Abstract

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

Consensus Refined

March 17, 2025

Contents

1 Introduction

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

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

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

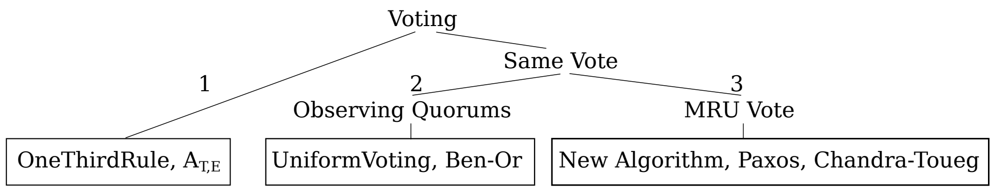

Figure 1: The consensus family tree. Boxes contain models of concrete algorithms.

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

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

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

2 Preliminaries

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

2.1 Prover configuration

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

2.2 Forward reasoning ("attributes")

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

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

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

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

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

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

2.3 General results

2.3.1 Maps

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

lemma *map-definedness*:
  $f\, x = Some\ y \Longrightarrow x \in dom\ f$
⟨*proof*⟩

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

⟨*proof*⟩

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

2.3.2 Set

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

lemma *vimage-image-subset*: $A \subseteq f-`(f`A)$
⟨*proof*⟩

2.3.3 Relations

lemma *Image-compose* [*simp*]:
  $(R1\ O\ R2)``A\ = R2``(R1``A)$
⟨*proof*⟩

2.3.4 Lists

lemma *map-id*: *map id* = *id*
⟨*proof*⟩
lemma *map-comp*: *map* (*g o f*) = *map g o map f*
⟨*proof*⟩

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

lemma *take-prefix*: $[\![\ take\ n\ l = xs\ ]\!] \Longrightarrow \exists\, xs'.\ l = xs\ @\ xs'$
⟨*proof*⟩

2.3.5 Finite sets

Cardinality.

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

lemma *finite-positive-cardI* [*intro*!]:
  $[\![\ A \neq \{\};\ finite\ A\ ]\!] \Longrightarrow 0 < card\ A$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *finite-zero-cardI* [*intro*!]:
$[\![A = \{\}; \textit{finite } A]\!] \Longrightarrow \textit{card } A = 0$
⟨*proof*⟩

lemma *finite-zero-cardD* [*dest*!]:
$[\![\textit{card } A = 0; \textit{finite } A]\!] \Longrightarrow A = \{\}$
⟨*proof*⟩

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* ⟨*proof*⟩

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

typedecl *val* — Type of values to choose from

type-synonym *round* = *nat*

end

2.5 Quorums

locale *quorum* =
fixes *Quorum* :: *'a set set*
assumes
qintersect: $[\![Q \in \textit{Quorum}; Q' \in \textit{Quorum}]\!] \Longrightarrow Q \cap Q' \neq \{\}$

— Non-emptiness needed for some invariants of Coordinated Voting
and *Quorum-not-empty*: $\exists\, Q.\ Q \in Quorum$

lemma (**in** *quorum*) *quorum-non-empty*: $Q \in Quorum \Longrightarrow Q \neq \{\}$
⟨*proof*⟩

lemma (**in** *quorum*) *empty-not-quorum*: $\{\} \in Quorum \Longrightarrow False$
⟨*proof*⟩

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

locale *mono-quorum* = *quorum-process* +
assumes *mono-quorum*: $[\![\ Q \in Quorum;\ Q \subseteq Q'\ ]\!] \Longrightarrow Q' \in Quorum$

lemma (**in** *mono-quorum*) *UNIV-quorum*:
$UNIV \in Quorum$
⟨*proof*⟩

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

lemma *majsI*:
$N\ div\ 2 < card\ S \Longrightarrow S \in majs$
⟨*proof*⟩

lemma *card-Compl*:
fixes S :: (*'a* :: *finite*) *set*
shows $card\ (-S) = card\ (UNIV :: 'a\ set) - card\ S$
⟨*proof*⟩

lemma *majorities-intersect*:
$card\ (Q :: process\ set) + card\ Q' > N \Longrightarrow Q \cap Q' \neq \{\}$
⟨*proof*⟩

interpretation *majorities*: *mono-quorum majs*
⟨*proof*⟩

end

2.6 Miscellaneous lemmas

$\langle ML \rangle$

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

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

2.7 Argmax

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

lemma *Max-by-dest*:
assumes $finite\ A$ **and** $A \neq \{\}$
shows $Max\text{-}by\ f\ A \in A \wedge f\ (Max\text{-}by\ f\ A) = Max\ (f\ `\ A)$ (**is** $?P\ (Max\text{-}by\ f\ A)$)
$\langle proof \rangle$

lemma *Max-by-in*:
assumes $finite\ A$ **and** $A \neq \{\}$
shows $Max\text{-}by\ f\ A \in A$ $\langle proof \rangle$

lemma *Max-by-ge*:
assumes $finite\ A\ x \in A$
shows $f\ x \leq f\ (Max\text{-}by\ f\ A)$
$\langle proof \rangle$

lemma *finite-UN-D*:
$finite\ (\bigcup S) \Longrightarrow \forall A \in S.\ finite\ A$
$\langle proof \rangle$

lemma *Max-by-eqI*:
assumes
fin: $finite\ A$
and $\bigwedge y.\ y \in A \Longrightarrow cmp\text{-}f\ y \leq cmp\text{-}f\ x$
and *in-X*: $x \in A$

and *inj*: *inj-on cmp-f A*
shows *Max-by cmp-f A = x*
⟨*proof*⟩

lemma *Max-by-Union-distrib*:
$[\![$ *finite A*; $A = \bigcup S$; $S \neq \{\}$; $\{\} \notin S$; *inj-on cmp-f A* $]\!] \Longrightarrow$
Max-by cmp-f A = Max-by cmp-f (Max-by cmp-f ‘ S)
⟨*proof*⟩

lemma *Max-by-UNION-distrib*:
$[\![$*finite A*; $A = (\bigcup x{\in}S.\ f\ x)$; $S \neq \{\}$; $\{\} \notin f$ ‘ S; *inj-on cmp-f A*$]\!] \Longrightarrow$
Max-by cmp-f A = Max-by cmp-f (Max-by cmp-f ‘ (f ‘ S))
⟨*proof*⟩

lemma *Max-by-eta*:
Max-by f = (λ*S*. (*SOME x*. $x \in S \wedge f\ x =$ *Max* (*f* ‘ *S*)))
⟨*proof*⟩

lemma *Max-is-Max-by-id*:
$[\![$ *finite S*; $S \neq \{\}$ $]\!] \Longrightarrow$ *Max S = Max-by id S*
⟨*proof*⟩

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

2.8 Function and map graphs

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

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

lemma *map-graph-mem*[*simp*]:
((*x*, *y*) ∈ *map-graph f*) = (*f x* = *Some y*)
⟨*proof*⟩

lemma *finite-fun-graph*:
finite A ⟹ *finite* (*fun-graph f* ∩ (*A* × *UNIV*))
⟨*proof*⟩

lemma *finite-map-graph*:
  $finite\ A \Longrightarrow finite\ (map\text{-}graph\ f \cap (A \times UNIV))$
  ⟨*proof*⟩

lemma *finite-dom-finite-map-graph*:
  $finite\ (dom\ f) \Longrightarrow finite\ (map\text{-}graph\ f)$
  ⟨*proof*⟩

lemma *ran-map-addD*:
  $x \in ran\ (m\ {+}{+}\ f) \Longrightarrow x \in ran\ m \vee x \in ran\ f$
  ⟨*proof*⟩

2.9 Constant maps

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

lemma *const-map-empty*[*simp*]:
  $const\text{-}map\ v\ \{\} = Map.empty$
  ⟨*proof*⟩

lemma *const-map-ran*[*simp*]: $x \in ran\ (const\text{-}map\ v\ S) = (S \neq \{\} \wedge x = v)$
  ⟨*proof*⟩

lemma *const-map-is-None*:
  $(const\text{-}map\ y\ A\ \ x = None) = (x \notin A)$
  ⟨*proof*⟩

lemma *const-map-is-Some*:
  $(const\text{-}map\ y\ A\ x = Some\ z) = (z = y \wedge x \in A)$
  ⟨*proof*⟩

lemma *const-map-in-set*:
  $x \in A \Longrightarrow const\text{-}map\ v\ A\ x = Some\ v$
  ⟨*proof*⟩

lemma *const-map-notin-set*:
  $x \notin A \Longrightarrow const\text{-}map\ v\ A\ x = None$

$\langle proof \rangle$

lemma *dom-const-map*:
dom (*const-map* *v* *S*) = *S*
$\langle proof \rangle$

2.10 Votes with maximum timestamps.

definition *vote-set* :: ('*round* ⇒ ('*process*, '*val*)*map*) ⇒ '*process set* ⇒ ('*round* × '*val*)*set* **where**
vote-set vs Q ≡ {(*r*, *v*)|*a r v*. ((*r*, *a*), *v*) ∈ *map-graph* (*case-prod vs*) ∧ *a* ∈ *Q*}

lemma *inj-on-fst-vote-set*:
inj-on fst (*vote-set v-hist* {*p*})
$\langle proof \rangle$

lemma *finite-vote-set*:
assumes ∀ *r*'≥(*r* :: *nat*). *v-hist r*' = *Map.empty*
finite S
shows *finite* (*vote-set v-hist S*)
$\langle proof \rangle$

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

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

lemma *process-mru-is-None*:
(*process-mru v-f a* = *None*) = (*vote-set v-f* {*a*} = {})
$\langle proof \rangle$

lemma *process-mru-is-Some*:
(*process-mru v-f a* = *Some rv*) = (*vote-set v-f* {*a*} ≠ {} ∧ *rv* = *Max-by fst* (*vote-set v-f* {*a*}))
$\langle proof \rangle$

lemma *vote-set-upd*:
$\mathit{vote\text{-}set}\ (\mathit{v\text{-}hist}(r := \mathit{v\text{-}f}))\ \{p\} =$
$(\mathit{if}\ p \in \mathit{dom}\ \mathit{v\text{-}f}$
$\mathit{then}\ \mathit{insert}\ (r,\ \mathit{the}\ (\mathit{v\text{-}f}\ p))$
$\mathit{else}\ \mathit{id}$
$)$
$(\mathit{if}\ \mathit{v\text{-}hist}\ r\ p = \mathit{None}$
$\mathit{then}\ \mathit{vote\text{-}set}\ \mathit{v\text{-}hist}\ \{p\}$
$\mathit{else}\ \mathit{vote\text{-}set}\ \mathit{v\text{-}hist}\ \{p\} - \{(r,\ \mathit{the}\ (\mathit{v\text{-}hist}\ r\ p))\}$
$)$

⟨*proof*⟩

lemma *finite-vote-set-upd*:
$\mathit{finite}\ (\mathit{vote\text{-}set}\ \mathit{v\text{-}hist}\ \{a\}) \Longrightarrow$
$\mathit{finite}\ (\mathit{vote\text{-}set}\ (\mathit{v\text{-}hist}(r := \mathit{v\text{-}f}))\ \{a\})$
⟨*proof*⟩

lemma *vote-setD*:
$\mathit{rv} \in \mathit{vote\text{-}set}\ \mathit{v\text{-}f}\ \{a\} \Longrightarrow \mathit{v\text{-}f}\ (\mathit{fst}\ \mathit{rv})\ a = \mathit{Some}\ (\mathit{snd}\ \mathit{rv})$
⟨*proof*⟩

lemma *process-mru-new-votes*:
assumes
$\forall\, r' \geq (r :: \mathit{nat}).\ \mathit{v\text{-}hist}\ r' = \mathit{Map.empty}$
shows
$\mathit{process\text{-}mru}\ (\mathit{v\text{-}hist}(r := \mathit{v\text{-}f})) =$
$(\mathit{process\text{-}mru}\ \mathit{v\text{-}hist} \mathbin{++} (\lambda p.\ \mathit{map\text{-}option}\ (\mathit{Pair}\ r)\ (\mathit{v\text{-}f}\ p)))$
⟨*proof*⟩

end

2.11 Step definitions for 2-step algorithms

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

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

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

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

lemma *two-phase-step*: (*two-phase r* ∗ *2*) + *two-step r* = *r*
⟨*proof*⟩

lemma *two-step-phase-Suc*:
two-step r = *0* ⟹ *two-phase* (*Suc r*) = *two-phase r*
two-step r = *0* ⟹ *two-step* (*Suc r*) = *1*
two-step r = *0* ⟹ *two-phase* (*Suc* (*Suc r*)) = *Suc* (*two-phase r*)
two-step r = (*Suc 0*) ⟹ *two-phase* (*Suc r*) = *Suc* (*two-phase r*)
two-step r = (*Suc 0*) ⟹ *two-step* (*Suc r*) = *0*
⟨*proof*⟩

end

2.12 Step definitions for 3-step algorithms

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

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

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

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

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

lemma *three-phase-step*: (*three-phase r* ∗ *nr-steps*) + *three-step r* = *r*
⟨*proof*⟩

lemma *three-step-Suc*:
three-step r = *0* ⟹ *three-step* (*Suc* (*Suc r*)) = *2*
three-step r = *0* ⟹ *three-step* (*Suc r*) = *1*
three-step r = (*Suc 0*) ⟹ *three-step* (*Suc r*) = *2*

$\textit{three-step}\ r = (\textit{Suc}\ \textit{0}) \Longrightarrow \textit{three-step}\ (\textit{Suc}\ (\textit{Suc}\ r)) = \textit{0}$
$\textit{three-step}\ r = (\textit{Suc}\ (\textit{Suc}\ \textit{0})) \Longrightarrow \textit{three-step}\ ((\textit{Suc}\ r)) = \textit{0}$
⟨*proof*⟩

lemma *three-step-phase-Suc*:
$\textit{three-step}\ r = \textit{0} \Longrightarrow \textit{three-phase}\ (\textit{Suc}\ r) = \textit{three-phase}\ r$
$\textit{three-step}\ r = \textit{0} \Longrightarrow \textit{three-phase}\ (\textit{Suc}\ (\textit{Suc}\ r)) = \textit{three-phase}\ r$
$\textit{three-step}\ r = \textit{0} \Longrightarrow \textit{three-phase}\ (\textit{Suc}\ (\textit{Suc}\ (\textit{Suc}\ r))) = \textit{Suc}\ (\textit{three-phase}\ r)$
$\textit{three-step}\ r = (\textit{Suc}\ \textit{0}) \Longrightarrow \textit{three-phase}\ (\textit{Suc}\ r) = \textit{three-phase}\ r$
$\textit{three-step}\ r = (\textit{Suc}\ \textit{0}) \Longrightarrow \textit{three-phase}\ (\textit{Suc}\ (\textit{Suc}\ r)) = \textit{Suc}\ (\textit{three-phase}\ r)$
$\textit{three-step}\ r = (\textit{Suc}\ (\textit{Suc}\ \textit{0})) \Longrightarrow \textit{three-phase}\ (\textit{Suc}\ r) = \textit{Suc}\ (\textit{three-phase}\ r)$
⟨*proof*⟩

lemma *three-step2-phase-Suc*:
$\textit{three-step}\ r = \textit{2} \Longrightarrow (\textit{3} * (\textit{Suc}\ (\textit{three-phase}\ r)) - \textit{1}) = r$
⟨*proof*⟩

lemma *three-stepE*:
$[\![\ \textit{three-step}\ r = \textit{0} \Longrightarrow P;\ \textit{three-step}\ r = \textit{1} \Longrightarrow P;\ \textit{three-step}\ r = \textit{2} \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

end

3 Models, Invariants and Refinements

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

3.1 Specifications, reachability, and behaviours.

Transition systems are multi-pointed graphs.

