

# Formalization of Timely Dataflow’s Progress Tracking Protocol

Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel

February 6, 2026

## Abstract

Large-scale stream processing systems often follow the dataflow paradigm, which enforces a program structure that exposes a high degree of parallelism. The Timely Dataflow distributed system supports expressive cyclic dataflows for which it offers low-latency data- and pipeline-parallel stream processing. To achieve high expressiveness and performance, Timely Dataflow uses an intricate distributed protocol for tracking the computations progress. We formalize this progress tracking protocol and verify its safety. Our formalization is described in detail in the forthcoming ITP’21 paper [3].

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Auxiliary Lemmas</b>	<b>3</b>
2.1	General . . . . .	3
2.2	Sums . . . . .	4
2.3	Partial Orders . . . . .	4
2.4	Multisets . . . . .	5
2.5	Signed Multisets . . . . .	5
2.5.1	Image of a Signed Multiset . . . . .	8
2.6	Streams . . . . .	11
2.7	Notation . . . . .	12
<b>3</b>	<b>Clocks Protocol</b>	<b>12</b>
<b>4</b>	<b>Exchange Protocol</b>	<b>33</b>
4.1	Specification . . . . .	33
4.2	Auxiliary Lemmas . . . . .	38
4.2.1	Transition lemmas . . . . .	41
4.2.2	Facts about <i>justified</i> ’ness . . . . .	45
4.2.3	Facts about <i>justified-with</i> ’ness . . . . .	55

4.3	Invariants . . . . .	68
4.3.1	InvRecordCount . . . . .	68
4.3.2	InvCapsNonneg and InvRecordsNonneg . . . . .	70
4.3.3	Resulting lemmas . . . . .	71
4.3.4	SafeRecordsMono . . . . .	71
4.3.5	InvJustifiedII and InvJustifiedGII . . . . .	74
4.3.6	InvTempJustified . . . . .	83
4.3.7	InvGlobNonposImpRecordsNonpos . . . . .	85
4.3.8	SafeGlobVacantUptoImpliesStickyNrec . . . . .	87
4.3.9	InvGlobNonposEqVacant . . . . .	88
4.3.10	InvInfoJustifiedWithII and InvInfoJustifiedWithGII . . . . .	89
4.3.11	SafeGlobMono and InvMsgInGlob . . . . .	93
<b>5</b>	<b>Antichains</b>	<b>102</b>
<b>6</b>	<b>Multigraphs with Partially Ordered Weights</b>	<b>105</b>
6.1	Paths . . . . .	106
6.2	Path Weights . . . . .	108
<b>7</b>	<b>Local Progress Propagation</b>	<b>113</b>
7.1	Specification . . . . .	113
7.2	Auxiliary . . . . .	116
7.3	Invariants . . . . .	118
7.3.1	Invariant: <i>inv-imps-work-sum</i> . . . . .	118
7.3.2	Invariant: <i>inv-imp-plus-work-nonneg</i> . . . . .	125
7.3.3	Invariant: <i>inv-implications-nonneg</i> . . . . .	126
7.4	Proof of Safety . . . . .	134
7.5	A Better (More Invariant) Safety . . . . .	140
7.6	Implied Frontier . . . . .	144
<b>8</b>	<b>Combined Progress Tracking Protocol</b>	<b>150</b>
8.1	Could-result-in Relation . . . . .	151
8.2	Specification . . . . .	152
8.2.1	Configuration . . . . .	152
8.2.2	Initial state and state transitions . . . . .	153
8.3	Auxiliary Lemmas . . . . .	156
8.3.1	Auxiliary Lemmas for CM Conversion . . . . .	156
8.4	Exchange is a Subsystem of Tracker . . . . .	163
8.5	Definitions . . . . .	164
8.6	Propagate is a Subsystem of Tracker . . . . .	165
8.6.1	CM Conditions . . . . .	165
8.6.2	Propagate Safety and InvGlobPointstampsEq . . . . .	165
8.6.3	Propagate is a Subsystem . . . . .	173
8.7	Safety Proofs . . . . .	175

# 1 Introduction

The dataflow programming model represents a program as a directed graph of interconnected operators that perform per-tuple data transformations. A message (an incoming datum) arrives at an input (a root of the dataflow) and flows along the graph’s edges into operators. Each operator takes the message, processes it, and emits any resulting derived messages.

In a dataflow system, all messages are associated with a timestamp, and operator instances need to know up-to-date (timestamp) *frontiers*—lower bounds on what timestamps may still appear as their inputs. When informed that all data for a range of timestamps has been delivered, an operator instance can complete the computation on input data for that range of timestamps, produce the resulting output, and retire those timestamps.

A *progress tracking mechanism* is a core component of the dataflow system. It receives information on outstanding timestamps from operator instances, exchanges this information with other system workers (cores, nodes) and disseminates up-to-date approximations of the frontiers to all operator instances. This AFP entry formally models and proves the safety of the progress tracking protocol of *Timely Dataflow* [1, 4], a dataflow programming model and a state-of-the-art streaming, data-parallel, distributed data processor. Specifically, we prove that the progress tracking protocol computes frontiers that always constitute safe lower bounds on what timestamps may still appear on the operator inputs. The formalization is described in detail in the forthcoming ITP’21 paper [3].

The ITP paper [3] closely follows this formalization’s structure. In particular, the paper’s presentation is split into four main sections each of which is present in the formalization (each in a separate theory file):

Algorithm/protocol	Section in this proof document	Section in [3]	Theory file
Abadi et al. [2]’s clocks protocol	Section 3	Section 3	Exchange_Abadi
Exchange protocol	Section 4	Section 4	Exchange
Local propagation algorithm	Section 7	Section 5	Propagate
Combined protocol	Section 8	Section 6	Combined

## 2 Auxiliary Lemmas

`unbundle` *multiset.lifting*

### 2.1 General

`lemma` *sum-list-hd-tl*:

**fixes**  $xs :: (- :: \text{group-add}) \text{ list}$   
**shows**  $xs \neq [] \implies \text{sum-list } (tl\ xs) = (-\ hd\ xs) + \text{sum-list } xs$   
**by**  $(\text{cases } xs) \text{ simp-all}$

## 2.2 Sums

**lemma** *Sum-eq-pick-changed-elem*:  
**assumes**  $\text{finite } M$   
**and**  $m \in M \ f\ m = g\ m + \Delta$   
**and**  $\bigwedge n. n \neq m \wedge n \in M \implies f\ n = g\ n$   
**shows**  $(\sum x \in M. f\ x) = (\sum x \in M. g\ x) + \Delta$   
**using**  $\text{assms}$   
**proof**  $(\text{induct } M \text{ arbitrary; } m \text{ rule: } \text{finite-induct})$   
**case**  $\text{empty}$   
**then show**  $?case \text{ by } \text{simp}$   
**next**  
**case**  $(\text{insert } x\ F)$   
**then show**  $?case$   
**proof**  $(\text{cases } x=m)$   
**case**  $\text{True}$   
**with**  $\text{insert have } \text{sum } f\ F = \text{sum } g\ F$   
**by**  $(\text{intro } \text{sum.cong}[OF\ \text{refl}]) \text{ force}$   
**with**  $\text{insert True show } ?thesis$   
**by**  $(\text{auto simp: } \text{add.commute } \text{add.left-commute})$   
**next**  
**case**  $\text{False}$   
**with**  $\text{insert show } ?thesis$   
**by**  $(\text{auto simp: } \text{add.assoc})$   
**qed**  
**qed**

**lemma** *sum-pos-ex-elem-pos*:  $(0 :: \text{int}) < (\sum m \in M. f\ m) \implies \exists m \in M. 0 < f\ m$   
**by**  $(\text{meson not-le sum-nonpos})$

**lemma** *sum-if-distrib-add*:  $\text{finite } A \implies b \in A \implies (\sum a \in A. \text{if } a=b \text{ then } X\ b + Y\ a \text{ else } X\ a) = (\sum a \in A. X\ a) + Y\ b$   
**by**  $(\text{simp add: } \text{Sum-eq-pick-changed-elem})$

## 2.3 Partial Orders

**lemma**  $(\text{in } \text{order}) \text{ order-finite-set-exists-foundation}$ :  
**fixes**  $t :: 'a$   
**assumes**  $\text{finite } M$   
**and**  $t \in M$   
**shows**  $\exists s \in M. s \leq t \wedge (\forall u \in M. \neg u < s)$   
**using**  $\text{assms}$   
**by**  $(\text{simp add: } \text{dual-order.strict-iff-order finite-has-minimal2})$

**lemma** *order-finite-set-obtain-foundation*:  
**fixes**  $t :: - :: \text{order}$

**assumes** *finite M*  
**and**  $t \in M$   
**obtains** *s* **where**  $s \in M$   $s \leq t \vee u \in M. \neg u < s$   
**using** *assms order-finite-set-exists-foundation* **by** *blast*

## 2.4 Multisets

**lemma** *finite-nonzero-count*: *finite*  $\{t. \text{count } M t > 0\}$   
**using** *count* **by** *auto*

**lemma** *finite-count[simp]*: *finite*  $\{t. \text{count } M t > i\}$   
**by** (*rule finite-subset[OF - finite-nonzero-count[of M]]*) (*auto simp only: set-mset-def*)

## 2.5 Signed Multisets

**lemma** *zcount-zmset-of-nonneg[simp]*:  $0 \leq \text{zcount} (\text{zmset-of } M) t$   
**by** *simp*

**lemma** *finite-zcount-pos[simp]*: *finite*  $\{t. \text{zcount } M t > 0\}$   
**apply** *transfer*  
**subgoal for** *M*  
**apply** (*rule finite-subset[OF - finite-Un[THEN iffD2, OF conjI[OF finite-nonzero-count finite-nonzero-count]], of - fst M snd M*)  
**apply** (*auto simp only: set-mset-def fst-conv snd-conv split: prod.splits*)  
**done**  
**done**

**lemma** *finite-zcount-neg[simp]*: *finite*  $\{t. \text{zcount } M t < 0\}$   
**apply** *transfer*  
**subgoal for** *M*  
**apply** (*rule finite-subset[OF - finite-Un[THEN iffD2, OF conjI[OF finite-nonzero-count finite-nonzero-count]], of - fst M snd M*)  
**apply** (*auto simp only: set-mset-def fst-conv snd-conv split: prod.splits*)  
**done**  
**done**

**lemma** *pos-zcount-in-zmset*:  $0 < \text{zcount } M x \implies x \in \#_z M$   
**by** (*simp add: zcount-inI*)

**lemma** *zmset-elem-nonneg*:  $x \in \#_z M \implies (\bigwedge x. x \in \#_z M \implies 0 \leq \text{zcount } M x)$   
 $\implies 0 < \text{zcount } M x$   
**by** (*simp add: order.order-iff-strict zcount-eq-zero-iff*)

**lemma** *zero-le-sum-single*:  $0 \leq \text{zcount} (\sum x \in M. \{\#f x\}_z) t$   
**by** (*induct M rule: infinite-finite-induct*) *auto*

**lemma** *mem-zmset-of[simp]*:  $x \in \#_z \text{zmset-of } M \longleftrightarrow x \in \# M$   
**by** (*simp add: set-zmset-def*)

**lemma** *mset-neg-minus*: *mset-neg* (*abs-zmultiset* (*Mp*, *Mn*)) = *Mn - Mp*

by (simp add: mset-neg.abs-eq)

**lemma** mset-pos-minus:  $mset\text{-pos} (abs\text{-zmultiset} (Mp, Mn)) = Mp - Mn$   
by (simp add: mset-pos.abs-eq)

**lemma** mset-neg-sum-set:  $(\bigwedge m. m \in M \implies mset\text{-neg} (f\ m) = \{\#\}) \implies mset\text{-neg} (\sum_{m \in M}. f\ m) = \{\#\}$   
by (induct M rule: infinite-finite-induct) auto

**lemma** mset-neg-empty-iff:  $mset\text{-neg} M = \{\#\} \longleftrightarrow (\forall t. 0 \leq zcount\ M\ t)$   
**apply** rule  
**subgoal**  
by (metis add.commute add.right-neutral mset-pos-as-neg zcount-zmset-of-nonneg zmset-of-empty)  
**subgoal**  
**apply** (induct rule: zmultiset.abs-induct)  
**subgoal for** y  
**apply** (induct y)  
**apply** (subst mset-neg-minus)  
**apply** transfer'  
**apply** (simp add: Diff-eq-empty-iff-mset mset-subset-eqI)  
**done**  
**done**  
**done**

**lemma** mset-neg-zcount-nonneg:  $mset\text{-neg} M = \{\#\} \implies 0 \leq zcount\ M\ t$   
by (subst (asm) mset-neg-empty-iff) simp

**lemma** in-zmset-conv-pos-neg-disj:  $x \in\#_z M \longleftrightarrow x \in\# mset\text{-pos} M \vee x \in\# mset\text{-neg} M$   
by (metis count-mset-pos in-diff-zcount mem-zmset-of mset-pos-neg-partition nat-code(2) not-in-iff zcount-ne-zero-iff)

**lemma** in-zmset-notin-mset-pos[simp]:  $x \in\#_z M \implies x \notin\# mset\text{-pos} M \implies x \in\# mset\text{-neg} M$   
by (auto simp: in-zmset-conv-pos-neg-disj)

**lemma** in-zmset-notin-mset-neg[simp]:  $x \in\#_z M \implies x \notin\# mset\text{-neg} M \implies x \in\# mset\text{-pos} M$   
by (auto simp: in-zmset-conv-pos-neg-disj)

**lemma** in-mset-pos-in-zmset:  $x \in\# mset\text{-pos} M \implies x \in\#_z M$   
by (auto intro: iffD2[OF in-zmset-conv-pos-neg-disj])

**lemma** in-mset-neg-in-zmset:  $x \in\# mset\text{-neg} M \implies x \in\#_z M$   
by (auto intro: iffD2[OF in-zmset-conv-pos-neg-disj])

**lemma** set-zmset-eq-set-mset-union:  $set\text{-zmset} M = set\text{-mset} (mset\text{-pos} M) \cup set\text{-mset} (mset\text{-neg} M)$

by (auto dest: in-mset-pos-in-zmset in-mset-neg-in-zmset)

**lemma** member-mset-pos-iff-zcount:  $x \in\# \text{ mset-pos } M \longleftrightarrow 0 < \text{zcount } M x$   
 using not-in-iff pos-zcount-in-zmset by force

**lemma** member-mset-neg-iff-zcount:  $x \in\# \text{ mset-neg } M \longleftrightarrow \text{zcount } M x < 0$   
 by (metis member-mset-pos-iff-zcount mset-pos-uminus neg-le-0-iff-le not-le zcount-uminus)

**lemma** mset-pos-mset-neg-disjoint[simp]:  $\text{set-mset } (\text{mset-pos } \Delta) \cap \text{set-mset } (\text{mset-neg } \Delta) = \{\}$   
 by (auto simp: member-mset-pos-iff-zcount member-mset-neg-iff-zcount)

**lemma** zcount-sum:  $\text{zcount } (\sum M \in MM. f M) t = (\sum M \in MM. \text{zcount } (f M) t)$   
 by (induct MM rule: infinite-finite-induct) auto

**lemma** zcount-filter-invariant:  $\text{zcount } \{\# t' \in\#_z M. t' = t \#\} t = \text{zcount } M t$   
 by auto

**lemma** in-filter-zmset-in-zmset[simp]:  $x \in\#_z \text{ filter-zmset } P M \implies x \in\#_z M$   
 by (metis count-filter-zmset zcount-ne-zero-iff)

**lemma** pos-filter-zmset-pos-zmset[simp]:  $0 < \text{zcount } (\text{filter-zmset } P M) x \implies 0 < \text{zcount } M x$   
 by (metis (full-types) count-filter-zmset less-irrefl)

**lemma** neg-filter-zmset-neg-zmset[simp]:  $0 > \text{zcount } (\text{filter-zmset } P M) x \implies 0 > \text{zcount } M x$   
 by (metis (full-types) count-filter-zmset less-irrefl)

**lift-definition** update-zmultiset ::  $'t \text{ zmultiset} \Rightarrow 't \Rightarrow \text{int} \Rightarrow 't \text{ zmultiset}$  is  
 $\lambda(A,B) T D. (\text{if } D > 0 \text{ then } (A + \text{replicate-mset } (\text{nat } D) T, B)$   
 $\text{else } (A, B + \text{replicate-mset } (\text{nat } (-D)) T))$   
 by (auto simp: equiv-zmset-def if-split)

**lemma** zcount-update-zmultiset:  $\text{zcount } (\text{update-zmultiset } M t n) t' = \text{zcount } M t' + (\text{if } t = t' \text{ then } n \text{ else } 0)$   
 by transfer auto

**lemma** (in order) order-zmset-exists-foundation:  
 fixes  $t :: 'a$   
 assumes  $0 < \text{zcount } M t$   
 shows  $\exists s. s \leq t \wedge 0 < \text{zcount } M s \wedge (\forall u. 0 < \text{zcount } M u \longrightarrow \neg u < s)$   
 using assms  
 proof –  
 let  $?M = \{t. 0 < \text{zcount } M t\}$   
 from assms have  $t \in ?M$   
 by simp  
 then have  $\exists s \in ?M. s \leq t \wedge (\forall u \in ?M. \neg u < s)$

by – (drule order-finite-set-exists-foundation[rotated 1], auto)  
then show ?thesis by auto  
qed

**lemma** (in order) order-zmset-exists-foundation':  
**fixes**  $t :: 'a$   
**assumes**  $0 < \text{zcount } M t$   
**shows**  $\exists s. s \leq t \wedge 0 < \text{zcount } M s \wedge (\forall u < s. \text{zcount } M u \leq 0)$   
**using** *assms order-zmset-exists-foundation*  
**by** (meson le-less-linear)

**lemma** (in order) order-zmset-exists-foundation-neg:  
**fixes**  $t :: 'a$   
**assumes**  $\text{zcount } M t < 0$   
**shows**  $\exists s. s \leq t \wedge \text{zcount } M s < 0 \wedge (\forall u. \text{zcount } M u < 0 \longrightarrow \neg u < s)$   
**using** *assms*  
**proof** –  
**let**  $?M = \{t. \text{zcount } M t < 0\}$   
**from** *assms* **have**  $t \in ?M$   
**by** *simp*  
**then have**  $\exists s \in ?M. s \leq t \wedge (\forall u \in ?M. \neg u < s)$   
**by** – (drule order-finite-set-exists-foundation[rotated 1], auto)  
**then show** ?thesis by auto  
qed

**lemma** (in order) order-zmset-exists-foundation-neg':  
**fixes**  $t :: 'a$   
**assumes**  $\text{zcount } M t < 0$   
**shows**  $\exists s. s \leq t \wedge \text{zcount } M s < 0 \wedge (\forall u < s. 0 \leq \text{zcount } M u)$   
**using** *assms order-zmset-exists-foundation-neg*  
**by** (meson le-less-linear)

**lemma** (in order) elem-order-zmset-exists-foundation:  
**fixes**  $x :: 'a$   
**assumes**  $x \in \#_z M$   
**shows**  $\exists s \in \#_z M. s \leq x \wedge (\forall u \in \#_z M. \neg u < s)$   
**by** (rule order-finite-set-exists-foundation[OF finite-set-zmset, OF assms(1)])

## 2.5.1 Image of a Signed Multiset

**lift-definition** *image-zmset* ::  $('a \Rightarrow 'b) \Rightarrow 'a \text{ zmset} \Rightarrow 'b \text{ zmset}$  **is**  
 $\lambda f (M, N). (\text{image-mset } f M, \text{image-mset } f N)$   
**by** (auto *simp: equiv-zmset-def simp flip: image-mset-union*)

**syntax** (ASCI)  
*-comprehension-zmset* ::  $'a \Rightarrow 'b \Rightarrow 'b \text{ zmset} \Rightarrow 'a \text{ zmset}$   $\langle\langle\{\#\!/.\ - : \#_z$   
 $\ - \#\}\rangle\rangle$   
**syntax**  
*-comprehension-zmset* ::  $'a \Rightarrow 'b \Rightarrow 'b \text{ zmset} \Rightarrow 'a \text{ zmset}$   $\langle\langle\{\#\!/.\ - \in \#_z$

-#})›)

**syntax-consts**

-comprehension-zmset  $\equiv$  image-zmset

**translations**

$\{\#e. x \in \#_z M \#\} \equiv \text{CONST image-zmset } (\lambda x. e) M$

**lemma** *image-zmset-empty[simp]*:  $\text{image-zmset } f \{\#\}_z = \{\#\}_z$

**by** *transfer (auto simp: equiv-zmset-def)*

**lemma** *image-zmset-single[simp]*:  $\text{image-zmset } f \{\#x\# \}_z = \{\#f x\# \}_z$

**by** *transfer (simp add: equiv-zmset-def)*

**lemma** *image-zmset-union[simp]*:  $\text{image-zmset } f (M + N) = \text{image-zmset } f M + \text{image-zmset } f N$

**by** *transfer (auto simp: equiv-zmset-def)*

**lemma** *image-zmset-Diff[simp]*:  $\text{image-zmset } f (A - B) = \text{image-zmset } f A - \text{image-zmset } f B$

**proof** -

**have**  $\text{image-zmset } f (A - B + B) = \text{image-zmset } f (A - B) + \text{image-zmset } f B$

**using** *image-zmset-union* **by** *blast*

**then show** *?thesis* **by** *simp*

**qed**

**lemma** *mset-neg-image-zmset*:  $\text{mset-neg } M = \{\#\} \implies \text{mset-neg } (\text{image-zmset } f M) = \{\#\}$

**unfolding** *multiset-eq-iff count-empty*

**by** *transfer (auto simp add: image-mset-subseteq-mono mset-subset-eqI mset-subset-eq-count)*

**lemma** *nonneg-zcount-image-zmset[simp]*:  $(\bigwedge t. 0 \leq \text{zcount } M t) \implies 0 \leq \text{zcount } (\text{image-zmset } f M) t$

**by** *(meson mset-neg-empty-iff mset-neg-image-zmset)*

**lemma** *image-zmset-add-zmset[simp]*:  $\text{image-zmset } f (\text{add-zmset } t M) = \text{add-zmset } (f t) (\text{image-zmset } f M)$

**by** *transfer (auto simp: equiv-zmset-def)*

**lemma** *pos-zcount-image-zmset[simp]*:  $(\bigwedge t. 0 \leq \text{zcount } M t) \implies 0 < \text{zcount } M t \implies 0 < \text{zcount } (\text{image-zmset } f M) (f t)$

**apply** *transfer*

**subgoal for**  $M t f$

**apply** *(induct M)*

**subgoal for**  $Mp Mn$

**apply** *simp*

**apply** *(metis count-diff count-image-mset-ge-count image-mset-Diff less-le-trans subseteq-mset-def zero-less-diff)*

**done**

**done**

**done**

**lemma** *set-zmset-transfer*[*transfer-rule*]:  
 (*rel-fun* (*pcr-zmultiset* (=)) (*rel-set* (=)))  
 ( $\lambda(Mp, Mn). \text{set-mset } Mp \cup \text{set-mset } Mn - \{x. \text{count } Mp \ x = \text{count } Mn \ x\}$ )  
*set-zmset*  
**by** (*auto simp: rel-fun-def pcr-zmultiset-def cr-zmultiset-def*  
*rel-set-eq multiset.rel-eq set-zmset-def zcount.abs-eq count-eq-zero-iff*[*symmetric*]  
*simp del: zcount-ne-zero-iff*)

**lemma** *zcount-image-zmset*:  
 $zcount (\text{image-zmset } f \ M) \ x = (\sum y \in f \ -' \ \{x\} \cap \text{set-zmset } M. \ zcount \ M \ y)$   
**apply** (*transfer fixing: f x*)  
**subgoal for** *M*  
**apply** (*cases M; clarify*)  
**subgoal for** *Mp Mn*  
**unfolding** *count-image-mset int-sum*  
**proof** -  
**have** ( $\sum x \in f \ -' \ \{x\} \cap \text{set-mset } Mp. \ \text{int} (\text{count } Mp \ x) =$   
 $(\sum x \in f \ -' \ \{x\} \cap (\text{set-mset } Mp \cup \text{set-mset } Mn). \ \text{int} (\text{count } Mp \ x))$ ) (**is** ?*S1*  
 = -)  
**by** (*subst sum.same-carrier*[**where**  $C = f \ -' \ \{x\} \cap (\text{set-mset } Mp \cup \text{set-mset } Mn)$ ])  
 (*auto simp: count-eq-zero-iff*)  
**moreover**  
**have** ( $\sum x \in f \ -' \ \{x\} \cap \text{set-mset } Mn. \ \text{int} (\text{count } Mn \ x) =$   
 $(\sum x \in f \ -' \ \{x\} \cap (\text{set-mset } Mp \cup \text{set-mset } Mn). \ \text{int} (\text{count } Mn \ x))$ ) (**is** ?*S2*  
 = -)  
**by** (*subst sum.same-carrier*[**where**  $C = f \ -' \ \{x\} \cap (\text{set-mset } Mp \cup \text{set-mset } Mn)$ ])  
 (*auto simp: count-eq-zero-iff*)  
**moreover**  
**have** ( $\sum x \in f \ -' \ \{x\} \cap (\text{set-mset } Mp \cup \text{set-mset } Mn - \{x. \text{count } Mp \ x =$   
 $\text{count } Mn \ x\}). \ \text{int} (\text{count } Mp \ x) - \text{int} (\text{count } Mn \ x)$   
 $= (\sum x \in f \ -' \ \{x\} \cap (\text{set-mset } Mp \cup \text{set-mset } Mn). \ \text{int} (\text{count } Mp \ x) - \text{int}$   
 $(\text{count } Mn \ x))$ )  
 (**is** ?*S* = -)  
**by** (*subst sum.same-carrier*[**where**  $C = f \ -' \ \{x\} \cap (\text{set-mset } Mp \cup \text{set-mset } Mn)$ ]) *auto*  
**ultimately show** ?*S1* - ?*S2* = ?*S*  
**by** (*auto simp: sum-subtractf*)  
**qed**  
**done**  
**done**

**lemma** *zmset-empty-image-zmset-empty*: ( $\bigwedge t. \ zcount \ M \ t = 0$ )  $\implies zcount (\text{image-zmset } f \ M) \ t = 0$   
**by** (*auto simp: zcount-image-zmset*)

**lemma** *in-image-zmset-in-zmset*:  $t \in \#_z \ \text{image-zmset } f \ M \implies \exists t. \ t \in \#_z \ M$

by (rule ccontr) simp

**lemma** zcount-image-zmset-zero:  $(\bigwedge m. m \in \#_z M \implies f m \neq x) \implies x \notin \#_z \text{image-zmset } f M$

**unfolding** set-zmset-def

by (simp add: zcount-image-zmset) (metis Int-emptyI sum.empty vimage-singleton-eq)

**lemma** image-zmset-pre:  $t \in \#_z \text{image-zmset } f M \implies \exists m. m \in \#_z M \wedge f m = t$

**proof** (rule ccontr)

**assume**  $t: t \in \#_z \text{image-zmset } f M$

**assume**  $\nexists m. m \in \#_z M \wedge f m = t$

**then have**  $m \in \#_z M \implies \neg f m = t$  **for**  $m$

by blast

**then have**  $\text{zcount } (\text{image-zmset } f M) t = 0$

by (meson t zcount-image-zmset-zero)

**with**  $t$  **show** False

by (meson zcount-ne-zero-iff)

qed

**lemma** pos-image-zmset-obtain-pre:

$(\bigwedge t. 0 \leq \text{zcount } M t) \implies 0 < \text{zcount } (\text{image-zmset } f M) t \implies \exists m. 0 < \text{zcount } M m \wedge f m = t$

**proof** –

**assume**  $\text{nonneg}: 0 \leq \text{zcount } M t$  **for**  $t$

**assume**  $0 < \text{zcount } (\text{image-zmset } f M) t$

**then have**  $t \in \#_z \text{image-zmset } f M$

by (simp add: pos-zcount-in-zmset)

**then obtain**  $x$  **where**  $x: x \in \#_z M \wedge f x = t$

by (auto dest: image-zmset-pre)

**with**  $\text{nonneg}$  **have**  $0 < \text{zcount } M x$

by (meson zmset-elem-nonneg)

**with**  $x$  **show** ?thesis

by auto

qed

## 2.6 Streams

**definition**  $\text{relates} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ stream} \Rightarrow \text{bool}$  **where**

$\text{relates } \varphi s = \varphi (\text{shd } s) (\text{shd } (\text{stl } s))$

**lemma**  $\text{relatesD}[dest]: \text{relates } P s \implies P (\text{shd } s) (\text{shd } (\text{stl } s))$

**unfolding**  $\text{relates-def}$  **by** simp

**lemma**  $\text{alw-relatesD}[dest]: \text{alw } (\text{relates } P) s \implies P (\text{shd } s) (\text{shd } (\text{stl } s))$

**by** auto

**lemma**  $\text{relatesI}[intro]: P (\text{shd } s) (\text{shd } (\text{stl } s)) \implies \text{relates } P s$

**by** (auto simp:  $\text{relates-def}$ )

**lemma** *alw-holds-smap-conv-comp*:  $alw (holds P) (smap f s) = alw (\lambda s. (P o f) (shd s)) s$   
**apply** (*rule iffI*)  
**apply** (*coinduction arbitrary: s*)  
**apply** *auto* []  
**apply** (*coinduction arbitrary: s*)  
**apply** *auto*  
**done**

**lemma** *alw-relates*:  $alw (relates P) s \longleftrightarrow P (shd s) (shd (stl s)) \wedge alw (relates P) (stl s)$   
**apply** (*rule iffI*)  
**apply** (*auto simp: relates-def dest: alwD*) []  
**apply** (*coinduction arbitrary: s*)  
**apply** (*auto simp: relates-def*)  
**done**

## 2.7 Notation

**no-notation** *AND* (**infix**  $\langle aand \rangle$  60)  
**no-notation** *OR* (**infix**  $\langle or \rangle$  60)  
**no-notation** *IMPL* (**infix**  $\langle imp \rangle$  60)

**notation** *AND* (**infixr**  $\langle aand \rangle$  70)  
**notation** *OR* (**infixr**  $\langle or \rangle$  65)  
**notation** *IMPL* (**infixr**  $\langle imp \rangle$  60)

**lifting-update** *multiset.lifting*  
**lifting-forget** *multiset.lifting*

## 3 Clocks Protocol

**type-synonym**  $'t \text{ count-vec} = 't \text{ multiset}$   
**type-synonym**  $'t \text{ delta-vec} = 't \text{ zmultiset}$

**definition** *vacant-upto* ::  $'t \text{ delta-vec} \Rightarrow 't :: \text{order} \Rightarrow \text{bool}$  **where**  
*vacant-upto*  $a t = (\forall s. s \leq t \longrightarrow zcount a s = 0)$

**abbreviation** *nonpos-upto* ::  $'t \text{ delta-vec} \Rightarrow 't :: \text{order} \Rightarrow \text{bool}$  **where**  
*nonpos-upto*  $a t \equiv \forall s. s \leq t \longrightarrow zcount a s \leq 0$

**definition** *supported-strong* ::  $'t \text{ delta-vec} \Rightarrow 't :: \text{order} \Rightarrow \text{bool}$  **where**  
*supported-strong*  $a t = (\exists s. s < t \wedge zcount a s < 0 \wedge \text{nonpos-upto } a s)$

**definition** *supported* ::  $'t \text{ delta-vec} \Rightarrow 't :: \text{order} \Rightarrow \text{bool}$  **where**  
*supported*  $a t = (\exists s. s < t \wedge zcount a s < 0)$

**definition** *upright* :: 't :: order delta-vec  $\Rightarrow$  bool **where**  
*upright* a = ( $\forall t. \text{zcount a } t > 0 \longrightarrow \text{supported a } t$ )

**lemma** *upright-alt*: *upright* a  $\longleftrightarrow$  ( $\forall t. \text{zcount a } t > 0 \longrightarrow \text{supported-strong a } t$ )  
**unfolding** *upright-def supported-def supported-strong-def*  
**by** (*rule iffI*) (*meson order.strict-trans2 order.strict-trans1 order-zmset-exists-foundation*)<sup>+</sup>

**definition** *beta-upright* :: 't :: order delta-vec  $\Rightarrow$  't :: order delta-vec  $\Rightarrow$  bool **where**  
*beta-upright* va vb = ( $\forall t. \text{zcount va } t > 0 \longrightarrow (\exists s. s < t \wedge (\text{zcount va } s < 0 \vee \text{zcount vb } s < 0))$ )

**lemma** *beta-upright-alt*:  
*beta-upright* va vb = ( $\forall t. \text{zcount va } t > 0 \longrightarrow (\exists s. s < t \wedge (\text{zcount va } s < 0 \vee \text{zcount vb } s < 0) \wedge \text{nonpos-upto va } s)$ )  
**unfolding** *beta-upright-def*  
**apply** (*rule iffI*)  
**apply** *clarsimp*  
**apply** (*drule order-zmset-exists-foundation*)  
**apply** (*metis le-less-linear less-le-trans order.strict-trans1*)  
**apply** *blast*  
**done**

**record** ('p, 't) *configuration* =  
*c-records* :: 't delta-vec  
*c-temp* :: 'p  $\Rightarrow$  't delta-vec  
*c-msg* :: 'p  $\Rightarrow$  'p  $\Rightarrow$  't delta-vec list  
*c-glob* :: 'p  $\Rightarrow$  't delta-vec

**type-synonym** ('p, 't) *computation* = ('p, 't) *configuration stream*

**definition** *init-config* :: ('p :: finite, 't :: order) *configuration*  $\Rightarrow$  bool **where**  
*init-config* c =  
( $(\forall p. \text{c-temp } c \text{ } p = \{\#\}_z) \wedge$   
 $(\forall p1 \text{ } p2. \text{c-msg } c \text{ } p1 \text{ } p2 = []) \wedge$   
 $(\forall p. \text{c-glob } c \text{ } p = \text{c-records } c) \wedge$   
 $(\forall t. 0 \leq \text{zcount } (\text{c-records } c) \text{ } t)$ )

**definition** *next-performop'* :: ('p, 't :: order) *configuration*  $\Rightarrow$  ('p, 't) *configuration*  
 $\Rightarrow$  'p  $\Rightarrow$  't count-vec  $\Rightarrow$  't count-vec  $\Rightarrow$  bool **where**  
*next-performop'* c0 c1 p c r =  
(*let*  $\Delta = \text{zmset-of } r - \text{zmset-of } c$  *in*  
 $(\forall t. \text{int } (\text{count } c \text{ } t) \leq \text{zcount } (\text{c-records } c0) \text{ } t)$   
 $\wedge \text{upright } \Delta$   
 $\wedge c1 = c0 \setminus (\text{c-records } := \text{c-records } c0 + \Delta,$   
 $\text{c-temp } := (\text{c-temp } c0)(p := \text{c-temp } c0 \text{ } p + \Delta))$ )

**abbreviation** *next-performop* **where**  
*next-performop* s  $\equiv$  ( $\exists p (c :: 't :: order \text{ count-vec}) (r :: 't \text{ count-vec}). \text{next-performop}'$ )

$(shd\ s)\ (shd\ (stl\ s))\ p\ c\ r)$

**definition** *next-sendupd'* **where**

$next\ sendupd'\ c0\ c1\ p\ tt =$   
 $(let\ \gamma = \{\#t \in \#_z\ c\text{-temp}\ c0\ p.\ t \in tt\#\} in$   
 $\gamma \neq 0$   
 $\wedge\ upright\ (c\text{-temp}\ c0\ p - \gamma)$   
 $\wedge\ c1 = c0\ (c\text{-msg} := (c\text{-msg}\ c0)(p := \lambda q.\ c\text{-msg}\ c0\ p\ q\ @\ [\gamma]),$   
 $\quad c\text{-temp} := (c\text{-temp}\ c0)(p := c\text{-temp}\ c0\ p - \gamma))$ )

**abbreviation** *next-sendupd* **where**

$next\ sendupd\ s \equiv (\exists p\ tt.\ next\ sendupd'\ (shd\ s)\ (shd\ (stl\ s))\ p\ tt)$

**definition** *next-recvupd'* **where**

$next\ recvupd'\ c0\ c1\ p\ q =$   
 $(c\text{-msg}\ c0\ p\ q \neq \square)$   
 $\wedge\ c1 = c0\ (c\text{-msg} := (c\text{-msg}\ c0)(p := (c\text{-msg}\ c0\ p)(q := tl\ (c\text{-msg}\ c0\ p\ q))),$   
 $\quad c\text{-glob} := (c\text{-glob}\ c0)(q := c\text{-glob}\ c0\ q + hd\ (c\text{-msg}\ c0\ p\ q)))$ )

**abbreviation** *next-recvupd* **where**

$next\ recvupd\ s \equiv (\exists p\ q.\ next\ recvupd'\ (shd\ s)\ (shd\ (stl\ s))\ p\ q)$

**definition** *next* **where**

$next\ s = (next\ performop\ s \vee next\ sendupd\ s \vee next\ recvupd\ s \vee (shd\ (stl\ s) = shd\ s))$

**definition** *spec* :: ('p :: finite, 't :: order) *computation*  $\Rightarrow$  *bool* **where**

$spec\ s = (holds\ init\ config\ s \wedge alw\ next\ s)$

**abbreviation** *GlobVacantUpto* **where**

$Glob\ VacantUpto\ c\ q\ t \equiv vacant\ upto\ (c\text{-glob}\ c\ q)\ t$

**abbreviation** *NrecVacantUpto* **where**

$Nrec\ VacantUpto\ c\ t \equiv vacant\ upto\ (c\text{-records}\ c)\ t$

**definition** *SafeGlobVacantUptoImpliesStickyNrec* :: ('p :: finite, 't :: order) *computation*  $\Rightarrow$  *bool* **where**

$Safe\ Glob\ VacantUptoImpliesStickyNrec\ s =$   
 $(let\ c = shd\ s\ in\ \forall t\ q.\ Glob\ VacantUpto\ c\ q\ t \longrightarrow alw\ (holds\ (\lambda c.\ Nrec\ VacantUpto\ c\ t))\ s)$

**definition** *SafeStickyNrecVacantUpto* :: ('p :: finite, 't :: order) *computation*  $\Rightarrow$  *bool* **where**

$Safe\ StickyNrec\ VacantUpto\ s =$   
 $(let\ c = shd\ s\ in\ \forall t.\ Nrec\ VacantUpto\ c\ t \longrightarrow alw\ (holds\ (\lambda c.\ Nrec\ VacantUpto\ c\ t))\ s)$

**definition** *InvGlobVacantUptoImpliesNrec* :: ('p :: finite, 't :: order) configuration  
 $\Rightarrow$  bool **where**  
*InvGlobVacantUptoImpliesNrec* c =  
 $(\forall t q. \text{vacant-upto } (c\text{-glob } c \ q) \ t \longrightarrow \text{vacant-upto } (c\text{-records } c) \ t)$

**definition** *InvTempUpright* **where**  
*InvTempUpright* c =  $(\forall p. \text{upright } (c\text{-temp } c \ p))$

**lemma** *init-InvTempUpright*: *init-config* c  $\Longrightarrow$  *InvTempUpright* c  
**by** (*simp* *add*: *InvTempUpright-def* *init-config-def* *upright-def*)

**lemma** *upright-obtain-support*:  
**assumes** *upright* a  
**and** *zcount* a t > 0  
**obtains** s **where** s < t *zcount* a s < 0 *nonpos-upto* a s  
**using** *assms* **unfolding** *upright-alt* *supported-strong-def*  
**apply** *atomize-elim*  
**using** *order.strict-implies-order* **apply** *blast*  
**done**

**lemma** *upright-vec-add*:  
**assumes** *upright* v1  
**and** *upright* v2  
**shows** *upright* (v1 + v2)  
**proof** –  
**let** ?v0 = v1 + v2  
**{** **fix** t  
**assume** *upr1*: *upright* v1  
**assume** *upr2*: *upright* v2  
**assume** *zcnt*: 0 < *zcount* ?v0 t  
**{** **fix** va vb :: 'a *zmultiset*  
**fix** t  
**assume** *upra*: *upright* va  
**assume** *uprb*: *upright* vb  
**assume** *zcnt*: 0 < *zcount* va t  
**from** *upra* *zcnt* **obtain** x **where** x: x < t *zcount* va x < 0 *nonpos-upto* va x  
**using** *upright-obtain-support* **by** *blast*  
**with** *uprb* **have** *supported-strong* (va+vb) t  
**apply** (*cases*  $\exists s. s \leq x \wedge 0 < \text{zcount } vb \ s$ )  
**apply** (*clarsimp* *simp*: *upright-alt* *supported-strong-def*)  
**apply** (*meson* *add-nonpos-neg* *less-imp-le* *order.strict-trans2* *order.trans*  
*add-nonpos-nonpos*)  
**apply** *simp*  
**apply** (*force* *simp*: *supported-strong-def* *intro!*: *exI*[*of* - x])  
**done**  
**}**  
**}**  
**with** *upr1* *upr2* *zcnt* **have** *supported-strong* ?v0 t **unfolding** *supported-strong-def*  
**apply** (*cases* 0 < *zcount* v1 t; *cases* 0 < *zcount* v2 t)

```

    apply auto [2]
    apply (subst (1 2) add.commute)
    apply auto
  done
}
with assms show ?thesis
  by (simp add: upright-alt)
qed

```

**lemma** *next-InvTempUpright*: holds *InvTempUpright*  $s \implies \text{next } s \implies \text{nxt}$  (holds *InvTempUpright*)  $s$

```

  unfolding next-def apply simp
  apply (elim disjE)
  subgoal
    unfolding InvTempUpright-def next-performop'-def
    by (auto simp: Let-def upright-vec-add)
  subgoal
    unfolding InvTempUpright-def next-sendupd'-def
    by (auto simp: Let-def upright-vec-add)
  subgoal
    unfolding InvTempUpright-def next-recvupd'-def
    by (auto simp: upright-vec-add)
  subgoal by simp
done

```

**lemma** *alw-InvTempUpright*: spec  $s \implies \text{alw}$  (holds *InvTempUpright*)  $s$

```

  apply (rule alw-invar)
  apply (simp add: spec-def init-InvTempUpright)
  apply (metis (no-types, lifting) alw-iff-sdrop next-InvTempUpright spec-def)
done

```

**definition** *IncomingInfo* **where**

*IncomingInfo*  $c k p q = (\text{sum-list } (\text{drop } k (c\text{-msg } c p q)) + c\text{-temp } c p)$

**definition** *InvIncomingInfoUpright* **where**

*InvIncomingInfoUpright*  $c = (\forall k p q. \text{upright } (\text{IncomingInfo } c k p q))$

**lemma** *upright-0*: upright  $0$

by (simp add: upright-def)

**lemma** *init-InvIncomingInfoUpright*: init-config  $c \implies \text{InvIncomingInfoUpright } c$

by (simp add: upright-0 upright-vec-add init-config-def InvIncomingInfoUpright-def IncomingInfo-def)

**lemma** *next-InvIncomingInfoUpright*: holds *InvIncomingInfoUpright*  $s \implies \text{next } s \implies \text{nxt}$  (holds *InvIncomingInfoUpright*)  $s$

```

  unfolding next-def
  apply simp
  apply (elim disjE)

```

**subgoal**  
**by** (*auto simp: add.assoc[symmetric] upright-vec-add next-performop'-def Let-def InvIncomingInfoUpright-def IncomingInfo-def*)  
**subgoal**  
**unfolding** *next-sendupd'-def Let-def InvIncomingInfoUpright-def Incoming-Info-def*  
**apply** *clarsimp*  
**subgoal for** *p tt k q*  
**apply** (*cases k ≤ length (c-msg (shd s) p q)*)  
**apply** *auto*  
**done**  
**done**  
**subgoal**  
**unfolding** *next-recvupd'-def Let-def InvIncomingInfoUpright-def IncomingInfo-def*  
**apply** (*clarsimp simp: drop-Suc[symmetric]*)  
**done**  
**subgoal**  
**by** *simp*  
**done**

**lemma** *alw-InvIncomingInfoUpright: spec s ⇒ alw (holds InvIncomingInfoUpright) s*

**by** (*metis (mono-tags, lifting) alw-iff-sdrop alw-invar holds.elims(2) holds.elims(3) init-InvIncomingInfoUpright next-InvIncomingInfoUpright spec-def*)

**definition** *GlobalIncomingInfo :: ('p :: finite, 't) configuration ⇒ nat ⇒ 'p ⇒ 'p ⇒ 't delta-vec where*

*GlobalIncomingInfo c k p q = (∑ p' ∈ UNIV. IncomingInfo c (if p' = p then k else 0) p' q)*

**abbreviation** *GlobalIncomingInfoAt where*

*GlobalIncomingInfoAt c q ≡ GlobalIncomingInfo c 0 q q*

**definition** *InvGlobalRecordCount where*

*InvGlobalRecordCount c = (∀ q. c-records c = GlobalIncomingInfoAt c q + c-glob c q)*

**lemma** *init-InvGlobalRecordCount: holds init-config s ⇒ holds InvGlobalRecordCount s*

**by** (*simp add: InvGlobalRecordCount-def init-config-def GlobalIncomingInfo-def IncomingInfo-def*)

**lemma** *if-eq-same: (if a = b then f b else f a) = f a*

**by** *auto*

**lemma** *next-InvGlobalRecordCount: holds InvGlobalRecordCount s ⇒ next s ⇒ next (holds InvGlobalRecordCount) s*

**unfolding** *InvGlobalRecordCount-def init-config-def GlobalIncomingInfo-def In-*

```

comingInfo-def next-def
  apply (elim disjE)
  subgoal
    apply (clarsimp simp: next-performop'-def Let-def)
    subgoal for p c q r
      apply (simp add: sum.distrib)
      apply (subst sum-if-distrib-add)
      apply (simp-all add: add.assoc)
    done
  done
  subgoal
    apply (clarsimp simp: next-sendupd'-def Let-def)
    subgoal for p tt q
      apply (simp add: if-distrib[of  $\lambda f. f$  -])
      apply (simp add: if-distrib[of sum-list])
      apply (subst sum-list-append)
      apply (simp add: sum.distrib)
      apply (subst sum-if-distrib-add)
      apply simp
      apply simp
      apply (subst diff-conv-add-uminus)
      apply (subst sum-if-distrib-add)
      apply (auto simp: sum-if-distrib-add)
    done
  done
  subgoal
    apply (clarsimp simp: next-recvupd'-def Let-def fun-upd-def)
    subgoal for p q q'
      apply (simp add: if-distrib[of  $\lambda f. f$  -])
      apply safe
      apply (simp add: if-distrib[of sum-list])
      apply (subst sum-list-hd-tl)
      apply simp
      apply (subst add commute)
      apply (simp add: sum.distrib)
      apply (subst sum-if-distrib-add)
      apply simp
      apply simp
      apply (simp add: add.assoc)
      apply (subst if-eq-same)
      apply simp
    done
  done
  subgoal
    by simp
  done

```

lemma *alw-InvGlobalRecordCount*:  $\text{spec } s \implies \text{alw (holds InvGlobalRecordCount)}$

s

**by** (*metis* (*no-types*, *lifting*) *alw-iff-sdrop* *alw-invar* *init-InvGlobalRecordCount* *next-InvGlobalRecordCount* *spec-def*)

**definition** *InvGlobalIncomingInfoUpright* **where**

*InvGlobalIncomingInfoUpright*  $c = (\forall k p q. \text{upright } (GlobalIncomingInfo\ c\ k\ p\ q))$

**lemma** *upright-sum-upright*: *finite*  $X \implies \forall x. \text{upright } (A\ x) \implies \text{upright } (\sum_{x \in X}. A\ x)$

**by** (*induct*  $X$  *rule*: *finite-induct*) (*auto simp*: *upright-0* *upright-vec-add*)

**lemma** *InvIncomingInfoUpright-imp-InvGlobalIncomingInfoUpright*: *holds* *InvIncomingInfoUpright*  $s \implies \text{holds } InvGlobalIncomingInfoUpright\ s$

**by** (*simp add*: *InvIncomingInfoUpright-def* *InvGlobalIncomingInfoUpright-def* *GlobalIncomingInfo-def* *upright-sum-upright*)

**lemma** *alw-InvGlobalIncomingInfoUpright*: *spec*  $s \implies \text{alw } (\text{holds } InvGlobalIncomingInfoUpright)\ s$

**using** *InvIncomingInfoUpright-imp-InvGlobalIncomingInfoUpright* *alw-InvIncomingInfoUpright* *alw-mono* **by** *blast*

**abbreviation** *nrec-pos* **where**

*nrec-pos*  $c \equiv \forall t. \text{zcount } (c\text{-records } c)\ t \geq 0$

**lemma** *init-nrec-pos*: *holds* *init-config*  $s \implies \text{holds } nrec\text{-pos } s$

**by** (*simp add*: *init-config-def*)

**lemma** *next-nrec-pos*: *holds* *nrec-pos*  $s \implies \text{next } s \implies \text{next } (\text{holds } nrec\text{-pos})\ s$

**unfolding** *next-def*

**apply** *simp*

**apply** *clarify*

**apply** (*elim* *disjE*)

**subgoal for**  $t$

**unfolding** *next-performop'-def* *Let-def*

**apply** *clarify*

**subgoal for**  $p\ c\ r$

**apply** (*simp add*: *add-diff-eq* *add commute* *add-increasing*)

**done**

**done**

**subgoal for**  $t$

**by** (*auto simp*: *next-sendupd'-def* *Let-def*)

**subgoal for**  $t$

**by** (*auto simp*: *next-recvupd'-def* *Let-def*)

**subgoal**

**by** *simp*

**done**

**lemma** *alw-nrec-pos: spec s  $\implies$  alw (holds nrec-pos) s*  
**by** (*metis (mono-tags, lifting) alw-iff-sdrop alw-invar init-nrec-pos next-nrec-pos spec-def*)

**lemma** *next-performop-vacant:*  
*vacant-upto (c-records (shd s)) t  $\implies$  next-performop s  $\implies$  vacant-upto (c-records (shd (stl s))) t*  
**unfolding** *next-performop'-def Let-def vacant-upto-def*  
**apply** *clarsimp*  
**subgoal for** *p c u r*  
**apply** (*clarsimp simp: upright-def supported-def*)  
**apply** (*metis (no-types, opaque-lifting) gr-implies-not-zero of-nat-le-0-iff order.strict-implies-order order-trans zero-less-iff-neq-zero*)  
**done**  
**done**

**lemma** *next-sendupd-vacant:*  
*vacant-upto (c-records (shd s)) t  $\implies$  next-sendupd s  $\implies$  vacant-upto (c-records (shd (stl s))) t*  
**by** (*auto simp add: next-sendupd'-def Let-def*)

**lemma** *next-recvupd-vacant:*  
*vacant-upto (c-records (shd s)) t  $\implies$  next-recvupd s  $\implies$  vacant-upto (c-records (shd (stl s))) t*  
**by** (*auto simp add: next-recvupd'-def Let-def*)

**lemma** *spec-imp-SafeStickyNrecVacantUpto-aux: alw next s  $\implies$  alw SafeStickyNrecVacantUpto s*  
**apply** (*coinduction arbitrary: s*)  
**subgoal for** *s*  
**unfolding** *spec-def next-def SafeStickyNrecVacantUpto-def Let-def*  
**apply** (*rule exI[of - s]*)  
**apply** *safe*  
**subgoal for** *t*  
**apply** (*coinduction arbitrary: s rule: alw.coinduct*)  
**apply** *clarsimp*  
**apply** (*rule conjI*)  
**apply** *blast*  
**apply** (*erule alw.cases*)  
**apply** *clarsimp*  
**apply** (*elim disjE*)  
**apply** (*simp-all add: next-performop-vacant next-sendupd-vacant next-recvupd-vacant*)  
**done**  
**by** *blast*  
**done**

**lemma** *spec-imp-SafeStickyNrecVacantUpto: spec s  $\implies$  alw SafeStickyNrecVacantUpto s*  
**unfolding** *spec-def*

by (blast intro: spec-imp-SafeStickyNrec VacantUpto-aux)

**lemma** *invs-imp-InvGlob VacantUptoImpliesNrec:*

**assumes** *holds InvGlobalIncomingInfoUpright s*

**assumes** *holds InvGlobalRecordCount s*

**assumes** *holds nrec-pos s*

**shows** *holds InvGlob VacantUptoImpliesNrec s*

**using** *assms unfolding InvGlob VacantUptoImpliesNrec-def*

**apply** *simp*

**apply** *clarify*

**apply** *(rule ccontr)*

**apply** *(simp add: vacant-upto-def)*

**apply** *clarify*

**subgoal for** *t q u*

**proof** –

**assume** *globvut:  $\forall sa \leq t. zcount (c-glob (shd s) q) sa = 0$*

**assume** *uleqt:  $u \leq t$*

**assume**  *$u \in \#_z c\text{-records} (shd s)$*

**with** *assms(3) have*  $0 < zcount (c\text{-records} (shd s)) u$

**by** *(simp add: order.not-eq-order-implies-strict)*

**with** *assms(2) globvut uleqt have*  $*$ :  $0 < zcount (GlobalIncomingInfoAt (shd s) q) u$

**unfolding** *InvGlobalRecordCount-def*

**by** *(auto dest: spec[of - q])*

**from** *assms(1)[unfolded InvGlobalIncomingInfoUpright-def] have* *upright (GlobalIncomingInfoAt (shd s) q)*

**by** *simp*

**with**  $*$  **obtain** *v where*  $**$ :  $v \leq u$   $zcount (GlobalIncomingInfoAt (shd s) q) v < 0$

**by** *(meson order.strict-iff-order upright-def supported-def)*

**with** *assms(2) have*  $zcount (c\text{-records} (shd s)) v < 0$

**by** *(metis (no-types, opaque-lifting) InvGlobalRecordCount-def add.right-neutral order.trans globvut holds.elims(2) uleqt zcount-union)*

**with** *assms(3) show* *False*

**using** *atLeastatMost-empty by auto*

**qed**

**done**

**lemma** *spec-imp-inv1: spec s  $\implies$  alw (holds InvGlob VacantUptoImpliesNrec) s*

**by** *(metis (mono-tags, lifting) alw-iff-sdrop invs-imp-InvGlob VacantUptoImpliesNrec alw-InvGlobalIncomingInfoUpright alw-InvGlobalRecordCount alw-nrec-pos)*

**lemma** *safe2-inv1-imp-safe: SafeStickyNrec VacantUpto s  $\implies$  holds InvGlob VacantUptoImpliesNrec s  $\implies$  SafeGlob VacantUptoImpliesStickyNrec s*

**by** *(simp add: InvGlob VacantUptoImpliesNrec-def SafeStickyNrec VacantUpto-def SafeGlob VacantUptoImpliesStickyNrec-def)*

**lemma** *spec-imp-safe: spec s  $\implies$  alw SafeGlob VacantUptoImpliesStickyNrec s*

**by** *(meson alw-iff-sdrop safe2-inv1-imp-safe spec-imp-SafeStickyNrec VacantUpto)*

*spec-imp-inv1*)

**lemma** *beta-upright-0*: *beta-upright 0 vb*  
**unfolding** *beta-upright-def*  
**by** *auto*

**definition** *PositiveImplies* **where**  
*PositiveImplies v w = ( $\forall t. zcount v t > 0 \longrightarrow zcount w t > 0$ )*

**lemma** *betaupright-PositiveImplies*: *upright (va + vb)  $\implies$  PositiveImplies va (va + vb)  $\implies$  beta-upright va vb*  
**unfolding** *beta-upright-def PositiveImplies-def*  
**apply** *clarify*  
**subgoal for** *t*  
**apply** (*erule upright-obtain-support*[of - *t*])  
**apply** *simp*  
**subgoal for** *s*  
**apply** (*rule exI*[of - *s*])  
**apply** *simp*  
**apply** (*simp add: add-less-zeroD*)  
**done**  
**done**  
**done**

**lemma** *betaupright-obtain-support*:  
**assumes** *beta-upright va vb*  
*zcount va t > 0*  
**obtains** *s* **where** *s < t zcount va s < 0  $\vee$  zcount vb s < 0 nonpos-upto va s*  
**using** *assms* **by** (*auto simp: beta-upright-alt*)

**lemma** *betaupright-upright-vut*:  
**assumes** *beta-upright va vb*  
**and** *upright vb*  
**and** *vacant-upto (va + vb) t*  
**shows** *vacant-upto va t*  
**proof** –  
**{** **fix** *s*  
**assume** *s: s  $\leq$  t zcount va s > 0*  
**with** *assms* **obtain** *x* **where** *x: x < s zcount va x < 0  $\vee$  zcount vb x < 0*  
*nonpos-upto va x*  
**using** *betaupright-obtain-support* **by** *blast*  
**then** **have** *False*  
**proof** (*cases zcount va x < 0*)  
**case** *True*  
**with** *assms(2,3) s x(1,3)* **show** *?thesis*

```

    unfolding vacant-upto-def
    apply clarsimp
    apply (erule upright-obtain-support[of vb x])
    apply (metis add-less-same-cancel2 order.trans order.strict-implies-order)
    apply (metis add-less-same-cancel1 add-neg-neg order.order-iff-strict or-
der.trans less-irrefl)
  done
next
case False
with assms s x have  $x \leq t$  zcount va  $x > 0$ 
  apply -
  apply simp
  apply (metis (no-types, opaque-lifting) add.left-neutral order.order-iff-strict
order.trans vacant-upto-def zcount-union)
  done
  with assms(2,3) s x show ?thesis
  by force
qed
}
note r = this
from assms(2,3) show ?thesis
  unfolding vacant-upto-def
  apply clarsimp
  apply (metis (no-types, opaque-lifting) r add-cancel-right-left order.order-iff-strict
order.trans le-less-linear less-add-same-cancel2 upright-obtain-support)
  done
qed

