TreeSync

Nicole

August 4, 2025

Contents

theory *FormalLanguages*
imports *Main HOL−Library.LaTeXsugar HOL−Library.OptionalSugar*
begin

1 Formal Languages

type-synonym *'a word* = *'a list*
type-synonym *'a language* = *'a word set*

1.1 Words

abbreviation *emptyWord* :: $'a$ *word* (ε) **where**
  $\varepsilon \equiv []$

abbreviation *concat* :: $'a$ *word* $\Rightarrow$ $'a$ *word* $\Rightarrow$ $'a$ *word* (**infixl** $\cdot$ *60*) **where**
  $v \cdot w \equiv v$ @ w

abbreviation *length-of-word* :: $'a$ *word* $\Rightarrow$ *nat* ($|\text{-}|$ [*90*] *60*) **where**
  $|w| \equiv$ *length* w

1.2 Alphabets

locale *Alphabet* =
  fixes *Letters* :: $'a$ *set* (Σ)
  assumes *not-empty*: $\Sigma \neq \{\}$
    and *finite-letters*: *finite* Σ
begin

inductive-set *WordsOverAlphabet* :: $'a$ *word set* (Σ^* *100*) **where**
EmptyWord: $\varepsilon \in \Sigma^*$ |
Composed: $[\![a \in \Sigma;\ w \in \Sigma^*]\!] \Longrightarrow (a\#w) \in \Sigma^*$

lemma *word-over-alphabet-rev*:
  fixes a :: $'a$
   and w :: $'a$ *word*
  assumes $([a] \cdot w) \in \Sigma^*$
  shows $a \in \Sigma$ **and** $w \in \Sigma^*$
  using *assms WordsOverAlphabet.cases*[*of* $a\#w$]
  by *auto*

lemma *concat-words-over-an-alphabet*:
  fixes v w :: $'a$ *word*
  assumes $v \in \Sigma^*$
    and $w \in \Sigma^*$
   shows $(v \cdot w) \in \Sigma^*$
  using *assms*
proof (*induct* v)
  case *EmptyWord*
  assume $w \in \Sigma^*$
  thus $(\varepsilon \cdot w) \in \Sigma^*$
   by *simp*
next
  case (*Composed* a v)
  assume $a \in \Sigma$
  moreover assume $w \in \Sigma^* \Longrightarrow (v \cdot w) \in \Sigma^*$ **and** $w \in \Sigma^*$
  hence $(v \cdot w) \in \Sigma^*$ **.**
  ultimately show $((a\#v) \cdot w) \in \Sigma^*$
   using *WordsOverAlphabet.Composed*[*of* a $v \cdot w$]
   by *simp*

```
qed

lemma split-a-word-over-an-alphabet:
  fixes v w :: 'a word
  assumes (v · w) ∈ Σ*
  shows v ∈ Σ* and w ∈ Σ*
  using assms
proof (induct v)
  case Nil
  {
    case 1
    show ε ∈ Σ*
      using EmptyWord
      by simp
  next
    case 2
    assume ε · w ∈ Σ*
    thus w ∈ Σ*
      by simp
  }
next
  case (Cons a v)
  assume a#v · w ∈ Σ*
  hence A1: a ∈ Σ and A2: v · w ∈ Σ*
    using word-over-alphabet-rev[of a v · w]
    by simp-all
  assume IH1: v · w ∈ Σ* ⟹ v ∈ Σ* and IH2: v · w ∈ Σ* ⟹ w ∈ Σ*
  {
    case 1
    from A1 A2 IH1 show a#v ∈ Σ*
      using Composed[of a v]
      by simp
  next
    case 2
    from A2 IH2 show w ∈ Σ*
      by simp
  }
qed

end

end

theory Defs
  imports HOL−Library.Sublist FormalLanguages

begin
```

2 Communicating Automata

2.1 Messages and Actions

datatype ($'information$, $'peer$) *message* =
  Message $'information$ $'peer$ $'peer$ ($\text{-}^{\text{-}\to\text{-}}$ [*120*, *120*, *120*] *100*)

primrec *get-information* :: ($'information$, $'peer$) *message* $\Rightarrow$ $'information$ **where**
  get-information ($i^{p \to q}$) = *i*

primrec *get-sender* :: ($'information$, $'peer$) *message* $\Rightarrow$ $'peer$ **where**
  get-sender ($i^{p \to q}$) = *p*

primrec *get-receiver* :: ($'information$, $'peer$) *message* $\Rightarrow$ $'peer$ **where**
  get-receiver ($i^{p \to q}$) = *q*

datatype ($'information$, $'peer$) *action* =
  Output ($'information$, $'peer$) *message* (!⟨-⟩ [*120*] *100*) |
  Input ($'information$, $'peer$) *message* (*?*⟨-⟩ [*120*] *100*)

primrec *is-output* :: ($'information$, $'peer$) *action* $\Rightarrow$ *bool* **where**
  is-output (!⟨*m*⟩) = *True* |
  is-output (*?*⟨*m*⟩) = *False*

abbreviation *is-input* :: ($'information$, $'peer$) *action* $\Rightarrow$ *bool* **where**
  is-input a $\equiv$ $\neg$(*is-output a*)

primrec *get-message* :: ($'information$, $'peer$) *action* $\Rightarrow$ ($'information$, $'peer$) *message* **where**
  get-message (!⟨*m*⟩) = *m* |
  get-message (*?*⟨*m*⟩) = *m*

primrec *get-actor* :: ($'information$, $'peer$) *action* $\Rightarrow$ $'peer$ **where**
  get-actor (!⟨*m*⟩) = *get-sender m* |
  get-actor (*?*⟨*m*⟩) = *get-receiver m*

primrec *get-object* :: ($'information$, $'peer$) *action* $\Rightarrow$ $'peer$ **where**
  get-object (!⟨*m*⟩) = *get-receiver m* |
  get-object (*?*⟨*m*⟩) = *get-sender m*

abbreviation *get-info* :: ($'information$, $'peer$) *action* $\Rightarrow$ $'information$ **where**
  get-info a $\equiv$ *get-information* (*get-message a*)

2.2 Projections Languages

abbreviation *projection-on-outputs*
  :: ($'information$, $'peer$) *action word* $\Rightarrow$ ($'information$, $'peer$) *action word* ($\text{-}\downarrow_!$ [*90*] *110*)

where
$w{\downarrow_!} \equiv$ *filter is-output w*

abbreviation *projection-on-outputs-language*
:: ('*information*, '*peer*) *action language* ⇒ ('*information*, '*peer*) *action language*
($-{\restriction_!}$ [*120*] *100*)
where
$L{\restriction_!} \equiv \{w{\downarrow_!} \mid w.\ w \in L\}$

abbreviation *projection-on-inputs*
:: ('*information*, '*peer*) *action word* ⇒ ('*information*, '*peer*) *action word* ($-{\downarrow_?}$ [*90*] *110*)
where
$w{\downarrow_?} \equiv$ *filter is-input w*

abbreviation *projection-on-inputs-language*
:: ('*information*, '*peer*) *action language* ⇒ ('*information*, '*peer*) *action language*
($-{\restriction_?}$ [*120*] *100*)
where
$L{\restriction_?} \equiv \{w{\downarrow_?} \mid w.\ w \in L\}$

abbreviation *ignore-signs*
:: ('*information*, '*peer*) *action word* ⇒ ('*information*, '*peer*) *message word* ($-{\downarrow_{!?}}$ [*90*] *110*)
where
$w{\downarrow_{!?}} \equiv$ *map get-message w*

abbreviation *ignore-signs-in-language*
:: ('*information*, '*peer*) *action language* ⇒ ('*information*, '*peer*) *message language*
($-{\restriction_{!?}}$ [*90*] *110*) **where**
$L{\restriction_{!?}} \equiv \{w{\downarrow_{!?}} \mid w.\ w \in L\}$

— projection on receives towards p and sends from p
abbreviation *projection-on-single-peer* :: ('*information*, '*peer*) *action word* ⇒ '*peer* ⇒ ('*information*, '*peer*) *action word* ($-{\downarrow_{-}}$ [*90*, *90*] *110*)
where
$w{\downarrow_p} \equiv$ *filter* ($\lambda x.$ *get-actor* $x = p$) *w*

abbreviation *projection-on-single-peer-language*
:: ('*information*, '*peer*) *action language* ⇒ '*peer* ⇒ ('*information*, '*peer*) *action language*
($-{\restriction_{-}}$ [*90*, *90*] *110*) **where**
$(L{\restriction_p}) \equiv \{(w{\downarrow_p}) \mid w.\ w \in L\}$

abbreviation *projection-on-peer-pair*
:: ('*information*, '*peer*) *action word* ⇒ '*peer* ⇒ '*peer* ⇒ ('*information*, '*peer*) *action word* ($-{\downarrow_{\{-,-\}}}$ [*90*, *90*, *90*] *110*)
where
$w{\downarrow_{\{p,q\}}} \equiv$ *filter* ($\lambda x.$ (*get-object* $x = q \wedge$ *get-actor* $x = p$) $\vee$ (*get-object* $x = p$

$\wedge$ *get-actor* $x = q$)) w

abbreviation *projection-on-peer-pair-language*
:: ('*information*, '*peer*) *action language* $\Rightarrow$ '*peer* $\Rightarrow$ '*peer* $\Rightarrow$ ('*information*, '*peer*) *action language*
(-$\downharpoonright_{\{-,-\}}$ [*90*, *90*, *90*] *110*) **where**
$(L\downharpoonright_{\{p,q\}}) \equiv \{(w\downarrow_{\{p,q\}}) \mid w.\ w \in L\}$

2.3 Shuffled Language

inductive *shuffled* ::('*information*, '*peer*) *action word* $\Rightarrow$ ('*information*, '*peer*) *action word* $\Rightarrow$ *bool* **where**

refl: *shuffled* w w |

swap: $[\![$ *is-output* a; *is-input* b ; $w = (xs\ @\ a\ \#\ b\ \#\ ys)$ $]\!]$
$\Longrightarrow$ *shuffled* w $(xs\ @\ b\ \#\ a\ \#\ ys)$ |

trans: $[\![$ *shuffled* w w'; *shuffled* w' w'' $]\!]$ $\Longrightarrow$ *shuffled* w w''

abbreviation *valid-input-shuffles-of-w* :: ('*information*, '*peer*) *action word* $\Rightarrow$ ('*information*, '*peer*) *action language* **where**
valid-input-shuffles-of-w $w \equiv \{w'.$ *shuffled* w $w'\}$

abbreviation *valid-input-shuffle* ::
('*information*, '*peer*) *action word* $\Rightarrow$ ('*information*, '*peer*) *action word* $\Rightarrow$ *bool*
(**infixl** $\sqcup\sqcup_?$ *60*) **where**
$w' \sqcup\sqcup_? w \equiv$ *shuffled* w w'

definition *all-shuffles* :: ('*information*, '*peer*) *action word* $\Rightarrow$ ('*information*, '*peer*) *action word set* **where**
all-shuffles $w = \{w'.$ *shuffled* w $w'\}$

definition *shuffled-lang* :: ('*information*, '*peer*) *action language* $\Rightarrow$ ('*information*, '*peer*) *action language* **where**
shuffled-lang $L = (\bigcup w{\in}L.$ *all-shuffles* $w)$

abbreviation *shuffling-possible* :: ('*information*, '*peer*) *action word* $\Rightarrow$ *bool* **where**
shuffling-possible $w \equiv (\exists\ xs\ a\ b\ ys.$ *is-output* $a \wedge$ *is-input* $b \wedge w = (xs\ @\ a\ \#\ b\ \#\ ys))$

abbreviation *shuffling-occurred* :: ('*information*, '*peer*) *action word* $\Rightarrow$ *bool* **where**
shuffling-occurred $w \equiv (\exists\ xs\ a\ b\ ys.$ *is-output* $a \wedge$ *is-input* $b \wedge w = (xs\ @\ b\ \#\ a\ \#\ ys))$

abbreviation *rightmost-shuffle* :: ('*information*, '*peer*) *action word* $\Rightarrow$ ('*information*,

′peer) action word ⇒ bool **where**
rightmost-shuffle w w′ ≡ (∃ xs a b ys. is-output a ∧ is-input b ∧ w = (xs @ a # b # ys) ∧ (¬ shuffling-possible ys) ∧ w′ = (xs @ b # a # ys))

locale CommunicatingAutomaton =
fixes peer :: ′peer
and States :: ′state set
and initial :: ′state
and Messages :: (′information, ′peer) message set
and Transitions :: (′state × (′information, ′peer) action × ′state) set
assumes finite-states: finite States
and initial-state: initial ∈ States
and message-alphabet: Alphabet Messages
and well-formed-transition: ⋀s1 a s2. (s1, a, s2) ∈ Transitions ⟹
s1 ∈ States ∧ get-message a ∈ Messages ∧ get-actor a = peer ∧
get-object a ≠ peer ∧ s2 ∈ States
begin

inductive-set ActionsOverMessages :: (′information, ′peer) action set **where**
AOMOutput: m ∈ Messages ⟹ !⟨m⟩ ∈ ActionsOverMessages |
AOMInput: m ∈ Messages ⟹ ?⟨m⟩ ∈ ActionsOverMessages

inductive-set Actions :: (′information, ′peer) action set (Act) **where**
ActOfTrans: (s1, a, s2) ∈ Transitions ⟹ a ∈ Act

inductive-set CommunicationPartners :: ′peer set **where**
CPAction: (s1, a, s2) ∈ Transitions ⟹ get-object a ∈ CommunicationPartners

inductive-set SendingToPeers :: ′peer set **where**
SPSend: ⟦(s1, a, s2) ∈ Transitions; is-output a⟧ ⟹ get-object a ∈ SendingToPeers

inductive-set ReceivingFromPeers :: ′peer set **where**
RPRecv: ⟦(s1, a, s2) ∈ Transitions; is-input a⟧ ⟹ get-object a ∈ ReceivingFromPeers

abbreviation step
:: ′state ⇒ (′information, ′peer) action ⇒ ′state ⇒ bool (- −-→$_{\mathcal{C}}$ - [90, 90, 90] 110)
where
s1 −a→$_{\mathcal{C}}$ s2 ≡ (s1, a, s2) ∈ Transitions

inductive run :: ′state ⇒ (′information, ′peer) action word ⇒ ′state list ⇒ bool **where**
REmpty2: run s ε ([]) |
RComposed2: ⟦run s1 w xs; s0 −a→$_{\mathcal{C}}$ s1⟧ ⟹ run s0 (a # w) (s1 # xs)

inductive-set Traces :: (′information, ′peer) action word set **where**

STRun: $run\ initial\ w\ xs \Longrightarrow w \in Traces$

abbreviation *Lang* :: ('*information*, '*peer*) *action language* **where**
$Lang \equiv Traces$

abbreviation *LangSend* :: ('*information*, '*peer*) *action language* **where**
$LangSend \equiv Lang\lfloor_{!}$

abbreviation *LangRecv* :: ('*information*, '*peer*) *action language* **where**
$LangRecv \equiv Lang\lfloor_{?}$

end

2.4 Network of Communicating Automata

locale *NetworkOfCA* =
fixes *automata* :: '*peer* ⇒ ('*state set* × '*state* ×
('*state* × ('*information*, '*peer*) *action* × '*state*) *set*) ($\mathcal{A}$ *1000*)
and *messages* :: ('*information*, '*peer*) *message set* ($\mathcal{M}$ *1000*)
assumes *finite-peers*: *finite* (*UNIV* :: '*peer set*)
and *automaton-of-peer*: $\bigwedge p.\ CommunicatingAutomaton\ p\ (fst\ (\mathcal{A}\ p))\ (fst\ (snd\ (\mathcal{A}\ p)))\ \mathcal{M}$
$(snd\ (snd\ (\mathcal{A}\ p)))$
and *message-alphabet*: $Alphabet\ \mathcal{M}$
and *peers-of-message*: $\bigwedge m.\ m \in \mathcal{M} \Longrightarrow get\text{-}sender\ m \neq get\text{-}receiver\ m$
and *messages-used*: $\forall m \in \mathcal{M}.\ \exists s1\ a\ s2\ p.\ (s1, a, s2) \in snd\ (snd\ (\mathcal{A}\ p)) \wedge m = get\text{-}message\ a$
begin