record $'s\ \textit{TS} =$
$\textit{init} :: {'s}\ \textit{set}$
$\textit{trans} :: ({'s} \times {'s})\ \textit{set}$

The inductive set of reachable states.

inductive-set
$\textit{reach} :: ({'s}, {'a})\ \textit{TS-scheme} \Rightarrow {'s}\ \textit{set}$

for T :: ($'s$, $'a$) *TS-scheme*
where
r-init [*intro*]: $s \in init\ T \Longrightarrow s \in reach\ T$
| *r-trans* [*intro*]: $\llbracket (s, t) \in trans\ T;\ s \in reach\ T \rrbracket \Longrightarrow t \in reach\ T$

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* $\Rightarrow$ ($'s$ *list*) *set*
for T :: ($'s$, $'a$) *TS-scheme*
where
b-empty [*iff*]: $[] \in beh\ T$
| *b-init* [*intro*]: $s \in init\ T \Longrightarrow [s] \in beh\ T$
| *b-trans* [*intro*]: $\llbracket s \# b \in beh\ T;\ (s, t) \in trans\ T \rrbracket \Longrightarrow t \# s \# b \in beh\ T$

inductive-cases *beh-non-empty*: $s \# b \in beh\ T$

Behaviours are prefix closed.

lemma *beh-immediate-prefix-closed*:
$s \# b \in beh\ T \Longrightarrow b \in beh\ T$
⟨*proof*⟩

lemma *beh-prefix-closed*:
$c\ @\ b \in beh\ T \Longrightarrow b \in beh\ T$
⟨*proof*⟩

States in behaviours are exactly reachable.

lemma *beh-in-reach* [*rule-format*]:
$b \in beh\ T \Longrightarrow (\forall s \in set\ b.\ s \in reach\ T)$
⟨*proof*⟩

lemma *reach-in-beh*:
$s \in reach\ T \Longrightarrow \exists\, b \in beh\ T.\ s \in set\ b$
⟨*proof*⟩

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

Consecutive states in a behavior are connected by the transition relation

lemma *beh-consecutive-in-trans*:
  assumes $b \in beh\ TS$
  and $Suc\ i < length\ b$
  and $s = b\ !\ Suc\ i$
  and $t = b\ !\ i$
  shows $(s,\ t) \in trans\ TS$
⟨*proof*⟩

3.1.2 Specifications, observability, and implementation

Specifications add an observer function to transition systems.

record $('s,\ 'o)\ spec = 's\ TS\ +$
  $obs :: 's \Rightarrow 'o$

lemma *beh-obs-upd* [*simp*]: $beh\ (S(|\ obs := x\ |)) = beh\ S$
⟨*proof*⟩

lemma *reach-obs-upd* [*simp*]: $reach\ (S(|\ obs := x\ |)) = reach\ S$
⟨*proof*⟩

Observable behaviour and reachability.

definition
  $obeh :: ('s,\ 'o)\ spec \Rightarrow ('o\ list)\ set$ **where**
  $obeh\ S \equiv (map\ (obs\ S))\ `(beh\ S)$

definition
  $oreach :: ('s,\ 'o)\ spec \Rightarrow 'o\ set$ **where**
  $oreach\ S \equiv (obs\ S)\ `(reach\ S)$

lemma *oreach-equiv-obeh-states*: $oreach\ S = (\bigcup b{\in}obeh\ S.\ set\ b)$
⟨*proof*⟩

lemma *obeh-pi-translation*:
  $(map\ pi)\ `(obeh\ S) = obeh\ (S(|\ obs := pi\ o\ (obs\ S)\ |))$
  ⟨*proof*⟩

lemma *oreach-pi-translation*:

$pi`(oreach\ S) = oreach\ (S(|\ obs := pi\ o\ (obs\ S)\ |))$
⟨*proof*⟩

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*]:
$[\![observable\ ob\ P;\ ob\ s = ob\ s';\ s' \in P]\!] \Longrightarrow s \in P$
⟨*proof*⟩

lemma *observable2-equiv-observable*: $observable2\ ob\ P = observable\ ob\ P$
⟨*proof*⟩

lemma *observable3-equiv-observable2*: $observable3\ ob\ P = observable2\ ob\ P$
⟨*proof*⟩

lemma *observable-id* [*simp*]: $observable\ id\ P$
⟨*proof*⟩

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*: $(ob`P \subseteq Q) = (P \subseteq ob-`Q)$
⟨*proof*⟩

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

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

lemma *image-r-vimage-l*: $[\![\ Q \subseteq ob`P;\ observable\ ob\ P\ ]\!] \Longrightarrow ob -`Q \subseteq P$
⟨*proof*⟩

lemma *vimage-l-image-r*: $[\![\ ob -`Q \subseteq P;\ Q \subseteq range\ ob\ ]\!] \Longrightarrow Q \subseteq ob`P$
⟨*proof*⟩

Internal and external invariants

lemma *external-from-internal-invariant*:
$[\![\ reach\ S \subseteq P;\ (obs\ S)`P \subseteq Q\ ]\!]$
$\Longrightarrow oreach\ S \subseteq Q$
⟨*proof*⟩

lemma *external-from-internal-invariant-vimage*:
$[\![\ reach\ S \subseteq P;\ P \subseteq (obs\ S) -`Q\ ]\!]$
$\Longrightarrow oreach\ S \subseteq Q$
⟨*proof*⟩

lemma *external-to-internal-invariant-vimage*:
$[\![\ oreach\ S \subseteq Q;\ (obs\ S) -`Q \subseteq P\ ]\!]$
$\Longrightarrow reach\ S \subseteq P$
⟨*proof*⟩

lemma *external-to-internal-invariant*:
$[\![\ oreach\ S \subseteq Q;\ Q \subseteq (obs\ S)`P;\ observable\ (obs\ S)\ P\ ]\!]$
$\Longrightarrow reach\ S \subseteq P$
⟨*proof*⟩

lemma *external-equiv-internal-invariant-vimage*:
$[\![\ P = (obs\ S) -`Q\ ]\!]$
$\Longrightarrow (oreach\ S \subseteq Q) = (reach\ S \subseteq P)$
⟨*proof*⟩

lemma *external-equiv-internal-invariant*:
$[\![(obs\ S)`P = Q;\ observable\ (obs\ S)\ P\ ]\!]$
$\Longrightarrow (oreach\ S \subseteq Q) = (reach\ S \subseteq P)$
⟨*proof*⟩

Our notion of implementation is inclusion of observable behaviours.

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

Reflexivity and transitivity

lemma *implements-refl*: *implements id S S*
⟨*proof*⟩

lemma *implements-trans*:
$[\![implements\ pi1\ S1\ S2;\ implements\ pi2\ S2\ S3\ ]\!]$
$\Longrightarrow implements\ (pi1\ o\ pi2)\ S1\ S3$
⟨*proof*⟩

Preservation of external invariants

lemma *implements-oreach*:
$implements\ pi\ Sa\ Sc \Longrightarrow pi`(oreach\ Sc) \subseteq oreach\ Sa$
⟨*proof*⟩

lemma *external-invariant-preservation*:
$[\![oreach\ Sa \subseteq Q;\ implements\ pi\ Sa\ Sc\ ]\!]$
$\Longrightarrow pi`(oreach\ Sc) \subseteq Q$
⟨*proof*⟩

lemma *external-invariant-translation*:
$[\![oreach\ Sa \subseteq Q;\ pi-`Q \subseteq P;\ implements\ pi\ Sa\ Sc\ ]\!]$
$\Longrightarrow oreach\ Sc \subseteq P$
⟨*proof*⟩

Preservation of internal invariants

lemma *internal-invariant-translation*:
$[\![reach\ Sa \subseteq Pa;\ Pa \subseteq obs\ Sa -`\ Qa;\ pi -`\ Qa \subseteq Q;\ obs\ S -`\ Q \subseteq P;$
$implements\ pi\ Sa\ S\ ]\!]$
$\Longrightarrow reach\ S \subseteq P$
⟨*proof*⟩

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

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

lemma $\{P\}\ R\ \{>\ Q\} = (\forall\, s\ t.\ s \in P \longrightarrow (s,\ t) \in R \longrightarrow t \in Q)$
⟨*proof*⟩

lemma *hoareD*:
  $[\![\ \{I\}\ R\ \{>J\};\ s \in I;\ (s,\ s') \in R\ ]\!] \Longrightarrow s' \in J$
  ⟨*proof*⟩

Some essential facts about Hoare triples.

lemma *hoare-conseq-left* [*intro*]:
  $[\![\ \{P'\}\ R\ \{>\ Q\};\ P \subseteq P'\ ]\!]$
  $\Longrightarrow \{P\}\ R\ \{>\ Q\}$
⟨*proof*⟩

lemma *hoare-conseq-right*:
  $[\![\ \{P\}\ R\ \{>\ Q'\};\ Q' \subseteq Q\ ]\!]$
  $\Longrightarrow \{P\}\ R\ \{>\ Q\}$
⟨*proof*⟩

lemma *hoare-false-left* [*simp*]:
  $\{\{\}\}\ R\ \{>\ Q\}$
⟨*proof*⟩

lemma *hoare-true-right* [*simp*]:
  $\{P\}\ R\ \{>\ \mathit{UNIV}\}$
⟨*proof*⟩

lemma *hoare-conj-right* [*intro*!]:
$\llbracket$ $\{P\}$ R $\{> Q1\}$; $\{P\}$ R $\{> Q2\}$ $\rrbracket$
$\Longrightarrow$ $\{P\}$ R $\{> Q1 \cap Q2\}$
⟨*proof*⟩

Special transition relations.

lemma *hoare-stop* [*simp*, *intro*!]:
$\{P\}$ $\{\}$ $\{> Q\}$
⟨*proof*⟩

lemma *hoare-skip* [*simp*, *intro*!]:
$P \subseteq Q \Longrightarrow \{P\}$ *Id* $\{> Q\}$
⟨*proof*⟩

lemma *hoare-trans-Un* [*iff*]:
$\{P\}$ $R1 \cup R2$ $\{> Q\}$ = ($\{P\}$ $R1$ $\{> Q\}$ $\wedge$ $\{P\}$ $R2$ $\{> Q\}$)
⟨*proof*⟩

lemma *hoare-trans-UN* [*iff*]:
$\{P\}$ $\bigcup$ $x.$ R x $\{> Q\}$ = ($\forall x.$ $\{P\}$ R x $\{> Q\}$)
⟨*proof*⟩

3.2.2 Characterization of reachability

lemma *reach-init*: *reach* $T \subseteq I \Longrightarrow$ *init* $T \subseteq I$
⟨*proof*⟩

lemma *reach-trans*: *reach* $T \subseteq I \Longrightarrow$ $\{$*reach* $T\}$ *trans* T $\{> I\}$
⟨*proof*⟩

Useful consequences.

corollary *init-reach* [*iff*]: *init* $T \subseteq$ *reach* T
⟨*proof*⟩

corollary *trans-reach* [*iff*]: $\{$*reach* $T\}$ *trans* T $\{>$ *reach* $T\}$
⟨*proof*⟩

3.2.3 Invariant proof rules

Basic proof rule for invariants.

lemma *inv-rule-basic*:
$[\![$ *init* $T \subseteq P$; $\{P\}$ (*trans* T) $\{> P\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq P$
⟨*proof*⟩

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

lemma *inv-rule*:
$[\![$ *init* $T \subseteq I$; $I \subseteq P$; $\{I\}$ (*trans* T) $\{> I\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq P$
⟨*proof*⟩

The following rule is equivalent to the previous one.

lemma *INV-rule*:
$[\![$ *init* $T \subseteq I$; $\{I \cap$ *reach* $T\}$ (*trans* T) $\{> I\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq I$
⟨*proof*⟩

Proof of equivalence.

lemma *inv-rule-from-INV-rule*:
$[\![$ *init* $T \subseteq I$; $I \subseteq P$; $\{I\}$ (*trans* T) $\{> I\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq P$
⟨*proof*⟩

lemma *INV-rule-from-inv-rule*:
$[\![$ *init* $T \subseteq I$; $\{I \cap$ *reach* $T\}$ (*trans* T) $\{> I\}$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq I$
⟨*proof*⟩

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

lemma *inv-rule-incr*:
$[\![$ *init* $T \subseteq I$; $\{I \cap J\}$ (*trans* T) $\{> I\}$; *reach* $T \subseteq J$ $]\!]$
$\Longrightarrow$ *reach* $T \subseteq I$
⟨*proof*⟩

3.3 Refinement

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

3.3.1 Relational Hoare tuples

Relational Hoare tuples formalize the following generalized simulation diagram:

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

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

definition
  PO-rhoare ::
    $[('s \times 't)\ set,\ ('s \times 's)\ set,\ ('t \times 't)\ set,\ ('s \times 't)\ set] \Rightarrow bool$
    (‹(4{-} -, - {> -})› [0, 0, 0] 90)
where
  $\{pre\}\ Ra,\ Rc\ \{>\ post\} \equiv pre\ O\ Rc \subseteq Ra\ O\ post$

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

Facts about relational Hoare tuples.

lemma *relhoare-conseq-left* [*intro*]:
  $[\![\ \{pre'\}\ Ra,\ Rc\ \{>\ post\};\ pre \subseteq pre'\ ]\!]$
  $\Longrightarrow \{pre\}\ Ra,\ Rc\ \{>\ post\}$
⟨*proof*⟩

lemma *relhoare-conseq-right*: — do NOT declare [intro]
  $[\![\ \{pre\}\ Ra,\ Rc\ \{>\ post'\};\ post' \subseteq post\ ]\!]$
  $\Longrightarrow \{pre\}\ Ra,\ Rc\ \{>\ post\}$
⟨*proof*⟩

lemma *relhoare-false-left* [*simp*]: — do NOT declare [intro]
{ {} } *Ra*, *Rc* {> *post*}
⟨*proof*⟩

lemma *relhoare-true-right* [*simp*]: — not true in general
{*pre*} *Ra*, *Rc* {> *UNIV*} = (*Domain* (*pre O Rc*) ⊆ *Domain Ra*)
⟨*proof*⟩

lemma *Domain-rel-comp* [*intro*]:
Domain pre ⊆ *R* ⟹ *Domain* (*pre O Rc*) ⊆ *R*
⟨*proof*⟩

lemma *rel-hoare-skip* [*iff*]: {*R*} *Id*, *Id* {> *R*}
⟨*proof*⟩

Reflexivity and transitivity.

lemma *relhoare-refl* [*simp*]: {*Id*} *R*, *R* {> *Id*}
⟨*proof*⟩

lemma *rhoare-trans*:
⟦ {*R1*} *T1*, *T2* {> *R1*}; {*R2*} *T2*, *T3* {> *R2*} ⟧
⟹ {*R1 O R2*} *T1*, *T3* {> *R1 O R2*}
⟨*proof*⟩

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

lemma *relhoare-conj-right-det*:
⟦ {*pre*} *Ra*, *Rc* {> *post1*}; {*pre*} *Ra*, *Rc* {> *post2*};
single-valued Ra ⟧ — only for deterministic *Ra*!
⟹ {*pre*} *Ra*, *Rc* {> *post1* ∩ *post2*}
⟨*proof*⟩

lemma *relhoare-conj-right-cartesian* [*intro*]:
⟦ {*Domain pre*} *Ra* {> *I*}; {*Range pre*} *Rc* {> *J*};
{*pre*} *Ra*, *Rc* {> *post*} ⟧
⟹ {*pre*} *Ra*, *Rc* {> *post* ∩ *I* × *J*}
⟨*proof*⟩