```

**lemma** *beta-upright-add*:

```

  assumes upright vb
  and upright vc
  and beta-upright va vb
  shows beta-upright va (vb + vc)
proof -
  { fix t
    assume  $0 < zcount va t$ 
    assume assm:  $\neg(\exists s < t. (zcount va s < 0 \vee zcount vb s + zcount vc s < 0) \wedge$ 
 $\neg(\exists u \leq s. zcount va u > 0))$ 
    from  $\langle 0 < zcount va t \rangle$  assms(3) obtain x where x:  $x < t \wedge (zcount va x <$ 
 $0 \vee zcount vb x < 0) \wedge nonpos-upto va x$ 
    using betaupright-obtain-support by blast
    then have  $\neg zcount va x < 0$ 
    using assm by force
    with x have  $zcount vb x < 0$ 
    by blast
    from assm x have  $\neg zcount vb x + zcount vc x < 0$ 
    using not-le by blast
    with  $\langle zcount vb x < 0 \rangle$  have  $zcount vc x > 0$ 
    by clarsimp
  }

```

```

with assms(2) obtain y where y:  $y < x \wedge \text{zcount } vc \ y < 0 \wedge \text{nonpos-upto } vc$ 
y
  using upright-obtain-support by blast
with x have  $y < t$ 
  using order.strict-trans by blast
from assm x y have  $\neg \text{zcount } vb \ y + \text{zcount } vc \ y < 0$ 
  by (metis order.strict-implies-order order.strict-trans1 not-less)
with y have  $\text{zcount } vb \ y > 0$ 
  by linarith
with assms(1) obtain z where z:  $z < y \wedge \text{zcount } vb \ z < 0$ 
  by (auto simp: upright-def supported-def)
with  $\langle y < t \rangle$  have  $z < t$ 
  using order.strict-trans by blast
with x y z have  $\neg \text{zcount } vb \ z + \text{zcount } vc \ z < 0$ 
  by (metis assm less-imp-le not-less order.strict-trans order.strict-trans1)
with z have  $\text{zcount } vc \ z > 0$ 
  by linarith
with y z have False
  using order.strict-implies-order not-less by blast
}
then show ?thesis
  using beta-upright-def zcount-union by fastforce
qed

```

**definition** *InfoAt* **where**

*InfoAt* *c k p q* = (if  $0 \leq k \wedge k < \text{length } (c\text{-msg } c \ p \ q)$  then  $(c\text{-msg } c \ p \ q) ! k$  else 0)

**definition** *InvInfoAtBetaUpright* **where**

*InvInfoAtBetaUpright* *c* = ( $\forall k \ p \ q. \text{beta-upright } (InfoAt \ c \ k \ p \ q) \ (\text{IncomingInfo } c \ (k+1) \ p \ q)$ )

**lemma** *init-InvInfoAtBetaUpright*: *init-config* *c*  $\implies$  *InvInfoAtBetaUpright* *c*

**unfolding** *init-config-def* *InvInfoAtBetaUpright-def* *beta-upright-def* *Incoming-Info-def* *InfoAt-def*  
**by** *simp*

**lemma** *next-inv*[*consumes 1*, *case-names next-performop next-sendupd next-recvupd stutter*]:

**assumes** *next* *s*  
**and** *next-performop* *s*  $\implies$  *P*  
**and** *next-sendupd* *s*  $\implies$  *P*  
**and** *next-recvupd* *s*  $\implies$  *P*  
**and** *shd* (*stl* *s*) = *shd* *s*  $\implies$  *P*  
**shows** *P*

**using** *assms* **unfolding** *next-def* **by** *blast*

**lemma** *next-InvInfoAtBetaUpright*:

```

assumes  $a1$ : next s
  and    $a2$ : InvInfoAtBetaUpright (shd s)
  and    $a3$ : InvIncomingInfoUpright (shd s)
  and    $a4$ : InvTempUpright (shd s)
shows  InvInfoAtBetaUpright (shd (stl s))
using  assms
proof (cases rule: next-inv)
  case next-performop
  then show ?thesis
    unfolding next-performop'-def Let-def InvInfoAtBetaUpright-def
    apply clarify
    subgoal for  $p\ c\ r\ k'\ p'\ q'$ 
    proof (cases p=p')
      let  $?\Delta = \text{zmset-of } r - \text{zmset-of } c$ 
      assume upright- $\Delta$ : upright  $?\Delta$ 
      assume conf: shd (stl s) = shd s[(c-records := c-records (shd s) + (zmset-of
r - zmset-of c),
        c-temp := (c-temp (shd s))(p := c-temp (shd s) p + (zmset-of r - zmset-of
c)))]
      case True
      then have iid: IncomingInfo (shd (stl s)) (k'+1) p' q' = IncomingInfo (shd
s) (k'+1) p' q' +  $?\Delta$ 
        by (simp add: IncomingInfo-def conf)
      from  $a2$  have bu: beta-upright (InfoAt (shd s) k' p' q') (IncomingInfo (shd
s) (k'+1) p' q')
        using InvInfoAtBetaUpright-def by fastforce
      show ?thesis
      unfolding iid
      apply (rule beta-upright-add)
      apply (meson InvIncomingInfoUpright-def a3)
      apply (rule upright- $\Delta$ )
      using bu unfolding conf InfoAt-def
      apply auto
      done
    next
      let  $?\Delta = \text{zmset-of } r - \text{zmset-of } c$ 
      assume conf: shd (stl s) = shd s[(c-records := c-records (shd s) + (zmset-of
r - zmset-of c),
        c-temp := (c-temp (shd s))(p := c-temp (shd s) p + (zmset-of r - zmset-of
c)))]
      from  $a2$  have bu: beta-upright (InfoAt (shd s) k p q) (IncomingInfo (shd s)
(k + 1) p q) for  $k\ p\ q$ 
        using InvInfoAtBetaUpright-def by fastforce
      case False
      then have ii: IncomingInfo (shd (stl s)) (k'+1) p' q' = IncomingInfo (shd s)
(k'+1) p' q'
        unfolding IncomingInfo-def by (simp add: conf)
      with bu[of k' p' q'] show ?thesis unfolding conf InfoAt-def
by auto

```

```

qed
done
next
case next-sendupd
then show ?thesis
  unfolding next-sendupd'-def Let-def InvInfoAtBetaUpright-def
  apply clarify
  subgoal for p tt k' p' q'
  proof (cases p=p')
    let ?γ = {#t ∈ #z c-temp (shd s) p. t ∈ tt#}
    assume conf: shd (stl s) = (shd s)(c-msg := (c-msg (shd s))(p := λq. c-msg
(shd s) p q @ [?γ]),
      c-temp := (c-temp (shd s))(p := c-temp (shd s) p - ?γ))
    from a2 have buia: beta-upright (InfoAt (shd s) k' p' q') (IncomingInfo (shd
s) (k'+1) p' q')
      using InvInfoAtBetaUpright-def by force
    from a4 have tu: upright (c-temp (shd s) p)
      by (simp add: InvTempUpright-def)
    case True
    then show ?thesis
    proof (cases k' rule: linorder-cases[where y = length (c-msg (shd s) p' q')])
      case greater
      then have InfoAt (shd (stl s)) k' p' q' = 0
        by (auto simp: conf InfoAt-def)
      then show ?thesis
        by (simp add: beta-upright-0)
    next
    case equal
    with True conf have InfoAt (shd (stl s)) k' p' q' = ?γ
      by (simp add: InfoAt-def)
    then have pi: PositiveImplies (InfoAt (shd (stl s)) k' p' q') (c-temp (shd s)
p)
      by (simp add: PositiveImplies-def)
    from conf have c-temp (shd s) p = c-temp (shd (stl s)) p + ?γ
      by simp
    with equal True conf tu pi have butemp: beta-upright (InfoAt (shd (stl s))
k' p' q') (c-temp (shd (stl s)) p)
      apply -
      apply (rule betaupright-PositiveImplies)
      apply (auto simp add: InfoAt-def)
    done
    with True equal conf have IncomingInfo (shd (stl s)) (k'+1) p' q' = c-temp
(shd (stl s)) p
      by (simp add: IncomingInfo-def)
    with butemp show ?thesis
      by simp
  next
  case less
  with conf have unch-ia: InfoAt (shd (stl s)) k' p' q' = InfoAt (shd s) k' p'

```

$q'$   
**by** (*auto simp: nth-append InfoAt-def*)  
**from** *conf less* **have** *IncomingInfo* (*shd* (*stl s*)) ( $k'+1$ )  $p' q' = \text{IncomingInfo}$   
(*shd s*) ( $k'+1$ )  $p' q'$   
**by** (*auto simp: IncomingInfo-def*)  
**with** *buia unch-ia* **show** *?thesis* **by** *simp*  
**qed**  
**next**  
**let**  $? \gamma = \{ \#t \in \#_z \text{ c-temp } (\text{shd } s) \text{ p. } t \in \text{tt}\# \}$   
**assume** *conf*:  $\text{shd } (\text{stl } s) = (\text{shd } s) \langle \text{c-msg} := (\text{c-msg } (\text{shd } s)) (p := \lambda q. \text{c-msg}$   
(*shd s*)  $p \ q \ @ \ [? \gamma])$ ,  
 $\text{c-temp} := (\text{c-temp } (\text{shd } s)) (p := \text{c-temp } (\text{shd } s) \text{ p} - ? \gamma) \rangle$   
**from** *a2* **have** *buia*: *beta-upright* (*InfoAt* (*shd s*)  $k' p' q'$ ) (*IncomingInfo* (*shd*  
*s*) ( $k'+1$ )  $p' q'$ )  
**using** *InvInfoAtBetaUpright-def* **by** *force*  
**case** *False*  
**with** *conf* **have** *unchia*: *InfoAt* (*shd* (*stl s*))  $k' p' q' = \text{InfoAt}$  (*shd s*)  $k' p' q'$   
**by** (*simp add: InfoAt-def*)  
**from** *False conf* **have** *unchii*: *IncomingInfo* (*shd* (*stl s*)) ( $k'+1$ )  $p' q' =$   
*IncomingInfo* (*shd s*) ( $k'+1$ )  $p' q'$   
**by** (*simp add: IncomingInfo-def*)  
**from** *unchia unchii buia* **show** *?thesis*  
**by** *simp*  
**qed**  
**done**  
**next**  
**case** *next-recvupd*  
**then** **show** *?thesis*  
**unfolding** *next-recvupd'-def Let-def InvInfoAtBetaUpright-def*  
**apply** *clarify*  
**subgoal** **for**  $p \ q \ k' \ p' \ q'$   
**proof** (*cases*  $p = p' \wedge q = q'$ )  
**assume** *conf*:  $\text{shd } (\text{stl } s) = (\text{shd } s) \langle \text{c-msg} := (\text{c-msg } (\text{shd } s)) (p := (\text{c-msg}$   
(*shd s*)  $p) (q := \text{tl } (\text{c-msg } (\text{shd } s) \text{ p } q)))$ ,  
 $\text{c-glob} := (\text{c-glob } (\text{shd } s)) (q := \text{c-glob } (\text{shd } s) \text{ q} + \text{hd } (\text{c-msg } (\text{shd } s) \text{ p}$   
 $q)) \rangle$   
**case** *True*  
**with** *conf* **have** *iisuc*: *IncomingInfo* (*shd* (*stl s*)) ( $k'+1$ )  $p' q' = \text{IncomingInfo}$   
(*shd s*) ( $k'+2$ )  $p' q'$   
**by** (*simp add: drop-Suc IncomingInfo-def*)  
**with** *True conf* **have** *iasuc*: *InfoAt* (*shd* (*stl s*))  $k' p' q' = \text{InfoAt}$  (*shd s*)  
( $k'+1$ )  $p' q'$   
**by** (*simp add: less-diff-conv nth-tl InfoAt-def*)  
**from** *a2* **have** *beta-upright* (*InfoAt* (*shd s*) ( $k'+1$ )  $p' q'$ ) (*IncomingInfo* (*shd*  
*s*) ( $k'+2$ )  $p' q'$ )  
**using** *InvInfoAtBetaUpright-def* **by** *fastforce*  
**with** *iisuc iasuc* **show** *?thesis*  
**by** *simp*  
**next**

**assume** *conf*:  $\text{shd } (\text{stl } s) = (\text{shd } s) \parallel (\text{c-msg } := (\text{c-msg } (\text{shd } s))(p := (\text{c-msg } (\text{shd } s) p)(q := \text{tl } (\text{c-msg } (\text{shd } s) p q))),$   
 $\text{c-glob } := (\text{c-glob } (\text{shd } s))(q := \text{c-glob } (\text{shd } s) q + \text{hd } (\text{c-msg } (\text{shd } s) p q)) \parallel$   
**from** *a2* **have** *buia*:  $\text{beta-upright } (\text{InfoAt } (\text{shd } s) k' p' q') (\text{IncomingInfo } (\text{shd } s) (k'+1) p' q')$   
**by** (*simp add: InvInfoAtBetaUpright-def*)  
**case** *False*  
**with** *conf* **have** *unchii*:  $\text{IncomingInfo } (\text{shd } (\text{stl } s)) (k'+1) p' q' = \text{IncomingInfo } (\text{shd } s) (k'+1) p' q'$   
**by** (*auto simp: IncomingInfo-def*)  
**from** *False conf* **have** *unchia*:  $\text{InfoAt } (\text{shd } (\text{stl } s)) k' p' q' = \text{InfoAt } (\text{shd } s) k' p' q'$   
**by** (*auto simp: InfoAt-def*)  
**from** *unchii unchia buia* **show** *?thesis*  
**by** *simp*  
**qed**  
**done**  
**qed** *simp*

**lemma** *alw-InvInfoAtBetaUpright-aux*:  $\text{alw } (\text{holds } \text{InvTempUpright}) s \implies \text{alw } (\text{holds } \text{InvIncomingInfoUpright}) s \implies \text{holds } \text{InvInfoAtBetaUpright } s \implies \text{alw } \text{next } s \implies \text{alw } (\text{holds } \text{InvInfoAtBetaUpright}) s$   
**by** (*coinduction arbitrary: s rule: alw.coinduct*) (*auto intro!: next-InvInfoAtBetaUpright*)

**lemma** *alw-InvInfoAtBetaUpright*:  $\text{spec } s \implies \text{alw } (\text{holds } \text{InvInfoAtBetaUpright}) s$   
**by** (*simp add: alw-InvTempUpright alw-InvIncomingInfoUpright alw-InvInfoAtBetaUpright-aux init-InvInfoAtBetaUpright spec-def*)

**definition** *InvGlobalInfoAtBetaUpright* **where**

$\text{InvGlobalInfoAtBetaUpright } c = (\forall k p q. \text{beta-upright } (\text{InfoAt } c k p q) (\text{GlobalIncomingInfo } c (k+1) p q))$

**lemma** *finite-induct-select* [*consumes 1, case-names empty select*]:

**assumes** *finite S*  
**and** *empty: P {}*  
**and** *select:  $\bigwedge T. \text{finite } T \implies T \subset S \implies P T \implies \exists s \in S - T. P (\text{insert } s T)$*   
**shows**  $P S$   
**proof** –  
**from** *assms(1)* **have**  $P S \wedge \text{finite } S$   
**by** (*induct S rule: finite-induct-select*) (*auto intro: empty select*)  
**then show** *?thesis* **by** *blast*  
**qed**

**lemma** *predicate-sum-decompose*:

**fixes**  $f :: 'a \Rightarrow ('b :: \text{ab-group-add})$   
**assumes** *finite X*  
**and**  $x \in X$   
**and**  $A (f x)$

```

and     $\forall Z. B (\text{sum } f Z)$ 
and     $\bigwedge x Z. A (f x) \implies B (\text{sum } f Z) \implies A (f x + \text{sum } f Z)$ 
and     $\bigwedge x Z. B (f x) \implies A (\text{sum } f Z) \implies A (f x + \text{sum } f Z)$ 
shows  $A (\sum_{x \in X}. f x)$ 
using  $\text{assms}(1,2,3)$ 
apply ( $\text{induct } X \text{ rule: finite-induct-select}$ )
apply  $\text{simp}$ 
apply ( $\text{simp only: sum.insert-remove}$ )
subgoal for  $T$ 
apply ( $\text{cases } x \in T; \text{simp add: assms}(3)$ )
apply ( $\text{drule psubset-imp-ex-mem}$ )
apply  $\text{clarsimp}$ 
subgoal for  $z$ 
apply ( $\text{rule bexI}[of - z]$ )
apply ( $\text{rule assms}(6)[of z T]$ )
apply ( $\text{rule assms}(4)[\text{THEN spec, of } \{z\}, \text{simplified}]$ )
apply  $\text{simp}$ 
apply  $\text{simp}$ 
done
apply  $\text{clarsimp}$ 
apply ( $\text{drule bspec}[of - - x]$ )
apply  $\text{safe}$ 
apply ( $\text{rule assms}(2)$ )
using  $\text{assms}(4,5)$  apply  $\text{blast}$ 
done
done

```

**lemma**  $\text{invs-imp-InvGlobalInfoAtBetaUpright}$ :

```

assumes  $\text{holds InvInfoAtBetaUpright } s$ 
and     $\text{holds InvGlobalIncomingInfoUpright } s$ 
and     $\text{holds InvIncomingInfoUpright } s$ 
shows   $\text{holds InvGlobalInfoAtBetaUpright } s$ 
proof –
have  $\text{uui: } \forall k p q. \text{upright } (\text{IncomingInfo } (\text{shd } s) k p q)$ 
by ( $\text{rule assms}(3)[\text{unfolded InvIncomingInfoUpright-def, simplified}]$ )
have  $\text{ugii: } \forall k p q. \text{upright } (\text{GlobalIncomingInfo } (\text{shd } s) k p q)$ 
by ( $\text{rule assms}(2)[\text{unfolded InvGlobalIncomingInfoUpright-def, simplified}]$ )
have  $\text{buia: } \forall k p q. \text{beta-upright } (\text{InfoAt } (\text{shd } s) k p q) (\text{IncomingInfo } (\text{shd } s) (\text{Suc } k) p q)$ 
by ( $\text{rule assms}(1)[\text{unfolded InvInfoAtBetaUpright-def, simplified}]$ )
from  $\text{uui ugii buia}$  have  $\forall k p q. \text{beta-upright } (\text{InfoAt } (\text{shd } s) k p q) (\text{GlobalIncomingInfo } (\text{shd } s) (\text{Suc } k) p q)$ 
unfolding  $\text{GlobalIncomingInfo-def}$ 
apply –
apply ( $\text{rule allI}$ )+
subgoal for  $k p q$ 
apply ( $\text{rule predicate-sum-decompose}[of UNIV p \lambda v. \text{beta-upright } (\text{InfoAt } (\text{shd } s) k p q) v \lambda p'. \text{IncomingInfo } (\text{shd } s) (\text{if } p' = p \text{ then Suc } k \text{ else } 0) p' q \text{ upright}]$ )
apply  $\text{simp}$ 

```

```

    apply simp
    apply simp
    apply (simp add: upright-sum-upright)
  subgoal for p' X
    apply (rule beta-upright-add)
    apply simp
    apply simp
    apply simp
    done
  subgoal for p' X
    apply (subst add.commute)
    apply (rule beta-upright-add)
    apply simp
    apply (simp add: upright-sum-upright)
    apply clarsimp
    apply simp
    done
  done
done
then show ?thesis
  by (simp add: InvGlobalInfoAtBetaUpright-def)
qed

```

**lemma** *alw-InvGlobalInfoAtBetaUpright*:  $\text{spec } s \implies \text{alw } (\text{holds } \text{InvGlobalInfoAtBetaUpright}) s$

**by** (*meson alw-InvGlobalIncomingInfoUpright alw-InvIncomingInfoUpright alw-InvInfoAtBetaUpright alw-iff-sdrop invs-imp-InvGlobalInfoAtBetaUpright*)

**definition** *SafeStickyGlobVacantUpto* :: ('p :: finite, 't :: order) *computation*  $\Rightarrow$  *bool* **where**

*SafeStickyGlobVacantUpto* s =  $(\forall q t. \text{GlobVacantUpto } (\text{shd } s) q t \longrightarrow \text{alw } (\text{holds } (\lambda c. \text{GlobVacantUpto } c q t)) s)$

**lemma** *gvut1*:

*GlobVacantUpto* (shd s) q t  $\implies$  *next-performop* s  $\implies$  *GlobVacantUpto* (shd (stl s)) q t

**by** (*auto simp add: next-performop'-def Let-def vacant-upto-def upright-def*)

**lemma** *gvut2*:

*GlobVacantUpto* (shd s) q t  $\implies$  *next-sendupd* s  $\implies$  *GlobVacantUpto* (shd (stl s)) q t

**by** (*auto simp add: next-sendupd'-def Let-def*)

**lemma** *gvut3*:

**assumes**

*gvu*: *GlobVacantUpto* (shd s) q t **and**

*igvvin*: *InvGlobVacantUptoImpliesNrec* (shd s) **and**

*igr*: *InvGlobalRecordCount* (shd s) **and**

*igiu*: *InvGlobalIncomingInfoUpright* (shd s) **and**

*igiabu: InvGlobalInfoAtBetaUpright (shd s) and*  
*next: next-recvupd s*  
**shows** *GlobVacantUpto (shd (stl s)) q t*  
**proof** –  
{ **fix** *p*  
**let** *?GII0 = GlobalIncomingInfo (shd s) 0 p q*  
**let** *?GII1 = GlobalIncomingInfo (shd s) 1 p q*  
**let** *?κ = hd (c-msg (shd s) p q)*  
**from** *igiuv* **have** *uGII1: upright ?GII1*  
**unfolding** *InvGlobalIncomingInfoUpright-def* **by** *simp*  
**assume** *globk: c-glob (shd (stl s)) q = c-glob (shd s) q + ?κ*  
**assume** *nonempty: c-msg (shd s) p q ≠ []*  
**then** **have** *sumGIIsk: ?GII0 = ?GII1 + ?κ*  
**unfolding** *GlobalIncomingInfo-def IncomingInfo-def*  
**by** *(auto simp: sum.remove ac-simps neg-Nil-conv)*  
**from** *nonempty* **have** *IA0k: ?κ = InfoAt (shd s) 0 p q*  
**by** *(simp add: InfoAt-def hd-conv-nth)*  
**from** *igiabu nonempty* **have** *bukGII1: beta-upright ?κ ?GII1*  
**proof** –  
**note** *igiabu*  
**then** **have** *beta-upright (InfoAt (shd s) 0 p q) (GlobalIncomingInfo (shd s) 1*  
*p q)*  
**by** *(simp add: InvGlobalInfoAtBetaUpright-def)*  
**with** *IA0k* **show** *?thesis*  
**by** *simp*  
**qed**  
**from** *igvuv* **have** *nvu: NrecVacantUpto (shd s) t*  
**unfolding** *InvGlobVacantUptoImpliesNrec-def* **by** *blast*  
**with** *igr* **have** *c-records (shd s) = c-glob (shd s) q + ?GII0*  
**unfolding** *GlobalIncomingInfo-def IncomingInfo-def InvGlobalRecordCount-def*  
**by** *(simp add: add commute)*  
**with** *gvu nvu* **have** *vuGII0: vacant-upto ?GII0 t*  
**by** *(simp add: vacant-upto-def)*  
**from** *bukGII1 uGII1* **have** *vacant-upto ?κ t*  
**by** *(rule betaupright-upright-vut[of ?κ ?GII1]) (metis vuGII0 add commute*  
*sumGIIsk)*  
**with** *gvu* **have** *GlobVacantUpto (shd (stl s)) q t*  
**by** *(simp add: globk vacant-upto-def)*  
**}**  
**then** **show** *?thesis*  
**using** *assms* **unfolding** *next-recvupd'-def*  
**by** *auto*  
**qed**

**lemma** *spec-imp-SafeStickyGlobVacantUpto-ax:*  
**assumes**  
*alw (holds (λc. InvGlobVacantUptoImpliesNrec c)) s and*  
*alw (holds (λc. InvGlobalRecordCount c)) s and*  
*alw (holds (λc. InvGlobalIncomingInfoUpright c)) s and*

```

    alw (holds ( $\lambda c. \text{InvGlobalInfoAtBetaUpright } c$ )) s and
    alw next s
shows alw SafeStickyGlobVacantUpto s
using assms apply (coinduction arbitrary: s)
subgoal for s
  unfolding spec-def next-def SafeStickyGlobVacantUpto-def Let-def
  apply (rule exI[of - s])
  apply safe
  subgoal for q t
    apply (coinduction arbitrary: s rule: alw.coinduct)
    apply clarsimp
    apply (rule conjI)
    apply blast
  proof -
    fix sb :: ('a, 'b) configuration stream
    assume a1: alw (holds InvGlobVacantUptoImpliesNrec) sb
    assume a2: alw (holds InvGlobalRecordCount) sb
    assume a3: alw (holds InvGlobalIncomingInfoUpright) sb
    assume a4: alw (holds InvGlobalInfoAtBetaUpright) sb
    assume a5: alw ( $\lambda s. \text{next-performop } s \vee \text{next-sendupd } s \vee \text{next-recvupd } s \vee$ 
    shd (stl s) = shd s) sb
    assume a6: GlobVacantUpto (shd sb) q t
    have  $\text{next-performop } sb \vee \text{next-sendupd } sb \vee \text{next-recvupd } sb \vee \text{shd (stl sb) =}$ 
    shd sb
      using a5 by blast
    then have GlobVacantUpto (shd (stl sb)) q t
      using a6 a4 a3 a2 a1 by (metis (no-types) alwD gvut1 gvut2 gvut3
    holds.elims(2))
    then show alw (holds InvGlobalRecordCount) (stl sb)  $\wedge$  alw (holds InvGlobal-
    incomingInfoUpright) (stl sb)  $\wedge$  alw (holds InvGlobalInfoAtBetaUpright) (stl sb)
     $\wedge$  alw ( $\lambda s. \text{next-performop } s \vee \text{next-sendupd } s \vee \text{next-recvupd } s \vee \text{shd (stl s) = shd}$ 
    s) (stl sb)  $\wedge$  GlobVacantUpto (shd (stl sb)) q t
      using a5 a4 a3 a2 by blast
    qed
    apply blast
  done
done

```

**lemma** *spec-imp-SafeStickyGlobVacantUpto*:  $\text{spec } s \implies \text{alw SafeStickyGlobVacantUpto } s$

```

apply (rule spec-imp-SafeStickyGlobVacantUpto-aux)
  apply (simp add: spec-imp-inv1)
  apply (simp add: alw-InvGlobalRecordCount)
  apply (simp add: alw-InvGlobalIncomingInfoUpright)
  apply (simp add: alw-InvGlobalInfoAtBetaUpright)
apply (simp add: spec-def)
done

```

**definition** *SafeGlobMono* **where**

$\text{SafeGlobMono } c0 \ c1 = (\forall p \ t. \text{GlobVacantUpto } c0 \ p \ t \longrightarrow \text{GlobVacantUpto } c1 \ p \ t)$

**lemma** *alw-SafeGlobMono*:  $\text{spec } s \Longrightarrow \text{alw } (\text{relates } \text{SafeGlobMono}) \ s$   
**apply** (*drule spec-imp-SafeStickyGlobVacantUpto*)  
**apply** (*erule alw-mono*)  
**apply** (*fastforce simp: SafeStickyGlobVacantUpto-def SafeGlobMono-def relates-def*)  
**done**

## 4 Exchange Protocol

### 4.1 Specification

**record**  $(p, t)$  *configuration* =  
*c-temp* ::  $'p \Rightarrow 't \text{ zmultiset}$   
*c-msg* ::  $'p \Rightarrow 'p \Rightarrow 't \text{ zmultiset list}$   
*c-glob* ::  $'p \Rightarrow 't \text{ zmultiset}$   
*c-caps* ::  $'p \Rightarrow 't \text{ zmultiset}$   
*c-data-msg* ::  $(p \times t) \text{ multiset}$

Description of the configuration: *c-msg*  $c \ p \ q$  global, all progress messages currently in-flight from  $p$  to  $q$  *c-data-msg*  $c$  global, capabilities carried by in-flight data messages *c-temp*  $c \ p$  local, aggregated progress updates of worker  $p$  that haven't been sent yet *c-glob*  $c \ p$  local, worker  $p$ 's conservative approximation of all capabilities in the system *c-caps*  $c \ p$  local, worker  $p$ 's capabilities

global = state of the whole system to which no worker has access local = state that is kept locally by each worker and which it can access

**type-synonym**  $(p, t)$  *computation* =  $(p, t)$  *configuration stream*

**context** *order begin*

**abbreviation** *timestamps*  $M \equiv \{\# \ t. (x, t) \in \#_z \ M \ \#\}$

**definition** *vacant-upto* ::  $'a \text{ zmultiset} \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
*vacant-upto*  $a \ t \equiv (\forall s. s \leq t \longrightarrow \text{zcount } a \ s = 0)$

**definition** *nonpos-upto* ::  $'a \text{ zmultiset} \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
*nonpos-upto*  $a \ t \equiv (\forall s. s \leq t \longrightarrow \text{zcount } a \ s \leq 0)$

**definition** *supported* ::  $'a \text{ zmultiset} \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
*supported*  $a \ t \equiv (\exists s. s < t \wedge \text{zcount } a \ s < 0)$

**definition** *supported-strong* ::  $'a \text{ zmultiset} \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
*supported-strong*  $a \ t \equiv (\exists s. s < t \wedge \text{zcount } a \ s < 0 \wedge \text{nonpos-upto } a \ s)$

**definition** *justified* **where**

$justified\ C\ M = (\forall t. 0 < zcount\ M\ t \longrightarrow supported\ M\ t \vee (\exists t' < t. 0 < zcount\ C\ t') \vee zcount\ M\ t < zcount\ C\ t)$

**lemma** *justified-alt*:

$justified\ C\ M = (\forall t. 0 < zcount\ M\ t \longrightarrow supported-strong\ M\ t \vee (\exists t' < t. 0 < zcount\ C\ t') \vee zcount\ M\ t < zcount\ C\ t)$

**unfolding** *justified-def supported-def supported-strong-def*

**apply** (*rule iffI*)

**apply** *clarsimp*

**apply** (*drule order-zmset-exists-foundation'*)

**apply** *clarsimp*

**subgoal for** *t s*

**apply** (*drule spec[of - s]*)

**apply** *safe*

**apply** (*meson le-less-trans less-le-trans nonpos-upto-def*)

**using** *order.strict-trans2* **apply** *blast*

**using** *order.order-iff-strict* **apply** *auto*

**done**

**apply** *blast*

**done**

**definition** *justified-with where*

$justified-with\ C\ M\ N =$

$(\forall t. 0 < zcount\ M\ t \longrightarrow$

$(\exists s < t. (zcount\ M\ s < 0 \vee zcount\ N\ s < 0)) \vee$

$(\exists s < t. 0 < zcount\ C\ s) \vee$

$zcount\ (M+N)\ t < zcount\ C\ t)$

**lemma** *justified-with-alt: justified-with C M N =*

$(\forall t. 0 < zcount\ M\ t \longrightarrow$

$(\exists s < t. (zcount\ M\ s < 0 \vee zcount\ N\ s < 0) \wedge (\forall s' < s. zcount\ M\ s' \leq 0)) \vee$

$(\exists s < t. 0 < zcount\ C\ s) \vee$

$zcount\ (M+N)\ t < zcount\ C\ t)$

**unfolding** *justified-with-def*

**apply** (*rule iffI*)

**apply** *clarsimp*

**apply** (*drule order-zmset-exists-foundation'*)

**apply** *clarsimp*

**subgoal for** *t s*

**apply** (*drule spec[of - s]*)

**apply** *safe*

**using** *order.strict-trans order.strict-trans2* **apply** *blast+*

**apply** (*simp add: not-less not-le*)

**apply** (*smt (verit, best) antisym-conv2*)

**done**

**apply** *blast*

**done**

**definition** *PositiveImplies where*

*PositiveImplies*  $v w \equiv \forall t. \text{zcount } v t > 0 \longrightarrow \text{zcount } w t > 0$

— A worker can mint capabilities greater or equal to any owned capability

**definition** *minting-self* **where**

*minting-self*  $C M = (\forall t \in \#M. \exists t' \leq t. 0 < \text{zcount } C t')$

— Sending messages mints a capability at a strictly greater pointstamp

**definition** *minting-msg* **where**

*minting-msg*  $C M = (\forall (p,t) \in \#M. \exists t' < t. 0 < \text{zcount } C t')$

**definition** *records* **where**

*records*  $c = (\sum p \in UNIV. \text{c-caps } c p) + \text{timestamps } (\text{zmsset-of } (c\text{-data-msg } c))$

**definition** *InfoAt* **where**

*InfoAt*  $c k p q = (\text{if } 0 \leq k \wedge k < \text{length } (c\text{-msg } c p q) \text{ then } (c\text{-msg } c p q) ! k \text{ else } \{\#\}_z)$

**definition** *IncomingInfo*  $:: ('p, 'a) \text{ configuration} \Rightarrow \text{nat} \Rightarrow 'p \Rightarrow 'p \Rightarrow 'a \text{ zmultiset}$   
**where**

*IncomingInfo*  $c k p q \equiv \text{sum-list } (\text{drop } k (c\text{-msg } c p q)) + c\text{-temp } c p$

**definition** *GlobalIncomingInfo*  $:: ('p :: \text{finite}, 'a) \text{ configuration} \Rightarrow \text{nat} \Rightarrow 'p \Rightarrow 'p$   
 $\Rightarrow 'a \text{ zmultiset}$  **where**

*GlobalIncomingInfo*  $c k p q \equiv \sum p' \in UNIV. \text{IncomingInfo } c (\text{if } p' = p \text{ then } k \text{ else } 0) p' q$

**abbreviation** *GlobalIncomingInfoAt* **where**

*GlobalIncomingInfoAt*  $c q \equiv \text{GlobalIncomingInfo } c 0 q q$

**definition** *init-config*  $:: ('p :: \text{finite}, 'a) \text{ configuration} \Rightarrow \text{bool}$  **where**

*init-config*  $c \equiv$

$(\forall p. c\text{-temp } c p = \{\#\}_z) \wedge$

$(\forall p1 p2. c\text{-msg } c p1 p2 = []) \wedge$

— Capabilities have non-negative multiplicities

$(\forall p t. 0 \leq \text{zcount } (c\text{-caps } c p) t) \wedge$

— The pointstamps in *glob* are exactly those in *records*

$(\forall p. c\text{-glob } c p = \text{records } c) \wedge$

— All capabilities are being tracked

$c\text{-data-msg } c = \{\#\}$

**definition** *next-recvcap'*  $:: ('p :: \text{finite}, 'a) \text{ configuration} \Rightarrow ('p, 'a) \text{ configuration}$   
 $\Rightarrow 'p \Rightarrow 'a \Rightarrow \text{bool}$  **where**

*next-recvcap'*  $c0 c1 p t = ($

$(p,t) \in \# c\text{-data-msg } c0$

$\wedge c1 = c0 \setminus (c\text{-caps } := (c\text{-caps } c0)(p := c\text{-caps } c0 p + \{\#t\#}_z),$   
 $c\text{-data-msg } := c\text{-data-msg } c0 - \{\#(p,t)\#})$ )

**abbreviation** *next-recvcap* **where**

$$\text{next-recvcap } c0 \ c1 \equiv \exists p \ t. \text{ next-recvcap}' \ c0 \ c1 \ p \ t$$

Can minting of capabilities be described as a refinement of the Abadi model? Short answer: No, not in general. Long answer: Could slightly modify Abadi model, such that a capability always comes with a multiplicity  $2^64$  (or similar, could be parametrized over arbitrarily large constant). In that case minting new capabilities can be described as an upright change, dropping one of the capabilities, to make the change upright. This only works as long as no capability is required more than the constant number of times. Issues: - Not fully general, due to the arbitrary constant - Not clear whether refinement proofs would be easier than simply altering the model to support the operations

Rationale for the condition on *c-caps* *c0 p*: In Abadi, the operation *next-performop'* has the premise  $\forall t. \text{int}(\text{count } \Delta \text{neg } t) \leq \text{zcount}(\text{records } c0) \ t$ , (records corresponds to the global field *nrec* in that model) which means the processor performing the transition must verify that this condition is met. Since *records c* is "global" state, which no processor can know, an implementation of this protocol has to include some other protocol or reasoning for when it is safe to do this transition.

Naively using a processor's *c-glob c p* to approximate *records c* and justify transitions can cause a race condition, where a processor drops a pointstamp, e.g.,  $\Delta \text{neg} = \{\#t\# \}$ , after which  $\text{zcount}(\text{records } c) \ t = 0$  but other processors might still use the pointstamp to justify the creation of pointstamps that violate the safety property.

Instead we model ownership of pointstamps, calling "owned pointstamps" **capabilities**, which are tracked in *c-caps c*. In place of *nrec* we define *records c*, which is the sum of all capabilities, as well as *c-data-msg c*, which contains the capabilities carried by data messages. Since  $\forall p \ t. \text{zcount}(c\text{-caps } c \ p) \ t \leq \text{zcount}(\text{records } c) \ t$ , our condition  $\forall t. \text{int}(\text{count } \Delta \text{neg } t) \leq \text{zcount}(c\text{-caps } c0 \ p) \ t$  implies the one on *nrec* in Abadi's model.

Conditions in *performop*:

The *performop* transition takes three msets of pointstamps,  $\Delta \text{neg}$ ,  $\Delta \text{mint-msg}$ , and  $\Delta \text{mint-self}$   $\Delta \text{neg}$  contains dropped capabilities (a subset of *c-caps*)  $\Delta \text{mint-msg}$  contains pairs  $(p, t)$ , where a data message is sent (i.e. capability added to the pool), creating a capability at *t*, owned by *p*  $\Delta \text{mint-self}$  contains pointstamps minted and owned by worker *p*

$\Delta \text{neg}$  in combination with  $\Delta \text{mint-msg}$  also allows any upright updates to be made as in the Abadi model, meaning this definition allows strictly more behaviors.

The  $\Delta \text{mint-msg} \neq \{\#\} \vee \text{zmsset-of } \Delta \text{mint-self} - \text{zmsset-of } \Delta \text{neg} \neq \{\#\}_z$  condition ensures that no-ops aren't possible. However, it's still possible

that the combined  $\Delta$  is empty. E.g. a processor has capabilities 1 and 2, uses cap 1 to send a message, minting capability 2. Simultaneously it drops a capability 2 (for unrelated reasons), cancelling out the overall change but shifting a capability to the pool, possibly with a different owner than itself.

**definition**  $next\text{-performop}' :: ('p::finite, 'a) configuration \Rightarrow ('p, 'a) configuration \Rightarrow 'p \Rightarrow 'a \text{ multiset} \Rightarrow ('p \times 'a) \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$  **where**

$next\text{-performop}' c0 c1 p \Delta_{neg} \Delta_{mint\text{-msg}} \Delta_{mint\text{-self}} =$   
—  $\Delta_{pos}$  contains all positive changes,  $\Delta$  the combined positive and negative changes

(let  $\Delta_{pos} = \text{timestamps} (\text{zmsset-of } \Delta_{mint\text{-msg}}) + \text{zmsset-of } \Delta_{mint\text{-self}};$   
 $\Delta = \Delta_{pos} - \text{zmsset-of } \Delta_{neg}$

in

$(\Delta_{mint\text{-msg}} \neq \{\#\}) \vee \text{zmsset-of } \Delta_{mint\text{-self}} - \text{zmsset-of } \Delta_{neg} \neq \{\#\}_z)$

$\wedge (\forall t. \text{int} (\text{count } \Delta_{neg} t) \leq \text{zcount} (c\text{-caps } c0 p) t)$

— Pointstamps added in  $\Delta_{mint\text{-self}}$  are minted at  $p$

$\wedge \text{minting-self } (c\text{-caps } c0 p) \Delta_{mint\text{-self}}$

— Pointstamps added in  $\Delta_{mint\text{-msg}}$  correspond to sent data messages

$\wedge \text{minting-msg } (c\text{-caps } c0 p) \Delta_{mint\text{-msg}}$

— Worker immediately knows about dropped and minted capabilities

$\wedge c1 = c0 \llbracket c\text{-caps} := (c\text{-caps } c0)(p := c\text{-caps } c0 p + \text{zmsset-of } \Delta_{mint\text{-self}} - \text{zmsset-of } \Delta_{neg}),$

— Sending a data message creates a capability, once that message arrives. This is modelled as a pool of capabilities that may (will) appear at processors at some point.

$c\text{-data-msg} := c\text{-data-msg } c0 + \Delta_{mint\text{-msg}},$   
 $c\text{-temp} := (c\text{-temp } c0)(p := c\text{-temp } c0 p + \Delta)$

**abbreviation**  $next\text{-performop}$  **where**

$next\text{-performop } c0 c1 \equiv (\exists p \Delta_{neg} \Delta_{mint\text{-msg}} \Delta_{mint\text{-self}}. next\text{-performop}' c0 c1 p \Delta_{neg} \Delta_{mint\text{-msg}} \Delta_{mint\text{-self}})$

**definition**  $next\text{-sendupd}' :: ('p::finite, 'a) configuration \Rightarrow ('p, 'a) configuration \Rightarrow 'p \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$  **where**

$next\text{-sendupd}' c0 c1 p tt =$

(let  $\gamma = \{\#t \in \#_z c\text{-temp } c0 p. t \in tt\# \}$  in

$\gamma \neq 0$

$\wedge \text{justified } (c\text{-caps } c0 p) (c\text{-temp } c0 p - \gamma)$

$\wedge c1 = c0 \llbracket c\text{-msg} := (c\text{-msg } c0)(p := \lambda q. c\text{-msg } c0 p q @ [\gamma]),$

$c\text{-temp} := (c\text{-temp } c0)(p := c\text{-temp } c0 p - \gamma)\rrbracket)$

**abbreviation**  $next\text{-sendupd}$  **where**

$next\text{-sendupd } c0 c1 \equiv (\exists p tt. next\text{-sendupd}' c0 c1 p tt)$

**definition**  $next\text{-recvupd}' :: ('p::finite, 'a) configuration \Rightarrow ('p, 'a) configuration \Rightarrow 'p \Rightarrow 'p \Rightarrow \text{bool}$  **where**

$next\text{-recvupd}' c0 c1 p q \equiv$

$c\text{-msg } c0 p q \neq \square$

$\wedge c1 = c0 \llbracket c\text{-msg} := (c\text{-msg } c0)(p := (c\text{-msg } c0 p)(q := \text{tl } (c\text{-msg } c0 p q))),$

$c\text{-glob} := (c\text{-glob } c0)(q := c\text{-glob } c0 q + \text{hd } (c\text{-msg } c0 p q))\rrbracket)$

**abbreviation** *next-recvupd* **where**

$$\text{next-recvupd } c0 \ c1 \equiv (\exists p \ q. \text{next-recvupd}' \ c0 \ c1 \ p \ q)$$

**definition** *next'* **where**

$$\text{next}' \ c0 \ c1 = (\text{next-performop } c0 \ c1 \vee \text{next-sendupd } c0 \ c1 \vee \text{next-recvupd } c0 \ c1 \vee \text{next-recvcap } c0 \ c1 \vee c1 = c0)$$

**abbreviation** *next* **where**

$$\text{next } s \equiv \text{next}' \ (\text{shd } s) \ (\text{shd } (\text{stl } s))$$

**definition** *spec* :: ('p :: finite, 'a) computation  $\Rightarrow$  bool **where**

$$\text{spec } s \equiv \text{holds init-config } s \wedge \text{alw next } s$$

**abbreviation** *GlobVacantUpto* **where**

$$\text{GlobVacantUpto } c \ q \ t \equiv \text{vacant-upto } (c\text{-glob } c \ q) \ t$$

**abbreviation** *GlobNonposUpto* **where**

$$\text{GlobNonposUpto } c \ q \ t \equiv \text{nonpos-upto } (c\text{-glob } c \ q) \ t$$

**abbreviation** *RecordsVacantUpto* **where**

$$\text{RecordsVacantUpto } c \ t \equiv \text{vacant-upto } (\text{records } c) \ t$$

**definition** *SafeGlobVacantUptoImpliesStickyNrec* :: ('p :: finite, 'a) computation  $\Rightarrow$  bool **where**

$$\begin{aligned} \text{SafeGlobVacantUptoImpliesStickyNrec } s = \\ (\text{let } c = \text{shd } s \text{ in } \forall t \ q. \text{GlobVacantUpto } c \ q \ t \longrightarrow \text{alw } (\text{holds } (\lambda c. \text{RecordsVacantUpto } c \ t)) \ s) \end{aligned}$$

## 4.2 Auxiliary Lemmas

**lemma** *finite-induct-select* [consumes 1, case-names empty select]:

**assumes** *finite* *S*

**and** *empty*:  $P \ \{\}$

**and** *select*:  $\bigwedge T. \text{finite } T \Longrightarrow T \subset S \Longrightarrow P \ T \Longrightarrow \exists s \in S - T. P \ (\text{insert } s \ T)$

**shows**  $P \ S$

**proof** –

**from** *assms*(1) **have**  $P \ S \wedge \text{finite } S$

**by** (*induct* *S* rule: *finite-induct-select*) (*auto intro*: *empty select*)

**then** **show** *?thesis* **by** *blast*

**qed**

**lemma** *finite-induct-decompose-sum*:

**fixes**  $f :: 'c \Rightarrow ('b :: \text{comm-monoid-add})$

**assumes** *finite* *X*

**and**  $x \in X$

**and**  $A \ (f \ x)$

**and**  $\forall Z. B \ (\text{sum } f \ Z)$

**and**  $\bigwedge x \ Z. A \ (f \ x) \Longrightarrow B \ (\text{sum } f \ Z) \Longrightarrow A \ (f \ x + \text{sum } f \ Z)$

```

and  $\bigwedge x Z. B (f x) \implies A (sum f Z) \implies A (f x + sum f Z)$ 
shows  $A (\sum_{x \in X}. f x)$ 
using assms(1,2,3)
apply (induct X rule: finite-induct-select)
apply simp
apply (simp add: sum.insert-remove)
subgoal for T
apply (cases x ∈ T; simp add: assms(3))
apply (drule psubset-imp-ex-mem)
apply clarsimp
subgoal for z
apply (rule bexI[of - z])
apply (rule conjI)
apply clarsimp
apply (rule assms(6)[of z T])
apply (rule assms(4)[THEN spec, of {z}, simplified])
apply simp
apply simp
done
apply clarsimp
apply (drule bspec[of - - x])
apply safe
apply (rule assms(2))
using assms(4) assms(5) apply blast
done
done

```

**lemma** *minting-msg-add-records*:  $minting\text{-}msg\ C1\ M \implies \forall t. 0 \leq zcount\ C2\ t \implies minting\text{-}msg\ (C1 + C2)\ M$   
**by** (*auto simp: minting-msg-def intro: add-strict-increasing dest!: bspec*)

**lemma** *add-less*:  $(a::int) < c \implies b \leq 0 \implies a + b < c$   
**by** *linarith*

**lemma** *disj3-split*:  $P \vee Q \vee R \implies (P \implies thesis) \implies (\neg P \wedge Q \implies thesis) \implies (\neg P \implies \neg Q \implies R \implies thesis) \implies thesis$   
**by** *blast*

**lemma** *filter-zmset-conclude-predicate*:  $0 < zcount\ \{\# x \in \#_z\ M. P\ x\ \#\}\ x \implies 0 < zcount\ M\ x \implies P\ x$   
**by** (*auto split: if-splits*)

**lemma** *alw-holds2*:  $alw\ (holds\ P)\ ss = (P\ (shd\ ss) \wedge alw\ (holds\ P)\ (stl\ ss))$   
**by** (*meson alw.simps holds.elims(2) holds.elims(3)*)

**lemma** *zmset-of-remove1-mset*:  $x \in \# M \implies zmset\text{-}of\ (remove1\text{-}mset\ x\ M) = zmset\text{-}of\ M - \{\#x\#\}_z$   
**by** (*induct M*) *auto*

**lemma** *timestamps-zmset-of-pair-image[simp]*:  $\text{timestamps } (\text{zmset-of } \{\# (c,t). t \in \# M \#\}) = \text{zmset-of } M$   
**by** (*induct M*) *auto*

**lemma** *timestamps-image-zmset-fst[simp]*:  $\text{timestamps } \{\# (f x, t). (x, t) \in \#_z M \#\} = \text{timestamps } M$   
**apply** *transfer*  
**apply** (*clarsimp simp: equiv-zmset-def*)  
**apply** (*metis (no-types, lifting) case-prod-unfold image-mset-cong prod.collapse prod.inject*)  
**done**

**lemma** *lift-invariant-to-spec*:  
**assumes** ( $\bigwedge c. \text{init-config } c \implies P c$ )  
**and** ( $\bigwedge s. \text{holds } P s \implies \text{next } s \implies \text{next } (\text{holds } P) s$ )  
**shows**  $\text{spec } s \implies \text{alw } (\text{holds } P) s$   
**unfolding** *spec-def*  
**apply** (*elim conjE*)  
**apply** (*coinduction arbitrary: s*)  
**apply** *clarsimp*  
**apply** (*intro conjI assms(1)*)  
**apply** *safe*  
**subgoal**  
**proof** –  
**fix** *sa* :: (*'b, 'a*) *configuration stream*  
**assume** *a1*: *init-config (shd sa)*  
**assume** *a2*: *alw next sa*  
**assume**  $\neg \text{alw } (\text{holds } P) (\text{stl } sa)$   
**then have**  $\neg \text{alw } (\lambda s. \text{holds } P s \longrightarrow \text{next } (\text{holds } P) s) sa$   
**using** *a1* **by** (*metis (no-types) alw.cases alw-invar assms(1) holds.elims(3)*)  
**then show** *init-config (shd (stl sa))*  
**using** *a2* **by** (*metis (lifting) alw-iff-sdrop assms(2)*)  
**qed**  
**apply** *auto*  
**done**

**lemma** *timestamps-sum-distrib[simp]*:  $(\sum p \in A. \text{timestamps } (f p)) = \text{timestamps } (\sum p \in A. f p)$   
**by** (*induction A rule: infinite-finite-induct*) *auto*

**lemma** *timestamps-zmset-of[simp]*:  $\text{timestamps } (\text{zmset-of } M) = \text{zmset-of } \{\# t. (p,t) \in \# M \#\}$   
**by** (*induct M*) *auto*

**lemma** *vacant-upto-add*:  $\text{vacant-upto } a t \implies \text{vacant-upto } b t \implies \text{vacant-upto } (a+b) t$   
**by** (*simp add: vacant-upto-def*)

**lemma** *nonpos-upto-add*:  $\text{nonpos-upto } a t \implies \text{nonpos-upto } b t \implies \text{nonpos-upto } (a+b) t$

$(a+b) t$   
**by** (*auto intro: add-nonpos-nonpos simp: nonpos-upto-def*)

**lemma nonzero-lt-gtD:**  $(n:::\text{linorder}) \neq 0 \implies 0 < n \vee n < 0$   
**by** *auto*

**lemma zero-lt-diff:**  $(0:::\text{int}) < a - b \implies b \geq 0 \implies 0 < a$   
**by** *auto*

**lemma zero-lt-add-disj:**  $0 < (a:::\text{int}) + b \implies 0 \leq a \implies 0 \leq b \implies 0 < a \vee 0 < b$   
**by** *auto*

#### 4.2.1 Transition lemmas

**lemma next-performopD:**

**assumes** *next-performop' c0 c1 p  $\Delta$ neg  $\Delta$ mint-msg  $\Delta$ mint-self*

**shows**

$\Delta$ mint-msg  $\neq \{\#\}$   $\vee$  *zmsset-of  $\Delta$ mint-self - zmsset-of  $\Delta$ neg  $\neq \{\#\}_z$*   
 $\forall t. \text{int} (\text{count } \Delta$ neg  $t) \leq \text{zcount} (c$ -caps  $c0 p) t$   
*minting-self (c-caps c0 p)  $\Delta$ mint-self*  
*minting-msg (c-caps c0 p)  $\Delta$ mint-msg*  
 $c$ -temp  $c1 = (c$ -temp  $c0)(p := c$ -temp  $c0 p + (\text{timestamps} (\text{zmsset-of } \Delta$ mint-msg)  
 $+ \text{zmsset-of } \Delta$ mint-self -  $\text{zmsset-of } \Delta$ neg))  
 $c$ -msg  $c1 = c$ -msg  $c0$   
 $c$ -glob  $c1 = c$ -glob  $c0$   
 $c$ -data-msg  $c1 = c$ -data-msg  $c0 + \Delta$ mint-msg  
 $c$ -caps  $c1 = (c$ -caps  $c0)(p := c$ -caps  $c0 p + (\text{zmsset-of } \Delta$ mint-self -  $\text{zmsset-of } \Delta$ neg))

**using** *assms by (simp-all add: next-performop'-def Let-def algebra-simps)*

**lemma next-performop-complexD:**

**assumes** *next-performop' c0 c1 p  $\Delta$ neg  $\Delta$ mint-msg  $\Delta$ mint-self*

**shows**

*records c1 = records c0 + (timestamps (zmsset-of  $\Delta$ mint-msg) + zmsset-of  $\Delta$ mint-self - zmsset-of  $\Delta$ neg)*  
 $\text{GlobalIncomingInfoAt } c1 q = \text{GlobalIncomingInfoAt } c0 q + (\text{timestamps} (\text{zmsset-of } \Delta$ mint-msg) +  $\text{zmsset-of } \Delta$ mint-self -  $\text{zmsset-of } \Delta$ neg)  
 $\text{IncomingInfo } c1 k p' q = (\text{if } p' = p$   
*then IncomingInfo c0 k p' q + (timestamps (zmsset-of  $\Delta$ mint-msg) + zmsset-of  $\Delta$ mint-self - zmsset-of  $\Delta$ neg)*  
*else IncomingInfo c0 k p' q)*  
 $\forall t' < t. \text{zcount} (c$ -caps  $c0 p) t' = 0 \implies \text{zcount} (\text{timestamps} (\text{zmsset-of } \Delta$ mint-msg))  
 $t = 0$   
 $\text{InfoAt } c1 k p' q = \text{InfoAt } c0 k p' q$

**proof** –

**let**  $?\Delta = \text{timestamps} (\text{zmsset-of } \Delta$ mint-msg) +  $\text{zmsset-of } \Delta$ mint-self -  $\text{zmsset-of } \Delta$ neg

**note**  $\text{change} = \text{next-performopD}[OF \text{assms}(1)]$

**show**  $records\ c1 = records\ c0 + ?\Delta$   
**unfolding**  $records\text{-}def\ change$   
**apply**  $simp$   
**apply**  $(subst\ add\text{-}diff\text{-}eq[symmetric])$   
**apply**  $(subst\ sum\text{-}if\text{-}distrib\text{-}add)$   
**apply**  $(simp\text{-}all\ add:\ algebra\text{-}simps\ zmsset\text{-}of\text{-}plus)$   
**done**  
**show**  $IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p$   
 $then\ IncomingInfo\ c0\ k\ p'\ q + (timestamps\ (zmsset\text{-}of\ \Delta\ mint\text{-}msg) + zmsset\text{-}of$   
 $\Delta\ mint\text{-}self - zmsset\text{-}of\ \Delta\ neg)$   
 $else\ IncomingInfo\ c0\ k\ p'\ q)$   
**unfolding**  $IncomingInfo\text{-}def\ change$   
**by**  $(auto\ simp:\ algebra\text{-}simps)$   
**show**  $GlobalIncomingInfoAt\ c1\ q = GlobalIncomingInfoAt\ c0\ q + ?\Delta\ for\ q$   
**unfolding**  $GlobalIncomingInfo\text{-}def\ IncomingInfo\text{-}def$   
**by**  $(rule\ Sum\text{-}eq\text{-}pick\text{-}changed\text{-}elem[\mathbf{where}\ m = p])\ (simp\text{-}all\ add:\ change\ alge\text{-}bra\text{-}simps)$   
**show**  $\forall t' < t. zcount\ (c\text{-}caps\ c0\ p)\ t' = 0 \implies zcount\ (timestamps\ (zmsset\text{-}of$   
 $\Delta\ mint\text{-}msg))\ t = 0\ for\ t$   
**by**  $(rule\ ccontr)\ (clarsimp\ dest!:\ image\text{-}zmsset\text{-}pre\ change(4)[unfolding\ mint\text{-}ing\text{-}msg\text{-}def,\ rule\text{-}format])$   
**show**  $InfoAt\ c1\ k\ p'\ q = InfoAt\ c0\ k\ p'\ q$   
**unfolding**  $InfoAt\text{-}def\ change\ by\ simp$   
**qed**

**lemma**  $next\text{-}sendupdD$ :

**assumes**  $next\text{-}sendupd'\ c0\ c1\ p\ tt$   
**shows**  
 $\{\#t \in \#_z\ c\text{-}temp\ c0\ p. t \in tt\# \} \neq \{\#\}_z$   
 $justified\ (c\text{-}caps\ c0\ p)\ (c\text{-}temp\ c0\ p - \{\#t \in \#_z\ c\text{-}temp\ c0\ p. t \in tt\# \})$   
 $c\text{-}temp\ c1\ p' = (if\ p' = p\ then\ c\text{-}temp\ c0\ p - \{\#t \in \#_z\ c\text{-}temp\ c0\ p. t \in tt\# \}$   
 $else\ c\text{-}temp\ c0\ p')$   
 $c\text{-}msg\ c1 = (\lambda p'\ q. if\ p' = p\ then\ c\text{-}msg\ c0\ p\ q @ [\{\#t \in \#_z\ c\text{-}temp\ c0\ p. t \in$   
 $tt\#\}] else\ c\text{-}msg\ c0\ p'\ q)$   
 $c\text{-}glob\ c1 = c\text{-}glob\ c0$   
 $c\text{-}caps\ c1 = c\text{-}caps\ c0$   
 $c\text{-}data\text{-}msg\ c1 = c\text{-}data\text{-}msg\ c0$   
**using**  $assms\ by\ (simp\text{-}all\ add:\ next\text{-}sendupd'\text{-}def\ Let\text{-}def\ fun\text{-}eq\text{-}iff)$

**lemma**  $next\text{-}sendupd\text{-}complexD$ :

**assumes**  $next\text{-}sendupd'\ c0\ c1\ p\ tt$   
**shows**  
 $records\ c1 = records\ c0$   
 $IncomingInfo\ c1\ 0 = IncomingInfo\ c0\ 0$   
 $IncomingInfo\ c1\ k\ p'\ q = (if\ p' = p \wedge length\ (c\text{-}msg\ c0\ p\ q) < k$   
 $then\ IncomingInfo\ c0\ k\ p'\ q - \{\#t \in \#_z\ c\text{-}temp\ c0\ p'. t \in$   
 $tt\#\}$   
 $else\ IncomingInfo\ c0\ k\ p'\ q)$   
 $k \leq length\ (c\text{-}msg\ c0\ p\ q) \implies IncomingInfo\ c1\ k\ p'\ q = IncomingInfo\ c0\ k\ p'\ q$