— all peers in network
abbreviation *get-peers* :: '*peer set* ($\mathcal{P}$ *110*) **where**
$\mathcal{P} \equiv (UNIV :: 'peer\ set)$

abbreviation *get-states* :: '*peer* ⇒ '*state set* ($\mathcal{S}$ - [*90*] *110*) **where**
$\mathcal{S}(p) \equiv fst\ (\mathcal{A}\ p)$

abbreviation *get-initial-state* :: '*peer* ⇒ '*state* ($\mathcal{I}$ - [*90*] *110*) **where**
$\mathcal{I}(p) \equiv fst\ (snd\ (\mathcal{A}\ p))$

abbreviation *get-transitions*
:: '*peer* ⇒ ('*state* × ('*information*, '*peer*) *action* × '*state*) *set* ($\mathcal{R}$ - [*90*] *110*)
where
$\mathcal{R}(p) \equiv snd\ (snd\ (\mathcal{A}\ p))$

abbreviation *WordsOverMessages* :: ('*information*, '*peer*) *message word set* ($\mathcal{M}^*$ *100*) **where**
$\mathcal{M}^* \equiv Alphabet.WordsOverAlphabet\ \mathcal{M}$

— all q that p sends to in Ap (for which there is a transition !p->q in Ap)

abbreviation *sendingToPeers-of-peer* :: $'peer \Rightarrow 'peer\ set$ ($\mathcal{P}_!$ - [*90*] *110*) **where**
$\mathcal{P}_!(p) \equiv CommunicatingAutomaton.SendingToPeers\ (snd\ (snd\ (\mathcal{A}\ p)))$

— all q that p receives from in Ap (for which there is a transition ?q->p in Ap)
abbreviation *receivingFromPeers-of-peer* :: $'peer \Rightarrow 'peer\ set$ ($\mathcal{P}_?$ - [*90*] *110*) **where**
$\mathcal{P}_?(p) \equiv CommunicatingAutomaton.ReceivingFromPeers\ (snd\ (snd\ (\mathcal{A}\ p)))$

abbreviation *Peers-of* :: $'peer \Rightarrow 'peer\ set$ **where**
Peers-of p $\equiv CommunicatingAutomaton.CommunicationPartners\ (snd\ (snd\ (\mathcal{A}\ p)))$

abbreviation *step-of-peer*
:: $'state \Rightarrow ('information, 'peer)\ action \Rightarrow 'peer \Rightarrow 'state \Rightarrow bool$
(- $-\text{-}\rightarrow_{\mathcal{C}}$ - - [*90*, *90*, *90*, *90*] *110*) **where**
$s1\ -a\rightarrow_{\mathcal{C}} p\ s2 \equiv (s1,\ a,\ s2) \in snd\ (snd\ (\mathcal{A}\ p))$

abbreviation *language-of-peer*
:: $'peer \Rightarrow ('information, 'peer)\ action\ language$ ($\mathcal{L}$ - [*90*] *110*) **where**
$\mathcal{L}(p) \equiv CommunicatingAutomaton.Lang\ (fst\ (snd\ (\mathcal{A}\ p)))\ (snd\ (snd\ (\mathcal{A}\ p)))$

abbreviation *output-language-of-peer*
:: $'peer \Rightarrow ('information, 'peer)\ action\ language$ ($\mathcal{L}_!$ - [*90*] *110*) **where**
$\mathcal{L}_!(p) \equiv CommunicatingAutomaton.LangSend\ (fst\ (snd\ (\mathcal{A}\ p)))\ (snd\ (snd\ (\mathcal{A}\ p)))$

abbreviation *input-language-of-peer*
:: $'peer \Rightarrow ('information, 'peer)\ action\ language$ ($\mathcal{L}_?$ - [*90*] *110*) **where**
$\mathcal{L}_?(p) \equiv CommunicatingAutomaton.LangRecv\ (fst\ (snd\ (\mathcal{A}\ p)))\ (snd\ (snd\ (\mathcal{A}\ p)))$

— start in s1, read w (in 0 or more steps) and end in s2
abbreviation *path-of-peer*
:: $'state \Rightarrow ('information, 'peer)\ action\ word \Rightarrow 'peer \Rightarrow 'state \Rightarrow bool$
(- $-\text{-}\rightarrow^*$ - - [*90*, *90*, *90*, *90*] *110*) **where**
$s1\ -w\rightarrow^* p\ s2 \equiv (s1{=}s2 \wedge w = \varepsilon \wedge s1 \in \mathcal{S}\ p) \vee (\exists xs.\ CommunicatingAutomaton.run\ (\mathcal{R}\ p)\ s1\ w\ xs \wedge last\ xs = s2)$

abbreviation *run-of-peer*
:: $'peer \Rightarrow ('information, 'peer)\ action\ word \Rightarrow 'state\ list \Rightarrow bool$ **where**
$run\text{-}of\text{-}peer\ p\ w\ xs \equiv (CommunicatingAutomaton.run\ (\mathcal{R}\ p)\ (\mathcal{I}\ p)\ w\ xs)$

abbreviation *run-of-peer-from-state*
:: $'peer \Rightarrow 'state \Rightarrow ('information, 'peer)\ action\ word \Rightarrow 'state\ list \Rightarrow bool$ **where**
$run\text{-}of\text{-}peer\text{-}from\text{-}state\ p\ s\ w\ xs \equiv (CommunicatingAutomaton.run\ (\mathcal{R}\ p)\ s\ w\ xs)$

fun *get-trans-of-run* :: $'state \Rightarrow ('information, 'peer)\ action\ word \Rightarrow 'state\ list \Rightarrow ('state \times ('information, 'peer)\ action \times 'state)\ list$ **where**

$get\text{-}trans\text{-}of\text{-}run\ s0\ \varepsilon\ []\ =\ []\ |$
$get\text{-}trans\text{-}of\text{-}run\ s0\ [a]\ [s1]\ =\ [(s0,\ a,\ s1)]\ |$
$get\text{-}trans\text{-}of\text{-}run\ s0\ (a\ \#\ as)\ (s1\ \#\ xs)\ =\ (s0,\ a,\ s1)\ \#\ get\text{-}trans\text{-}of\text{-}run\ s1\ as\ xs$

2.5 Synchronous System

definition *is-sync-config* :: $('peer \Rightarrow 'state) \Rightarrow bool$ **where**
$is\text{-}sync\text{-}config\ C \equiv (\forall p.\ C\ p \in \mathcal{S}(p))$

abbreviation *initial-sync-config* :: $'peer \Rightarrow 'state\ \ (\mathcal{C}_{\mathcal{I}\mathbf{0}})$ **where**
$\mathcal{C}_{\mathcal{I}\mathbf{0}} \equiv \lambda p.\ \mathcal{I}(p)$

inductive *sync-step*
:: $('peer \Rightarrow 'state) \Rightarrow ('information,\ 'peer)\ action \Rightarrow ('peer \Rightarrow 'state) \Rightarrow bool$
$(\text{-}\ -\langle\text{-},\ \mathbf{0}\rangle\rightarrow\ \text{-}\ [90,\ 90,\ 90]\ 110)$ **where**
$SynchStep: [\![is\text{-}sync\text{-}config\ C1;\ a = !\langle(i^{p\rightarrow q})\rangle;\ C1\ p\ -!\langle(i^{p\rightarrow q})\rangle\rightarrow_{\mathcal{C}} p\ (C2\ p);$
$C1\ q\ -?\langle(i^{p\rightarrow q})\rangle\rightarrow_{\mathcal{C}} q\ (C2\ q);\ \forall x.\ x \notin \{p,\ q\} \longrightarrow C1(x) = C2(x)]\!] \Longrightarrow$
$C1\ -\langle a,\ \mathbf{0}\rangle\rightarrow\ C2$

inductive *sync-run*
:: $('peer \Rightarrow 'state) \Rightarrow ('information,\ 'peer)\ action\ word \Rightarrow ('peer \Rightarrow 'state)\ list$
$\Rightarrow bool$
where
$SREmpty: \quad sync\text{-}run\ C\ \varepsilon\ ([])\ |$
$SRComposed: [\![sync\text{-}run\ C0\ w\ xc;\ last\ (C0\#xc)\ -\langle a,\ \mathbf{0}\rangle\rightarrow\ C]\!] \Longrightarrow sync\text{-}run\ C0$
$(w\cdot[a])\ (xc@[C])$

— E(Nsync)
inductive-set *SyncTraces* :: $('information,\ 'peer)\ action\ language\ \ (\mathcal{T}_{\mathbf{0}}\ 120)$ **where**
$STRun: sync\text{-}run\ \mathcal{C}_{\mathcal{I}\mathbf{0}}\ w\ xc \Longrightarrow w \in \mathcal{T}_{\mathbf{0}}$

— T(Nsync)
abbreviation *SyncLang* :: $('information,\ 'peer)\ action\ language\ \ (\mathcal{L}_{\mathbf{0}}\ 120)$ **where**
$\mathcal{L}_{\mathbf{0}} \equiv \mathcal{T}_{\mathbf{0}}$

2.6 Mailbox System

2.6.1 Semantics and Language

definition *is-mbox-config*
:: $('peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)) \Rightarrow bool$ **where**
$is\text{-}mbox\text{-}config\ C \equiv (\forall p.\ fst\ (C\ p) \in \mathcal{S}(p) \wedge snd\ (C\ p) \in \mathcal{M}^*)$

— all mbox configurations of system
abbreviation *mbox-configs*
:: $('peer \Rightarrow 'state \times ('information,\ 'peer)\ message\ list)\ set\ \ (\mathcal{C}_{\mathfrak{m}})$ **where**
$\mathcal{C}_{\mathfrak{m}} \equiv \{C \mid C.\ is\text{-}mbox\text{-}config\ C\}$

abbreviation *initial-mbox-config*
:: $'peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)\ \ (\mathcal{C}_{\mathcal{I}\mathfrak{m}})$ **where**

$\mathcal{C}_{\mathcal{I}\mathfrak{m}} \equiv \lambda p.\ (\mathcal{I}\ p,\ \varepsilon)$

definition *is-stable*
:: $('peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)) \Rightarrow bool$ **where**
$is\text{-}stable\ C \equiv is\text{-}mbox\text{-}config\ C \wedge (\forall\, p.\ snd\ (C\ p) = \varepsilon)$

type-synonym *bound* = *nat option*

abbreviation *nat-bound* :: $nat \Rightarrow bound$ ($\mathcal{B}$ - [90] 110) **where**
$\mathcal{B}\ k \equiv Some\ k$

abbreviation *unbounded* :: *bound* (∞ 100) **where**
$\infty \equiv None$

primrec *is-bounded* :: $nat \Rightarrow bound \Rightarrow bool$ (- $<_{\mathcal{B}}$ - [90, 90] 110) **where**
$n <_{\mathcal{B}} \infty = True$ |
$n <_{\mathcal{B}} \mathcal{B}\ k = (n < k)$

inductive *mbox-step*
:: $('peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)) \Rightarrow ('information,\ 'peer)\ action \Rightarrow$
$bound \Rightarrow ('peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)) \Rightarrow bool$
where
MboxSend: $[\![is\text{-}mbox\text{-}config\ C1;\ a = !\langle(i^{p \rightarrow q})\rangle;\ fst\ (C1\ p) -!\langle(i^{p \rightarrow q})\rangle\rightarrow_{\mathcal{C}} p\ (fst\ (C2\ p));$
$snd\ (C1\ p) = snd\ (C2\ p);\ (\ |\ (snd\ (C1\ q))\ |\ ) <_{\mathcal{B}} k;$
$C2\ q = (fst\ (C1\ q),\ (snd\ (C1\ q)) \cdot [(i^{p \rightarrow q})]);\ \forall\, x.\ x \notin \{p,\ q\} \longrightarrow C1(x) = C2(x)]\!] \Longrightarrow$
$mbox\text{-}step\ C1\ a\ k\ C2$ |
MboxRecv: $[\![is\text{-}mbox\text{-}config\ C1;\ a = ?\langle(i^{p \rightarrow q})\rangle;\ fst\ (C1\ q) -?\langle(i^{p \rightarrow q})\rangle\rightarrow_{\mathcal{C}} q\ (fst\ (C2\ q));$
$(snd\ (C1\ q)) = [(i^{p \rightarrow q})] \cdot snd\ (C2\ q);\ \forall\, x.\ x \neq q \longrightarrow C1(x) = C2(x)]\!] \Longrightarrow$
$mbox\text{-}step\ C1\ a\ k\ C2$

abbreviation *mbox-step-bounded*
:: $('peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)) \Rightarrow ('information,\ 'peer)\ action \Rightarrow$
$nat \Rightarrow ('peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)) \Rightarrow bool$
(- $-\langle$-, -$\rangle\rightarrow$ - [90, 90, 90, 90] 110) **where**
$C1 -\langle a,\ n\rangle\rightarrow C2 \equiv mbox\text{-}step\ C1\ a\ (Some\ n)\ C2$

abbreviation *mbox-step-unbounded*
:: $('peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)) \Rightarrow ('information,\ 'peer)\ action \Rightarrow$
$('peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)) \Rightarrow bool$
(- $-\langle$-, $\infty\rangle\rightarrow$ - [90, 90, 90] 110) **where**
$C1 -\langle a,\ \infty\rangle\rightarrow C2 \equiv mbox\text{-}step\ C1\ a\ None\ C2$