Separate rule for cartesian products.

corollary *relhoare-cartesian*:
$[\![\{Domain\ pre\}\ Ra\ \{>I\};\ \{Range\ pre\}\ Rc\ \{>J\};$
$\{pre\}\ Ra,\ Rc\ \{>post\}\ ]\!]$ — any *post*, including *UNIV*!
$\Longrightarrow \{pre\}\ Ra,\ Rc\ \{>I \times J\}$
⟨*proof*⟩

Unions of transition relations.

lemma *relhoare-concrete-Un* [*simp*]:
$\{pre\}\ Ra,\ Rc1 \cup Rc2\ \{>post\}$
$= (\{pre\}\ Ra,\ Rc1\ \{>post\} \wedge \{pre\}\ Ra,\ Rc2\ \{>post\})$
⟨*proof*⟩

lemma *relhoare-concrete-UN* [*simp*]:
$\{pre\}\ Ra,\ \bigcup x.\ Rc\ x\ \{>post\} = (\forall x.\ \{pre\}\ Ra,\ Rc\ x\ \{>post\})$
⟨*proof*⟩

lemma *relhoare-abstract-Un-left* [*intro*]:
$[\![\{pre\}\ Ra1,\ Rc\ \{>post\}\ ]\!]$
$\Longrightarrow \{pre\}\ Ra1 \cup Ra2,\ Rc\ \{>post\}$
⟨*proof*⟩

lemma *relhoare-abstract-Un-right* [*intro*]:
$[\![\{pre\}\ Ra2,\ Rc\ \{>post\}\ ]\!]$
$\Longrightarrow \{pre\}\ Ra1 \cup Ra2,\ Rc\ \{>post\}$
⟨*proof*⟩

lemma *relhoare-abstract-UN* [*intro*!]: — might be too aggressive?
$[\![\{pre\}\ Ra\ x,\ Rc\ \{>post\}\ ]\!]$
$\Longrightarrow \{pre\}\ \bigcup x.\ Ra\ x,\ Rc\ \{>post\}$
⟨*proof*⟩

3.3.2 Refinement proof obligations

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

definition

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

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

lemma *refine-basic*:
  $[\![\ init\ Tc \subseteq R``(init\ Ta);\ \{R\}\ (trans\ Ta),\ (trans\ Tc)\ \{>\ R\}\ ]\!]$
  $\Longrightarrow PO\text{-}refines\ R\ Ta\ Tc$
⟨*proof*⟩

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

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

lemma *refine-init-using-invariants*:
  $[\![\ init\ Tc \subseteq R``(init\ Ta);\ init\ Ta \subseteq I;\ init\ Tc \subseteq J\ ]\!]$
  $\Longrightarrow init\ Tc \subseteq (R \cap I \times J)``(init\ Ta)$
⟨*proof*⟩

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

This is our main rule for refinements.

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

$\{J \cap \mathit{Range}\ (R \cap I \times \mathit{UNIV})\}\ (\mathit{trans}\ \mathit{Tc})\ \{> J\}$;
$\mathit{init}\ \mathit{Tc} \subseteq R``(\mathit{init}\ \mathit{Ta})$;
$\mathit{init}\ \mathit{Ta} \subseteq I$; $\mathit{init}\ \mathit{Tc} \subseteq J$ ⟧
$\Longrightarrow$ *PO-refines* $(R \cap I \times J)$ *Ta Tc*
⟨*proof*⟩

3.3.3 Deriving invariants from refinements

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

lemma *PO-refines-implies-Range-init*:
PO-refines R Ta Tc $\Longrightarrow$ *init Tc* $\subseteq$ *Range R*
⟨*proof*⟩

lemma *PO-refines-implies-Range-trans*:
PO-refines R Ta Tc $\Longrightarrow$ $\{$*Range R*$\}$ *trans Tc* $\{>$ *Range R*$\}$
⟨*proof*⟩

lemma *PO-refines-implies-Range-invariant*:
PO-refines R Ta Tc $\Longrightarrow$ *reach Tc* $\subseteq$ *Range R*
⟨*proof*⟩

The following rules are more useful in proofs.

corollary *INV-init-from-refinement*:
⟦ *PO-refines R Ta Tc*; *Range R* $\subseteq$ *I* ⟧
$\Longrightarrow$ *init Tc* $\subseteq$ *I*
⟨*proof*⟩

corollary *INV-trans-from-refinement*:
⟦ *PO-refines R Ta Tc*; *K* $\subseteq$ *Range R*; *Range R* $\subseteq$ *I* ⟧
$\Longrightarrow$ $\{K\}$ *trans Tc* $\{> I\}$
⟨*proof*⟩

corollary *INV-from-refinement*:
⟦ *PO-refines R Ta Tc*; *Range R* $\subseteq$ *I* ⟧
$\Longrightarrow$ *reach Tc* $\subseteq$ *I*
⟨*proof*⟩

3.3.4 Transfering abstract invariants to concrete systems

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

lemma *PO-refines-implies-R-image-init*:
PO-refines R Ta Tc $\Longrightarrow$ *init Tc* $\subseteq$ *R* ‘‘ (*init Ta*)
⟨*proof*⟩

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

lemma *PO-refines-implies-R-image-trans*:
assumes *PO-refines R Ta Tc*
shows {*R* ‘‘ *reach Ta*} *trans Tc* {> *R* ‘‘ *reach Ta*} ⟨*proof*⟩

lemma *PO-refines-implies-R-image-invariant*:
assumes *PO-refines R Ta Tc*
shows *reach Tc* $\subseteq$ *R* ‘‘ *reach Ta*
⟨*proof*⟩

lemma *abs-INV-init-transfer*:
assumes
PO-refines R Ta Tc
init Ta $\subseteq$ *I*
shows *init Tc* $\subseteq$ *R* ‘‘ *I* ⟨*proof*⟩

lemma *abs-INV-trans-transfer*:
assumes
ref: *PO-refines R Ta Tc*
and *abs-hoare*: {*I*} *trans Ta* {> *J*}
shows {*R* ‘‘ *I*} *trans Tc* {> *R* ‘‘ *J*}
⟨*proof*⟩

lemma *abs-INV-transfer*:
assumes
PO-refines R Ta Tc
reach Ta $\subseteq$ *I*
shows *reach Tc* $\subseteq$ *R* ‘‘ *I* ⟨*proof*⟩

3.3.5 Refinement of specifications

Lift relation membership to finite sequences

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

inductive-cases *sl-cons-right-invert*: (*ba'*, *t* # *bc*) ∈ *seq-lift R*

For each concrete behaviour there is a related abstract one.

lemma *behaviour-refinement*:
assumes *PO-refines R Ta Tc bc* ∈ *beh Tc*
shows ∃ *ba* ∈ *beh Ta*. (*ba*, *bc*) ∈ *seq-lift R*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *obs-consistent-empty*: *obs-consistent* {} *pi Sa Sc*
⟨*proof*⟩

lemma *obs-consistent-conj1* [*intro*]:

obs-consistent R pi Sa Sc ⟹ *obs-consistent* (*R* ∩ *R*′) *pi Sa Sc*
⟨*proof*⟩

lemma *obs-consistent-conj2* [*intro*]:
obs-consistent R pi Sa Sc ⟹ *obs-consistent* (*R*′ ∩ *R*) *pi Sa Sc*
⟨*proof*⟩

lemma *obs-consistent-behaviours*:
⟦ *obs-consistent R pi Sa Sc*; *bc* ∈ *beh Sc*; *ba* ∈ *beh Sa*; (*ba*, *bc*) ∈ *seq-lift R*⟧
⟹ *map pi* (*map* (*obs Sc*) *bc*) = *map* (*obs Sa*) *ba*
⟨*proof*⟩

Definition of refinement proof obligations.

definition
refines ::
[(′*s* × ′*t*) *set*, ′*p* ⇒ ′*o*, (′*s*, ′*o*) *spec*, (′*t*, ′*p*) *spec*] ⇒ *bool*
where
refines R pi Sa Sc ≡ *obs-consistent R pi Sa Sc* ∧ *PO-refines R Sa Sc*

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

lemma *refinesI*:
⟦ *PO-refines R Sa Sc*; *obs-consistent R pi Sa Sc* ⟧
⟹ *refines R pi Sa Sc*
⟨*proof*⟩

lemma *PO-refines-from-refines*:
refines R pi Sa Sc ⟹ *PO-refines R Sa Sc*
⟨*proof*⟩

Reflexivity and transitivity of refinement.

lemma *refinement-reflexive*: *refines Id id S S*
⟨*proof*⟩

lemma *refinement-transitive*:
⟦ *refines R1 pi1 S1 S2*; *refines R2 pi2 S2 S3* ⟧
⟹ *refines* (*R1 O R2*) (*pi1 o pi2*) *S1 S3*
⟨*proof*⟩

Soundness of refinement for proving implementation

lemma *observable-behaviour-refinement*:
  $[\![$ *refines R pi Sa Sc*; *bc* $\in$ *obeh Sc* $]\!] \Longrightarrow$ *map pi bc* $\in$ *obeh Sa*
$\langle$*proof*$\rangle$

theorem *refinement-soundness*:
  refines R pi Sa Sc $\Longrightarrow$ *implements pi Sa Sc*
$\langle$*proof*$\rangle$

Extended versions of proof rules including observations

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

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

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

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

end

3.4 Transition system semantics for HO models

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

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

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

definition *get-msgs* ::
  $('proc \Rightarrow 'proc \Rightarrow 'pst \Rightarrow 'msg)$
  $\Rightarrow ('proc \Rightarrow 'pst)$
  $\Rightarrow 'proc\ HO$
  $\Rightarrow 'proc\ HO$
  $\Rightarrow 'proc \Rightarrow ('proc \rightharpoonup 'msg)set$
where
  $get\text{-}msgs\ snd\text{-}f\ cfg\ HO\ SHO \equiv \lambda p.$
   $\{\mu.\ (\forall q.\ q \in HO\ p \longleftrightarrow \mu\ q \neq None)$
     $\wedge (\forall q.\ q \in SHO\ p \cap HO\ p \longrightarrow \mu\ q = Some\ (snd\text{-}f\ q\ p\ (cfg\ q)))\}$

definition *CSHO-trans-alt*
  ::
  $(nat \Rightarrow 'proc \Rightarrow 'proc \Rightarrow 'pst \Rightarrow 'msg)$
  $\Rightarrow (nat \Rightarrow 'proc \Rightarrow 'pst \Rightarrow ('proc \rightharpoonup 'msg) \Rightarrow 'proc \Rightarrow 'pst \Rightarrow bool)$
  $\Rightarrow (nat \Rightarrow 'proc\ HO)$
  $\Rightarrow (nat \Rightarrow 'proc\ HO)$
  $\Rightarrow (nat \Rightarrow 'proc \Rightarrow 'proc)$
  $\Rightarrow ((nat \times ('proc \Rightarrow 'pst)) \times (nat \times ('proc \Rightarrow 'pst)))set$
where
  $CSHO\text{-}trans\text{-}alt\ snd\text{-}f\ nxt\text{-}st\ HOs\ SHOs\ coords \equiv$
   $\bigcup r\ \mu.\ \{((r, cfg), (Suc\ r, cfg'))|cfg\ cfg'.\ \forall p.$
     $\mu\ p \in (get\text{-}msgs\ (snd\text{-}f\ r)\ cfg\ (HOs\ r)\ (SHOs\ r)\ p)$
     $\wedge (\forall p.\ nxt\text{-}st\ r\ p\ (cfg\ p)\ (\mu\ p)\ (coords\ r\ p)\ (cfg'\ p))$
   $\}$

lemma *CHO-trans-alt*:

CHO-trans A HOs SHOs coords = *CSHO-trans-alt* (*sendMsg A*) (*CnextState A*) *HOs SHOs coords*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *CSHORun-get-msgs*:
CSHORun (*A* :: (*'proc*, *'pst*, *'msg*) *CHOAlgorithm*) *rho HOs SHOs coords* = (
CHOinitConfig A (*rho 0*) (*coords 0*)
∧ (∀ *r*. ∃ *μ*.
(∀ *p*.
μ p ∈ *get-msgs* (*sendMsg A r*) (*rho r*) (*HOs r*) (*SHOs r*) *p*
∧ *CnextState A r p* (*rho r p*) (*μ p*) (*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*))))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *get-msgs-benign*:
get-msgs snd-f cfg HOs HOs p = { (*Some o* (*λq*. (*snd-f q p* (*cfg q*)))) |' (*HOs p*)}
⟨*proof*⟩

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*))
⟨*proof*⟩

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

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

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

definition *no-defection* :: *v-state* ⇒ (*process*, *val*)*map* ⇒ *round* ⇒ *bool* **where**
no-defection-def′:
no-defection s r-votes r ≡
$\forall r' < r.\ \forall Q \in Quorum.\ \forall v.\ (votes\ s\ r')\ `\ Q = \{Some\ v\} \longrightarrow r\text{-}votes\ `\ Q \subseteq \{None,\ Some\ v\}$

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

lemma *no-defection-def*:
no-defection s round-votes r =
$(\forall r' < r.\ \forall a\ Q\ v.\ quorum\text{-}for\ Q\ v\ (votes\ s\ r') \land a \in Q \longrightarrow round\text{-}votes\ a \in \{None,\ Some\ v\})$
⟨*proof*⟩

definition *locked* :: *v-state* ⇒ *val set* **where**
$locked\ s = \{v.\ \exists r.\ locked\text{-}in\ s\ r\ v\}$

The sole system event.