$length (c\text{-msg } c0 \ p \ q) < k \implies$   
 $IncomingInfo \ c1 \ k \ p' \ q = (if \ p' = p$   
 $\quad then \ IncomingInfo \ c0 \ k \ p' \ q - \{\#t \in \#_z \ c\text{-temp } c0 \ p'. \ t \in$   
 $tt\#\}$   
 $\quad else \ IncomingInfo \ c0 \ k \ p' \ q)$   
 $GlobalIncomingInfoAt \ c1 \ q = GlobalIncomingInfoAt \ c0 \ q$   
 $InfoAt \ c1 \ k \ p' \ q = (if \ p' = p \wedge k = length (c\text{-msg } c0 \ p \ q) \ then \ \{\#t \in \#_z \ c\text{-temp}$   
 $c0 \ p'. \ t \in tt\#\} \ else \ InfoAt \ c0 \ k \ p' \ q)$   
**proof** –  
**note**  $change = next\text{-sendupdD}[OF \ assms]$   
**show**  $records \ c1 = records \ c0$   
**by** (*simp add: records-def change*)  
**show** *ii*:  $IncomingInfo \ c1 \ k \ p' \ q = (if \ p' = p \wedge length (c\text{-msg } c0 \ p \ q) < k$   
 $\quad then \ IncomingInfo \ c0 \ k \ p' \ q - \{\#t \in \#_z \ c\text{-temp } c0$   
 $p'. \ t \in tt\#\}$   
 $\quad else \ IncomingInfo \ c0 \ k \ p' \ q)$   
**by** (*simp add: algebra-simps IncomingInfo-def change*)  
**then show**  $k \leq length (c\text{-msg } c0 \ p \ q) \implies IncomingInfo \ c1 \ k \ p' \ q = IncomingInfo$   
 $c0 \ k \ p' \ q$   
**by** *auto*  
**from** *ii* **show**  $length (c\text{-msg } c0 \ p \ q) < k \implies$   
 $IncomingInfo \ c1 \ k \ p' \ q = (if \ p' = p$   
 $\quad then \ IncomingInfo \ c0 \ k \ p' \ q - \{\#t \in \#_z \ c\text{-temp } c0 \ p'. \ t \in$   
 $tt\#\}$   
 $\quad else \ IncomingInfo \ c0 \ k \ p' \ q)$   
**by** *auto*  
**have**  $IncomingInfo \ c1 \ 0 \ p \ q = IncomingInfo \ c0 \ 0 \ p \ q$  **for**  $p \ q$   
**by** (*simp add: algebra-simps IncomingInfo-def change*)  
**then show**  $IncomingInfo \ c1 \ 0 = IncomingInfo \ c0 \ 0$   
**by** *auto*  
**then show**  $GlobalIncomingInfoAt \ c1 \ q = GlobalIncomingInfoAt \ c0 \ q$   
**unfolding** *GlobalIncomingInfo-def* **by** *auto*  
**show**  $InfoAt \ c1 \ k \ p' \ q = (if \ p' = p \wedge k = length (c\text{-msg } c0 \ p \ q) \ then \ \{\#t \in \#_z$   
 $c\text{-temp } c0 \ p'. \ t \in tt\#\} \ else \ InfoAt \ c0 \ k \ p' \ q)$   
**unfolding** *InfoAt-def change*  
**by** (*auto simp: nth-append*)  
**qed**

**lemma** *next-recvupdD*:

**assumes**  $next\text{-recvupd}' \ c0 \ c1 \ p \ q$

**shows**

$c\text{-msg } c0 \ p \ q \neq []$

$c\text{-temp } c1 = c\text{-temp } c0$

$c\text{-msg } c1 = (\lambda p' \ q'. \ if \ p' = p \wedge q' = q \ then \ tl (c\text{-msg } c0 \ p \ q) \ else \ c\text{-msg } c0 \ p'$   
 $q')$

$c\text{-glob } c1 = (c\text{-glob } c0)(q := c\text{-glob } c0 \ q + hd (c\text{-msg } c0 \ p \ q))$

$c\text{-caps } c1 = c\text{-caps } c0$

$c\text{-data-msg } c1 = c\text{-data-msg } c0$

**using** *assms* **by** (*simp-all add: next-recvupd'-def fun-eq-iff*)

**lemma** *next-recvupd-complexD*:

**assumes** *next-recvupd'*  $c0\ c1\ p\ q$

**shows**

*records*  $c1 = \text{records } c0$

*IncomingInfo*  $c1\ 0\ p'\ q' = (\text{if } p' = p \wedge q' = q \text{ then } \text{IncomingInfo } c0\ 0\ p'\ q' - \text{hd } (c\text{-msg } c0\ p\ q) \text{ else } \text{IncomingInfo } c0\ 0\ p'\ q')$

*IncomingInfo*  $c1\ k\ p'\ q' = (\text{if } p' = p \wedge q' = q$   
      $\text{then } \text{IncomingInfo } c0\ (k+1)\ p'\ q'$   
      $\text{else } \text{IncomingInfo } c0\ k\ p'\ q')$

*GlobalIncomingInfoAt*  $c1\ q' = (\text{if } q' = q \text{ then } \text{GlobalIncomingInfoAt } c0\ q' - \text{hd } (c\text{-msg } c0\ p\ q) \text{ else } \text{GlobalIncomingInfoAt } c0\ q')$

*InfoAt*  $c1\ k\ p\ q = \text{InfoAt } c0\ (k+1)\ p\ q$

*InfoAt*  $c1\ k\ p'\ q' = (\text{if } p' = p \wedge q' = q \text{ then } \text{InfoAt } c0\ (k+1)\ p\ q \text{ else } \text{InfoAt } c0\ k\ p'\ q')$

**proof** –

**note** *change* = *next-recvupdD[OF assms]*

**show** *records*  $c1 = \text{records } c0$

**by** (*simp add: records-def change*)

**show** *ii*: *IncomingInfo*  $c1\ 0\ p'\ q' = (\text{if } p' = p \wedge q' = q \text{ then } \text{IncomingInfo } c0\ 0\ p'\ q' - \text{hd } (c\text{-msg } c0\ p\ q) \text{ else } \text{IncomingInfo } c0\ 0\ p'\ q')$  **for**  $p'\ q'$

**by** (*auto simp: IncomingInfo-def change algebra-simps sum-list-hd-tl*)

**show** *IncomingInfo*  $c1\ k\ p'\ q' = (\text{if } p' = p \wedge q' = q$   
      $\text{then } \text{IncomingInfo } c0\ (k+1)\ p'\ q'$   
      $\text{else } \text{IncomingInfo } c0\ k\ p'\ q')$

**by** (*auto simp: IncomingInfo-def change algebra-simps sum-list-hd-tl drop-Suc*)

**show** *GlobalIncomingInfoAt*  $c1\ q' = (\text{if } q' = q \text{ then } \text{GlobalIncomingInfoAt } c0\ q' - \text{hd } (c\text{-msg } c0\ p\ q) \text{ else } \text{GlobalIncomingInfoAt } c0\ q')$

**unfolding** *GlobalIncomingInfo-def*

**apply** (*cases*  $q'=q$ )

**apply** *simp*

**apply** (*subst diff-conv-add-uminus*)

**apply** (*intro Sum-eq-pick-changed-elem*[**where**  $m = p$ ])

**apply** (*simp-all add: ii*)

**done**

**show** *InfoAt*  $c1\ k\ p\ q = \text{InfoAt } c0\ (k+1)\ p\ q$

**unfolding** *InfoAt-def change*

**by** (*auto simp: nth-tl*)

**show** *InfoAt*  $c1\ k\ p'\ q' = (\text{if } p' = p \wedge q' = q \text{ then } \text{InfoAt } c0\ (k+1)\ p\ q \text{ else } \text{InfoAt } c0\ k\ p'\ q')$

**unfolding** *InfoAt-def change*

**by** (*auto simp: nth-tl*)

**qed**

**lemma** *next-recvcapD*:

**assumes** *next-recvcap'*  $c0\ c1\ p\ t$

**shows**

$(p,t) \in \# c\text{-data-msg } c0$

*c-temp*  $c1 = c\text{-temp } c0$

```

c-msg c1 = c-msg c0
c-glob c1 = c-glob c0
c-caps c1 = (c-caps c0)(p := c-caps c0 p + {#t#}_z)
c-data-msg c1 = c-data-msg c0 - {#(p,t)#}
using assms by (simp-all add: next-recvcap'-def)

```

**lemma** *next-recvcap-complexD*:

**assumes** *next-recvcap'* c0 c1 p t

**shows**

```

records c1 = records c0
IncomingInfo c1 = IncomingInfo c0
GlobalIncomingInfo c1 = GlobalIncomingInfo c0
InfoAt c1 k p' q = InfoAt c0 k p' q

```

**proof** –

**note** *change* = *next-recvcapD*[*OF* *assms*]

**show** *records* c1 = *records* c0

**unfolding** *records-def* *change* *fun-upd-apply*

**apply** (*subst* *sum-if-distrib-add*)

**using** *change*(1) **apply** (*simp-all* add: *zmset-of-remove1-mset algebra-simps*  
*records-def* *change*)

**done**

**show** *IncomingInfo* c1 = *IncomingInfo* c0

**unfolding** *IncomingInfo-def* *change* **by** *simp*

**then show** *GlobalIncomingInfo* c1 = *GlobalIncomingInfo* c0

**unfolding** *GlobalIncomingInfo-def* **by** *simp*

**show** *InfoAt* c1 k p' q = *InfoAt* c0 k p' q

**unfolding** *InfoAt-def* *change* **by** *simp*

**qed**

**lemma** *ex-next-recvupd*:

**assumes** *c-msg* c0 p q ≠ []

**shows** ∃ c1. *next-recvupd'* c0 c1 p q

**using** *assms* **unfolding** *next-recvupd'-def*

**by** (*intro*

*exI*[*of* - *c0*(*c-msg* := (λp' q'. *if* p' = p ∧ q' = q *then* *tl* (*c-msg* c0 p q) *else*  
*c-msg* c0 p' q'),

*c-glob* := (λq'. *if* q' = q *then* *c-glob* c0 q + *hd* (*c-msg* c0 p q) *else*  
*c-glob* c0 q')]])

(*auto simp: fun-eq-iff*)

## 4.2.2 Facts about *justified*'ness

**lemma** *justified-empty*[*simp*]: *justified* {#}\_z {#}\_z

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

It's sufficient to show *justified* for least pointstamps in M.

**lemma** *justified-leastI*:

**assumes** ∀ t. 0 < *zcount* M t → (∀ t' < t. *zcount* M t' ≤ 0) → *supported-strong*  
M t ∨ (∃ t' < t. 0 < *zcount* C t') ∨ (*zcount* M t < *zcount* C t)

```

shows justified C M
unfolding justified-alt supported-strong-def
apply (intro allI impI)
apply (drule order-zmset-exists-foundation)
apply (elim exE conjE)
subgoal for t' s
  apply (drule assms(1)[unfolded supported-strong-def, rule-format])
  apply (auto intro: ccontr) []
  apply (elim disj3-split)
  apply (rule disjI1)
  using order.strict-trans2 apply blast
  apply (rule disjI2, rule disjI1)
  using order.strict-trans2 apply blast
  apply (clarsimp simp: nonpos-upto-def)
  apply (metis le-less-linear linear le-imp-less-or-eq preorder-class.le-less-trans)
done
done

```

lemma justified-add:

```

assumes justified C1 M1
  and justified C2 M2
  and  $\forall t. 0 \leq \text{zcount } C1 \ t$ 
  and  $\forall t. 0 \leq \text{zcount } C2 \ t$ 
shows justified (C1+C2) (M1+M2)
apply (rule justified-leastI)
apply (intro allI impI)
subgoal for t
  apply (cases 0 < zcount M1 t)
subgoal
  apply (drule assms(1)[unfolded justified-alt supported-strong-def, rule-format])
  apply (elim disj3-split)
subgoal
  apply (elim exE conjE)
  apply (drule order-zmset-exists-foundation-neg)
  apply (elim exE conjE)
subgoal for s s'
  apply (cases zcount (M1 + M2) s' < 0)
subgoal
  apply (rule disjI1)
  apply (auto intro!: exI[of - s'] simp: nonpos-upto-def supported-strong-def)
done
subgoal
  apply (subst (asm) not-less)
  apply (cases 0 < zcount M2 s')
  prefer 2
  subgoal by auto
subgoal
  apply (drule assms(2)[unfolded justified-alt supported-strong-def,

```

□

```

rule-format])
  apply (elim disj3-split)
  subgoal
    apply (rule disjI1)
    apply (elim exE)
    subgoal for s''
      by (auto intro!: exI[of - s''] simp: nonpos-upto-def supported-strong-def
add-nonpos-neg)
    done
  subgoal
    apply (rule disjI2, rule disjI1)
    apply (elim exE conjE)
    subgoal for s''
      using assms(3) by (auto simp: add-nonneg-pos intro!: exI[of - s''])
    done
  subgoal
    by (metis add.right-neutral add-strict-increasing2 assms(3) less-add-same-cancel1
order.strict-trans1 pos-add-strict zcount-union)
  done
done
done
done
done
subgoal
  by (metis add commute add-mono-thms-linordered-field(4) assms(4) add-0
zcount-union)
subgoal
  apply (cases supported-strong M2 t)
subgoal
  apply (rule disjI1)
  using assms(1)[unfolded justified-alt]
  apply (subst supported-strong-def)
  apply (subst (asm) supported-strong-def)
  apply (elim exE conjE)
  unfolding not-ex
  subgoal for s
    apply clarsimp
    apply (rule exI[of - s])
    apply (intro conjI)
    apply blast
    apply (rule add-nonpos-neg)
    apply (metis assms(3) le-less-linear less-trans order-class.le-less sup-
ported-strong-def)
    apply simp
    apply (clarsimp simp: nonpos-upto-def)
  done
done
subgoal
  apply (cases  $\exists t' < t. 0 < zcount C2 t'$ )
subgoal

```

```

    by (metis add-cancel-left-left assms(3) order-class.order.not-eq-order-implies-strict
zcount-union)
    subgoal
      apply (intro disjI2)
      apply clarsimp
      using assms(2)[unfolded justified-alt, rule-format, of t]
      apply (metis add.commute add-cancel-left-right add-mono-thms-linordered-field(5)
add-strict-increasing2 assms(4) nonzero-lt-gtD)
    done
  done
done
done
subgoal
  apply (cases 0 < zcount M2 t)
  prefer 2
  subgoal by auto
  subgoal
  apply (drule assms(2)[unfolded justified-alt supported-strong-def, rule-format])
  apply (elim disj3-split)
  subgoal
    apply (elim exE conjE)
    apply (drule order-zmset-exists-foundation-neg)
    apply (elim exE conjE)
    subgoal for s s'
      apply (cases zcount (M1 + M2) s' < 0)
      subgoal
        apply (rule disjI1)
      apply (auto intro!: exI[of - s'] simp: nonpos-upto-def supported-strong-def)
done
subgoal
  apply (subst (asm) not-less)
  apply (cases 0 < zcount M1 s')
  prefer 2
  subgoal by auto
  subgoal
  apply (drule assms(1)[unfolded justified-alt supported-strong-def,
rule-format])
  apply (elim disj3-split)
  subgoal
    apply (rule disjI1)
    apply (elim exE)
    subgoal for s''
      by (auto intro!: exI[of - s''] simp: nonpos-upto-def supported-strong-def
add-neg-nonpos)
  done
  subgoal
    apply (rule disjI2, rule disjI1)
    apply (elim exE conjE)

```

```

    subgoal for  $s''$ 
    using  $assms(4)$  by (auto simp: add-pos-nonneg intro!: exI[of -  $s''$ ])
    done
  subgoal
    apply (rule disjI2, rule disjI1, rule exI[of -  $s'$ ], rule conjI)
    using  $assms(4)$  by (auto intro!: add-pos-nonneg)
  done
done
done
done
subgoal
by (metis add-mono-thms-linordered-field(4)  $assms(3)$  add-0 zcount-union)
subgoal
apply (cases supported-strong M1 t)
subgoal
  apply (rule disjI1)
  apply (simp only: supported-strong-def)
  apply (elim exE)
  subgoal for  $s$ 
    apply (clarsimp simp: nonpos-upto-def intro!: exI[of -  $s$ ])
    using  $assms(2)$ [unfolded justified-alt nonpos-upto-def supported-strong-def,
rule-format, of  $s$ ]
       $assms(4)$ [rule-format, of  $s$ ]
      apply (smt (verit, best) dual-order.strict-trans)
    done
  done
  subgoal
    apply (cases  $\exists t' < t. 0 < zcount C1 t'$ )
    subgoal
      by (metis add commute add-cancel-left-left  $assms(4)$  order-class.order.not-eq-order-implies-strict
zcount-union)
    subgoal
      apply (intro disjI2)
      apply (metis add commute add-strict-increasing2  $assms(3)$  not-le
sublist-order.add-less zcount-union)
    done
  done
done
done
done
done
done
done

```

```

lemma justified-sum:
  assumes  $\forall p \in P. justified (f p) (g p)$ 
  and  $\forall p \in P. \forall t. 0 \leq zcount (f p) t$ 
  shows  $justified (\sum p \in P. f p) (\sum p \in P. g p)$ 
  using  $assms$ 
  by (induct P rule: infinite-finite-induct)

```

(*auto intro!*: *justified-add sum-nonneg simp: zcount-sum*)

**lemma** *justified-add-records*:  
 **assumes** *justified C M*  
 **and**  $\forall t. 0 \leq \text{zcount } C' t$   
 **shows** *justified (C+C') M*  
 **using** *assms unfolding justified-def*  
 **apply** (*clarsimp intro: add-pos-nonneg*)  
 **apply** (*metis add commute add-strict-increasing2 assms(2)*)  
 **done**

**lemma** *justified-add-zmset-records*:  
 **assumes** *justified C M*  
 **shows** *justified (add-zmset t C) M*  
 **using** *assms*  
 **apply** (*subst add-zmset-add-single*)  
 **apply** (*rule justified-add-records*)  
 **apply** *simp-all*  
 **done**

**lemma** *justified-diff*:  
 **assumes** *justified C M*  
 **and**  $\forall t. 0 \leq \text{zcount } C t$   
 **and**  $\forall t. \text{count } \Delta t \leq \text{zcount } C t$   
 **shows** *justified (C - zmset-of  $\Delta$ ) (M - zmset-of  $\Delta$ )*  
**proof** (*intro justified-leastI allI impI*)  
 **fix** *t*  
 **assume** *least*:  $\forall t' < t. \text{zcount } (M - \text{zmset-of } \Delta) t' \leq 0$   
 **assume**  $0 < \text{zcount } (M - \text{zmset-of } \Delta) t$   
 **then have** *Mt*:  $0 < \text{zcount } M t$   
 **by** *auto*  
 **note** *assms(1)[unfolded justified-alt, rule-format, OF Mt]*  
 **then consider** *supported-strong M t |  $\exists t' < t. 0 < \text{zcount } C t' | \text{zcount } M t < \text{zcount } C t$*   
 **by** *blast*  
 **then show** *supported-strong (M - zmset-of  $\Delta$ ) t  $\vee$  ( $\exists t' < t. 0 < \text{zcount } (C - \text{zmset-of } \Delta) t'$ )  $\vee \text{zcount } (M - \text{zmset-of } \Delta) t < \text{zcount } (C - \text{zmset-of } \Delta) t$*   
 **proof** *cases*  
 **case 1**  
 **then show** *?thesis*  
 **unfolding** *supported-strong-def*  
 **apply** (*elim exE*)  
 **subgoal for** *s*  
 **by** (*auto intro!: disjI1 exI[of - s] simp: nonpos-upto-def*)  
 **done**  
**next**  
 **case 2**  
 **then obtain** *s* **where** *s*:  $s < t \ 0 < \text{zcount } C s \ \forall s' < s. \text{zcount } C s' = 0$   
 **apply** *atomize-elim*

```

apply (elim exE conjE)
apply (drule order-zmset-exists-foundation')
apply (elim exE conjE)
subgoal for  $t' s$ 
  apply (rule exI[of - s])
  apply (intro conjI)
  apply auto [2]
  apply (intro allI impI)
  subgoal for  $s'$ 
    using assms(2)[rule-format, of s]
    apply auto
    done
  done
done
then consider
   $0 < \text{zcount } (C - \text{zmset-of } \Delta) s \mid$ 
   $\text{zcount } (C - \text{zmset-of } \Delta) s = 0 \text{ zcount } (M - \text{zmset-of } \Delta) s < 0 \mid$ 
   $\text{zcount } (C - \text{zmset-of } \Delta) s = 0 \text{ zcount } (M - \text{zmset-of } \Delta) s = 0 \mid$ 
   $\text{zcount } (C - \text{zmset-of } \Delta) s = 0 \text{ zcount } (M - \text{zmset-of } \Delta) s > 0$ 
  using assms(3)[rule-format, of s] by atomize-elim auto
then show ?thesis
proof cases
  case 1
    then show ?thesis
    using  $s$  by auto
  next
  case 2
    then show ?thesis
    using  $s$  least by (auto simp: nonpos-upto-def supported-strong-def)
  next
  case 3
    note  $\text{case3} = 3$ 
    with  $s(2)$  have  $M s: 0 < \text{zcount } M s$ 
    by  $-$  (rule ccontr, auto simp: not-less)
    note assms(1)[unfolded justified-alt, rule-format, OF Ms]
    then consider supported-strong M s  $\mid \exists t' < s. 0 < \text{zcount } C t' \mid \text{zcount } M s$ 
     $< \text{zcount } C s$ 
    using not-less by blast
    then show ?thesis
  proof cases
  case 1
    then show ?thesis
    unfolding supported-strong-def
    apply (elim exE conjE)
    subgoal for  $s'$ 
      apply (intro disjI1 exI[of - s'])
      using  $s(1,2)$  apply (auto intro: exI[of - s'] simp: nonpos-upto-def)
      done
    done

```

```

next
  case 2
  then show ?thesis
    using s(3) by auto
next
  case 3
  from case3 have zcount C s = zcount M s
    by auto
  with 3 show ?thesis
    by linarith
qed
next
  case 4
  then show ?thesis
    using least s(1) by auto
qed
next
  case 3
  then show ?thesis
    by auto
qed
qed

```

**lemma** *justified-add-msg-delta*:

```

assumes justified C M
and minting-msg C Δ
and  $\forall t. 0 \leq \text{zcount } C t$ 
shows justified C (M + timestamps (zmset-of Δ))
proof (intro allI impI justified-leastI)
  fix t
  assume t:  $0 < \text{zcount } (M + \text{timestamps } (\text{zmset-of } \Delta)) t$ 
  assume least:  $\forall t' < t. \text{zcount } (M + \text{timestamps } (\text{zmset-of } \Delta)) t' \leq 0$ 
  have Δt:  $0 < \text{zcount } (\text{timestamps } (\text{zmset-of } \Delta)) t \implies \text{supported-strong } (M + \text{timestamps } (\text{zmset-of } \Delta)) t \vee (\exists t' < t. 0 < \text{zcount } C t') \vee \text{zcount } (M + \text{timestamps } (\text{zmset-of } \Delta)) t < \text{zcount } C t$ 
    by (auto dest: pos-image-zmset-obtain-pre[rotated] assms(2)[unfolded minting-msg-def, rule-format])
  { assume Δt:  $\text{zcount } (\text{timestamps } (\text{zmset-of } \Delta)) t \leq 0$ 
    with t have Mt:  $0 < \text{zcount } M t$ 
    by auto
    note assms(1)[unfolded justified-alt, rule-format, OF Mt]
    then consider
      supported-strong M t  $\forall t' < t. \text{zcount } C t' = 0 \text{zcount } M t \geq \text{zcount } C t \mid$ 
 $\exists t' < t. 0 < \text{zcount } C t' \mid$ 
 $\text{zcount } M t < \text{zcount } C t$ 
    using not-less assms(3)
    by (metis order-class.order.not-eq-order-implies-strict)
    then have supported-strong (M + timestamps (zmset-of Δ)) t  $\vee (\exists t' < t. 0 < \text{zcount } C t') \vee \text{zcount } (M + \text{timestamps } (\text{zmset-of } \Delta)) t < \text{zcount } C t$ 

```

```

proof cases
  case 1
  then obtain s where s:  $s < t$  zcount M  $s < 0 \forall s' < s. \text{zcount } M \text{ } s' = 0$ 
    unfolding supported-strong-def
    apply atomize-elim
    apply (elim exE conjE)
    apply (drule order-zmset-exists-foundation-neg')
    apply (elim exE conjE)
    subgoal for s s'
    by (auto intro!: exI[of - s'] simp: nonpos-upto-def order-class.order-antisym)
    done
  then show ?thesis
    apply (cases  $\exists s' \leq s. \text{zcount } (\text{timestamps } (\text{zmset-of } \Delta)) \text{ } s' > 0$ )
    apply (elim exE conjE)
    subgoal for s'
      apply (drule pos-image-zmset-obtain-pre[rotated])
      apply simp
      apply (elim exE conjE)
      apply simp
      apply (drule assms(2)[unfolded minting-msg-def, rule-format])
      apply (auto simp: supported-strong-def)
      done
    subgoal
      apply (intro disjI1 exI[of - s])
      unfolding not-less
      apply (metis (full-types) le-less-linear least eq-refl order.strict-trans1
nonpos-upto-def supported-strong-def sublist-order.add-less zcount-union)
      done
    done
  next
  case 2
  then show ?thesis by auto
  next
  case 3
  then show ?thesis
    using  $\Delta t$  by auto
  qed
}
then show supported-strong ( $M + \text{timestamps } (\text{zmset-of } \Delta)$ )  $t \vee (\exists t' < t. 0 < \text{zcount } C \text{ } t') \vee \text{zcount } (M + \text{timestamps } (\text{zmset-of } \Delta)) \text{ } t < \text{zcount } C \text{ } t$ 
  apply (cases  $\text{zcount } (\text{timestamps } (\text{zmset-of } \Delta)) \text{ } t \leq 0$ )
  apply blast
  apply (rule  $\Delta t$ )
  apply auto
  done
qed

lemma justified-add-same:
  assumes justified C M

```

```

    and minting-self C Δ
    and ∀ t. 0 ≤ zcount C t
  shows justified (C + zmsset-of Δ) (M + zmsset-of Δ)
proof (intro allI impI justified-leastI)
  fix t
  assume t: 0 < zcount (M + zmsset-of Δ) t
  assume least: ∀ t' < t. zcount (M + zmsset-of Δ) t' ≤ 0
  from t consider
    0 < zcount M t |
    0 < zcount (zmsset-of Δ) t zcount M t ≤ 0
  by atomize-elim (auto simp: not-less count-inI)
  then show supported-strong (M + zmsset-of Δ) t ∨ (∃ t' < t. 0 < zcount (C +
zmsset-of Δ) t') ∨ zcount (M + zmsset-of Δ) t < zcount (C + zmsset-of Δ) t
  proof cases
    case 1
    note assms(1)[unfolded justified-alt, rule-format, OF 1]
    then consider
      supported-strong M t |
      ∃ t' < t. 0 < zcount C t' |
      zcount M t < zcount C t
    by blast
  then show ?thesis
proof cases
  case 1
  then show ?thesis
  unfolding supported-strong-def
  apply (elim exE conjE)
  subgoal for t'
    apply (cases ∃ s ≤ t'. zcount (zmsset-of Δ) s > 0)
    apply (elim exE conjE)
    subgoal for s
      apply simp
      apply (drule assms(2)[unfolded minting-self-def, rule-format])
      apply (smt (z3) order.strict-trans1 of-nat-0-le-iff)
      done
    subgoal
      apply (intro disjI1 exI[of - t'] conjI)
      apply simp
      apply simp
      apply (metis add-cancel-right-left add-mono-thms-linordered-field(1)
count-eq-zero-iff order.order-iff-strict of-nat-eq-0-iff)
      using least nonpos-upto-def apply auto
      done
    done
  done
done
next
case 2
then show ?thesis
by auto

```

```

next
  case 3
  then show ?thesis
    by auto
qed
next
case 2
then obtain t' where t': t' ≤ t 0 < zcount C t'
  using assms(2)[unfolded minting-self-def]
  by auto
then show ?thesis
  apply (cases t'=t)
  subgoal
    using 2(2) by auto
  subgoal
    apply (rule disjI2, rule disjI1)
    using assms(3) order.not-eq-order-implies-strict apply fastforce
    done
  done
qed
qed

```

#### 4.2.3 Facts about *justified-with*'ness

```

lemma justified-with-add-records:
  assumes justified-with C1 M N
  and   ∀ t. 0 ≤ zcount C2 t
  shows justified-with (C1+C2) M N
  unfolding justified-with-def
  apply (intro allI impI)
  subgoal for t
    apply (drule assms(1)[unfolded justified-with-def, rule-format])
    apply (elim disjE)
    subgoal
      by blast
    subgoal
      apply (elim exE)
      subgoal for s
        apply (rule disjI2, rule disjI1)
        using assms(2)[rule-format, of s] by auto
      done
    subgoal
      apply (intro disjI2)
      using assms(2)[rule-format, of t]
      by auto
    done
  done
done

```

```

lemma justified-with-leastI:

```

```

assumes
  ( $\forall t. 0 < \text{zcount } M t \longrightarrow (\forall t' < t. \text{zcount } M t' \leq 0) \longrightarrow$ 
   ( $\exists s < t. (\text{zcount } M s < 0 \vee \text{zcount } N s < 0) \wedge (\forall s' < s. \text{zcount } M s' \leq 0)$ )  $\vee$ 
   ( $\exists s < t. 0 < \text{zcount } C s$ )  $\vee$ 
    $\text{zcount } (M+N) t < \text{zcount } C t$ )
shows justified-with C M N
unfolding justified-with-alt
proof (intro allI impI)
  fix t
  assume t: 0 < zcount M t
  from t obtain t' where t': t' ≤ t 0 < zcount M t' ∀ s < t'. zcount M s ≤ 0
    by atomize-elim (drule order-zmset-exists-foundation^)
  note assms[rule-format, OF t'(2)]
  with t'(3) consider
     $\exists s < t'. (\text{zcount } M s < 0 \vee \text{zcount } N s < 0) \wedge (\forall s' < s. \text{zcount } M s' \leq 0) \mid$ 
     $\exists s < t'. 0 < \text{zcount } C s \mid$ 
     $\text{zcount } (M+N) t' < \text{zcount } C t'$ 
    using not-less by blast
  then show ( $\exists s < t. (\text{zcount } M s < 0 \vee \text{zcount } N s < 0) \wedge (\forall s' < s. \text{zcount } M s' \leq 0)$ )  $\vee$  ( $\exists s < t. 0 < \text{zcount } C s$ )  $\vee$   $\text{zcount } (M+N) t < \text{zcount } C t$ 
  proof cases
    case 1
    then show ?thesis
    using order.strict-trans2 t'(1) by blast
  next
    case 2
    then show ?thesis
    using order.strict-trans2 t'(1) by blast
  next
    case 3
    then consider
       $\text{zcount } (M+N) t' < 0 \mid$ 
       $\text{zcount } C t' > 0$ 
      by atomize-elim auto
    then show ?thesis
  proof cases
    case 1
    then have  $\text{zcount } N t' < 0$ 
    using t'(2) by auto
    with t'(3) show ?thesis
    apply (cases t'=t)
    subgoal
    using 3(1) by blast
    subgoal
    using t'(1) by (auto intro: exI[of - t'])
    done
  next
    case 2
    then show ?thesis

```

```

apply (cases t'=t)
subgoal
  apply (intro disjI2)
  using 3(1) apply blast
  done
subgoal
  apply (rule disjI2)
  apply (rule disjI1)
  using order.not-eq-order-implies-strict t'(1) apply blast
  done
done
qed
qed
qed

```

```

lemma justified-with-add:
  assumes justified-with C1 M N1
  and justified C1 N1
  and justified C2 N2
  and  $\forall t. 0 \leq \text{zcount } C1 \ t$ 
  and  $\forall t. 0 \leq \text{zcount } C2 \ t$ 
  shows justified-with (C1+C2) M (N1+N2)
proof (intro justified-with-leastI allI impI)
  fix t
  assume count-t:  $0 < \text{zcount } M \ t$ 
  assume least:  $\forall t' < t. \text{zcount } M \ t' \leq 0$ 
  note asms(1)[unfolded justified-with-alt, rule-format, OF count-t]
  then consider
     $\exists s < t. \text{zcount } M \ s < 0 \wedge (\forall s' < s. \text{zcount } M \ s' \leq 0) \mid$ 
     $\exists s < t. \text{zcount } N1 \ s < 0 \wedge (\forall s' < s. \text{zcount } M \ s' \leq 0) \mid$ 
     $\exists s < t. 0 < \text{zcount } C1 \ s \mid$ 
     $\text{zcount } (M + N1) \ t < \text{zcount } C1 \ t$ 
  by blast
  then show ( $\exists s < t. (\text{zcount } M \ s < 0 \vee \text{zcount } (N1 + N2) \ s < 0) \wedge (\forall s' < s. \text{zcount } M \ s' \leq 0)$ )  $\vee$ 
    ( $\exists s < t. 0 < \text{zcount } (C1 + C2) \ s \vee \text{zcount } (M + (N1 + N2)) \ t < \text{zcount } (C1 + C2) \ t$ )
  proof cases
  case 1
  then show ?thesis
  by blast
  next
  case 2
  then obtain s where  $s < t \ \text{zcount } N1 \ s < 0 \ \forall s' < s. 0 \leq \text{zcount } N1 \ s'$ 
  apply atomize-elim
  apply (elim exE conjE)
  apply (drule order-zmset-exists-foundation-neg')
  using order.strict-trans order.strict-trans1 apply blast
  done

```

```

then consider
   $zcount (N1 + N2) s < 0 \mid$ 
   $0 < zcount N2 s \wedge zcount (N1 + N2) s \geq 0$ 
  by atomize-elim auto
then show ?thesis
proof cases
  case 1
  then show ?thesis
    using least order.strict-trans s(1) by blast
next
  case 2
  note assms(3)[unfolded justified-alt, rule-format, OF 2(1)]
  then consider
     $supported-strong N2 s \forall t' < s. zcount C2 t' \leq 0 \wedge zcount N2 s \geq zcount C2 s \mid$ 
     $\exists t' < s. 0 < zcount C2 t' \mid$ 
     $zcount N2 s < zcount C2 s$ 
    using not-less by blast
  then show ?thesis
proof cases
  case 1
  then obtain s' where s': s' < s zcount N2 s' < 0 nonpos-up-to N2 s'
    unfolding supported-strong-def
    by blast
  from s'(2) have nonneg: 0 ≤ zcount (N1+N2) s' ⇒ 0 < zcount N1 s'
    by auto
  show ?thesis
    apply (cases zcount (N1 + N2) s' < 0)
    subgoal
      using least order.strict-trans s'(1) s(1) by (intro disjI1) blast
    subgoal
      unfolding not-less
      apply (drule nonneg)
      apply (drule assms(2)[unfolded justified-alt supported-strong-def,
rule-format])
      apply (elim disjE exE conjE)
      subgoal for u
        by (meson local.order.strict-trans not-less s'(1) s(3))
      subgoal for u
        by (metis add-strict-increasing assms(5) local.less-trans s'(1) s(1)
zcount-union)
      subgoal
        by (smt (verit, ccfv-threshold) assms(5) local.dual-order.strict-trans
s'(1) s(1) s(3) zcount-union)
      done
    done
  next
  case 2
  then show ?thesis
    apply (elim exE conjE)

```

```

      subgoal for  $s'$ 
        apply (rule disjI2)
        apply (rule disjI1)
        using assms(4)[rule-format, of  $s'$ ] s(1)
        apply (auto intro!: exI[of -  $s'$ ])
        done
      done
    next
      case 3
      then show ?thesis
        by (smt (verit) 2(1) assms(4) s(1) zcount-union)
    qed
  qed
next
  case 3
  then show ?thesis
    using assms(5)
    apply -
    apply (rule disjI2)
    apply (rule disjI1)
    apply (metis add-strict-increasing zcount-union)
    done
next
  case 4
  then show ?thesis
  proof (cases zcount (M + (N1 + N2)) t < zcount (C1 + C2) t)
    case True
    then show ?thesis
      by blast
  next
    case False
    then have N2t: 0 < zcount N2 t
      using 4 assms(5)[rule-format, of t]
      unfolding not-less zcount-union
      by linarith
    then show ?thesis
      using assms(3)[unfolded justified-alt supported-strong-def, rule-format, OF
N2t]
      apply (elim exE conjE disjE)
      subgoal for s
        apply (cases 0 < zcount N1 s)
        subgoal
          apply (drule assms(2)[unfolded justified-alt supported-strong-def,
rule-format])
          apply (elim exE conjE disjE)
          subgoal for  $s'$ 
            apply (rule disjI1)
            apply (rule exI[of -  $s'$ ])
            apply (intro conjI)

```

```

      apply simp
      apply (metis add-cancel-right-right add-neg-neg order.strict-implies-order
nonpos-upto-def order-class.order.not-eq-order-implies-strict zcount-union)
      apply (meson least less-trans)
      done
      subgoal for s'
      by (metis add-pos-nonneg assms(5) less-trans zcount-union)
      subgoal
      apply (rule ccontr)
      apply (clarsimp simp: not-le not-less)
      apply (metis (no-types, opaque-lifting) add-cancel-right-right add-neg-neg
assms(4) assms(5) least less-trans not-less order-class.order.not-eq-order-implies-strict
pos-add-strict)
      done
      done
      subgoal
      apply (intro disjI1 exI[of - s])
      apply (intro disjI2 conjI)
      apply simp
      apply simp
      using least apply simp
      done
      done
      subgoal for s
      by (metis add.comm-neutral add-mono-thms-linordered-field(4) assms(4)
zcount-union)
      subgoal
      using 4 by auto
      done
      qed
      qed
      qed

```

```

lemma justified-with-sum':
  assumes finite X X≠{}
    and  $\forall x \in X. \text{justified-with } (C\ x)\ M\ (N\ x)$ 
    and  $\forall x \in X. \text{justified } (C\ x)\ (N\ x)$ 
    and  $\forall x \in X. \forall t. 0 \leq \text{zcount } (C\ x)\ t$ 
  shows  $\text{justified-with } (\sum_{x \in X}. C\ x)\ M\ (\sum_{x \in X}. N\ x)$ 
  using assms
proof (induct X rule: finite-induct)
  case empty
  then show ?case by simp
next
  case (insert x F)
  show ?case
    apply (cases F={})
    subgoal
      using insert(5) by simp

```

```

subgoal
  apply (subst (1 2) sum.insert-remove)
  using insert(1) apply simp
  using insert(2) apply simp
  apply (rule justified-with-add)
  using insert(5) apply simp
  using insert(6) apply simp
  apply (rule justified-sum)
  using insert(6) apply simp
  using insert(7) apply simp
  using insert(7) apply simp
  apply (intro allI)
  unfolding zcount-sum
  apply (rule sum-nonneg)
  using insert(7) apply simp
done
done
qed

lemma justified-with-sum:
  assumes finite X X≠{}
  and  $x \in X$ 
  and justified-with ( $C\ x$ )  $M$  ( $N\ x$ )
  and  $\forall x \in X. \text{justified } (C\ x) (N\ x)$ 
  and  $\forall x \in X. \forall t. 0 \leq \text{zcount } (C\ x)\ t$ 
  shows justified-with ( $\sum x \in X. C\ x$ )  $M$  ( $\sum x \in X. N\ x$ )
  using assms
proof (induct X rule: finite-induct)
  case empty
  then show ?case
  by simp
next
  case (insert y F)
  thm insert
  show ?case
  apply (cases F={})
  subgoal
  using insert by simp
  subgoal
  apply (subst (1 2) sum.insert-remove)
  using insert(1) apply simp
  using insert(2) apply simp
  apply (cases y=x)
  subgoal
  apply (rule justified-with-add)
  using insert(6) apply simp
  using insert(7) apply simp
  apply (rule justified-sum)
  using insert(7) apply simp

```

```

    using insert(8) apply simp
    using insert(8) apply simp
    apply (intro allI)
    unfolding zcount-sum
    apply (rule sum-nonneg)
    using insert(8) apply simp
    done
  subgoal
    apply (subst (1 2) add.commute)
    apply (rule justified-with-add)
      apply (rule insert(3))
        apply simp
      using insert(5) apply simp
      using insert(6) apply simp
      using insert(7) apply simp
      using insert(8) apply simp
        apply (rule justified-sum)
      using insert(7) apply simp
      using insert(8) apply simp
      using insert(7) apply simp
      apply (intro allI)
    unfolding zcount-sum
      apply (rule sum-nonneg)
    using insert(8) apply simp
    using insert(8) apply simp
    done
  done
done
qed

lemma justified-with-add-same:
  assumes justified-with C M N
  and  $\forall t. 0 \leq \text{zcount } C \ t$ 
  shows justified-with (C + zmsset-of  $\Delta$ ) M (N + zmsset-of  $\Delta$ )
  unfolding justified-with-def
  proof (intro allI impI)
    fix t
    assume Mt:  $0 < \text{zcount } M \ t$ 
    note assms(1)[unfolded justified-with-alt, rule-format, OF Mt]
    with Mt consider
       $\exists s < t. \text{zcount } M \ s < 0 \wedge (\forall s' < s. \text{zcount } M \ s' \leq 0) \mid$ 
       $\exists s < t. \text{zcount } N \ s < 0 \wedge (\forall s' < s. \text{zcount } M \ s' \leq 0) \mid$ 
       $\exists s < t. 0 < \text{zcount } C \ s \mid$ 
       $\text{zcount } (M + N) \ t < \text{zcount } C \ t$ 
    using not-less by blast
    then show  $(\exists s < t. (\text{zcount } M \ s < 0 \vee \text{zcount } (N + \text{zmsset-of } \Delta) \ s < 0)) \vee$ 
       $(\exists s < t. 0 < \text{zcount } (C + \text{zmsset-of } \Delta) \ s) \vee \text{zcount } (M + (N + \text{zmsset-of}$ 
 $\Delta)) \ t < \text{zcount } (C + \text{zmsset-of } \Delta) \ t$ 
    proof cases

```

```

    case 1
  then show ?thesis
    by blast
next
  case 2
  then show ?thesis
    by (metis add-less-same-cancel2 assms(2) not-less preorder-class.le-less-trans
zcount-union)
next
  case 3
  then show ?thesis
    by fastforce
next
  case 4
  then show ?thesis
    by auto
qed
qed

```

**lemma** *justified-with-add-msg-delta*:

```

  assumes justified-with C M N
    and minting-msg C Δ
    and  $\forall t. 0 \leq \text{zcount } C t$ 
  shows justified-with C M (N + timestamps (zmsset-of Δ))
  unfolding justified-with-def
proof (intro allI impI)
  fix t
  assume Mt: 0 < zcount M t
  note assms(1)[unfolded justified-with-alt, rule-format, OF Mt]
  with Mt consider
     $\exists s < t. \text{zcount } M s < 0 \wedge (\forall s' < s. \text{zcount } M s' \leq 0) \mid$ 
     $\exists s < t. \text{zcount } N s < 0 \wedge (\forall s' < s. \text{zcount } M s' \leq 0) \mid$ 
     $\exists s < t. 0 < \text{zcount } C s \mid$ 
     $\text{zcount } (M + N) t < \text{zcount } C t$ 
  using not-less by blast
  then show  $(\exists s < t. (\text{zcount } M s < 0 \vee \text{zcount } (N + \text{timestamps } (\text{zmsset-of } \Delta))$ 
s < 0))  $\vee$ 
     $(\exists s < t. 0 < \text{zcount } C s) \vee \text{zcount } (M + (N + \text{timestamps } (\text{zmsset-of } \Delta)))$ 
t < zcount C t
proof cases
  case 1
  then show ?thesis
    by blast
next
  case 2
  then obtain s where s: s < t zcount N s < 0
    by blast
  then show ?thesis
proof (cases  $\exists (p, s') \in \# \Delta. s' \leq s$ )

```

```

case True
then show ?thesis
  apply  $-$ 
  apply clarsimp
  apply (drule assms(2)[unfolded minting-msg-def, rule-format])
  using order.strict-trans order.strict-trans1 s(1) not-less apply blast
  done
next
  case False
  then have zcount (timestamps (zmsset-of  $\Delta$ )) s = 0
    by (force intro: ccontr dest: pos-image-zmsset-obtain-pre[rotated])
  then show ?thesis
    by (metis plus-int-code(1) s(1,2) zcount-union)
  qed
next
  case 3
  then show ?thesis
    by blast
  next
  case 4
  then show ?thesis
    apply (cases zcount (timestamps (zmsset-of  $\Delta$ )) t > 0)
    apply (auto dest: pos-image-zmsset-obtain-pre[rotated] assms(2)[unfolded
minting-msg-def, rule-format]) []
    unfolding not-less apply auto
    done
  qed
qed

```

**lemma** *justified-with-diff*:

```

assumes justified-with C M N
  and  $\forall t. 0 \leq \text{zcount } C \ t$ 
  and  $\forall t. \text{count } \Delta \ t \leq \text{zcount } C \ t$ 
  and justified C N
shows justified-with (C - zmsset-of  $\Delta$ ) M (N - zmsset-of  $\Delta$ )
proof (intro allI impI justified-with-leastI)
  fix t
  assume Mt: 0 < zcount M t
  assume least:  $\forall t' < t. \text{zcount } M \ t' \leq 0$ 
  note assms(1)[unfolded justified-with-alt, rule-format, OF Mt]
  with Mt consider
     $\exists s < t. \text{zcount } M \ s < 0 \wedge (\forall s' < s. \text{zcount } M \ s' \leq 0) \mid$ 
     $\exists s < t. \text{zcount } N \ s < 0 \wedge (\forall s' < s. \text{zcount } M \ s' \leq 0) \mid$ 
     $\exists s < t. 0 < \text{zcount } C \ s \ \forall s' < t. \text{zcount } M \ s \geq 0 \vee \neg (\forall s' < s. \text{zcount } M \ s' \leq 0)$ 
 $\forall s < t. \text{zcount } N \ s \geq 0 \vee \neg (\forall s' < s. \text{zcount } M \ s' \leq 0) \ \text{zcount } (M + N) \ t \geq \text{zcount}$ 
 $C \ t \mid$ 
     $\text{zcount } (M + N) \ t < \text{zcount } C \ t$ 
  using not-less by blast
  then show  $(\exists s < t. (\text{zcount } M \ s < 0 \vee \text{zcount } (N - \text{zmsset-of } \Delta) \ s < 0) \wedge$ 

```

```

(∀ s' < s. zcount M s' ≤ 0) ∨
  (∃ s < t. 0 < zcount (C - zmsset-of Δ) s) ∨ zcount (M + (N - zmsset-of
Δ)) t < zcount (C - zmsset-of Δ) t
proof cases
  case 1
  then show ?thesis
  by blast
next
  case 2
  then show ?thesis
  using diff-add-cancel zcount-union zcount-zmsset-of-nonneg by auto
next
  case 3
  then obtain s where s: s < t 0 < zcount C s ∀ s' < s. zcount C s' = 0
  apply atomize-elim
  apply (elim exE conjE)
  apply (drule order-zmsset-exists-foundation')
  apply (elim exE conjE)
  subgoal for t' s
  apply (rule exI[of - s])
  apply (intro conjI)
  apply auto [2]
  apply (intro allI impI)
  subgoal for s'
  using assms(2)[rule-format, of s'] by auto
  done
  done
then consider
  0 < zcount (C - zmsset-of Δ) s |
  zcount (C - zmsset-of Δ) s = 0 zcount (N - zmsset-of Δ) s < 0 |
  zcount (C - zmsset-of Δ) s = 0 zcount (N - zmsset-of Δ) s = 0 |
  zcount (C - zmsset-of Δ) s = 0 zcount (N - zmsset-of Δ) s > 0
  using assms(3)[rule-format, of s] by atomize-elim auto
then show ?thesis
proof cases
  case 1
  then show ?thesis
  using s by auto
next
  case 2
  then show ?thesis
  using s least by (auto simp: nonpos-upto-def)
next
  case 3
  note case3 = 3
  with s(2) have Ns: 0 < zcount N s
  by (auto intro: ccontr simp: not-less)
  note assms(4)[unfolded justified-alt, rule-format, OF Ns]
  then consider supported-strong N s | ∃ t' < s. 0 < zcount C t' | zcount N s <

```

```

zcount C s
  using not-less by blast
then show ?thesis
proof cases
  case 1
  then show ?thesis
    unfolding supported-strong-def
    apply (elim exE conjE)
    subgoal for s'
      using s(1,2) least by (auto intro: exI[of - s'] simp: nonpos-upto-def)
    done
next
  case 2
  then show ?thesis
    using s(3) by auto
next
  case 3
  from case3 have zcount C s = zcount N s
    by auto
  with 3 show ?thesis
    by linarith
qed
next
  case 4
  then have Ns: 0 < zcount N s
    by auto
  note assms(4)[unfolded justified-alt, rule-format, OF Ns]
  then consider supported-strong N s |  $\exists t' < s. 0 < zcount C t' | zcount N s <$ 
zcount C s
  using not-less by blast
then show ?thesis
proof cases
  case 1
  then show ?thesis
    unfolding supported-strong-def
    apply (elim exE conjE)
    subgoal for s'
      using s(1,2) least by (auto intro: exI[of - s'] simp: nonpos-upto-def)
    done
next
  case 2
  then show ?thesis
    using s(3) by auto
next
  case 3
  have zcount C s = zcount N s
    using 3 4(1,2) by auto
  with 3 show ?thesis
    by linarith

```

```

      qed
    qed
  next
  case 4
  then show ?thesis
    by auto
  qed
qed

```

```

lemma PositiveImplies-justified-with:
  assumes justified C (M+N)
    and PositiveImplies M (M+N)
  shows justified-with C M N
  unfolding justified-with-def
  apply (intro allI impI)
  apply (drule assms(2)[unfolded PositiveImplies-def, rule-format])
  apply (frule assms(1)[unfolded justified-alt supported-strong-def, rule-format])
  apply (elim disjE)
  subgoal for t
    apply (intro disjI1)
    apply (elim exE)
    subgoal for s
      apply (clarsimp intro!: exI[of - s])
    done
  done
  subgoal for t
    using less-imp-le by blast
  subgoal for t
    by (intro disjI2 exI[of - t]) auto
  done

```

```

lemma justified-with-add-zmset[simp]:
  assumes justified-with C M N
  shows justified-with (add-zmset c C) M N
  using assms
  apply (subst add-zmset-add-single)
  apply (rule justified-with-add-records)
  apply simp-all
  done

```

```

lemma next-performop'-preserves-justified-with:
  assumes justified-with (c-caps c0 p) M N
    and next-performop' c0 c1 p Δneg Δmint-msg Δmint-self
    and  $\forall t. 0 \leq \text{zcount } (c\text{-caps } c0 \text{ } p) \text{ } t$ 
    and justified (c-caps c0 p) N
  shows justified-with (c-caps c0 p + zmset-of Δmint-self - zmset-of Δneg) M
  (N + zmset-of Δmint-self + timestamps (zmset-of Δmint-msg) - zmset-of Δneg)
  apply (rule justified-with-diff)
  apply (rule justified-with-add-msg-delta)

```

```

    apply (rule justified-with-add-same)
  using assms(1) apply simp
  using assms(3) apply simp
    apply (rule minting-msg-add-records)
  using next-performopD(4)[OF assms(2)] apply simp
    apply simp
  using assms(3) apply simp
  using assms(3) apply simp
  using next-performopD(2)[OF assms(2)] apply (simp add: add.commute add-increasing)
  apply (rule justified-add-msg-delta)
    apply (rule justified-add-same)
  using assms(4) apply simp
  using next-performopD(3)[OF assms(2)] apply simp
  using assms(3) apply simp
    apply (rule minting-msg-add-records)
  using next-performopD(4)[OF assms(2)] apply simp
    apply simp
  using assms(3) apply (simp add: add.commute add-increasing)
done

```

## 4.3 Invariants

### 4.3.1 InvRecordCount

InvRecordCount states that for every processor, its local approximation  $c\text{-glob } c \ q$  and the sum of all incoming progress updates  $GlobalIncomingInfoAt \ c \ q$  together are equal to the sum of all capabilities in the system.

**definition** *InvRecordCount* **where**

$$InvRecordCount \ c \equiv \forall q. \ records \ c = GlobalIncomingInfoAt \ c \ q + c\text{-glob } c \ q$$

**lemma** *init-config-implies-InvRecordCount*:  $init\text{-config } c \implies InvRecordCount \ c$

**by** (*simp add: InvRecordCount-def init-config-def GlobalIncomingInfo-def IncomingInfo-def*)

**lemma** *performop-preserves-InvRecordCount*:

**assumes** *InvRecordCount*  $c0$

**and**  $next\text{-performop}' \ c0 \ c1 \ p \ \Delta neg \ \Delta mint\text{-msg} \ \Delta mint\text{-self}$

**shows** *InvRecordCount*  $c1$

**proof** –

**note**  $change = next\text{-performop}D[OF \ assms(2)]$

**note**  $complex\text{-change} = next\text{-performop-complex}D[OF \ assms(2)]$

**show** *InvRecordCount*  $c1$

**unfolding** *InvRecordCount-def complex-change*

**by** (*auto intro: assms(1)[unfolded InvRecordCount-def, rule-format] simp: change*)

**qed**

**lemma** *sendupd-preserves-InvRecordCount*:

**assumes** *InvRecordCount*  $c0$

**and**  $next\text{-sendupd}' \ c0 \ c1 \ p \ tt$

**shows**  $InvRecordCount\ c1$   
**proof** –  
**note**  $change = next-sendupdD[OF\ assms(2)]$   
**note**  $complex-change = next-sendupd-complexD[OF\ assms(2)]$   
**from**  $assms(1)$  **show**  $InvRecordCount\ c1$   
**unfolding**  $InvRecordCount-def\ complex-change\ change(5)$  **by**  $assumption$   
**qed**

**lemma**  $recvupd-preserves-InvRecordCount$ :  
**assumes**  $InvRecordCount\ c0$   
**and**  $next-recvupd'\ c0\ c1\ p\ q$   
**shows**  $InvRecordCount\ c1$   
**proof** –  
**note**  $change = next-recvupdD[OF\ assms(2)]$   
**note**  $complex-change = next-recvupd-complexD[OF\ assms(2)]$   
**show**  $InvRecordCount\ c1$   
**unfolding**  $InvRecordCount-def\ complex-change\ change(4)$   
**by**  $(auto\ simp:\ assms(1)[unfolded\ InvRecordCount-def,\ rule-format])$   
**qed**

**lemma**  $recvcap-preserves-InvRecordCount$ :  
**assumes**  $InvRecordCount\ c0$   
**and**  $next-recvcap'\ c0\ c1\ p\ t$   
**shows**  $InvRecordCount\ c1$   
**proof** –  
**note**  $change = next-recvcapD[OF\ assms(2)]$   
**note**  $complex-change = next-recvcap-complexD[OF\ assms(2)]$   
**show**  $InvRecordCount\ c1$   
**unfolding**  $InvRecordCount-def\ complex-change\ change(4)$   
**by**  $(auto\ simp:\ assms(1)[unfolded\ InvRecordCount-def,\ rule-format])$   
**qed**

**lemma**  $next-preserves-InvRecordCount$ :  $InvRecordCount\ c0 \implies next'\ c0\ c1 \implies$   
 $InvRecordCount\ c1$   
**unfolding**  $next'-def$   
**apply**  $(elim\ disjE)$   
**subgoal**  
**using**  $performop-preserves-InvRecordCount$  **by**  $auto$   
**subgoal**  
**using**  $sendupd-preserves-InvRecordCount$  **by**  $auto$   
**subgoal**  
**using**  $recvupd-preserves-InvRecordCount$  **by**  $auto$   
**subgoal**  
**using**  $recvcap-preserves-InvRecordCount$  **by**  $auto$   
**subgoal**  
**by**  $simp$   
**done**

**lemma**  $alw-InvRecordCount$ :  $spec\ s \implies alw\ (holds\ InvRecordCount)\ s$

**using** *lift-invariant-to-spec init-config-implies-InvRecordCount next-preserves-InvRecordCount*  
**by** (*metis (mono-tags, lifting) holds.elims(2) holds.elims(3) nrt.simps*)

### 4.3.2 InvCapsNonneg and InvRecordsNonneg

InvCapsNonneg states that elements in a processor's *c-caps c p* always have non-negative cardinality. InvRecordsNonneg lifts this result to *records c*

**definition** *InvCapsNonneg* :: ('p :: finite, 'a) configuration  $\Rightarrow$  bool **where**  
*InvCapsNonneg c* = ( $\forall p t. 0 \leq \text{zcount } (c\text{-caps } c p) t$ )

**definition** *InvRecordsNonneg* **where**  
*InvRecordsNonneg c* = ( $\forall t. 0 \leq \text{zcount } (records c) t$ )

**lemma** *init-config-implies-InvCapsNonneg*: *init-config c*  $\implies$  *InvCapsNonneg c*  
**unfolding** *init-config-def InvCapsNonneg-def* **by** *simp*

**lemma** *performop-preserves-InvCapsNonneg*:  
**assumes** *InvCapsNonneg c0*  
**and** *next-performop' c0 c1 p  $\Delta_m \Delta_{p1} \Delta_{p2}$*   
**shows** *InvCapsNonneg c1*  
**using** *assms unfolding InvCapsNonneg-def next-performop'-def Let-def*  
**by** *clarsimp (metis add.right-neutral add-mono-thms-linordered-semiring(1) of-nat-0-le-iff)*

**lemma** *sendupd-performs-InvCapsNonneg*:  
**assumes** *InvCapsNonneg c0*  
**and** *next-sendupd' c0 c1 p tt*  
**shows** *InvCapsNonneg c1*  
**using** *assms by (simp add: InvCapsNonneg-def next-sendupd'-def Let-def)*

**lemma** *recvupd-preserves-InvCapsNonneg*:  
**assumes** *InvCapsNonneg c0*  
**and** *next-recvupd' c0 c1 p q*  
**shows** *InvCapsNonneg c1*  
**using** *assms unfolding InvCapsNonneg-def next-recvupd'-def*  
**by** *simp*

**lemma** *recvcap-preserves-InvCapsNonneg*:  
**assumes** *InvCapsNonneg c0*  
**and** *next-recvcap' c0 c1 p t*  
**shows** *InvCapsNonneg c1*  
**using** *assms unfolding InvCapsNonneg-def next-recvcap'-def*  
**by** *simp*

**lemma** *next-preserves-InvCapsNonneg*: *holds InvCapsNonneg s*  $\implies$  *next s*  $\implies$  *nrt*  
(*holds InvCapsNonneg*) *s*  
**unfolding** *next'-def*  
**apply** (*elim disjE*)  
**subgoal**  
**using** *performop-preserves-InvCapsNonneg* **by** *auto*

```

subgoal
  using sendupd-performs-InvCapsNonneg by auto
subgoal
  using recvupd-preserves-InvCapsNonneg by auto
subgoal
  using recvcap-preserves-InvCapsNonneg by auto
subgoal
  by simp
done

```

```

lemma alw-InvCapsNonneg: spec s  $\implies$  alw (holds InvCapsNonneg) s
  using lift-invariant-to-spec next-preserves-InvCapsNonneg init-config-implies-InvCapsNonneg
  by blast

```

```

lemma alw-InvRecordsNonneg: spec s  $\implies$  alw (holds InvRecordsNonneg) s
  apply (rule alw-mp[where  $\varphi = \text{holds } \text{InvCapsNonneg}$ ])
  using alw-InvCapsNonneg apply assumption
  apply (rule all-imp-alw)
  unfolding InvCapsNonneg-def InvRecordsNonneg-def records-def
  apply (auto intro!: add-nonneg-nonneg sum-nonneg simp: zcount-sum)
  done

```

### 4.3.3 Resulting lemmas

```

lemma pos-caps-pos-records:
  assumes InvCapsNonneg c
  shows  $0 < \text{zcount } (c\text{-caps } c \ p) \ x \implies 0 < \text{zcount } (\text{records } c) \ x$ 
proof –
  fix x
  assume  $0 < \text{zcount } (c\text{-caps } c \ p) \ x$ 
  then have  $0 < \text{zcount } (\sum_{p \in \text{UNIV}. c\text{-caps } c \ p) \ x$ 
    using assms(1)[unfolded InvCapsNonneg-def]
    by (auto intro!: sum-pos2 simp: zcount-sum)
  then show  $0 < \text{zcount } (\text{records } c) \ x$ 
    unfolding records-def by simp
qed

```

### 4.3.4 SafeRecordsMono

The records in the system are monotonic, i.e. once *records c* contains no records up to some timestamp *t*, then it will stay that way forever.