2.6.2 mbox run

inductive *mbox-run*
$:: ('peer \Rightarrow ('state \times ('information, 'peer)\ message\ word)) \Rightarrow bound \Rightarrow$
$('information, 'peer)\ action\ word \Rightarrow$
$('peer \Rightarrow ('state \times ('information, 'peer)\ message\ word))\ list \Rightarrow bool$ **where**
$MREmpty$: $\quad mbox\text{-}run\ C\ k\ \varepsilon\ ([])\ |$
$MRComposedNat$: $[\![mbox\text{-}run\ C0\ (Some\ k)\ w\ xc;\ last\ (C0\#xc)\ -\langle a, k\rangle\rightarrow\ C]\!] \Longrightarrow$
$mbox\text{-}run\ C0\ (Some\ k)\ (w\cdot[a])\ (xc@[C])\ |$
$MRComposedInf$: $[\![mbox\text{-}run\ C0\ None\ w\ xc;\ last\ (C0\#xc)\ -\langle a, \infty\rangle\rightarrow\ C]\!] \Longrightarrow$
$mbox\text{-}run\ C0\ None\ (w\cdot[a])\ (xc@[C])$

2.6.3 mbox traces

inductive-set *MboxTraces*
$:: nat\ option \Rightarrow ('information, 'peer)\ action\ language\ \ (\mathcal{T}_{\_}\ [100]\ 120)$
for $k :: nat\ option$ **where**
$MTRun$: $mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ k\ w\ xc \Longrightarrow w \in \mathcal{T}_k$

— T(mbox)
abbreviation *MboxLang* $:: bound \Rightarrow ('information, 'peer)\ action\ language\ \ (\mathcal{L}_{\_}\ [100]\ 120)$
where
$\mathcal{L}_k \equiv \{\ w\downarrow_!\ |\ w.\ w \in \mathcal{T}_k\ \}$

abbreviation *MboxLang-bounded-by-one* $:: ('information, 'peer)\ action\ language\ (\mathcal{L}_{\mathbf{1}}\ 120)$ **where**
$\mathcal{L}_{\mathbf{1}} \equiv \mathcal{L}_{\mathcal{B}\ 1}$

abbreviation *MboxLang-unbounded* $:: ('information, 'peer)\ action\ language\ \ (\mathcal{L}_\infty\ 120)$ **where**
$\mathcal{L}_\infty \equiv \mathcal{L}_\infty$

abbreviation *MboxLangSend* $:: bound \Rightarrow ('information, 'peer)\ action\ language\ (\mathcal{L}_{!\_}\ [100]\ 120)$
where
$\mathcal{L}_{!k} \equiv (\mathcal{L}_k)\lfloor_!$

abbreviation *MboxLangRecv* $:: bound \Rightarrow ('information, 'peer)\ action\ language\ (\mathcal{L}_{?\_}\ [100]\ 120)$
where
$\mathcal{L}_{?k} \equiv (\mathcal{L}_k)\lfloor_?$

3 Synchronisability

abbreviation *is-synchronisable* $:: bool$ **where**
$is\text{-}synchronisable \equiv \mathcal{L}_\infty = \mathcal{L}_{\mathbf{0}}$

type-synonym $'a\ topology = ('a \times 'a)\ set$

— the topology graph of all peers
inductive-set *Edges* :: *'peer topology* ($\mathcal{G}$ *110*) **where**
TEdge: $i^{p \rightarrow q} \in \mathcal{M} \Longrightarrow (p, q) \in \mathcal{G}$

abbreviation *Successors* :: *'peer topology* ⇒ *'peer* ⇒ *'peer set* (-⟨-→⟩ [*90*, *90*] *110*) **where**
$E\langle p\rightarrow\rangle \equiv \{q.\ (p, q) \in E\}$

abbreviation *Predecessors* :: *'peer topology* ⇒ *'peer* ⇒ *'peer set* (-⟨→-⟩ [*90*, *90*] *110*) **where**
$E\langle\rightarrow q\rangle \equiv \{p.\ (p, q) \in E\}$

3.1 Synchronisability is Deciable for Tree Topology in Mailbox Communication

3.1.1 Topology is a Tree

inductive *is-tree* :: *'peer set* ⇒ *'peer topology* ⇒ *bool* **where**
ITRoot: *is-tree* $\{p\}$ $\{\}$ |
ITNode: ⟦*is-tree P E*; $p \in P$; $q \notin P$⟧ ⟹ *is-tree* (*insert q P*) (*insert* (p, q) *E*)

abbreviation *tree-topology* :: *bool* **where**
tree-topology ≡ *is-tree* (*UNIV* :: *'peer set*) ($\mathcal{G}$)

abbreviation *is-root-from-topology* :: *'peer* ⇒ *bool* **where**
is-root-from-topology p ≡ (*tree-topology* ∧ $\mathcal{G}\langle\rightarrow p\rangle = \{\}$)

abbreviation *is-root-from-local* :: *'peer* ⇒ *bool* **where**
is-root-from-local p ≡ *tree-topology* ∧ $\mathcal{P}_?(p) = \{\} \wedge (\forall q.\ p \notin \mathcal{P}_!(q))$

abbreviation *is-root* :: *'peer* ⇒ *bool* **where**
is-root p ≡ *is-root-from-local p* ∨ *is-root-from-topology p*

abbreviation *is-node-from-topology* :: *'peer* ⇒ *bool* **where**
is-node-from-topology p ≡ (*tree-topology* ∧ $(\exists q.\ \mathcal{G}\langle\rightarrow p\rangle = \{q\})$)

abbreviation *is-node-from-local* :: *'peer* ⇒ *bool* **where**
is-node-from-local p ≡ *tree-topology* ∧ $(\exists q.\ \mathcal{P}_?(p) = \{q\} \vee p \in \mathcal{P}_!(q))$

abbreviation *is-node* :: *'peer* ⇒ *bool* **where**
is-node p ≡ *is-node-from-topology p* ∨ *is-node-from-local p*

3.1.2 parent-child relationship in tree

inductive *is-parent-of* :: *'peer* ⇒ *'peer* ⇒ *bool* **where**
node-parent : ⟦*is-node p*; $\mathcal{G}\langle\rightarrow p\rangle = \{q\}$⟧ ⟹ *is-parent-of p q*

3.1.3 path to root and path related lemmas

inductive *path-to-root* :: *'peer* $\Rightarrow$ *'peer list* $\Rightarrow$ *bool* **where**
PTRRoot: $[\![$*is-root p*$]\!] \Longrightarrow$ *path-to-root p* $[p]$ |
PTRNode: $[\![$*tree-topology*; *is-parent-of p q*; *path-to-root q as*; *distinct* $(p \# as)]\!]$ $\Longrightarrow$ *path-to-root p* $(p \# as)$

definition *get-root* :: *'peer topology* $\Rightarrow$ *'peer* **where** *get-root E* = (*THE x. is-root x*)

abbreviation *get-path-to-root* :: *'peer* $\Rightarrow$ *'peer list* **where**
get-path-to-root p $\equiv$ (*THE ps. path-to-root p ps*)

inductive *path-from-root* :: *'peer* $\Rightarrow$ *'peer list* $\Rightarrow$ *bool* **where**
PFRRoot: $[\![$*is-root p*$]\!] \Longrightarrow$ *path-from-root p* $[p]$ |
PFRNode: $[\![$*tree-topology*; *is-parent-of p q*; *path-from-root q as*; *distinct* $(as\ @\ [p])]\!] \Longrightarrow$ *path-from-root p* $(as\ @\ [p])$

inductive *path-from-to* :: *'peer* $\Rightarrow$ *'peer* $\Rightarrow$ *'peer list* $\Rightarrow$ *bool* **where**
path-refl: $[\![$*tree-topology*; $p \in \mathcal{P}]\!] \Longrightarrow$ *path-from-to p p* $[p]$ |
path-step: $[\![$*tree-topology*; *is-parent-of p q*; *path-from-to r q as*; *distinct* $(as\ @\ [p])]\!]$ $\Longrightarrow$ *path-from-to r p* $(as\ @\ [p])$

3.2 Influenced Language

inductive *is-in-infl-lang* :: *'peer* $\Rightarrow$ (*'information*, *'peer*) *action word* $\Rightarrow$ *bool* **where**
IL-root: $[\![$*is-root r*; $w \in \mathcal{L}(r)]\!] \Longrightarrow$ *is-in-infl-lang r w* | — influenced language of root r is language of r
IL-node: $[\![$*tree-topology*; *is-parent-of p q*; $w \in \mathcal{L}(p)$; *is-in-infl-lang q w'*; $((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})]\!] \Longrightarrow$ *is-in-infl-lang p w* — p is any node and q its parent has a matching send for each of p's receives

abbreviation *InfluencedLanguage* :: *'peer* $\Rightarrow$ (*'information*, *'peer*) *action language* ($\mathcal{L}^*$ - [*90*] *100*) **where**
$\mathcal{L}^*\ p \equiv \{w.\ \textit{is-in-infl-lang}\ p\ w\}$

abbreviation *InfluencedLanguageSend* :: *'peer* $\Rightarrow$ (*'information*, *'peer*) *action language* ($\mathcal{L}_!{}^*$ - [*90*] *100*) **where**
$\mathcal{L}_!{}^*\ p \equiv (\mathcal{L}^*\ p)|_!$

abbreviation *InfluencedLanguageRecv* :: *'peer* $\Rightarrow$ (*'information*, *'peer*) *action language* ($\mathcal{L}_?{}^*$ - [*90*] *100*) **where**
$\mathcal{L}_?{}^*\ p \equiv (\mathcal{L}^*\ p)|_?$

abbreviation *ShuffledInfluencedLanguage* :: *'peer* $\Rightarrow$ (*'information*, *'peer*) *action language* ($\mathcal{L}^*{}_{\sqcup\sqcup}$ - [*90*] *100*) **where**
$\mathcal{L}^*{}_{\sqcup\sqcup}\ p \equiv$ *shuffled-lang* $(\mathcal{L}^*\ p)$

— p receives from no one and there is no q that sends to p
abbreviation *no-sends-to-or-recvs-in* :: *'peer* $\Rightarrow$ *bool* **where**

$no\text{-}sends\text{-}to\text{-}or\text{-}recvs\text{-}in\ p \equiv (\mathcal{P}_?(p) = \{\} \wedge (\forall\, q.\ p \notin \mathcal{P}_!(q)))$

3.2.1 simulate sync with mbox word

fun *add-matching-recvs* :: ($'information$, $'peer$) *action word* $\Rightarrow$ ($'information$, $'peer$) *action word* **where**
  $add\text{-}matching\text{-}recvs\ [] = []\ |$
  $add\text{-}matching\text{-}recvs\ (a\ \#\ w) = (if\ is\text{-}output\ a$
    $then\ a\ \#\ (?\langle get\text{-}message\ a\rangle)\ \#\ add\text{-}matching\text{-}recvs\ w$
    $else\ a\ \#\ add\text{-}matching\text{-}recvs\ w)$

3.2.2 Lemma 4.4 and preparations

inductive *acc-infl-lang-word* :: $'peer \Rightarrow ('information, 'peer)\ action\ word \Rightarrow bool$ **where**
  $ACC\text{-}root$: $[\![is\text{-}root\ r;\ w \in \mathcal{L}^*(r)]\!] \Longrightarrow acc\text{-}infl\text{-}lang\text{-}word\ r\ w$ | — influenced language of root r is language of r
  $ACC\text{-}node$: $[\![tree\text{-}topology;\ is\text{-}parent\text{-}of\ p\ q;\ w \in \mathcal{L}^*(p);\ acc\text{-}infl\text{-}lang\text{-}word\ q\ w';\ ((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})]\!] \Longrightarrow acc\text{-}infl\text{-}lang\text{-}word\ p\ (w'\ @\ w)$ — p is any node and q its parent has a matching send for each of p's receives

inductive *concat-infl* :: $'peer \Rightarrow ('information, 'peer)\ action\ word \Rightarrow 'peer\ list \Rightarrow ('information, 'peer)\ action\ word \Rightarrow bool$ **for** p::$'peer$ **and** w:: ($'information$, $'peer$) *action word* **where**
  $at\text{-}p$: $[\![tree\text{-}topology;\ w \in \mathcal{L}^*(p);\ path\text{-}to\text{-}root\ p\ ps]\!] \Longrightarrow concat\text{-}infl\ p\ w\ ps\ w$ |
  $reach\text{-}root$: $[\![is\text{-}root\ q;\ qw \in \mathcal{L}^*(q);\ path\text{-}to\text{-}root\ x\ (x\ \#\ [q]);\ (\forall\, g.\ w\text{-}acc{\downarrow_g} \in \mathcal{L}^*(g));\ concat\text{-}infl\ p\ w\ (x\ \#\ [q])\ w\text{-}acc;\ (((w\text{-}acc{\downarrow_x}){\downarrow_?}){\downarrow_{!?}}) = (((qw{\downarrow_{\{x,q\}}}){\downarrow_!}){\downarrow_{!?}})]\!] \Longrightarrow concat\text{-}infl\ p\ w\ [q]\ (qw \cdot w\text{-}acc)$ |
  $node\text{-}step$: $[\![tree\text{-}topology;\ \mathcal{P}_?(x) = \{q\};\ (\forall\, g.\ w\text{-}acc{\downarrow_g} \in \mathcal{L}^*(g));\ path\text{-}to\text{-}root\ x\ (x\ \#\ q\ \#\ ps);\ qw \in \mathcal{L}^*(q);\ concat\text{-}infl\ p\ w\ (x\ \#\ q\ \#\ ps)\ w\text{-}acc;\ (((w\text{-}acc{\downarrow_x}){\downarrow_?}){\downarrow_{!?}}) = (((qw{\downarrow_{\{x,q\}}}){\downarrow_!}){\downarrow_{!?}})]\!] \Longrightarrow concat\text{-}infl\ p\ w\ (q\#ps)\ (qw \cdot w\text{-}acc)$

4 new formalization

abbreviation *possible-recv-suffixes* :: $('information, 'peer)\ action\ word \Rightarrow 'peer \Rightarrow ('information, 'peer)\ action\ language$ (‡-‡- [*90*, *90*] *110*) **where**
  $‡w‡_p \equiv \{x{\downarrow_?} \mid x.\ (w \cdot x) \in \mathcal{L}^*(p)\}$

abbreviation *possible-send-suffixes-to-peer* :: $'peer \Rightarrow ('information, 'peer)\ action\ word \Rightarrow 'peer \Rightarrow ('information, 'peer)\ action\ language$ (-‡-‡- [*90*, *90*, *90*] *110*) **where**
  ${}_q‡w‡_p \equiv \{(x{\downarrow_!}){\downarrow_{\{p,q\}}} \mid x.\ (w \cdot x) \in \mathcal{L}^*(q)\}$

definition *subset-condition* :: $'peer \Rightarrow 'peer \Rightarrow bool$
  where $subset\text{-}condition\ p\ q \longleftrightarrow (\forall\ w \in \mathcal{L}^*(p).\ \forall\ w' \in \mathcal{L}^*(q).$