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

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

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

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

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

4.2 Invariants

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

definition *Vinv1* **where**
$Vinv1 = \{s.\ \forall r.\ next\text{-}round\ s \leq r \longrightarrow votes\ s\ r = Map.empty\ \}$

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

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

definition *Vinv2* **where**
$Vinv2 = \{s.\ \forall r.\ no\text{-}defection\ s\ (votes\ s\ r)\ r\ \}$

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

definition *Vinv3* **where**
$Vinv3 = \{s.\ ran\ (decisions\ s) \subseteq locked\ s\}$

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

4.2.1 Proofs of invariants

lemma *Vinv1-v-round*:
$\{Vinv1\}\ v\text{-}round\ r\ v\text{-}f\ d\text{-}f\ \{>\ Vinv1\}$
⟨*proof*⟩

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

lemma *Vinv1-inductive*:
$init\ v\text{-}TS \subseteq Vinv1$
$\{Vinv1\}\ trans\ v\text{-}TS\ \{>\ Vinv1\}$
⟨*proof*⟩

lemma *Vinv1-invariant*: *reach v-TS* $\subseteq$ *Vinv1*
⟨*proof*⟩

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* $\Longrightarrow$ *finite* (*map-graph* (*case-prod* (*votes s*)))
⟨*proof*⟩

lemma *Vinv1-finite-vote-set*:
$s \in$ *Vinv1* $\Longrightarrow$ *finite* (*vote-set* (*votes s*) Q)
⟨*proof*⟩

lemma *process-mru-map-add*:
assumes
$s \in$ *Vinv1*
shows
process-mru ((*votes s*)(*next-round s* := *v-f*)) =
(*process-mru* (*votes s*) ++ (λp. *map-option* (*Pair* (*next-round s*)) (*v-f p*)))
⟨*proof*⟩

lemma *no-defection-empty*:
no-defection s Map.empty r'
⟨*proof*⟩

lemma *no-defection-preserved*:
assumes
$s \in$ *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' ⟨*proof*⟩

lemma *Vinv2-v-round*:
  {*Vinv2* ∩ *Vinv1*} *v-round r v-f d-f* {> *Vinv2*}
  ⟨*proof*⟩

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

lemma *Vinv2-inductive*:
  init v-TS ⊆ *Vinv2*
  {*Vinv2* ∩ *Vinv1*} *trans v-TS* {> *Vinv2*}
  ⟨*proof*⟩

lemma *Vinv2-invariant*: *reach v-TS* ⊆ *Vinv2*
  ⟨*proof*⟩

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

lemma *Vinv3-v-round*:
  {*Vinv3* ∩ *Vinv1*} *v-round r v-f d-f* {> *Vinv3*}
⟨*proof*⟩

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

lemma *Vinv3-inductive*:
  init v-TS ⊆ *Vinv3*
  {*Vinv3* ∩ *Vinv1*} *trans v-TS* {> *Vinv3*}
  ⟨*proof*⟩

lemma *Vinv3-invariant*: *reach v-TS* ⊆ *Vinv3*
  ⟨*proof*⟩

4.3 Agreement and stability

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

lemma *locked-in-vf-same*:
$[\![$ *locked-in-vf v-f v*; *locked-in-vf v-f w* $]\!] \Longrightarrow v = w$ ⟨*proof*⟩

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

theorem *locked-in-different*:
assumes
$s \in Vinv2$
locked-in s r1 v
locked-in s r2 w
$r1 < r2$
shows
$v = w$
⟨*proof*⟩

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

theorem *locked-unique*:
assumes
$s \in Vinv2$
$v \in$ *locked s* $w \in$ *locked s*
shows
$v = w$
⟨*proof*⟩

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*: $tr \in$ *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*⟩

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

lemma *Voting-agreement*:
assumes *beh*: *tr* ∈ *beh v-TS*
and *len*: i < *length tr*
and *s*: s = *nth tr i*
and *t*: t = *nth tr* ($i - j$)
and *dec*:
decisions s p = *Some v*
decisions t q = *Some w*
shows $w = v$
⟨*proof*⟩

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})
⟨*proof*⟩

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*⟩

lemma *opt-no-defection-imp-no-defection*:
assumes
conc-guard: *opt-no-defection sc round-votes*
and *R*: (*sa*, *sc*) ∈ *flv-ref-rel*
and *ainv*: *sa* ∈ *Vinv1 sa* ∈ *Vinv2*
shows
no-defection sa round-votes r
⟨*proof*⟩

5.2.2 Action refinement

lemma *act-ref*:
assumes
inv: *s* ∈ *Vinv1*

shows
map-option snd o (*process-mru* ((*votes s*)(*v-state.next-round s* := *v-f*)))
= ((*map-option snd o* (*process-mru* (*votes s*))) ++ *v-f*)
⟨*proof*⟩

5.2.3 The complete refinement proof

lemma *flv-round-refines*:
{*flv-ref-rel* ∩ (*Vinv1* ∩ *Vinv2*) × *UNIV*}
v-round r v-f d-f, *flv-round r v-f d-f*
{> *flv-ref-rel*}
⟨*proof*⟩

lemma *Last-Voting-Refines*:
PO-refines (*flv-ref-rel* ∩ (*Vinv1* ∩ *Vinv2*) × *UNIV*) *v-TS flv-TS*
⟨*proof*⟩

end

end

6 The OneThirdRule Algorithm

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

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

6.1 Model of the algorithm

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

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

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

definition *OTR-initState* **where**
  OTR-initState p st ≡ *decision st* = *None*

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

definition *HOV* :: (*process* ⇒ *'val option*) ⇒ *'val* ⇒ *process set* **where**
  HOV msgs v ≡ { *q* . *msgs q* = *Some v* }

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

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

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

lemma *MFR-exists*: ∃ *v*. *MFR msgs v*
⟨*proof*⟩

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

lemma *MFR-in-msgs*:
  assumes *HO*:*HOs m p* ≠ {}
    and *v*: *MFR* (*HOrcvdMsgs OTR-M m p* (*HOs m p*) (*rho m*)) *v*
      (**is** *MFR ?msgs v*)
  shows ∃ *q* ∈ *HOs m p*. *v* = *the* (*?msgs q*)
⟨*proof*⟩

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

definition *TwoThirds* **where**
  TwoThirds msgs v ≡ (*2*∗*N*) *div 3* < *card* (*HOV msgs v*)

The next-state relation of algorithm *One-Third Rule* for every process is defined as follows: if the process has received values from more than 2/3 of all processes, the *last-vote* field is set to the smallest among the most frequently received values, and the process decides value *v* if it received *v* from more than 2/3 of all processes. If *p* hasn't heard from more than

2/3 of all processes, the state remains unchanged. (Note that *Some* is the constructor of the option datatype, whereas ϵ is Hilbert's choice operator.) We require the type of values to be linearly ordered so that the minimum is guaranteed to be well-defined.

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

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

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

6.2 Communication predicate for *One-Third Rule*

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

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

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

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

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

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

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

end

6.4 Proofs

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

lemma *card-Compl*:
fixes S :: (*'a* :: *finite*) *set*
shows $card\ (-S) = card\ (UNIV :: 'a\ set) - card\ S$
⟨*proof*⟩

lemma *m-mult-div-Suc-m*:
$n > 0 \Longrightarrow m * n\ div\ Suc\ m < n$
⟨*proof*⟩

interpretation *majorities*: *quorum-process majs*
⟨*proof*⟩

lemma *card-Un-le*:
$[\![\ finite\ A;\ finite\ B\ ]\!] \Longrightarrow card\ (A \cup B) \leq card\ A + card\ B$
⟨*proof*⟩

lemma *qintersect-card*:
assumes $Q \in majs$ $Q' \in majs$
shows $card\ (Q \cap Q') > card\ (Q \cap -Q')$
⟨*proof*⟩

axiomatization where *val-linorder*:

$OFCLASS(val,\ linorder\text{-}class)$

instance *val* :: *linorder* ⟨*proof*⟩

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

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

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

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

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

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

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

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

6.4.1 Refinement

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

lemma *decide-origin*:
assumes
send: *μ p* ∈ *get-msgs* (*OTR-sendMsg r*) *sc* (*HOs r*) (*HOs r*) *p*
and *nxt*: *OTR-nextState r p* (*sc p*) (*μ p*) (*sc′ p*)
and *new-dec*: *decision* (*sc′ p*) ≠ *decision* (*sc p*)
shows
∃ *v*. *decision* (*sc′ p*) = *Some v* ∧ {*q*. *last-vote* (*sc q*) = *v*} ∈ *majs*
⟨*proof*⟩

lemma *MFR-in-msgs*:
assumes *HO*:*dom msgs* ≠ {}
and *v*: *MFR msgs v*
shows ∃ *q* ∈ *dom msgs*. *v* = *the* (*msgs q*)
⟨*proof*⟩

lemma *step-ref*:
{*otr-ref-rel*}
(⋃ *r v-f d-f*. *majorities.flv-round r v-f d-f*),
OTR-trans-step HOs
{> *otr-ref-rel*}
⟨*proof*⟩

lemma *OTR-Refines-LV-VOting*:
PO-refines (*otr-ref-rel*)
majorities.flv-TS (*OTR-TS HOs HOs crds*)
⟨*proof*⟩

6.4.2 Termination

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

end

7 The $A_{T,E}$ Algorithm

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

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

7.1 Model of the algorithm

The following record models the local state of a process.

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

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

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

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

begin

The following are consequences of the assumptions on the parameters.

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

lemma *Egta*: $E > \alpha$
⟨*proof*⟩

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

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

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

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

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

definition
Ate-nextState :: *nat* ⇒ *process* ⇒ (′*val*::*linorder*) *pstate* ⇒ (*process* ⇒ ′*val option*)
⇒ ′*val pstate* ⇒ *bool*
where
Ate-nextState r p st msgs st′ ≡
(*if card* {*q*. *msgs q* ≠ *None*} > *T*
then x st′ = *Min* (*mostOftenRcvd msgs*)
else x st′ = *x st*)
∧ ((∃ *v*. *card* {*q*. *msgs q* = *Some v*} > *E* ∧ *decide st*′ = *Some v*)
∨ ¬ (∃ *v*. *card* {*q*. *msgs q* = *Some v*} > *E*)
∧ *decide st*′ = *decide st*)

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

definition *Ate-commPerRd* **where**
Ate-commPerRd HOrs SHOrs ≡
∀ *p*. *card* (*HOrs p* − *SHOrs p*) ≤ α

The global communication predicate stipulates the three following conditions:

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

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

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 = (λ *p st crd. Ate-initState p* (*st*::('*val*::*linorder*) *pstate*)),
sendMsg = *Ate-sendMsg*,
CnextState = (λ *r p st msgs crd st'. Ate-nextState r p st msgs st'*),
SHOcommPerRd = (*Ate-commPerRd*:: *process HO* ⇒ *process HO* ⇒ *bool*),
SHOcommGlobal = *Ate-commGlobal*
|)

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

end — locale *ate-parameters*

end

7.4 Proofs

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

instance *val* :: *linorder* ⟨*proof*⟩

context *ate-parameters*
begin

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

interpretation *majorities*: *quorum-process majs*
⟨*proof*⟩

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

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

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

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

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

definition

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

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

lemma *decide-origin*:
assumes
send: $\mu\ p \in get\text{-}msgs\ (Ate\text{-}sendMsg\ r)\ sc\ (HOs\ r)\ (HOs\ r)\ p$
and *nxt*: $Ate\text{-}nextState\ r\ p\ (sc\ p)\ (\mu\ p)\ (sc'\ p)$
and *new-dec*: $decide\ (sc'\ p) \neq decide\ (sc\ p)$
shows
$\exists\, v.\ decide\ (sc'\ p) = Some\ v \wedge \{q.\ x\ (sc\ q) = v\} \in majs$ ⟨*proof*⟩

lemma *other-values-received*:
assumes *nxt*: $Ate\text{-}nextState\ \ r\ q\ (sc\ q)\ \mu q\ ((sc')\ q)$
and *muq*: $\mu q \in get\text{-}msgs\ (Ate\text{-}sendMsg\ r)\ sc\ (HOs\ r)\ (HOs\ r)\ q$
and *vsent*: $card\ \{qq.\ sendMsg\ Ate\text{-}M\ r\ qq\ q\ (sc\ qq) = v\} > E - \alpha$
(**is** *card ?vsent* > -)
shows $card\ (\{qq.\ \mu q\ qq \neq Some\ v\} \cap HOs\ r\ q) \leq N + 2{*}\alpha - E$
⟨*proof*⟩

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

lemma *mostOftenRcvd-v*:
  assumes *nxt*: *Ate-nextState r q (sc q) μq ((sc′) q)*
  and *muq*: $\mu q \in$ *get-msgs (Ate-sendMsg r) sc (HOs r) (HOs r) q*
  and *threshold-T*: *card* $\{qq.\ \mu q\ qq \neq None\} > T$
  and *threshold-E*: *card* $\{qq.$ *sendMsg Ate-M r qq q (sc qq)* $= v\} > E - \alpha$
  shows *mostOftenRcvd* $\mu q = \{v\}$
⟨*proof*⟩

lemma *step-ref*:
  {*ate-ref-rel*}
    ($\bigcup$ *r v-f d-f. majorities.flv-round r v-f d-f*),
    Ate-trans-step HOs
  {> *ate-ref-rel*}
⟨*proof*⟩

lemma *Ate-Refines-LV-VOting*:
  PO-refines (ate-ref-rel)
    majorities.flv-TS (Ate-TS HOs HOs crds)
⟨*proof*⟩

end — context *ate-parameters*

7.4.2 Termination

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

end

8 The Same Vote Model

theory *Same-Vote*
imports *Voting*
begin

context *quorum-process* **begin**

8.1 Model definition

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

definition *safe* :: *v-state* ⇒ *round* ⇒ *val* ⇒ *bool* **where**
  safe-def′: $safe\ s\ r\ v \equiv$
    $\forall r' < r.\ \forall Q \in Quorum.\ \forall w.\ (votes\ s\ r')\ `\ Q = \{Some\ w\} \longrightarrow v = w$

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

lemma *safe-def*:
  $safe\ s\ r\ v =$
    $(\forall r' < r.\ \forall Q\ w.\ quorum\text{-}for\ Q\ w\ (votes\ s\ r') \longrightarrow v = w)$
  ⟨*proof*⟩

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

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

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

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

8.2 Refinement

lemma *safe-imp-no-defection*:

safe s (*next-round s*) *v* $\Longrightarrow$ *no-defection s* (*const-map v S*) (*next-round s*)
⟨*proof*⟩

lemma *const-map-quorum-locked*:
S ∈ *Quorum* $\Longrightarrow$ *locked-in-vf* (*const-map v S*) *v*
⟨*proof*⟩

lemma *sv-round-refines*:
{*Id*} *v-round r* (*const-map v S*) *r-decisions*, *sv-round r S v r-decisions* {> *Id*}
⟨*proof*⟩

lemma *Same-Vote-Refines*:
PO-refines Id v-TS sv-TS
⟨*proof*⟩

8.3 Invariants

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

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

8.3.1 Proof of invariants

lemma *SV-inv3-v-round*:
{*SV-inv3*} *sv-round r S v D* {> *SV-inv3*}
⟨*proof*⟩

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

lemma *SV-inv3-inductive*:
init sv-TS ⊆ *SV-inv3*
{*SV-inv3*} *trans sv-TS* {> *SV-inv3*}
⟨*proof*⟩

lemma *SV-inv3-invariant*: *reach sv-TS* ⊆ *SV-inv3*

⟨*proof*⟩

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

8.3.2 Transfer of abstract invariants

lemma *SV-inv1-inductive*:
  $init\ sv\text{-}TS \subseteq Vinv1$
  $\{Vinv1\}\ trans\ sv\text{-}TS\ \{>\ Vinv1\}$
  ⟨*proof*⟩

lemma *SV-inv1-invariant*:
  $reach\ sv\text{-}TS \subseteq Vinv1$
  ⟨*proof*⟩

lemma *SV-inv2-inductive*:
  $init\ sv\text{-}TS \subseteq Vinv2$
  $\{Vinv2 \cap Vinv1\}\ trans\ sv\text{-}TS\ \{>\ Vinv2\}$
  ⟨*proof*⟩

lemma *SV-inv2-invariant*:
  $reach\ sv\text{-}TS \subseteq Vinv2$
  ⟨*proof*⟩

8.3.3 Additional invariants

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

definition *SV-inv4* :: *v-state set* **where**
  $SV\text{-}inv4 = \{s.\ \forall\, v\ a\ r.\ votes\ s\ r\ a = Some\ v \longrightarrow safe\ s\ (Suc\ r)\ v\ \}$

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*:
  $\{SV\text{-}inv4 \cap (Vinv1 \cap Vinv2)\}$ *sv-round r S v D* $\{> SV\text{-}inv4\}$
⟨*proof*⟩

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

lemma *SV-inv4-inductive*:
  init sv-TS $\subseteq$ *SV-inv4*
  $\{SV\text{-}inv4 \cap (Vinv1 \cap Vinv2)\}$ *trans sv-TS* $\{> SV\text{-}inv4\}$
  ⟨*proof*⟩

lemma *SV-inv4-invariant*: *reach sv-TS* $\subseteq$ *SV-inv4*
  ⟨*proof*⟩

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

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

definition *obsv-init* :: *obsv-state set* **where**
obsv-init = { ⦇ *next-round* = *0*, *votes* = λ*r a*. *None*, *decisions* = *Map.empty*, *obs* = λ*r a*. *None* ⦈ }

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

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

9.2 Invariants

definition *OV-inv1* **where**
OV-inv1 = {*s*. ∀ *r Q v*. *quorum-for Q v* (*votes s r*) ⟶
(∀ *Q*′ ∈ *Quorum*. *quorum-for Q*′ *v* (*obs s r*))}