**definition** *SafeRecordsMono* ::  $(\text{'p} :: \text{finite}, \text{'a}) \text{ computation} \Rightarrow \text{bool}$  **where**  
*SafeRecordsMono s* =  $(\forall t. \text{RecordsVacantUpto } (\text{shd } s) \ t \longrightarrow \text{alw } (\text{holds } (\lambda c. \text{RecordsVacantUpto } c \ t)) \ s)$

```

lemma performop-preserves-RecordsVacantUpto:
  assumes RecordsVacantUpto c0 t
  and next-performop' c0 c1 p  $\Delta$ neg  $\Delta$ mint-msg  $\Delta$ mint-self
  and InvRecordsNonneg c1

```

```

    and InvCapsNonneg c0
  shows RecordsVacantUpto c1 t
proof -
  note InvRecordsNonneg = assms(3)[rule-format]
  { fix s
    let ?Δpos = timestamps (zmsset-of Δmint-msg) + zmsset-of Δmint-self
    let ?Δ = ?Δpos - zmsset-of Δneg
    note change = next-performopD[OF assms(2)]
    note complex-change = next-performop-complexD[OF assms(2)]
    assume s: s ≤ t zcount (records c1) s ≠ 0
    then have s-pos: 0 < zcount (records c1) s
      using InvRecordsNonneg
      by (simp add: order-class.order.not-eq-order-implies-strict InvRecordsNon-
neg-def)
    have Δ-in-nrec: 0 < zcount ?Δ t ⇒ ∃ t' ≤ t. 0 < zcount (records c0) t' for t
      apply (subst (asm) zcount-diff)
      apply (subst (asm) zcount-union)
      apply (drule zero-lt-diff)
      apply simp
      apply (drule zero-lt-add-disj)
      apply simp
      apply simp
      apply (erule disjE)
    subgoal
      apply (drule pos-image-zmsset-obtain-pre[rotated])
      apply (auto dest!: change(4)[unfolded minting-msg-def, rule-format]
pos-caps-pos-records[OF assms(4)] less-imp-le)
    done
    subgoal
      by (auto dest!: change(3)[unfolded minting-self-def, rule-format] pos-caps-pos-records[OF
assms(4)])
    done
    have nrec0s: zcount (records c0) s = 0
      by (rule assms(1)[unfolded vacant-upto-def, rule-format, OF s(1)])
    have zcount (records c1) s ≤ 0
      unfolding complex-change
      apply (subst zcount-union)
      apply (subst nrec0s)
      apply (subst add-0)
      apply (rule ccontr)
      unfolding not-le
      apply (drule Δ-in-nrec[of s])
      apply (meson assms(1) order-trans pos-zcount-in-zmsset s(1) vacant-upto-def
zcount-ne-zero-iff)
    done
    with s-pos have False
      by linarith
  }
  note r = this

```

```

from assms show ?thesis
  unfolding next-performop'-def Let-def vacant-upto-def
  apply clarify
  apply (rule ccontr)
  apply (rule r)
  apply auto
  done
qed

lemma next'-preserves-RecordsVacantUpto:
  fixes c0 :: ('p::finite, 'a) configuration
  shows InvCapsNonneg c0  $\implies$  InvRecordsNonneg c1  $\implies$  RecordsVacantUpto c0
t  $\implies$  next' c0 c1  $\implies$  RecordsVacantUpto c1 t
  unfolding next'-def
  apply (elim disjE)
  subgoal
    by (auto intro: performop-preserves-RecordsVacantUpto)
  subgoal
    by (auto simp: next-sendupd'-def Let-def records-def)
  subgoal
    by (auto simp: next-recvupd'-def records-def)
  subgoal
    by (auto dest: next-recvcap-complexD)
  subgoal
    by simp
  done

lemma alw-next-implies-alw-SafeRecordsMono:
  alw next s  $\implies$  alw (holds InvCapsNonneg) s  $\implies$  alw (holds InvRecordsNonneg)
s  $\implies$  alw SafeRecordsMono s
  apply (coinduction arbitrary: s)
  subgoal for s
    unfolding spec-def next'-def SafeRecordsMono-def Let-def
    apply (rule exI[of - s])
    apply safe
    subgoal for t
      apply (coinduction arbitrary: s rule: alw.coinduct)
      apply clarsimp
      apply (rule conjI)
      apply blast
      apply (erule alw.cases)
      apply clarsimp
      apply (metis (no-types, lifting) next'-def next'-preserves-RecordsVacantUpto
alw-holds2)
    done
    apply blast
  done
done

```

**lemma** *alw-SafeRecordsMono*:  $\text{spec } s \implies \text{alw SafeRecordsMono } s$

by (*auto intro!*: *alw-next-implies-alw-SafeRecordsMono* *alw-InvRecordsNonneg* *alw-InvCapsNonneg simp*: *spec-def*)

#### 4.3.5 InvJustifiedII and InvJustifiedGII

These two invariants state that any net-positive change in the sum of incoming progress updates is "justified" by one of several statements being true.

**definition** *InvJustifiedII* where

$\text{InvJustifiedII } c = (\forall k p q. \text{justified } (c\text{-caps } c p) (\text{IncomingInfo } c k p q))$

**definition** *InvJustifiedGII* where

$\text{InvJustifiedGII } c = (\forall k p q. \text{justified } (\text{records } c) (\text{GlobalIncomingInfo } c k p q))$

Given some *zmset*  $M$  justified wrt to *caps*  $c0 p$ , after a *performop*  $M + \Delta$  is justified wrt to *c-caps*  $c1 p$ . This lemma captures the identical argument used for preservation of *InvTempJustified* and *InvJustifiedII*.

**lemma** *next-performop'-preserves-justified*:

**assumes** *justified* (*c-caps*  $c0 p$ )  $M$

**and** *next-performop'*  $c0 c1 p \Delta_{\text{neg}} \Delta_{\text{mint-msg}} \Delta_{\text{mint-self}}$

**and** *InvCapsNonneg*  $c0$

**shows** *justified* (*c-caps*  $c1 p$ ) ( $M + (\text{timestamps } (\text{zmset-of } \Delta_{\text{mint-msg}}) + \text{zmset-of } \Delta_{\text{mint-self}} - \text{zmset-of } \Delta_{\text{neg}}))$

**proof** –

**let**  $?\Delta_{\text{pos}} = \text{timestamps } (\text{zmset-of } \Delta_{\text{mint-msg}}) + \text{zmset-of } \Delta_{\text{mint-self}}$

**let**  $?\Delta = ?\Delta_{\text{pos}} - \text{zmset-of } \Delta_{\text{neg}}$

**let**  $?M1 = M + ?\Delta$

**note** *change* = *next-performopD*[*OF assms*(2)]

**note** *complex-change* = *next-performop-complexD*[*OF assms*(2)]

**note** *inv0* = *assms*(1)[*unfolded InvJustifiedII-def justified-alt, rule-format*]

{ **fix**  $k q t$

**assume**  $t: 0 < \text{zcount } ?M1 t$

**assume** *least*:  $\forall t' < t. \text{zcount } ?M1 t' \leq 0$

**from**  $t$  **consider**  $0 < \text{zcount } M t \mid \text{zcount } M t \leq 0 \ 0 < \text{zcount } ?\Delta t$

**by** *atomize-elim* (*auto simp*: *complex-change*)

**then have** *supported-strong*  $?M1 t \vee (\exists t' < t. 0 < \text{zcount } (c\text{-caps } c1 p) t') \vee \text{zcount } ?M1 t < \text{zcount } (c\text{-caps } c1 p) t$

**proof** *cases*

**case** 1 —  $M$  was already positive at  $t$  in  $c0$

**note**  $M_{\text{count}} = 1$

**note** *assms*(1)[*unfolded InvJustifiedII-def justified-alt, rule-format, OF Mcount*]

**then consider**

*supported-strong*  $M t \mid$

$\neg \text{supported-strong } M t \exists t' < t. 0 < \text{zcount } (c\text{-caps } c0 p) t' \mid$

$\forall t' < t. \text{zcount } (c\text{-caps } c0 p) t' \leq 0 \ \text{zcount } M t < \text{zcount } (c\text{-caps } c0 p) t$

**by** *atomize-elim auto*

**then show** *?thesis*

```

proof cases
  case 1
    { assume nosupp:  $\neg$  supported-strong ?M1 t
      assume  $\forall t' < t. \neg 0 < \text{zcount } (c\text{-caps } c1 \text{ } p) \ t'$ 
      then have nocaps:  $\forall t' < t. \text{zcount } (c\text{-caps } c1 \text{ } p) \ t' = 0$ 
        using InvCapsNonneg-def assms(2) assms(3) order-class.le-less performop-preserves-InvCapsNonneg by fastforce
        from 1 obtain s where  $s: s < t \text{ zcount } M \ s < 0 \wedge s'. s' < s \implies \text{zcount } M \ s' = 0$ 
          unfolding nonpos-upto-def supported-strong-def
          apply atomize-elim
          apply (elim exE conjE)
          apply (drule order-zmset-exists-foundation-neg)
          apply (elim exE)
          subgoal for - s
            apply (rule exI[of - s])
            apply (rule conjI)
            using le-less-trans apply blast
          using less-imp-le order-trans order-class.order.not-eq-order-implies-strict
apply blast
        done
        done
        have count1s:  $0 \leq \text{zcount } ?M1 \ s$ 
          apply (rule ccontr)
          apply (subst (asm) not-le)
          using nosupp[unfolded nonpos-upto-def supported-strong-def, simplified, rule-format]
          using least order.strict-trans2 s(1) apply fastforce
          done
          have  $\Delta inc: 0 < \text{zcount } ?\Delta \ s$ 
            using complex-change(3) count1s s(2) by auto
          have  $0 < \text{zcount } (\text{timestamps } (\text{zmset-of } \Delta \text{mint-msg})) \ s$ 
            using  $\Delta inc$  change(2)[rule-format, of s] nocaps[rule-format, OF s(1)]
            unfolding change(9) fun-upd-same zcount-union zcount-diff
            by linarith
          then obtain u where  $u: u < s \ 0 < \text{zcount } (c\text{-caps } c0 \text{ } p) \ u \ \forall u' < u. \text{zcount } (c\text{-caps } c0 \text{ } p) \ u' = 0$ 
            apply atomize-elim
            apply (drule pos-image-zmset-obtain-pre[rotated])
            apply simp
            apply clarify
            subgoal for p'
              apply simp
              apply (drule change(4)[unfolded minting-msg-def, rule-format])
              apply simp
              apply (elim exE conjE)
              apply (drule order-zmset-exists-foundation)
              apply clarsimp
              subgoal for s' u

```

```

      apply (rule exI[of - u])
      apply clarsimp
      using assms(3)[unfolded InvCapsNonneg-def, rule-format]
      apply (metis le-less-trans order-class.order.not-eq-order-implies-strict)
      done
    done
  done
  have count1u: zcount ?M1 u < 0
    using complex-change(4)[of u] nocaps[unfolded change(9) fun-upd-same]
    order.strict-trans[OF u(1) s(1)] s(3)[OF u(1)] u(2) u(3)
    by auto
  then have nonpos-upto ?M1 u
    unfolding nonpos-upto-def
    using least order.strict-implies-order order.strict-trans1 s(1) u(1) by
blast
  then have supported-strong ?M1 t
    using count1u order.strict-trans s(1) u(1) supported-strong-def by blast
  with nosupp have False
    by blast
}
then show ?thesis
  by blast
next
case 2
{ assume nosupp: ¬ supported-strong ?M1 t
  assume ∀ t' < t. ¬ 0 < zcount (c-caps c1 p) t'
  then have nocaps: ∀ t' < t. zcount (c-caps c1 p) t' = 0
    using InvCapsNonneg-def assms(2) assms(3) order-class.le-less perfor-
mop-preserves-InvCapsNonneg by fastforce
  from 2(2) obtain s where s: s < t 0 < zcount (c-caps c0 p) s ∀ s' < s.
zcount (c-caps c0 p) s' = 0
  apply atomize-elim
  apply (elim exE conjE)
  apply (drule order-zmset-exists-foundation)
  apply (elim exE conjE)
  subgoal for - s
    apply (rule exI[of - s])
    apply (rule conjI, simp)
    apply (rule conjI, simp)
    apply (intro allI impI)
    apply (rule ccontr)
  subgoal
    using assms(3)[unfolded InvCapsNonneg-def, rule-format]
    apply auto
    by (metis order-class.le-less zcount-ne-zero-iff)
  done
done
have Δcounts:
  ∧ s. s < t ⇒ count Δneg s = zcount (c-caps c0 p) s

```

```

 $\bigwedge s. s < t \implies \text{count } \Delta \text{mint-self } s = 0$ 
 $\bigwedge p s'. s' \leq s \implies \text{count } \Delta \text{mint-msg } (p, s') = 0$ 
subgoal for  $s'$ 
using  $\text{change}(2)$   $\text{nocaps } s(1)$   $\text{order-class.order.not-eq-order-implies-strict}$ 
by  $(\text{fastforce simp: change}(9))$ 
subgoal for  $s'$ 
using  $\text{nocaps}[\text{rule-format, of } s']$ 
by  $(\text{simp add: change}(9) \langle \bigwedge s'. s' < t \implies \text{int } (\text{count } \Delta \text{neg } s') = \text{zcount}$ 
 $(\text{c-caps } c0 p) s' \rangle)$ 
subgoal for  $p s'$ 
using  $\text{change}(4)[\text{unfolded minting-msg-def, rule-format, of } (p, s')]$   $s(3)$ 
by  $(\text{force intro: ccontr})$ 
done
have  $\text{caps-le-ii0: zcount } (c\text{-caps } c0 p) s \leq \text{zcount } M s$ 
proof  $(\text{rule ccontr})$ 
assume  $\text{nle: } \neg \text{zcount } (c\text{-caps } c0 p) s \leq \text{zcount } M s$ 
have  $\text{zcount } ?M1 s < 0$ 
unfolding  $\text{complex-change}(3)$ 
using  $\text{complex-change}(4)$   $\text{nle } s(3)$  by  $(\text{auto simp: } \Delta \text{counts}(1,2)[OF$ 
 $s(1)])$ 
then show  $\text{False}$ 
using  $s(1)$   $\text{least}$ 
by  $(\text{force dest!: nosupp}[\text{unfolded supported-strong-def, simplified,}$ 
 $\text{rule-format}] \text{simp: nonpos-upto-def})$ 
qed
with  $s(2)$  have  $\text{count0s: } 0 < \text{zcount } M s$ 
by  $\text{auto}$ 
have  $\text{False}$ 
using  $\text{inv0}[OF \text{count0s}]$ 
apply  $(\text{elim disj3-split})$ 
using  $2(1)$   $\text{order.strict-trans } s(1)$   $\text{supported-strong-def}$  apply  $\text{blast}$ 
using  $s(3)$  apply  $\text{auto []}$ 
using  $\text{caps-le-ii0}$  apply  $\text{linarith}$ 
done
}
then show  $?thesis$ 
by  $\text{blast}$ 
next
case  $3$ 
then show  $?thesis$ 
apply  $(\text{intro disjI2})$ 
unfolding  $\text{complex-change}(3)$   $\text{change}(9)$ 
apply  $(\text{simp only: if-P zcount-union zcount-diff})$ 
apply  $(\text{subst complex-change}(4)[\text{of } t])$ 
using  $\text{assms}(3)[\text{unfolded InvCapsNonneg-def, rule-format}]$ 
apply  $(\text{simp-all add: antisym})$ 
done
qed
next — Adding  $\Delta$  made  $M$  positive at  $t$  in  $c1$ 

```

```

case 2
{ assume nosupp:  $\neg$  supported-strong ?M1 t
  assume  $\forall t' < t. \neg 0 < \text{zcount} (c\text{-caps } c1\ p) t'$ 
  then have nocaps:  $\forall t' < t. \text{zcount} (c\text{-caps } c1\ p) t' = 0$ 
    using InvCapsNonneg-def assms(2) assms(3) order-class.le-less performop-preserves-InvCapsNonneg by metis
  assume  $\neg \text{zcount} ?M1\ t < \text{zcount} (c\text{-caps } c1\ p) t$ 
  then have caps-le:  $\text{zcount} (c\text{-caps } c1\ p) t \leq \text{zcount} ?M1\ t$ 
    by linarith
  from 2 have count  $\Delta_{neg}\ t < \text{zcount} ?\Delta_{pos}\ t$ 
    by auto
  then have  $0 < \text{count } \Delta_{mint\text{-self}}\ t \vee 0 < \text{zcount} (\text{timestamps } (\text{zmset-of } \Delta_{mint\text{-msg}}))\ t$ 
    by (metis 2(2) add.commute add.left-neutral not-gr-zero of-nat-0 zero-lt-diff zcount-diff zcount-of-mset zcount-union zcount-zmset-of-nonneg)
  then obtain s where s:  $s \leq t$   $0 < \text{zcount} (c\text{-caps } c0\ p) s \forall s' < s. \text{zcount} (c\text{-caps } c0\ p) s' = 0$ 
    apply atomize-elim
    apply (elim disjE)
  subgoal
    apply simp
    apply (drule change(3)[unfolded minting-self-def, rule-format])
    apply (elim exE conjE)
    apply (drule order-zmset-exists-foundation)
    apply clarsimp
    subgoal for s' u
      apply (rule exI[of - u])
      apply clarsimp
      using assms(3)[unfolded InvCapsNonneg-def, rule-format]
      apply (metis order.strict order-class.order.not-eq-order-implies-strict)
    done
  done
subgoal
  apply (drule pos-image-zmset-obtain-pre[rotated])
  apply simp
  apply clarify
  subgoal for p'
    apply simp
    apply (drule change(4)[unfolded minting-msg-def, rule-format])
    apply simp
    apply (elim exE conjE)
    apply (drule order-zmset-exists-foundation)
    apply clarsimp
    subgoal for s' u
      apply (rule exI[of - u])
      apply clarsimp
      using assms(3)[unfolded InvCapsNonneg-def, rule-format]
    apply (metis order.strict-trans1 less-imp-le order-class.order.not-eq-order-implies-strict)
  done

```

```

done
done
done
have  $\Delta$ counts:
 $\wedge s. s < t \implies \text{count } \Delta \text{neg } s = \text{zcount } (c\text{-caps } c0\ p) s$ 
 $\wedge s. s < t \implies \text{count } \Delta \text{mint-self } s = 0$ 
 $\wedge p\ s'. s' \leq s \implies \text{count } \Delta \text{mint-msg } (p, s') = 0$ 
subgoal for  $s'$ 
using change(2) nocaps  $s(1)$  order-class.order.not-eq-order-implies-strict
by (fastforce simp: change(9))
subgoal for  $s'$ 
using nocaps[rule-format, of  $s'$ ]
by (simp add: change(9)  $\wedge s'. s' < t \implies \text{int } (\text{count } \Delta \text{neg } s') = \text{zcount}$ 
(c-caps  $c0\ p) s'$ )
subgoal for  $p\ s'$ 
using change(4)[unfolding minting-msg-def, rule-format, of  $(p, s')$ ]  $s(3)$ 
by (force intro: ccontr)
done
{ assume less:  $s < t$ 
have caps-le-ii0:  $\text{zcount } (c\text{-caps } c0\ p) s \leq \text{zcount } M\ s$ 
proof (rule ccontr)
assume nle:  $\neg \text{zcount } (c\text{-caps } c0\ p) s \leq \text{zcount } M\ s$ 
have  $\text{zcount } ?M1\ s < 0$ 
unfolding complex-change(3)
using complex-change(4) nle  $s(3)$  by (auto simp:  $\Delta$ counts(1,2)[OF
less])
then show False
using less least order.strict-trans2
by (force dest!: nosupp[unfolding supported-strong-def, simplified,
rule-format] simp: nonpos-upto-def)
qed
with  $s(2)$  have count0s:  $0 < \text{zcount } M\ s$ 
by auto
have False
using inv0[OF count0s]
apply (elim disj3-split)
subgoal
proof -
assume supported-strong  $M\ s$ 
then obtain  $u$  where  $u < s$   $\text{zcount } M\ u < 0$ 
unfolding supported-strong-def
by blast
have  $0 \leq \text{zcount } ?M1\ u$ 
using least nosupp[unfolding supported-strong-def nonpos-upto-def,
simplified, rule-format, of  $u$ ] order.strict-trans[OF  $u(1)$  less]
by fastforce
then have  $0 < \text{zcount } ?\Delta\text{pos } u$ 
using  $\Delta$ counts(1)[of  $u$ ]  $s(3)$   $u(1)$   $u(2)$  less by force
then have  $0 < \text{count } \Delta \text{mint-self } u \vee 0 < \text{zcount } (\text{timestamps } (\text{zmset-of}$ 

```

```

Δmint-msg)) u
  using gr0I by fastforce
  then obtain u' where u' ≤ u 0 < zcount (c-caps c0 p) u'
  apply atomize-elim
  apply (elim disjE)
  subgoal
    apply simp
    apply (drule change(3)[unfolded minting-self-def, rule-format])
    using s(1) s(2) apply blast
    done
  subgoal
    apply (drule pos-image-zmset-obtain-pre[rotated])
    apply simp
    apply clarify
    subgoal for p'
      apply simp
      apply (drule change(4)[unfolded minting-msg-def, rule-format])
      using order.strict-iff-order apply auto
      done
    done
  done
  done
  then show False
  using s(3) u(1) by auto
qed
using s(3) apply auto []
using caps-le-ii0 apply linarith
done
}
moreover
{ assume eq: s = t
  have count0t: 0 < zcount M t
  using eq caps-le change(9) complex-change(3,4) s(2,3) by auto
  have False
  using 2(1) count0t by auto
}
ultimately have False
using order.not-eq-order-implies-strict s(1) by blast
}
then show ?thesis
by blast
qed
}
then show justified (c-caps c1 p) ?M1
by (intro justified-leastI) blast
qed

```

lemma *InvJustifiedII-implies-InvJustifiedGII*:  
 assumes *InvJustifiedII c*  
 and *InvCapsNonneg c*

```

shows InvJustifiedGII c
using assms
unfolding
  InvJustifiedGII-def
  InvJustifiedII-def
  GlobalIncomingInfo-def
  records-def
  InvCapsNonneg-def
by (auto intro!: justified-add-records justified-sum)

lemma init-config-implies-InvJustifiedII: init-config c  $\implies$  InvJustifiedII c
by (simp add: init-config-def InvJustifiedII-def justified-alt IncomingInfo-def)

lemma performop-preserves-InvJustifiedII:
assumes InvJustifiedII c0
  and next-performop' c0 c1 p  $\Delta$ neg  $\Delta$ mint-msg  $\Delta$ mint-self
  and InvCapsNonneg c0
shows InvJustifiedII c1
unfolding InvJustifiedII-def
apply clarify
subgoal for k p' q
  apply (cases p'=p)
  subgoal
    unfolding next-performop-complexD[OF assms(2)]
    apply (simp only: if-P)
    apply (rule next-performop'-preserves-justified[
      OF assms(1)[unfolded InvJustifiedII-def, rule-format, of p k q],
      OF assms(2,3)])
    done
  subgoal
    unfolding
      next-performopD[OF assms(2)]
      next-performop-complexD[OF assms(2)]
    using assms(1)[unfolded InvJustifiedII-def]
    by simp
  done
done

lemma sendupd-preserves-InvJustifiedII:
assumes InvJustifiedII c0
  and next-sendupd' c0 c1 p tt
shows InvJustifiedII c1
unfolding InvJustifiedII-def justified-alt supported-strong-def next-sendupdD(6)[OF
assms(2)]
apply (intro allI)
subgoal for k p' q t
  apply (cases k  $\leq$  length (c-msg c0 p q))
  subgoal
    apply (drule next-sendupd-complexD(4)[OF assms(2), of - - p'])

```

```

apply (auto dest: assms(1)[unfolded InvJustifiedII-def justified-alt supported-strong-def,
rule-format])
done
subgoal
apply (subst (asm) not-le)
apply (subst (1 2 3 4) next-sendupd-complexD(5)[OF assms(2), of - - p'])
apply simp
apply simp
apply (cases p'=p)
subgoal
apply rule
apply (subst (asm) if-P)
apply simp
apply (subst (1 2 3) if-P)
apply simp
apply simp
unfolding IncomingInfo-def
apply (subst (asm) drop-all)
apply simp
apply (subst drop-all)
apply simp
apply (simp del: zcount-diff)
— The justified condition ensures that anything remaining in temp satisfies
this invariant
apply (drule next-sendupdD(2)[OF assms(2), unfolded justified-alt sup-
ported-strong-def, rule-format])
apply simp
done
subgoal
by (auto intro!: assms(1)[unfolded InvJustifiedII-def justified-alt supported-strong-def,
rule-format])
done
done
done

```

**lemma** *recvupd-preserved-InvJustifiedII:*

```

assumes InvJustifiedII c0
and next-recvupd' c0 c1 p q
shows InvJustifiedII c1
using assms(1)
unfolding
InvJustifiedII-def
next-recvupd-complexD[OF assms(2)]
next-recvupdD[OF assms(2)]
by auto

```

**lemma** *recvcap-preserved-InvJustifiedII:*

```

assumes InvJustifiedII c0
and next-recvcap' c0 c1 p t

```

**shows**  $InvJustifiedII\ c1$   
**unfolding**  $InvJustifiedII-def\ justified-alt\ supported-strong-def\ next-recvcap-complexD[OF\ assms(2)]\ next-recvcapD(5)[OF\ assms(2)]$   
**by** ( $auto\ dest!:\ assms(1)[unfolding\ InvJustifiedII-def\ justified-alt\ supported-strong-def,\ rule-format]$ )

**lemma**  $next'-preserves-InvJustifiedII:$   
 $InvCapsNonneg\ c0 \implies InvJustifiedII\ c0 \implies next'\ c0\ c1 \implies InvJustifiedII\ c1$   
**using**  
 $next'-def$   
 $performop-preserves-InvJustifiedII$   
 $recvcap-preserves-InvJustifiedII$   
 $recvupd-preserves-InvJustifiedII$   
 $sendupd-preserves-InvJustifiedII$   
**by**  $blast$

**lemma**  $alw-InvJustifiedII:$   $spec\ s \implies alw\ (holds\ InvJustifiedII)\ s$   
**apply** ( $frule\ alw-InvCapsNonneg$ )  
**unfolding**  $spec-def$   
**apply** ( $elim\ conjE$ )  
**apply** ( $subst\ (asm)\ holds.simps$ )  
**apply** ( $drule\ init-config-implies-InvJustifiedII$ )  
**apply** ( $coinduction\ arbitrary:\ s\ rule:\ alw.coinduct$ )  
**apply** ( $subst\ (asm)\ (1\ 2)\ alw-nxt$ )  
**apply**  $clarsimp$   
**using**  $next'-preserves-InvJustifiedII$  **apply**  $blast$   
**done**

**lemma**  $alw-InvJustifiedGII:$   $spec\ s \implies alw\ (holds\ InvJustifiedGII)\ s$   
**apply** ( $frule\ alw-InvJustifiedII$ )  
**apply** ( $drule\ alw-InvCapsNonneg$ )  
**apply** ( $simp\ add:\ alw-iff-sdrop\ InvJustifiedII-implies-InvJustifiedGII$ )  
**done**

#### 4.3.6 InvTempJustified

**definition**  $InvTempJustified\ where$   
 $InvTempJustified\ c = (\forall p.\ justified\ (c-caps\ c\ p)\ (c-temp\ c\ p))$

**lemma**  $init-config-implies-InvTempJustified:$   $init-config\ c \implies InvTempJustified\ c$   
**unfolding**  $init-config-def\ InvTempJustified-def$   
**using**  $justified-add-records[OF\ justified-empty]$   
**by**  $auto$

**lemma**  $recvcap-preserves-InvTempJustified:$   
**assumes**  $InvTempJustified\ c0$   
**and**  $next-recvcap'\ c0\ c1\ p\ t$   
**shows**  $InvTempJustified\ c1$   
**using**  $assms(1)[unfolding\ InvTempJustified-def,\ rule-format,\ of\ p]\ assms(1)$

**unfolding** *next-recvcapD*[*OF assms*(2)] *InvTempJustified-def*  
**by** – (*frule justified-add-records*[*of* - - {*#t#*}<sub>*z*</sub>], *auto*)

**lemma** *recvupd-preserves-InvTempJustified*:

**assumes** *InvTempJustified c0*  
**and** *next-recvupd' c0 c1 p t*  
**shows** *InvTempJustified c1*  
**using** *assms*(1)  
**unfolding** *next-recvupdD*[*OF assms*(2)] *InvTempJustified-def*  
**by** *assumption*

**lemma** *sendupd-preserves-InvTempJustified*:

**assumes** *InvTempJustified c0*  
**and** *next-sendupd' c0 c1 p tt*  
**shows** *InvTempJustified c1*  
**using** *assms*(1)  
**unfolding** *next-sendupdD*[*OF assms*(2)] *InvTempJustified-def*  
**using** *next-sendupdD*(2)[*OF assms*(2)]  
**by** *auto*

**lemma** *performop-preserves-InvTempJustified*:

**assumes** *InvTempJustified c0*  
**and** *next-performop' c0 c1 p Δneg Δmint-msg Δmint-self*  
**and** *InvCapsNonneg c0*  
**shows** *InvTempJustified c1*  
**unfolding** *InvTempJustified-def*  
**apply** *clarify*  
**subgoal for** *p'*  
**apply** (*cases p'=p*)  
**subgoal**  
**unfolding** *next-performopD*(5)[*OF assms*(2)] *fun-upd-apply*  
**apply** (*simp only: if-P*)  
**apply** (*rule next-performop'-preserves-justified*[  
*OF assms*(1)[*unfolded InvTempJustified-def, rule-format, of p*],  
*OF assms*(2,3)])  
**done**  
**subgoal**  
**unfolding**  
*next-performopD*[*OF assms*(2)]  
*next-performop-complexD*[*OF assms*(2)]  
**using** *assms*(1)[*unfolded InvTempJustified-def*]  
**by** *simp*  
**done**  
**done**

**lemma** *next'-preserves-InvTempJustified*:

*InvCapsNonneg c0*  $\implies$  *InvTempJustified c0*  $\implies$  *next' c0 c1*  $\implies$  *InvTempJustified c1*  
**using**

```

next'-def
performop-preserves-InvTempJustified
recvcap-preserves-InvTempJustified
recvupd-preserves-InvTempJustified
sendupd-preserves-InvTempJustified
by blast

```

```

lemma alw-InvTempJustified: spec s  $\implies$  alw (holds InvTempJustified) s
apply (frule alw-InvCapsNonneg)
unfolding spec-def
apply (elim conjE)
apply (subst (asm) holds.simps)
apply (drule init-config-implies-InvTempJustified)
apply (coinduction arbitrary: s rule: alw.coinduct)
apply (subst (asm) (1 2) alw-nxt)
apply clarsimp
using next'-preserves-InvTempJustified apply blast
done

```

#### 4.3.7 InvGlobNonposImpRecordsNonpos

InvGlobNonposImpRecordsNonpos states that each processor's *c-glob c q* is a conservative approximation of *records c*.

**definition** *InvGlobNonposImpRecordsNonpos* :: ('p :: finite, 'a) configuration  $\Rightarrow$  bool **where**

*InvGlobNonposImpRecordsNonpos c* =  $(\forall t q. \text{nonpos-upto } (c\text{-glob } c \ q) \ t \longrightarrow \text{nonpos-upto } (\text{records } c) \ t)$

**definition** *InvGlobVacantImpRecordsVacant* :: ('p :: finite, 'a) configuration  $\Rightarrow$  bool **where**

*InvGlobVacantImpRecordsVacant c* =  $(\forall t q. \text{GlobVacantUpto } c \ q \ t \longrightarrow \text{RecordsVacantUpto } c \ t)$

**lemma** *inv-imp-InvGlobNonposImpRecordsNonpos*:

```

assumes InvJustifiedGII c
and InvRecordCount c
and InvRecordsNonneg c
shows InvGlobNonposImpRecordsNonpos c
unfolding InvGlobNonposImpRecordsNonpos-def
apply (rule ccontr)
apply (clarsimp simp: nonpos-upto-def)
subgoal for t q u
proof –
let ?GII = GlobalIncomingInfoAt c q
assume gvu:  $\forall sa \leq t. \text{zcount } (c\text{-glob } c \ q) \ sa \leq 0$ 
assume uleqt:  $u \leq t$ 
assume  $\neg \text{zcount } (\text{records } c) \ u \leq 0$ 
— u is pointstamp that violates InvGlobNonposImpRecordsNonpos
with assms(2) have  $w: 0 < \text{zcount } (\text{records } c) \ u$ 

```

```

    by linarith
      —  $u'$  is the least pointstamp with positive records
    with uleqt obtain  $u'$  where  $u'$ :  $0 < \text{zcount } (\text{records } c) \ u' \ \forall u. \ 0 < \text{zcount } (\text{records } c) \ u \longrightarrow \neg u < u' \ u' \leq t$ 
    using order-zmset-exists-foundation[OF u] by auto
      — from the records count we know that GII also has positive count
    from  $u'(1,3)$  assms(2) gvu have pos-gii:  $0 < \text{zcount } ?GII \ u'$ 
    by (simp add: InvRecordCount-def) (smt (verit) zcount-union)
      — Case distinction on which part of the partition GII's  $u$  is in
    { — Original proof from Abadi paper, change is justified by uprightness
    assume supported-strong  $?GII \ u'$ 
      — uprightness gives us a lesser pointstamp with negative count in GII..
    then obtain  $v$  where  $v$ :  $v \leq u' \ \text{zcount } ?GII \ v < 0$ 
      using order.strict-implies-order supported-strong-def by blast
        — ..which is also negative in records..
    with  $u'(3)$   $v(1)$  assms(2) have  $\text{zcount } (\text{records } c) \ v < 0$ 
      by (smt (verit, ccfv-threshold) InvRecordCount-def gvu local.order-trans zcount-union)
        — ..contradicting InvNrecNonneg
    with assms(3) have False
      unfolding InvRecordsNonneg-def
      using order-class.leD by blast
    }
  moreover
  { — Change is justified by strictly lesser pointstamp in records
  assume  $\exists t' < u'. \ 0 < \text{zcount } (\text{records } c) \ t'$ 
    —  $v$  is a strictly lesser positive count in records..
  then obtain  $v$  where  $v$ :  $v < u' \ 0 < \text{zcount } (\text{records } c) \ v$ 
    by auto
    — ..which contradicts the fact that we obtained  $u'$  as the least, positive
    pointstamp in records
  with  $u'(2)$  have False
    by auto
  }
  moreover
  { — Change is justified by records
  assume  $\text{zcount } ?GII \ u' < \text{zcount } (\text{records } c) \ u'$ 
  then have  $0 < \text{zcount } (c\text{-glob } c \ q) \ u'$ 
    by (simp add: assms(2)[unfolded InvRecordCount-def, rule-format, of q])
  then have False
    using gvu u'(3) by auto
  }
  ultimately show False
    using pos-gii assms(1)[unfolded InvJustifiedGII-def justified-alt, rule-format, OF pos-gii]
    by auto
  qed
done

```

*InvGlobVacantImpRecordsVacant* is the one proved in the Abadi paper. We

prove  $\text{InvGlobNonposImpRecordsNonpos}$ , which implies this.

**lemma** *invs-imp-InvGlobVacantImpRecordsVacant*:

**assumes** *InvJustifiedGII*  $c$

**and** *InvRecordCount*  $c$

**and** *InvRecordsNonneg*  $c$

**shows** *InvGlobVacantImpRecordsVacant*  $c$

**proof** –

{ **fix**  $p\ x$

**assume** *GlobVacantUpto*  $c\ p\ x$

**then have** *GlobNonposUpto*  $c\ p\ x$

**unfolding** *nonpos-upto-def vacant-upto-def* **by** *simp*

**note** *invs-imp-InvGlobNonposImpRecordsNonpos*[*OF* *assms*, *unfolded InvGlobNonposImpRecordsNonpos-def*, *rule-format*, *OF* *this*]

**then have** *RecordsVacantUpto*  $c\ x$

**using** *assms*(3)

**unfolding** *nonpos-upto-def vacant-upto-def InvRecordsNonneg-def*

**by** (*simp add: order-class.order.antisym*)

}

**then show** *?thesis*

**unfolding** *InvGlobVacantImpRecordsVacant-def* **by** *simp*

**qed**

**lemma** *alw-InvGlobNonposImpRecordsNonpos*: *spec*  $s \implies \text{alw}(\text{holds } \text{InvGlobNonposImpRecordsNonpos})\ s$

**apply** (*frule alw-InvJustifiedGII*)

**apply** (*frule alw-InvRecordCount*)

**apply** (*drule alw-InvRecordsNonneg*)

**apply** (*simp add: alw-iff-sdrop invs-imp-InvGlobNonposImpRecordsNonpos*)

**done**

**lemma** *alw-InvGlobVacantImpRecordsVacant*: *spec*  $s \implies \text{alw}(\text{holds } \text{InvGlobVacantImpRecordsVacant})\ s$

**apply** (*frule alw-InvGlobNonposImpRecordsNonpos*)

**apply** (*frule alw-InvJustifiedGII*)

**apply** (*frule alw-InvRecordCount*)

**apply** (*drule alw-InvRecordsNonneg*)

**apply** (*simp add: alw-iff-sdrop invs-imp-InvGlobVacantImpRecordsVacant*)

**done**

#### 4.3.8 SafeGlobVacantUptoImpliesStickyNrec

This is the main safety property proved in the Abadi paper.

**lemma** *invs-imp-SafeGlobVacantUptoImpliesStickyNrec*:

*SafeRecordsMono*  $s \implies \text{holds } \text{InvGlobVacantImpRecordsVacant}\ s \implies \text{SafeGlobVacantUptoImpliesStickyNrec}\ s$

**by** (*simp add: InvGlobVacantImpRecordsVacant-def SafeRecordsMono-def SafeGlobVacantUptoImpliesStickyNrec-def*)

**lemma** *alw-SafeGlobVacantUptoImpliesStickyNrec*:  
*spec s*  $\implies$  *alw SafeGlobVacantUptoImpliesStickyNrec s*  
**by** (*meson alw-iff-sdrop invs-imp-SafeGlobVacantUptoImpliesStickyNrec alw-SafeRecordsMono alw-InvGlobVacantImpRecordsVacant*)

### 4.3.9 InvGlobNonposEqVacant

The least pointstamps in glob are always positive, i.e. *nonpos-upto* and *vacant-upto* on glob are equivalent.

**definition** *InvGlobNonposEqVacant* **where**

*InvGlobNonposEqVacant c* =  $(\forall q t. \text{GlobVacantUpto } c \ q \ t = \text{GlobNonposUpto } c \ q \ t)$

**lemma** *invs-imp-InvGlobNonposEqVacant*:

**assumes** *InvRecordCount c*

**and** *InvJustifiedGII c*

**and** *InvRecordsNonneg c*

**shows** *InvGlobNonposEqVacant c*

**proof** –

**note** *safe* = *invs-imp-InvGlobNonposImpRecordsNonpos*[*OF assms(2,1,3)*, *unfolded InvGlobNonposImpRecordsNonpos-def nonpos-upto-def*, *THEN spec2*, *THEN mp*]

**note** *nonneg* = *assms(3)*[*unfolded InvRecordsNonneg-def*, *rule-format*]

**note** *count* = *assms(1)*[*unfolded InvRecordCount-def*, *rule-format*]

{ **fix** *q t*

**assume** *np: GlobNonposUpto c q t*

**assume** *nv:  $\neg$  GlobVacantUpto c q t*

— Obtain the least, negative pointstamp in glob

**obtain** *s* **where** *s: s  $\leq$  t zcount (c-glob c q) s < 0  $\forall$  s' < s. zcount (c-glob c q) s' = 0*

**apply** *atomize-elim*

**using** *nv*[*unfolded vacant-upto-def*]

**apply** *clarsimp*

**apply** (*drule elem-order-zmset-exists-foundation*)

**apply** *clarsimp*

**subgoal for** - *s*

**apply** (*rule exI*[*of - s*])

**apply** (*meson order.ordering-axioms nonpos-upto-def np order-class.le-less order-trans zcount-ne-zero-iff*)

**done**

**done**

— No records  $s' \leq s$  can exist, since that would violate *InvGlobNonposImpRecordsNonpos*

**have** *norec: s'  $\leq$  s  $\implies$  zcount (records c) s' = 0* **for** *s'*

**by** (*metis (full-types) nonneg nonpos-upto-def np order-class.antisym-conv order-trans s(1) safe*)

— Hence GII must be positive at *s*

**have** *gii: zcount (GlobalIncomingInfoAt c q) s > 0*

**using** *count*[*of q*] *s(2) norec*[*of s, simplified*]

**by** (*metis add commute less-add-same-cancel1 zcount-union*)  
 — which means it must be justified by one of these three disjuncts  
**then consider**  
*supported-strong (GlobalIncomingInfoAt c q) s |*  
 $\exists t' < s. 0 < \text{zcount}(\text{records } c) t' |$   
*zcount (GlobalIncomingInfoAt c q) s < zcount (records c) s*  
**using** *assms(2)[unfolded InvJustifiedGII-def justified-alt, rule-format, OF gii]*  
**by** *atomize-elim auto*  
**then have** *False*  
**proof cases**  
**case 1** — *s* can't be *supported-strong*, since either *glob* or *records* would have  
 to be positive at the support  
**then show** *False*  
**using** *norec count s(3)*  
**unfolding** *supported-strong-def*  
**by** (*metis (full-types) add commute add.left-neutral less-le preorder-class.less-irrefl*  
*zcount-union*)  
**next**  
**case 2** — no lesser capabilities exist  
**then show** *?thesis*  
**using** *norec s(2,3) order.order-iff-strict by auto*  
**next**  
**case 3** — no capabilities at *s* exist  
**then show** *?thesis*  
**unfolding** *norec[of s, simplified]*  
**using** *gii by auto*  
**qed**  
**}**  
**then show** *?thesis*  
**unfolding** *InvGlobNonposEqVacant-def*  
**apply** (*intro allI*)  
**apply** (*rule iffI*)  
**apply** (*simp add: nonpos-upto-def vacant-upto-def*)  
**apply** *auto*  
**done**  
**qed**

**lemma** *alw-InvGlobNonposEqVacant: spec s  $\implies$  alw (holds InvGlobNonposEqVacant) s*

**using**  
*alw-InvJustifiedGII*  
*alw-InvRecordCount*  
*alw-InvRecordsNonneg*  
*invs-imp-InvGlobNonposEqVacant*  
**by** (*metis alw-iff-sdrop holds.elims(2) holds.elims(3)*)

#### 4.3.10 InvInfoJustifiedWithII and InvInfoJustifiedWithGII

**definition** *InvInfoJustifiedWithII* where

$InvInfoJustifiedWithII\ c = (\forall k\ p\ q. \text{justified-with } (c\text{-caps } c\ p) (\text{InfoAt } c\ k\ p\ q) (\text{IncomingInfo } c\ (k+1)\ p\ q))$

**definition**  $InvInfoJustifiedWithGII$  **where**

$InvInfoJustifiedWithGII\ c = (\forall k\ p\ q. \text{justified-with } (\text{records } c) (\text{InfoAt } c\ k\ p\ q) (\text{GlobalIncomingInfo } c\ (k+1)\ p\ q))$

**lemma**  $init\text{-config}\text{-implies}\text{-}InvInfoJustifiedWithII$ :  $init\text{-config } c \implies InvInfoJustifiedWithII\ c$

**unfolding**  $init\text{-config}\text{-def } InvInfoJustifiedWithII\text{-def } \text{justified-with}\text{-def } InfoAt\text{-def}$   
**by**  $auto$

This proof relies heavily on the addition properties summarized in the lemma  $\llbracket \text{justified-with } (c\text{-caps } ?c0.0\ ?p) ?M\ ?N; \text{next-performop}'\ ?c0.0\ ?c1.0\ ?p\ ?\Delta neg\ ?\Delta mint\text{-msg}\ ?\Delta mint\text{-self}; \forall t. 0 \leq \text{zcount } (c\text{-caps } ?c0.0\ ?p)\ t; \text{justified } (c\text{-caps } ?c0.0\ ?p)\ ?N \rrbracket \implies \text{justified-with } (c\text{-caps } ?c0.0\ ?p + \text{zmsset-of } ?\Delta mint\text{-self} - \text{zmsset-of } ?\Delta neg)\ ?M\ (?N + \text{zmsset-of } ?\Delta mint\text{-self} + \text{times-tamps } (\text{zmsset-of } ?\Delta mint\text{-msg}) - \text{zmsset-of } ?\Delta neg)$

**lemma**  $performop\text{-preserves}\text{-}InvInfoJustifiedWithII$ :

**assumes**  $InvInfoJustifiedWithII\ c0$

**and**  $\text{next-performop}'\ c0\ c1\ p'\ \Delta neg\ \Delta mint\text{-msg}\ \Delta mint\text{-self}$

**and**  $InvJustifiedII\ c0$

**and**  $InvCapsNonneg\ c0$

**shows**  $InvInfoJustifiedWithII\ c1$

**unfolding**  $InvInfoJustifiedWithII\text{-def}$

**apply** ( $intro\ allI$ )

**subgoal for**  $k\ p\ q$

**apply** ( $cases\ p'=p$ )

**subgoal**

**unfolding**  $\text{next-performop-complexD}[OF\ \text{assms}(2)]\ \text{next-performopD}[OF\ \text{assms}(2)]$

**apply** ( $simp\ only: \text{add-diff-eq}\ \text{if-P}\ \text{fun-upd-same}$ )

**apply** ( $subst\ (4)\ \text{add-commute}$ )

**apply** ( $subst\ \text{add.assoc}[\text{symmetric}]$ )

**apply** ( $rule\ \text{next-performop}'\text{-preserves-justified-with}$ )

**using**  $\text{assms}(1)[\text{unfolded } InvInfoJustifiedWithII\text{-def}]$  **apply**  $simp$

**using**  $\text{assms}(2)$  **apply**  $simp$

**using**  $\text{assms}(4)[\text{unfolded } InvCapsNonneg\text{-def}]$  **apply**  $simp$

**using**  $\text{assms}(3)[\text{unfolded } InvJustifiedII\text{-def}]$  **apply**  $simp$

**done**

**subgoal**

**using**  $\text{assms}(1)[\text{unfolded } InvInfoJustifiedWithII\text{-def } \text{justified-with}\text{-def}]$

**by** ( $auto\ simp: \text{not-less } \text{justified-with}\text{-def } \text{next-performop-complexD}[OF\ \text{assms}(2)]\ \text{next-performopD}[OF\ \text{assms}(2)]$ )

**done**

**done**

**lemma**  $sendupd\text{-preserves}\text{-}InvInfoJustifiedWithII$ :

**assumes**  $InvInfoJustifiedWithII\ c0$

**and**  $\text{next-sendupd}'\ c0\ c1\ p'\ tt$

```

  and InvTempJustified c0
shows InvInfoJustifiedWithII c1
unfolding InvInfoJustifiedWithII-def
proof (intro allI impI)
  fix k :: nat and p q
  note complex = next-sendupd-complexD[OF assms(2)]
  note change = next-sendupdD[OF assms(2)]
  consider
    p' = p k < length (c-msg c0 p q) |
    p' = p k = length (c-msg c0 p q) |
    p' = p k > length (c-msg c0 p q) |
    p' ≠ p
  by atomize-elim auto
  then show justified-with (c-caps c1 p) (InfoAt c1 k p q) (IncomingInfo c1 (k+1)
p q)
  proof cases
    case 1
    then show ?thesis
    by (metis InvInfoJustifiedWithII-def Suc-eq-plus1 Suc-le-eq assms(1) change(6)
complex(4,7) order-class.less-le)
  next
    case 2
    have temp0: c-temp c0 p = InfoAt c1 k p q + c-temp c1 p
    unfolding complex change
    using 2 by (auto simp: algebra-simps)
    have pi: PositiveImplies (InfoAt c1 k p q) (c-temp c0 p)
    unfolding PositiveImplies-def complex
    using 2(2) by (auto simp: InfoAt-def)
    have itemp: IncomingInfo c1 (k+1) p q = c-temp c1 p
    unfolding IncomingInfo-def
    using 2 by (simp add: change)
    show ?thesis
    apply (rule PositiveImplies-justified-with)
    unfolding itemp temp0[symmetric]
    unfolding change
    using assms(3) apply (simp add: InvTempJustified-def)
    using pi apply simp
    done
  next
    case 3
    then show ?thesis
    by (metis add-cancel-right-left complex(7) justified-with-leastI preorder-class.less-asy
InfoAt-def zcount-union)
  next
    case 4
    then show ?thesis
    unfolding complex change
    using assms(1)[unfolded InvInfoJustifiedWithII-def, rule-format]
    apply simp

```

done  
qed  
qed

**lemma** *recvupd-preserves-InvInfoJustifiedWithII:*

**assumes** *InvInfoJustifiedWithII c0*  
**and** *next-recvupd' c0 c1 p q*  
**shows** *InvInfoJustifiedWithII c1*  
**using** *assms(1)*  
**unfolding**  
*InvInfoJustifiedWithII-def*  
*next-recvupd-complexD[OF assms(2)]*  
*next-recvupdD[OF assms(2)]*  
**by** *auto*

**lemma** *recvcap-preserves-InvInfoJustifiedWithII:*

**assumes** *InvInfoJustifiedWithII c0*  
**and** *next-recvcap' c0 c1 p t*  
**shows** *InvInfoJustifiedWithII c1*  
**using** *assms(1)*  
**unfolding**  
*InvInfoJustifiedWithII-def*  
*next-recvcap-complexD[OF assms(2)]*  
*next-recvcapD[OF assms(2)]*  
**by** *simp*

**lemma** *invs-imp-InvInfoJustifiedWithGII:*

**assumes** *InvInfoJustifiedWithII c*  
**and** *InvJustifiedII c*  
**and** *InvCapsNonneg c*  
**shows** *InvInfoJustifiedWithGII c*  
**unfolding** *InvInfoJustifiedWithGII-def*  
**apply** *clarify*  
**subgoal for** *k p q*  
**unfolding** *GlobalIncomingInfo-def records-def*  
**apply** *(rule justified-with-add-records)*  
**subgoal**  
**apply** *(rule justified-with-sum[where x = p])*  
**apply** *simp*  
**apply** *simp*  
**apply** *simp*  
**using** *assms(1)[unfolded InvInfoJustifiedWithII-def, rule-format, of p k q]*  
**apply** *simp*  
**using** *assms(2)[unfolded InvJustifiedII-def]* **apply** *simp*  
**using** *assms(3)[unfolded InvCapsNonneg-def]* **apply** *simp*  
**done**  
**apply** *simp*  
**done**  
**done**

```

lemma next'-preserves-InvInfoJustifiedWithII:
  assumes InvInfoJustifiedWithII c0
    and next' c0 c1
    and InvCapsNonneg c0
    and InvJustifiedII c0
    and InvTempJustified c0
  shows InvInfoJustifiedWithII c1
  using assms unfolding next'-def
  apply (elim disjE exE)
    apply (drule (4) performop-preserves-InvInfoJustifiedWithII)
    apply (drule (3) sendupd-preserves-InvInfoJustifiedWithII)
    apply (drule (2) recvupd-preserves-InvInfoJustifiedWithII)
    apply (drule (2) recvcap-preserves-InvInfoJustifiedWithII)
  apply simp
  done

lemma alw-InvInfoJustifiedWithII: spec s  $\implies$  alw (holds InvInfoJustifiedWithII)
s
  apply (frule alw-InvCapsNonneg)
  apply (frule alw-InvJustifiedII)
  apply (frule alw-InvTempJustified)
  unfolding spec-def
  apply (elim conjE)
  apply (subst (asm) holds.simps)
  apply (drule init-config-implies-InvInfoJustifiedWithII)
  apply (coinduction arbitrary: s rule: alw.coinduct)
  apply (subst (asm) (1 2 3) alw-nxt)
  apply clarsimp
  using next'-preserves-InvInfoJustifiedWithII
  apply blast
  done

```

```

lemma alw-InvInfoJustifiedWithGII: spec s  $\implies$  alw (holds InvInfoJustifiedWith-
GII) s
  by (metis alw-InvCapsNonneg alw-InvInfoJustifiedWithII alw-InvJustifiedII alw-iff-sdrop
holds.elims(2,3) invs-imp-InvInfoJustifiedWithGII)

```

#### 4.3.11 SafeGlobMono and InvMsgInGlob

The records in glob are monotonic. This implies the corollary `InvMsgInGlob`; No incoming message carries a timestamp change that would cause glob to regress.