$(((w'\downarrow_{!})\downarrow_{\{p,q\}})\downarrow_{!?} = ((w\downarrow_{?})\downarrow_{!?})) \longrightarrow ((_{q}\ddagger w'\ddagger_{p})\vert_{!?} \subseteq (\ddagger w\ddagger_{p})\vert_{!?}\ ))$

definition *theorem-rightside* :: *bool*
where *theorem-rightside* $\longleftrightarrow (\forall\, p \in \mathcal{P}.\ \forall\, q \in \mathcal{P}.\ ((\textit{is-parent-of}\ p\ q) \longrightarrow ((\textit{subset-condition}\ p\ q) \wedge ((\mathcal{L}^{*}(p)) = (\mathcal{L}^{*}{}_{\sqcup\sqcup}(p))))\ ))$

4.1 => 1.

fun *mix-pair* :: ('*information*, '*peer*) *action word* ⇒ ('*information*, '*peer*) *action word* ⇒ ('*information*, '*peer*) *action word* ⇒ ('*information*, '*peer*) *action word*
where
mix-pair [] [] *acc* = *acc* |
mix-pair (*a* # *w'*) [] *acc* = *mix-pair w'* [] (*a* # *acc*) |
mix-pair [] (*a* # *w*) *acc* = *mix-pair* [] *w* (*a* # *acc*) |
mix-pair (*a* # *w'*) (*b* # *w*) *acc* = (*if a* = !⟨*get-message b*⟩
then (*if b* = ?⟨*get-message a*⟩ *then mix-pair w' w* (*a* # *b* # *acc*) *else mix-pair* (*a* # *w'*) *w* (*b* # *acc*))
else mix-pair w' (*b* # *w*) (*a* # *acc*))

4.2 => 2.

inductive *mix-shuf* :: ('*information*, '*peer*) *action word* ⇒ ('*information*, '*peer*) *action word* ⇒ ('*information*, '*peer*) *action word* ⇒ ('*information*, '*peer*) *action word* ⇒ *bool* **where**
mix-shuf-constr: $[\![vq\downarrow_{!}\downarrow_{\{p,q\}}\downarrow_{!?} = v\downarrow_{?}\downarrow_{!?};\ v' \in \mathcal{L}^{*}{}_{\sqcup\sqcup}(p);\ v' \sqcup\sqcup_{?}\ v;\ v \in \mathcal{L}^{*}(p);\ vq \in \mathcal{L}^{*}(q);$
$vq = (as \cdot \textit{a-send}\ \#\ bs);\ v = xs \cdot b\ \#\ \textit{a-recv}\ \#\ ys;\ \textit{get-message a-recv} = \textit{get-message a-send};\ \textit{is-input a-recv};\ \textit{is-output a-send};\ \textit{is-output}\ b]\!]$
$\Longrightarrow \textit{mix-shuf}\ vq\ v\ v'\ ((\textit{mix-pair}\ as\ xs\ []) \cdot \textit{a-send}\ \#\ b\ \#\ \textit{a-recv}\ \#\ (\textit{mix-pair}\ bs\ ys\ []))$

end
end

theory *CommunicatingAutomaton*
imports *Defs*

begin

declare [[*quick-and-dirty=true*]]

5 Communicating Automata

5.0.1 projection simplifications on words/general cases

lemma *proj-trio-inv*:
  shows $((w{\downarrow_q}){\downarrow_!}){\downarrow_{\{p,q\}}} = ((w{\downarrow_!}){\downarrow_q}){\downarrow_{\{p,q\}}}$
proof (*induct w*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a w*)
  then show *?case* **by** *fastforce*
qed

lemma *proj-trio-inv2*:
  shows $(((w'{\downarrow_!}){\downarrow_q}){\downarrow_{\{p,q\}}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_q})$
proof (*induct w'*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a w*)
  then show *?case* **by** (*metis* (*no-types*, *lifting*) *filter.simps*(*2*))
qed

lemma *filter-recursion* : *filter f* (*filter f xs*) = *filter f xs* **by** *simp*

lemma *filter-head-helper* :
  assumes *x* # (*filter f xs*) = (*filter f* (*x#xs*))
  shows *f x*
proof (*induction xs*)
  case *Nil*
  then show *?case* **by** (*meson Cons-eq-filterD assms*)
next
  case (*Cons a xs*)
  then show *?case* **by** *simp*
qed

lemma *output-proj-input-yields-eps* :
  assumes $(w{\downarrow_!}) = w$
  shows $(w{\downarrow_?}) = \varepsilon$
  by (*metis assms filter-False filter-id-conv*)

lemma *input-proj-output-yields-eps* :
  assumes $(w{\downarrow_?}) = w$
  shows $(w{\downarrow_!}) = \varepsilon$
  by (*metis assms filter-False filter-id-conv*)

lemma *input-proj-nonempty-impl-input-act* :

assumes $(w{\downarrow_?}) \neq \varepsilon$
shows $\exists$ *xs a ys*. $((w{\downarrow_?}) = (xs\ @\ [a]\ @\ ys)) \wedge$ *is-input a*
by (*metis append.left-neutral append-Cons assms filter.simps*(*2*) *filter-recursion input-proj-output-yields-eps list.distinct*(*1*) *list.exhaust*)

lemma *output-proj-nonempty-impl-input-act* :
assumes $(w{\downarrow_!}) \neq \varepsilon$
shows $\exists$ *xs a ys*. $((w{\downarrow_!}) = (xs\ @\ [a]\ @\ ys)) \wedge$ *is-output a*
by (*metis append.left-neutral append-Cons assms filter-empty-conv filter-recursion split-list*)

lemma *decompose-send* :
assumes $(w{\downarrow_!}) \neq \varepsilon$
shows $\exists\, v\ a\ q\ p.\ (w{\downarrow_!}) = v \cdot [!\langle(a^{q \to p})\rangle]$
proof –
have $\exists\ v\ x.\ (w{\downarrow_!}) = v \cdot [x]$ **by** (*metis assms rev-exhaust*)
then obtain *v x* **where** $(w{\downarrow_!}) = v \cdot [x]$ **by** *auto*
then have *is-output x* **by** (*metis assms filter-id-conv filter-recursion last-in-set last-snoc*)
then obtain *a q p* **where** $x = !\langle(a^{q \to p})\rangle$ **by** (*metis action.exhaust is-output.simps*(*2*) *message.exhaust*)
then show *?thesis* **by** (*simp add*: ‹$w{\downarrow_!} = v \cdot x \# \varepsilon$›)
qed

lemma *only-one-actor-proj*:
assumes $w = w{\downarrow_q}$ **and** $p \neq q$
shows $w{\downarrow_p} = \varepsilon$
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*,*2*) *filter-False filter-id-conv*)

lemma *filter-pair-commutative*:
shows *filter g* (*filter f xs*) = *filter f* (*filter g xs*)
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
then show *?case*
by (*simp add*: *conj-commute*)
qed

lemma *pair-proj-to-object-proj*:
assumes $(w{\downarrow_p}) = w$
shows $w{\downarrow_{\{p,q\}}}$ = (*filter* (λx. *get-object* $x = q$) *w*)
by (*smt* (*verit*, *del-insts*) *assms filter-cong filter-id-conv*)

lemma *actor-proj-app-inv*:
assumes $(u@v){\downarrow_p} = (u@v)$
shows $u = u{\downarrow_p} \wedge v = v{\downarrow_p}$
using *assms*

proof –
from *assms* **have** $(u@v)\downarrow_p = u$ @ v
by *simp*
moreover have $(u@v)\downarrow_p = (u)\downarrow_p$ @ $(v)\downarrow_p$
by (*rule filter-append*)
ultimately have *eq*: $(u)\downarrow_p$ @ $(v)\downarrow_p = u$ @ v **by** *argo*
have *u-len*: *length* $(u\downarrow_p) \leq$ *length* u **using** *length-filter-le* **by** *blast*
have *v-len*: *length* $(v\downarrow_p) \leq$ *length* v **using** *length-filter-le* **by** *blast*
have *t1*: $(u)\downarrow_p = u$
proof (*rule ccontr*)
assume $u\downarrow_p \neq u$
then have *length* $(u\downarrow_p) <$ *length* u **by** (*metis u-len* ‹$(u \cdot v)\downarrow_p = u\downarrow_p \cdot v\downarrow_p$› ‹$u\downarrow_p \neq u$› *append-eq-append-conv assms le-neq-implies-less*)
then have *length* $((u)\downarrow_p$ @ $(v)\downarrow_p) \leq$ *length* $((u@v))$ **by** (*metis* ‹$(u \cdot v)\downarrow_p = u\downarrow_p \cdot v\downarrow_p$› *length-filter-le*)
have *length* $((u)\downarrow_p$ @ $(v)\downarrow_p) =$ *length* $(u\downarrow_p)$ + *length* $(v\downarrow_p)$ **by** *simp*
have *length* $(u\downarrow_p)$ + *length* $(v\downarrow_p) <$ *length* (u) + *length* (v) **by** (*simp add*: ‹$|u\downarrow_p| < |u|$› *add-less-le-mono*)
then show *False* **using** *eq length-append less-not-refl* **by** *metis*
qed
have *t2*: $(v)\downarrow_p = v$
proof (*rule ccontr*)
assume $v\downarrow_p \neq v$
then have *length* $(v\downarrow_p) <$ *length* v **by** (*metis v-len* ‹$(u \cdot v)\downarrow_p = u\downarrow_p \cdot v\downarrow_p$› ‹$v\downarrow_p \neq v$› *append-eq-append-conv assms le-neq-implies-less*)
then have *length* $((u)\downarrow_p$ @ $(v)\downarrow_p) \leq$ *length* $((u@v))$ **by** (*metis* ‹$(u \cdot v)\downarrow_p = u\downarrow_p \cdot v\downarrow_p$› *length-filter-le*)
have *length* $((u)\downarrow_p$ @ $(v)\downarrow_p) =$ *length* $(u\downarrow_p)$ + *length* $(v\downarrow_p)$ **by** *simp*
then show *False* **using** ‹$(u \cdot v)\downarrow_p = u\downarrow_p \cdot v\downarrow_p$› ‹$u\downarrow_p = u$› ‹$v\downarrow_p \neq v$› *assms same-append-eq* **by** *metis*
qed

show *?thesis* **using** *t1 t2* **by** *simp*
qed

lemma *actors-4-proj-app-inv*:
assumes $(a$ @ b @ c @ $d)\downarrow_p = (a$ @ b @ c @ $d)$
shows $a\downarrow_p = a \wedge b\downarrow_p = b \wedge c\downarrow_p = c \wedge d\downarrow_p = d$
by (*metis actor-proj-app-inv assms*)

lemma *not-only-sends-impl-recv*:
assumes $w \neq w\downarrow_!$
shows $\exists x.\ x \in$ *set* $w \wedge$ *is-input* x
by (*metis assms filter-True*)

lemma *orderings-inv-for-prepend*:
assumes $w\downarrow_? = w'\downarrow_?$ **and** $w\downarrow_! = w'\downarrow_!$
shows $(a \# w)\downarrow_? = (a \# w')\downarrow_? \wedge (a \# w)\downarrow_! = (a \# w')\downarrow_!$
by (*simp add*: *assms*(*1*,*2*))

lemma *orderings-inv-for-prepend-rev*:
assumes $(a \ \# \ w)\downarrow_? = (a \ \# \ w')\downarrow_?$ **and** $(a \ \# \ w)\downarrow_! = (a \ \# \ w')\downarrow_!$
shows $w\downarrow_? = w'\downarrow_? \wedge w\downarrow_! = w'\downarrow_!$
by (*metis* (*no-types*, *lifting*) *assms*(*1*,*2*) *filter.simps*(*2*) *list.inject*)

lemma *prefix-trans*:
assumes *prefix x z*
shows $\exists \ y.$ *prefix y z* $\wedge x = y$
by (*simp add*: *assms*)

lemma *prefix-inv-no-signs*:
assumes *prefix w w′*
shows *prefix* $(w\downarrow_{!?})$ $(w'\downarrow_{!?})$
using *map-mono-prefix assms* **by** *auto*

5.1 Shuffled Language

lemma *shuffled-rev*:
assumes *shuffled w w′*
shows $w = w' \vee (\exists \ a \ b \ xs \ ys. \ w = \ (xs \ @ \ a \ \# \ b \ \# \ ys) \wedge$ *is-output a* $\wedge$ *is-input b* $\wedge w' = (xs \ @ \ b \ \# \ a \ \# \ ys)) \vee (\exists \, tmp.$ *shuffled w tmp* $\wedge$ *shuffled tmp w′*)
using *assms shuffled.refl* **by** *blast*

lemma *shuffled-prepend-inductive*:
assumes *shuffled w w′*
shows *shuffled* $(a \ \# \ w)$ $(a \ \# \ w')$
using *assms*
proof (*induct*)
case (*refl w*)
then show *?case* **using** *shuffled.refl* **by** *auto*
next
case (*swap a b w xs ys*)
then show *?case* **by** (*metis* (*no-types*, *lifting*) *Cons-eq-appendI shuffled.simps*)
next
case (*trans w w′ w′′*)
then show *?case* **using** *shuffled.trans* **by** *auto*
qed

lemma *fully-shuffled-gen*:
assumes $xs = xs\downarrow_!$
shows *shuffled* $(xs \ @ \ [?\langle(a^{q \rightarrow p})\rangle])$ $([?\langle(a^{q \rightarrow p})\rangle] \ @ \ xs)$
using *assms*
proof (*induct xs*)
case *Nil*
then show *?case* **by** (*simp add*: *shuffled.refl*)
next
case (*Cons y ys*)

then have *ys* = *ys*$\downarrow_!$ **by** (*metis filter.simps*(*2*) *impossible-Cons length-filter-le list.inject*)
then have *shuffled* (*ys* $\cdot$ *?*⟨($a^{q \rightarrow p}$)⟩ # ε) (*?*⟨($a^{q \rightarrow p}$)⟩ # ε $\cdot$ *ys*) **using** *Cons.hyps* **by** *blast*
have *is-output y* **by** (*meson Cons.prems Cons-eq-filterD*)
then have *last-step*: *shuffled* (*y* # *?*⟨($a^{q \rightarrow p}$)⟩ # *ys*) (*?*⟨($a^{q \rightarrow p}$)⟩ # ε $\cdot$ *y* # *ys*) **by** (*metis Cons-eq-appendI eq-Nil-appendI is-output.simps*(*2*) *shuffled.swap*)
have *shuffled* (*y* # *ys* $\cdot$ *?*⟨($a^{q \rightarrow p}$)⟩ # ε) (*y* # *?*⟨($a^{q \rightarrow p}$)⟩ # *ys*) **using** ‹*shuffled* (*ys* $\cdot$ *?*⟨($a^{q \rightarrow p}$)⟩ # ε) (*?*⟨($a^{q \rightarrow p}$)⟩ # ε $\cdot$ *ys*)› *shuffled-prepend-inductive* **by** *fastforce*
then show *?case* **by** (*meson last-step shuffled.trans*)
qed