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

9.2.1 Proofs of invariants

lemma *OV-inv1-obsv-round*:
$\{OV\text{-}inv1\}$ *obsv-round r S v d-f Ob* $\{> OV\text{-}inv1\}$
⟨*proof*⟩

lemma *OV-inv1-inductive*:
init obsv-TS $\subseteq$ *OV-inv1*
$\{OV\text{-}inv1\}$ *trans obsv-TS* $\{> OV\text{-}inv1\}$
⟨*proof*⟩

lemma *quorum-for-const-map*:
(*quorum-for Q w* (*const-map v S*)) $=$ ($Q \in Quorum \wedge Q \subseteq S \wedge w = v$)
⟨*proof*⟩

9.3 Refinement

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

lemma *obsv-round-refines*:
{*obsv-ref-rel* $\cap$ *UNIV* $\times$ *OV-inv1*} *sv-round r S v dec-f*, *obsv-round r S v dec-f Ob* {> *obsv-ref-rel*}
⟨*proof*⟩

lemma *Observable-Refines*:
PO-refines (*obsv-ref-rel* $\cap$ *UNIV* $\times$ *OV-inv1*) *sv-TS obsv-TS*
⟨*proof*⟩

9.4 Additional invariants

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

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

definition *OV-inv3* **where**

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

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

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

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

9.4.1 Proofs of additional invariants

lemma *OV-inv2-inductive*:
init obsv-TS ⊆ *OV-inv2*
{*OV-inv2*} *trans obsv-TS* {> *OV-inv2*}
⟨*proof*⟩

lemma *SVinv3-inductive*:
init obsv-TS ⊆ *SV-inv3*
{*SV-inv3*} *trans obsv-TS* {> *SV-inv3*}
⟨*proof*⟩

lemma *OV-inv3-obsv-round*:
{*OV-inv3* ∩ *OV-inv2*} *obsv-round r S v D Ob* {> *OV-inv3*}
⟨*proof*⟩

lemma *OV-inv3-inductive*:
init obsv-TS ⊆ *OV-inv3*
{*OV-inv3* ∩ *OV-inv2*} *trans obsv-TS* {> *OV-inv3*}
⟨*proof*⟩

lemma *OV-inv4-inductive*:
init obsv-TS ⊆ *OV-inv4*
{*OV-inv4*} *trans obsv-TS* {> *OV-inv4*}
⟨*proof*⟩

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 ≡ ∃ *p*. *obs-f p* ∈ {*None*, *Some v*}

definition *olv-round* **where**
  olv-round r S v r-decisions Ob ≡ {(*s*, *s'*).
    — guards
    r = *next-round s*
    ∧ (*S* ≠ {} ⟶ *opt-obs-safe* (*last-obs s*) *v*)
    ∧ (*S* ∈ *Quorum* ⟶ *Ob* = *UNIV*)
    ∧ *d-guard r-decisions* (*const-map v S*)
    ∧ (*Ob* ≠ {} ⟶ *S* ≠ {})
    ∧ — actions
    s' = *s*⦇
     next-round := *Suc r*
     , *decisions* := *decisions s* ++ *r-decisions*
     , *last-obs* := *last-obs s* ++ *const-map v Ob*
   ⦈
  }

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

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

definition *olv-TS* :: *opt-obsv-state TS* **where**
olv-TS = ⦇ *init* = *olv-init*, *trans* = *olv-trans* ⦈

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

10.2 Refinement

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

lemma *OV-inv2-finite-map-graph*:
s ∈ *OV-inv2* ⟹ *finite* (*map-graph* (*case-prod* (*obsv-state.obs s*)))
⟨*proof*⟩

lemma *OV-inv2-finite-obs-set*:
s ∈ *OV-inv2* ⟹ *finite* (*vote-set* (*obsv-state.obs s*) *Q*)
⟨*proof*⟩

lemma *olv-round-refines*:
{*olv-ref-rel* ∩ (*OV-inv2* ∩ *OV-inv3* ∩ *OV-inv4*) × *UNIV*} *obsv-round r S v D Ob*, *olv-round r S v D Ob* {>*olv-ref-rel*}
⟨*proof*⟩

lemma *OLV-Refines*:
PO-refines (*olv-ref-rel* ∩ (*OV-inv2* ∩ *OV-inv3* ∩ *OV-inv4*) × *UNIV*) *obsv-TS olv-TS*
⟨*proof*⟩

end

end

11 Two-Step Observing Quorums Model

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

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

11.1 Model definition

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

context *mono-quorum*
begin

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

definition *obs-guard* :: (*process*, *val*)*map* ⇒ (*process*, *val*)*map* ⇒ *bool* **where**
  obs-guard r-obs r-v ≡ ∀ *p*.

$(\forall v.\ r\text{-}obs\ p = Some\ v \longrightarrow (\exists q.\ r\text{-}v\ q = Some\ v))$
$\wedge\ (dom\ r\text{-}v \in Quorum \longrightarrow (\exists q \in dom\ r\text{-}v.\ r\text{-}obs\ p = r\text{-}v\ q))$

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

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

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

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

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

11.2 Refinement

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

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

definition *step1-add-rel* :: (*opt-obsv-state* × *tso-state*)*set* **where**
$step1\text{-}add\text{-}rel = \{(sa,\ sc).\ \exists\, S\ v.$
$r\text{-}votes\ sc = const\text{-}map\ v\ S$
$\wedge\ (S \neq \{\} \longrightarrow opt\text{-}obs\text{-}safe\ (last\text{-}obs\ sc)\ v)$
$\}$

definition *step1-rel* :: (*opt-obsv-state* × *tso-state*)*set* **where**
$step1\text{-}rel = basic\text{-}rel \cap step1\text{-}add\text{-}rel$

definition *tso-ref-rel* :: (*opt-obsv-state* × *tso-state*)*set* **where**
$tso\text{-}ref\text{-}rel \equiv \{(sa,\ sc).$
$(two\text{-}step\ (next\text{-}round\ sc) = 0 \longrightarrow (sa,\ sc) \in step0\text{-}rel)$
$\wedge\ (two\text{-}step\ (next\text{-}round\ sc) = 1 \longrightarrow$
$(sa,\ sc) \in step1\text{-}rel$
$\wedge\ (\exists\, sc'\ r\ S\ v.\ (sc',\ sc) \in tso\text{-}round0\ r\ S\ v \wedge (sa,\ sc') \in step0\text{-}rel))$
$\}$

lemma *const-map-equality*:
$(const\text{-}map\ v\ S = const\text{-}map\ v'\ S') = (S = S' \wedge (S = \{\} \vee v = v'))$
⟨*proof*⟩

lemma *rhoare-skipI*:
$[\![\bigwedge sa\ sc\ sc'.\ [\![(sa,\ sc) \in Pre;\ (sc,\ sc') \in Tc]\!] \Longrightarrow (sa,\ sc') \in Post]\!] \Longrightarrow \{Pre\}$
$Id,\ Tc\ \{>Post\}$
⟨*proof*⟩

lemma *tso-round0-refines*:
$\{tso\text{-}ref\text{-}rel\}\ Id,\ tso\text{-}round0\ r\ S\ v\ \{>tso\text{-}ref\text{-}rel\}$
⟨*proof*⟩

lemma *tso-round1-refines*:
$\{tso\text{-}ref\text{-}rel\} \bigcup r\ S\ v\ dec\text{-}f\ Ob.\ olv\text{-}round\ r\ S\ v\ dec\text{-}f\ Ob,\ tso\text{-}round1\ r\ dec\text{-}f\ o\text{-}f$
$\{>tso\text{-}ref\text{-}rel\}$
⟨*proof*⟩

lemma *TS-Observing-Refines*:

PO-refines tso-ref-rel olv-TS tso-TS
⟨*proof*⟩

11.3 Invariants

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

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

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

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

11.3.1 Proofs of invariants

lemma *TSO-inv1-inductive*:
init tso-TS ⊆ *TSO-inv1*
{*TSO-inv1*} *TS.trans tso-TS* {> *TSO-inv1*}
⟨*proof*⟩

lemma *TSO-inv1-invariant*:
reach tso-TS ⊆ *TSO-inv1*
⟨*proof*⟩

lemma *TSO-inv2-inductive*:
init tso-TS ⊆ *TSO-inv2*
{*TSO-inv2*} *TS.trans tso-TS* {> *TSO-inv2*}
⟨*proof*⟩

lemma *TSO-inv2-invariant*:
reach tso-TS ⊆ *TSO-inv2*
⟨*proof*⟩

end

end

12 The UniformVoting Algorithm

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

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

12.1 Model of the algorithm

abbreviation *nSteps* ≡ *2*

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

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

The following record models the local state of a process.

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

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

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

definition *isValVote* **where** *isValVote m* ≡ ∃ *z v*. *m* = *ValVote z v*

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

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

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

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

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

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

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

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

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

It updates its *last-obs* field to the smallest value it has received. If the process has received the same value v from all processes from which it has heard, it updates its *agreed-vote* field to v.

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

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

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

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

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

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

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

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

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

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

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

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

definition *UV-sendMsg* **where**

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

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

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

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

12.2 The *UniformVoting* Heard-Of machine

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

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

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

end

12.3 Proofs

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

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

instance *val* :: *linorder* ⟨*proof*⟩

lemma *two-step-step*:
  step = *two-step*
  phase = *two-phase*
  ⟨*proof*⟩

context *mono-quorum*
begin

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

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

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

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

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

lemma *step-less-D*:
  0 < *step r* ⟹ *step r* = *Suc 0*
  ⟨*proof*⟩

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

∪ *UV-trans-step HOs SHOs next1 send1 1*

⟨*proof*⟩

12.3.1 Invariants

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

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

lemma *UV-inv1-inductive*:
init (*UV-TS HOs SHOs crds*) ⊆ *UV-inv1*
{*UV-inv1*} *TS.trans* (*UV-TS HOs SHOs crds*) {> *UV-inv1*}
⟨*proof*⟩

lemma *UV-inv1-invariant*:
reach (*UV-TS HOs SHOs crds*) ⊆ *UV-inv1*
⟨*proof*⟩

12.3.2 Refinement

definition *ref-rel* :: (*tso-state* × *uv-TS-state*)*set* **where**
ref-rel ≡ {(*sa*, (*r*, *sc*)).
r = *next-round sa*
∧ (*step r* = *1* ⟶ *r-votes sa* = *agreed-vote o sc*)
∧ (∀ *p v*. *last-obs* (*sc p*) = *v* ⟶ (∃ *q*. *opt-obsv-state.last-obs sa q* ∈ {*None*, *Some v*}))
∧ *decisions sa* = *decide o sc*
}

Agreement for UV only holds if the communication predicates hold

context
fixes
HOs :: *nat* ⇒ *process* ⇒ *process set*

and *rho* :: *nat* ⇒ *process* ⇒ *'val pstate*
assumes *global*: *UV-commGlobal HOs*
and *per-rd*: ∀ *r*. *UV-commPerRd* (*HOs r*)
and *run*: *HORun fA rho HOs*
begin

lemma *HOs-intersect*:
HOs r p ∩ *HOs r' q* ≠ {} ⟨*proof*⟩

lemma *HOs-nonempty*:
HOs r p ≠ {}
⟨*proof*⟩

lemma *vote-origin*:
assumes
send: ∀ *p*. *μ p* ∈ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: ∀ *p*. *next0 r p* (*cfg p*) (*μ p*) (*cfg' p*)
and *inv*: (*r*, *cfg*) ∈ *UV-inv1*
and *step-r*: *two-step r* = *0*
shows
agreed-vote (*cfg' p*) = *Some v* ⟷ (∀ *q* ∈ *HOs r p*. *last-obs* (*cfg q*) = *v*)
⟨*proof*⟩

lemma *same-new-vote*:
assumes
send: ∀ *p*. *μ p* ∈ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: ∀ *p*. *next0 r p* (*cfg p*) (*μ p*) (*cfg' p*)
and *inv*: (*r*, *cfg*) ∈ *UV-inv1*
and *step-r*: *two-step r* = *0*
obtains *v* **where** ∀ *p w*. *agreed-vote* (*cfg' p*) = *Some w* ⟶ *w* = *v*
⟨*proof*⟩

lemma *x-origin1*:
assumes
send: ∀ *p*. *μ p* ∈ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
and *step*: ∀ *p*. *next0 r p* (*cfg p*) (*μ p*) (*cfg' p*)
and *step-r*: *two-step r* = *0*
and *last-obs*: *last-obs* (*cfg' p*) = *v*
shows

$\exists\, q.\ \textit{last-obs}\ (\textit{cfg}\ q) = v$
⟨proof⟩

lemma *step0-ref*:
$\{\textit{ref-rel} \cap \textit{UNIV} \times \textit{UV-inv1}\} \bigcup r\ S\ v.\ \textit{tso-round0}\ r\ S\ v,$
$\textit{UV-trans-step}\ \textit{HOs}\ \textit{HOs}\ \textit{next0}\ \textit{send0}\ 0\ \{>\ \textit{ref-rel}\}$
⟨proof⟩

lemma *x-origin2*:
assumes
send: $\forall\, p.\ \mu\ p \in \textit{get-msgs}\ (\textit{send1}\ r)\ \textit{cfg}\ (\textit{HOs}\ r)\ (\textit{HOs}\ r)\ p$
and *step*: $\forall\, p.\ \textit{next1}\ r\ p\ (\textit{cfg}\ p)\ (\mu\ p)\ (\textit{cfg}'\ p)$
and *step-r*: $\textit{two-step}\ r = \textit{Suc}\ 0$
and *last-obs*: $\textit{last-obs}\ (\textit{cfg}'\ p) = v$
shows
$(\exists\, q.\ \textit{last-obs}\ (\textit{cfg}\ q) = v) \vee (\exists\, q.\ \textit{agreed-vote}\ (\textit{cfg}\ q) = \textit{Some}\ v)$
⟨proof⟩

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

lemma *decide-origin*:
assumes
send: $\forall\, p.\ \mu\ p \in \textit{get-msgs}\ (\textit{send1}\ r)\ \textit{cfg}\ (\textit{HOs}\ r)\ (\textit{HOs}\ r)\ p$
and *step*: $\forall\, p.\ \textit{next1}\ r\ p\ (\textit{cfg}\ p)\ (\mu\ p)\ (\textit{cfg}'\ p)$
and *step-r*: $\textit{two-step}\ r = \textit{Suc}\ 0$
shows
$D\ \textit{cfg}\ \textit{cfg}' \subseteq \{p.\ \exists\, v.\ \textit{decide}\ (\textit{cfg}'\ p) = \textit{Some}\ v \wedge (\forall\, q \in \textit{HOs}\ r\ p.\ \textit{agreed-vote}$
$(\textit{cfg}\ q) = \textit{Some}\ v)\}$
⟨proof⟩

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

lemma *UV-Refines-votes*:
$\textit{PO-refines}\ (\textit{ref-rel} \cap (\textit{TSO-inv1} \cap \textit{TSO-inv2}) \times \textit{UV-inv1})$
$\textit{tso-TS}\ (\textit{UV-TS}\ \textit{HOs}\ \textit{HOs}\ \textit{crds})$

⟨*proof*⟩

end

end

12.3.3 Termination

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

end

13 The Ben-Or Algorithm

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

13.1 The *Ben-Or* Heard-Of machine

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

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

definition (**in** *quorum-process*) *BenOr-HOMachine* **where**
BenOr-HOMachine = (|
CinitState = (λ*p st crd*. *BenOr-initState p st*),
sendMsg = *BenOr-sendMsg*,
CnextState = (λ*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 ≡ (*BenOr-HOMachine*::(*process*, *val pstate*, *val msg*) *HOMachine*)

end

13.2 Proofs

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

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

Ben-Or works only on binary values.