**definition** *SafeGlobMono* **where**

```

SafeGlobMono c0 c1 = ( $\forall p t. \text{GlobVacantUpto } c0 p t \longrightarrow \text{GlobVacantUpto } c1 p t$ )

```

**definition** *InvMsgInGlob* **where**

$InvMsgInGlob\ c = (\forall p\ q\ t. c\text{-msg}\ c\ p\ q \neq [] \longrightarrow t \in \#_z\ hd\ (c\text{-msg}\ c\ p\ q) \longrightarrow (\exists t' \leq t. 0 < zcount\ (c\text{-glob}\ c\ q)\ t'))$

**lemma** *not-InvMsgInGlob-imp-not-SafeGlobMono*:

**assumes**  $\neg\ InvMsgInGlob\ c0$

**and**  $InvGlobNonposEqVacant\ c0$

**shows**  $\exists\ c1. next\text{-recvupd}\ c0\ c1 \wedge \neg\ SafeGlobMono\ c0\ c1$

**proof** –

**note**  $npeq0 = assms(2)[unfolded\ InvGlobNonposEqVacant\text{-def},\ rule\text{-format}]$

**from**  $assms(1)$  **obtain**  $p\ q\ t\ c1$  **where**  $t$ :

$next\text{-recvupd}'\ c0\ c1\ p\ q$

$t \in \#_z\ hd\ (c\text{-msg}\ c0\ p\ q)$

$GlobVacantUpto\ c0\ q\ t$

**by** (*auto simp: InvMsgInGlob-def npeq0 nonpos-upto-def not-less dest!: ex-next-recvupd*)

**have**  $nvu: \neg\ GlobVacantUpto\ c1\ q\ t$

**unfolding**  $vacant\text{-upto-def}\ next\text{-recvupd}D(4)[OF\ t(1)]$

**using**  $t(2)\ t(3)\ vacant\text{-upto-def}$  **by** *auto*

**from**  $t(3)\ nvu$  **have**  $\neg\ SafeGlobMono\ c0\ c1$

**by** (*auto simp: SafeGlobMono-def*)

**with**  $t(1)$  **show** *?thesis*

**by** *blast*

**qed**

**lemma** *GII-eq-GIA: GlobalIncomingInfo c 1 p q = (if c-msg c p q = [] then GlobalIncomingInfoAt c q else GlobalIncomingInfoAt c q - hd (c-msg c p q))*

**unfolding**  $GlobalIncomingInfo\text{-def}$

**apply** (*cases c-msg c p q = []*)

**apply** (*simp add: IncomingInfo-def*)

**apply** (*rule sum.cong[OF refl], simp*)

**apply** *simp*

**apply** (*subst diff-conv-add-uminus*)

**apply** (*rule Sum-eq-pick-changed-elem*)

**apply** (*auto simp: IncomingInfo-def drop-Suc sum-list-hd-tl algebra-simps*)

**done**

**lemma** *recvupd-preserves-GlobVacantUpto*:

**assumes**  $GlobVacantUpto\ c0\ q\ t$

**and**  $next\text{-recvupd}'\ c0\ c1\ p\ q$

**and**  $InvInfoJustifiedWithGII\ c0$

**and**  $InvGlobNonposEqVacant\ c1$

**and**  $InvGlobVacantImpRecordsVacant\ c0$

**and**  $InvRecordCount\ c0$

**shows**  $GlobVacantUpto\ c1\ q\ t$

**proof** (*rule ccontr*)

**note**  $npeq1 = assms(4)[unfolded\ InvGlobNonposEqVacant\text{-def},\ rule\text{-format}]$

**note**  $gvu0 = assms(1)[unfolded\ vacant\text{-upto-def},\ rule\text{-format}]$

**note**  $nvu0 = assms(5)[unfolded\ InvGlobVacantImpRecordsVacant\text{-def},\ rule\text{-format},\ OF\ assms(1)]$

**note**  $recordcount = assms(6)[unfolded\ InvRecordCount\text{-def}\ zmultiset\text{-eq}\text{-iff}\ zcount\text{-union},$

```

rule-format]
note change = next-recvupdD[OF assms(2)]
let ?κ = hd (c-msg c0 p q)
from change(1) have iaκ: ?κ = InfoAt c0 0 p q
  unfolding InfoAt-def by (simp add: hd-conv-nth)
assume gvu1: ¬ GlobVacantUpto c1 q t
obtain t' where t':
  t' ≤ t
  0 < zcount (c-glob c1 q) t'
  zcount (c-glob c0 q) t' = 0
  ∀ s < t'. zcount (c-glob c1 q) s ≤ 0
apply atomize-elim
using gvu1
unfolding npeq1 nonpos-upto-def
apply (simp add: not-le)
apply (elim exE conjE)
apply (drule order-zmset-exists-foundation')
apply (elim exE conjE)
subgoal for s s'
  apply (rule exI[of - s'])
  using gvu0 apply auto
  done
done
from t'(2,3) have countκ: 0 < zcount ?κ t'
  by (auto simp: change(4))
note assms(3)[unfolded InvInfoJustifiedWithGII-def justified-with-alt, rule-format,
OF countκ[unfolded iaκ]]
then consider
  ∃ s < t'. zcount (InfoAt c0 0 p q) s < 0 ∧ (∀ s' < s. zcount (InfoAt c0 0 p q) s' ≤
0) |
  ∃ s < t'. zcount (GlobalIncomingInfo c0 1 p q) s < 0 ∧ (∀ s' < s. zcount (InfoAt
c0 0 p q) s' ≤ 0)
  ∄ s. s < t' ∧ zcount (InfoAt c0 0 p q) s < 0 ∧ (∀ s' < s. zcount (InfoAt c0 0 p
q) s' ≤ 0) |
  ∃ s < t'. 0 < zcount (records c0) s |
  zcount (InfoAt c0 0 p q + GlobalIncomingInfo c0 1 p q) t' < zcount (records
c0) t'
  by atomize-elim auto
then show False
proof cases
  case 1
  then obtain s where s: s < t' zcount (InfoAt c0 0 p q) s < 0 ∀ s' < s. zcount
(InfoAt c0 0 p q) s' ≤ 0
  by blast
  then have globs: zcount (c-glob c1 q) s < 0
  using gvu0 iaκ t'(1) by (auto simp: change(4))
  have zcount (c-glob c1 q) s ≤ 0
  using globs by linarith
  then have ∀ s' ≤ s. zcount (c-glob c1 q) s' ≤ 0

```

```

    using s(1) t'(4) by auto
  with globs show False
  by (auto simp: npeq1[unfolded nonpos-upto-def, symmetric] vacant-upto-def)
next
case 2
  then obtain s where s: s < t' zcount (GlobalIncomingInfo c0 1 p q) s < 0
  ∀ s' < s. zcount (InfoAt c0 0 p q) s' ≤ 0
  by blast
  from 2(2) s(1,3) have zcount (InfoAt c0 0 p q) s ≥ 0
  by force
  have rc: zcount (records c0) s = 0
  using order.strict-implies-order order.strict-trans2 nvu0 s(1) t'(1) vacant-upto-def
by blast
show False
  using assms(6) change(1,4) s(1,2) rc t'(4)
  unfolding GII-eq-GIA InvRecordCount-def fun-upd-same
  by clarsimp (metis add commute add-mono-thms-linordered-field(1) not-le
recordcount)
next
case 3
  with nvu0 t'(1) show False
  unfolding vacant-upto-def by auto
next
case 4
  then have 0 < zcount (c-glob c0 q) t'
  using change(1) iak t'(3)
  unfolding GII-eq-GIA
  apply clarsimp
  apply (metis add.right-neutral preorder-class.less-irrefl recordcount)
  done
  then show False
  by (simp add: t'(3))
qed
qed

```

```

lemma recvupd-imp-SafeGlobMono:
  assumes next-recvupd' c0 c1 p q
  and InvInfoJustifiedWithGII c0
  and InvGlobNonposEqVacant c1
  and InvGlobVacantImpRecordsVacant c0
  and InvRecordCount c0
  shows SafeGlobMono c0 c1
  unfolding SafeGlobMono-def
  apply clarify
  subgoal for q' t
  apply (cases q'=q)
  subgoal
  apply (rule recvupd-preserves-GlobVacantUpto)
  using assms apply simp-all

```

```

done
subgoal
  unfolding next-recvupdD[OF assms(1)]
  by simp
done
done

```

```

lemma next'-imp-SafeGlobMono:
  assumes next' c0 c1
    and InvInfoJustifiedWithGII c0
    and InvGlobNonposEqVacant c1
    and InvGlobVacantImpRecordsVacant c0
    and InvRecordCount c0
  shows SafeGlobMono c0 c1
  using assms unfolding next'-def
  apply (elim disjE exE)
  subgoal
    by (auto simp: SafeGlobMono-def dest: next-performopD(7))
  subgoal
    by (auto simp: SafeGlobMono-def dest: next-sendupdD(5))
  subgoal
    by (auto intro: recvupd-imp-SafeGlobMono)
  subgoal
    by (auto simp: SafeGlobMono-def dest: next-recvcapD(4))
  subgoal
    by (simp add: SafeGlobMono-def)
done

```

```

lemma invs-imp-InvMsgInGlob:
  fixes c0 :: ('p::finite, 'a) configuration
  assumes InvInfoJustifiedWithGII c0
    and InvGlobNonposEqVacant c0
    and InvGlobVacantImpRecordsVacant c0
    and InvRecordCount c0
    and InvJustifiedII c0
    and InvCapsNonneg c0
    and InvRecordsNonneg c0
  shows InvMsgInGlob c0
  using assms
  apply -
  apply (rule ccontr)
  apply (drule not-InvMsgInGlob-imp-not-SafeGlobMono[rotated, OF assms(2)])
  apply (elim exE conjE)
  apply (frule recvupd-imp-SafeGlobMono)
  apply simp
  subgoal
    apply (rule invs-imp-InvGlobNonposEqVacant)
    apply (drule (2) recvupd-preserves-InvRecordCount)
    apply (drule (1) recvupd-preserves-InvJustifiedII)

```

```

apply (rule InvJustifiedII-implies-InvJustifiedGII)
apply simp
apply (drule (2) recvupd-preserves-InvCapsNonneg)
apply (simp add: InvRecordsNonneg-def next-recvupd-complexD)
done
apply simp
apply simp
apply simp
done

```

**lemma** *alw-SafeGlobMono: spec s  $\implies$  alw (relates SafeGlobMono) s*  
**proof** –

```

assume spec: spec s
{ assume alw next s
  moreover assume alw (holds InvGlobNonposEqVacant) s
  moreover assume alw (holds InvGlobVacantImpRecordsVacant) s
  moreover assume alw (holds InvInfoJustifiedWithGII) s
  moreover assume alw (holds InvRecordCount) s
  ultimately have alw (relates SafeGlobMono) s
    apply (coinduction arbitrary: s)
    apply (clarsimp simp: relates-def intro!: next'-imp-SafeGlobMono)
    apply (rule conjI)
    apply (rule next'-imp-SafeGlobMono)
      apply auto [2]
      apply (subst (asm) alw-holds2)
      apply clarify
      apply (drule alwD)+
      apply simp
      apply auto
    done
  }
with spec show ?thesis
apply –
apply (frule alw-InvGlobNonposEqVacant)
apply (frule alw-InvGlobVacantImpRecordsVacant)
apply (frule alw-InvInfoJustifiedWithGII)
apply (frule alw-InvRecordCount)
unfolding spec-def
apply auto
done

```

**qed**

```

lemma alw-InvMsgInGlob: spec s  $\implies$  alw (holds InvMsgInGlob) s
apply (frule alw-InvInfoJustifiedWithGII)
apply (frule alw-InvGlobNonposEqVacant)
apply (frule alw-InvGlobVacantImpRecordsVacant)
apply (frule alw-InvRecordCount)
apply (frule alw-InvJustifiedII)
apply (frule alw-InvCapsNonneg)

```

**apply** (*drule alw-InvRecordsNonneg*)  
**apply** (*simp add: alw-iff-sdrop invs-imp-InvMsgInGlob*)  
**done**

**lemma** *SafeGlobMono-preserves-vacant*:  
**assumes**  $\forall t' \leq t. \text{zcount } (c\text{-glob } c0 \ q) \ t' = 0$   
**and**  $(\lambda c0 \ c1. \text{SafeGlobMono } c0 \ c1)^{**} \ c0 \ c1$   
**shows**  $\forall t' \leq t. \text{zcount } (c\text{-glob } c1 \ q) \ t' = 0$   
**using** *assms(2,1)*  
**by** (*induct rule: rtranclp-induct*)(*auto simp: SafeGlobMono-def vacant-upto-def*)

**lemma** *rtranclp-all-imp-rel*:  $r^{**} \ x \ y \implies \forall a \ b. \ r \ a \ b \longrightarrow r' \ a \ b \implies r'^{**} \ x \ y$   
**by** (*metis mono-rtranclp*)

**lemma** *rtranclp-rel-and-invar*:  $r^{**} \ x \ y \implies Q \ x \implies \forall a \ b. \ Q \ a \ \wedge \ r \ a \ b \longrightarrow P \ a \ b$   
 $\wedge \ Q \ b \implies (\lambda x \ y. \ P \ x \ y \ \wedge \ Q \ y)^{**} \ x \ y$   
**apply** (*induct rule: rtranclp-induct*)  
**apply** *auto* []  
**apply** (*metis (no-types, lifting) rtranclp.simps*)  
**done**

**lemma** *rtranclp-invar-conclude-last*:  $(\lambda x \ y. \ P \ x \ y \ \wedge \ Q \ y)^{**} \ x \ y \implies Q \ x \implies Q \ y$   
**using** *rtranclp.cases by fastforce*

**lemma** *InvCapsNonneg-imp-InvRecordsNonneg*:  $\text{InvCapsNonneg } c \implies \text{InvRecordsNonneg } c$   
**unfolding** *InvCapsNonneg-def InvRecordsNonneg-def records-def*  
**by** (*auto simp: zcount-sum intro!: sum-nonneg add-nonneg-nonneg*)

**lemma** *invs-imp-msg-in-glob*:  
**fixes**  $c :: ('p::\text{finite}, 'a) \ \text{configuration}$   
**assumes**  $M \in \text{set } (c\text{-msg } c \ p \ q)$   
**and**  $t \in \#_z \ M$   
**and** *InvGlobNonposEqVacant c*  
**and** *InvJustifiedII c*  
**and** *InvInfoJustifiedWithII c*  
**and** *InvGlobVacantImpRecordsVacant c*  
**and** *InvRecordCount c*  
**and** *InvCapsNonneg c*  
**and** *InvMsgInGlob c*  
**shows**  $\exists t' \leq t. \ 0 < \text{zcount } (c\text{-glob } c \ q) \ t'$   
**proof** (*rule ccontr*)  
**assume**  $\neg (\exists t' \leq t. \ 0 < \text{zcount } (c\text{-glob } c \ q) \ t')$   
**then have** *vacant:  $\forall t' \leq t. \ \text{zcount } (c\text{-glob } c \ q) \ t' = 0$*   
**using** *assms(3)*  
**by** (*auto simp: InvGlobNonposEqVacant-def vacant-upto-def nonpos-upto-def*)  
**let**  $?c1 = \text{the } (\text{while-option } (\lambda c. \ \text{hd } (c\text{-msg } c \ p \ q) \neq M) \ (\lambda c. \ \text{SOME } c'. \ \text{next-recvupd}' \ c \ c' \ p \ q) \ c)$   
**have**  $r[\text{simp}]: \ M \in \text{set } (c\text{-msg } c \ p \ q) \implies c\text{-msg } (\text{SOME } c'. \ \text{next-recvupd}' \ c \ c' \ p$

$q) p q = tl (c\text{-msg } c p q)$  **for**  $c$   
**by** (*rule someI2-ex[OF ex-next-recvupd]*) (*auto simp: next-recvupd'-def*)  
**obtain**  $c1$  **where** *while-some*:  
*while-option* ( $\lambda c. hd (c\text{-msg } c p q) \neq M$ ) ( $\lambda c. SOME c'. next\text{-recvupd}' c c' p q$ )  
 $c = Some\ c1$   
**apply** *atomize-elim*  
**apply** (*rule measure-while-option-Some[where P= $\lambda c. M \in set (c\text{-msg } c p q)$*   
*and f= $\lambda c. Min \{i. i < length (c\text{-msg } c p q) \wedge M = c\text{-msg } c p q ! i\}$* )  
**apply** *clarsimp*  
**apply** *safe*  
**apply** (*metis list.exhaust-sel list.sel(2) set-ConsD*)  
**apply** (*subst Min-gr-iff*)  
**apply** (*auto simp: in-set-conv-nth nth-tl*) [2]  
**apply** *clarsimp*  
**apply** (*subst Min-less-iff*)  
**apply** (*auto simp: in-set-conv-nth nth-tl*) []  
**apply** (*clarsimp simp: in-set-conv-nth nth-tl*)  
**apply** (*metis (no-types, opaque-lifting) Suc-less-eq Suc-pred hd-conv-nth*  
*list.size(3) not-gr-zero not-less-zero*)  
**apply** (*clarsimp simp: in-set-conv-nth nth-tl*)  
**subgoal for**  $s\ x$   
**by** (*rule exI[of - x-1]*)  
(*metis One-nat-def Suc-le-eq Suc-pred' diff-less diff-less-mono hd-conv-nth*  
*length-tl list.size(3) not-gr-zero nth-tl zero-less-Suc*)  
**apply** (*meson assms(1) in-set-conv-nth*)  
**done**  
**have**  $c1: (\lambda c0\ c1. next\text{-recvupd}' c0\ c1\ p\ q)^{**} c\ c1\ c\text{-msg } c1\ p\ q \neq []\ hd (c\text{-msg } c1\ p\ q) = M$   
**subgoal**  
**apply** (*rule conjunct2*)  
**apply** (*rule while-option-rule[OF - while-some, where P= $\lambda d. M \in set (c\text{-msg } d\ p\ q) \wedge (\lambda c0\ c1. next\text{-recvupd}' c0\ c1\ p\ q)^{**} c\ d]$* )  
**apply** (*rule conjI*)  
**apply** (*metis list.sel(1) list.sel(3) list.set-cases r*)  
**apply** (*auto simp: assms(1) elim!: rtrancl-into-rtrancl[to-pred] intro: someI2-ex[OF ex-next-recvupd]*)  
**done**  
**subgoal**  
**using** *while-option-rule[OF - while-some, where P= $\lambda c. M \in set (c\text{-msg } c\ p\ q)$*   
 $q]$   
**by** (*metis assms(1) empty-iff hd-Cons-tl list.set(1) r set-ConsD*)  
**subgoal**  
**using** *while-option-stop[OF while-some]* **by** *simp*  
**done**  
**have** *invs-trancl*:  
( $\lambda c0\ c1. SafeGlobMono\ c0\ c1 \wedge InvJustifiedII\ c1 \wedge InvGlobNonposEqVacant\ c1$   
 $\wedge InvInfoJustifiedWithII\ c1 \wedge InvGlobVacantImpRecordsVacant\ c1 \wedge InvRecordCount\ c1 \wedge InvCapsNonneg\ c1)^{**} c\ c1$ )  
**apply** (*rule rtranclp-rel-and-invar*)

```

using c1(1) apply simp
using assms(3-8) apply simp
apply clarsimp
subgoal for a b
  apply (frule InvJustifiedII-implies-InvJustifiedGII)
  apply simp
  apply (frule recvupd-preserves-InvJustifiedII)
  apply simp
  apply (frule InvCapsNonneg-imp-InvRecordsNonneg)
  apply (frule next-preserves-InvRecordCount[of - b])
  apply (auto simp: next'-def) []
  apply (frule recvupd-preserves-InvCapsNonneg[of - b])
  apply simp
  apply (frule InvJustifiedII-implies-InvJustifiedGII[of b])
  apply simp
  apply (frule InvCapsNonneg-imp-InvRecordsNonneg[of b])
  apply (frule invs-imp-InvGlobNonposEqVacant[of b])
  apply simp-all [2]
  apply (frule recvupd-preserves-InvInfoJustifiedWithII[of - b])
  apply simp
  apply (frule invs-imp-InvInfoJustifiedWithGII)
  apply simp-all [2]
  apply (frule invs-imp-InvInfoJustifiedWithGII[of b])
  apply simp-all [2]
  apply (intro conjI)
    apply (rule next'-imp-SafeGlobMono)
    apply (clarsimp simp: next'-def)
    apply simp-all [7]
    apply (rule invs-imp-InvGlobVacantImpRecordsVacant)
    apply simp-all
  done
done
then have trancl-mono: ( $\lambda c0 c1. \text{SafeGlobMono } c0 c1$ )** c c1
  by (metis (no-types, lifting) rtranclp-all-imp-rel)
then have vacant-c1:  $\forall t' \leq t. \text{zcount } (c\text{-glob } c1 q) t' = 0$ 
  by (rule SafeGlobMono-preserves-vacant[OF vacant])
have InvMsgInGlob: InvMsgInGlob c1
  using rtranclp-invar-conclude-last[OF invs-trancl] assms
  apply (intro invs-imp-InvMsgInGlob)
  using invs-imp-InvInfoJustifiedWithGII InvCapsNonneg-imp-InvRecordsNonneg
apply blast+
done
have  $\exists t' \leq t. 0 < \text{zcount } (c\text{-glob } c1 q) t'$ 
  using InvMsgInGlob[unfolded InvMsgInGlob-def] c1(2,3) assms(1,2)
  by auto
then show False
  using vacant-c1 by auto
qed

```

```

lemma alw-msg-glob: spec s  $\implies$ 
  alw (holds ( $\lambda c. \forall p q t. (\exists M \in \text{set } (c\text{-msg } c p q). t \in \#_z M) \longrightarrow (\exists t' \leq t. 0 <$ 
zcount (c-glob c q) t')) s
  apply (frule alw-InvGlobNonposEqVacant)
  apply (frule alw-InvJustifiedII)
  apply (frule alw-InvInfoJustifiedWithII)
  apply (frule alw-InvGlobVacantImpRecordsVacant)
  apply (frule alw-InvRecordCount)
  apply (frule alw-InvCapsNonneg)
  apply (drule alw-InvMsgInGlob)
  apply (coinduction arbitrary: s)
  apply clarsimp
  apply (rule conjI)
  apply clarify
  apply (rule invs-imp-msg-in-glob)
    apply auto [2]
  using holds.elims(2) apply blast+
  done

end

```

## 5 Antichains

**definition** *incomparable where*

*incomparable A = ( $\forall x \in A. \forall y \in A. x \neq y \longrightarrow \neg x < y \wedge \neg y < x$ )*

**lemma** *incomparable-empty[simp, intro]: incomparable {}*

**unfolding** *incomparable-def* **by** *auto*

**typedef** (**overloaded**) *'a :: order antichain =*

*{A :: 'a set. finite A  $\wedge$  incomparable A}*

**morphisms** *set-antichain antichain*

**by** *auto*

**setup-lifting** *type-definition-antichain*

**lift-definition** *member-antichain :: 'a :: order  $\Rightarrow$  'a antichain  $\Rightarrow$  bool ( $\langle (-) \in_A - \rangle$ )*

[51, 51] 50) **is** *Set.member* .

**abbreviation** *not-member-antichain :: 'a :: order  $\Rightarrow$  'a antichain  $\Rightarrow$  bool ( $\langle (-) \notin_A$*

*-) \rangle* [51, 51] 50) **where**

*x  $\notin_A A \equiv \neg x \in_A A$*

**lift-definition** *empty-antichain :: 'a :: order antichain ( $\langle \{\}_A \rangle$ ) **is** {} **by** *simp**

**lemma** *mem-antichain-nonempty[simp]: s  $\in_A A \implies A \neq \{\}_A$*

**by** *transfer auto*

**definition** *minimal-antichain*  $A = \{x \in A. \neg(\exists y \in A. y < x)\}$

**lemma** *in-minimal-antichain*:  $x \in \text{minimal-antichain } A \longleftrightarrow x \in A \wedge \neg(\exists y \in A. y < x)$

**unfolding** *minimal-antichain-def* **by** *auto*

**lemma** *in-antichain-minimal-antichain*[*simp*]:  $\text{finite } M \implies x \in_A \text{antichain } (\text{minimal-antichain } M) \longleftrightarrow x \in \text{minimal-antichain } M$

**apply** (*clarsimp simp: minimal-antichain-def member-antichain.rep-eq*)

**apply** (*intro conjI iffI*)

**apply** (*subst (asm) antichain-inverse*)

**apply** (*simp add: incomparable-def*)

**apply** *simp*

**apply** (*subst (asm) antichain-inverse*)

**apply** (*simp add: incomparable-def*)

**apply** *simp*

**apply** (*subst antichain-inverse*)

**apply** (*simp add: incomparable-def*)

**apply** *simp*

**done**

**lemma** *incomparable-minimal-antichain*[*simp*]:  $\text{incomparable } (\text{minimal-antichain } A)$

**unfolding** *incomparable-def minimal-antichain-def*

**by** *auto*

**lemma** *finite-minimal-antichain*[*simp*]:  $\text{finite } A \implies \text{finite } (\text{minimal-antichain } A)$

**unfolding** *minimal-antichain-def* **by** *auto*

**lemma** *finite-set-antichain*[*simp, intro*]:  $\text{finite } (\text{set-antichain } A)$

**by** *transfer auto*

**lemma** *minimal-antichain-subset*:  $\text{minimal-antichain } A \subseteq A$

**unfolding** *minimal-antichain-def* **by** *auto*

**lift-definition** *frontier* :: '*t* :: order zmultiset  $\Rightarrow$  '*t* antichain is

$\lambda M. \text{minimal-antichain } \{t. \text{zcount } M t > 0\}$

**by** (*auto simp: finite-subset[OF minimal-antichain-subset finite-zcount-pos]*)

**lemma** *member-frontier-pos-zmset*:  $t \in_A \text{frontier } M \implies 0 < \text{zcount } M t$

**by** (*simp add: frontier-def in-minimal-antichain*)

**lemma** *frontier-comparable-False*[*simp*]:  $x \in_A \text{frontier } M \implies y \in_A \text{frontier } M \implies$

$x < y \implies \text{False}$

**by** *transfer (auto simp: minimal-antichain-def)*

**lemma** *minimal-antichain-idempotent*[*simp*]:  $\text{minimal-antichain } (\text{minimal-antichain } A) = \text{minimal-antichain } A$

**by** (*auto simp: minimal-antichain-def*)

**instantiation** *antichain* :: (*order*) *minus* **begin**  
**lift-definition** *minus-antichain* :: 'a *antichain*  $\Rightarrow$  'a *antichain*  $\Rightarrow$  'a *antichain* **is**  
(-)  
**by** (*auto simp: incomparable-def*)  
**instance** ..  
**end**

**instantiation** *antichain* :: (*order*) *plus* **begin**  
**lift-definition** *plus-antichain* :: 'a *antichain*  $\Rightarrow$  'a *antichain*  $\Rightarrow$  'a *antichain* **is**  $\lambda M$   
*N. minimal-antichain* ( $M \cup N$ )  
**by** (*auto simp: incomparable-def minimal-antichain-def*)  
**instance** ..  
**end**

**lemma** *antichain-add-commute*: ( $M :: 'a :: \text{order antichain}$ ) +  $N = N + M$   
**by** *transfer (auto simp: incomparable-def sup-commute)*

**lift-definition** *filter-antichain* :: ('a :: *order*  $\Rightarrow$  *bool*)  $\Rightarrow$  'a *antichain*  $\Rightarrow$  'a *antichain*  
**is** *Set.filter*  
**by** (*auto simp: incomparable-def*)

**syntax** (*ASCII*)  
-*ACCollect* :: *pttrn*  $\Rightarrow$  'a :: *order antichain*  $\Rightarrow$  *bool*  $\Rightarrow$  'a *antichain* ( $\langle (1\{- :_A \ - /$   
 $\ - \}) \rangle$ )  
**syntax**  
-*ACCollect* :: *pttrn*  $\Rightarrow$  'a :: *order antichain*  $\Rightarrow$  *bool*  $\Rightarrow$  'a *antichain* ( $\langle (1\{- \in_A \ - /$   
 $\ - \}) \rangle$ )  
**syntax-consts**  
-*ACCollect* == *filter-antichain*  
**translations**  
 $\{x \in_A M. P\} == \text{CONST } \text{filter-antichain } (\lambda x. P) M$

**declare** *empty-antichain.rep-eq[simp]*

**lemma** *minimal-antichain-empty[simp]*: *minimal-antichain* {} = {}  
**by** (*simp add: minimal-antichain-def*)

**lemma** *minimal-antichain-singleton[simp]*: *minimal-antichain* {*x*:- ::*order*} = {*x*}  
**by** (*auto simp: minimal-antichain-def*)

**lemma** *minimal-antichain-nonempty*:  
*finite A*  $\Longrightarrow$  (*t*:::*order*)  $\in A \Longrightarrow$  *minimal-antichain A*  $\neq$  {}  
**by** (*auto simp: minimal-antichain-def dest: order-finite-set-exists-foundation[of -*  
*t]*)

**lemma** *minimal-antichain-member*:

*finite*  $A \implies (t :: \text{order}) \in A \implies \exists t'. t' \in \text{minimal-antichain } A \wedge t' \leq t$   
**by** (*auto simp: minimal-antichain-def dest: order-finite-set-exists-foundation*[of  $t$ ])

**lemma** *minimal-antichain-union*:  $\text{minimal-antichain } ((A :: (- :: \text{order}) \text{ set}) \cup B) \subseteq$   
 $\text{minimal-antichain } (\text{minimal-antichain } A \cup \text{minimal-antichain } B)$   
**by** (*auto simp: minimal-antichain-def*)

**lemma** *ac-Diff-iff*:  $c \in_A A - B \longleftrightarrow c \in_A A \wedge c \notin_A B$   
**by** *transfer simp*

**lemma** *ac-DiffD2*:  $c \in_A A - B \implies c \in_A B \implies P$   
**by** *transfer simp*

**lemma** *ac-notin-Diff*:  $\neg x \in_A A - B \implies \neg x \in_A A \vee x \in_A B$   
**by** *transfer simp*

**lemma** *ac-eq-iff*:  $A = B \longleftrightarrow (\forall x. x \in_A A \longleftrightarrow x \in_A B)$   
**by** *transfer auto*

**lemma** *antichain-obtain-foundation*:  
**assumes**  $t \in_A M$   
**obtains**  $s$  **where**  $s \in_A M \wedge s \leq t \wedge (\forall u. u \in_A M \longrightarrow \neg u < s)$   
**using** *assms unfolding member-antichain.rep-eq*  
**by** - (*rule order-finite-set-obtain-foundation*[of *set-antichain*  $M$   $t$ ]; *auto*)

**lemma** *set-antichain1*[*simp*]:  $x \in \text{set-antichain } X \implies x \in_A X$   
**by** *transfer simp*

**lemma** *set-antichain2*[*simp*]:  $x \in_A X \implies x \in \text{set-antichain } X$   
**by** *transfer simp*

## 6 Multigraphs with Partially Ordered Weights

**abbreviation** (*input*) *FROM* **where**  
 $FROM \equiv \lambda(s, l, t). s$

**abbreviation** (*input*) *LBL* **where**  
 $LBL \equiv \lambda(s, l, t). l$

**abbreviation** (*input*) *TO* **where**  
 $TO \equiv \lambda(s, l, t). t$

**notation** *subseq* (**infix**  $\langle \preceq \rangle$  50)

**locale** *graph* =  
**fixes** *weights* :: '*vtx*' :: *finite*  $\Rightarrow$  '*vtx*'  $\Rightarrow$  '*lbl*' :: {*order, monoid-add*} *antichain*

**assumes** *zero-le[simp]*:  $0 \leq (s::'lbl)$   
**and** *plus-mono*:  $(s1::'lbl) \leq s2 \implies s3 \leq s4 \implies s1 + s3 \leq s2 + s4$   
**and** *summary-self*:  $\text{weights } \text{loc } \text{loc} = \{\}_A$   
**begin**

**lemma** *le-plus*:  $(s::'lbl) \leq s + s' \ (s'::'lbl) \leq s + s'$   
**by** (*intro plus-mono*[of  $s \ 0 \ s'$ , *simplified*] *plus-mono*[of  $0 \ s \ s'$ , *simplified*])+

## 6.1 Paths

**inductive** *path* ::  $'vtx \Rightarrow 'vtx \Rightarrow ('vtx \times 'lbl \times 'vtx) \text{ list} \Rightarrow \text{bool}$  **where**  
*path0*:  $l1 = l2 \implies \text{path } l1 \ l2$  []  
| *path*:  $\text{path } l1 \ l2 \ xs \implies \text{lbl} \in_A \ \text{weights } l2 \ l3 \implies \text{path } l1 \ l3 \ (xs \ @ \ [(l2, \text{lbl}, l3)])$

**inductive-cases** *path0E*:  $\text{path } l1 \ l2$  []  
**inductive-cases** *path-AppendE*:  $\text{path } l1 \ l3 \ (xs \ @ \ [(l2, s, l2')])$

**lemma** *path-trans*:  $\text{path } l1 \ l2 \ xs \implies \text{path } l2 \ l3 \ ys \implies \text{path } l1 \ l3 \ (xs \ @ \ ys)$   
**by** (*rotate-tac*, *induct*  $l2 \ l3 \ ys$  *rule*: *path.induct*)  
(auto *intro*: *path.path simp flip*: *append-assoc*)

**lemma** *path-take-from*:  $\text{path } l1 \ l2 \ xs \implies m < \text{length } xs \implies \text{FROM } (xs \ ! \ m) = l2'$   
 $\implies \text{path } l1 \ l2' \ (\text{take } m \ xs)$

**proof** (*induct*  $l1 \ l2 \ xs$  *rule*: *path.induct*)  
**case** ( $\text{path } l1 \ l2 \ xs \ \text{lbl } l3$ )  
**then show** *?case*  
  **apply** (*unfold take-append*)  
  **apply** *simp*  
  **apply** (*cases l2=l2'*)  
  **apply** (*metis linorder-not-less nth-append take-all*)  
  **apply** (*metis case-prod-conv less-Suc-eq nth-append nth-append-length*)  
  **done**  
**qed** *simp*

**lemma** *path-take-to*:  $\text{path } l1 \ l2 \ xs \implies m < \text{length } xs \implies \text{TO } (xs \ ! \ m) = l2' \implies$   
 $\text{path } l1 \ l2' \ (\text{take } (m+1) \ xs)$

**proof** (*induct*  $l1 \ l2 \ xs$  *rule*: *path.induct*)  
**case** ( $\text{path } l1 \ l2 \ xs \ \text{lbl } l3$ )  
**then show** *?case*  
  **apply** (*cases m < length xs*)  
  **apply** (*simp add*: *nth-append*)  
  **apply** *clarsimp*  
  **apply** (*metis case-prod-conv less-antisym nth-append-length path.path*)  
  **done**  
**qed** *simp*

**lemma** *path-determines-loc*:  $\text{path } l1 \ l2 \ xs \implies \text{path } l1 \ l3 \ xs \implies l2 = l3$   
**by** (*induct*  $l1 \ l2 \ xs$  *rule*: *path.induct*) (auto *elim*: *path.cases*)

**lemma** *path-first-loc*:  $\text{path } loc \ loc' \ xs \implies xs \neq [] \implies FROM \ (xs \ ! \ 0) = loc$   
**proof** (*induct rule: path.induct*)  
  **case** (*path l1 l2 xs lbl l3*)  
  **then show** *?case*  
  **by** (*auto elim: path0E simp: nth-append*)  
**qed** *simp*

**lemma** *path-to-eq-from*:  $\text{path } loc1 \ loc2 \ xs \implies i + 1 < \text{length } xs \implies FROM \ (xs \ ! \ (i+1)) = TO \ (xs \ ! \ i)$   
**proof** (*induct rule: path.induct*)  
  **case** (*path l1 l2 xs lbl l3*)  
  **then show** *?case*  
  **apply** (*cases i + 1 < length xs*)  
  **apply** (*simp add: nth-append*)  
  **apply** (*simp add: nth-append*)  
  **apply** (*metis add.commute drop-eq-Nil hd-drop-conv-nth id-take-nth-drop linorder-not-less path-determines-loc path-take-to plus-1-eq-Suc take-hd-drop*)  
  **done**  
**qed** *simp*

**lemma** *path-singleton*[*intro, simp*]:  $s \in_A \text{weights } l1 \ l2 \implies \text{path } l1 \ l2 \ [(l1, s, l2)]$   
**by** (*subst path.simps*) (*auto simp: path.intros*)

**lemma** *path-appendE*:  $\text{path } l1 \ l3 \ (xs \ @ \ ys) \implies \exists \ l2. \ \text{path } l2 \ l3 \ ys \wedge \ \text{path } l1 \ l2 \ xs$   
**proof** (*induct l1 l3 xs@ys arbitrary: xs ys rule: path.induct*)  
  **case** (*path0 l1 l2*)  
  **then show** *?case* **by** (*auto intro: path.intros*)  
**next**  
  **case** (*path l1 l2 xs lbl l3 xs' ys'*)  
  **from** *path(1,3-)* **show** *?case*  
  **apply** –  
  **apply** (*subst (asm) append-eq-append-conv2[of xs [(l2, lbl, l3)] xs' ys']*)  
  **apply** (*elim exE conjE disjE*)  
  **subgoal for** *us*  
  **using** *path(2)[of xs' us]*  
  **by** (*auto intro: path.intros*)  
  **subgoal for** *us*  
  **by** (*cases us=[]*) (*auto intro: path.intros simp: Cons-eq-append-conv*)  
  **done**  
**qed**

**lemma** *path-replace-prefix*:  
 $\text{path } l1 \ l3 \ (xs \ @ \ zs) \implies \text{path } l1 \ l2 \ ys \implies \text{path } l1 \ l3 \ (ys \ @ \ zs)$   
**by** (*drule path-appendE*) (*auto elim!: path-trans dest: path-determines-loc*)

**lemma** *drop-subseq*:  $n \leq \text{length } xs \implies \text{drop } n \ xs \preceq xs$   
**by** (*auto simp: suffix-def intro!: exI[of - take n xs]*)

**lemma** *take-subseq*[*simp, intro*]:  $\text{take } n \ xs \preceq xs$

```

by (induct xs) auto

lemma map-take-subseq[simp, intro]: map f (take n xs)  $\preceq$  map f xs
by (rule subseq-map, induct xs) auto

lemma path-distinct:
  path l1 l2 xs  $\implies$   $\exists$  xs'. distinct xs'  $\wedge$  path l1 l2 xs'  $\wedge$  map LBL xs'  $\preceq$  map LBL xs
proof (induct rule: path.induct)
  case (path0 l1 l2)
  then show ?case
    by (intro exI[of - []]) (auto intro: path.intros)
next
  case (path l1 l2 xs lbl l3)
  then obtain xs' where ih: path l1 l2 xs' distinct xs' map LBL xs'  $\preceq$  map LBL
  xs
    by blast
  then show ?case
proof (cases (l2, lbl, l3)  $\in$  set xs')
  case True
  then obtain m where m: m < length xs'  $\wedge$  m = (l2, lbl, l3)
    unfolding in-set-conv-nth by blast
  from m ih have path l1 l2 (take m xs')
    by (auto intro: path-take-from)
  with m ih path show ?thesis
    apply (intro exI[of - take m xs' @ [(l2, lbl, l3)]])
    apply (rule conjI)
    apply (metis distinct-take take-Suc-conv-app-nth)
    apply (rule conjI)
    apply (rule path.intros)
    apply simp
    apply simp
    apply simp
    apply (metis ih(3) subseq-order.trans take-map take-subseq)
  done
next
  case False
  with ih path(3) show ?thesis
    by (auto intro!: exI[of - xs' @ [(l2, lbl, l3)]] intro: path.intros)
qed
qed

```

```

lemma path-edge: (l1', lbl, l2')  $\in$  set xs  $\implies$  path l1 l2 xs  $\implies$  lbl  $\in_A$  weights l1'
l2'
by (rotate-tac, induct rule: path.induct) auto

```

## 6.2 Path Weights

```

abbreviation sum-weights :: 'lbl list  $\Rightarrow$  'lbl where
  sum-weights xs  $\equiv$  foldr (+) xs 0

```

**abbreviation**  $sum\text{-}path\text{-}weights\ xs \equiv sum\text{-}weights\ (map\ LBL\ xs)$

**definition**  $path\text{-}weightp\ l1\ l2\ s \equiv (\exists\ xs.\ path\ l1\ l2\ xs \wedge s = sum\text{-}path\text{-}weights\ xs)$

**lemma**  $sum\text{-}not\text{-}less\text{-}zero[simp, dest]: (s::'lbl) < 0 \implies False$   
**by** ( $simp\ add: less\text{-}le\text{-}not\text{-}le$ )

**lemma**  $sum\text{-}le\text{-}zero[simp]: (s::'lbl) \leq 0 \longleftrightarrow s = 0$   
**by** ( $simp\ add: eq\text{-}iff$ )

**lemma**  $sum\text{-}le\text{-}zeroD[dest]: (x::'lbl) \leq 0 \implies x = 0$   
**by**  $simp$

**lemma**  $foldr\text{-}plus\text{-}mono: (n::'lbl) \leq m \implies foldr\ (+)\ xs\ n \leq foldr\ (+)\ xs\ m$   
**by** ( $induct\ xs$ ) ( $auto\ simp: plus\text{-}mono$ )

**lemma**  $sum\text{-}weights\text{-}append:$   
 $sum\text{-}weights\ (ys\ @\ xs) = sum\text{-}weights\ ys + sum\text{-}weights\ xs$   
**by** ( $induct\ ys$ ) ( $auto\ simp: add.\text{assoc}$ )

**lemma**  $sum\text{-}summary\text{-}prepend\text{-}le: sum\text{-}path\text{-}weights\ ys \leq sum\text{-}path\text{-}weights\ xs \implies$   
 $sum\text{-}path\text{-}weights\ (zs\ @\ ys) \leq sum\text{-}path\text{-}weights\ (zs\ @\ xs)$   
**by** ( $induct\ zs\ arbitrary: xs\ ys$ ) ( $auto\ intro: plus\text{-}mono$ )

**lemma**  $sum\text{-}summary\text{-}append\text{-}le: sum\text{-}path\text{-}weights\ ys \leq sum\text{-}path\text{-}weights\ xs \implies$   
 $sum\text{-}path\text{-}weights\ (ys\ @\ zs) \leq sum\text{-}path\text{-}weights\ (xs\ @\ zs)$

**proof** ( $induct\ zs\ arbitrary: xs\ ys$ )

**case** ( $Cons\ a\ zs$ )

**then show**  $?case$

**by** ( $metis\ plus\text{-}mono\ map\text{-}append\ order\text{-}refl\ sum\text{-}weights\text{-}append$ )

**qed**  $simp$

**lemma**  $foldr\text{-}plus\text{-}zero\text{-}le: foldr\ (+)\ xs\ (0::'lbl) \leq foldr\ (+)\ xs\ a$   
**by** ( $induct\ xs$ ) ( $simp\text{-}all\ add: plus\text{-}mono$ )

**lemma**  $subseq\text{-}sum\text{-}weights\text{-}le:$

**assumes**  $xs \preceq ys$

**shows**  $sum\text{-}weights\ xs \leq sum\text{-}weights\ ys$

**using**  $assms$

**proof** ( $induct\ rule: list\text{-}emb.\text{induct}$ )

**case** ( $list\text{-}emb\text{-}Nil\ ys$ )

**then show**  $?case$  **by**  $auto$

**next**

**case** ( $list\text{-}emb\text{-}Cons\ xs\ ys\ y$ )

**then show**  $?case$  **by** ( $auto\ elim!: order\text{-}trans\ simp: le\text{-}plus$ )

**next**

**case** ( $list\text{-}emb\text{-}Cons2\ x\ y\ xs\ ys$ )

**then show**  $?case$  **by** ( $auto\ elim!: order\text{-}trans\ simp: plus\text{-}mono$ )

**qed**

**lemma** *subseq-sum-path-weights-le*:

*map LBL xs*  $\preceq$  *map LBL ys*  $\implies$  *sum-path-weights xs*  $\leq$  *sum-path-weights ys*  
**by** (*rule subseq-sum-weights-le*)

**lemma** *sum-path-weights-take-le*[*simp, intro*]: *sum-path-weights (take i xs)*  $\leq$  *sum-path-weights xs*

**by** (*auto intro!*: *subseq-sum-path-weights-le*)

**lemma** *sum-weights-append-singleton*:

*sum-weights (xs @ [x])* = *sum-weights xs* + *x*  
**by** (*induct xs*) (*simp-all add: add.assoc*)

**lemma** *sum-path-weights-append-singleton*:

*sum-path-weights (xs @ [(l,x,l')])* = *sum-path-weights xs* + *x*  
**by** (*induct xs*) (*simp-all add: add.assoc*)

**lemma** *path-weightp-ex-path*:

*path-weightp l1 l2 s*  $\implies \exists xs.$   
(*let s' = sum-path-weights xs in s'  $\leq$  s  $\wedge$  path-weightp l1 l2 s'  $\wedge$  distinct xs  $\wedge$*   
( $\forall (l1,s,l2) \in \text{set } xs. s \in_A \text{weights } l1 \ l2$ ))

**unfolding** *path-weightp-def*

**apply** (*erule exE conjE*)+

**apply** (*drule path-distinct*)

**apply** (*erule exE conjE*)+

**subgoal for** *xs xs'*

**apply** (*rule exI[of - xs']*)

**apply** (*auto simp: Let-def dest!: path-edge intro: subseq-sum-path-weights-le*)

**done**

**done**

**lemma** *finite-set-summaries*:

*finite (( $\lambda((l1,l2),s). (l1,s,l2)$ ) ' (*Sigma UNIV* ( $\lambda(l1,l2). \text{set-antichain (weights } l1 \ l2)$ ))))*

**by** *force*

**lemma** *finite-summaries*: *finite {xs. distinct xs  $\wedge$  ( $\forall (l1, s, l2) \in \text{set } xs. s \in_A \text{weights } l1 \ l2$ )}*

**apply** (*rule finite-subset[OF - finite-subset-distinct[of (( $\lambda((l1,l2),s). (l1,s,l2)$ ) ' (*Sigma UNIV* ( $\lambda(l1,l2). \text{set-antichain (weights } l1 \ l2)$ ))))]]*)

**apply** (*force simp: finite-set-summaries*)+

**done**

**lemma** *finite-minimal-antichain-path-weightp*:

*finite (minimal-antichain {x. path-weightp l1 l2 x})*

**apply** (*rule finite-surj[OF finite-summaries, where f = sum-path-weights]*)

**apply** (*clarsimp simp: minimal-antichain-def image-iff dest!: path-weightp-ex-path*)

**apply** (*fastforce simp: Let-def*)

**done**

**lift-definition** *path-weight* :: 'vtx  $\Rightarrow$  'vtx  $\Rightarrow$  'lbl antichain  
 is  $\lambda l1\ l2. \text{minimal-antichain } \{x. \text{path-weightp } l1\ l2\ x\}$   
 using *finite-minimal-antichain-path-weightp*  
 by *auto*

**definition** *reachable*  $l1\ l2 \equiv \text{path-weight } l1\ l2 \neq \{\}_A$

**lemma** *in-path-weight*:  $s \in_A \text{path-weight } loc1\ loc2 \longleftrightarrow s \in \text{minimal-antichain } \{s. \text{path-weightp } loc1\ loc2\ s\}$   
 by *transfer simp*

**lemma** *path-weight-refl*[*simp*]:  $0 \in_A \text{path-weight } loc\ loc$

**proof** –

have \*: *path*  $loc\ loc$  []  
 by (*simp add: path0*)  
 then have  $0 = \text{sum-path-weights } []$  by *auto*  
 with \* have *path-weightp*  $loc\ loc\ 0$   
 using *path-weightp-def* by *blast*  
 then show ?thesis  
 by (*auto simp: in-path-weight in-minimal-antichain*)

qed

**lemma** *zero-in-minimal-antichain*[*simp*]:  $(0::'lbl) \in S \Longrightarrow 0 \in \text{minimal-antichain } S$

by (*auto simp: in-minimal-antichain intro: sum-not-less-zero*)

**definition** *path-weightp-distinct*  $l1\ l2\ s \equiv (\exists xs. \text{distinct } xs \wedge \text{path } l1\ l2\ xs \wedge s = \text{sum-path-weights } xs)$

**lemma** *minimal-antichain-path-weightp-distinct*:

$\text{minimal-antichain } \{xs. \text{path-weightp } l1\ l2\ xs\} = \text{minimal-antichain } \{xs. \text{path-weightp-distinct } l1\ l2\ xs\}$

**unfolding** *path-weightp-def path-weightp-distinct-def minimal-antichain-def*  
**apply** *safe*  
**apply** *clarsimp*  
**apply** (*metis path-distinct order.strict-iff-order subseq-sum-path-weights-le*)  
**apply** (*blast+*) [2]  
**apply** *clarsimp*  
**apply** (*metis (no-types, lifting) le-less-trans path-distinct subseq-sum-weights-le*)  
**done**

**lemma** *finite-path-weightp-distinct*[*simp, intro*]:  $\text{finite } \{xs. \text{path-weightp-distinct } l1\ l2\ xs\}$

**unfolding** *path-weightp-distinct-def*  
**apply** (*rule finite-subset*[**where**  $B = \text{sum-path-weights } \{xs. \text{distinct } xs \wedge \text{path } l1\ l2\ xs\}$ ])  
**apply** *clarsimp*

```

apply (rule finite-imageI)
apply (rule finite-subset[OF - finite-summaries])
apply (clarsimp simp: path-edge)
done

```

**lemma** *path-weightp-distinct-nonempty*:

```

{xs. path-weightp l1 l2 xs} ≠ {} ↔ {xs. path-weightp-distinct l1 l2 xs} ≠ {}
by (auto dest: path-distinct simp: path-weightp-def path-weightp-distinct-def)

```

**lemma** *path-weightp-distinct-member*:

```

s ∈ {s. path-weightp l1 l2 s} ⇒ ∃ u. u ∈ {s. path-weightp-distinct l1 l2 s} ∧ u ≤ s
apply (clarsimp simp: path-weightp-def path-weightp-distinct-def)
apply (drule path-distinct)
apply (auto dest: subseq-sum-path-weights-le)
done

```

**lemma** *minimal-antichain-path-weightp-member*:

```

s ∈ {xs. path-weightp l1 l2 xs} ⇒ ∃ u. u ∈ minimal-antichain {xs. path-weightp l1 l2 xs} ∧ u ≤ s

```

**proof** –

```

assume s ∈ {xs. path-weightp l1 l2 xs}
then obtain u where u: u ∈ {s. path-weightp-distinct l1 l2 s} ∧ u ≤ s
using path-weightp-distinct-member by blast
have finite: finite {xs. path-weightp-distinct l1 l2 xs} ..
from u finite obtain v where v ∈ minimal-antichain {xs. path-weightp-distinct l1 l2 xs} ∧ v ≤ u
by atomize-elim (auto intro: minimal-antichain-member)
with u show ?thesis
by (auto simp: minimal-antichain-path-weightp-distinct)
qed

```

**lemma** *path-path-weight*: path l1 l2 xs ⇒ ∃ s. s ∈<sub>A</sub> path-weight l1 l2 ∧ s ≤ sum-path-weights xs

**proof** –

```

assume path l1 l2 xs
then have sum-path-weights xs ∈ {x. path-weightp l1 l2 x}
by (auto simp: path-weightp-def)
then obtain u where u ∈ minimal-antichain {x. path-weightp l1 l2 x} ∧ u ≤ sum-path-weights xs
apply atomize-elim
apply (drule minimal-antichain-path-weightp-member)
apply auto
done
then show ?thesis
by transfer auto
qed

```

**lemma** *path-weight-conv-path*:

$s \in_A \text{path-weight } l1 \ l2 \implies \exists xs. \text{path } l1 \ l2 \ xs \wedge s = \text{sum-path-weights } xs \wedge (\forall ys. \text{path } l1 \ l2 \ ys \longrightarrow \neg \text{sum-path-weights } ys < \text{sum-path-weights } xs)$

**by** *transfer (auto simp: in-minimal-antichain path-weightp-def)*

**abbreviation**  $\text{optimal-path } loc1 \ loc2 \ xs \equiv \text{path } loc1 \ loc2 \ xs \wedge (\forall ys. \text{path } loc1 \ loc2 \ ys \longrightarrow \neg \text{sum-path-weights } ys < \text{sum-path-weights } xs)$

**lemma** *path-weight-path*:  $s \in_A \text{path-weight } loc1 \ loc2 \implies (\bigwedge xs. \text{optimal-path } loc1 \ loc2 \ xs \implies \text{distinct } xs \implies \text{sum-path-weights } xs = s \implies P) \implies P$

**apply** *atomize-elim*

**apply** *transfer*

**apply** *(clarsimp simp: in-minimal-antichain path-weightp-def)*

**apply** *(drule path-distinct)*

**apply** *(erule exE)*

**subgoal for**  $loc1 \ loc2 \ xs \ xs'$

**apply** *(rule exI[of - xs'])*

**apply** *safe*

**using** *order.strict-iff-order subseq-sum-path-weights-le* **apply** *metis*

**using** *less-le subseq-sum-path-weights-le* **apply** *fastforce*

**done**

**done**

**lemma** *path-weight-elem-trans*:

$s \in_A \text{path-weight } l1 \ l2 \implies s' \in_A \text{path-weight } l2 \ l3 \implies \exists u. u \in_A \text{path-weight } l1 \ l3 \wedge u \leq s + s'$

**proof** –

**assume**  $ps1: s \in_A \text{path-weight } l1 \ l2$

**assume**  $ps2: s' \in_A \text{path-weight } l2 \ l3$

**from**  $ps1$  **obtain**  $xs$  **where**  $path1: \text{path } l1 \ l2 \ xs \ s = \text{sum-path-weights } xs$

**by** *(auto intro: path-weight-path)*

**from**  $ps2$  **obtain**  $ys$  **where**  $path2: \text{path } l2 \ l3 \ ys \ s' = \text{sum-path-weights } ys$

**by** *(auto intro: path-weight-path)*

**from**  $path1(1) \ path2(1)$  **have**  $\text{path } l1 \ l3 \ (xs \ @ \ ys)$

**by** *(rule path-trans)*

**with**  $path1(2) \ path2(2)$  **have**  $s + s' \in \{s. \text{path-weightp } l1 \ l3 \ s\}$

**by** *(auto simp: path-weightp-def sum-weights-append[symmetric])*

**then show**  $\exists u. u \in_A \text{path-weight } l1 \ l3 \wedge u \leq s + s'$

**by** *transfer (simp add: minimal-antichain-path-weightp-member)*

**qed**

**end**

## 7 Local Progress Propagation

### 7.1 Specification

**record** (overloaded)  $(loc, 't)$  *configuration* =

*c-work* :: 'loc  $\Rightarrow$  't zmultiset  
*c-pts* :: 'loc  $\Rightarrow$  't zmultiset  
*c-imp* :: 'loc  $\Rightarrow$  't zmultiset

**type-synonym** ('loc, 't) *computation* = ('loc, 't) *configuration stream*

**locale** *dataflow-topology* = *flow?*: *graph summary*  
**for** *summary* :: 'loc  $\Rightarrow$  'loc :: *finite*  $\Rightarrow$  'sum :: {*order*, *monoid-add*} *antichain* +  
**fixes** *results-in* :: 't :: *order*  $\Rightarrow$  'sum  $\Rightarrow$  't  
**assumes** *results-in-zero*: *results-in* t 0 = t  
**and** *results-in-mono-raw*:  $t1 \leq t2 \implies s1 \leq s2 \implies \text{results-in } t1 \ s1 \leq \text{results-in } t2 \ s2$   
**and** *followed-by-summary*:  $\text{results-in } (\text{results-in } t \ s1) \ s2 = \text{results-in } t \ (s1 + s2)$   
**and** *no-zero-cycle*:  $\text{path } loc \ loc \ xs \implies xs \neq [] \implies s = \text{sum-path-weights } xs \implies t < \text{results-in } t \ s$   
**begin**

**lemma** *results-in-mono*:  
 $t1 \leq t2 \implies \text{results-in } t1 \ s \leq \text{results-in } t2 \ s$   
 $s1 \leq s2 \implies \text{results-in } t \ s1 \leq \text{results-in } t \ s2$   
**using** *results-in-mono-raw* **by** *auto*

**abbreviation** *path-summary*  $\equiv$  *path-weight*  
**abbreviation** *followed-by* :: 'sum  $\Rightarrow$  'sum  $\Rightarrow$  'sum **where**  
*followed-by*  $\equiv$  *plus*

**definition** *safe* :: ('loc, 't) *configuration*  $\Rightarrow$  *bool* **where**  
*safe* c  $\equiv \forall loc1 \ loc2 \ t \ s. \text{zcount } (c\text{-pts } c \ loc1) \ t > 0 \wedge s \in_A \text{path-summary } loc1 \ loc2$   
 $\longrightarrow (\exists t' \leq \text{results-in } t \ s. t' \in_A \text{frontier } (c\text{-imp } c \ loc2))$

Implications are always non-negative.