lemma *fully-shuffled-w-prepend*:
assumes *xs* = *xs*$\downarrow_!$
shows *shuffled* (*w* @ *xs* @ [*?*⟨($a^{q \rightarrow p}$)⟩]) (*w* @ [*?*⟨($a^{q \rightarrow p}$)⟩] @ *xs*)
using *assms*
proof (*induct w*)
case *Nil*
then show *?case* **by** (*metis append-Nil fully-shuffled-gen*)
next
case (*Cons a w*)
then show *?case* **using** *shuffled-prepend-inductive* **by** *auto*
qed

lemma *shuffle-preserves-length*:
shuffled w w′ $\Longrightarrow$ *length w* = *length w*′
by (*induction rule*: *shuffled.induct*) *auto*

lemma *shuffled-lang-subset-lang* :
assumes *w* ∈ *L*
shows *valid-input-shuffles-of-w w* ⊆ *shuffled-lang L*
using *all-shuffles-def assms shuffled-lang-def* **by** *fastforce*

lemma *input-shuffle-implies-shuffled-lang* :
assumes *w* ∈ *L* **and** *w*′ ∈ *valid-input-shuffles-of-w w*
shows *w*′ ∈ *shuffled-lang L*
using *assms*(*1*,*2*) *shuffled-lang-subset-lang* **by** *auto*

lemma *shuffled-lang-not-empty* :
shows (*valid-input-shuffles-of-w w*) ≠ {}
using *shuffled.refl* **by** *auto*

lemma *valid-input-shuffles-of-lang* :
assumes *w* ∈ *L*
shows ∃ *w*′. (*w*′ $\sqcup\sqcup_?$ *w* ∧ *w*′ ∈ *shuffled-lang L*)
by (*metis assms input-shuffle-implies-shuffled-lang mem-Collect-eq shuffled.refl*)

lemma *valid-input-shuffle-partner* :
  assumes $\{\} \neq$ *valid-input-shuffles-of-w w*
  shows $\exists\, w'.\ w' \sqcup\sqcup_?\ w$
  using *assms* **by** *auto*

lemma *shuffle-id* :
  assumes $w \in L$
  shows $w \in$ *shuffled-lang L*
  using *assms* **by** (*simp add*: *input-shuffle-implies-shuffled-lang shuffled.refl*)

lemma *shuffled-prepend*:
  assumes $w'\ \sqcup\sqcup_?\ \ w$
  shows $a\ \#\ w'\ \ \sqcup\sqcup_?\ a\ \#\ w$
  using *assms*
proof (*induct rule*: *shuffled.induct*)
  case (*refl w*)
  then show *?case* **using** *shuffled.refl* **by** *blast*
next
  case (*swap a b w xs ys*)
  then show *?case* **by** (*metis append-Cons shuffled.swap*)
next
  case (*trans w w′ w″*)
  then show *?case* **using** *shuffled.trans* **by** *auto*
qed

lemma *fully-shuffled-implies-output-right* :
  assumes $xs = xs{\downarrow_?}$ **and** *is-output a*
  shows *shuffled* ([*a*] @ *xs*) (*xs* @ [*a*])
  using *assms*
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** (*simp add*: *shuffled.refl*)
next
  case (*Cons y ys*)
  then have ys @ $[a] \sqcup\sqcup_?\ (a\ \#\ \ ys)$
    by (*metis append-Cons append-eq-append-conv-if drop-eq-Nil2 filter.simps*(*2*) *impossible-Cons length-filter-le list.sel*(*3*))
  have *is-input y* **by** (*metis Cons.prems*(*1*) *filter-id-conv list.set-intros*(*1*))
  then have $y\ \#\ [a] \sqcup\sqcup_?\ (a\ \#\ \ [y])$ **using** *append.assoc append.right-neutral assms*(*2*) *same-append-eq shuffled.simps* **by** *fastforce*
  then have $y\ \#\ a\ \#\ ys\ \ \sqcup\sqcup_?\ a\ \#\ y\ \#\ ys$ **by** (*metis* ‹*is-input y*› *append-self-conv2 assms*(*2*) *shuffled.swap*)
  then have $y\ \#\ ys$ @ $[a]\ \ \sqcup\sqcup_?\ y\ \#\ a\ \#\ ys$ **using** ‹$ys \cdot a\ \#\ \varepsilon \sqcup\sqcup_?\ a\ \#\ ys$› *shuffled-prepend* **by** *auto*
  then show *?case* **using** ‹$y\ \#\ a\ \#\ ys \sqcup\sqcup_?\ a\ \#\ y\ \#\ ys$› *shuffled.trans* **by** *auto*
qed

lemma *shuffle-keeps-outputs-right-shuffled*:
  assumes *shuffled w w′* **and** *is-output* (*last w*)

shows *is-output* (*last* w')
using *assms*
proof (*induct rule*: *shuffled.induct*)
case (*refl w*)
then show *?case* **by** *simp*
next
case (*swap a b w xs ys*)
then show *?case* **by** *auto*
next
case (*trans w* w' w'')
then show *?case* **by** *simp*
qed

lemma *all-shuffles-rev*:
assumes $w' \in$ *all-shuffles w*
shows *shuffled w* w'
using *all-shuffles-def assms* **by** *auto*

lemma *shuffled-lang-rev*:
assumes $w \in$ *shuffled-lang L*
shows $\exists\ w'.\ w' \in L \wedge w \in$ *all-shuffles* w'
using *assms shuffled-lang-def* **by** *auto*

lemma *shuffled-lang-impl-valid-shuffle* :
assumes $v \in$ *shuffled-lang L*
shows $\exists v'.\ (\ v \sqcup\sqcup_{?}\ v' \wedge v' \in L)$
by (*meson all-shuffles-rev assms shuffled-lang-rev*)

lemma *fully-shuffled-valid-gen*:
assumes $(xs\ @\ [?\langle(a^{q \rightarrow p})\rangle]) \in L$ **and** $xs = xs\downarrow_{!}$
shows $([?\langle(a^{q \rightarrow p})\rangle]\ @\ xs) \sqcup\sqcup_{?} (xs\ @\ [?\langle(a^{q \rightarrow p})\rangle])$
by (*meson assms*(*2*) *fully-shuffled-gen*)

lemma *shuffling-possible-to-existing-shuffle*:
assumes *shuffling-possible w*
shows $\exists w'.$ *shuffled w* $w' \wedge w \neq w'$ **using** *assms shuffled.swap* **by** *fastforce*

5.1.1 rightmost shuffle related

lemma *rightmost-shuffle-exists*:
assumes $v \in$ *shuffled-lang L* **and** *shuffling-occurred v*
shows $\exists\ xs\ a\ b\ ys.\ v = (xs\ @\ b\ \#\ a\ \#\ ys) \wedge\ v \sqcup\sqcup_{?} (xs\ @\ a\ \#\ b\ \#\ ys)$
using *assms*(*2*) *shuffled.swap* **by** *blast*

lemma *length-index-bound*:
shows *Suc* (*length xs*) < *length* (*xs* @ *a* # *b* # *ys*)
proof −
have *length* (*xs* @ *a* # *b* # *ys*) = *length xs* + *length* (*a* # *b* # *ys*)

by *simp*
also have *length* (*a* # *b* # *ys*) = *2* + *length ys*
by *simp*
finally show *?thesis*
by *simp*
qed

lemma *shuffle-index-exists*:
assumes *shuffling-possible v*
shows ∃ *i*. *is-output* (*v*!*i*) ∧ *is-input* (*v*!(*Suc i*)) ∧ (*Suc i*) < *length v*
proof −
obtain *xs a b ys* **where** *is-output a* **and** *is-input b* **and** *v* = (*xs* @ *a* # *b* # *ys*) **using** *assms* **by** *auto*
have *t1*: *v*!(*length xs*) = *a* **by** (*simp add*: ‹*v* = *xs* · *a* # *b* # *ys*›)
then have *t2*: *v*!(*Suc* (*length xs*)) = *b* **by** (*metis Cons-nth-drop-Suc* ‹*v* = *xs* · *a* # *b* # *ys*› *append-eq-conv-conj drop-all linorder-le-less-linear list.distinct*(*1*) *list.inject*)
have *t3*: (*Suc* (*length xs*)) < *length v* **by** (*simp add*: ‹*v* = *xs* · *a* # *b* # *ys*›)
from *t1 t2 t3* **have** *is-output* (*v*!(*length xs*)) ∧ *is-input* (*v*!(*Suc* (*length xs*))) ∧ (*Suc* (*length xs*)) < *length v*
by (*simp add*: ‹*is-input b*› ‹*is-output a*›)
then show *?thesis* **by** *auto*
qed

lemma *rightmost-shuffle-index-exists*:
assumes *shuffling-possible v*
shows ∃ *i*. *is-output* (*v*!*i*) ∧ *is-input* (*v*!(*Suc i*)) ∧ (*Suc i*) < *length v* ∧ ¬ (*shuffling-possible* (*drop* (*Suc i*) *v*))
using *assms*
proof (*induct v*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a w*)
then show *?case*
proof (*cases shuffling-possible w*)
case *True*
then obtain *xs ys x y* **where** *w-decomp*: *is-output x* ∧ *is-input y* ∧ *w* = *xs* · *x* # *y* # *ys* **by** *blast*
then obtain *i* **where** *i-def*: *is-output* (*w* ! *i*) ∧
is-input (*w* ! *Suc i*) ∧
Suc i < |*w*| ∧ (∄*xs a b ys*. *is-output a* ∧ *is-input b* ∧ *drop* (*Suc i*) *w* = *xs* · *a* # *b* # *ys*)
using *Cons.hyps* **by** *blast*

have (*a* # *w*) = *a* # (*xs* · *x* # *y* # *ys*) **by** (*simp add*: *w-decomp*)
have *t1*: *is-output* ((*a* # *w*) ! (*Suc i*)) **by** (*simp add*: *i-def*)

have *t2*: *is-input* ((*a* # *w*) ! (*Suc* (*Suc* *i*))) **by** (*simp add*: *i-def*)
have *t3*: (*Suc* (*Suc* *i*)) < |(*a* # *w*)| **by** (*simp add*: *i-def*)
have *t4*: ¬ (*shuffling-possible* (*drop* (*Suc* (*Suc* *i*)) (*a*#*w*))) **by** (*metis drop-Suc-Cons i-def*)
show *?thesis* **using** *t1 t2 t3 t4* **by** *blast*
next
case *False*
then have ∃ *b ys*. (*a* # *w*) = (*a* # *b* # *ys*) ∧ *is-input b* ∧ *is-output a* **by** (*metis Cons.prems list.sel*(*1*,*3*) *self-append-conv2 tl-append2*)
then obtain *b ys* **where** (*a* # *w*) = (*a* # *b* # *ys*) ∧ *is-input b* ∧ *is-output a* **by** *blast*
then have ¬ *shuffling-possible* (*b*#*ys*) **using** *False* **by** *blast*
have *is-output* ((*a* # *w*) ! *0*) ∧
is-input ((*a* # *w*) ! *Suc 0*) ∧
Suc 0 < |(*a* # *w*)| **by** (*simp add*: ‹*a* # *w* = *a* # *b* # *ys* ∧ *is-input b* ∧ *is-output a*›)
then show *?thesis* **by** (*metis Cons-nth-drop-Suc False Suc-lessD drop0 list.inject*)
qed
qed

lemma *rightmost-shuffle-concrete*:
assumes *shuffling-possible v*
shows ∃ *xs a b ys*. *is-output a* ∧ *is-input b* ∧ *v* = (*xs* @ *a* # *b* # *ys*) ∧ ¬ (*shuffling-possible ys*)
using *assms*
proof (*induct v*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a w*)
then show *?case* **using** *Cons assms*
proof (*cases shuffling-possible w*)
case *True*
then have ∃*xs a b ys*. *is-output a* ∧ *is-input b* ∧ *w* = *xs* · *a* # *b* # *ys* **by** *blast*
then have ∃*xs a b ys*.
is-output a ∧
is-input b ∧ *w* = *xs* · *a* # *b* # *ys* ∧ (∄*xs a b ysa*. *is-output a* ∧ *is-input b* ∧ *ys* = *xs* · *a* # *b* # *ysa*) **using** *Cons* **by** *blast*
then obtain *xs ys x y* **where** *w-decomp*: *is-output x* ∧ *is-input y* ∧ *w* = *xs* · *x* # *y* # *ys* ∧ ¬ (*shuffling-possible ys*) **by** *blast*

have (*a* # *w*) = *a* # (*xs* · *x* # *y* # *ys*) **by** (*simp add*: *w-decomp*)
then have *is-output x* ∧ *is-input y* ∧ (*a*#*w*) = (*a*#*xs*) · *x* # *y* # *ys* ∧ ¬ (*shuffling-possible ys*)
using *w-decomp* **by** *auto*
then show *?thesis* **by** *blast*
next
case *False*
then have ∃ *b ys*. (*a* # *w*) = (*a* # *b* # *ys*) ∧ *is-input b* ∧ *is-output a* **by**