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

context *mono-quorum*
begin

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

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

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

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

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

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

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

⟨*proof*⟩

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

13.2.1 Refinement

Agreement for BenOr only holds if the communication predicates hold

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

definition *no-vote-diff* **where**
no-vote-diff sc p ≡ *vote* (*sc p*) = *None* ⟶
(∃ *q q'*. *x* (*sc q*) ≠ *x* (*sc q'*))

definition *ref-rel* :: (*tso-state* × *ben-or-TS-state*)*set* **where**
ref-rel ≡ {(*sa*, (*r*, *sc*)).
r = *next-round sa*
∧ (*two-step r* = *1* ⟶ *r-votes sa* = *vote o sc*)
∧ (*two-step r* = *1* ⟶ (∀ *p*. *no-vote-diff sc p*))
∧ (∀ *p v*. *x* (*sc p*) = *v* ⟶ (∃ *q*. *last-obs sa q* ∈ {*None*, *Some v*}))
∧ *decisions sa* = *decide o sc*
}

lemma *HOs-intersect*:
HOs r p ∩ *HOs r' q* ≠ {} ⟨*proof*⟩

lemma *HOs-nonempty*:

$HOs\ r\ p \neq \{\}$
⟨*proof*⟩

lemma *vote-origin*:
assumes
send: $\forall p.\ \mu\ p \in get\text{-}msgs\ (send0\ r)\ cfg\ (HOs\ r)\ (HOs\ r)\ p$
and *step*: $\forall p.\ next0\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p)$
and *step-r*: $two\text{-}step\ r = 0$
shows
$vote\ (cfg'\ p) = Some\ v \longleftrightarrow (\forall q \in HOs\ r\ p.\ x\ (cfg\ q) = v)$
⟨*proof*⟩

lemma *same-new-vote*:
assumes
send: $\forall p.\ \mu\ p \in get\text{-}msgs\ (send0\ r)\ cfg\ (HOs\ r)\ (HOs\ r)\ p$
and *step*: $\forall p.\ next0\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p)$
and *step-r*: $two\text{-}step\ r = 0$
obtains v **where** $\forall p\ w.\ vote\ (cfg'\ p) = Some\ w \longrightarrow w = v$
⟨*proof*⟩

lemma *no-x-change*:
assumes
send: $\forall p.\ \mu\ p \in get\text{-}msgs\ (send0\ r)\ cfg\ (HOs\ r)\ (HOs\ r)\ p$
and *step*: $\forall p.\ next0\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p)$
and *step-r*: $two\text{-}step\ r = 0$
shows
$x\ (cfg'\ p) = x\ (cfg\ p)$
⟨*proof*⟩

lemma *no-vote*:
assumes
send: $\forall p.\ \mu\ p \in get\text{-}msgs\ (send0\ r)\ cfg\ (HOs\ r)\ (HOs\ r)\ p$
and *step*: $\forall p.\ next0\ r\ p\ (cfg\ p)\ (\mu\ p)\ (cfg'\ p)$
and *step-r*: $two\text{-}step\ r = 0$
shows
$no\text{-}vote\text{-}diff\ cfg'\ p$
⟨*proof*⟩

lemma *step0-ref*:

$\{ref\text{-}rel\} \bigcup r\ S\ v.\ tso\text{-}round0\ r\ S\ v,$
  $BenOr\text{-}trans\text{-}step\ HOs\ HOs\ next0\ send0\ 0\ \{>\ ref\text{-}rel\}$
⟨*proof*⟩

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

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

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

lemma *BenOr-Refines-Two-Step-Obs*:
$PO\text{-}refines\ (ref\text{-}rel \cap (TSO\text{-}inv1 \cap TSO\text{-}inv2) \times UNIV)$
  $tso\text{-}TS\ (BenOr\text{-}TS\ HOs\ HOs\ crds)$
⟨*proof*⟩

13.2.2 Termination

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

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

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

end

end

end

14 The MRU Vote Model

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

context *quorum-process*
begin

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

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

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

lemma *mru-vote-implies-safe*:
  assumes
    inv4: *s* ∈ *SV-inv4*
    and *inv1*: *s* ∈ *Vinv1*
    and *mru-vote*: *mru-guard s Q v*
    and *is-Quorum*: *Q* ∈ *Quorum*
  shows *safe s* (*v-state.next-round s*) *v* ⟨*proof*⟩

end

end

15 Optimized MRU Vote Model

theory *MRU-Vote-Opt*
imports *MRU-Vote*
begin

15.1 Model definition

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

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

context *quorum-process* **begin**

definition *opt-mru-vote* :: (*process*, *round* × *val*)*map* ⇒ (*process set*, *round* × *val*)*map* **where**
  opt-mru-vote lvs Q = *option-Max-by fst* (*ran* (*lvs* |‘ *Q*))

definition *opt-mru-guard* :: (*process*, *round* × *val*)*map* ⇒ *process set* ⇒ *val* ⇒ *bool* **where**
  opt-mru-guard mru-votes Q v ≡ *Q* ∈ *Quorum* ∧
    (*let mru* = *opt-mru-vote mru-votes Q in mru* = *None* ∨ (∃ *r*. *mru* = *Some* (*r*, *v*)))

definition *opt-mru-round*
 :: *round* ⇒ *process set* ⇒ *process set* ⇒ *val* ⇒ (*process*, *val*)*map* ⇒ (*opt-mru-state* × *opt-mru-state*) *set*
  where
  opt-mru-round r Q S v r-decisions = {(*s*, *s*′).
    — guards
    r = *next-round s*
    ∧ (*S* ≠ {} ⟶ *opt-mru-guard* (*mru-vote s*) *Q v*)
    ∧ *d-guard r-decisions* (*const-map v S*)
    ∧ — actions
    s′ = *s*⦇

mru-vote := *mru-vote s* ++ *const-map* (*r*, *v*) *S*
, *next-round* := *Suc r*
, *decisions* := *decisions s* ++ *r-decisions*
⦈
}

lemmas *lv-evt-defs* = *opt-mru-round-def opt-mru-guard-def*

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

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

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

15.2 Refinement

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

15.2.1 The concrete guard implies the abstract guard

definition *voters* :: (*round* ⇒ (*process*, *val*)*map*) ⇒ *process set* **where**
voters vs = {*a*|*a r v*. ((*r*, *a*), *v*) ∈ *map-graph* (*case-prod vs*)}

lemma *vote-set-as-Union*:
vote-set vs Q = (⋃ *a*∈(*Q* ∩ *voters vs*). *vote-set vs* {*a*})
⟨*proof*⟩

lemma *empty-ran*:
(*ran f* = {}) = (∀ *x*. *f x* = *None*)
⟨*proof*⟩

lemma *empty-ran-restrict*:
$(ran\ (f\ |`\ A) = \{\}) = (\forall x \in A.\ f\ x = None)$
⟨*proof*⟩

lemma *option-Max-by-eqI*:
$[\![(S = \{\}) \longleftrightarrow (S' = \{\});\ S \neq \{\} \wedge S' \neq \{\} \Longrightarrow Max\text{-}by\ f\ S = Max\text{-}by\ g\ S']\!]$
$\Longrightarrow\ option\text{-}Max\text{-}by\ f\ S = option\text{-}Max\text{-}by\ g\ S'$
⟨*proof*⟩

lemma *ran-process-mru-only-voters*:
$ran\ (process\text{-}mru\ vs\ |`\ Q) = ran\ (process\text{-}mru\ vs\ |`\ (Q \cap voters\ vs))$
⟨*proof*⟩

lemma *SV-inv3-inj-on-fst-vote-set*:
$s \in SV\text{-}inv3 \Longrightarrow inj\text{-}on\ fst\ (vote\text{-}set\ (votes\ s)\ Q)$
⟨*proof*⟩

lemma *opt-mru-vote-mru-of-set*:
assumes
inv1: $s \in Vinv1$
and *inv3*: $s \in SV\text{-}inv3$
defines $vs \equiv votes\ s$
shows
$opt\text{-}mru\text{-}vote\ (process\text{-}mru\ vs)\ Q = mru\text{-}of\text{-}set\ vs\ Q$
⟨*proof*⟩

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

15.2.2 The concrete action refines the abstract action

lemma *act-ref*:
assumes
$s \in Vinv1$
shows
$process\text{-}mru\ (votes\ s) +\!+\ const\text{-}map\ (v\text{-}state.next\text{-}round\ s,\ v)\ S$
$= process\text{-}mru\ ((votes\ s)(v\text{-}state.next\text{-}round\ s := const\text{-}map\ v\ S))$

⟨*proof*⟩

15.2.3 The complete refinement

lemma *opt-mru-guard-imp-Quorum*:
opt-mru-guard vs Q v $\Longrightarrow$ *Q* $\in$ *Quorum*
⟨*proof*⟩

lemma *opt-mru-round-refines*:
{*lv-ref-rel* ∩ (*Vinv1* ∩ *SV-inv3* ∩ *SV-inv4*) × *UNIV*}
sv-round r S v d-f, *opt-mru-round r Q S v d-f*
{> *lv-ref-rel*}
⟨*proof*⟩

lemma *Opt-MRU-Vote-Refines*:
PO-refines (*lv-ref-rel* ∩ (*Vinv1* ∩ *Vinv2* ∩ *SV-inv3* ∩ *SV-inv4*) × *UNIV*) *sv-TS mru-opt-TS*
⟨*proof*⟩

15.3 Invariants

definition *OMRU-inv1* :: *opt-mru-state set* **where**
OMRU-inv1 = {*s*. ∀ *p*. (*case mru-vote s p of*
Some (*r*, -) ⇒ *r* < *next-round s*
| *None* ⇒ *True*)
}

lemma *OMRU-inv1-inductive*:
init mru-opt-TS ⊆ *OMRU-inv1*
{*OMRU-inv1*} *trans mru-opt-TS* {> *OMRU-inv1*}
⟨*proof*⟩

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

end

end

16 Three-step Optimized MRU Model

theory *Three-Step-MRU*
imports ../*MRU-Vote-Opt Three-Steps*
begin

To make the coming proofs of concrete algorithms easier, in this model we split the *opt-mru-round* into three steps

16.1 Model definition

record *three-step-mru-state* = *opt-mru-state* +
  candidates :: *val set*

context *mono-quorum*
begin

definition *opt-mru-step0* :: *round* ⇒ *val set* ⇒ (*three-step-mru-state* × *three-step-mru-state*) *set* **where**
  opt-mru-step0 r C = {(*s*, *s'*).
    — guards
    r = *next-round s* ∧ *three-step r* = *0*
    ∧ (∀ *cand* ∈ *C*. ∃ *Q*. *opt-mru-guard* (*mru-vote s*) *Q cand*)
    ∧ — actions
    s' = *s*⦇
      candidates := *C*
      , *next-round* := *Suc r*
    ⦈
  }

definition *opt-mru-step1* :: *round* ⇒ *process set* ⇒ *val* ⇒
  (*three-step-mru-state* × *three-step-mru-state*) *set* **where**
  opt-mru-step1 r S v = {(*s*, *s'*).
    — guards
    r = *next-round s* ∧ *three-step r* = *1*
    ∧ (*S* ≠ {} ⟶ *v* ∈ *candidates s*)
    ∧ — actions
    s' = *s*⦇
      mru-vote := *mru-vote s* ++ *const-map* (*three-phase r*, *v*) *S*
      , *next-round* := *Suc r*

$\|)$
}

definition *step2-d-guard* :: (*process*, *val*)*map* ⇒ (*process*, *val*)*map* ⇒ *bool* **where**
step2-d-guard r-decisions r-votes ≡ ∀ *p v*. *r-decisions p* = *Some v* ⟶
v ∈ *ran r-votes* ∧ *dom r-votes* ∈ *Quorum*

definition *r-votes* :: *three-step-mru-state* ⇒ *round* ⇒ (*process*, *val*)*map* **where**
r-votes s r ≡ λ*p*. *if* (∃ *v*. *mru-vote s p* = *Some* (*three-phase r*, *v*))
then map-option snd (*mru-vote s p*)
else None

definition *opt-mru-step2* :: *round* ⇒ (*process*, *val*)*map* ⇒ (*three-step-mru-state* × *three-step-mru-state*) *set* **where**
opt-mru-step2 r r-decisions = {(*s*, *s'*).
— guards
r = *next-round s* ∧ *three-step r* = *2*
∧ *step2-d-guard r-decisions* (*r-votes s r*)
∧ — actions
s' = *s*(|
next-round := *Suc r*
, *decisions* := *decisions s* ++ *r-decisions*
|)
}

lemmas *ts-mru-evt-defs* = *opt-mru-step0-def opt-mru-step1-def opt-mru-guard-def*

definition *ts-mru-trans* :: (*three-step-mru-state* × *three-step-mru-state*) *set* **where**
ts-mru-trans = (⋃ *r C*. *opt-mru-step0 r C*)
∪ (⋃ *r S v*. *opt-mru-step1 r S v*)
∪ (⋃ *r dec-f*. *opt-mru-step2 r dec-f*) ∪ *Id*

definition *ts-mru-init* **where**
ts-mru-init = { (| *next-round* = *0*, *mru-vote* = *Map.empty*, *decisions* = *Map.empty*, *candidates* = {} |) }

definition *ts-mru-TS* :: *three-step-mru-state TS* **where**
ts-mru-TS = (| *init* = *ts-mru-init*, *trans* = *ts-mru-trans* |)

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

16.2 Refinement

definition *basic-rel* **where**
$basic\text{-}rel \equiv \{(sa,\ sc).$
$decisions\ sc = decisions\ sa$
$\wedge\ next\text{-}round\ sa = three\text{-}phase\ (next\text{-}round\ sc)$
$\}$

definition *three-step0-rel* :: (*opt-mru-state* × *three-step-mru-state*)*set* **where**
$three\text{-}step0\text{-}rel \equiv basic\text{-}rel \cap \{(sa,\ sc).$
$three\text{-}step\ (next\text{-}round\ sc) = 0$
$\wedge\ mru\text{-}vote\ sc = mru\text{-}vote\ sa$
$\}$

definition *three-step1-rel* :: (*opt-mru-state* × *three-step-mru-state*)*set* **where**
$three\text{-}step1\text{-}rel \equiv basic\text{-}rel \cap \{(sa,\ sc).$
$(\exists\, sc'\ r\ C.\ (sa,\ sc') \in three\text{-}step0\text{-}rel \wedge (sc',\ sc) \in opt\text{-}mru\text{-}step0\ r\ C)$
$\wedge\ mru\text{-}vote\ sc = mru\text{-}vote\ sa$
$\}$

definition *three-step2-rel* :: (*opt-mru-state* × *three-step-mru-state*)*set* **where**
$three\text{-}step2\text{-}rel \equiv basic\text{-}rel \cap \{(sa,\ sc).$
$(\exists\, sc'\ r\ S\ v.\ (sa,\ sc') \in three\text{-}step1\text{-}rel \wedge (sc',\ sc) \in opt\text{-}mru\text{-}step1\ r\ S\ v)$
$\}$

definition *ts-ref-rel* **where**
$ts\text{-}ref\text{-}rel = \{(sa,\ sc).$
$(three\text{-}step\ (next\text{-}round\ sc) = 0 \longrightarrow (sa,\ sc) \in three\text{-}step0\text{-}rel)$
$\wedge\ (three\text{-}step\ (next\text{-}round\ sc) = 1 \longrightarrow (sa,\ sc) \in three\text{-}step1\text{-}rel)$
$\wedge\ (three\text{-}step\ (next\text{-}round\ sc) = 2 \longrightarrow (sa,\ sc) \in three\text{-}step2\text{-}rel)$
$\}$

lemmas *ts-ref-rel-defs* =
basic-rel-def
ts-ref-rel-def
three-step0-rel-def
three-step1-rel-def
three-step2-rel-def

lemma *step0-ref*:
  {*ts-ref-rel*} *Id*, *opt-mru-step0 r C* {> *ts-ref-rel*}
⟨*proof*⟩

lemma *step1-ref*:
  {*ts-ref-rel*} *Id*, *opt-mru-step1 r S v* {> *ts-ref-rel*}
⟨*proof*⟩

lemma *step2-ref*:
  {*ts-ref-rel* ∩ *OMRU-inv1* × *UNIV*}
    $\bigcup$ *r′ Q S′ v dec-f′*. *opt-mru-round r′ Q S′ v dec-f′*,
    opt-mru-step2 r dec-f {> *ts-ref-rel*}
⟨*proof*⟩

lemma *ThreeStep-Coordinated-Refines*:
  PO-refines (*ts-ref-rel* ∩ *OMRU-inv1* × *UNIV*)
    mru-opt-TS ts-mru-TS
⟨*proof*⟩

