{process} \Rightarrow 'val \ \text{pstate} \Rightarrow \text{bool}$

where

$CT\text{-nextState } r \equiv$
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**

$CT\text{-Alg} \equiv$
($C\text{initState} = CT\text{-initState}$,
 $sendMsg = CT\text{-sendMsg}$,
 $C\text{nextState} = CT\text{-nextState}$)

The *CT* algorithm relies on *waiting*: in each round, the coordinator waits until it hears from $\frac{N}{2}$ processes. This is 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-step $r = 0 \longrightarrow card\ (HOs\ (coord\ r)) > N\ div\ 2$

definition

CT-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-CHOMachine* **where**

$CT\text{-CHOMachine} \equiv$
($C\text{initState} = CT\text{-initState}$,
 $sendMsg = CT\text{-sendMsg}$,
 $C\text{nextState} = CT\text{-nextState}$,
 $CHOcommPerRd = CT\text{-commPerRd}$,

$CHOcommGlobal = CT-commGlobal \}$

abbreviation

$CT-M \equiv (CT-CHOMachine::(process, 'val pstate, 'val msg) CHOMachine)$

end

19.2 Proofs

type-synonym $ct-TS-state = (nat \times (process \Rightarrow (val pstate)))$

definition $CT-TS ::$

$(round \Rightarrow process HO)$
 $\Rightarrow (round \Rightarrow process HO)$
 $\Rightarrow (round \Rightarrow process \Rightarrow process)$
 $\Rightarrow ct-TS-state TS$

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**

$CT-trans-step HOs SHOs crds next-f snd-f stp \equiv \bigcup r \mu.$
 $\{((r, cfg), (Suc r, cfg')) | cfg \text{ cfg}'. \text{three-step } r = stp \wedge (\forall p.$
 $\mu p \in \text{get-msgs } (snd-f r) \text{ } cfg \text{ } (HOs r) \text{ } (SHOs r) \text{ } p$
 $\wedge \text{next-f } r \text{ } p \text{ } (cfg \text{ } p) \text{ } (\mu \text{ } p) \text{ } (crds \text{ } r \text{ } p) \text{ } (cfg' \text{ } p)$
 $)\}$

lemma $three-step-less-D:$

$0 < \text{three-step } r \implies \text{three-step } r = 1 \vee \text{three-step } r = 2$
by($\text{unfold } \text{three-step-def}, \text{arith}$)

lemma $CT-trans:$

$CCHO-trans-alt CT-sendMsg CT-nextState HOs SHOs crds =$
 $CT-trans-step HOs SHOs crds next0 send0 0$
 $\cup CT-trans-step HOs SHOs crds next1 send1 1$
 $\cup CT-trans-step HOs SHOs crds next2 send2 2$

proof(rule equalityI)

show *CSHO-trans-alt CT-sendMsg CT-nextState HOs SHOs crds*
 \subseteq *CT-trans-step HOs SHOs crds next0 send0 0* \cup
CT-trans-step HOs SHOs crds next1 send1 1 \cup
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* \cup
CT-trans-step HOs SHOs crds next1 send1 1 \cup
CT-trans-step HOs SHOs crds next2 send2 2
 \subseteq *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* \times *ct-TS-state*)*set* **where**
ct-ref-rel = $\{(sa, (r, sc))$.
 $opt\text{-}mru\text{-}state.next\text{-}round\ sa = r$
 $\wedge\ opt\text{-}mru\text{-}state.decisions\ sa = pstate.decide\ o\ sc$
 $\wedge\ opt\text{-}mru\text{-}state.mru\text{-}vote\ sa = pstate.mru\text{-}vote\ o\ sc$
 $\wedge\ (three\text{-}step\ r = Suc\ 0 \longrightarrow three\text{-}step\text{-}mru\text{-}state.candidates\ sa = \{commt\ (sc$
 $(coord\ r))\})$
 $\}$

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*:

$\forall p. next0\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \implies mru\text{-}vote\ o\ s' = mru\text{-}vote\ o\ s$
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*:

$\forall p. next2\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \implies mru\text{-}vote\ o\ s' = mru\text{-}vote\ o\ s$
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*:

$\forall p. \text{next0 } r \ p \ (s \ p) \ (\text{msgs } p) \ (\text{crd } p) \ (s' \ p) \implies \text{decide } o \ s = \text{decide } o \ s'$
 $\forall p. \text{next1 } r \ p \ (s \ p) \ (\text{msgs } p) \ (\text{crd } p) \ (s' \ p) \implies \text{decide } o \ s = \text{decide } o \ s'$
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
 $\mu \ (coord \ r) \in \text{get-msgs } (send0 \ r) \ \text{cfg} \ (HOs \ r) \ (HOs \ r) \ (coord \ r) \ (\text{is } \mu \ ?p \in -)$
shows $((\text{msgs-to-lvs } (\mu \ ?p)) \mid 'HOs \ r \ ?p) = (\text{mru-vote } o \ \text{cfg}) \mid 'HOs \ r \ ?p$ **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

$HOs :: nat \Rightarrow process \Rightarrow process \ set$
and $crds :: nat \Rightarrow process \Rightarrow process$

assumes

per-rd: $\forall r. CT-commPerRd \ r \ (HOs \ r) \ (crds \ r)$

begin

lemma *step0-ref*:

$\{ct-ref-rel\}$
 $(\bigcup r \ C. \text{majorities.opt-mru-step0 } r \ C),$
 $CT-trans-step \ HOs \ HOs \ crds \ next0 \ send0 \ 0 \ \{> \ ct-ref-rel\}$

proof(*clarsimp simp add: PO-rhoare-defs CT-trans-step-def all-conj-distrib*)

fix $r \ sa \ sc \ sc' \ \mu$

assume $R: (sa, (r, sc)) \in ct-ref-rel$

and $r: \text{three-step } r = 0$

and $\mu: \forall p. \mu \ p \in \text{get-msgs } (send0 \ r) \ sc \ (HOs \ r) \ (HOs \ r) \ p$

and $nxt: \forall p. \text{next0 } r \ p \ (sc \ p) \ (\mu \ p) \ (crds \ r \ p) \ (sc' \ p)$

note $\mu nxt = \mu \ nxt$

have $r\text{-phase-step}: nr\text{-steps} * \text{three-phase } r = r$ **using** $r \ \text{three-phase-step}[of \ r]$

by(*auto*)

from r **have** $\text{same-coord}: coord \ (Suc \ r) = coord \ r$

by(*auto simp add: three-step-phase-Suc intro: coord-phase*)

define $cand$ **where** $cand = \text{commt } (sc' \ (coord \ (Suc \ r)))$

define C **where** $C = \{cand\}$

```

have guard:  $\forall cand \in C. \exists Q. majorities.opt-mru-guard (mru-vote \circ sc) Q cand$ 
proof(simp add: C-def)
  let ?Q = HOs r (coord r)
  let ?lvs0 = mru-vote o sc
  have ?Q  $\in$  majS using per-rd  $\mu$ next[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)))  $\in$  {None, Some cand}
    using next[THEN spec, where x=coord r]
    msgs-mru-vote[where HOs=HOs and  $\mu$ = $\mu$ , OF  $\mu$ [THEN spec, where
x=coord r]]
    get-msgs-dom[OF  $\mu$ [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  $\exists Q. majorities.opt-mru-guard ?lvs0 Q cand$ 
  by blast
qed

define sa' where sa' = sa(|
  next-round := Suc r,
  candidates := C
|)
have (sa, sa')  $\in$  majorities.opt-mru-step0 r C using R r next guard
  by(auto simp add: majorities.opt-mru-step0-def sa'-def ct-ref-rel-def)
moreover have (sa', (Suc r, sc'))  $\in$  ct-ref-rel using R next
  apply(auto simp add: sa'-def ct-ref-rel-def intro!:
mru-vote-evolution0[OF next, symmetric] decide-evolution(1)[OF next])
  apply(auto simp add: Let-def C-def cand-def o-def intro!: ext)
done
ultimately show
   $\exists sa'. (\exists r C. (sa, sa') \in majorities.opt-mru-step0 r C)$ 
   $\wedge (sa', Suc r, sc') \in ct-ref-rel$ 
  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 next: ∀ p. next1 r p (sc p) (μ p) (crds r p) (sc' p)
  note μnext = μ next

  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 o sc) ++ const-map (three-phase r, v) S)
  proof(rule ext, simp)
  fix p
  show mru-vote (sc' p) = ((mru-vote o sc) ++ const-map (three-phase r, v)
S) p
  using μnext[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 next
  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 next: ∀ p. next2 r p (sc p) (μ p) (crds r p) (sc' p)
  note μnext = μ next

  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 next[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 next
  by(auto simp add: ct-ref-rel-def sa'-def dec-f three-step-Suc
mru-vote-evolution2[OF next])
  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 next[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
   $\exists sa'. (\exists r \text{ dec-f. } (sa, sa') \in \text{majorities.opt-mru-step2 } r \text{ dec-f})$ 
   $\wedge (sa', \text{Suc } r, sc') \in \text{ct-ref-rel}$ 
  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)  $\subseteq$  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
       $\wedge (\forall p. c \in HOs (nr-steps*ph+1) p)$ 
       $\wedge (\forall p. card (HOs (nr-steps*ph+2) p) > N \text{ div } 2)$ 
  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 $\mu0 \mu1 \mu2$ **where**
send0: $\forall p. \mu0 p \in \text{get-msgs } (\text{send0 } r0) \text{ } cfg0 \text{ } (HOs \ r0) \text{ } (HOs \ r0) \text{ } p$
and *three-step0*: $\forall p. \text{next0 } r0 \text{ } p \text{ } (cfg0 \text{ } p) \text{ } (\mu0 \text{ } p) \text{ } (crds \ (Suc \ r0) \ p) \text{ } (cfg1 \text{ } p)$
and *send1*: $\forall p. \mu1 p \in \text{get-msgs } (\text{send1 } r1) \text{ } cfg1 \text{ } (HOs \ r1) \text{ } (HOs \ r1) \text{ } p$
and *three-step1*: $\forall p. \text{next1 } r1 \text{ } p \text{ } (cfg1 \text{ } p) \text{ } (\mu1 \text{ } p) \text{ } (crds \ (Suc \ r1) \ p) \text{ } (cfg2 \text{ } p)$
and *send2*: $\forall p. \mu2 p \in \text{get-msgs } (\text{send2 } r2) \text{ } cfg2 \text{ } (HOs \ r2) \text{ } (HOs \ r2) \text{ } p$
and *three-step2*: $\forall p. \text{next2 } r2 \text{ } p \text{ } (cfg2 \text{ } p) \text{ } (\mu2 \text{ } p) \text{ } (crds \ (Suc \ r2) \ p) \text{ } (cfg3 \text{ } p)$
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*: $\forall p. \text{mru-vote } (cfg2 \text{ } p) = \text{Some } (\text{three-phase } r2, \text{dec-v})$
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*: $\forall p. \text{decide } (cfg3 \text{ } p) = \text{Some } \text{dec-v}$

```

proof
  fix p
  have votes-rcvd ( $\mu 2$  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

```

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.