**definition** *inv-implications-nonneg* **where**  
*inv-implications-nonneg* c =  $(\forall loc \ t. \text{zcount } (c\text{-imp } c \ loc) \ t \geq 0)$

**abbreviation** *unchanged* f c0 c1  $\equiv$  f c1 = f c0

**abbreviation** *zmsset-frontier* **where**  
*zmsset-frontier* M  $\equiv \text{zmsset-of } (mset\text{-set } (\text{set-antichain } (\text{frontier } M)))$

**definition** *init-config* **where**  
*init-config* c  $\equiv \forall loc.$   
*c-imp* c loc =  $\{\#\}_z \wedge$   
*c-work* c loc = *zmsset-frontier* (c-pts c loc)

**definition** *after-summary* :: 't zmultiset  $\Rightarrow$  'sum *antichain*  $\Rightarrow$  't zmultiset **where**  
*after-summary* M S  $\equiv (\sum s \in \text{set-antichain } S. \text{image-zmsset } (\lambda t. \text{results-in } t \ s))$

M)

**abbreviation** *frontier-changes* :: 't zmultiset  $\Rightarrow$  't zmultiset  $\Rightarrow$  't zmultiset **where**  
*frontier-changes* M N  $\equiv$  zmsset-frontier M - zmsset-frontier N

**definition** *next-change-multiplicity'* :: ('loc, 't) configuration  $\Rightarrow$  ('loc, 't) configuration  $\Rightarrow$  'loc  $\Rightarrow$  't  $\Rightarrow$  int  $\Rightarrow$  bool **where**

*next-change-multiplicity'* c0 c1 loc t n  $\equiv$

— n is the non-zero change in pointstamps at loc for timestamp t

$n \neq 0 \wedge$

— change can only happen at timestamps not in advance of implication-frontier

$(\exists t'. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ loc}) \wedge t' \leq t) \wedge$

— at loc, t is added to pointstamps n times

$c1 = c0 \langle c\text{-pts} := (c\text{-pts } c0)(\text{loc} := \text{update-zmultiset } (c\text{-pts } c0 \text{ loc}) t n),$

— worklist at loc is adjusted by frontier changes

$c\text{-work} := (c\text{-work } c0)(\text{loc} := c\text{-work } c0 \text{ loc} +$

$\text{frontier-changes } (\text{update-zmultiset } (c\text{-pts } c0 \text{ loc}) t n) (c\text{-pts } c0 \text{ loc})) \rangle$

**abbreviation** *next-change-multiplicity* :: ('loc, 't) configuration  $\Rightarrow$  ('loc, 't) configuration  $\Rightarrow$  bool **where**

*next-change-multiplicity* c0 c1  $\equiv \exists \text{loc } t n. \text{next-change-multiplicity}' c0 c1 \text{ loc } t n$

**lemma** *cm-unchanged-worklist*:

**assumes** *next-change-multiplicity'* c0 c1 loc t n

**and**  $\text{loc}' \neq \text{loc}$

**shows**  $c\text{-work } c1 \text{ loc}' = c\text{-work } c0 \text{ loc}'$

**using** *assms* **unfolding** *next-change-multiplicity'-def*

**by** *auto*

**definition** *next-propagate'* :: ('loc, 't) configuration  $\Rightarrow$  ('loc, 't) configuration  $\Rightarrow$  'loc  $\Rightarrow$  't  $\Rightarrow$  bool **where**

*next-propagate'* c0 c1 loc t  $\equiv$

— t is a least timestamp of all worklist entries

$(t \in \#_z c\text{-work } c0 \text{ loc} \wedge$

$(\forall t' \text{ loc}'. t' \in \#_z c\text{-work } c0 \text{ loc}' \longrightarrow \neg t' < t) \wedge$

$c1 = c0 \langle c\text{-imp} := (c\text{-imp } c0)(\text{loc} := c\text{-imp } c0 \text{ loc} + \{\#t' \in \#_z c\text{-work } c0 \text{ loc}. t' = t\# \}),$

$c\text{-work} := (\lambda \text{loc}'.$

— worklist entries for t are removed from loc's worklist

$\text{if } \text{loc}' = \text{loc} \text{ then } \{\#t' \in \#_z c\text{-work } c0 \text{ loc}'. t' \neq t\# \}$

— worklists at other locations change by the loc's frontier change

after adding summaries

$\text{else } c\text{-work } c0 \text{ loc}'$

$+ \text{after-summary}$

$(\text{frontier-changes } (c\text{-imp } c0 \text{ loc} + \{\#t' \in \#_z c\text{-work } c0$

$\text{loc}. t' = t\# \}) (c\text{-imp } c0 \text{ loc}))$

$(\text{summary } \text{loc } \text{loc}') \rangle \rangle$

**abbreviation** *next-propagate* :: ('loc, 't :: order) configuration  $\Rightarrow$  ('loc, 't) config-

uration  $\Rightarrow$  bool **where**  
*next-propagate*  $c0\ c1 \equiv \exists\ loc\ t.\ next-propagate'\ c0\ c1\ loc\ t$

**definition** *next'* **where**  
*next'*  $c0\ c1 = (next-propagate\ c0\ c1 \vee next-change-multiplicity\ c0\ c1 \vee c1 = c0)$

**abbreviation** *next* **where**  
*next*  $s \equiv next'\ (shd\ s)\ (shd\ (stl\ s))$

**abbreviation** *cm-valid* **where**  
*cm-valid*  $\equiv nxt\ (\lambda s.\ next-change-multiplicity\ (shd\ s)\ (shd\ (stl\ s)))\ impl$   
 $(\lambda s.\ next-change-multiplicity\ (shd\ s)\ (shd\ (stl\ s)))\ or\ nxt\ (holds\ (\lambda c.$   
 $(\forall l.\ c-work\ c\ l = \{\#\}_z)))$

**definition** *spec* :: ('loc, 't :: order) computation  $\Rightarrow$  bool **where**  
*spec*  $\equiv holds\ init-config\ aand\ alw\ next$

**lemma** *next'-inv*[consumes 1, case-names *next-change-multiplicity next-propagate next-finish-init*]:

**assumes** *next'*  $c0\ c1\ P\ c0$   
**and**  $\bigwedge loc\ t\ n.\ P\ c0 \Longrightarrow next-change-multiplicity'\ c0\ c1\ loc\ t\ n \Longrightarrow P\ c1$   
**and**  $\bigwedge loc\ t.\ P\ c0 \Longrightarrow next-propagate'\ c0\ c1\ loc\ t \Longrightarrow P\ c1$   
**shows**  $P\ c1$   
**using** *assms* **unfolding** *next'-def* **by** *auto*

## 7.2 Auxiliary

**lemma** *next-change-multiplicity'-unique*:

**assumes**  $n \neq 0$   
**and**  $\exists t'. t' \in_A frontier\ (c-imp\ c\ loc) \wedge t' \leq t$   
**shows**  $\exists! c'. next-change-multiplicity'\ c\ c'\ loc\ t\ n$

**proof** –

**let**  $?pointstamps' = (c-pts\ c)(loc := update-zmultiset\ (c-pts\ c\ loc)\ t\ n)$   
**let**  $?worklist' = \lambda loc'. c-work\ c\ loc' + frontier-changes\ (?pointstamps'\ loc')\ (c-pts\ c\ loc')$   
**let**  $?c' = c(c-pts := ?pointstamps', c-work := ?worklist')$   
**from** *assms* **have** *next-change-multiplicity'*  $c\ ?c'\ loc\ t\ n$   
**by** (*auto simp: next-change-multiplicity'-def intro!: configuration.equality*)  
**then show** *?thesis*  
**by** (*rule exII[of - ?c']*)  
 $(auto\ simp: next-change-multiplicity'-def intro!: configuration.equality)$

**qed**

**lemma** *frontier-change-zmset-frontier*:

**assumes**  $t \in_A frontier\ M1 - frontier\ M0$   
**shows**  $zcount\ (zmset-frontier\ M1)\ t = 1 \wedge zcount\ (zmset-frontier\ M0)\ t = 0$   
**using** *assms* **by** (*clarsimp simp: ac-Diff-iff*) (*simp add: member-antichain.rep-eq*)

**lemma** *frontier-empty*[*simp*]: *frontier*  $\{\#\}_z = \{\}_A$

by *transfer' simp*

**lemma** *zmsset-frontier-empty[simp]*: *zmsset-frontier*  $\{\#\}_z = \{\#\}_z$   
by *simp*

**lemma** *after-summary-empty[simp]*: *after-summary*  $\{\#\}_z S = \{\#\}_z$   
by (*simp add: after-summary-def*)

**lemma** *after-summary-empty-summary[simp]*: *after-summary*  $M \{\}_A = \{\#\}_z$   
by (*simp add: after-summary-def*)

**lemma** *mem-frontier-diff*:  
assumes  $t \in_A \text{frontier } M - \text{frontier } N$   
shows  $\text{zcount } (\text{frontier-changes } M N) t = 1$   
**proof** –  
note *assms*  
then have  $t: t \in_A \text{frontier } M \ t \notin_A \text{frontier } N$   
using *ac-Diff-iff* by *blast+*  
from  $t(1)$  have  $\text{zcount } (\text{zmsset-frontier } M) t = 1$   
by (*simp add: member-antichain.rep-eq*)  
moreover from  $t(2)$  have  $\text{zcount } (\text{zmsset-frontier } N) t = 0$   
by (*simp add: member-antichain.rep-eq*)  
ultimately show  $\text{zcount } (\text{frontier-changes } M N) t = 1$   
by *simp*  
**qed**

**lemma** *mem-frontier-diff'*:  
assumes  $t \in_A \text{frontier } N - \text{frontier } M$   
shows  $\text{zcount } (\text{frontier-changes } M N) t = -1$   
**proof** –  
note *assms*  
then have  $t: t \in_A \text{frontier } N \ t \notin_A \text{frontier } M$   
using *ac-Diff-iff* by *blast+*  
from  $t(2)$  have  $\text{zcount } (\text{zmsset-frontier } M) t = 0$   
by (*simp add: member-antichain.rep-eq*)  
moreover from  $t(1)$  have  $\text{zcount } (\text{zmsset-frontier } N) t = 1$   
by (*simp add: member-antichain.rep-eq*)  
ultimately show  $\text{zcount } (\text{frontier-changes } M N) t = -1$   
by *simp*  
**qed**

**lemma** *not-mem-frontier-diff*:  
assumes  $t \notin_A \text{frontier } M - \text{frontier } N$   
and  $t \notin_A \text{frontier } N - \text{frontier } M$   
shows  $\text{zcount } (\text{frontier-changes } M N) t = 0$   
**proof** –  
{ assume  $M: t \in_A \text{frontier } M$   
with *assms* have  $N: t \in_A \text{frontier } N$   
by (*auto dest: ac-notin-Diff*)

```

from  $M N$  have  $zcount (zmsset-frontier M) t = 1 zcount (zmsset-frontier N) t$ 
= 1
  by (simp-all add: member-antichain.rep-eq)
  then have  $zcount (frontier-changes M N) t = 0$ 
  by simp
}
moreover
{ assume  $M: t \notin_A frontier M$ 
  with assms have  $N: t \notin_A frontier N$ 
  by (auto dest: ac-notin-Diff)
  from  $M N$  have  $zcount (zmsset-frontier M) t = 0 zcount (zmsset-frontier N) t$ 
= 0
  by (simp-all add: member-antichain.rep-eq)
  then have  $zcount (frontier-changes M N) t = 0$ 
  by simp
}
ultimately show  $zcount (frontier-changes M N) t = 0$ 
by blast
qed

```

**lemma** *mset-neg-after-summary*:  $mset-neg M = \{\#\} \implies mset-neg (after-summary M S) = \{\#\}$   
**by** (*auto intro: mset-neg-image-zmset mset-neg-sum-set simp: after-summary-def*)

— Changes in loc’s frontier are reflected in the worklist of loc’.

**lemma** *next-p-frontier-change*:  
**assumes** *next-propagate' c0 c1 loc t*  
**and** *summary loc loc'  $\neq \{\}_A$*   
**shows**  $c-work c1 loc' =$   
 $c-work c0 loc'$   
 $+ after-summary$   
 $(frontier-changes (c-imp c1 loc) (c-imp c0 loc))$   
 $(summary loc loc')$   
**using** *assms* **by** (*auto simp: summary-self next-propagate'-def intro!: configuration.equality*)

**lemma** *after-summary-union*:  $after-summary (M + N) S = after-summary M S$   
 $+ after-summary N S$   
**by** (*simp add: sum.distrib after-summary-def*)

## 7.3 Invariants

### 7.3.1 Invariant: *inv-imps-work-sum*

**abbreviation** *union-frontiers* ::  $('loc, 't) configuration \Rightarrow 'loc \Rightarrow 't zmultiset$  **where**  
 $union-frontiers c loc \equiv$   
 $(\sum loc' \in UNIV. after-summary (zmsset-frontier (c-imp c loc')) (summary loc'$   
 $loc))$

— Implications + worklist is equal to the frontiers of pointstamps and all preceding

nodes (after accounting for summaries).

**definition** *inv-imps-work-sum* :: ('loc, 't) configuration  $\Rightarrow$  bool **where**

*inv-imps-work-sum* c  $\equiv$   
 $\forall$  loc. c-imp c loc + c-work c loc  
 $=$  zmsset-frontier (c-pts c loc) + union-frontiers c loc

— Version with zcount is easier to reason with

**definition** *inv-imps-work-sum-zcount* :: ('loc, 't) configuration  $\Rightarrow$  bool **where**

*inv-imps-work-sum-zcount* c  $\equiv$   
 $\forall$  loc t. zcount (c-imp c loc + c-work c loc) t  
 $=$  zcount (zmsset-frontier (c-pts c loc) + union-frontiers c loc) t

**lemma** *inv-imps-work-sum-zcount*: *inv-imps-work-sum* c  $\longleftrightarrow$  *inv-imps-work-sum-zcount* c

**unfolding** *inv-imps-work-sum-zcount-def* *inv-imps-work-sum-def*  
**by** (*simp add: zmultiset-eq-iff*)

**lemma** *union-frontiers-nonneg*:  $0 \leq$  zcount (union-frontiers c loc) t

**apply** (*subst zcount-sum*)  
**apply** (*rule sum-nonneg*)  
**apply** *simp*  
**apply** (*rule mset-neg-zcount-nonneg*)  
**apply** (*rule mset-neg-after-summary*)  
**apply** *simp*  
**done**

**lemma** *next-p-union-frontier-change*:

**assumes** *next-propagate'* c0 c1 loc t  
**and** *summary* loc loc'  $\neq$  {}<sub>A</sub>  
**shows** *union-frontiers* c1 loc' =  
*union-frontiers* c0 loc'  
 $+$  *after-summary*  
(*frontier-changes* (c-imp c1 loc) (c-imp c0 loc))  
(*summary* loc loc')

**using** *assms*

**apply** (*subst zmultiset-eq-iff*)

**apply** (*rule allI*)

**subgoal for** x

**apply** (*simp del: zcount-of-mset image-zmsset-Diff*)

**apply** (*subst* (1 2) *zcount-sum*)

**apply** (*rule Sum-eq-pick-changed-elem*[of UNIV loc])

**apply** *simp*

**apply** *simp*

**subgoal**

**apply** (*subst zcount-union*[*symmetric*])

**apply** (*subst after-summary-union*[*symmetric*])

**apply** *simp*

**done**

```

    apply (auto simp: next-propagate'-def)
  done
done

— init-config satisfies inv-imps-work-sum
lemma init-imp-inv-imps-work-sum: init-config c  $\implies$  inv-imps-work-sum c
  by (simp add: inv-imps-work-sum-def init-config-def)

— CM preserves inv-imps-work-sum
lemma cm-preserves-inv-imps-work-sum:
  assumes next-change-multiplicity' c0 c1 loc t n
    and inv-imps-work-sum c0
  shows inv-imps-work-sum c1
proof —
  — Given CM at loc, t, we show result for loc', t'
  { fix loc t loc' t' n
    assume cm': next-change-multiplicity' c0 c1 loc t n
    note cm = this[unfolded next-change-multiplicity'-def]
    from cm have unchanged-imps: unchanged c-imp c0 c1
      by simp
    assume inv-imps-work-sum c0
    then have iiws': inv-imps-work-sum-zcount c0
      by (simp add: inv-imps-work-sum-zcount)
    note iiws = iiws'[unfolded inv-imps-work-sum-zcount-def, THEN spec2]
    have unchanged-union: union-frontiers c1 locX = union-frontiers c0 locX for
locX
      using unchanged-imps by (auto intro: sum.cong)
      — For locations other than loc nothing changes.
    { assume loc: loc'  $\neq$  loc
      note iiws = iiws'[unfolded inv-imps-work-sum-zcount-def, THEN spec2, of
loc' t]
      from loc cm have unchanged-worklist:
        zcount (c-work c1 loc') t' = zcount (c-work c0 loc') t'
        by simp
      from loc cm have unchanged-frontier:
        zcount (zmsset-frontier (c-pts c1 loc')) t'
         $=$  zcount (zmsset-frontier (c-pts c0 loc')) t'
        by simp
      with loc have
        zcount (c-imp c1 loc' + c-work c1 loc') t'
         $=$  zcount (zmsset-frontier (c-pts c1 loc')
           $+$  union-frontiers c1 loc') t'
      apply (subst (1 2) zcount-union)
      unfolding
        unchanged-imps
        unchanged-union
        unchanged-frontier
        unchanged-worklist
      apply (subst (1 2) zcount-union[symmetric])

```

```

    apply (rule iiws)
  done
}
moreover
  — For pointstamps at location loc we make a case distinction on whether their
  "status" in the frontier has changed.
  { assume loc: loc' = loc
    note iiws = iiws["unfolded inv-imps-work-sum-zcount-def, simplified, THEN
spec, of loc, simplified"]
    — If t appeared in the frontier
    { assume t':  $t' \in_A \text{frontier } (c\text{-pts } c1 \text{ loc}) - \text{frontier } (c\text{-pts } c0 \text{ loc})$ 
      note t'["THEN mem-frontier-diff"]
      — then the worklist at t increased by 1
      then have  $zcount (c\text{-work } c1 \text{ loc}) t' = zcount (c\text{-work } c0 \text{ loc}) t' + 1$ 
        using cm by auto
        — and the frontier at t increased by 1
      moreover
      have  $zcount (z\text{mset-frontier } (c\text{-pts } c1 \text{ loc})) t'$ 
        =  $zcount (z\text{mset-frontier } (c\text{-pts } c0 \text{ loc})) t' + 1$ 
        using t'["THEN frontier-change-zmset-frontier"] by simp
        — and the sum didn't change
      moreover note unchanged-union
        — hence, the invariant is preserved.
      ultimately have
         $zcount (c\text{-imp } c1 \text{ loc} + c\text{-work } c1 \text{ loc}) t'$ 
        =  $zcount (z\text{mset-frontier } (c\text{-pts } c1 \text{ loc})$ 
          +  $union\text{-frontiers } c1 \text{ loc}) t'$ 
        using iiws unchanged-imps by simp
    }
  }
moreover
  — If t disappeared from the frontier
  { assume t':  $t' \in_A \text{frontier } (c\text{-pts } c0 \text{ loc}) - \text{frontier } (c\text{-pts } c1 \text{ loc})$ 
    note t'["THEN mem-frontier-diff"]
    — then the worklist at t decreased by 1
    then have  $zcount (c\text{-work } c1 \text{ loc}) t' = zcount (c\text{-work } c0 \text{ loc}) t' - 1$ 
      using cm by (auto simp: ac-Diff-iff)
      — and the frontier at t decreased by 1
    moreover
    have  $zcount (z\text{mset-frontier } (c\text{-pts } c1 \text{ loc})) t'$ 
      =  $zcount (z\text{mset-frontier } (c\text{-pts } c0 \text{ loc})) t' - 1$ 
      using t'["THEN frontier-change-zmset-frontier"] by simp
      — and the sum didn't change
    moreover note unchanged-union
      — hence, the invariant is preserved.
    ultimately have
       $zcount (c\text{-imp } c1 \text{ loc} + c\text{-work } c1 \text{ loc}) t'$ 
      =  $zcount (z\text{mset-frontier } (c\text{-pts } c1 \text{ loc})$ 
        +  $union\text{-frontiers } c1 \text{ loc}) t'$ 
      using iiws unchanged-imps by simp
  }

```

```

}
moreover
  — If t's multiplicity in the frontier didn't change
  { assume a1:  $\neg t' \in_A \text{frontier } (c\text{-pts } c1 \text{ loc}) - \text{frontier } (c\text{-pts } c0 \text{ loc})$ 
    assume a2:  $\neg t' \in_A \text{frontier } (c\text{-pts } c0 \text{ loc}) - \text{frontier } (c\text{-pts } c1 \text{ loc})$ 
    from a1 a2 have zcount (frontier-changes (c-pts c1 loc) (c-pts c0 loc)) t' =
0
    by (intro not-mem-frontier-diff)
      — then the worklist at t didn't change
    with cm have zcount (c-work c1 loc) t' = zcount (c-work c0 loc) t'
    by (auto simp: ac-Diff-iff)
      — and the frontier at t didn't change
    moreover
    have zcount (zmset-frontier (c-pts c1 loc)) t'
      = zcount (zmset-frontier (c-pts c0 loc)) t'
    using a1 a2
    apply (clarsimp simp: member-antichain.rep-eq dest!: ac-notin-Diff)
    apply (metis ac-Diff-iff count-mset-set(1) count-mset-set(3) finite-set-antichain
member-antichain.rep-eq)
    done
      — and the sum didn't change
    moreover note unchanged-union
      — hence, the invariant is preserved.
    ultimately have
      zcount (c-imp c1 loc + c-work c1 loc) t'
      = zcount (zmset-frontier (c-pts c1 loc)
        + union-frontiers c1 loc) t'
    using iivs unchanged-imps by simp
  }
ultimately have
  zcount (c-imp c1 loc' + c-work c1 loc') t'
  = zcount (zmset-frontier (c-pts c1 loc')
    + union-frontiers c1 loc') t'
  using loc by auto
}
ultimately have
  zcount (c-imp c1 loc' + c-work c1 loc') t'
  = zcount (zmset-frontier (c-pts c1 loc')
    + union-frontiers c1 loc') t'
  by auto
}
with assms show ?thesis
by (auto simp: Let-def inv-imps-work-sum-zcount inv-imps-work-sum-zcount-def)
qed

```

— PR preserves *inv-imps-work-sum*

**lemma** *p-preserves-inv-imps-work-sum*:

**assumes** *next-propagate'* *c0 c1 loc t*

**and** *inv-imps-work-sum c0*

shows *inv-imps-work-sum c1*

**proof** –

– Given *next-propagate'* for *loc*, *t*, we show the result for *loc'*, *t'*.

```

{ fix loc t loc' t'
  assume p': next-propagate' c0 c1 loc t
  note p = this[unfolding next-propagate'-def]
  from p have unchanged-ps: unchanged c-pts c0 c1
    by simp
  assume inv-imps-work-sum c0
  then have iiws': inv-imps-work-sum-zcount c0
    by (simp add: inv-imps-work-sum-zcount)
  note iiws = iiws'[unfolding inv-imps-work-sum-zcount-def, THEN spec2]
  { assume loc: loc=loc'
    note iiws
    moreover note unchanged-ps
      – The t entries in loc's worklist are shifted to the implications.
    moreover from p have zcount (c-work c1 loc) t = 0
      by simp
    moreover from p have
      zcount (c-imp c1 loc) t
      = zcount (c-imp c0 loc) t + zcount (c-work c0 loc) t
      by simp
      – Since the implications of other locations don't change and loc can't
      have an edge to itself,  $\lambda c \text{ loc. } \sum \text{loc}' \in \text{UNIV. after-summary (zmset-of (mset-set (set-antichain (frontier (c-imp c \text{loc}'))))) (summary \text{loc}' \text{loc})}$  at loc doesn't change.
    moreover from p have union-frontiers c1 loc = union-frontiers c0 loc
      using summary-self by (auto intro!: sum.cong arg-cong[where f = Sum])
      – For all the other timestamps the worklist and implications don't change.
    moreover from p have
      tX ≠ t ⇒ zcount (c-work c1 loc) tX = zcount (c-work c0 loc) tX for tX
      by simp
    moreover from p have
      tX ≠ t ⇒ zcount (c-imp c1 loc) tX = zcount (c-imp c0 loc) tX for tX
      by simp
    ultimately have
      zcount (c-imp c1 loc' + c-work c1 loc') t'
      = zcount (zmset-frontier (c-pts c1 loc') + union-frontiers c1 loc') t'
      unfolding loc
      by (cases t=t') simp-all
  }
}

```

**moreover**

```

{ assume loc: loc≠loc'
  – The implications are unchanged at all locations other than loc.
  from p loc have unchanged-imps: c-imp c1 loc' = c-imp c0 loc'
    by simp
  { assume sum: summary loc loc' = {}A
    note iiws
    moreover note unchanged-ps
    moreover note unchanged-imps
  }
}

```

— The worklist only changes if  $\text{loc}$ ,  $\text{loc}'$  are connected.

**moreover from**  $p \text{ loc } \text{sum}$  **have**  $c\text{-work } c1 \text{ loc}' = c\text{-work } c0 \text{ loc}'$   
**by** *simp*

— Since the implications only change at  $\text{loc}$  and  $\text{loc}$  is not connected to  $\text{loc}'$ ,  
 $\lambda c \text{ loc}. \sum \text{loc}' \in \text{UNIV}. \text{after-summary } (\text{zmsset-of } (\text{mset-set } (\text{set-antichain } (\text{frontier } (c\text{-imp } c \text{ loc}'))))) (\text{summary } \text{loc}' \text{ loc})$  doesn't change.

**moreover from**  $p \text{ loc } \text{sum}$  **have**  $\text{union-frontiers } c1 \text{ loc}' = \text{union-frontiers } c0 \text{ loc}'$   
**by** (*auto intro!*: *sum.cong arg-cong*[**where**  $f = \text{Sum}$ ])

**ultimately have**  
 $\text{zcount } (c\text{-imp } c1 \text{ loc}' + c\text{-work } c1 \text{ loc}') t'$   
 $= \text{zcount } (\text{zmsset-frontier } (c\text{-pts } c1 \text{ loc}') + \text{union-frontiers } c1 \text{ loc}') t'$   
**by** *simp*

}

**moreover**  
{ **assume**  $\text{sum}: \text{summary } \text{loc } \text{loc}' \neq \{\}_A$   
—  $\lambda c \text{ loc}. \sum \text{loc}' \in \text{UNIV}. \text{after-summary } (\text{zmsset-of } (\text{mset-set } (\text{set-antichain } (\text{frontier } (c\text{-imp } c \text{ loc}'))))) (\text{summary } \text{loc}' \text{ loc})$  at  $\text{loc}'$  changed by whatever amount the frontier changed.  
**note** *iiws*  
*unchanged-imps*  
*unchanged-ps*

**moreover from**  $p' \text{ sum}$  **have**  
 $\text{union-frontiers } c1 \text{ loc}' =$   
 $\text{union-frontiers } c0 \text{ loc}'$   
 $+ \text{after-summary}$   
 $(\text{zmsset-frontier } (c\text{-imp } c1 \text{ loc}) - \text{zmsset-frontier } (c\text{-imp } c0 \text{ loc}))$   
 $(\text{summary } \text{loc } \text{loc}')$   
**by** (*auto intro!*: *next-p-union-frontier-change*)  
— The worklist at  $\text{loc}'$  changed by the same amount

**moreover from**  $p' \text{ sum}$  **have**  
 $c\text{-work } c1 \text{ loc}' =$   
 $c\text{-work } c0 \text{ loc}'$   
 $+ \text{after-summary}$   
 $(\text{zmsset-frontier } (c\text{-imp } c1 \text{ loc}) - \text{zmsset-frontier } (c\text{-imp } c0 \text{ loc}))$   
 $(\text{summary } \text{loc } \text{loc}')$   
**by** (*auto intro!*: *next-p-frontier-change*)  
— The two changes cancel out.

**ultimately have**  
 $\text{zcount } (c\text{-imp } c1 \text{ loc}' + c\text{-work } c1 \text{ loc}') t'$   
 $= \text{zcount } (\text{zmsset-frontier } (c\text{-pts } c1 \text{ loc}') + \text{union-frontiers } c1 \text{ loc}') t'$   
**by** *simp*

}

**ultimately have**  
 $\text{zcount } (c\text{-imp } c1 \text{ loc}' + c\text{-work } c1 \text{ loc}') t'$   
 $= \text{zcount } (\text{zmsset-frontier } (c\text{-pts } c1 \text{ loc}') + \text{union-frontiers } c1 \text{ loc}') t'$   
**by** *auto*

}

**ultimately have**

```

      zcount (c-imp c1 loc' + c-work c1 loc') t'
      = zcount (zmsset-frontier (c-pts c1 loc') + union-frontiers c1 loc') t'
    by (cases loc=loc') auto
  }
  with assms show ?thesis
  by (auto simp: next-propagate'-def Let-def inv-imps-work-sum-zcount inv-imps-work-sum-zcount-def)
qed

```

**lemma** *next-preserves-inv-imps-work-sum*:

```

  assumes next s
  and holds inv-imps-work-sum s
  shows next (holds inv-imps-work-sum) s
  using assms
  cm-preserves-inv-imps-work-sum
  p-preserves-inv-imps-work-sum
  by (simp, cases rule: next'-inv)

```

**lemma** *spec-imp-iiws*:  $\text{spec } s \implies \text{alw } (\text{holds } \text{inv-imps-work-sum}) s$   
 using *init-imp-inv-imps-work-sum next-preserves-inv-imps-work-sum*  
 by (*auto intro: alw-invar simp: alw-mono spec-def*)

### 7.3.2 Invariant: *inv-imp-plus-work-nonneg*

There is never an update in the worklist that could cause implications to become negative.

**definition** *inv-imp-plus-work-nonneg* **where**

$\text{inv-imp-plus-work-nonneg } c \equiv \forall \text{loc } t. 0 \leq \text{zcount } (c\text{-imp } c \text{ loc}) t + \text{zcount } (c\text{-work } c \text{ loc}) t$

**lemma** *iiws-imp-iiwn*:

```

  assumes inv-imps-work-sum c
  shows inv-imp-plus-work-nonneg c

```

**proof** –

```

  {
    fix loc
    fix t
    assume inv-imps-work-sum c
    then have iiws': inv-imps-work-sum-zcount c
      by (simp add: inv-imps-work-sum-zcount)
    note iiws = iiws'[unfolded inv-imps-work-sum-zcount-def, THEN spec2]
    have  $0 \leq \text{zcount } (\text{union-frontiers } c \text{ loc}) t$ 
      by (simp add: union-frontiers-nonneg)
    with iiws have  $0 \leq \text{zcount } (c\text{-imp } c \text{ loc} + c\text{-work } c \text{ loc}) t$ 
      by simp
  }
  with assms show ?thesis
  by (simp add: inv-imp-plus-work-nonneg-def)
qed

```

**lemma** *spec-imp-iipwn*:  $spec\ s \implies alw\ (holds\ inv-imp-plus-work-nonneg)\ s$   
**using** *spec-imp-iivs iiws-imp-iipwn*  
*alw-mono holds-mono*  
**by** *blast*

### 7.3.3 Invariant: *inv-implications-nonneg*

**lemma** *init-imp-inv-implications-nonneg*:  
**assumes** *init-config c*  
**shows** *inv-implications-nonneg c*  
**using** *assms* **by** (*simp add: init-config-def inv-implications-nonneg-def*)

**lemma** *cm-preserves-inv-implications-nonneg*:  
**assumes** *next-change-multiplicity' c0 c1 loc t n*  
**and** *inv-implications-nonneg c0*  
**shows** *inv-implications-nonneg c1*  
**using** *assms* **by** (*simp add: next-change-multiplicity'-def inv-implications-nonneg-def*)

**lemma** *p-preserves-inv-implications-nonneg*:  
**assumes** *next-propagate' c0 c1 loc t*  
**and** *inv-implications-nonneg c0*  
**and** *inv-imp-plus-work-nonneg c0*  
**shows** *inv-implications-nonneg c1*  
**using** *assms*  
**by** (*auto simp: next-propagate'-def Let-def inv-implications-nonneg-def inv-imp-plus-work-nonneg-def*)

**lemma** *next-preserves-inv-implications-nonneg*:  
**assumes** *next s*  
**and** *holds inv-implications-nonneg s*  
**and** *holds inv-imp-plus-work-nonneg s*  
**shows** *next (holds inv-implications-nonneg) s*  
**using** *assms*  
*cm-preserves-inv-implications-nonneg*  
*p-preserves-inv-implications-nonneg*  
**by** (*simp, cases rule: next'-inv*)

**lemma** *alw-inv-implications-nonneg*:  $spec\ s \implies alw\ (holds\ inv-implications-nonneg)\ s$   
**apply** (*frule spec-imp-iipwn*)  
**unfolding** *spec-def*  
**apply** (*rule alw-invar*)  
**using** *init-imp-inv-implications-nonneg* **apply** *auto* []  
**using** *next-preserves-inv-implications-nonneg*  
**apply** (*metis (no-types, lifting) alw-iff-sdrop*)  
**done**

**lemma** *after-summary-Diff*:  $after\ summary\ (M - N)\ S = after\ summary\ M\ S - after\ summary\ N\ S$   
**by** (*simp add: sum-subtractf after-summary-def*)

**lemma** *mem-zmset-frontier*:  $x \in \#_z \text{zmset-frontier } M \longleftrightarrow x \in_A \text{frontier } M$   
**by** *transfer simp*

**lemma** *obtain-frontier-elim*:  
**assumes**  $0 < \text{zcount } M \ t$   
**obtains**  $u$  **where**  $u \in_A \text{frontier } M \ u \leq t$   
**using** *assms* **by** (*atomize-elim, transfer*)  
*(auto simp: minimal-antichain-def dest: order-zmset-exists-foundation)*

**lemma** *frontier-unionD*:  $t \in_A \text{frontier } (M+N) \implies 0 < \text{zcount } M \ t \vee 0 < \text{zcount } N \ t$   
**by** *transfer'* (*auto simp: minimal-antichain-def*)

**lemma** *ps-frontier-in-imps-wl*:  
**assumes** *inv-imps-work-sum c*  
**and**  $0 < \text{zcount } (\text{zmset-frontier } (c\text{-pts } c \ \text{loc})) \ t$   
**shows**  $0 < \text{zcount } (c\text{-imp } c \ \text{loc} + c\text{-work } c \ \text{loc}) \ t$   
**proof** –  
**have**  $0 \leq \text{zcount } (\text{union-frontiers } c \ \text{loc}) \ t$   
**by** (*rule union-frontiers-nonneg*)  
**with** *assms(2)* **show** *?thesis*  
**using** *assms(1)[unfolded inv-imps-work-sum-def, THEN spec, of loc]*  
**by** *fastforce*  
**qed**

**lemma** *obtain-elim-frontier*:  
**assumes**  $0 < \text{zcount } M \ t$   
**obtains**  $s$  **where**  $s \leq t \wedge s \in_A \text{frontier } M$   
**by** (*rule order-finite-set-obtain-foundation[of {s. zcount M s > 0} t thesis]*)  
*(auto simp: assms antichain-inverse frontier-def member-antichain.rep-eq in-minimal-antichain)*

**lemma** *obtain-elim-zmset-frontier*:  
**assumes**  $0 < \text{zcount } M \ t$   
**obtains**  $s$  **where**  $s \leq t \wedge 0 < \text{zcount } (\text{zmset-frontier } M) \ s$   
**using** *assms* **by** (*auto simp: member-antichain.rep-eq intro: obtain-elim-frontier*)

**lemma** *ps-in-imps-wl*:  
**assumes** *inv-imps-work-sum c*  
**and**  $0 < \text{zcount } (c\text{-pts } c \ \text{loc}) \ t$   
**obtains**  $s$  **where**  $s \leq t \wedge 0 < \text{zcount } (c\text{-imp } c \ \text{loc} + c\text{-work } c \ \text{loc}) \ s$   
**proof** *atomize-elim*  
**note** *iiws = assms(1)[unfolded inv-imps-work-sum-def, THEN spec, of loc]*  
**obtain**  $u$  **where**  $u: u \leq t \wedge u \in_A \text{frontier } (c\text{-pts } c \ \text{loc})$   
**using** *assms(2)* *obtain-elim-frontier* **by** *blast*  
**with** *assms(1)* **have**  $0 < \text{zcount } (c\text{-imp } c \ \text{loc} + c\text{-work } c \ \text{loc}) \ u$   
**apply** (*intro ps-frontier-in-imps-wl[of c loc u]*)  
**apply** (*auto intro: iffD1[OF member-antichain.rep-eq]*)

**done**  
**with**  $u$  **show**  $\exists s \leq t. 0 < \text{zcount } (c\text{-imp } c \text{ loc} + c\text{-work } c \text{ loc}) s$   
**by** (*auto intro: exI[of - u]*)  
**qed**

**lemma** *zero-le-after-summary-single[simp]*:  $0 \leq \text{zcount } (\text{after-summary } \{\#t\# \}_z S) x$   
**by** (*auto intro: zero-le-sum-single simp: after-summary-def*)

**lemma** *one-le-zcount-after-summary*:  $s \in_A S \implies 1 \leq \text{zcount } (\text{after-summary } \{\#t\# \}_z S) (\text{results-in } t s)$   
**unfolding** *image-zmset-single after-summary-def*  
**apply** (*subst zcount-sum*)  
**apply** (*subst forw-subst[of 1 zcount {\#results-in t s\#}\_z (results-in t s)]*)  
**apply** *simp*  
**apply** (*rule sum-nonneg-leq-bound[of set-antichain S \lambda u. zcount {\#results-in t u\#}\_z (results-in t s) - s]*)  
**apply** (*auto simp: member-antichain.rep-eq*)  
**done**

**lemma** *zero-lt-zcount-after-summary*:  $s \in_A S \implies 0 < \text{zcount } (\text{after-summary } \{\#t\# \}_z S) (\text{results-in } t s)$   
**apply** (*subst int-one-le-iff-zero-less[symmetric]*)  
**apply** (*intro one-le-zcount-after-summary*)  
**apply** *simp*  
**done**

**lemma** *pos-zcount-after-summary*:  
 $(\bigwedge t. 0 \leq \text{zcount } M t) \implies 0 < \text{zcount } M t \implies s \in_A S \implies 0 < \text{zcount } (\text{after-summary } M S) (\text{results-in } t s)$   
**by** (*auto intro!: sum-pos2 pos-zcount-image-zmset simp: member-antichain.rep-eq zcount-sum after-summary-def*)

**lemma** *after-summary-nonneg*:  $(\bigwedge t. 0 \leq \text{zcount } M t) \implies 0 \leq \text{zcount } (\text{after-summary } M S) t$   
**by** (*auto simp: zcount-sum after-summary-def intro: sum-nonneg*)

**lemma** *after-summary-zmset-of-nonneg[simp, intro]*:  $0 \leq \text{zcount } (\text{after-summary } (\text{zmset-of } M) S) t$   
**by** (*simp add: mset-neg-image-zmset mset-neg-sum-set mset-neg-zcount-nonneg after-summary-def*)

**lemma** *pos-zcount-union-frontiers*:  
 $\text{zcount } (\text{after-summary } (\text{zmset-frontier } (c\text{-imp } c \text{ l1})) (\text{summary } l1 \text{ l2})) (\text{results-in } t s)$   
 $\leq \text{zcount } (\text{union-frontiers } c \text{ l2}) (\text{results-in } t s)$   
**apply** (*subst zcount-sum*)  
**apply** (*rule member-le-sum*)  
**apply** (*auto intro!: pos-zcount-image-zmset*)

done

**lemma** *after-summary-Sum-fun: finite MM  $\implies$  after-summary  $(\sum M \in MM. f M)$*   
 $A = (\sum M \in MM. \text{after-summary } (f M) A)$   
**by** (*induct MM rule: finite-induct*) (*auto simp: after-summary-union*)

**lemma** *after-summary-obtain-pre:*  
**assumes**  $\bigwedge t. 0 \leq \text{zcount } M t$   
**and**  $0 < \text{zcount } (\text{after-summary } M S) t$   
**obtains**  $t' s$  **where**  $0 < \text{zcount } M t'$  *results-in*  $t' s = t s \in_A S$   
**using** *assms unfolding after-summary-def*  
**apply** *atomize-elim*  
**apply** (*subst (asm) zcount-sum*)  
**apply** (*drule sum-pos-ex-elem-pos*)  
**apply** *clarify*  
**subgoal for**  $s$   
**apply** (*subst ex-comm*)  
**apply** (*rule exI[of - s]*)  
**apply** (*drule pos-image-zmset-obtain-pre[rotated]*)  
**apply** *simp*  
**apply** (*simp add: member-antichain.rep-eq*)  
**done**  
**done**

**lemma** *empty-antichain[dest]:  $x \in_A \text{antichain } \{\} \implies \text{False}$*   
**by** (*metis empty-antichain.abs-eq mem-antichain-nonempty*)

**definition** *impWitnessPath where*  
 $\text{impWitnessPath } c \text{ loc1 } \text{ loc2 } xs \ t =$   
 $\text{path } \text{loc1 } \text{loc2 } xs \wedge$   
 $\text{distinct } xs \wedge$   
 $(\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc1}) \wedge t = \text{results-in } t' (\text{sum-path-weights } xs) \wedge$   
 $(\forall k < \text{length } xs. (\exists t. t \in_A \text{frontier } (c\text{-imp } c (TO (xs ! k))) \wedge t = \text{results-in } t'$   
 $(\text{sum-path-weights } (\text{take } (k+1) xs))))))$

**lemma** *impWitnessPathEx:*  
**assumes**  $t \in_A \text{frontier } (c\text{-imp } c \text{ loc2})$   
**shows**  $(\exists \text{loc1 } xs. \text{impWitnessPath } c \text{ loc1 } \text{loc2 } xs \ t)$   
**proof** –  
**have**  $1: \text{path } \text{loc2 } \text{loc2 } []$  **by** (*simp add: path0*)  
**have**  $2: \text{distinct } []$  **by** *auto*  
**have**  $0 = \text{sum-path-weights } []$  **using** *foldr-Nil list.map(1)* **by** *auto*  
**then have**  $3: t = \text{results-in } t (\text{sum-path-weights } [])$  **by** (*simp add: results-in-zero*)  
**with**  $1 \ 2$  **assms** **show** *?thesis*  
**unfolding** *impWitnessPath-def*  
**by** (*force simp: results-in-zero*)  
**qed**

**definition** *longestImpWitnessPath where*

$longestImpWitnessPath\ c\ loc1\ loc2\ xs\ t = ($   
 $impWitnessPath\ c\ loc1\ loc2\ xs\ t \wedge$   
 $(\forall\ loc'\ xs'.\ impWitnessPath\ c\ loc'\ loc2\ xs'\ t \longrightarrow length\ (xs') \leq length\ (xs)))$

**lemma** *finite-edges*:  $finite\ \{(loc1,s,loc2).\ s \in_A\ summary\ loc1\ loc2\}$

**proof** –

**let**  $?sums = (\bigcup\ ((\lambda(loc1,loc2).\ set-antichain\ (summary\ loc1\ loc2))\ 'UNIV))$   
**have**  $finite\ ?sums$   
**by** *auto*  
**then have**  $finite\ ((UNIV::'loc\ set) \times ?sums \times (UNIV::'loc\ set))$   
**by** *auto*  
**moreover have**  $\{(loc1,s,loc2).\ s \in_A\ summary\ loc1\ loc2\} \subseteq ((UNIV::'loc\ set) \times$   
 $?sums \times (UNIV::'loc\ set))$   
**by** (*force simp: member-antichain.rep-eq*)  
**ultimately show**  $?thesis$   
**by** (*rule rev-finite-subset*)

**qed**

**lemma** *longestImpWitnessPathEx*:

**assumes**  $t \in_A\ frontier\ (c-imp\ c\ loc2)$

**shows**  $(\exists\ loc1\ xs.\ longestImpWitnessPath\ c\ loc1\ loc2\ xs\ t)$

**proof** –

**define**  $paths$  **where**  $paths = \{(loc1, xs).\ impWitnessPath\ c\ loc1\ loc2\ xs\ t\}$   
**with**  $impWitnessPathEx[OF\ assms]$  **obtain**  $p$  **where**  $p \in paths$  **by** *auto*  
**have**  $\forall p.\ p \in paths \longrightarrow (length \circ snd)\ p < card\ \{(loc1,s,loc2).\ s \in_A\ summary$   
 $loc1\ loc2\} + 1$

**proof** (*intro allI impI*)

**fix**  $p$

**assume**  $p: p \in paths$

**then show**  $(length \circ snd)\ p < card\ \{(loc1,s,loc2).\ s \in_A\ summary\ loc1\ loc2\}$   
 $+ 1$

**by** (*auto simp: paths-def impWitnessPath-def less-Suc-eq-le finite-edges path-edge*  
*dest!: distinct-card[symmetric] intro!: card-mono*)

**qed**

**from**  $ex-has-greatest-nat[OF\ \langle p \in paths \rangle\ this]$  **show**  $?thesis$

**by** (*auto simp: paths-def longestImpWitnessPath-def*)

**qed**

**lemma** *path-first-loc*:  $path\ l1\ l2\ xs \implies xs \neq [] \implies xs\ !\ 0 = (l1',s,l2') \implies l1 =$   
 $l1'$

**proof** (*induct arbitrary: l1' s l2 rule: path.induct*)

**case** (*path0 l1 l2*)

**then show**  $?case$  **by** *auto*

**next**

**case** (*path l1 l2 xs lbl l3*)

**then show**  $?case$

**apply** (*cases xs=[]*)

**apply** (*auto elim: path0E*)  $[]$

**apply** (*rule path(2)[of l1' s]*)

```

    by (auto simp: nth-append)
qed

lemma find-witness-from-frontier:
  assumes  $t \in_A \text{frontier } (c\text{-imp } c \text{ loc2})$ 
  and  $\text{inv-imps-work-sum } c$ 
  shows  $\exists t' \text{ loc1 } xs. (\text{path } \text{loc1 } \text{loc2 } xs \wedge t = \text{results-in } t' (\text{sum-path-weights } xs) \wedge$ 
     $(t' \in_A \text{frontier } (c\text{-pts } c \text{ loc1}) \vee 0 > \text{zcount } (c\text{-work } c \text{ loc1}) t')$ )
  proof -
    obtain  $\text{loc1 } xs$  where  $\text{longestP}: \text{longestImpWitnessPath } c \text{ loc1 } \text{loc2 } xs \ t$ 
    using  $\text{assms}(1) \text{longestImpWitnessPathEx}$  by blast
    then obtain  $t'$  where  $t': t' \in_A \text{frontier } (c\text{-imp } c \text{ loc1}) \ t = \text{results-in } t' (\text{sum-path-weights } xs)$ 
    ( $\forall k < \text{length } xs. (\exists t. t \in_A \text{frontier } (c\text{-imp } c \text{ (TO } (xs \ ! \ k))) \wedge t = \text{results-in } t' (\text{sum-path-weights } (\text{take } (k+1) \ xs))))$ )
    by (auto simp add:  $\text{longestImpWitnessPath-def } \text{impWitnessPath-def}$ )
    from  $t'(1)$  have  $\text{cases}: 0 > \text{zcount } (c\text{-work } c \text{ loc1}) \ t' \vee$ 
       $(t' \in_{\#_z} (\text{zmsset-frontier } (c\text{-pts } c \text{ loc1}) + \text{union-frontiers } c \text{ loc1}))$ 
    using  $\text{assms}(2)$ 
    apply ( $\text{clarsimp intro!: } \text{verit-forall-inst}(6) \text{simp: } \text{inv-imps-work-sum-def } \text{not-less}$ )
    apply ( $\text{metis } \text{add-pos-nonneg } \text{mem-zmsset-frontier } \text{member-frontier-pos-zmsset-obtain-frontier-elem } \text{zcount-empty } \text{zcount-ne-zero-iff } \text{zcount-union } \text{zmsset-frontier-empty}$ )
    done
    then show ?thesis
  proof cases
    assume  $\text{case1}: 0 > \text{zcount } (c\text{-work } c \text{ loc1}) \ t'$ 
    then show ?thesis using  $t'$   $\text{longestP}$ 
    using  $\text{impWitnessPath-def } \text{longestImpWitnessPath-def } \text{dataflow-topology-axioms}$ 
  by blast
  next
    assume  $\text{case2}: \neg 0 > \text{zcount } (c\text{-work } c \text{ loc1}) \ t'$ 
    have  $(t' \in_{\#_z} (\text{zmsset-frontier } (c\text{-pts } c \text{ loc1}) + \text{union-frontiers } c \text{ loc1}))$ 
    using  $\text{case2}$  cases by auto
    then have  $\text{case-split2}: (t' \in_{\#_z} \text{zmsset-frontier } (c\text{-pts } c \text{ loc1})) \vee (t' \in_{\#_z} \text{union-frontiers } c \text{ loc1})$ 
    by ( $\text{metis } (\text{no-types, lifting}) \text{add-diff-cancel-left' } \text{in-diff-zcount } \text{zcount-ne-zero-iff}$ )
    then show ?thesis
  proof cases
    assume  $\text{case2-1}: t' \in_{\#_z} \text{zmsset-frontier } (c\text{-pts } c \text{ loc1})$ 
    have  $t' \in_A \text{frontier } (c\text{-pts } c \text{ loc1})$ 
    using  $\text{case2-1 } \text{mem-zmsset-frontier}$  by blast
    then show ?thesis
    using  $t' \text{impWitnessPath-def } \text{longestImpWitnessPath-def } \text{dataflow-topology-axioms}$ 
  longestP by blast
  next
    assume  $\neg t' \in_{\#_z} \text{zmsset-frontier } (c\text{-pts } c \text{ loc1})$ 
    then have  $\text{case2-2}: t' \in_{\#_z} \text{union-frontiers } c \text{ loc1}$  using  $\text{case-split2}$  by blast
    then obtain  $\text{loc0 } t0 \ s0$  where  $\text{loc0} : t0 \in_A \text{frontier } (c\text{-imp } c \ \text{loc0})$ 
       $s0 \in_A (\text{summary } \text{loc0 } \text{loc1})$ 

```

```

    t' = results-in t0 s0
  by (fastforce simp: after-summary-def set-zmset-def zcount-sum
      member-antichain.rep-eq[symmetric] zcount-image-zmset card-gt-0-iff
      simp del: zcount-ne-zero-iff
      elim!: sum.not-neutral-contains-not-neutral)
let ?xs' = (loc0, s0, loc1) # xs
have path-xs: path loc1 loc2 xs
  using impWitnessPath-def longestImpWitnessPath-def longestP by blast
have is-path-xs': path loc0 loc2 ?xs' using longestP
  apply (simp add: longestImpWitnessPath-def impWitnessPath-def)
  by (metis append-Cons append-Nil path-singleton path-trans loc0(2))
have  $\forall k < \text{length } ?xs'$ .
  results-in t0 (sum-path-weights (take (k+1) ?xs'))
   $\in_A \text{frontier } (c\text{-imp } c \text{ (TO } (?xs' ! k)))$ 
  apply clarify
subgoal for k
  apply (cases k=0)
  subgoal
    using loc0(3) t'(1) by (auto simp: results-in-zero)
  subgoal
    using t'(3)[rule-format, unfolded loc0(3) followed-by-summary, of k-1,
simplified]
    by auto
  done
done
note r = this[rule-format]
have distinctxs: distinct xs
  using longestP
  by (simp add: longestImpWitnessPath-def impWitnessPath-def)
then show ?thesis
proof cases
  assume case-distinct: distinct ?xs'

  have t = results-in t0 (sum-path-weights ?xs') using longestP loc0(3)
    apply (simp add: longestImpWitnessPath-def impWitnessPath-def)
    by (simp add: followed-by-summary t'(2))
  then have impPath: impWitnessPath c loc0 loc2 ?xs' t
    using is-path-xs' case-distinct loc0(1)
    apply (simp add: impWitnessPath-def)
    using r by auto
  have length ?xs' > length xs by auto
  then have  $\neg \text{longestImpWitnessPath } c \text{ loc1 loc2 } xs \ t$ 
    using impPath leD unfolding longestImpWitnessPath-def by blast
  then show ?thesis using longestP by blast
next
  assume  $\neg \text{distinct } ?xs'$ 

with distinctxs obtain k where k: TO (?xs' ! k) = loc0 k < length ?xs'
  apply atomize-elim

```

```

apply clarsimp
apply (subst (asm) in-set-conv-nth)
apply clarify
subgoal for i
  apply (cases i=0)
  subgoal
    using path-first-loc[OF path-xs]
    by force
  subgoal
    apply (rule exI[of - i])
    using path-xs
    apply (auto dest: path-to-eq-from[of - - xs i-1])
    done
  done
done
have results-in t0 (sum-path-weights (take (k+1) ?xs')) ∈A frontier (c-imp
c loc0)
  by (rule r[OF k(2), unfolded k(1)])
moreover have t0 < results-in t0 (sum-path-weights (take (k+1) ?xs'))
  apply simp
  apply (rule no-zero-cycle[of loc0 take (k+1) ?xs' sum-path-weights (take
(k+1) ?xs') t0, simplified])
  using is-path-xs' k path-take-to by fastforce
  ultimately show ?thesis
  using loc0(1) frontier-comparable-False by blast
qed
qed
qed
qed

```

**lemma** *implication-implies-pointstamp:*

**assumes**  $t \in_A \text{frontier } (c\text{-imp } c \text{ loc})$

**and** *inv-imps-work-sum c*

**shows**  $\exists t' \text{ loc}' s. s \in_A \text{path-summary loc}' \text{ loc} \wedge t \geq \text{results-in } t' s \wedge$   
 $(t' \in_A \text{frontier } (c\text{-pts } c \text{ loc}') \vee 0 > \text{zcount } (c\text{-work } c \text{ loc}') t')$

**proof** –

**obtain** *loc' t' xs* **where** *witness:*

*path loc' loc xs*

$t = \text{results-in } t' (\text{sum-path-weights } xs)$

$t' \in_A \text{frontier } (c\text{-pts } c \text{ loc}') \vee 0 > \text{zcount } (c\text{-work } c \text{ loc}') t'$

**using** *assms find-witness-from-frontier* **by** *blast*

**obtain** *s* **where**  $s \in_A \text{path-summary loc}' \text{ loc} \quad s \leq (\text{sum-path-weights } xs)$

**using** *witness(1) path-path-weight* **by** *blast*

**then have**  $t \geq \text{results-in } t' s$

**using** *witness(2) results-in-mono(2)* **by** *blast*

**then show** *?thesis*

**using** *witness(3) s* **by** *auto*

**qed**

## 7.4 Proof of Safety

**lemma** *results-in-sum-path-weights-append:*

$results\text{-}in\ t\ (sum\text{-}path\text{-}weights\ (xs\ @\ [(loc2,\ s,\ loc3)])) = results\text{-}in\ (results\text{-}in\ t\ (sum\text{-}path\text{-}weights\ xs))\ s$   
**by** (*metis followed-by-summary sum-path-weights-append-singleton*)

**context**

**fixes**  $c :: ('loc,\ 't)\ configuration$

**begin**

**inductive** *loc-imps-fw* **where**

$loc\text{-}imps\text{-}fw\ loc\ loc\ (c\text{-}imp\ c\ loc)\ []\ |$   
 $loc\text{-}imps\text{-}fw\ loc1\ loc2\ M\ xs \implies s \in_A\ summary\ loc2\ loc3 \implies distinct\ (xs\ @\ [(loc2,\ s,\ loc3)]) \implies$   
 $loc\text{-}imps\text{-}fw\ loc1\ loc3\ (\{\#\ results\text{-}in\ t\ s.\ t \in_{\#z}\ M\ \#\} + c\text{-}imp\ c\ loc3)\ (xs\ @\ [(loc2,\ s,\ loc3)])$

**end**

**lemma** *loc-imps-fw-conv-path:*  $loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs \implies path\ loc1\ loc2\ xs$

**by** (*induct rule: loc-imps-fw.induct*) (*auto intro: path.intros*)

**lemma** *path-conv-loc-imps-fw:*  $path\ loc1\ loc2\ xs \implies distinct\ xs \implies \exists M.\ loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs$

**proof** (*induct rule: path.induct*)

**case** (*path0 l1 l2*)

**then show** *?case* **by** (*auto intro: loc-imps-fw.intros*)

**next**

**case** (*path l1 l2 xs lbl l3*)

**then obtain**  $M$  **where**  $loc\text{-}imps\text{-}fw\ c\ l1\ l2\ M\ xs$

**by** *auto*

**with** *path* **show** *?case*

**by** (*force intro: loc-imps-fw.intros(2)*)

**qed**

**lemma** *path-summary-conv-loc-imps-fw:*

$s \in_A\ path\text{-}summary\ loc1\ loc2 \implies \exists M\ xs.\ loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs \wedge sum\text{-}path\text{-}weights\ xs = s$

**proof** –

**assume** *path-sum:*  $s \in_A\ path\text{-}summary\ loc1\ loc2$

**then obtain**  $M\ xs$  **where**  $le:\ path\ loc1\ loc2\ xs\ loc\text{-}imps\text{-}fw\ c\ loc1\ loc2\ M\ xs$   
 $sum\text{-}path\text{-}weights\ xs \leq s\ distinct\ xs$

**apply** *atomize-elim*

**apply** (*drule path-weight-conv-path*)

**apply** *clarsimp*

**apply** (*drule path-distinct*)

**apply** *clarsimp*

**subgoal for**  $ys\ xs$

**apply** (*rule exI[of - xs]*)

```

    apply (rule conjI, assumption)
    apply (drule path-conv-loc-imps-fw[of loc1 loc2 xs c])
    using subseq-sum-weights-le apply auto
  done
done
show ?thesis
proof (cases sum-path-weights xs = s)
  case True
    with le show ?thesis by auto
  next
  case False
    with le have sum-path-weights xs < s
      by auto
    with le(1) path-sum have False
      by (auto dest: path-weight-conv-path)
    then show ?thesis
      by blast
qed
qed

lemma image-zmset-id[simp]: {#x. x ∈ #z M#} = M
  by transfer (auto simp: equiv-zmset-def)

lemma sum-pos: finite M ⇒ ∀ x ∈ M. 0 ≤ f x ⇒ y ∈ M ⇒ 0 < (f y :: ordered-comm-monoid-add)
  ⇒ 0 < (∑ x ∈ M. f x)
proof (induct M rule: finite-induct)
  case empty
    then show ?case by simp
  next
  case (insert x F)
    then show ?case
      by (cases 0 < f x) (auto intro: sum-nonneg add-pos-nonneg add-nonneg-pos)
qed

lemma loc-imps-fw-M-in-implications:
  assumes loc-imps-fw c loc1 loc2 M xs
    and inv-imps-work-sum c
    and inv-implications-nonneg c
    and ∧ loc. c-work c loc = {#}#z
    and 0 < zcount M t
  shows ∃ s. s ≤ t ∧ s ∈A frontier (c-imp c loc2)
  using assms(1,5)
proof (induct arbitrary: t rule: loc-imps-fw.induct)
  note iiws = assms(2)[unfolded inv-imps-work-sum-def assms(4), simplified, rule-format]
  case (1 loc t)
    then show ?case
      by (auto elim: obtain-elem-frontier)
  next
  note iiws = assms(2)[unfolded inv-imps-work-sum-def assms(4), simplified, rule-format]

```

```

case (2 loc1 loc2 M xs s loc3 t)
from 2(5) have disj: 0 < zcount {# results-in t s. t ∈#z M #} t ∨ 0 < zcount
(c-imp c loc3) t
  by auto
show ?case
proof –
  { assume 0 < zcount {# results-in t s. t ∈#z M #} t
    then obtain t' where t': results-in t' s = t 0 < zcount M t'
      apply atomize-elim
      apply (rule ccontr)
      apply (subst (asm) zcount-image-zmset)
      apply (clarsimp simp: vimage-def)
      apply (metis (mono-tags, lifting) Int-iff mem-Collect-eq sum-pos-ex-elem-pos)
      done
    obtain u where u: u ≤ t' u ∈A frontier (c-imp c loc2)
      by atomize-elim (rule 2(2)[OF t'(2)])
    then have riu-le-rit': results-in u s ≤ t
      by (simp add: t'(1)[symmetric] results-in-mono)
    from u have 0 < zcount (union-frontiers c loc3) (results-in u s)
      apply (subst zcount-sum)
      apply (rule sum-pos[where y=loc2])
      apply simp-all [3]
      apply (clarsimp simp: after-summary-def)
      apply (subst zcount-sum)
      apply (rule sum-pos[where y=s])
      using 2(3) apply simp-all [3]
      apply (subst zcount-image-zmset)
      apply simp
      apply (subst card-eq-sum)
      apply (rule sum-pos[where y=u])
      apply simp-all
      done
    then have 0 < zcount (zmset-frontier (c-pts c loc3)) (results-in u s) + zcount
(union-frontiers c loc3) (results-in u s)
      by (auto intro: add-nonneg-pos)
    with riu-le-rit' have ?thesis
      apply (subst (asm) zcount-union[symmetric])
      apply (subst iiws)
      apply (erule obtain-elem-frontier)
      subgoal for u'
        by (auto intro!: exI[of - u'])
      done
  }
moreover
  { — Same as induction base case
    assume 0 < zcount (c-imp c loc3) t
    then have ?thesis
      by (auto elim: obtain-elem-frontier)
  }
}

```

**moreover note** *disj*  
**ultimately show** *?thesis*  
 by *blast*  
**qed**  
**qed**

**lemma** *loc-imps-fw-M-nonneg[simp]*:  
**assumes** *loc-imps-fw c loc1 loc2 M xs*  
**and** *inv-implications-nonneg c*  
**shows**  $0 \leq \text{zcount } M \ t$   
**using** *assms*  
**by** (*induct arbitrary: t rule: loc-imps-fw.induct*)  
 (*auto intro!: add-nonneg-nonneg sum-nonneg simp: zcount-image-zmset assms(2)[unfolded inv-implications-nonneg-def]*)