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* ⟨*proof*⟩

record *pstate* =

x :: *val* — current value held by process
prop-vote :: *val option*
mru-vote :: (*nat* × *val*) *option*
decide :: *val option* — value the process has decided on, if any

datatype *msg* =
MruVote (*nat* × *val*) *option val*
| *PreVote val*
| *Vote val*
| *Null* — dummy message in case nothing needs to be sent

Characteristic predicates on messages.

definition *isLV* **where** *isLV m* ≡ ∃ *rv*. *m* = *Vote rv*

definition *isPreVote* **where** *isPreVote m* ≡ ∃ *px*. *m* = *PreVote px*

definition *NA-initState* **where**
NA-initState p st - ≡
mru-vote st = *None*
∧ *prop-vote st* = *None*
∧ *decide st* = *None*

definition *send0* **where**
send0 r p q st ≡ *MruVote* (*mru-vote st*) (*x st*)

fun *msg-to-val-stamp* :: *msg* ⇒ (*round* × *val*)*option* **where**
msg-to-val-stamp (*MruVote rv -*) = *rv*

definition *msgs-to-lvs* ::
(*process* ⇀ *msg*)
⇒ (*process*, *round* × *val*) *map*
where
msgs-to-lvs msgs ≡ *msg-to-val-stamp* $\circ_m$ *msgs*

definition *smallest-proposal* **where**
smallest-proposal (*msgs*::*process* ⇀ *msg*) ≡
Min {*v*. ∃ *q mv*. *msgs q* = *Some* (*MruVote mv v*)}

definition *next0*

$:: nat$
$\Rightarrow process$
$\Rightarrow pstate$
$\Rightarrow (process \rightharpoonup msg)$
$\Rightarrow process$
$\Rightarrow pstate$
$\Rightarrow bool$

where

$next0\ r\ p\ st\ msgs\ crd\ st' \equiv let$
$\quad Q = dom\ msgs;$
$\quad lvs = msgs\text{-}to\text{-}lvs\ msgs;$
$\quad smallest = if\ Q = \{\}\ then\ x\ st\ else\ smallest\text{-}proposal\ msgs$
in
$st' = st\ (\!|$
$\quad prop\text{-}vote := if\ card\ Q > N\ div\ 2$
$\quad\quad then\ Some\ (case\text{-}option\ smallest\ snd\ (option\text{-}Max\text{-}by\ fst\ (ran\ (lvs\ |`\ Q))))$
$\quad\quad else\ None$
$|\!)$

definition *send1* **where**

$send1\ r\ p\ q\ st \equiv case\ prop\text{-}vote\ st\ of$
$\quad None \Rightarrow Null$
$\quad |\ Some\ v \Rightarrow PreVote\ v$

definition *Q-prevotes-v* **where**

$Q\text{-}prevotes\text{-}v\ msgs\ Q\ v \equiv let\ D = dom\ msgs\ in$
$\quad Q \subseteq D \land card\ Q > N\ div\ 2 \land (\forall q \in Q.\ msgs\ q = Some\ (PreVote\ v))$

definition *next1*

$:: nat$
$\Rightarrow process$
$\Rightarrow pstate$
$\Rightarrow (process \rightharpoonup msg)$
$\Rightarrow process$
$\Rightarrow pstate$
$\Rightarrow bool$

where

$next1\ r\ p\ st\ msgs\ crd\ st' \equiv$
$\quad decide\ st' = decide\ st$
$\quad \land x\ st' = x\ st$

$\wedge$ ($\forall$ *Q v. Q-prevotes-v msgs Q v*
$\longrightarrow$ *mru-vote st′* = *Some* (*three-phase r, v*))
$\wedge$ ($\neg$ ($\exists$ *Q v. Q-prevotes-v msgs Q v*)
$\longrightarrow$ *mru-vote st′* = *mru-vote st*)

definition *send2* **where**
send2 r p q st $\equiv$ *case mru-vote st of*
None $\Rightarrow$ *Null*
| *Some* (Φ, *v*) $\Rightarrow$ *if* Φ = *three-phase r then Vote v else Null*

definition *Q′-votes-v* **where**
Q′-votes-v r msgs Q Q′ v $\equiv$
Q′ $\subseteq$ *Q* $\wedge$ *card Q′* > *N div 2* $\wedge$ ($\forall$ *q* $\in$ *Q′. msgs q* = *Some* (*Vote v*))

definition *next2*
:: *nat*
$\Rightarrow$ *process*
$\Rightarrow$ *pstate*
$\Rightarrow$ (*process* $\rightharpoonup$ *msg*)
$\Rightarrow$ *process*
$\Rightarrow$ *pstate*
$\Rightarrow$ *bool*
where
next2 r p st msgs crd st′ $\equiv$ *let Q* = *dom msgs*; *lvs* = *msgs-to-lvs msgs in*
x st′ = *x st*
$\wedge$ *mru-vote st′* = *mru-vote st*
$\wedge$ ($\forall$ *Q′ v. Q′-votes-v r msgs Q Q′ v* $\longrightarrow$ *decide st′* = *Some v*)
$\wedge$ ($\neg$($\exists$ *Q′ v. Q′-votes-v r msgs Q Q′ v* $\longrightarrow$ *decide st′* = *decide st*))

definition *NA-sendMsg* :: *nat* $\Rightarrow$ *process* $\Rightarrow$ *process* $\Rightarrow$ *pstate* $\Rightarrow$ *msg* **where**
NA-sendMsg (*r*::*nat*) $\equiv$
if three-step r = *0 then send0 r*
else if three-step r = *1 then send1 r*
else send2 r

definition
NA-nextState :: *nat* $\Rightarrow$ *process* $\Rightarrow$ *pstate* $\Rightarrow$ (*process* $\rightharpoonup$ *msg*)
$\Rightarrow$ *process* $\Rightarrow$ *pstate* $\Rightarrow$ *bool*

where
NA-nextState r ≡
if three-step r = 0 then next0 r
else if three-step r = 1 then next1 r
else next2 r

17.2 The Heard-Of machine

definition
NA-commPerRd **where**
NA-commPerRd (*HOrs*::*process HO*) ≡ *True*

definition
NA-commGlobal **where**
NA-commGlobal HOs ≡
∃ *ph*::*nat*. ∀ *i* ∈ {*0*..*2*}.
(∀ *p*. *card* (*HOs* (*nr-steps*∗*ph*+*i*) *p*) > *N div 2*)
∧ (∀ *p q*. *HOs* (*nr-steps*∗*ph*+*i*) *p* = *HOs* (*nr-steps*∗*ph*) *q*)

definition *New-Algo-Alg* **where**
New-Algo-Alg ≡
(| *CinitState* = *NA-initState*,
sendMsg = *NA-sendMsg*,
CnextState = *NA-nextState* |)

definition *New-Algo-HOMachine* **where**
New-Algo-HOMachine ≡
(| *CinitState* = *NA-initState*,
sendMsg = *NA-sendMsg*,
CnextState = *NA-nextState*,
HOcommPerRd = *NA-commPerRd*,
HOcommGlobal = *NA-commGlobal* |)

abbreviation
New-Algo-M ≡ (*New-Algo-HOMachine*::(*process*, *pstate*, *msg*) *HOMachine*)

end

17.3 Proofs

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

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

lemmas *New-Algo-TS-defs* = *New-Algo-TS-def CHO-to-TS-def New-Algo-Alg-def CHOinitConfig-def*
NA-initState-def

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

lemma *three-step-less-D*:
0 < *three-step r* ⟹ *three-step r* = *1* ∨ *three-step r* = *2*
⟨*proof*⟩

lemma *New-Algo-trans*:
CSHO-trans-alt NA-sendMsg NA-nextState HOs SHOs (*K* ∘ *crds*) =
New-Algo-trans-step HOs SHOs crds next0 send0 0
∪ *New-Algo-trans-step HOs SHOs crds next1 send1 1*
∪ *New-Algo-trans-step HOs SHOs crds next2 send2 2*

⟨*proof*⟩

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

17.3.1 Refinement

definition *new-algo-ref-rel* :: (*three-step-mru-state* × *p-TS-state*)*set* **where**
new-algo-ref-rel = {(*sa*, (*r*, *sc*)).
opt-mru-state.next-round sa = *r*

$\wedge$ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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. next1 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) $\Longrightarrow$ *decide o s* = *decide o s′*
⟨*proof*⟩

lemma *msgs-mru-vote*:
assumes
μ *p* $\in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) *p*
shows ((*msgs-to-lvs* (μ *p*)) |‘ *HOs r p*) = (*mru-vote o cfg*) |‘ *HOs r p* ⟨*proof*⟩

lemma *step0-ref*:
{*new-algo-ref-rel*}
($\bigcup$ *r C. majorities.opt-mru-step0 r C*),
New-Algo-trans-step HOs HOs crds next0 send0 0 {> *new-algo-ref-rel*}
⟨*proof*⟩

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*⟩

lemma *step2-ref*:
{*new-algo-ref-rel*}

($\bigcup$ *r dec-f. majorities.opt-mru-step2 r dec-f*),
New-Algo-trans-step HOs HOs crds next2 send2 2 {> *new-algo-ref-rel*}
⟨*proof*⟩

lemma *New-Algo-Refines-votes*:
PO-refines new-algo-ref-rel
majorities.ts-mru-TS (*New-Algo-TS HOs HOs crds*)
⟨*proof*⟩

17.3.2 Termination

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

end

18 The Paxos Algorithm

theory *Paxos-Defs*
imports *Heard-Of.HOModel ../Consensus-Types ../Consensus-Misc Three-Steps*
begin

This is a modified version (closer to the original Paxos) of PaxosDefs from the Heard Of entry in the AFP.

18.1 Model of the algorithm

The following record models the local state of a process.

record *'val pstate* =
x :: *'val* — current value held by process
mru-vote :: (*nat* × *'val*) *option*
commt :: *'val option* — for coordinators: the value processes are asked to commit to
decide :: *'val option* — value the process has decided on, if any

The algorithm relies on a coordinator for each phase of the algorithm. A phase lasts three rounds. The HO model formalization already provides the infrastructure for this, but unfortunately the coordinator is not passed to the *sendMsg* function. Using the infrastructure would thus require additional invariants and proofs; for simplicity, we use a global constant instead.

consts *coord* :: *nat* ⇒ *process*
specification (*coord*)
  coord-phase[*rule-format*]: ∀ *r r'*. *three-phase r* = *three-phase r'* ⟶ *coord r* = *coord r'*
  ⟨*proof*⟩

Possible messages sent during the execution of the algorithm.

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

Characteristic predicates on messages.

definition *isValStamp* **where** *isValStamp m* ≡ ∃ *v ts*. *m* = *ValStamp v ts*

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

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

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

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

definition *Paxos-initState* **where**
  Paxos-initState p st crd ≡
   mru-vote st = *None*
   ∧ *commt st* = *None*
   ∧ *decide st* = *None*

definition *mru-vote-to-msg* :: *'val pstate* ⇒ *'val msg* **where**

mru-vote-to-msg st ≡ case mru-vote st of
Some (ts, v) ⇒ ValStamp v ts
| None ⇒ NeverVoted

fun *msg-to-val-stamp* :: *'val msg ⇒ (round × 'val)option* **where**
msg-to-val-stamp (ValStamp v ts) = Some (ts, v)
| msg-to-val-stamp - = None

definition *msgs-to-lvs* ::
(process ⇀ 'val msg)
⇒ (process, round × 'val) map
where
msgs-to-lvs msgs ≡ msg-to-val-stamp $\circ_m$ *msgs*

definition *send0* **where**
send0 r p q st ≡
if q = coord r then mru-vote-to-msg st else Null

definition *next0*
:: *nat*
⇒ process
⇒ 'val pstate
⇒ (process ⇀ 'val msg)
⇒ process
⇒ 'val pstate
⇒ bool
where
next0 r p st msgs crd st' ≡ let Q = dom msgs; lvs = msgs-to-lvs msgs in
if p = coord r ∧ card Q > N div 2
then (st' = st ⦇ commt := Some (case-option (x st) snd (option-Max-by fst
(ran (lvs |‘ Q)))) ⦈)
else st' = st⦇ commt := None ⦈

definition *send1* **where**
send1 r p q st ≡
if p = coord r ∧ commt st ≠ None then Vote (the (commt st)) else Null

definition *next1*
:: *nat*
⇒ process

⇒ ′*val pstate*
⇒ (*process* ⇀ ′*val msg*)
⇒ *process*
⇒ ′*val pstate*
⇒ *bool*

where

next1 r p st msgs crd st′ ≡
if msgs (*coord r*) ≠ *None* ∧ *isVote* (*the* (*msgs* (*coord r*)))
then st′ = *st* ⦇ *mru-vote* := *Some* (*three-phase r*, *val* (*the* (*msgs* (*coord r*)))) ⦈
else st′ = *st*

definition *send2* **where**

send2 r p q st ≡ (*case mru-vote st of*
Some (*phs*, *v*) ⇒ (*if phs* = *three-phase r then Vote v else Null*)
| - ⇒ *Null*
)

— processes from which a vote was received

definition *votes-rcvd* **where**

votes-rcvd (*msgs* :: *process* ⇀ ′*val msg*) ≡
{ (*q*, *v*) . *msgs q* = *Some* (*Vote v*) }

definition *the-rcvd-vote* **where**

the-rcvd-vote (*msgs* :: *process* ⇀ ′*val msg*) ≡ *SOME v*. *v* ∈ *snd* ‘ *votes-rcvd msgs*

definition *next2* **where**

next2 r p st msgs crd st′ ≡
if card (*votes-rcvd msgs*) > *N div 2*
then st′ = *st* ⦇ *decide* := *Some* (*the-rcvd-vote msgs*) ⦈
else st′ = *st*

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

definition *Paxos-sendMsg* :: *nat* ⇒ *process* ⇒ *process* ⇒ ′*val pstate* ⇒ ′*val msg*
where