(*metis Cons.prems list.sel(1,3) self-append-conv2 tl-append2*)
then obtain *b ys* **where** $(a \# w) = (a \# b \# ys) \wedge$ *is-input b* $\wedge$ *is-output a* **by** *blast*
then have $\neg$ *shuffling-possible* $(b\#ys)$ **using** *False* **by** *blast*
then have *is-output a* $\wedge$ *is-input b* $\wedge$ $(a\#w) = [] \cdot a \# b \# ys \wedge \neg$ (*shuffling-possible ys*) **by** (*metis Cons-eq-appendI* ‹$a \# w = a \# b \# ys \wedge$ *is-input b* $\wedge$ *is-output a*› *append-self-conv2*)
then show *?thesis* **by** *blast*
qed
qed

lemma *rightmost-shuffle-is-shuffle*:
assumes *rightmost-shuffle v w*
shows $w \sqcup\!\sqcup_? v$
using *assms*
proof −
have *rightmost-shuffle v w* **using** *assms* **by** *simp*
then have $(\exists$ *xs a b ys*. *is-output a* $\wedge$ *is-input b* $\wedge v = (xs @ a \# b \# ys) \wedge (\neg$ *shuffling-possible ys*$) \wedge w = (xs @ b \# a \# ys))$ **by** *blast*
then obtain *xs a b ys* **where** *shuf-decomp*: *is-output a* $\wedge$ *is-input b* $\wedge v = (xs @ a \# b \# ys) \wedge (\neg$ *shuffling-possible ys*$) \wedge w = (xs @ b \# a \# ys)$ **by** *blast*
have $(xs @ b \# a \# ys) \sqcup\!\sqcup_? (xs @ a \# b \# ys)$ **by** (*simp add*: *shuf-decomp shuffled.swap*)
then show *?thesis* **by** (*simp add*: *shuf-decomp*)
qed

lemma *rightmost-shuffle-exists-2*:
assumes *shuffling-possible v*
shows $\exists$ *w*. *rightmost-shuffle v w*
using *assms*
proof −
have *shuffling-possible v* **using** *assms* **by** *blast*
then have $\exists$ *xs a b ys*. *is-output a* $\wedge$ *is-input b* $\wedge v = (xs @ a \# b \# ys) \wedge \neg$ (*shuffling-possible ys*) **using** *rightmost-shuffle-concrete*[*of v*] **by** *blast*
then obtain *xs a b ys* **where** *is-output a* $\wedge$ *is-input b* $\wedge v = (xs @ a \# b \# ys) \wedge (\neg$ *shuffling-possible ys*) **by** *blast*
then have *rightmost-shuffle v* $(xs @ b \# a \# ys)$ **by** *blast*
then show $\exists$ *w*. *rightmost-shuffle v w* **by** *blast*
qed

lemma *fully-shuffled-valid-w-prepend*:
assumes $(w @ [?\langle(a^{q \rightarrow p})\rangle] @ xs) \in L$ **and** $xs = xs\downarrow_!$
shows $(w @ [?\langle(a^{q \rightarrow p})\rangle] @ xs) \sqcup\!\sqcup_? (w @ xs @ [?\langle(a^{q \rightarrow p})\rangle])$
by (*meson assms*(*2*) *fully-shuffled-w-prepend*)

lemma *shuffled-keeps-send-order*:
assumes *shuffled v v'*
shows $v\downarrow_! = v'\downarrow_!$

using *assms*
proof (*induct*)
  case (*refl w*)
  then show *?case* **by** *simp*
next
  case (*swap a b w xs ys*)
  have *w-decomp*: $w{\downarrow_!} = xs{\downarrow_!} \cdot [a,b]{\downarrow_!}$ @ $ys{\downarrow_!}$ **by** (*simp add*: *swap.hyps*(*3*))
  have *pair-decomp*: $[a,b]{\downarrow_!} = [b,a]{\downarrow_!}$ **by** (*simp add*: *swap.hyps*(*2*))
  then show *?case* **by** (*simp add*: *w-decomp*)
next
  case (*trans* $w\ w'\ w''$)
  then show *?case* **by** *simp*
qed

lemma *shuffle-keeps-send-order*:
  assumes $v' \sqcup\sqcup_? v$
  shows $v{\downarrow_!} = v'{\downarrow_!}$
  by (*simp add*: *assms shuffled-keeps-send-order*)

lemma *shuffled-keeps-recv-order*:
  assumes *shuffled* $v\ v'$
  shows $v{\downarrow_?} = v'{\downarrow_?}$
  using *assms*
proof (*induct*)
  case (*refl w*)
  then show *?case* **by** *simp*
next
  case (*swap a b w xs ys*)
  have *w-decomp*: $w{\downarrow_?} = xs{\downarrow_?} \cdot [a,b]{\downarrow_?}$ @ $ys{\downarrow_?}$ **by** (*simp add*: *swap.hyps*(*3*))
  have *pair-decomp*: $[a,b]{\downarrow_?} = [b,a]{\downarrow_?}$ **by** (*simp add*: *swap.hyps*(*1*))
  then show *?case* **by** (*simp add*: *w-decomp*)
next
  case (*trans* $w\ w'\ w''$)
  then show *?case* **by** *simp*
qed

lemma *shuffle-keeps-recv-order*:
  assumes $v' \sqcup\sqcup_? v$
  shows $v{\downarrow_?} = v'{\downarrow_?}$
  by (*simp add*: *assms shuffled-keeps-recv-order*)

5.2 A Communicating Automaton

context *CommunicatingAutomaton* **begin**

lemma *ActionsOverMessages-rev*:
  assumes $a \in$ *ActionsOverMessages*
  shows *get-message* $a \in$ *Messages*
  using *ActionsOverMessages.simps assms* **by** *force*

```
lemma ActionsOverMessages-is-finite:
  shows finite ActionsOverMessages
  using message-alphabet Alphabet.finite-letters[of Messages]
  by (simp add: ActionsOverMessages-def ActionsOverMessagesp.simps)

lemma action-is-action-over-message:
  fixes s1 s2 :: 'state
    and a     :: ('information, 'peer) action
  assumes (s1, a, s2) ∈ Transitions
  shows a ∈ ActionsOverMessages
  using assms
proof (induct a)
  case (Output m)
  assume (s1, !⟨m⟩, s2) ∈ Transitions
  thus !⟨m⟩ ∈ ActionsOverMessages
    using well-formed-transition[of s1 !⟨m⟩ s2] AOMOutput[of m]
    by simp
next
  case (Input  m)
  assume (s1, ?⟨m⟩, s2) ∈ Transitions
  thus ?⟨m⟩ ∈ ActionsOverMessages
    using well-formed-transition[of s1 ?⟨m⟩ s2] AOMInput[of m]
    by simp
qed

lemma transition-set-is-finite:
  shows finite Transitions
proof −
  have Transitions ⊆ {(s1, a, s2). s1 ∈ States ∧ a ∈ ActionsOverMessages ∧ s2
∈ States}
    using well-formed-transition action-is-action-over-message
    by blast
  moreover have finite {(s1, a, s2). s1 ∈ States ∧ a ∈ ActionsOverMessages ∧
s2 ∈ States}
    using finite-states ActionsOverMessages-is-finite
    by simp
  ultimately show finite Transitions
    using finite-subset[of Transitions
        {(s1, a, s2). s1 ∈ States ∧ a ∈ ActionsOverMessages ∧ s2 ∈ States}]
    by simp
qed


lemma Actions-rev :
  assumes a ∈ Act
  shows ∃ s1 s2. (s1, a, s2) ∈ Transitions
  by (meson Actions.cases assms)
```

lemma *Act-is-subset-of-ActionsOverMessages*:
  shows *Act* ⊆ *ActionsOverMessages*
proof
  fix *a* :: (*'information*, *'peer*) *action*
  assume *a* ∈ *Act*
  then obtain *s1 s2* **where** (*s1*, *a*, *s2*) ∈ *Transitions*
    by (*auto simp add*: *Actions-def Actionsp.simps*)
  hence *get-message a* ∈ *Messages*
    using *well-formed-transition*[*of s1 a s2*]
    by *simp*
  thus *a* ∈ *ActionsOverMessages*
  proof (*induct a*)
    case (*Output m*)
    assume *get-message* (!⟨*m*⟩) ∈ *Messages*
    thus !⟨*m*⟩ ∈ *ActionsOverMessages*
      using *AOMOutput*[*of m*]
      by *simp*
  next
    case (*Input m*)
    assume *get-message* (?⟨*m*⟩) ∈ *Messages*
    thus ?⟨*m*⟩ ∈ *ActionsOverMessages*
      using *AOMInput*[*of m*]
      by *simp*
  qed
qed

lemma *Act-is-finite*:
  shows *finite Act*
  using *ActionsOverMessages-is-finite Act-is-subset-of-ActionsOverMessages*
    finite-subset[*of Act ActionsOverMessages*]
  by *simp*

lemma *ComunicationPartners-is-finite*:
  shows *finite CommunicationPartners*
proof −
  have *CommunicationPartners* ⊆ {*p*. ∃ *a*. *a* ∈ *ActionsOverMessages* ∧ *p* = *get-object a*}
    using *action-is-action-over-message*
    by (*auto simp add*: *CommunicationPartners-def CommunicationPartnersp.simps*)
  moreover have *finite* {*p*. ∃ *a*. *a* ∈ *ActionsOverMessages* ∧ *p* = *get-object a*}
    using *ActionsOverMessages-is-finite*
    by *simp*
  ultimately show *finite CommunicationPartners*
    using *finite-subset*[*of CommunicationPartners*
      {*p*. ∃ *a*. *a* ∈ *ActionsOverMessages* ∧ *p* = *get-object a*}]
    by *simp*
qed

lemma *SendingToPeers-rev*:

```
  fixes p :: 'peer
  assumes p ∈ SendingToPeers
  shows ∃ s1 a s2. (s1, a, s2) ∈ Transitions ∧ is-output a ∧ get-object a = p
  using assms
  by (induct, blast)

lemma SendingToPeers-is-subset-of-CommunicationPartners:
  shows SendingToPeers ⊆ CommunicationPartners
  using CommunicationPartners.intros SendingToPeersp.simps SendingToPeersp-SendingToPeers-eq
  by auto

lemma ReceivingFromPeers-rev:
  fixes p :: 'peer
  assumes p ∈ ReceivingFromPeers
  shows ∃ s1 a s2. (s1, a, s2) ∈ Transitions ∧ is-input a ∧ get-object a = p
  using assms
  by (induct, blast)

lemma ReceivingFromPeers-is-subset-of-CommunicationPartners:
  shows ReceivingFromPeers ⊆ CommunicationPartners
  using CommunicationPartners.intros ReceivingFromPeersp.simps
    ReceivingFromPeersp-ReceivingFromPeers-eq
  by auto
```

— this is to show that if p receives from no one, then there is no transition where p is the receiver

```
lemma empty-receiving-from-peers :
  fixes p :: 'peer
  assumes p ∉ ReceivingFromPeers and (s1, a, s2) ∈ Transitions and is-input a
  shows get-object a ≠ p
proof (rule ccontr)
  assume ¬ get-object a ≠ p
  then show False
  proof
    have get-object a = p using ‹¬ get-object a ≠ p› by auto
    moreover have p ∈ ReceivingFromPeers
      using ReceivingFromPeers.intros ‹¬ get-object a ≠ p› assms(2,3) by auto
    moreover have False
      using assms(1) calculation by auto
    ultimately show get-object a ≠ p  using assms(1) by auto
  qed
qed


lemma run-rev :
  assumes run s0 (a # w) (s1 # xs)
  shows run s1 w xs ∧ s0 −a→C s1
  by (smt (verit, best) assms list.discI list.inject run.simps)
```

lemma *run-rev2*:
assumes *run s0* (*w*) (*xs*) **and** *w* ≠ ε
shows ∃ *v vs a s1*. *run s1 v vs* ∧ *s0* −*a*→$_{\mathcal{C}}$ *s1* ∧ *w* = (*a* # *v*) ∧ *xs* = (*s1* # *vs*)
using *assms*(*1*,*2*) *run.cases* **by** *fastforce*

lemma *run-app* :
assumes *run s0* (*u* @ *v*) *xs* **and** *u* ≠ ε
shows ∃ *us vs*. *run s0 u us* ∧ *run* (*last us*) *v vs* ∧*xs* = *us* @ *vs*
using *assms*
proof (*induct u*@*v xs arbitrary*: *u v rule*: *run.induct*)
case (*REmpty2 s*)
then show *?case* **by** *simp*
next
case (*RComposed2 s1 w xs s0 a*)
then have *a* # *w* = *u* · *v* **by** *simp*
then have ∃ *u*′. *w* = *u*′ · *v* ∧ *u* = *a* # *u*′
by (*metis RComposed2.prems append-eq-Cons-conv*)
then obtain *u*′ **where** *w-decomp*: *w* = *u*′ @ *v* **and** *u-decomp*: *u* = *a* # *u*′ **by** *auto*
then have *run s1* (*u*′ @ *v*) *xs* **using** *RComposed2.hyps*(*1*) **by** *auto*
then show *?case*
proof (*cases u*′ = ε)
case *True*
then have *run s1 v xs* **using** *RComposed2.hyps*(*1*) *w-decomp* **by** *auto*
then have *run s0* [*a*] [*s1*]
by (*metis CommunicatingAutomaton.RComposed2 CommunicatingAutomaton.REmpty2 CommunicatingAutomaton-axioms RComposed2.hyps*(*3*))
then have *run s0* (*a* # *v*) (*s1* # *xs*) **by** (*simp add*: *RComposed2.hyps*(*3*) ‹*run s1 v xs*› *run.RComposed2*)
then show *?thesis* **using** *True* ‹*run s0* (*a* # ε) (*s1* # ε)› ‹*run s1 v xs*› *u-decomp* **by** *auto*
next
case *False*
then obtain *us*′ *vs* **where** *xs-decomp*: *run s1 u*′ *us*′ ∧ *run* (*last us*′) *v vs* ∧ *xs* = *us*′ · *vs*
using *RComposed2.hyps*(*2*) *w-decomp* **by** *blast*
then have *run s0* (*a* # *w*) (*s1* # *us*′ @ *vs*) **using** *RComposed2.hyps*(*1*,*3*) *run.RComposed2* **by** *auto*
then have *full-run-decomp*: *run s0* (*a* # *u*′ @ *v*) (*s1* # *us*′ @ *vs*) **by** (*simp add*: *w-decomp*)
then have *run s1 u*′ *us*′ **by** (*simp add*: *xs-decomp*)
then have *run s0* [*a*] [*s1*] **by** (*simp add*: *RComposed2.hyps*(*3*) *REmpty2 run.RComposed2*)
then have *run* (*last us*′) *v vs* **by** (*simp add*: *xs-decomp*)
then have *run s0 u* (*s1* # *us*′) **by** (*simp add*: *RComposed2.hyps*(*3*) *run.RComposed2 u-decomp xs-decomp*)
then have *run s0 u* (*s1* # *us*′) ∧ *run* (*last* (*s1* # *us*′)) *v vs* ∧ *s1* # *xs* = (*s1*

us') ▪ *vs*
      using *False run.cases xs-decomp* **by** *force*
    then show *?thesis* **by** *blast*
  qed
qed

lemma *run-second* :
  assumes *run s0* (*u* @ *v*) (*us*@*vs*) **and** $u \neq \varepsilon$ **and** *run s0 u us*
  shows *run* (*last us*) *v vs*
  using *assms*
proof (*induct u*@*v us*@*vs arbitrary*: *u v us vs rule*: *run.induct*)
  case (*REmpty2 s*)
  then show *?case* **by** *simp*
next
  case (*RComposed2 s1 w xs s0 a*)
  then show *?case* **by** (*smt* (*verit*) *append-eq-Cons-conv append-self-conv2 last-ConsL last-ConsR list.discI*
        list.inject run.simps)
qed

lemma *Traces-rev* :
  fixes *w* :: ('*information*, '*peer*) *action word*
  assumes $w \in$ *Traces*
  shows ∃ *xs*. *run initial w xs*
  using *assms*
  by (*induct*, *blast*)

— since all states are final, if u sqdot> v is valid then u must also be valid otherwise some transition for u is missing and hence u sqdot> v would be invalid
lemma *Traces-app* :
  assumes (*u* @ *v*) ∈ *Traces*
  shows *u* ∈ *Traces*
  by (*metis CommunicatingAutomaton.REmpty2 CommunicatingAutomaton-axioms Traces.cases*
      Traces.intros assms run-app)

lemma *Traces-second* :
  assumes (*u* @ *v*) ∈ *Traces* **and** $u \neq \varepsilon$
  shows ∃ *s0 us vs*. *run s0* (*u* @ *v*) (*us*@*vs*) ∧ *run* (*last us*) *v vs*
  using *Traces-rev assms*(*1*,*2*) *run-app* **by** *blast*

end

5.3 Network of Communicating Automata

context *NetworkOfCA*
begin

lemma *peer-trans-to-message-in-network*:
assumes $(s1, a, s2) \in \mathcal{R}(p)$
shows *get-message a* $\in \mathcal{M}$
by (*meson CommunicatingAutomaton.ActionsOverMessages-rev CommunicatingAutomaton.action-is-action-over-message*
assms automaton-of-peer)