**lemma** *loc-imps-fw-implication-in-M*:  
**assumes** *inv-imps-work-sum c*  
**and** *inv-implications-nonneg c*  
**and** *loc-imps-fw c loc1 loc2 M xs*  
**and**  $0 < \text{zcount } (c\text{-imp } c \ \text{loc1}) \ t$   
**shows**  $0 < \text{zcount } M \ (\text{results-in } t \ (\text{sum-path-weights } xs))$   
**using** *assms(3,4)*  
**proof** (*induct rule: loc-imps-fw.induct*)  
**note** *iiws = assms(1)[unfolded inv-imps-work-sum-def assms(3), simplified, rule-format]*  
**case** (*1 loc*)  
**then show** *?case*  
 by (*simp add: results-in-zero*)  
**next**  
**case** (*2 loc1 loc2 M xs s loc3*)  
**have**  $0 < \text{zcount } M \ (\text{results-in } t \ (\text{sum-path-weights } xs))$   
 by (*rule 2(2)[OF 2(5)]*)  
**then show** *?case*  
**apply** (*subst results-in-sum-path-weights-append*)  
**apply** (*subst zcount-union*)  
**apply** (*rule add-pos-nonneg*)  
**apply** (*subst zcount-image-zmset*)  
**apply** (*rule sum-pos[where y = results-in t (sum-weights (map (\(s, l, t). l) xs))]*)  
**apply** *simp*  
**apply** (*auto simp: loc-imps-fw-M-nonneg[OF 2(1) assms(2)] zcount-inI [3]*)  
**apply** (*auto simp: assms(2)[unfolded inv-implications-nonneg-def]*)  
**done**  
**qed**

**definition** *impl-safe* :: (*'loc, 't*) *configuration*  $\Rightarrow$  *bool* **where**  
*impl-safe c*  $\equiv \forall \text{loc1 loc2 } t \ s. \ \text{zcount } (c\text{-imp } c \ \text{loc1}) \ t > 0 \wedge s \in_A \text{path-summary } \text{loc1 } \text{loc2}$   
 $\longrightarrow (\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \ \text{loc2}) \wedge t' \leq \text{results-in } t \ s)$

**lemma** *impl-safe*:  
**assumes** *inv-imps-work-sum* *c*  
**and** *inv-implications-nonneg* *c*  
**and**  $\bigwedge loc. c\text{-work } c \text{ loc} = \{\#\}_z$   
**shows** *impl-safe* *c*  
**unfolding** *impl-safe-def*  
**apply** *clarify*  
**proof** –  
**note** *iiws* = *assms(1)[unfolded inv-imps-work-sum-def assms(3), simplified, rule-format]*  
**fix** *loc1 loc2 t s*  
**assume**  $0 < zcount (c\text{-imp } c \text{ loc1}) t$   
**then obtain** *t'* **where**  $t' \in_A \text{frontier } (c\text{-imp } c \text{ loc1}) t' \leq t$   
**by** (*auto elim: obtain-elem-frontier*)  
**then have**  $t'\text{-zcount}: 0 < zcount (c\text{-imp } c \text{ loc1}) t'$   
**by** (*simp add: member-frontier-pos-zmset*)  
**assume** *path-sum*:  $s \in_A \text{path-summary } loc1 \text{ loc2}$   
**obtain** *M xs* **where** *Mxs*: *loc-imps-fw* *c loc1 loc2 M xs sum-path-weights xs = s*  
**by** *atomize-elim (rule path-summary-conv-loc-imps-fw[OF path-sum])*  
**have** *inM*:  $0 < zcount M (\text{results-in } t' (\text{sum-path-weights } xs))$   
**by** (*rule loc-imps-fw-implication-in-M[OF assms(1,2) Mxs(1) t'\text{-zcount}]*)  
**obtain** *u* **where**  $u \leq \text{results-in } t' (\text{sum-path-weights } xs) u \in_A \text{frontier } (c\text{-imp } c \text{ loc2})$   
**by** *atomize-elim (rule loc-imps-fw-M-in-implications[OF Mxs(1) assms(1,2,3) inM])*  
**then show**  $\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc2}) \wedge t' \leq \text{results-in } t s$   
**apply** (*intro exI[of - u]*)  
**apply** (*simp add: Mxs(2)*)  
**using** *t'(2)* **apply** (*meson order.trans results-in-mono(1)*)  
**done**  
**qed**

— Safety for states where worklist is non-empty

**lemma** *cm-preserves-impl-safe*:  
**assumes** *impl-safe* *c0*  
**and** *next-change-multiplicity'* *c0 c1 loc t n*  
**shows** *impl-safe* *c1*  
**using** *assms*  
**by** (*auto simp: impl-safe-def next-change-multiplicity'-def*)

**lemma** *cm-preserves-safe*:  
**assumes** *safe* *c0*  
**and** *impl-safe* *c0*  
**and** *next-change-multiplicity'* *c0 c1 loc t n*  
**shows** *safe* *c1*  
**proof** –  
**from** *assms(1)* **have** *safe*:  $0 < zcount (c\text{-pts } c0 \text{ loc1}) t \implies s \in_A \text{path-summary } loc1 \text{ loc2}$   
 $\implies \exists t' \leq \text{results-in } t s. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ loc2})$  **for** *loc1 loc2 t s*

```

  by (auto simp: safe-def)
  from assms(2) have impl-safe:  $0 < \text{zcount } (c\text{-imp } c0 \text{ loc1}) t \implies s \in_A \text{path-summary}$ 
  loc1 loc2
     $\implies \exists t'. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ loc2}) \wedge t' \leq \text{results-in } t \text{ s}$  for loc1 loc2 t s
  by (auto simp: impl-safe-def)
  from assms(3) haveimps:  $c\text{-imp } c1 = c\text{-imp } c0$ 
  unfolding next-change-multiplicity'-def by auto
  { fix loc1 loc2 u s — ‘safe c1’ variables
    assume u:  $0 < \text{zcount } (c\text{-pts } c1 \text{ loc1}) u$ 
    then obtain u' where u':  $u' \in_A \text{frontier } (c\text{-pts } c1 \text{ loc1}) u' \leq u$ 
      using obtain-frontier-elem by blast
    assume path-sum:  $s \in_A \text{path-summary } loc1 \text{ loc2}$ 
      — CM state changes:
    assume n-neq-zero:  $n \neq 0$ 
    assume impl:  $\exists t'. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ loc}) \wedge t' \leq t$ 
    assume pointstamps:
       $\forall loc'. c\text{-pts } c1 \text{ loc}' =$ 
        (if  $loc' = loc$  then  $\text{update-zmultiset } (c\text{-pts } c0 \text{ loc}') t n$ 
         else  $c\text{-pts } c0 \text{ loc}'$ )
    have  $\exists t' \leq \text{results-in } u \text{ s}. t' \in_A \text{frontier } (c\text{-imp } c1 \text{ loc2})$ 
    proof (cases  $n < 0$ )
      case True — Trivial, because no new pointstamp could have appeared
      with u have u-c0:  $0 < \text{zcount } (c\text{-pts } c0 \text{ loc1}) u$ 
        unfolding pointstamps[rule-format]
        by (cases  $loc1=loc$ ; cases  $t=u$ ) (auto simp: zcount-update-zmultiset)
      show ?thesis
        unfolding imps
        by (rule safe[OF u-c0 path-sum])
    next
      case False
      with n-neq-zero have  $n > 0$ 
        by linarith
      then show ?thesis
        unfolding imps
        apply (cases  $loc=loc1$ ; cases  $t=u$ )
        using impl
        apply (elim exE conjE)
        subgoal for t'
          apply simp
          apply (drule member-frontier-pos-zmset)
          apply (drule impl-safe[rotated, OF path-sum])
          apply (elim exE)
          subgoal for t''
            apply (rule exI[of - t'])
            using results-in-mono(1) order-trans apply blast
          done
        done
      using u path-sum apply (auto simp: zcount-update-zmultiset pointstamps[rule-format]
intro: safe)

```

```

    done
  qed
}
note r = this
show ?thesis
  unfolding safe-def
  apply clarify
  subgoal for loc1 loc2 t s
    using assms(3)[unfolded next-change-multiplicity'-def]
    by (intro r[of loc1 t s loc2]) auto
  done
done
qed

```

## 7.5 A Better (More Invariant) Safety

**definition** *worklists-vacant-to* :: ('loc, 't) configuration  $\Rightarrow$  't  $\Rightarrow$  bool **where**  
*worklists-vacant-to* c t =  
 $(\forall \text{loc1 loc2 s t'. s} \in_A \text{path-summary loc1 loc2} \wedge t' \in_{\#_z} \text{c-work c loc1} \longrightarrow \neg \text{results-in t' s} \leq t)$

**definition** *inv-safe* :: ('loc, 't) configuration  $\Rightarrow$  bool **where**  
*inv-safe* c =  $(\forall \text{loc1 loc2 t s. } 0 < \text{zcount (c-pts c loc1) t}$   
 $\wedge s \in_A \text{path-summary loc1 loc2}$   
 $\wedge \text{worklists-vacant-to c (results-in t s)}$   
 $\longrightarrow (\exists t' \leq \text{results-in t s. } t' \in_A \text{frontier (c-imp c loc2)}))$

Intuition: Unlike *safe*, *inv-safe* is an invariant because it only claims the safety property  $t' \in_A \text{frontier (c-imp c loc2)}$  for pointstamps that can't be modified by future propagated updates anymore (i.e. there are no upstream worklist entries which can result in a less or equal pointstamp).

**lemma** *in-frontier-diff*:  $\forall y \in_{\#_z} N. \neg y \leq x \implies x \in_A \text{frontier (M - N)} \longleftrightarrow x \in_A \text{frontier M}$   
**apply** *transfer'*  
**unfolding** *in-minimal-antichain*  
**apply** (*metis (mono-tags, lifting) diff-zero le-less mem-Collect-eq set-zmset-def zcount-diff*)  
**done**

**lemma** *worklists-vacant-to-trans*:  
*worklists-vacant-to* c t  $\implies t' \leq t \implies \text{worklists-vacant-to c t'}$   
**unfolding** *worklists-vacant-to-def*  
**using** *order.trans* **by** *blast*

**lemma** *loc-imps-fw-M-in-implications'*:  
**assumes** *loc-imps-fw c loc1 loc2 M xs*  
**and** *inv-imps-work-sum c*  
**and** *inv-implications-nonneg c*  
**and** *worklists-vacant-to c t*  
**and**  $0 < \text{zcount M t}$

```

shows  $\exists s \leq t. s \in_A \text{frontier } (c\text{-imp } c \text{ loc2})$ 
using assms(1,4,5)
proof (induct arbitrary: t rule: loc-imps-fw.induct)
  note iiws = assms(2)[unfolded inv-imps-work-sum-def, rule-format]
  case (1 loc t)
  then show ?case
    by (auto elim: obtain-elem-frontier)
next
  note iiws = assms(2)[unfolded inv-imps-work-sum-def eq-diff-eq[symmetric], rule-format]
  case (2 loc1 loc2 M xs s loc3 t)
  from 2(6) consider 0 < zcount {# results-in t s. t ∈#z M #} t | 0 < zcount
(c-imp c loc3) t
    by atomize-elim auto
  then show ?case
proof cases
  case 1
  then obtain t' where t': results-in t' s = t 0 < zcount M t'
    apply atomize-elim
    apply (rule ccontr)
    apply (subst (asm) zcount-image-zmset)
    apply (clarsimp simp: vimage-def)
    apply (metis (mono-tags, lifting) Int-iff mem-Collect-eq sum-pos-ex-elem-pos)
    done
  have vacant-to: worklists-vacant-to c t'
    apply (rule worklists-vacant-to-trans)
    apply (rule 2(5))
    using zero-le results-in-mono(2) results-in-zero t'(1) apply fastforce
    done
  obtain u where u: u ≤ t' u ∈A frontier (c-imp c loc2)
    using 2(2)[OF vacant-to t'(2)] by fast
  then have riu-le-rit': results-in u s ≤ t
    by (simp add: t'(1)[symmetric] results-in-mono)
  from u have 0 < zcount (union-frontiers c loc3) (results-in u s)
    apply (subst zcount-sum)
    apply (rule sum-pos[where y=loc2])
    apply simp-all [3]
    apply (clarsimp simp: after-summary-def)
    apply (subst zcount-sum)
    apply (rule sum-pos[where y=s])
    using 2(3) apply simp-all [3]
    apply (subst zcount-image-zmset)
    apply simp
    apply (subst card-eq-sum)
    apply (rule sum-pos[where y=u])
    apply simp-all
    done
  then have 0 < zcount (zmset-frontier (c-pts c loc3)) (results-in u s) + zcount
(union-frontiers c loc3) (results-in u s)
    by (auto intro: add-nonneg-pos)

```

```

with riu-le-rit' show ?thesis
  apply (subst (asm) zcount-union[symmetric])
  apply (subst iiws)
  apply (erule obtain-elem-frontier)
  subgoal for u'
    apply (rule exI[of - u'])
    apply (subst in-frontier-diff)
    apply clarify
  subgoal for t'
    using 2(5)[unfolded worklists-vacant-to-def, rule-format, of 0 loc3 loc3 t']
    apply -
    apply (drule meta-mp)
    apply (intro conjI)
    apply simp
    apply simp
    apply (metis order-trans results-in-zero)
    done
  apply auto
  done
done
next
case 2
then show ?thesis
  by (auto elim: obtain-elem-frontier)
qed
qed

```

```

lemma inv-safe:
  assumes inv-imps-work-sum c
    and inv-implications-nonneg c
  shows inv-safe c
  unfolding inv-safe-def
  apply clarify
proof -
  note iiws = assms(1)[unfolded inv-imps-work-sum-def, rule-format]
  fix loc1 loc2 t s
  assume vacant: worklists-vacant-to c (results-in t s)
  assume 0 < zcount (c-pts c loc1) t
  then obtain t' where t': t' ∈A frontier (c-pts c loc1) t' ≤ t
    using obtain-frontier-elem by blast
  have zcount-wl: zcount (c-work c loc1) t' = 0
    using vacant[unfolded worklists-vacant-to-def, rule-format, of 0 loc1 loc1 t',
simplified]
  by (metis add.left-neutral order.trans le-plus(1) results-in-mono(2) results-in-zero
t'(2) zcount-ne-zero-iff)
  obtain t'' where t'': t'' ∈A frontier (c-imp c loc1) t'' ≤ t'
  proof atomize-elim
    from t'(1) have 0 < zcount (zmset-frontier (c-pts c loc1)) t' + zcount
(union-frontiers c loc1) t'

```

```

    by (auto intro: add-pos-nonneg simp: union-frontiers-nonneg)
  then show  $\exists t''. t'' \in_A \text{frontier } (c\text{-imp } c \text{ loc1}) \wedge t'' \leq t'$ 
    apply (subst (asm) zcount-union[symmetric])
    apply (subst (asm) iiws[symmetric])
    using zcount-wl
    apply (auto elim: obtain-frontier-elem)
  done
qed
then have  $t''\text{-zcount}: 0 < \text{zcount } (c\text{-imp } c \text{ loc1}) t''$ 
  by (simp add: member-frontier-pos-zmset)
assume path-sum:  $s \in_A \text{path-summary loc1 loc2}$ 
obtain  $M \text{ xs}$  where  $M\text{xs}: \text{loc-imps-fw } c \text{ loc1 loc2 } M \text{ xs sum-path-weights xs} = s$ 
  by atomize-elim (rule path-summary-conv-loc-imps-fw[OF path-sum])
have  $\text{inM}: 0 < \text{zcount } M \text{ (results-in } t'' \text{ (sum-path-weights xs))}$ 
  by (rule loc-imps-fw-implication-in-M[OF assms(1,2) Mxs(1) t''-zcount])
have  $\text{vacant2}: \text{worklists-vacant-to } c \text{ (results-in } t'' \text{ (sum-weights (map } (\lambda(s, l, t).$ 
 $l) \text{ xs}))$ 
  apply (subst Mxs(2))
  apply (meson results-in-mono(1) worklists-vacant-to-trans t''(2) t'(2) vacant)
  done
obtain  $u$  where  $u: u \leq \text{results-in } t'' \text{ (sum-path-weights xs)}$   $u \in_A \text{frontier } (c\text{-imp}$ 
 $c \text{ loc2})$ 
  by atomize-elim (rule loc-imps-fw-M-in-implications'[OF Mxs(1) assms(1,2)
vacant2 inM])
  then show  $\exists t' \leq \text{results-in } t \text{ s. } t' \in_A \text{frontier } (c\text{-imp } c \text{ loc2})$ 
    apply (intro exI[of - u])
    apply (simp add: Mxs(2))
    using t''(2) t'(2) apply (meson order.trans results-in-mono(1))
  done
qed

lemma alw-conjI:  $\text{alw } P \text{ s} \implies \text{alw } Q \text{ s} \implies \text{alw } (\lambda s. P \text{ s} \wedge Q \text{ s}) \text{ s}$ 
  by (coinduction arbitrary: s) auto

lemma alw-inv-safe:  $\text{spec } s \implies \text{alw } (\text{holds inv-safe}) \text{ s}$ 
  apply (frule spec-imp-iiws)
  apply (drule alw-inv-implications-nonneg)
  apply (rule alw-mp[where  $\varphi = \lambda s. \text{holds inv-imps-work-sum } s \wedge \text{holds inv-implications-nonneg}$ 
 $s]$ )
  apply (rule alw-conjI)
  apply assumption+
  apply (simp add: alw-mono inv-safe)
  done

lemma empty-worklists-vacant-to:  $\forall \text{loc. } c\text{-work } c \text{ loc} = \{\#\}_z \implies \text{worklists-vacant-to}$ 
 $c \text{ t}$ 
  unfolding worklists-vacant-to-def
  by auto

```

**lemma** *inv-safe-safe*:  $(\bigwedge loc. c\text{-work } c \text{ loc} = \{\#\}_z) \implies \text{inv-safe } c \implies \text{safe } c$   
**unfolding** *inv-safe-def safe-def worklists-vacant-to-def* **by** *auto*

**lemma** *safe*:

**assumes** *inv-imps-work-sum* *c*  
**and** *inv-implications-nonneg* *c*  
**and**  $\bigwedge loc. c\text{-work } c \text{ loc} = \{\#\}_z$   
**shows** *safe* *c*  
**by** (*rule inv-safe-safe[OF assms(3) inv-safe[OF assms(1,2)]]*)

## 7.6 Implied Frontier

**abbreviation** *zmsset-pos* **where**  $zmsset\text{-pos } M \equiv zmsset\text{-of } (mset\text{-pos } M)$

**definition** *implied-frontier* **where**

*implied-frontier*  $P \text{ loc} = \text{frontier } (\sum loc' \in UNIV. \text{after-summary } (zmsset\text{-pos } (P \text{ loc}')) (\text{path-summary } loc' \text{ loc}))$

**definition** *implied-frontier-alt* **where**

*implied-frontier-alt*  $c \text{ loc} = \text{frontier } (\sum loc' \in UNIV. \text{after-summary } (zmsset\text{-frontier } (c\text{-pts } c \text{ loc}')) (\text{path-summary } loc' \text{ loc}))$

**lemma** *in-frontier-least*:  $x \in_A \text{frontier } M \implies \forall y. 0 < zcount \text{ } M \text{ } y \longrightarrow \neg y < x$   
**by** *transfer'* (*auto simp: minimal-antichain-def*)

**lemma** *in-frontier-trans*:  $0 < zcount \text{ } M \text{ } y \implies x \in_A \text{frontier } M \implies y \leq x \implies y \in_A \text{frontier } M$

**by** *transfer* (*simp add: le-less minimal-antichain-def*)

**lemma** *implied-frontier-alt-least*:

**assumes**  $b \in_A \text{implied-frontier-alt } c \text{ loc}2$

**shows**  $\forall loc \ a' \ s'. a' \in_A \text{frontier } (c\text{-pts } c \text{ loc}) \longrightarrow s' \in_A \text{path-summary } loc \text{ loc}2 \longrightarrow \neg \text{results-in } a' \ s' < b$

**proof** (*intro allI impI notI*)

**fix**  $loc \ a' \ s'$

**assume**  $a': a' \in_A \text{frontier } (c\text{-pts } c \text{ loc})$

**assume**  $s': s' \in_A \text{path-summary } loc \text{ loc}2$

**assume**  $lt: \text{results-in } a' \ s' < b$

**have**  $0 < zcount (\text{after-summary } (zmsset\text{-frontier } (c\text{-pts } c \text{ loc})) (\text{path-summary } loc \text{ loc}2)) (\text{results-in } a' \ s')$

**using**  $a' \ s'$  **by** (*auto intro!: pos-zcount-after-summary*)

**then have**  $0 < zcount (\sum loc' \in UNIV. \text{after-summary } (zmsset\text{-frontier } (c\text{-pts } c \text{ loc}')) (\text{path-summary } loc' \text{ loc}2)) (\text{results-in } a' \ s')$

**by** (*auto intro!: sum-pos[where y = loc] simp: zcount-sum*)

**then have**  $\text{results-in } a' \ s' \in_A \text{implied-frontier-alt } c \text{ loc}2$

**using** *assms lt* **unfolding** *implied-frontier-alt-def*

**by** (*auto dest!: in-frontier-trans[where y = results-in a' s' and x = b]*)

**with** *lt assms* **show** *False*

**unfolding** *implied-frontier-alt-def*

using *frontier-comparable-False*  
 by *blast*  
**qed**

**lemma** *implied-frontier-alt-in-pointstamps*:  
**assumes**  $b \in_A$  *implied-frontier-alt*  $c$   $loc2$   
**obtains**  $a$   $s$   $loc1$  **where**  
 $a \in_A$  *frontier* ( $c$ -pts  $c$   $loc1$ )  $s \in_A$  *path-summary*  $loc1$   $loc2$  *results-in*  $a$   $s = b$   
**using** *assms* **apply** *atomize-elim*  
**unfolding** *implied-frontier-alt-def*  
**apply** (*drule* *member-frontier-pos-zmset*)  
**apply** (*subst* (*asm*) *zcount-sum*)  
**apply** (*drule* *sum-pos-ex-elem-pos*)  
**apply** *clarify*  
**apply** (*rule* *after-summary-obtain-pre*[*rotated*])  
**apply** *simp*  
**subgoal for**  $loc1$   $a$   $s$   
**by** (*auto* *intro!*: *exI*[*of* -  $loc1$ ] *exI*[*of* -  $a$ ] *exI*[*of* -  $s$ ])  
**apply** *simp*  
**done**

**lemma** *in-implied-frontier-alt-in-implication-frontier*:  
**assumes** *inv-imps-work-sum*  $c$   
**and** *inv-implications-nonneg*  $c$   
**and** *worklists-vacant-to*  $c$   $b$   
**and**  $b \in_A$  *implied-frontier-alt*  $c$   $loc2$   
**shows**  $b \in_A$  *frontier* ( $c$ -imp  $c$   $loc2$ )  
**proof** –  
**have** *safe*:  $0 < zcount$  ( $c$ -pts  $c$   $loc1$ )  $t \implies s \in_A$  *path-summary*  $loc1$   $loc2 \implies$   
*results-in*  $t$   $s \leq b$   
 $\implies \exists t' \leq$  *results-in*  $t$   $s$ .  $t' \in_A$  *frontier* ( $c$ -imp  $c$   $loc2$ ) **for**  $loc1$   $loc2$   $t$   $s$   
**by** (*rule* *inv-safe*[*OF* *assms*(1,2), *unfolded* *inv-safe-def*, *rule-format*])  
(*auto* *elim*: *worklists-vacant-to-trans*[*OF* *assms*(3)])  
– Pointstamp  $b$  in the *implied-frontier-alt* is caused by a pointstamp  $a$  and  
summary  $s$  and *results-in*  $a$   $s$  is least among such pointstamps  
**from** *assms*(4) **obtain**  $loc1$   $a$   $s$  **where**  $loc1$ - $a$ - $s$ :  
 $a \in_A$  *frontier* ( $c$ -pts  $c$   $loc1$ )  $s \in_A$  *path-summary*  $loc1$   $loc2$  *results-in*  $a$   $s = b$   
 $\forall loc$   $a'$   $s'$ .  $a' \in_A$  *frontier* ( $c$ -pts  $c$   $loc$ )  $\implies s' \in_A$  *path-summary*  $loc$   $loc2 \implies \neg$   
*results-in*  $a'$   $s' < b$   
**apply** *atomize-elim*  
**apply** (*rule* *implied-frontier-alt-in-pointstamps*)  
**apply** *simp*  
**apply** (*drule* *implied-frontier-alt-least*)  
**apply** *fast*  
**done**  
**then have** *zcount-ps*:  $0 < zcount$  ( $c$ -pts  $c$   $loc1$ )  $a$   
**using** *member-frontier-pos-zmset* **by** *blast*  
– From ‘safe’ we know that pointstamp  $a$  is reflected in the implications by  
some pointstamp  $b' \leq b$

**obtain**  $b'$  **where**  $b': b' \in_A \text{frontier } (c\text{-imp } c \text{ loc2}) \ b' \leq \text{results-in } a \ s$   
**using**  $\text{safe}[OF \ \text{zcount-ps } \text{loc1-a-s}(2)] \ \text{loc1-a-s}(3)$  **by** *blast*  
**have**  $b' = \text{results-in } a \ s$   
**proof** –  
**have**  $\text{zcount } (c\text{-work } c \ \text{loc}) \ t = 0$  **if**  $\text{results-in } t \ s \leq b'$  **for**  $t \ s \ \text{loc}$   
**proof** –  
**have**  $\text{results-in } t \ s \leq b$   
**using**  $b'(2) \ \text{loc1-a-s}(3)$  **that** **by** *force*  
**then show** *?thesis*  
**by**  $(\text{meson } \text{assms}(3) \ \text{results-in-mono}(2) \ \text{worklists-vacant-to-def } \text{flow.zero-le}$   
*order-trans*  
*path-weight-refl zcount-inI*)  
**qed**  
– but the pointstamp can't be strictly less, because we know that *results-in*  
 $a \ s$  is least  
**then obtain**  $a' \ \text{loc1}' \ s'$  **where**  $a'$ :  
 $s' \in_A \text{path-summary } \text{loc1}' \ \text{loc2} \ \text{results-in } a' \ s' \leq b' \ a' \in_A \text{frontier } (c\text{-pts } c$   
 $\text{loc1}')$   
**using**  $\text{implication-implies-pointstamp}[OF \ b'(1) \ \text{assms}(1), \ \text{simplified}]$  **by** *force*  
**{ assume**  $b' \neq \text{results-in } a \ s$   
**with**  $b'(2)$  **have**  $b'\text{-lt}: b' < \text{results-in } a \ s$   
**by** *simp*  
**note**  $\text{loc1-a-s}(4)[\text{rule-format}, \ \text{unfolded } \text{loc1-a-s}(3)[\text{symmetric}], \ OF \ a'(\mathfrak{3},1)]$   
**with**  $b'\text{-lt } a'(\mathfrak{2})$  **have** *False*  
**by**  $(\text{simp } \text{add}: \text{leD } \text{less-le } \text{order-trans})$   
**}**  
**then show** *?thesis*  
**by**  $(\text{rule } \text{ccontr})$   
**qed**  
– Hence, the *implied-frontier-alt* pointstamp  $b$  is reflected in the implications  
**with**  $b'$  **show**  $b \in_A \text{frontier } (c\text{-imp } c \ \text{loc2})$   
**by**  $(\text{auto } \text{simp}: \text{loc1-a-s}(3))$   
**qed**

**lemma** *in-implication-frontier-in-implied-frontier-alt*:  
**assumes**  $\text{inv-imps-work-sum } c$   
**and**  $\text{inv-implications-nonneg } c$   
**and**  $\text{worklists-vacant-to } c \ b$   
**and**  $b \in_A \text{frontier } (c\text{-imp } c \ \text{loc2})$   
**shows**  $b \in_A \text{implied-frontier-alt } c \ \text{loc2}$   
**proof** –  
**have**  $\text{safe}: 0 < \text{zcount } (c\text{-pts } c \ \text{loc1}) \ t \implies s \in_A \text{path-summary } \text{loc1} \ \text{loc2} \implies$   
 $\text{results-in } t \ s \leq b$   
 $\implies \exists t' \leq \text{results-in } t \ s. \ t' \in_A \text{frontier } (c\text{-imp } c \ \text{loc2})$  **for**  $\text{loc1} \ \text{loc2} \ t \ s$   
**by**  $(\text{rule } \text{inv-safe}[OF \ \text{assms}(1,2), \ \text{unfolded } \text{inv-safe-def}, \ \text{rule-format}])$   
 $(\text{auto } \text{elim}: \text{worklists-vacant-to-trans}[OF \ \text{assms}(3)])$   
**have**  $\text{zcount } (c\text{-work } c \ \text{loc}) \ t = 0$  **if**  $\text{results-in } t \ s \leq b$  **for**  $t \ s \ \text{loc}$   
**using** **that** **by**  $(\text{meson } \text{assms}(3) \ \text{results-in-mono}(2) \ \text{worklists-vacant-to-def}$   
 $\text{flow.zero-le}$

*order-trans path-weight-refl zcount-inI*)

— Pointstamp  $b$  in the implications is caused by a pointstamp  $a$  and a summary

$s$

**then obtain**  $loc1\ a\ s$  **where**  $loc1\text{-}a\text{-}s$ :

$s \in_A\ path\text{-}summary\ loc1\ loc2\ results\text{-}in\ a\ s \leq b\ a \in_A\ frontier\ (c\text{-}pts\ c\ loc1)$

**using** *implication-implies-pointstamp*[*OF assms(4) assms(1), simplified*] **by**

*force*

**then have**  $zcount\text{-}a$ :  $0 < zcount\ (c\text{-}pts\ c\ loc1)\ a$

**using** *member-frontier-pos-zmset* **by** *blast*

**have**  $b\text{-}ria$ :  $results\text{-}in\ a\ s = b$

**proof** (*rule ccontr*)

**assume**  $results\text{-}in\ a\ s \neq b$

**with**  $loc1\text{-}a\text{-}s(2)$  **have**  $results\text{-}in\ a\ s < b$

**by** *simp*

**then show** *False*

**using** *safe*[*OF zcount-a loc1-a-s(1)*] *assms(4) loc1-a-s(2)*

**using** *order.strict-trans1 frontier-comparable-False* **by** *blast*

**qed**

—  $results\text{-}in\ a\ s$  is a candidate for inclusion in the *implied-frontier-alt*.

**have**  $0 < zcount\ (\sum_{loc' \in UNIV.\ after\text{-}summary\ (zmset\text{-}frontier\ (c\text{-}pts\ c\ loc'))\ (path\text{-}summary\ loc'\ loc2))\ (results\text{-}in\ a\ s)$

**unfolding** *after-summary-def*

**apply** (*subst zcount-sum*)

**apply** (*rule sum-pos*[*of - - loc1*])

**apply** *simp*

**apply** (*clarsimp simp: zcount-sum*)

**apply** (*rule sum-nonneg*)

**apply** (*subst zcount-image-zmset*)

**apply** *auto* [2]

**apply** (*subst zcount-sum*)

**apply** (*rule sum-pos*[*of - - s*])

**using**  $loc1\text{-}a\text{-}s(1)$  **apply** *simp-all* [3]

**apply** (*subst zcount-image-zmset*)

**apply** (*rule sum-pos*[*of - - a*])

**using**  $loc1\text{-}a\text{-}s(3)$  **apply** *auto*

**done**

— ..which means a pointstamp  $b' \leq results\text{-}in\ a\ s$  must exist in the *implied-frontier-alt*.

**then obtain**  $b'$  **where**  $b' \in_A\ implied\text{-}frontier\text{-}alt\ c\ loc2\ b' \leq results\text{-}in\ a\ s$

**by** (*auto simp: implied-frontier-alt-def elim: obtain-frontier-elem*)

**then have** *worklists-vacant-to*  $c\ b'$

**using**  $loc1\text{-}a\text{-}s(2)$  **by** (*auto intro: worklists-vacant-to-trans*[*OF assms(3)*])

**with**  $b'$  **have**  $b'\text{-}frontier$ :  $b' \in_A\ frontier\ (c\text{-}imp\ c\ loc2)$

**using** *in-implied-frontier-alt-in-implication-frontier assms* **by** *blast*

— ..and this pointstamp must be equal to  $b'$

**have**  $b'\text{-}ria$ :  $b' = results\text{-}in\ a\ s$

**proof** (*rule ccontr*)

**assume**  $b' \neq results\text{-}in\ a\ s$

**with**  $b'(2)$  **have**  $b'\text{-}lt$ :  $b' < results\text{-}in\ a\ s$

by *simp*  
 from *b'-frontier b'-lt b-ria assms(4)* **show** *False*  
 using *frontier-comparable-False* **by** *blast*  
**qed**  
 — Hence, the implication frontier pointstamp *b* is reflected in the *implied-frontier-alt*  
 from *b' b'-ria b-ria* **show**  $b \in_A \text{implied-frontier-alt } c \text{ loc2}$   
 by (*auto simp: implied-frontier-alt-def*)  
**qed**

**lemma** *implication-frontier-iff-implied-frontier-alt-vacant:*  
 assumes *inv-imps-work-sum c*  
 and *inv-implications-nonneg c*  
 and *worklists-vacant-to c b*  
 shows  $b \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \longleftrightarrow b \in_A \text{implied-frontier-alt } c \text{ loc}$   
 using  
*ac-eq-iff*  
*in-implication-frontier-in-implied-frontier-alt[OF assms]*  
*in-implied-frontier-alt-in-implication-frontier[OF assms]*  
 by *blast*

**lemma** *next-propagate-implied-frontier-alt-def:*  
*next-propagate c c'  $\implies$  implied-frontier-alt c loc = implied-frontier-alt c' loc*  
 by (*auto simp: next-propagate'-def implied-frontier-alt-def*)

**lemma** *implication-frontier-eq-implied-frontier-alt:*  
 assumes *inv-imps-work-sum c*  
 and *inv-implications-nonneg c*  
 and  $\bigwedge \text{loc. } c\text{-work } c \text{ loc} = \{\#\}_z$   
 shows  $\text{frontier } (c\text{-imp } c \text{ loc}) = \text{implied-frontier-alt } c \text{ loc}$   
 using *ac-eq-iff implication-frontier-iff-implied-frontier-alt-vacant empty-worklists-vacant-to*  
*assms*  
 by *blast*

**lemma** *alw-implication-frontier-eq-implied-frontier-alt-empty: spec s  $\implies$*   
*alw (holds ( $\lambda c. (\forall \text{loc. } c\text{-work } c \text{ loc} = \{\#\}_z) \longrightarrow \text{frontier } (c\text{-imp } c \text{ loc}) = \text{im-}$*   
*plied-frontier-alt } c \text{ loc})) s*  
 apply (*frule spec-imp-iiws*)  
 apply (*drule alw-inv-implications-nonneg*)  
 apply (*drule (1) alw-conjI*)  
 apply (*erule thin-rl*)  
 apply (*auto elim!: alw-mono simp: implication-frontier-eq-implied-frontier-alt*)  
**done**

**lemma** *alw-implication-frontier-eq-implied-frontier-alt-vacant: spec s  $\implies$*   
*alw (holds ( $\lambda c. \text{worklists-vacant-to } c \text{ b} \longrightarrow b \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \longleftrightarrow b \in_A$*   
*implied-frontier-alt } c \text{ loc})) s*  
 apply (*frule spec-imp-iiws*)  
 apply (*drule alw-inv-implications-nonneg*)  
 apply (*drule (1) alw-conjI*)

**apply** (*erule thin-rl*)  
**apply** (*auto elim! : alw-mono simp: implication-frontier-iff-implied-frontier-alt-vacant*)  
**done**

**lemma** *antichain-eqI*:  $(\bigwedge b. b \in_A A \longleftrightarrow b \in_A B) \implies A = B$   
**by** *transfer auto*

**lemma** *zmset-frontier-zmset-pos*:  $zmset\text{-frontier } A \subseteq\#_z zmset\text{-pos } A$   
**unfolding** *subseteq-zmset-def*  
**by** *transfer' (auto simp: count-mset-set-if minimal-antichain-def)*

**lemma** *image-mset-mono-pos*:  
 $\forall b. 0 \leq zcount A b \implies \forall b. 0 \leq zcount B b \implies A \subseteq\#_z B \implies image\text{-zmset } f A \subseteq\#_z image\text{-zmset } f B$   
**unfolding** *subseteq-zmset-def zcount-image-zmset*  
**apply** (*intro allI*)  
**apply** (*rule order-trans[OF sum-mono sum-mono2]*)  
**apply** *simp-all*  
**by** (*metis Int-subset-iff antisym subsetI zcount-ne-zero-iff*)

**lemma** *sum-mono-subseteq*:  
 $(\bigwedge i. i \in K \implies f i \subseteq\#_z g i) \implies (\sum i \in K. f i) \subseteq\#_z (\sum i \in K. g i)$   
**by** (*simp add: subseteq-zmset-def sum-mono zcount-sum*)

**lemma** *after-summary-zmset-frontier*:  
 $after\text{-summary } (zmset\text{-frontier } A) S \subseteq\#_z after\text{-summary } (zmset\text{-pos } A) S$   
**unfolding** *after-summary-def*  
**apply** (*rule sum-mono-subseteq*)  
**apply** (*rule image-mset-mono-pos[OF - - zmset-frontier-zmset-pos]*)  
**apply** *auto*  
**done**

**lemma** *frontier-eqI*:  $\forall b. 0 \leq zcount A b \implies \forall b. 0 \leq zcount B b \implies A \subseteq\#_z B \implies (\bigwedge b. b \in\#_z B \implies \exists a. a \in\#_z A \wedge a \leq b) \implies frontier A = frontier B$   
**apply** (*transfer fixing: A B*)  
**apply** (*clarsimp simp: minimal-antichain-def subseteq-zmset-def*)  
**apply** *safe*  
**apply** (*metis less-le-trans*)  
**apply** (*metis less-le less-le-trans zcount-ne-zero-iff*)  
**apply** (*metis antisym less-le zcount-ne-zero-iff*)  
**apply** (*meson less-le-trans*)  
**done**

**lemma** *implied-frontier-implied-frontier-alt*:  $implied\text{-frontier } (c\text{-pts } c) loc = implied\text{-frontier-alt } c loc$   
**unfolding** *implied-frontier-alt-def implied-frontier-def*  
**apply** (*rule frontier-eqI[symmetric]*)  
**subgoal** **by** (*auto simp: zcount-sum sum-nonneg*)

```

subgoal by (auto simp: zcount-sum sum-nonneg)
subgoal by (rule sum-mono-subseteq[OF after-summary-zmset-frontier])
apply (simp flip: zcount-ne-zero-iff add: zcount-sum)
apply (erule sum.not-neutral-contains-not-neutral)
apply (simp flip: zcount-ne-zero-iff add: after-summary-def zcount-sum)
apply (erule sum.not-neutral-contains-not-neutral)
apply (simp flip: zcount-ne-zero-iff add: zcount-image-zmset split: if-splits)
apply (erule sum.not-neutral-contains-not-neutral)
apply (simp flip: zcount-ne-zero-iff)
subgoal for u loc s t
  apply (rule obtain-elem-frontier[of c-pts c loc t])
  apply (metis le-less)
  subgoal for a
    apply (rule exI[of - results-in a s])
    apply (rule conjI[rotated])
    using results-in-mono(1) apply blast
    apply (subst sum-nonneg-eq-0-iff; simp add: sum-nonneg)
    apply (rule exI[of - loc])
    apply (subst sum-nonneg-eq-0-iff; simp)
    apply (rule bexI[of - s])
    apply auto
  done
done
done

```

```

lemmas alw-implication-frontier-eq-implied-frontier-vacant =
  alw-implication-frontier-eq-implied-frontier-alt-vacant[folded implied-frontier-implied-frontier-alt]
lemmas implication-frontier-iff-implied-frontier-vacant =
  implication-frontier-iff-implied-frontier-alt-vacant[folded implied-frontier-implied-frontier-alt]

```

**end**

## 8 Combined Progress Tracking Protocol

```

lemma fold-invar:
  assumes finite M
  and P z
  and  $\forall z. \forall x \in M. P z \longrightarrow P (f x z)$ 
  and comp-fun-commute f
  shows  $P (Finite-Set.fold f z M)$ 
proof –
  interpret commute: comp-fun-commute f
  by (fact <comp-fun-commute f>)
  from assms show ?thesis
  by (induct M arbitrary: z rule: finite-induct) auto
qed

```

## 8.1 Could-result-in Relation

**context** *dataflow-topology* **begin**

**definition** *cri-less-eq* :: ('loc × 't) ⇒ ('loc × 't) ⇒ bool (↖-≤<sub>p</sub>-↗ [51,51] 50) **where**  
*cri-less-eq* =  
 (λ(loc1,t1) (loc2,t2). (∃ s. s ∈<sub>A</sub> path-summary loc1 loc2 ∧ results-in t1 s ≤ t2))

**definition** *cri-less* :: ('loc × 't) ⇒ ('loc × 't) ⇒ bool (↖-<<sub>p</sub>-↗ [51,51] 50) **where**  
*cri-less* x y = (x ≤<sub>p</sub> y ∧ x ≠ y)

**lemma** *cri-asym1*: x <<sub>p</sub> y ⟶ ¬ y <<sub>p</sub> x

**for** x y **apply** (cases x; cases y)

**proof** (rule ccontr)

**fix** loc1 t1 loc2 t2

**assume** as: ¬ (x <<sub>p</sub> y ⟶ ¬ y <<sub>p</sub> x) x = (loc1, t1) y = (loc2, t2)

**then have** as1: loc1 ≠ loc2

**unfolding** *cri-less-def cri-less-eq-def*

**by** *clarsimp*

(metis add.right-neutral order.antisym order.trans le-plus(2) results-in-mono(2) results-in-zero)

**from** as **obtain** s1 **where** s1: s1 ∈<sub>A</sub> path-summary loc1 loc2  
 results-in t1 s1 ≤ t2

**using** *cri-less-def cri-less-eq-def* **by** *auto*

**then obtain** path1 **where** path1: flow.path loc1 loc2 path1

s1 = flow.sum-path-weights path1

path1 ≠ []

**using** as1 flow.path-weight-conv-path flow.path0E **by** *blast*

**from** as **obtain** s2 **where** s2: s2 ∈<sub>A</sub> path-summary loc2 loc1  
 results-in t2 s2 ≤ t1

**using** *cri-less-def cri-less-eq-def* **by** *auto*

**then obtain** path2 **where** path2: flow.path loc2 loc1 path2

s2 = flow.sum-path-weights path2

path2 ≠ []

**using** as1 flow.path-weight-conv-path flow.path0E **by** *blast*

**with** path1 **have** path-total: flow.path loc1 loc1 (path1@path2)

**using** flow.path-trans path2(3) **by** *blast*

**then obtain** s3 **where** s3: s3 = flow.sum-path-weights (path1@path2) **by** *auto*

**then have** s3-sum: s3 = followed-by s1 s2

**using** path1 path2 flow.sum-weights-append **by** *auto*

**have** ¬ t1 < results-in t1 s3

**using** s3-sum s1(2) s2(2) followed-by-summary

**by** (metis le-less-trans less-le-not-le results-in-mono(1))

**then show** False **using** path-total no-zero-cycle s3 path1(3) path2(3) **by** *blast*

**qed**

**lemma** *cri-asym2*: x <<sub>p</sub> y ⟶ x ≠ y

**by** (*simp add: cri-less-def*)

**sublocale** *cri*: order *cri-less-eq cri-less*

```

apply standard
subgoal
  using cri-asym1 cri-asym2 cri-less-def by blast
subgoal for x
  unfolding cri-less-eq-def
  using flow.path-weight-refl results-in-zero by fastforce
subgoal
  for x y z apply (cases x; cases y; cases z)
proof –
  fix a b aa ba ab bb
  assume as: x ≤p y y ≤p z x = (a, b) y = (aa, ba) z = (ab, bb)
  then obtain s1 where s1: s1 ∈A path-summary a aa results-in b s1 ≤ ba
    using cri-less-eq-def by auto
  from as(2,4,5) obtain s2 where s2: s2 ∈A path-summary aa ab results-in ba
s2 ≤ bb
    using cri-less-eq-def by auto
  with s1 obtain s3 where s3: s3 ∈A path-summary a ab s3 ≤ followed-by s1
s2
    using flow.path-weight-elem-trans by blast
  with s1 s2 have results-in b s3 ≤ bb
proof –
  have  $\forall s. \text{results-in } (results-in b s1) s \leq \text{results-in } ba s$ 
    by (meson results-in-mono(1) s1(2))
  then show ?thesis
    by (metis (no-types) results-in-mono(2) followed-by-summary order-trans
s2(2) s3(2))
  qed
  with as s3 show ?thesis unfolding cri-less-eq-def by blast
qed
using cri-asym1 cri-asym2 cri-less-def by blast

lemma wf-cri: wf {(l, l'). (l, t) <p (l', t)}
by (rule finite-acyclic-wf)
  (auto intro: cri.acyclicI-order[THEN acyclic-converse[THEN iffD1]])

end

```

## 8.2 Specification

### 8.2.1 Configuration

```

record ('p::finite, 't::order, 'loc) configuration =
  exchange-config :: ('p, ('loc × 't)) Exchange.configuration
  prop-config :: 'p ⇒ ('loc, 't) Propagate.configuration
  init :: 'p ⇒ bool

```

```

type-synonym ('p, 't, 'loc) computation = ('p, 't, 'loc) configuration stream

```

```

context dataflow-topology begin

```

**definition** *the-cm* **where**

*the-cm*  $c$   $loc$   $t$   $n = (THE\ c'.\ next\ change\ multiplicity'\ c\ c'\ loc\ t\ n)$

*the-cm* is not commutative in general, only if the necessary conditions hold. It can be converted to *apply-cm* for which we prove *comp-fun-commute*.

**definition** *apply-cm* **where**

*apply-cm*  $c$   $loc$   $t$   $n =$   
 (let *new-pointstamps* = ( $\lambda loc'$ .  
     (if  $loc' = loc$  then *update-zmultiset* (*c-pts*  $c$   $loc'$ )  $t$   $n$   
       else *c-pts*  $c$   $loc'$ )) in  
 $c$  ( $\mid$  *c-pts* := *new-pointstamps*  $\mid$ )  
 $\mid$  *c-work* :=  
     ( $\lambda loc'$ . *c-work*  $c$   $loc'$  + *frontier-changes* (*new-pointstamps*  $loc'$ ) (*c-pts*  $c$   
 $loc'$ ))))))

**definition** *cm-all'* **where**

*cm-all'*  $c0$   $\Delta =$   
*Finite-Set.fold* ( $\lambda(loc, t)$   $c.$  *apply-cm*  $c$   $loc$   $t$  (*zcount*  $\Delta$  ( $loc, t$ )))  $c0$  (*set-zmset*  
 $\Delta$ )

**definition** *cm-all* **where**

*cm-all*  $c0$   $\Delta =$   
*Finite-Set.fold* ( $\lambda(loc, t)$   $c.$  *the-cm*  $c$   $loc$   $t$  (*zcount*  $\Delta$  ( $loc, t$ )))  $c0$  (*set-zmset*  
 $\Delta$ )

**definition** *propagate-all*  $c0 = while\ option\ (\lambda c. \exists loc. (c\ work\ c\ loc) \neq \{\#\}_z)$

$(\lambda c. SOME\ c'. \exists loc\ t. next\ propagate'\ c\ c'\ loc$   
 $t)$   $c0$

## 8.2.2 Initial state and state transitions

**definition** *InitConfig* :: ( $'p::finite, 't::order, 'loc$ ) *configuration*  $\Rightarrow bool$  **where**

*InitConfig*  $c =$   
 ( $\forall p.$  *init*  $c$   $p = False$ )  
 $\wedge$  *cri.init-config* (*exchange-config*  $c$ )  
 $\wedge$  ( $\forall p\ loc\ t.$  *zcount* (*c-pts* (*prop-config*  $c$   $p$ )  $loc$ )  $t$   
 = *zcount* (*c-glob* (*exchange-config*  $c$ )  $p$ ) ( $loc, t$ )  
 $\wedge$  ( $\forall w.$  *init-config* (*prop-config*  $c$   $w$ )))

**definition** *NextPerformOp'* :: ( $'p::finite, 't::order, 'loc$ ) *configuration*  $\Rightarrow ('p, 't,$   
 $'loc)$  *configuration*

$\Rightarrow 'p \Rightarrow ('loc \times 't)$  *multiset*  $\Rightarrow ('p \times ('loc \times 't))$  *multiset*  
 $\Rightarrow ('loc \times 't)$  *multiset*  $\Rightarrow bool$  **where**

*NextPerformOp'*  $c0\ c1\ p\ \Delta_{neg}\ \Delta_{mint-msg}\ \Delta_{mint-self} =$   
*cri.next-performop'* (*exchange-config*  $c0$ ) (*exchange-config*  $c1$ )  $p\ \Delta_{neg}\ \Delta_{mint-msg}$   
 $\Delta_{mint-self}$   
 $\wedge$  *unchanged prop-config*  $c0\ c1$   
 $\wedge$  *unchanged init*  $c0\ c1$ )

**abbreviation** *NextPerformOp* **where**

$NextPerformOp\ c0\ c1 \equiv \exists p\ \Delta neg\ \Delta mint\text{-}msg\ \Delta mint\text{-}self. NextPerformOp'\ c0\ c1\ p\ \Delta neg\ \Delta mint\text{-}msg\ \Delta mint\text{-}self$

**definition** *NextRecvCap'*

$:: ('p::finite, 't::order, 'loc)\ configuration \Rightarrow ('p, 't, 'loc)\ configuration \Rightarrow 'p \Rightarrow 'loc \times 't \Rightarrow bool$  **where**

$NextRecvCap'\ c0\ c1\ p\ t = (\$   
 $\quad cri.next\text{-}recvcap'\ (exchange\text{-}config\ c0)\ (exchange\text{-}config\ c1)\ p\ t$   
 $\quad \wedge\ unchanged\ prop\text{-}config\ c0\ c1$   
 $\quad \wedge\ unchanged\ init\ c0\ c1)$

**abbreviation** *NextRecvCap* **where**

$NextRecvCap\ c0\ c1 \equiv \exists p\ t. NextRecvCap'\ c0\ c1\ p\ t$

**definition** *NextSendUpd'*  $:: ('p::finite, 't::order, 'loc)\ configuration \Rightarrow ('p, 't, 'loc)\ configuration$

$\Rightarrow 'p \Rightarrow ('loc \times 't)\ set \Rightarrow bool$  **where**

$NextSendUpd'\ c0\ c1\ p\ tt = (\$   
 $\quad cri.next\text{-}sendupd'\ (exchange\text{-}config\ c0)\ (exchange\text{-}config\ c1)\ p\ tt$   
 $\quad \wedge\ unchanged\ prop\text{-}config\ c0\ c1$   
 $\quad \wedge\ unchanged\ init\ c0\ c1)$

**abbreviation** *NextSendUpd* **where**

$NextSendUpd\ c0\ c1 \equiv \exists p\ tt. NextSendUpd'\ c0\ c1\ p\ tt$

**definition** *NextRecvUpd'*  $:: ('p::finite, 't::order, 'loc)\ configuration \Rightarrow ('p, 't, 'loc)\ configuration$

$\Rightarrow 'p \Rightarrow 'p \Rightarrow bool$  **where**

$NextRecvUpd'\ c0\ c1\ p\ q = (\$   
 $\quad init\ c0\ q$  — Once init is set we are guaranteed that the CM transitions' premises are satisfied

$\quad \wedge\ cri.next\text{-}recvupd'\ (exchange\text{-}config\ c0)\ (exchange\text{-}config\ c1)\ p\ q$   
 $\quad \wedge\ unchanged\ init\ c0\ c1$   
 $\quad \wedge\ (\forall p'. prop\text{-}config\ c1\ p' =$   
 $\quad\quad (if\ p' = q$   
 $\quad\quad\quad then\ cm\text{-}all\ (prop\text{-}config\ c0\ q)\ (hd\ (c\text{-}msg\ (exchange\text{-}config\ c0)\ p\ q))$   
 $\quad\quad\quad else\ prop\text{-}config\ c0\ p')))$

**abbreviation** *NextRecvUpd* **where**

$NextRecvUpd\ c0\ c1 \equiv \exists p\ q. NextRecvUpd'\ c0\ c1\ p\ q$

**definition** *NextPropagate'*  $:: ('p::finite, 't::order, 'loc)\ configuration \Rightarrow ('p, 't, 'loc)\ configuration$

$\Rightarrow 'p \Rightarrow bool$  **where**

$NextPropagate'\ c0\ c1\ p = (\$   
 $\quad unchanged\ exchange\text{-}config\ c0\ c1$   
 $\quad \wedge\ init\ c1 = (init\ c0)(p := True)$   
 $\quad \wedge\ (\forall p'. Some\ (prop\text{-}config\ c1\ p') =$

(if  $p' = p$   
 then *propagate-all* (*prop-config*  $c0$   $p'$ )  
 else *Some* (*prop-config*  $c0$   $p'$ )))

**abbreviation** *NextPropagate* **where**

$NextPropagate\ c0\ c1 \equiv \exists p. NextPropagate'\ c0\ c1\ p$

**definition** *Next'* **where**

$Next'\ c0\ c1 = (NextPerformOp\ c0\ c1 \vee NextSendUpd\ c0\ c1 \vee NextRecvUpd\ c0\ c1 \vee NextPropagate\ c0\ c1 \vee NextRecvCap\ c0\ c1 \vee c1 = c0)$

**abbreviation** *Next* **where**

$Next\ s \equiv Next'\ (shd\ s)\ (shd\ (stl\ s))$

**definition** *FullSpec* :: ( $'p :: finite, 't :: order, 'loc$ ) *computation*  $\Rightarrow bool$  **where**

$FullSpec\ s = (holds\ InitConfig\ s \wedge alw\ Next\ s)$

**lemma** *NextPerformOpD*:

**assumes**  $NextPerformOp'\ c0\ c1\ p\ \Delta_{neg}\ \Delta_{mint-msg}\ \Delta_{mint-self}$

**shows**

$cri.next-performop'\ (exchange-config\ c0)\ (exchange-config\ c1)\ p\ \Delta_{neg}\ \Delta_{mint-msg}\ \Delta_{mint-self}$

*unchanged prop-config*  $c0\ c1$

*unchanged init*  $c0\ c1$

**using** *assms* **unfolding** *NextPerformOp'-def* **by** *simp-all*

**lemma** *NextSendUpdD*:

**assumes**  $NextSendUpd'\ c0\ c1\ p\ tt$

**shows**

$cri.next-sendupd'\ (exchange-config\ c0)\ (exchange-config\ c1)\ p\ tt$

*unchanged prop-config*  $c0\ c1$

*unchanged init*  $c0\ c1$

**using** *assms* **unfolding** *NextSendUpd'-def* **by** *simp-all*

**lemma** *NextRecvUpdD*:

**assumes**  $NextRecvUpd'\ c0\ c1\ p\ q$

**shows**

*init*  $c0\ q$

$cri.next-recvupd'\ (exchange-config\ c0)\ (exchange-config\ c1)\ p\ q$

*unchanged init*  $c0\ c1$

$(\forall p'. prop-config\ c1\ p' =$

(if  $p' = q$

then *cm-all* (*prop-config*  $c0\ q$ ) (*hd* (*c-msg* (*exchange-config*  $c0$ )  $p\ q$ ))

else *prop-config*  $c0\ p')$ )

**using** *assms* **unfolding** *NextRecvUpd'-def* **by** *simp-all*

**lemma** *NextPropagateD*:

**assumes**  $NextPropagate'\ c0\ c1\ p$

**shows**

*unchanged exchange-config c0 c1*  
*init c1 = (init c0)(p := True)*  
*( $\forall p'. \text{Some (prop-config c1 p')} =$*   
*(if p' = p*  
*then propagate-all (prop-config c0 p')*  
*else Some (prop-config c0 p')))*  
**using** *assms unfolding NextPropagate'-def by simp-all*

**lemma** *NextRecvCapD:*  
**assumes** *NextRecvCap' c0 c1 p t*  
**shows**  
*cri.next-recvcap' (exchange-config c0) (exchange-config c1) p t*  
*unchanged prop-config c0 c1*  
*unchanged init c0 c1*  
**using** *assms unfolding NextRecvCap'-def by simp-all*

## 8.3 Auxiliary Lemmas

### 8.3.1 Auxiliary Lemmas for CM Conversion

**lemma** *apply-cm-is-cm:*  
 $\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \wedge t' \leq t \implies n \neq 0 \implies \text{next-change-multiplicity}'$   
 $c \text{ (apply-cm } c \text{ loc } t \text{ n) loc } t \text{ n}$   
**by** *(auto simp: next-change-multiplicity'-def apply-cm-def*  
*intro!: Propagate.configuration.equality)*

**lemma** *update-zmultiset-commute:*  
 $\text{update-zmultiset } (\text{update-zmultiset } M \text{ t' n}') \text{ t n} = \text{update-zmultiset } (\text{update-zmultiset}$   
 $M \text{ t n) t' n'}$   
**by** *transfer (auto simp: equiv-zmset-def split: if-splits)*

**lemma** *apply-cm-commute: apply-cm (apply-cm c loc t n) loc' t' n' = apply-cm*  
*(apply-cm c loc' t' n') loc t n*  
**unfolding** *apply-cm-def Let-def*  
**by** *(auto intro!: Propagate.configuration.equality simp: update-zmultiset-commute)*

**lemma** *comp-fun-commute-apply-cm[simp]: comp-fun-commute ( $\lambda(\text{loc}, t) c. \text{ap-}$*   
*ply-cm } c \text{ loc } t \text{ (f loc t))*  
**by** *(auto intro!: Propagate.configuration.equality simp: update-zmultiset-commute*  
*comp-fun-commute-def o-def apply-cm-commute)*

**lemma** *ex-cm-imp-conds:*  
**assumes**  $\exists c'. \text{next-change-multiplicity}' c' c' \text{ loc } t \text{ n}$   
**shows**  $\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \wedge t' \leq t \text{ n} \neq 0$   
**using** *assms by (auto simp: next-change-multiplicity'-def)*