Paxos-sendMsg (*r*::*nat*) ≡
if three-step r = *0 then send0 r*
else if three-step r = *1 then send1 r*
else send2 r

definition
$Paxos\text{-}nextState :: nat \Rightarrow process \Rightarrow {'}val\ pstate \Rightarrow (process \rightharpoonup {'}val\ msg)$
$\quad \Rightarrow process \Rightarrow {'}val\ pstate \Rightarrow bool$
where
$Paxos\text{-}nextState\ r \equiv$
$\ \ if\ three\text{-}step\ r = 0\ then\ next0\ r$
$\ \ else\ if\ three\text{-}step\ r = 1\ then\ next1\ r$
$\ \ else\ next2\ r$

definition
Paxos-commPerRd **where**
$Paxos\text{-}commPerRd\ r\ (HO{::}process\ HO)\ (crd{::}process\ coord) \equiv True$

definition
Paxos-commGlobal **where**
$Paxos\text{-}commGlobal\ HOs\ coords \equiv$
$\quad \exists\, ph{::}nat.\ \exists\, c{::}process.$
$\quad\quad coord\ (nr\text{-}steps{*}ph) = c$
$\quad\quad \wedge\ card\ (HOs\ (nr\text{-}steps{*}ph)\ c) > N\ div\ 2$
$\quad\quad \wedge\ (\forall\, p.\ c \in HOs\ (nr\text{-}steps{*}ph{+}1)\ p)$
$\quad\quad \wedge\ (\forall\, p.\ card\ (HOs\ (nr\text{-}steps{*}ph{+}2)\ p) > N\ div\ 2)$

18.2 The *Paxos* Heard-Of machine

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

definition *Paxos-Alg* **where**
$Paxos\text{-}Alg \equiv$
$\quad (\!|\ CinitState = Paxos\text{-}initState,$
$\quad\ \ sendMsg = Paxos\text{-}sendMsg,$
$\quad\ \ CnextState = Paxos\text{-}nextState\ |\!)$

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

CHOcommGlobal = *Paxos-commGlobal* ⦈

abbreviation
Paxos-M ≡ (*Paxos-CHOMachine*::(*process*, ′*val pstate*, ′*val msg*) *CHOMachine*)

end

18.3 Proofs

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

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

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

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

lemma *three-step-less-D*:
0 < *three-step r* ⟹ *three-step r* = *1* ∨ *three-step r* = *2*
⟨*proof*⟩

lemma *Paxos-trans*:
CSHO-trans-alt Paxos-sendMsg Paxos-nextState HOs SHOs (*K* ∘ *crds*) =
Paxos-trans-step HOs SHOs crds next0 send0 0
∪ *Paxos-trans-step HOs SHOs crds next1 send1 1*
∪ *Paxos-trans-step HOs SHOs crds next2 send2 2*

⟨*proof*⟩

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

18.3.1 Refinement

definition *coord-vote-to-set* :: *nat* ⇒ (*process* ⇒ (*val pstate*)) ⇒ *val set* **where**
  coord-vote-to-set r sc ≡ (*let v* = *pstate.commt* (*sc* (*coord r*)) *in*
      if v = *None*
        then {}
        else {*the v*})

definition *paxos-ref-rel* :: (*three-step-mru-state* × *p-TS-state*)*set* **where**
  paxos-ref-rel = {(*sa*, (*r*, *sc*)).
    opt-mru-state.next-round sa = *r*
    ∧ *opt-mru-state.decisions sa* = *pstate.decide o sc*
    ∧ *opt-mru-state.mru-vote sa* = *pstate.mru-vote o sc*
   ∧ (*three-step r* = *Suc 0* ⟶ *three-step-mru-state.candidates sa* = *coord-vote-to-set*
r sc)
  }

lemma *mru-vote-evolution0*:
  ∀ *p*. *next0 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *mru-vote o s′* = *mru-vote o s*
  ⟨*proof*⟩

lemma *mru-vote-evolution2*:
  ∀ *p*. *next2 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *mru-vote o s′* = *mru-vote o s*
  ⟨*proof*⟩

lemma *decide-evolution*:
  ∀ *p*. *next0 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *decide o s* = *decide o s′*
  ∀ *p*. *next1 r p* (*s p*) (*msgs p*) (*crd p*) (*s′ p*) ⟹ *decide o s* = *decide o s′*
  ⟨*proof*⟩

lemma *msgs-mru-vote*:
  assumes
  μ (*coord r*) ∈ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) (*coord r*) (**is** μ *?p* ∈ -)
  shows ((*msgs-to-lvs* (μ *?p*)) |‘ *HOs r ?p*) = (*mru-vote o cfg*) |‘ *HOs r ?p* ⟨*proof*⟩

lemma *step0-ref*:

{*paxos-ref-rel*}
  ($\bigcup$ *r C. majorities.opt-mru-step0 r C*),
  Paxos-trans-step HOs HOs crds next0 send0 0 {> *paxos-ref-rel*}
⟨*proof*⟩

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*⟩

lemma *step2-ref*:
{*paxos-ref-rel*}
  ($\bigcup$ *r dec-f. majorities.opt-mru-step2 r dec-f*),
  Paxos-trans-step HOs HOs crds next2 send2 2 {> *paxos-ref-rel*}
⟨*proof*⟩

lemma *Paxos-Refines-ThreeStep-MRU*:
PO-refines paxos-ref-rel
  majorities.ts-mru-TS (*Paxos-TS HOs HOs crds*)
⟨*proof*⟩

18.3.2 Termination

theorem *Paxos-termination*:
assumes *run*: *CHORun Paxos-Alg rho HOs crds*
  and *commR*: ∀ *r. CHOcommPerRd Paxos-M r* (*HOs r*) (*crds r*)
  and *commG*: *CHOcommGlobal Paxos-M HOs crds*
shows ∃ *r v. decide* (*rho r p*) = *Some v*
⟨*proof*⟩

end

19 Chandra-Toueg $\diamond S$ Algorithm

theory *CT-Defs*
imports *Heard-Of.HOModel ../Consensus-Types ../Consensus-Misc Three-Steps*
begin

The following record models the local state of a process.

record *'val pstate* =
x :: *'val* — current value held by process
mru-vote :: (*nat* × *'val*) *option*
commt :: *'val* — for coordinators: the value processes are asked to commit to
decide :: *'val option* — value the process has decided on, if any

The algorithm relies on a coordinator for each phase of the algorithm. A phase lasts three rounds. The HO model formalization already provides the infrastructure for this, but unfortunately the coordinator is not passed to the *sendMsg* function. Using the infrastructure would thus require additional invariants and proofs; for simplicity, we use a global constant instead.

consts *coord* :: *nat* ⇒ *process*
specification (*coord*)
coord-phase[*rule-format*]: $\forall\, r\ r'.\ \textit{three-phase}\ r = \textit{three-phase}\ r' \longrightarrow \textit{coord}\ r = \textit{coord}\ r'$
⟨*proof*⟩

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** $\textit{isValStamp}\ m \equiv \exists\, v\ ts.\ m = \textit{ValStamp}\ v\ ts$

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

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 ≡

mru-vote st = *None*
∧ *decide st* = *None*

definition *mru-vote-to-msg* :: *'val pstate* ⇒ *'val msg* **where**
mru-vote-to-msg st ≡ *case mru-vote st of*
Some (*ts*, *v*) ⇒ *ValStamp v ts*
| *None* ⇒ *NeverVoted*

fun *msg-to-val-stamp* :: *'val msg* ⇒ (*round* × *'val*)*option* **where**
msg-to-val-stamp (*ValStamp v ts*) = *Some* (*ts*, *v*)
| *msg-to-val-stamp* - = *None*

definition *msgs-to-lvs* ::
(*process* ⇀ *'val msg*)
⇒ (*process*, *round* × *'val*) *map*
where
msgs-to-lvs msgs ≡ *msg-to-val-stamp* $\circ_m$ *msgs*

definition *send0* **where**
send0 r p q st ≡
if q = *coord r then mru-vote-to-msg st else Null*

definition *next0*
:: *nat*
⇒ *process*
⇒ *'val pstate*
⇒ (*process* ⇀ *'val msg*)
⇒ *process*
⇒ *'val pstate*
⇒ *bool*
where
next0 r p st msgs crd st' ≡ *let Q* = *dom msgs*; *lvs* = *msgs-to-lvs msgs in*
if p = *coord r*
then (*st'* = *st* ⦇ *commt* := (*case-option* (*x st*) *snd* (*option-Max-by fst* (*ran* (*lvs* |' *Q*)))) ⦈)
else st' = *st*

definition *send1* **where**
send1 r p q st ≡

$if\ p = coord\ r\ then\ Vote\ (commt\ st)\ else\ Null$

definition *next1*
$:: nat$
$\Rightarrow process$
$\Rightarrow {'val}\ pstate$
$\Rightarrow (process \rightharpoonup {'val}\ msg)$
$\Rightarrow process$
$\Rightarrow {'val}\ pstate$
$\Rightarrow bool$
where
$next1\ r\ p\ st\ msgs\ crd\ st' \equiv$
$if\ msgs\ (coord\ r) \neq None$
$then\ st' = st\ (\!|\ mru\text{-}vote := Some\ (three\text{-}phase\ r,\ val\ (the\ (msgs\ (coord\ r))))\ |\!)$
$else\ st' = st$

definition *send2* **where**
$send2\ r\ p\ q\ st \equiv (case\ mru\text{-}vote\ st\ of$
$Some\ (phs,\ v) \Rightarrow (if\ phs = three\text{-}phase\ r\ then\ Vote\ v\ else\ Null)$
$|\ \text{-} \Rightarrow Null$
$)$

— processes from which a vote was received
definition *votes-rcvd* **where**
$votes\text{-}rcvd\ (msgs :: process \rightharpoonup {'val}\ msg) \equiv$
$\{\ (q,\ v)\ .\ msgs\ q = Some\ (Vote\ v)\ \}$

definition *the-rcvd-vote* **where**
$the\text{-}rcvd\text{-}vote\ (msgs :: process \rightharpoonup {'val}\ msg) \equiv SOME\ v.\ v \in snd\ `\ votes\text{-}rcvd\ msgs$

definition *next2* **where**
$next2\ r\ p\ st\ msgs\ crd\ st' \equiv$
$if\ card\ (votes\text{-}rcvd\ msgs) > N\ div\ 2$
$then\ st' = st\ (\!|\ decide := Some\ (the\text{-}rcvd\text{-}vote\ msgs)\ |\!)$
$else\ st' = st$

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

definition $CT\text{-}sendMsg :: nat \Rightarrow process \Rightarrow process \Rightarrow {'val}\ pstate \Rightarrow {'val}\ msg$

where
CT-sendMsg (*r*::*nat*) ≡
if three-step r = 0 then send0 r
else if three-step r = 1 then send1 r
else send2 r

definition
CT-nextState :: *nat* ⇒ *process* ⇒ *'val pstate* ⇒ (*process* ⇀ *'val msg*)
⇒ *process* ⇒ *'val pstate* ⇒ *bool*
where
CT-nextState r ≡
if three-step r = 0 then next0 r
else if three-step r = 1 then next1 r
else next2 r

19.1 The *CT* Heard-Of machine

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

definition *CT-Alg* **where**
CT-Alg ≡
⦇ *CinitState* = *CT-initState*,
sendMsg = *CT-sendMsg*,
CnextState = *CT-nextState* ⦈

The CT algorithm relies on *waiting*: in each round, the coordinator waits until it hears from $\frac{N}{2}$ processes. This it reflected in the following per-round predicate.

definition
CT-commPerRd :: *nat* ⇒ *process HO* ⇒ *process coord* ⇒ *bool*
where
CT-commPerRd r HOs crds ≡
three-step r = 0 ⟶ *card* (*HOs* (*coord r*)) > *N div 2*

definition
CT-commGlobal **where**
CT-commGlobal HOs coords ≡
∃ *ph*::*nat*. ∃ *c*::*process*.
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 div 2*)

definition *CT-CHOMachine* **where**
CT-CHOMachine ≡
(| *CinitState* = *CT-initState*,
sendMsg = *CT-sendMsg*,
CnextState = *CT-nextState*,
CHOcommPerRd = *CT-commPerRd*,
CHOcommGlobal = *CT-commGlobal* |)

abbreviation
CT-M ≡ (*CT-CHOMachine*::(*process*, ′*val pstate*, ′*val msg*) *CHOMachine*)

end

19.2 Proofs

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

definition *CT-TS* ::
(*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process HO*)
⇒ (*round* ⇒ *process* ⇒ *process*)
⇒ *ct-TS-state TS*
where
CT-TS HOs SHOs crds = *CHO-to-TS CT-Alg HOs SHOs crds*

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

definition *CT-trans-step* **where**
CT-trans-step HOs SHOs crds nxt-f snd-f stp ≡ $\bigcup$ *r* μ.
{((*r*, *cfg*), (*Suc r*, *cfg*′))|*cfg cfg*′. *three-step r* = *stp* $\wedge$ ($\forall$ *p*.
μ *p* $\in$ *get-msgs* (*snd-f r*) *cfg* (*HOs r*) (*SHOs r*) *p*
$\wedge$ *nxt-f r p* (*cfg p*) (μ *p*) (*crds r p*) (*cfg*′ *p*)
)}

lemma *three-step-less-D*:
0 < *three-step r* ⟹ *three-step r* = *1* $\vee$ *three-step r* = *2*

⟨*proof*⟩

lemma *CT-trans*:
CSHO-trans-alt CT-sendMsg CT-nextState HOs SHOs crds =
CT-trans-step HOs SHOs crds next0 send0 0
∪ *CT-trans-step HOs SHOs crds next1 send1 1*
∪ *CT-trans-step HOs SHOs crds next2 send2 2*

⟨*proof*⟩

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

19.2.1 Refinement

definition *ct-ref-rel* :: (*three-step-mru-state* × *ct-TS-state*)*set* **where**
ct-ref-rel = {(*sa*, (*r*, *sc*)).
opt-mru-state.next-round sa = *r*
∧ *opt-mru-state.decisions sa* = *pstate.decide o sc*
∧ *opt-mru-state.mru-vote sa* = *pstate.mru-vote o sc*
∧ (*three-step r* = *Suc 0* ⟶ *three-step-mru-state.candidates sa* = {*commt* (*sc* (*coord r*))})
}

Now we need to use the fact that SHOs = HOs (i.e. the setting is non-Byzantine), and also the fact that the coordinator receives enough messages in each round

lemma *mru-vote-evolution0*:
$\forall p.\ next0\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow mru\text{-}vote\ o\ s' = mru\text{-}vote\ o\ s$
⟨*proof*⟩

lemma *mru-vote-evolution2*:
$\forall p.\ next2\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow mru\text{-}vote\ o\ s' = mru\text{-}vote\ o\ s$
⟨*proof*⟩

lemma *decide-evolution*:
$\forall p.\ next0\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow decide\ o\ s = decide\ o\ s'$
$\forall p.\ next1\ r\ p\ (s\ p)\ (msgs\ p)\ (crd\ p)\ (s'\ p) \Longrightarrow decide\ o\ s = decide\ o\ s'$
⟨*proof*⟩

lemma *msgs-mru-vote*:

assumes
μ (*coord r*) $\in$ *get-msgs* (*send0 r*) *cfg* (*HOs r*) (*HOs r*) (*coord r*) (**is** μ *?p* $\in$ -)
shows ((*msgs-to-lvs* (μ *?p*)) |' *HOs r ?p*) = (*mru-vote o cfg*) |' *HOs r ?p* ⟨*proof*⟩

context
fixes
HOs :: *nat* ⇒ *process* ⇒ *process set*
and *crds* :: *nat* ⇒ *process* ⇒*process*
assumes
per-rd: ∀ *r*. *CT-commPerRd r* (*HOs r*) (*crds r*)
begin

lemma *step0-ref*:
{*ct-ref-rel*}
(⋃ *r C*. *majorities.opt-mru-step0 r C*),
CT-trans-step HOs HOs crds next0 send0 0 {> *ct-ref-rel*}
⟨*proof*⟩

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*⟩

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*⟩

lemma *CT-Refines-ThreeStep-MRU*:
PO-refines ct-ref-rel majorities.ts-mru-TS (*CT-TS HOs HOs crds*)
⟨*proof*⟩

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*⟩

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.