5.4 helpful conclusions about language/ runs / etc. in concrete cases and peer runs

lemma *empty-receiving-from-peers2* :
fixes p :: *'peer*
assumes $p \notin$ *ReceivingFromPeers* **and** $(s1, a, s2) \in \mathcal{R}(p)$ **and** *is-input a*
shows *get-object a* $\neq p$
proof (*rule ccontr*)
assume $\neg$ *get-object a* $\neq p$
then show *False*
proof
have *get-object a* $= p$ **using** ‹$\neg$ *get-object a* $\neq p$› **by** *auto*
moreover have *False*
by (*metis CommunicatingAutomaton.well-formed-transition* ‹$\neg$ *get-object a* $\neq p$› *assms*(*2*)
automaton-of-peer)
ultimately show *get-object a* $\neq p$ **using** *assms*(*1*) **by** *auto*
qed
qed

lemma *empty-receiving-from-peers3* :
fixes p :: *'peer*
assumes $\mathcal{P}_?(p) = \{\}$ **and** $(s1, a, s2) \in \mathcal{R}(p)$ **and** *is-input a*
shows *get-object a* $\neq p$
proof (*rule ccontr*)
assume $\neg$ *get-object a* $\neq p$
then show *False*
proof
have *get-object a* $= p$ **using** ‹$\neg$ *get-object a* $\neq p$› **by** *auto*
moreover have *False*
by (*metis CommunicatingAutomaton.well-formed-transition* ‹$\neg$ *get-object a* $\neq p$› *assms*(*2*)
automaton-of-peer)
ultimately show *get-object a* $\neq p$ **using** *assms*(*1*) **by** *auto*
qed
qed

lemma *empty-receiving-from-peers4* :
fixes p :: *'peer*
assumes $\mathcal{P}_?(p) = \{\}$ **and** $(s1, a, s2) \in \mathcal{R}(p)$

shows *is-output a*
by (*metis CommunicatingAutomaton.ReceivingFromPeers.intros assms*(*1*,*2*) *automaton-of-peer empty-iff*)

lemma *no-input-trans-root* :
fixes p :: *'peer*
assumes *is-input a* **and** $\mathcal{P}_?(p) = \{\}$
shows $(s1,\ a,\ s2) \notin \mathcal{R}(p)$
using *assms*(*1*,*2*) *empty-receiving-from-peers4* **by** *auto*

lemma *act-in-lang-means-trans-exists* :
fixes p :: *'peer*
assumes $[a] \in \mathcal{L}(p)$
shows $\exists\, s1\ s2.\ (s1,\ a,\ s2) \in \mathcal{R}(p)$
by (*smt* (*verit*) *CommunicatingAutomaton.Traces-rev CommunicatingAutomaton.run.cases assms automaton-of-peer list.distinct*(*1*) *list.inject*)

lemma *act-not-in-lang-no-trans* :
fixes p :: *'peer*
assumes $\forall\, s1\ s2.\ (s1,\ a,\ s2) \notin \mathcal{R}(p)$
shows $[a] \notin \mathcal{L}(p)$
using *act-in-lang-means-trans-exists assms* **by** *auto*

lemma *no-input-trans-no-word-in-lang* :
fixes p :: *'peer*
assumes $(a\ \#\ w) \in \mathcal{L}(p)$
shows $\exists\, s1\ s2.\ (s1,\ a,\ s2) \in \mathcal{R}(p)$
by (*smt* (*verit*, *ccfv-SIG*) *CommunicatingAutomaton.Traces-rev CommunicatingAutomaton.run.cases assms automaton-of-peer list.distinct*(*1*) *list.inject*)

lemma *no-word-no-trans* :
fixes p :: *'peer*
assumes $\forall\, s1\ s2.\ (s1,\ a,\ s2) \notin \mathcal{R}(p)$
shows $(a\ \#\ w) \notin \mathcal{L}(p)$
using *assms no-input-trans-no-word-in-lang* **by** *blast*

lemma *root-head-is-output* :
fixes p :: *'peer*
assumes $\mathcal{P}_?(p) = \{\}$ **and** $(a\ \#\ w)\ \in \mathcal{L}(p)$
shows *is-output a*
using *assms*(*1*,*2*) *no-input-trans-root no-word-no-trans* **by** *blast*

lemma *root-head-is-not-input* :
fixes p :: *'peer*
assumes $\mathcal{P}_?(p) = \{\}$ **and** *is-input a*
shows $(a\ \#\ w)\ \notin \mathcal{L}(p)$
using *assms*(*1*,*2*) *root-head-is-output* **by** *auto*

lemma *eps-always-in-lang* :
fixes p :: $'peer$
assumes $\mathcal{L}(p) \neq \{\}$
shows $\varepsilon \in \mathcal{L}(p)$
by (*meson CommunicatingAutomaton.Traces.simps CommunicatingAutomaton.run.simps automaton-of-peer*)

lemma *no-recvs-no-input-trans* :
fixes p :: $'peer$
assumes $\mathcal{P}_?(p) = \{\}$
shows $\forall$ *s1 a s2*. (*is-input a* $\longrightarrow$ (*s1*, *a*, *s2*) $\notin \mathcal{R}(p)$)
by (*simp add*: *assms no-input-trans-root*)

lemma *no-input-trans-no-recvs* :
fixes p :: $'peer$
assumes $\forall$ *s1 a s2*. (*is-input a* $\longrightarrow$ (*s1*, *a*, *s2*) $\notin \mathcal{R}(p)$)
shows $\mathcal{P}_?(p) = \{\}$
by (*meson CommunicatingAutomaton.ReceivingFromPeers.simps assms automaton-of-peer subsetI subset-empty*)

lemma *Lang-app* :
assumes (u @ v) $\in \mathcal{L}(p)$
shows $u \in \mathcal{L}(p)$
by (*meson CommunicatingAutomaton.Traces-app assms automaton-of-peer*)

lemma *lang-implies-run*:
assumes $w \in \mathcal{L}(p)$
shows $\exists$ *xs*. *CommunicatingAutomaton.run* ($\mathcal{R}$ p) ($\mathcal{I}$ p) *w xs*
by (*meson CommunicatingAutomaton.Traces.simps assms automaton-of-peer*)

lemma *lang-implies-prepend-run* :
assumes (a # w) $\in \mathcal{L}(p)$
shows $\exists$ *xs s1*. *CommunicatingAutomaton.run* ($\mathcal{R}$ p) (*s1*) *w xs* $\wedge$ *CommunicatingAutomaton.run* ($\mathcal{R}$ p) ($\mathcal{I}$ p) [a] [*s1*]
by (*smt* (*verit*) *CommunicatingAutomaton.RComposed2 CommunicatingAutomaton.REmpty2*
CommunicatingAutomaton.run.cases assms automaton-of-peer concat.simps(*1*) *list.distinct*(*1*)
list.inject lang-implies-run)

lemma *trans-to-edge* :
assumes (*s1*, *a*, *s2*) $\in \mathcal{R}(p)$
shows *get-message a* $\in \mathcal{M}$
by (*meson CommunicatingAutomaton.well-formed-transition assms automaton-of-peer*)

lemma *valid-message-to-valid-act* :
assumes *get-message a* $\in \mathcal{M}$
shows $\exists$ *i p q*. $i^{p \rightarrow q} \in \mathcal{M} \wedge (i^{p \rightarrow q})$ = *get-message a*

by (*metis assms message.exhaust*)

lemma *input-message-to-act* :
assumes *get-message a* $\in \mathcal{M}$ **and** *is-input a* **and** *get-actor a* = *p*
shows $\exists$ *i q*. $i^{q \rightarrow p} \in \mathcal{M} \wedge (i^{q \rightarrow p})$ = *get-message a*
by (*metis action.exhaust assms*(*1*,*2*,*3*) *get-actor.simps*(*2*) *get-message.simps*(*2*) *get-receiver.simps is-output.simps*(*1*)
valid-message-to-valid-act)

lemma *output-message-to-act* :
assumes *get-message a* $\in \mathcal{M}$ **and** *is-output a* **and** *get-actor a* = *p*
shows $\exists$ *i q*. $i^{p \rightarrow q} \in \mathcal{M} \wedge (i^{p \rightarrow q})$ = *get-message a*
by (*metis action.exhaust assms*(*1*,*2*,*3*) *get-actor.simps*(*1*) *get-message.simps*(*1*) *get-sender.simps is-output.simps*(*2*)
valid-message-to-valid-act)

lemma *input-message-to-act-both-known* :
assumes *get-message a* $\in \mathcal{M}$ **and** *is-input a* **and** *get-actor a* = *p* **and** *get-object a* = *q*
shows $\exists$ *i*. $i^{q \rightarrow p} \in \mathcal{M} \wedge (i^{q \rightarrow p})$ = *get-message a*
by (*metis action.exhaust assms*(*1*,*2*,*3*,*4*) *get-message.simps*(*2*) *get-object.simps*(*2*) *get-sender.simps*
input-message-to-act is-output.simps(*1*))

lemma *output-message-to-act-both-known* :
assumes *get-message a* $\in \mathcal{M}$ **and** *is-output a* **and** *get-actor a* = *p* **and** *get-object a* = *q*
shows $\exists$ *i*. $i^{p \rightarrow q} \in \mathcal{M} \wedge (i^{p \rightarrow q})$ = *get-message a*
by (*metis action.exhaust assms*(*1*,*2*,*3*,*4*) *get-message.simps*(*1*) *get-object.simps*(*1*) *get-receiver.simps*
is-output.simps(*2*) *output-message-to-act*)

lemma *trans-to-act-in-lang* :
fixes *p* :: *'peer*
assumes $(\mathcal{I}\ p,\ a,\ s2) \in \mathcal{R}(p)$
shows $[a] \in \mathcal{L}(p)$
proof –
have *CommunicatingAutomaton.run* $(\mathcal{R}\ p)$ $(\mathcal{I}\ p)$ $[a]$ $[s2]$ **by** (*meson CommunicatingAutomaton.run.simps assms automaton-of-peer concat.simps*(*1*))
then show *?thesis* **by** (*meson CommunicatingAutomaton.Traces.intros automaton-of-peer*)
qed

lemma *lang-implies-run-alt* :
assumes $w \in \mathcal{L}(p)$
shows $\exists\, s2.\ (\mathcal{I}\ p)\ {-w\rightarrow^*}p\ s2$
using *assms lang-implies-run* **by** *blast*

lemma *Lang-app-both* :
  assumes $(u @ v) \in \mathcal{L}(p)$
  shows $\exists\, s2\ s3.\ (\mathcal{I}\ p)\ {-u\rightarrow^*} p\ s2 \wedge s2\ {-v\rightarrow^*} p\ s3$
  by (*metis CommunicatingAutomaton.initial-state CommunicatingAutomaton.run-app assms*
      automaton-of-peer lang-implies-run self-append-conv2)

lemma *lang-implies-trans* :
  assumes $s1\ {-[a]\rightarrow^*} p\ s2$
  shows $s1\ {-a\rightarrow_{\mathcal{C}}} p\ s2$
  by (*smt* (*verit*, *best*) *CommunicatingAutomaton.run.cases assms automaton-of-peer last.simps*
      list.distinct(*1*) *list.inject*)

lemma *Lang-last-act-app* :
  assumes $(u @ [a]) \in \mathcal{L}(p)$
  shows $\exists\, s1\ s2.\ s1\ {-a\rightarrow_{\mathcal{C}}} p\ s2$
  by (*meson Lang-app-both assms lang-implies-trans*)

lemma *Lang-last-act-trans*:
  assumes $(u @ [a]) \in \mathcal{L}(p)$
  shows $\exists\, s1\ s2.\ (s1,\ a,\ s2) \in \mathcal{R}\ p$
  using *Lang-last-act-app assms* **by** *auto*

lemma *act-in-word-has-trans*:
  assumes $w \in \mathcal{L}(p)$ **and** $a \in set\ w$
  shows $\exists\, s1\ s2.\ (s1,\ a,\ s2) \in \mathcal{R}\ p$
proof −
  have $\exists\ xs\ ys.\ (xs @ [a] @ ys) = w$ **by** (*metis Cons-eq-appendI append-self-conv2 assms*(*2*) *in-set-conv-decomp-first*)
  then obtain *xs ys* **where** $(xs @ [a] @ ys) = w$ **by** *blast*
  then have $(xs @ [a] @ ys) \in \mathcal{L}(p)$ **by** (*simp add*: *assms*(*1*))
  then have $(xs @ [a]) \in \mathcal{L}(p)$ **by** (*metis Lang-app append-assoc*)
  then show *?thesis* **by** (*simp add*: *Lang-last-act-trans*)
qed

lemma *recv-proj-w-prepend-is-in-w*:
  assumes $(w{\downarrow_?}) = (x \# xs)$ **and** $w \in \mathcal{L}(p)$
  shows $\exists\ ys\ zs.\ w = ys @ [x] @ zs$
  using *assms*
proof (*induct length* $(w{\downarrow_?})$ *arbitrary*: *w x xs*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  then show *?case* **by** (*metis Cons-eq-filterD append-Cons append-Nil*)
qed

lemma *recv-proj-w-prepend-has-trans*:
  assumes $(w\downarrow_?) = (x \# xs)$ **and** $w \in \mathcal{L}(p)$
  shows $\exists$ *s1 s2*. (*s1*, *x*, *s2*) $\in \mathcal{R}$ *p*
  using *assms*
proof (*induct length* $(w\downarrow_?)$ *arbitrary*: *w x xs*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  then obtain *ys zs* **where** *w-def*: *w* = *ys* @ [*x*] @ *zs* **using** *recv-proj-w-prepend-is-in-w*
by *blast*
  then have (*ys* @ [*x*] @ *zs*) $\in \mathcal{L}(p)$ **using** *Suc.prems*(*2*) **by** *blast*
  then have (*ys* @ [*x*]) $\in \mathcal{L}(p)$ **by** (*metis Lang-app append-assoc*)
  then have $\exists$ *s1 s2*. (*s1*, *x*, *s2*) $\in \mathcal{R}$ *p* **using** *Lang-app-both lang-implies-trans*
by *blast*
  then show *?case* **by** *simp*
qed

lemma *send-proj-w-prepend-is-in-w*:
  assumes $(w\downarrow_!) = (x \# xs)$ **and** $w \in \mathcal{L}(p)$
  shows $\exists$ *ys zs*. *w* = *ys* @ [*x*] @ *zs*
  using *assms*
proof (*induct length* $(w\downarrow_!)$ *arbitrary*: *w x xs*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  then show *?case* **by** (*metis Cons-eq-filterD append-Cons append-Nil*)
qed