**lemma** *the-cm-eq-apply-cm:*  
**assumes**  $\exists c'. \text{next-change-multiplicity}' c' c' \text{ loc } t \text{ n}$   
**shows** *the-cm } c \text{ loc } t \text{ n} = \text{apply-cm } c \text{ loc } t \text{ n}*  
**proof** –

```

from assms have cond:  $\exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \wedge t' \leq t \ n \neq 0$ 
  using ex-cm-imp-conds by blast+
show ?thesis
  unfolding the-cm-def
  apply (rule the1-equality)
  apply (rule next-change-multiplicity'-unique[OF cond(2,1)])
  unfolding apply-cm-def next-change-multiplicity'-def
  using cond apply (auto intro!: Propagate.configuration.equality)
  done
qed

lemma apply-cm-preserves-cond:
  assumes  $\forall (loc,t) \in \text{set-zmset } \Delta. \exists t'. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ loc}) \wedge t' \leq t$ 
  shows  $\forall (loc,t) \in \text{set-zmset } \Delta. \exists t'. t' \in_A \text{frontier } (c\text{-imp } (\text{apply-cm } c0 \text{ loc } t' \ n) \text{ loc}) \wedge t' \leq t$ 
  using assms by (auto simp: apply-cm-def)

lemma cm-all-eq-cm-all':
  assumes  $\forall (loc,t) \in \text{set-zmset } \Delta. \exists t'. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ loc}) \wedge t' \leq t$ 
  shows  $\text{cm-all } c0 \ \Delta = \text{cm-all}' \ c0 \ \Delta$ 
  unfolding cm-all-def cm-all'-def
  apply (rule fold-closed-eq[where B = {c.  $\forall (loc,t) \in \text{set-zmset } \Delta. \exists t'. t' \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \wedge t' \leq t$ }])
  subgoal for a  $\Delta$ 
    apply (cases a)
    apply simp
    apply (rule the-cm-eq-apply-cm)
    apply (rule ex1-implies-ex)
    apply (rule next-change-multiplicity'-unique)
    apply auto
    done
  subgoal for a  $\Delta$ 
    apply (cases a)
    apply simp
    apply (rule apply-cm-preserves-cond)
    apply auto
    done
  subgoal
    using assms by simp
  done

lemma cm-eq-the-cm:
  assumes next-change-multiplicity' c c' loc t n
  shows the-cm c loc t n = c'
proof –
  from assms have cond:  $\exists u. u \in_A \text{frontier } (c\text{-imp } c \text{ loc}) \wedge u \leq t \ n \neq 0$ 
  unfolding next-change-multiplicity'-def by auto
  then show ?thesis
    using next-change-multiplicity'-unique[OF cond(2,1)] assms the-cm-def

```

by auto  
qed

**lemma** *zcount-ps-apply-cm*:

$zcount (c\text{-pts} (apply\text{-cm} c loc t n) loc') t' = zcount (c\text{-pts} c loc') t' + (if loc = loc' \wedge t = t' then n else 0)$

by (auto simp: apply-cm-def zcount-update-zmultiset)

**lemma** *zcount-pointstamps-update*:  $zcount (c\text{-pts} (c\{c\text{-pts}=M\}) loc) x = zcount (M loc) x$

by auto

**lemma** *nop*:  $loc1 \neq loc2 \vee t1 \neq t2 \longrightarrow$

$zcount (c\text{-pts} (apply\text{-cm} c loc2 t2 (zcount \Delta (loc2, t2))) loc1) t1 = zcount (c\text{-pts} c loc1) t1$

apply (simp add: apply-cm-def)

using zcount-update-zmultiset

by (simp add: zcount-update-zmultiset)

**lemma** *fold-nop*:

$zcount (c\text{-pts} (Finite\text{-Set.fold} (\lambda(loc', t') c. apply\text{-cm} c loc' t' (zcount \Delta' (loc', t')))) c$

$(set\text{-zmset} \Delta - \{(loc, t)\})) loc) t$

$= zcount (c\text{-pts} c loc) t$

**proof** –

have finite (set-zmset  $\Delta - \{(loc, t)\}$ ) using finite-set-zmset by blast

then show ?thesis

**proof** (induct set-zmset  $\Delta - \{(loc, t)\}$  arbitrary:  $\Delta$  rule: finite-induct)

case empty

then show ?case using fold-empty by (auto simp del: diff-shunt)

next

let ?f =  $(\lambda(loc', t') c. apply\text{-cm} c loc' t' (zcount \Delta' (loc', t')))$

case (insert x F)

obtain loc' t' where x-pair:  $x = (loc', t')$  by (meson surj-pair)

from insert have nonmember:  $x \neq (loc, t)$  by auto

then have finite-s: finite F using insert by auto

interpret commute: comp-fun-commute ?f

by (simp add: comp-fun-commute-def comp-def apply-cm-commute)

from finite-s have step1:

$Finite\text{-Set.fold} ?f c (insert x F) = ?f x (Finite\text{-Set.fold} ?f c F)$

by (metis (mono-tags, lifting) commute.fold-insert insert.hyps(1) insert.hyps(2))

from nonmember have step2:

$zcount (c\text{-pts} (?f x (Finite\text{-Set.fold} ?f c F)) loc) t$

$= zcount (c\text{-pts} (Finite\text{-Set.fold} ?f c F) loc) t$

using x-pair

by (metis (mono-tags, lifting) case-prod-conv nop)

with step1 and x-pair have final:

$zcount (c\text{-pts} (Finite\text{-Set.fold} ?f c (insert x F)) loc) t$

$= zcount (c\text{-pts} (Finite\text{-Set.fold} ?f c F) loc) t$

```

    by simp
    from insert(2,4) obtain Δ2 where Δ2: Δ2 = filter-zmset (λy. y≠x) Δ by
blast
    then have F = set-zmset Δ2 - {(loc, t)}
    proof -
      from Δ2 have *: x ∉#z Δ2 by (simp add: not-in-iff-zmset)
      from insert(4) and nonmember have **: x ∈#z Δ by blast
      from Δ2 ** have ***: ∀ y. y ∈#z Δ ↔ (y ∈#z Δ2 ∨ y = x)
        by (metis (mono-tags, lifting) count-filter-zmset zcount-ne-zero-iff)
      with *** have ∀ y. (y ∈ set-zmset Δ = (y ∈ (set-zmset Δ2 ∪ {x}))) by blast
      then have set-zmset Δ = set-zmset Δ2 ∪ {x} by (auto simp add: set-eq-iff)
      with insert(2,3,4) * show ?thesis
        by (metis (mono-tags, lifting) Diff-insert Diff-insert2 Diff-insert-absorb
Un-empty-right Un-insert-right)
    qed
    with final insert show ?case by metis
  qed
qed

```

**lemma** *zcount-pointstamps-cm-all'*:

```

zcount (c-pts (cm-all' c Δ) loc) x
= zcount (c-pts c loc) x + zcount Δ (loc,x)

```

**proof** -

```

let ?f = (λ(loc', t') c. apply-cm c loc' t' (zcount Δ (loc',t')))
have ?thesis
proof (cases zcount Δ (loc,x) = 0)
  case case-nonmember: True
  then have set-zmsetΔ: set-zmset Δ - {(loc, x)} = set-zmset Δ using zcount-eq-zero-iff
by fastforce
  have zcount (c-pts (cm-all' c Δ) loc) x
    = zcount (c-pts c loc) x
  unfolding cm-all'-def
  apply (subst set-zmsetΔ[symmetric])
  apply (simp add: fold-nop)
  done
  with case-nonmember show ?thesis by auto
next
  case case-member: False
  then have fold-rec: Finite-Set.fold ?f c (set-zmset Δ)
    = ?f (loc, x) (Finite-Set.fold ?f c (set-zmset Δ - {(loc, x)}))
  proof -
    interpret commute: comp-fun-commute (λ(loc, t) c. apply-cm c loc t (f loc
t)) for f
    by (fact comp-fun-commute-apply-cm)
    have (loc, x) ∈#z Δ
      by (meson case-member zcount-inI)
    then show ?thesis
      by (intro commute.fold-rec) simp-all
  qed

```

```

have zcount (c-pts (Finite-Set.fold ?f c (set-zmset  $\Delta - \{(loc, x)\}$ )) loc) x
  = zcount (c-pts c loc) x by (simp add: fold-nop)
then have zcount (c-pts (Finite-Set.fold ?f c (set-zmset  $\Delta$ )) loc) x
  = zcount (c-pts (?f (loc, x) c) loc) x
  using fold-nop fold-rec by (simp add: zcount-ps-apply-cm)
then show ?thesis
  by (simp add: zcount-ps-apply-cm cm-all'-def)
qed
then show ?thesis ..
qed

```

```

lemma implications-apply-cm[simp]: c-imp (apply-cm c loc t n) = c-imp c
  by (auto simp: apply-cm-def)

```

```

lemma implications-cm-all[simp]:
  c-imp (cm-all' c  $\Delta$ ) = c-imp c
  unfolding cm-all'-def Let-def
  apply (rule fold-invar[OF finite-set-zmset])
  apply auto
  done

```

```

lemma lift-cm-inv-cm-all':
  assumes ( $\bigwedge c0 c1 loc t n. P c0 \implies next-change-multiplicity' c0 c1 loc t n \implies$ 
   $P c1$ )
  and  $\forall (loc, t) \in \#_z \Delta. \exists t'. t' \in_A frontier (c-imp c0 loc) \wedge t' \leq t$ 
  and  $P c0$ 
  shows  $P (cm-all' c0 \Delta)$ 

```

```

proof -
  let ?cond-invar =  $\lambda c. \forall (loc, t) \in \#_z \Delta. \exists t'. t' \in_A frontier (c-imp c loc) \wedge t' \leq t$ 
  let ?invar =  $\lambda c. ?cond-invar c \wedge P c$ 
  show ?thesis
  unfolding cm-all'-def
  apply (rule conjunct2[OF fold-invar[OF finite-set-zmset, of ?invar]])
  using assms(2,3) apply simp
  subgoal
  apply safe
  apply auto []
  apply (rule assms(1))
  apply simp
  apply (rule apply-cm-is-cm)
  apply auto
  done
  apply simp
  done
qed

```

```

lemma lift-cm-inv-cm-all:
  assumes  $\bigwedge c0 c1 loc t n. P c0 \implies next-change-multiplicity' c0 c1 loc t n \implies P$ 
   $c1$ 

```

**and**  $\forall (loc, t) \in \#_z \Delta. \exists t'. t' \in_A \text{frontier } (c\text{-imp } c0 \text{ } loc) \wedge t' \leq t$   
**and**  $P \text{ } c0$   
**shows**  $P (cm\text{-all } c0 \text{ } \Delta)$   
**apply** (*subst cm-all-eq-cm-all'*)  
**using** *assms(2)* **apply** *simp*  
**using** *assms* **apply** (*rule lift-cm-inv-cm-all'*)  
**apply** *simp-all*  
**done**

**lemma** *obtain-min-worklist:*

**assumes** ( $a (loc': (- :: \text{finite})) :: ((t :: \text{order}) \text{ zmultiset}) \neq \{\#\}_z$ )  
**obtains**  $loc \ t$   
**where**  $t \in \#_z a \text{ } loc$   
**and**  $\forall t' \text{ } loc'. t' \in \#_z a \text{ } loc' \longrightarrow \neg t' < t$   
**using** *assms*  
**proof** *atomize-elim*  
**obtain**  $f$  **where**  $f: f = \text{minimal-antichain } (\bigcup loc'. \text{set-zmset } (a \text{ } loc'))$   
**by** *blast*  
**from** *assms* **obtain**  $x'$  **where**  $x': x' \in (\bigcup loc'. \text{set-zmset } (a \text{ } loc'))$   
**using** *pos-zcount-in-zmset* **by** *fastforce*  
**from** *assms* **have** *finite*  $(\bigcup loc'. \text{set-zmset } (a \text{ } loc'))$   
**using** *finite-UNIV* **by** *blast*  
**with**  $x'$  **have**  $f \neq \{\}$  **unfolding**  $f$   
**using** *minimal-antichain-nonempty* **by** *blast*  
**then obtain**  $t$  **where**  $t: t \in f (\forall s \in f. \neg s < t)$   
**using** *ex-min-if-finite*  $f$  *minimal-antichain-def* **by** *fastforce*  
**with**  $f$  **have** *thesis1*:  $\forall t' \text{ } loc'. t' \in \#_z a \text{ } loc' \longrightarrow \neg t' < t \exists loc. t \in \#_z a \text{ } loc$   
**by** (*simp add: minimal-antichain-def*)  
**then show**  $\exists t \text{ } loc. t \in \#_z a \text{ } loc \wedge (\forall t' \text{ } loc'. t' \in \#_z a \text{ } loc' \longrightarrow \neg t' < t)$  **by** *blast*  
**qed**

**lemma** *propagate-pointstamps-eq:*

**assumes**  $c\text{-work } c \text{ } loc \neq \{\#\}_z$   
**shows**  $c\text{-pts } c = c\text{-pts } (SOME \ c'. \exists loc \ t. \text{next-propagate}' \ c \ c' \ loc \ t)$   
**proof** –  
**from** *assms* **obtain**  $loc' \ t$  **where**  $loc': t \in \#_z c\text{-work } c \text{ } loc'$   
 $\forall t' \text{ } loc'. t' \in \#_z c\text{-work } c \text{ } loc' \longrightarrow \neg t' < t$   
**apply** (*rule obtain-min-worklist[of c-work c loc]*) **by** *blast*  
**let**  $?imps = \lambda locX. \text{if } locX = loc' \text{ then } c\text{-imp } c \text{ } locX + \{\#t' \in \#_z c\text{-work } c \text{ } locX. t' = t\# \}$   
 $\text{else } c\text{-imp } c \text{ } locX$   
**let**  $?wl = \lambda locX. \text{if } locX = loc' \text{ then } \{\#t' \in \#_z c\text{-work } c \text{ } locX. t' \neq t\# \}$   
 $\text{else } c\text{-work } c \text{ } locX$   
 $+ \text{after-summary}$   
 $(\text{frontier-changes } (?imps \ loc') (c\text{-imp } c \text{ } loc'))$   
 $(\text{summary } loc' \ locX)$   
**let**  $?c = c(c\text{-imp} := ?imps, c\text{-work} := ?wl)$   
**from**  $loc' \ t$  *assms* **have** *propagate*:  $\exists c'. \exists loc \ t. \text{next-propagate}' \ c \ c' \ loc \ t$

```

  by (intro exI[of - ?c] exI[of - loc'] exI[of - t])
    (auto simp add: next-propagate'-def intro!: Propagate.configuration.equality)
{ fix c' loc t
  assume next-propagate' c c' loc t
  then have c-pts c = c-pts c'
    by (simp add: next-propagate'-def)
}
with propagate show ?thesis
by (simp add: verit-sko-ex')
qed

```

**lemma** *propagate-all-imp-InvGlobPointstampsEq*:  
*Some c1 = propagate-all c0  $\implies$  c-pts c0 = c-pts c1*  
**unfolding** *propagate-all-def*  
**using** *while-option-rule*[**where**  $P=(\lambda c. c\text{-pts } c0 = c\text{-pts } c)$   
**and**  $b=(\lambda c. \exists loc. c\text{-work } c \text{ loc} \neq \{\#\}_z)$   
**and**  $c=(\lambda c. \text{SOME } c'. \exists loc t. \text{next-propagate}' c c' \text{ loc } t)$ ]  
*propagate-pointstamps-eq* **by** (*metis (no-types, lifting)*)

**lemma** *exists-next-propagate'*:  
**assumes**  $c\text{-work } c \text{ loc} \neq \{\#\}_z$   
**shows**  $\exists c' \text{ loc } t. \text{next-propagate}' c c' \text{ loc } t$   
**proof** –  
**from** *assms* **obtain**  $loc' t$  **where**  $loc't$ :  
 $t \in \{\#\}_z c\text{-work } c \text{ loc}'$   
 $\forall t' \text{ loc}'. t' \in \{\#\}_z c\text{-work } c \text{ loc}' \longrightarrow \neg t' < t$   
**by** (*rule obtain-min-worklist*)  
**let**  $?imps = \lambda locX. \text{if } locX = loc' \text{ then } c\text{-imp } c \text{ loc}X + \{\#t' \in \{\#\}_z c\text{-work } c \text{ loc}X.$   
 $t' = t\#\}$   
 $\text{else } c\text{-imp } c \text{ loc}X$   
**let**  $?wl = \lambda locX. \text{if } locX = loc' \text{ then } \{\#t' \in \{\#\}_z c\text{-work } c \text{ loc}X. t' \neq t\#\}$   
 $\text{else } c\text{-work } c \text{ loc}X$   
 $+ \text{after-summary}$   
 $(\text{frontier-changes } (?imps \text{ loc}') (c\text{-imp } c \text{ loc}'))$   
 $(\text{summary } loc' \text{ loc}X)$   
**let**  $?c = c(c\text{-imp} := ?imps, c\text{-work} := ?wl)$   
**from**  $loc't$  *assms* **show** *?thesis*  
**by** (*intro exI[of - ?c] exI[of - loc'] exI[of - t]*)  
*(auto simp: next-propagate'-def intro!: Propagate.configuration.equality)*  
**qed**

**lemma** *lift-propagate-inv-propagate-all*:  
**assumes**  $(\bigwedge c0 c1 \text{ loc } t. P c0 \implies \text{next-propagate}' c0 c1 \text{ loc } t \implies P c1)$   
**and**  $P c0$   
**and**  $\text{propagate-all } c0 = \text{Some } c1$   
**shows**  $P c1$   
**apply** (*rule while-option-rule*[*of*  $P$  -, *rotated*, *OF* *assms*(3)][*unfolded propagate-all-def*],  
 $OF$  *assms*(2)])  
**apply** *clarify*

```

subgoal for  $c$  loc
  apply (drule exists-next-propagate')
  apply (simp add: assms(1) verit-sko-ex')
  done
done

```

## 8.4 Exchange is a Subsystem of Tracker

Steps in the Tracker are valid steps in the Exchange protocol.

**lemma** *next-imp-exchange-next*:

```

 $Next' c0 c1 \implies cri.next' (exchange-config c0) (exchange-config c1)$ 
unfolding Next'-def cri.next'-def NextPerformOp'-def NextRecvUpd'-def NextSendUpd'-def
NextPropagate'-def NextRecvCap'-def
by auto

```

**lemma** *alw-next-imp-exchange-next*:  $alw Next s \implies alw cri.next (smap exchange-config s)$

**by** (*coinduction arbitrary: s rule: alw.coinduct*) (*auto dest: alwD next-imp-exchange-next*)

Any Tracker trace is a valid Exchange trace

**lemma** *spec-imp-exchange-spec*:  $FullSpec s \implies cri.spec (smap exchange-config s)$

```

unfolding cri.spec-def FullSpec-def
by (auto simp: InitConfig-def intro: alw-next-imp-exchange-next)

```

**lemma** *lift-exchange-invariant*:

```

assumes  $\bigwedge s. cri.spec s \implies alw (holds P) s$ 
shows  $FullSpec s \implies alw (\lambda s. P (exchange-config (shd s))) s$ 

```

**proof** –

```

assume FullSpec s
note spec-imp-exchange-spec[OF this]
note assms[OF this]
then show ?thesis
  by (auto simp: alw-holds-smap-conv-comp)

```

**qed**

Lifted Exchange invariants

**lemmas**

```

 $exch-alw-InvCapsNonneg = lift-exchange-invariant[OF cri.alw-InvCapsNonneg,$ 
unfolded atomize-imp, simplified, folded atomize-imp] and
 $exch-alw-InvRecordCount = lift-exchange-invariant[OF cri.alw-InvRecordCount,$ 
simplified atomize-imp, simplified, folded atomize-imp] and
 $exch-alw-InvRecordsNonneg = lift-exchange-invariant[OF cri.alw-InvRecordsNonneg,$ 
simplified atomize-imp, simplified, folded atomize-imp] and
 $exch-alw-InvGlobVacantImpRecordsVacant = lift-exchange-invariant[OF cri.alw-InvGlobVacantImpRecordsVacant,$ 
simplified atomize-imp, simplified, folded atomize-imp] and
 $exch-alw-InvGlobNonposImpRecordsNonpos = lift-exchange-invariant[OF cri.alw-InvGlobNonposImpRecordsNonpos,$ 
simplified atomize-imp, simplified, folded atomize-imp] and
 $exch-alw-InvJustifiedGII = lift-exchange-invariant[OF cri.alw-InvJustifiedGII,$ 
simplified atomize-imp, simplified, folded atomize-imp] and

```

$exch\text{-}alw\text{-}InvJustifiedII = lift\text{-}exchange\text{-}invariant[OF\ cri.\ alw\text{-}InvJustifiedII,$   
 $simplified\ atomize\text{-}imp, simplified, folded\ atomize\text{-}imp]$  **and**  
 $exch\text{-}alw\text{-}InvGlobNonposEqVacant = lift\text{-}exchange\text{-}invariant[OF\ cri.\ alw\text{-}InvGlobNonposEqVacant,$   
 $simplified\ atomize\text{-}imp, simplified, folded\ atomize\text{-}imp]$  **and**  
 $exch\text{-}alw\text{-}InvMsgInGlob = lift\text{-}exchange\text{-}invariant[OF\ cri.\ alw\text{-}InvMsgInGlob,$   
 $simplified\ atomize\text{-}imp, simplified, folded\ atomize\text{-}imp]$  **and**  
 $exch\text{-}alw\text{-}InvTempJustified = lift\text{-}exchange\text{-}invariant[OF\ cri.\ alw\text{-}InvTempJustified,$   
 $simplified\ atomize\text{-}imp, simplified, folded\ atomize\text{-}imp]$

## 8.5 Definitions

**definition**  $safe\text{-}combined :: ('p::finite, 't::order, 'loc)\ configuration \Rightarrow bool$  **where**  
 $safe\text{-}combined\ c \equiv \forall\ loc1\ loc2\ t\ s\ p.$   
 $zcount\ (cri.\ records\ (exchange\text{-}config\ c))\ (loc1, t) > 0 \wedge s \in_A\ path\text{-}summary$   
 $loc1\ loc2 \wedge init\ c\ p$   
 $\longrightarrow (\exists\ t'. t' \in_A\ frontier\ (c\text{-}imp\ (prop\text{-}config\ c\ p)\ loc2) \wedge t' \leq results\text{-}in\ t\ s)$

**definition**  $safe\text{-}combined2 :: ('p::finite, 't::order, 'loc)\ configuration \Rightarrow bool$  **where**  
 $safe\text{-}combined2\ c \equiv \forall\ loc1\ loc2\ t\ s\ p1\ p2.$   
 $zcount\ (c\text{-}caps\ (exchange\text{-}config\ c)\ p1)\ (loc1, t) > 0 \wedge s \in_A\ path\text{-}summary$   
 $loc1\ loc2 \wedge init\ c\ p2$   
 $\longrightarrow (\exists\ t'. t' \in_A\ frontier\ (c\text{-}imp\ (prop\text{-}config\ c\ p2)\ loc2) \wedge t' \leq results\text{-}in\ t\ s)$

**definition**  $InvGlobPointstampsEq :: ('p :: finite, 't :: order, 'loc)\ configuration \Rightarrow bool$  **where**  
 $InvGlobPointstampsEq\ c = ($   
 $(\forall\ p\ loc\ t. zcount\ (c\text{-}pts\ (prop\text{-}config\ c\ p)\ loc)\ t$   
 $= zcount\ (c\text{-}glob\ (exchange\text{-}config\ c)\ p)\ (loc, t)))$

**lemma**  $safe\text{-}combined\text{-}implies\text{-}safe\text{-}combined2:$   
**assumes**  $cri.\ InvCapsNonneg\ (exchange\text{-}config\ c)$   
**and**  $safe\text{-}combined\ c$   
**shows**  $safe\text{-}combined2\ c$   
**unfolding**  $safe\text{-}combined2\text{-}def$   
**apply**  $clarify$   
**subgoal for**  $loc1\ loc2\ t\ s\ p1\ p2$   
**apply**  $(rule\ assms(2)[unfolded\ safe\text{-}combined\text{-}def, rule\text{-}format, of\ loc1\ t\ s\ loc2\ p2])$   
**apply**  $(simp\ add:\ cri.\ records\text{-}def)$   
**apply**  $(rule\ add\text{-}pos\text{-}nonneg)$   
**apply**  $(subst\ zcount\text{-}sum)$   
**apply**  $(rule\ sum\text{-}pos[where\ y = p1])$   
**using**  $assms(1)$   
**apply**  $(simp\text{-}all\ add:\ cri.\ InvCapsNonneg\text{-}def)$   
**done**  
**done**

## 8.6 Propagate is a Subsystem of Tracker

### 8.6.1 CM Conditions

**definition** *InvMsgCMConditions* **where**

*InvMsgCMConditions*  $c = (\forall p q.$   
 $\text{init } c \ q \longrightarrow c\text{-msg } (\text{exchange-config } c) \ p \ q \neq \square \longrightarrow$   
 $(\forall (loc, t) \in \#_z (\text{hd } (c\text{-msg } (\text{exchange-config } c) \ p \ q))). \exists t'. t' \in_A \text{frontier } (c\text{-imp}$   
 $(\text{prop-config } c \ q) \ loc) \wedge t' \leq t)$

Pointstamps in incoming messages all satisfy the CM premise, which is required during NextRecvUpd' steps.

**lemma** *msg-is-cm-safe*:

**fixes**  $c :: ('p::\text{finite}, 't::\text{order}, 'loc) \text{ configuration}$   
**assumes** *safe*  $(\text{prop-config } c \ q)$   
**and** *InvGlobPointstampsEq*  $c$   
**and** *cri.InvMsgInGlob*  $(\text{exchange-config } c)$   
**and**  $c\text{-msg } (\text{exchange-config } c) \ p \ q \neq \square$   
**shows**  $\forall (loc, t) \in \#_z (\text{hd } (c\text{-msg } (\text{exchange-config } c) \ p \ q)). \exists t'. t' \in_A \text{frontier}$   
 $(c\text{-imp } (\text{prop-config } c \ q) \ loc) \wedge t' \leq t$   
**using** *assms(3)[unfolded cri.InvMsgInGlob-def, rule-format, OF assms(4)] assms(1)[unfolded*  
*safe-def, rule-format]*  
**apply**  $(\text{clarsimp simp: cri-less-eq-def assms(2)[unfolded InvGlobPointstampsEq-def,$   
*rule-format, symmetric])*  
**using** *order-trans* **apply** *blast*  
**done**

### 8.6.2 Propagate Safety and InvGlobPointstampsEq

To be able to use the *msg-is-cm-safe* lemma at all times and show that Propagate is a subsystem we need to prove that the specification implies Propagate's safe and the InvGlobPointstampsEq. Both of these depend on the CM conditions being satisfied during the NextRecvUpd' step and the safety proof additionally depends on other Propagate invariants, which means that we need to prove all of these jointly.

**abbreviation** *prop-invs* **where**

*prop-invs*  $c \equiv \text{inv-implications-nonneg } c \wedge \text{inv-imps-work-sum } c$

**abbreviation** *prop-safe* **where**

*prop-safe*  $c \equiv \text{impl-safe } c \wedge \text{safe } c$

**definition** *inv-init-imp-prop-safe* **where**

*inv-init-imp-prop-safe*  $c = (\forall p. \text{init } c \ p \longrightarrow \text{prop-safe } (\text{prop-config } c \ p))$

**lemma** *NextRecvUpd'-preserves-prop-safe*:

**assumes** *prop-safe*  $(\text{prop-config } c0 \ q)$   
**and** *InvGlobPointstampsEq*  $c0$   
**and** *cri.InvMsgInGlob*  $(\text{exchange-config } c0)$   
**and** *NextRecvUpd'*  $c0 \ c1 \ p \ q$

**shows**  $prop\text{-}safe\ (prop\text{-}config\ c1\ q)$   
**proof** –  
**have**  $safe: safe\ (prop\text{-}config\ c0\ q)$   
**using**  $assms(1)$  **by**  $blast$   
**note**  $recvupd\text{-}change = cri.next\text{-}recvupdD(1)[OF\ NextRecvUpdD(2)[OF\ assms(4)]]$   
**note**  $cm\text{-}conds = msg\text{-}is\text{-}cm\text{-}safe[OF\ safe\ assms(2,3)\ recvupd\text{-}change]$   
**have**  $safes:$   
 $prop\text{-}safe\ c0 \implies next\text{-}change\text{-}multiplicity'\ c0\ c1\ loc\ t\ n \implies prop\text{-}safe\ c1$  **for**  
 $c0\ c1\ loc\ t\ n$   
**using**  
 $cm\text{-}preserves\text{-}safe$   
 $cm\text{-}preserves\text{-}impl\text{-}safe$   
**by**  $auto$   
**show**  $prop\text{-}safe\ (prop\text{-}config\ c1\ q)$   
**using**  
 $lift\text{-}cm\text{-}inv\text{-}cm\text{-}all[rotated,\ OF\ cm\text{-}conds,\ of\ prop\text{-}safe,\ OF\ assms(1)]$   
 $safes$   
 $NextRecvUpdD(4)[OF\ assms(4)]$   
**by**  $metis$   
**qed**

**lemma**  $NextRecvUpd'\text{-}preserves\text{-}InvGlobPointstampsEq:$   
**assumes**  $impl\text{-}safe\ (prop\text{-}config\ c0\ q) \wedge safe\ (prop\text{-}config\ c0\ q)$   
**and**  $InvGlobPointstampsEq\ c0$   
**and**  $cri.InvMsgInGlob\ (exchange\text{-}config\ c0)$   
**and**  $NextRecvUpd'\ c0\ c1\ p\ q$   
**shows**  $InvGlobPointstampsEq\ c1$   
**proof** –  
**have**  $safe: safe\ (prop\text{-}config\ c0\ q)$   
**using**  $assms(1)$  **by**  $blast$   
**note**  $recvupd\text{-}change = cri.next\text{-}recvupdD(1)[OF\ NextRecvUpdD(2)[OF\ assms(4)]]$   
**note**  $cm\text{-}conds = msg\text{-}is\text{-}cm\text{-}safe[OF\ safe\ assms(2,3)\ recvupd\text{-}change]$   
**show**  $InvGlobPointstampsEq\ c1$   
**using**  
 $assms(2,4)$   
 $cm\text{-}conds$   
**unfolding**  $NextRecvUpd'\text{-}def\ cri.next\text{-}recvupd'\text{-}def\ Let\text{-}def\ InvGlobPointstampsEq\text{-}def$   
**by**  $(clarsimp\ simp: zcount\text{-}pointstamps\text{-}cm\text{-}all'\ cm\text{-}all\text{-}eq\text{-}cm\text{-}all')+$   
**qed**

— Whenever some worker  $p$  propagates it ends up in a Propagate-safe state

**lemma**  $NextPropagate'\text{-}causes\text{-}safe:$   
**assumes**  $NextPropagate'\ c0\ c1\ p$   
**and**  $inv\text{-}imps\text{-}work\text{-}sum\ (prop\text{-}config\ c1\ p)$   
**and**  $inv\text{-}implications\text{-}nonneg\ (prop\text{-}config\ c1\ p)$   
**shows**  $safe\ (prop\text{-}config\ c1\ p)\ impl\text{-}safe\ (prop\text{-}config\ c1\ p)$   
**proof** –  
**from**  $assms(1)$  **have**  $Some\ (prop\text{-}config\ c1\ p) = propagate\text{-}all\ (prop\text{-}config\ c0\ p)$   
**by**  $(simp\ add: NextPropagate'\text{-}def)$

**then have**  $wl: c\text{-work } (prop\text{-config } c1\ p)\ loc = \{\#\}_z$  **for**  $loc$   
**unfolding**  $propagate\text{-all}\text{-def}$   
**by**  $(subst\ (asm)\ eq\text{-commute})\ (auto\ dest: while\text{-option}\text{-stop})$   
**show**  $safe\ (prop\text{-config } c1\ p)\ impl\text{-safe}\ (prop\text{-config } c1\ p)$   
**by**  $(rule\ safe[OF\ assms(2,3)\ wl])\ (rule\ impl\text{-safe}[OF\ assms(2,3)\ wl])$   
**qed**

— NextPropagate' preserves Propagate-safe at all workers

**lemma**  $NextPropagate'\text{-preserves}\text{-safe}$ :  
**assumes**  $NextPropagate'\ c0\ c1\ q$   
**and**  $inv\text{-imps}\text{-work}\text{-sum}\ (prop\text{-config } c1\ p)$   
**and**  $inv\text{-implications}\text{-nonneg}\ (prop\text{-config } c1\ p)$   
**and**  $safe\ (prop\text{-config } c0\ p)$   
**shows**  $safe\ (prop\text{-config } c1\ p)$   
**apply**  $(cases\ p=q)$   
**subgoal**  
**using**  $assms(1-3)$  **by**  $(auto\ intro: NextPropagate'\text{-causes}\text{-safe})$   
**subgoal**  
**using**  $assms(1,4)$  **by**  $(auto\ dest: spec[of - p]\ simp: NextPropagate'\text{-def})$   
**done**

**lemma**  $NextPropagate'\text{-preserves}\text{-impl}\text{-safe}$ :  
**assumes**  $NextPropagate'\ c0\ c1\ q$   
**and**  $inv\text{-imps}\text{-work}\text{-sum}\ (prop\text{-config } c1\ p)$   
**and**  $inv\text{-implications}\text{-nonneg}\ (prop\text{-config } c1\ p)$   
**and**  $impl\text{-safe}\ (prop\text{-config } c0\ p)$   
**shows**  $impl\text{-safe}\ (prop\text{-config } c1\ p)$   
**apply**  $(cases\ p=q)$   
**subgoal**  
**using**  $assms(1-3)$  **by**  $(auto\ intro: NextPropagate'\text{-causes}\text{-safe})$   
**subgoal**  
**using**  $assms(1,4)$  **by**  $(auto\ dest: spec[of - p]\ simp: NextPropagate'\text{-def})$   
**done**

**lemma**  $NextRecvUpd'\text{-preserves}\text{-inv}\text{-init}\text{-imp}\text{-prop}\text{-safe}$ :  
**assumes**  $cri.InvMsgInGlob\ (exchange\text{-config } c0)$   
**and**  $inv\text{-init}\text{-imp}\text{-prop}\text{-safe } c0$   
**and**  $InvGlobPointstampsEq\ c0$   
**and**  $NextRecvUpd'\ c0\ c1\ p\ q$   
**shows**  $inv\text{-init}\text{-imp}\text{-prop}\text{-safe } c1$   
**using**  $assms(2)$  **unfolding**  $inv\text{-init}\text{-imp}\text{-prop}\text{-safe}\text{-def}$   
**apply**  $clarify$   
**subgoal for**  $p$   
**apply**  $(cases\ p=q)$   
**subgoal**  
**apply**  $(drule\ spec[of -p])$   
**apply**  $(simp\ add: NextRecvUpdD(1)[OF\ assms(4)])$   
**apply**  $(drule\ NextRecvUpd'\text{-preserves}\text{-prop}\text{-safe}[OF - assms(3,1,4)])$   
**apply**  $simp$

```

    done
  subgoal
    using NextRecvUpdD(3,4)[OF assms(4)] by fastforce
  done
done

lemma NextRecvUpd'-preserves-prop-invs:
  assumes cri.InvMsgInGlob (exchange-config c0)
    and inv-init-imp-prop-safe c0
    and  $\forall p. \text{prop-invs (prop-config c0 p)}$ 
    and InvGlobPointstampsEq c0
    and NextRecvUpd' c0 c1 p q
  shows  $\forall p. \text{prop-invs (prop-config c1 p)}$ 
proof -
  have safe: safe (prop-config c0 q)
    using NextRecvUpdD(1) assms(2,5) inv-init-imp-prop-safe-def by blast
  note recvupd-change = cri.next-recvupdD(1)[OF NextRecvUpdD(2)[OF assms(5)]]
  note cm-conds = msg-is-cm-safe[OF safe assms(4,1) recvupd-change]
  have invs:
    prop-invs c0  $\implies$  next-change-multiplicity' c0 c1 loc t n  $\implies$  prop-invs c1 for
    c0 c1 loc t n
  using
    cm-preserves-inv-imps-work-sum
    cm-preserves-inv-implications-nonneg
  by auto
  show  $\forall q. \text{prop-invs (prop-config c1 q)}$ 
  apply rule
  subgoal for q'
    apply (cases q'=q)
  subgoal
    using
      lift-cm-inv-cm-all[rotated, OF cm-conds, of prop-invs, OF assms(3)][rule-format]]
      invs
      NextRecvUpdD(4)[OF assms(5)]
    by metis
  subgoal
    using NextRecvUpdD(4) assms(3) assms(5) by fastforce
  done
done
qed

lemma NextPropagate'-preserves-prop-invs:
  assumes prop-invs (prop-config c0 q)
    and NextPropagate' c0 c1 p
  shows prop-invs (prop-config c1 q)
  apply (cases p=q)
  subgoal
    using
      assms(1)

```

```

lift-propagate-inv-propagate-all[
  of prop-invs,
  rotated 2,
  OF NextPropagateD(3)[OF assms(2), rule-format, of p, simplified, symmet-
ric]]
by (simp add: iiws-imp-iipwn p-preserves-inv-implications-nonneg p-preserves-inv-imps-work-sum)
subgoal
  by (metis NextPropagateD(3) assms(1) assms(2) option.simps(1))
done

```

```

lemma NextPropagate'-preserves-inv-init-imp-prop-safe:
  assumes prop-invs (prop-config c0 p)
  and inv-init-imp-prop-safe c0
  and NextPropagate' c0 c1 p
  shows inv-init-imp-prop-safe c1
  using assms(2) unfolding inv-init-imp-prop-safe-def
  apply clarsimp
  subgoal for p'
    apply (cases p'=p)
    subgoal
      using NextPropagate'-preserves-prop-invs[OF assms(1,3)]
      using NextPropagate'-causes-safe(1,2)[OF assms(3)] by blast
    subgoal
      using NextPropagateD(2,3)[OF assms(3)]
      by (auto dest: spec[of - p'])
  done
done

```

```

lemma Next'-preserves-invs:
  assumes cri.InvMsgInGlob (exchange-config c0)
  and inv-init-imp-prop-safe c0
  and InvGlobPointstampsEq c0
  and Next' c0 c1
  and  $\forall p. \text{prop-invs (prop-config c0 p)}$ 
  shows
    inv-init-imp-prop-safe c1
     $\forall p. \text{prop-invs (prop-config c1 p)}$ 
    InvGlobPointstampsEq c1
  subgoal
    using assms(4) unfolding Next'-def
    using assms(2)
    apply (elim disjE)
    subgoal
      unfolding inv-init-imp-prop-safe-def
      using NextPerformOpD(2,3) by fastforce
    subgoal
      unfolding inv-init-imp-prop-safe-def
      using NextSendUpdD(2,3) by fastforce
    subgoal

```

```

    using NextRecvUpd'-preserves-inv-init-imp-prop-safe[OF assms(1,2,3)]
    by blast
  subgoal
    using NextPropagate'-preserves-inv-init-imp-prop-safe assms(5) by blast
  subgoal
    unfolding inv-init-imp-prop-safe-def
    using NextRecvCapD(2,3) by fastforce
  subgoal by simp
  done
subgoal
  using assms(4) unfolding Next'-def
  using assms(5)
  apply (elim disjE)
  subgoal
    using NextPerformOpD(2) by fastforce
  subgoal
    using NextSendUpdD(2) by fastforce
  subgoal
    using assms(1,2,3) NextRecvUpd'-preserves-prop-invs by blast
  subgoal
    using NextPropagate'-preserves-prop-invs by blast
  subgoal
    unfolding inv-init-imp-prop-safe-def
    using NextRecvCapD(2,3) by fastforce
  subgoal by simp
  done
subgoal
  using assms(4) unfolding Next'-def
  using assms(3)
  apply (elim disjE)
  subgoal
    by (metis InvGlobPointstampsEq-def NextPerformOpD(1,2) cri.next-performopD(7))
  subgoal
    by (metis InvGlobPointstampsEq-def NextSendUpdD(1,2) cri.next-sendupdD(5))
  subgoal
    using NextRecvUpdD(1) NextRecvUpd'-preserves-InvGlobPointstampsEq assms(1,2)
    inv-init-imp-prop-safe-def by blast
  subgoal
    unfolding NextPropagate'-def InvGlobPointstampsEq-def
    using propagate-all-imp-InvGlobPointstampsEq
    by (metis option.inject)
  subgoal
    by (metis InvGlobPointstampsEq-def NextRecvCapD(1) NextRecvCapD(2)
    cri.next-recvcapD(4))
  subgoal by simp
  done
done

```

**lemma** *init-imp-InvGlobPointstampsEq*:  $\text{InitConfig } c \implies \text{InvGlobPointstampsEq}$

*c*  
**by** (*simp add: InitConfig-def cri.init-config-def InvGlobPointstampsEq-def*)

**lemma** *init-imp-inv-init-imp-prop-safe*: *InitConfig c*  $\implies$  *inv-init-imp-prop-safe c*  
**by** (*simp add: inv-init-imp-prop-safe-def InitConfig-def*)

**lemma** *init-imp-prop-invs*: *InitConfig c*  $\implies$   $\forall p.$  *prop-invs (prop-config c p)*  
**by** (*simp add: InitConfig-def init-imp-inv-implications-nonneg init-imp-inv-imps-work-sum*)

**abbreviation** *all-invs where*  
*all-invs c*  $\equiv$  *InvGlobPointstampsEq c*  $\wedge$  *inv-init-imp-prop-safe c*  $\wedge$  ( $\forall p.$  *prop-invs (prop-config c p)*)

**lemma** *alw-Next'-alw-invs*:  
**assumes** *holds all-invs s*  
**and** *alw (holds ( $\lambda c.$  *cri.InvMsgInGlob (exchange-config c)*)) s*  
**and** *alw Next s*  
**shows** *alw (holds all-invs) s*  
**using** *assms*  
**apply** (*coinduction arbitrary: s*)  
**apply** *clarsimp*  
**apply** *safe*  
**apply** (*metis (mono-tags, lifting) alw-holds2 Next'-preserves-invs(3) alwD*)  
**apply** (*metis (mono-tags, lifting) alw-holds2 Next'-preserves-invs(1) alwD*)  
**apply** (*metis (mono-tags, lifting) alw-holds2 Next'-preserves-invs(2) alwD*)  
**apply** (*metis (mono-tags, lifting) alw-holds2 Next'-preserves-invs(2) alwD*)  
**apply** *auto*  
**done**

**lemma** *alw-invs*: *FullSpec s*  $\implies$  *alw (holds all-invs) s*  
**apply** (*frule exch-alw-InvMsgInGlob*)  
**unfolding** *FullSpec-def*  
**apply** *clarsimp*  
**apply** (*frule init-imp-InvGlobPointstampsEq*)  
**apply** (*frule init-imp-inv-init-imp-prop-safe*)  
**apply** (*drule init-imp-prop-invs*)  
**by** (*simp add: alw-Next'-alw-invs alw-mono*)

**lemma** *alw-InvGlobPointstampsEq*: *FullSpec s*  $\implies$  *alw (holds InvGlobPointstampsEq) s*  
**using** *alw-invs alw-mono holds-mono* **by** *blast*

**lemma** *alw-inv-init-imp-prop-safe*: *FullSpec s*  $\implies$  *alw (holds inv-init-imp-prop-safe) s*  
**using** *alw-invs alw-mono holds-mono* **by** *blast*

**lemma** *alw-holds-conv-shd*: *alw (holds  $\varphi$ ) s* = *alw ( $\lambda s.$   $\varphi$  (*shd s*)) s*  
**by** (*simp add: alw-iff-sdrop*)

**lemma** *alw-prop-invs*:  $FullSpec\ s \implies alw\ (holds\ (\lambda c.\ \forall p.\ prop\text{-}invs\ (prop\text{-}config\ c\ p)))\ s$   
**by** (*auto*  
*intro*: *alw-mono*[*of holds all-invs s holds*  $(\lambda c.\ \forall p.\ prop\text{-}invs\ (prop\text{-}config\ c\ p))$ ]  
*dest*: *alw-invs*)

**lemma** *nrec-pts-delayed*:  
**assumes** *cri.InvGlobNonposImpRecordsNonpos* (*exchange-config c*)  
**and** *zcount* (*cri.records* (*exchange-config c*))  $x > 0$   
**shows**  $\exists x'. x' \leq_p x \wedge zcount\ (c\text{-glob}\ (exchange\text{-}config\ c)\ p)\ x' > 0$   
**proof** –  
**from** *assms* **have**  $r: \forall p.\ \neg\ cri.\ nonpos\text{-}upto\ (c\text{-glob}\ (exchange\text{-}config\ c)\ p)\ x$   
**unfolding** *cri.InvGlobNonposImpRecordsNonpos-def cri.nonpos-upto-def*  
**by** (*metis linorder-not-less cri.order.order-iff-strict*)  
**show** *?thesis*  
**using** *r*[*rule-format, of p*]  
**by** (*auto simp: cri.nonpos-upto-def not-le*)  
**qed**

**lemma** *help-lemma*:  
**assumes**  $0 < zcount\ (c\text{-pts}\ (prop\text{-}config\ c\ p)\ loc0)\ t0$   
**and**  $(loc0, t0) \leq_p (loc1, t1)$   
**and**  $s2 \in_A\ path\text{-}summary\ loc1\ loc2$   
**and** *safe* (*prop-config c p*)  
**shows**  $\exists t2.\ (t2 \leq results\text{-}in\ t1\ s2$   
 $\wedge t2 \in_A\ frontier\ (c\text{-imp}\ (prop\text{-}config\ c\ p)\ loc2))$   
**proof** –  
**from** *assms*(2) **obtain** *s1* **where**  $s1: s1 \in_A\ path\text{-}summary\ loc0\ loc1\ results\text{-}in\ t0\ s1 \leq t1$   
**by** (*auto simp add: cri-less-eq-def*)  
**from** *s1*(1) *assms*(3) **obtain** *s-full* **where**  $s\text{-full}: s\text{-full} \in_A\ path\text{-}summary\ loc0\ loc2\ s\text{-full} \leq followed\text{-}by\ s1\ s2$   
**using** *flow.path-weight-elem-trans* **by** *blast*  
**from** *s-full*(1) *assms*(1,4) **obtain** *t2* **where**  $t2: t2 \in_A\ frontier\ (c\text{-imp}\ (prop\text{-}config\ c\ p)\ loc2)\ t2 \leq results\text{-}in\ t0\ s\text{-full}$   
**unfolding** *safe-def* **by** *blast*  
**from** *t2*(2) **and** *s-full*(2) **have**  $t2 \leq results\text{-}in\ (results\text{-}in\ t0\ s1)\ s2$   
**by** (*metis followed-by-summary order-trans results-in-mono*(2))  
**with** *s1*(2) **have**  $t2 \leq results\text{-}in\ t1\ s2$  **by** (*meson order.trans results-in-mono*(1))  
**with** *t2*(1) **show** *?thesis* **by** *auto*  
**qed**

— Lift an invariant’s preservation proof over *next-propagate'* to *NextPropagate'* transitions

**lemma** *lift-prop-inv-NextPropagate'*:  
**assumes**  $(\bigwedge c0\ c1\ loc\ t.\ P\ c0 \implies next\text{-}propagate'\ c0\ c1\ loc\ t \implies P\ c1)$   
**shows**  $P\ (prop\text{-}config\ c0\ p') \implies NextPropagate'\ c0\ c1\ p \implies P\ (prop\text{-}config\ c1\ p')$   
**proof** –

```

assume  $pc0$ :  $P$  ( $prop\text{-}config\ c0\ p'$ )
assume  $np$ :  $NextPropagate'\ c0\ c1\ p$ 
have  $n\text{-}p$ : ( $\bigwedge c0\ c1. P\ c0 \implies next\text{-}propagate\ c0\ c1 \implies P\ c1$ )
  using  $assms$  by  $auto$ 
let  $?f = \lambda c. SOME\ c'. next\text{-}propagate\ c\ c'$ 
let  $?b = \lambda c. \exists loc. c\text{-}work\ c\ loc \neq \{\#\}_z$ 
from  $np$  have  $pc1$ :  $Some\ (prop\text{-}config\ c1\ p) = propagate\text{-}all\ (prop\text{-}config\ c0\ p)$ 
  by ( $simp\ add: NextPropagate'\text{-}def$ )
show  $?thesis$ 
  apply ( $cases\ p'=p$ )
  subgoal
    apply ( $rule\ while\text{-}option\text{-}rule[of\ P\ ?b\ ?f\ prop\text{-}config\ c0\ p]$ )
    apply ( $rule\ n\text{-}p$ )
    apply  $assumption$ 
    apply ( $rule\ iffD1[OF\ verit\text{-}sko\text{-}ex]$ )
    apply ( $elim\ exE$ )
    apply ( $rule\ exists\text{-}next\text{-}propagate'$ )
    apply  $assumption$ 
    using  $pc1$  apply ( $simp\ add: propagate\text{-}all\text{-}def$ )
    using  $pc0$  apply  $simp$ 
    done
  subgoal
    using  $np\ pc0$  by ( $auto\ simp: NextPropagate'\text{-}def\ dest!: spec[of\ \text{-}\ p']$ )
  done
qed

```

### 8.6.3 Propagate is a Subsystem

```

lemma  $NextRecvUpd'\text{-}next'$ :
assumes  $safe\ (prop\text{-}config\ c0\ q)$ 
  and  $InvGlobPointstampsEq\ c0$ 
  and  $cri.InvMsgInGlob\ (exchange\text{-}config\ c0)$ 
  and  $NextRecvUpd'\ c0\ c1\ p\ q$ 
shows  $next'^{++}\ (prop\text{-}config\ c0\ q')\ (prop\text{-}config\ c1\ q')$ 
apply ( $subst\ NextRecvUpdD(4)[OF\ assms(4),\ rule\text{-}format]$ )
apply  $simp$ 
apply  $safe$ 
subgoal
  apply ( $subst\ cm\text{-}all\text{-}eq\text{-}cm\text{-}all'$ )
  apply  $clarsimp$ 
  apply ( $drule\ assms(3)[unfolded\ cri.InvMsgInGlob\text{-}def,\ rule\text{-}format,\ OF\ cri.next\text{-}recvupdD(1)[OF\ NextRecvUpdD(2)[OF\ assms(4)]]]$ )
  apply  $clarsimp$ 
  subgoal for  $loc\ t\ loc'\ t'$ 
  apply ( $subst\ (asm)\ assms(2)[unfolded\ InvGlobPointstampsEq\text{-}def,\ rule\text{-}format,\ symmetric]$ )
  apply ( $clarsimp\ simp: cri\text{-}less\text{-}eq\text{-}def$ )
  subgoal for  $s$ 
    using  $assms(1)[unfolded\ safe\text{-}def,\ rule\text{-}format,\ of\ loc'\ t'\ s\ loc]$ 

```

```

apply –
apply (drule meta-mp)
apply simp
apply clarsimp
subgoal for t''
apply (clarsimp intro!: exI[of - t''])
using order-trans apply blast
done
done
done
apply (rule lift-cm-inv-cm-all')
apply (rule tranclp.intros(2))
apply (auto simp: next'-def) [2]
apply clarsimp
apply (drule assms(3)[unfolded cri.InvMsgInGlob-def, rule-format, OF cri.next-recvupdD(1)[OF
NextRecvUpdD(2)[OF assms(4)]]])
apply clarsimp
subgoal for loc t loc' t'
apply (subst (asm) assms(2)[unfolded InvGlobPointstampsEq-def, rule-format,
symmetric])
apply (clarsimp simp: cri-less-eq-def)
subgoal for s
using assms(1)[unfolded safe-def, rule-format, of loc' t' s loc]
apply –
apply (drule meta-mp)
apply simp
apply clarsimp
subgoal for t''
apply (clarsimp intro!: exI[of - t''])
using order-trans apply blast
done
done
done
apply (auto simp: next'-def)
done
apply (auto simp: next'-def)
done

```

```

lemma NextPropagate'-next':
assumes NextPropagate' c0 c1 p
shows next'^{++} (prop-config c0 q) (prop-config c1 q)
apply (cases p=q)
subgoal
apply (rule lift-propagate-inv-propagate-all[of - prop-config c0 p])
apply (rule tranclp.intros(2))
apply (auto simp: next'-def NextPropagateD(3)[OF assms, rule-format])
done
subgoal
by (metis NextPropagateD(3) assms next'-def option.simps(1) tranclp.intros(1))

```

done

**lemma** *next-imp-propagate-next*:

**assumes** *inv-init-imp-prop-safe* *c0*

**and** *InvGlobPointstampsEq* *c0*

**and** *cri.InvMsgInGlob* (*exchange-config* *c0*)

**shows**  $Next' c0 c1 \implies next'^{++}$  (*prop-config* *c0* *p*) (*prop-config* *c1* *p*)

**unfolding** *Next'-def* *NextPerformOp'-def* *NextSendUpd'-def* *NextRecvCap'-def*

**apply** *safe*

**subgoal by** (*auto simp: next'-def*)

**subgoal by** (*auto simp: next'-def*)

**subgoal for** *p' q*

**using** *assms(1)[unfolded inv-init-imp-prop-safe-def, rule-format, of q]*

**apply**  $-$

**apply** (*drule meta-mp*)

**apply** (*rule NextRecvUpdD(1)*)

**apply** *simp*

**apply** (*cases q=p*)

**apply** (*auto dest!: NextRecvUpd'-next'[rotated, OF assms(2-)]*)  $\square$

**apply** (*auto simp add: NextRecvUpd'-def next'-def*)

**done**

**subgoal by** (*rule NextPropagate'-next'*)

**subgoal by** (*auto simp: next'-def*)

**subgoal by** (*auto simp: next'-def*)

**done**

**lemma** *alw-next-imp-propagate-next*:

**assumes** *alw* (*holds inv-init-imp-prop-safe*) *s*

**and** *alw* (*holds InvGlobPointstampsEq*) *s*

**and** *alw* (*holds cri.InvMsgInGlob*) (*smap exchange-config* *s*)

**and** *alw* *Next* *s*

**shows** *alw* (*relates* ( $next'^{++}$ )) (*smap* ( $\lambda s. prop-config$  *s* *p*) *s*)

**using** *assms* **by** (*coinduction arbitrary: s rule: alw.coinduct*) (*auto intro!: next-imp-propagate-next simp: relates-def alw-holds-smap-conv-comp*)

Any Tracker trace is a valid Propagate trace (using the transitive closure of next, since tracker may take multiple propagate steps at once).

**lemma** *spec-imp-propagate-spec*: *FullSpec* *s*  $\implies$  (*holds init-config* *aand* *alw* (*relates* ( $next'^{++}$ ))) (*smap* ( $\lambda c. prop-config$  *c* *p*) *s*)

**apply** (*frule alw-inv-init-imp-prop-safe*)

**apply** (*frule alw-InvGlobPointstampsEq*)

**apply** (*frule spec-imp-exchange-spec*)

**apply** (*drule cri.alw-InvMsgInGlob*)

**apply** (*auto intro!: alw-next-imp-propagate-next simp: FullSpec-def InitConfig-def*)

**done**

## 8.7 Safety Proofs

**lemma** *safe-satisfied*:

```

assumes cri.InvGlobNonposImpRecordsNonpos (exchange-config c)
  and inv-init-imp-prop-safe c
  and InvGlobPointstampsEq c
shows safe-combined c
proof –
{
  fix loc1 loc2 t s p
  assume as: 0 < zcount (cri.records (exchange-config c)) (loc1, t)
  s ∈A path-summary loc1 loc2 init c p
  obtain loc0 t0 where delayed:
  (loc0, t0)  $\leq_p$  (loc1, t) 0 < zcount (c-glob (exchange-config c) p) (loc0, t0)
  using nrec-pts-delayed[OF assms(1) as(1)]
  by fast
  with as(2,3) assms(2) have
   $\exists t2. t2 \leq \text{results-in } t \text{ s} \wedge t2 \in_A \text{frontier } (c\text{-imp } (prop\text{-config } c \text{ p}) \text{ loc2})$ 
  using help-lemma delayed
  by (metis InvGlobPointstampsEq-def assms(3) inv-init-imp-prop-safe-def)
}
then show ?thesis
  unfolding safe-combined-def by blast
qed

```

```

lemma alw-safe-combined: FullSpec s  $\implies$  alw (holds safe-combined) s
apply (frule alw-inv-init-imp-prop-safe)
apply (frule exch-alw-InvGlobNonposImpRecordsNonpos)
apply (drule alw-InvGlobPointstampsEq)
apply (coinduction arbitrary: s rule: alw.coinduct)
apply clarsimp
apply (rule conjI)
  apply (metis alwD alw-holds2 safe-satisfied)
apply (rule disjI1)
apply blast
done

```

```

lemma alw-safe-combined2: FullSpec s  $\implies$  alw (holds safe-combined2) s
apply (frule exch-alw-InvCapsNonneg)
apply (drule alw-safe-combined)
apply (simp add: alw-iff-sdrop safe-combined-implies-safe-combined2)
done

```

```

lemma alw-implication-frontier-eq-implied-frontier:
  FullSpec s  $\implies$ 
  alw (holds ( $\lambda c. \text{worklists-vacant-to } (prop\text{-config } c \text{ p}) \text{ b} \longrightarrow$ 
    b ∈A frontier (c-imp (prop-config c p) loc)  $\longleftrightarrow$  b ∈A implied-frontier (c-pts
    (prop-config c p) loc)) s
  by (drule alw-prop-invs)
  (auto simp: implication-frontier-iff-implied-frontier-vacant all-imp-alw elim:
alw-mp)

```

end

## References

- [1] Github: Timely dataflow.
- [2] M. Abadi, F. McSherry, D. G. Murray, and T. L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In D. Beyer and M. Boreale, editors, *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 5–19. Springer, 2013.
- [3] M. Brun, S. Decova, A. Lattuada, and D. Traytel. Verified progress tracking for timely dataflow. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. To appear.
- [4] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In M. Kaminsky and M. Dahlin, editors, *SOSP 2013*, pages 439–455. ACM, 2013.