lemma *send-proj-w-prepend-has-trans*:
  assumes $(w\downarrow_!) = (x \# xs)$ **and** $w \in \mathcal{L}(p)$
  shows $\exists$ *s1 s2*. (*s1*, *x*, *s2*) $\in \mathcal{R}$ *p*
  using *assms*
proof (*induct length* $(w\downarrow_!)$ *arbitrary*: *w x xs*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  then obtain *ys zs* **where** *w-def*: *w* = *ys* @ [*x*] @ *zs* **using** *send-proj-w-prepend-is-in-w*
by *blast*
  then have (*ys* @ [*x*] @ *zs*) $\in \mathcal{L}(p)$ **using** *Suc.prems*(*2*) **by** *blast*
  then have (*ys* @ [*x*]) $\in \mathcal{L}(p)$ **by** (*metis Lang-app append-assoc*)
  then have $\exists$ *s1 s2*. (*s1*, *x*, *s2*) $\in \mathcal{R}$ *p* **using** *Lang-app-both lang-implies-trans*
by *blast*
  then show *?case* **by** *simp*
qed

lemma *no-inputs-implies-only-sends* :

assumes $\mathcal{P}_?(p) = \{\}$
shows $\forall w.\ w \in \mathcal{L}(p) \longrightarrow (w = w{\downarrow_!})$
using *assms*
proof *auto*
fix w
show $\mathcal{P}_?\ p = \{\} \Longrightarrow w \in \mathcal{L}\ p \Longrightarrow w = w{\downarrow_!}$
proof –
assume $w \in \mathcal{L}\ p$
then show $w = w{\downarrow_!}$
proof (*induct length w arbitrary*: w)
case *0*
then show *?case* **by** *simp*
next
case (*Suc x*)
then obtain $v\ a$ **where** *w-def*: $w = v\ @\ [a]$ **and** *v-len*: *length* $v = x$ **by** (*metis length-Suc-conv-rev*)
then have $v \in \mathcal{L}\ p$ **using** *Lang-app Suc.prems* **by** *blast*
then have $v = v{\downarrow_!}$ **by** (*simp add*: *Suc.hyps*(*1*) *v-len*)
then obtain *s2 s3* **where** *v-run*: $(\mathcal{I}\ p)\ -v\rightarrow^* p\ s2$ **and** *a-run* : $s2\ -[a]\rightarrow^* p\ s3$
using *Lang-app-both Suc.prems w-def* **by** *blast*
then have $\forall\, s1\ s2.\ (s1,\ a,\ s2) \in \mathcal{R}(p) \longrightarrow$ *is-output* a **using** *assms no-recvs-no-input-trans* **by** *blast*
then have $(s2,\ a,\ s3) \in \mathcal{R}(p)$ **using** *a-run lang-implies-trans* **by** *force*
then have *is-output* a **by** (*simp add*: ‹$\forall\, s1\ s2.\ s1\ -a\rightarrow_{\mathcal{C}} p\ s2 \longrightarrow$ *is-output* a›)
then show *?case* **using** ‹$v = v{\downarrow_!}$› *w-def* **by** *auto*
qed
qed
qed

lemma *no-inputs-implies-only-sends-alt* :
assumes $\mathcal{P}_?(p) = \{\}$ **and** $w \in \mathcal{L}(p)$
shows $w = w{\downarrow_!}$
using *assms*(*1*,*2*) *no-inputs-implies-only-sends* **by** *auto*

lemma *no-inputs-implies-send-lang* :
assumes $\mathcal{P}_?(p) = \{\}$
shows $\mathcal{L}(p) = (\mathcal{L}(p))|_!$
proof
show $\mathcal{L}\ p \subseteq (\mathcal{L}\ p)|_!$ **using** *assms no-inputs-implies-only-sends-alt* **by** *auto*
next
show $(\mathcal{L}\ p)|_! \subseteq \mathcal{L}\ p$ **using** *assms no-inputs-implies-only-sends-alt* **by** *auto*
qed

5.5 Synchronous System

lemma *initial-configuration-is-synchronous-configuration*:
shows *is-sync-config* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$

unfolding *is-sync-config-def*
proof *clarify*
fix p :: $'peer$
show $\mathcal{C}_{\mathcal{I}\mathbf{0}}(p) \in \mathcal{S}(p)$
using *automaton-of-peer*[*of p*]
CommunicatingAutomaton.initial-state[*of* p $\mathcal{S}$ p $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ p $\mathcal{M}$ $\mathcal{R}$ p]
by *simp*
qed

lemma *sync-step-rev*:
fixes *C1 C2* :: $'peer \Rightarrow 'state$
and a :: ($'information$, $'peer$) *action*
assumes *C1* $-\langle a, \mathbf{0}\rangle\rightarrow$ *C2*
shows *is-sync-config C1* **and** *is-sync-config C2* **and** $\exists\, i\ p\ q.\ a = !\langle(i^{p \rightarrow q})\rangle$
and *get-actor a* $\neq$ *get-object a* **and** *C1* (*get-actor a*) $-a\rightarrow_{\mathcal{C}}$(*get-actor a*) (*C2* (*get-actor a*))
and $\exists\, m.\ a = !\langle m\rangle \wedge$ *C1* (*get-object a*) $-(?\langle m\rangle)\rightarrow_{\mathcal{C}}$(*get-object a*) (*C2* (*get-object a*))
and $\forall\, x.\ x \notin \{$*get-actor a*, *get-object a*$\} \longrightarrow$ *C1*(x) = *C2*(x)
using *assms*
proof *induct*
case (*SynchStep C1 a i p q C2*)
assume *A1*: *is-sync-config C1*
thus *is-sync-config C1* **.**
assume *A2*: $a = !\langle(i^{p \rightarrow q})\rangle$
thus $\exists\, i\ p\ q.\ a = !\langle(i^{p \rightarrow q})\rangle$
by *blast*
assume *A3*: *C1 p* $-(!\langle(i^{p \rightarrow q})\rangle)\rightarrow_{\mathcal{C}} p$ (*C2 p*)
with *A2* **show** *C1* (*get-actor a*) $-a\rightarrow_{\mathcal{C}}$(*get-actor a*) (*C2* (*get-actor a*))
by *simp*
have *A4*: *CommunicatingAutomaton p* ($\mathcal{S}$ p) ($\mathcal{I}$ p) $\mathcal{M}$ ($\mathcal{R}$ p)
using *automaton-of-peer*[*of p*]
by *simp*
with *A2 A3* **show** *get-actor a* $\neq$ *get-object a*
using *CommunicatingAutomaton.well-formed-transition*[*of* p $\mathcal{S}$ p $\mathcal{I}$ p $\mathcal{M}$ $\mathcal{R}$ p *C1 p a C2 p*]
by *auto*
assume *A5*: *C1 q* $-(?\langle(i^{p \rightarrow q})\rangle)\rightarrow_{\mathcal{C}} q$ (*C2 q*)
with *A2* **show** $\exists\, m.\ a = !\langle m\rangle \wedge$ *C1* (*get-object a*) $-(?\langle m\rangle)\rightarrow_{\mathcal{C}}$(*get-object a*) (*C2* (*get-object a*))
by *auto*
assume *A6*: $\forall\, x.\ x \notin \{p, q\} \longrightarrow$ *C1 x* = *C2 x*
with *A2* **show** $\forall\, x.\ x \notin \{$*get-actor a*, *get-object a*$\} \longrightarrow$ *C1*(x) = *C2*(x)
by *simp*
show *is-sync-config C2*
unfolding *is-sync-config-def*
proof *clarify*
fix x :: $'peer$

show $C2(x) \in \mathcal{S}(x)$
proof (*cases* $x = p$)
assume $x = p$
with *A3 A4* **show** $C2(x) \in \mathcal{S}(x)$
using *CommunicatingAutomaton.well-formed-transition*[*of* p $\mathcal{S}$ p $\mathcal{I}$ p $\mathcal{M}$ $\mathcal{R}$ p *C1* p
$!\langle(i^{p \rightarrow q})\rangle$ *C2* p]
by *simp*
next
assume B: $x \neq p$
show $C2(x) \in \mathcal{S}(x)$
proof (*cases* $x = q$)
assume $x = q$
with *A5* **show** $C2(x) \in \mathcal{S}(x)$
using *automaton-of-peer*[*of* q]
CommunicatingAutomaton.well-formed-transition[*of* q $\mathcal{S}$ q $\mathcal{I}$ q $\mathcal{M}$ $\mathcal{R}$ q *C1* q
$?\langle(i^{p \rightarrow q})\rangle$ *C2* q]
by *simp*
next
assume $x \neq q$
with *A1 A6 B* **show** $C2(x) \in \mathcal{S}(x)$
unfolding *is-sync-config-def*
by (*metis empty-iff insertE*)
qed
qed
qed
qed

lemma *sync-step-output-rev*:
fixes *C1 C2* :: *'peer* $\Rightarrow$ *'state*
and i :: *'information*
and p q :: *'peer*
assumes *C1* $-\langle !\langle(i^{p \rightarrow q})\rangle, \mathbf{0}\rangle\rightarrow$ *C2*
shows *is-sync-config C1* **and** *is-sync-config C2* **and** $p \neq q$ **and** *C1* p $-(!\langle(i^{p \rightarrow q})\rangle)\rightarrow_{\mathcal{C}} p$ (*C2* p)
and *C1* q $-(?\langle(i^{p \rightarrow q})\rangle)\rightarrow_{\mathcal{C}} q$ (*C2* q) **and** $\forall x.\ x \notin \{p, q\} \longrightarrow C1(x) = C2(x)$
using *assms sync-step-rev*[*of C1* $!\langle(i^{p \rightarrow q})\rangle$ *C2*]
by *simp-all*

lemma *sync-run-rev* :
assumes *sync-run C0* (w•[a]) (*xc*@[C])
shows *sync-run C0 w xc* $\wedge$ *last* (*C0*#*xc*) $-\langle a, \mathbf{0}\rangle\rightarrow$ C
using *NetworkOfCA.sync-run.cases NetworkOfCA-axioms assms* **by** *blast*

lemma *run-produces-synchronous-configurations*:
fixes C C' :: *'peer* $\Rightarrow$ *'state*
and w :: (*'information*, *'peer*) *action word*

and *xc* :: (*'peer* ⇒ *'state*) *list*
assumes *sync-run C w xc*
and $C' \in$ *set xc*
shows *is-sync-config C'*
using *assms*
proof *induct*
case (*SREmpty C*)
assume $C' \in$ *set* []
hence *False*
by *simp*
thus *is-sync-config C'*
by *simp*
next
case (*SRComposed C0 w xc a C*)
assume *A1*: $C' \in$ *set xc* $\Longrightarrow$ *is-sync-config C'* **and** *A2*: *last* (*C0#xc*) $-\langle a, \mathbf{0}\rangle\rightarrow$ *C*
and *A3*: $C' \in$ *set* (*xc*·[*C*])
show *is-sync-config C'*
proof (*cases* $C = C'$)
assume $C = C'$
with *A2* **show** *is-sync-config C'*
using *sync-step-rev*(*2*)[*of last* (*C0#xc*) *a C*]
by *simp*
next
assume $C \neq C'$
with *A1 A3* **show** *is-sync-config C'*
by *simp*
qed
qed

lemma *run-produces-no-inputs*:
fixes *C C'* :: *'peer* ⇒ *'state*
and *w* :: (*'information*, *'peer*) *action word*
and *xc* :: (*'peer* ⇒ *'state*) *list*
assumes *sync-run C w xc*
shows $w{\downarrow_!} = w$ **and** $w{\downarrow_?} = \varepsilon$
using *assms*
proof *induct*
case (*SREmpty C*)
show $\varepsilon{\downarrow_!} = \varepsilon$ **and** $\varepsilon{\downarrow_?} = \varepsilon$
by *simp-all*
next
case (*SRComposed C0 w xc a C*)
assume $w{\downarrow_!} = w$
moreover assume *last* (*C0#xc*) $-\langle a, \mathbf{0}\rangle\rightarrow$ *C*
hence *A*: *is-output a*
using *sync-step-rev*(*3*)[*of last* (*C0#xc*) *a C*]
by *auto*
ultimately show $(w \cdot [a]){\downarrow_!} = w \cdot [a]$

by *simp*
assume $w{\downarrow_?} = \varepsilon$
with *A* **show** $(w \cdot [a]){\downarrow_?} = \varepsilon$
by *simp*
qed

lemma *SyncTraces-rev* :
assumes $w \in \mathcal{T}_{\mathbf{0}}$
shows $\exists\, xc.\ \mathit{sync\text{-}run}\ \mathcal{C}_{\mathcal{I}\mathbf{0}}\ w\ xc$
using *SyncTraces.simps assms* **by** *auto*

lemma *no-inputs-in-synchronous-communication*:
shows $\mathcal{L}_{\mathbf{0}}{\downharpoonright_!} = \mathcal{L}_{\mathbf{0}}$ **and** $\mathcal{L}_{\mathbf{0}}{\downharpoonright_?} \subseteq \{\varepsilon\}$
proof −
have $\forall\, w \in \mathcal{L}_{\mathbf{0}}.\ w{\downarrow_!} = w$
using *SyncTraces.simps run-produces-no-inputs*(*1*)
by *blast*
thus $\mathcal{L}_{\mathbf{0}}{\downharpoonright_!} = \mathcal{L}_{\mathbf{0}}$
by *force*
have $\forall\, w \in \mathcal{L}_{\mathbf{0}}.\ w{\downarrow_?} = \varepsilon$
using *SyncTraces.simps run-produces-no-inputs*(*2*)
by *blast*
thus $\mathcal{L}_{\mathbf{0}}{\downharpoonright_?} \subseteq \{\varepsilon\}$
by *auto*
qed

lemma *sync-send-step-to-recv-step*:
assumes $C1\ {-}\langle !\langle (i^{p \rightarrow q}) \rangle, \mathbf{0} \rangle{\rightarrow}\ C2$
shows $C1\ q\ {-}(?\langle (i^{p \rightarrow q}) \rangle){\rightarrow_{\mathcal{C}}} q\ (C2\ q)$
using *assms sync-step-output-rev*(*5*) **by** *auto*

lemma *act-in-sync-word-to-sync-step*:
assumes $w \in \mathcal{L}_{\mathbf{0}}$ **and** $a \in \mathit{set}\ w$
shows $\exists\ C1\ C2.\ C1\ {-}\langle a, \mathbf{0} \rangle{\rightarrow}\ C2$
sorry

lemma *act-in-sync-word-to-matching-peer-steps*:
assumes $w \in \mathcal{L}_{\mathbf{0}}$ **and** $(!\langle (i^{p \rightarrow q}) \rangle) \in \mathit{set}\ w$
shows $\exists\, C1\ C2.\ C1\ p\ {-}(!\langle (i^{p \rightarrow q}) \rangle){\rightarrow_{\mathcal{C}}} p\ (C2\ p) \wedge C1\ q\ {-}(?\langle (i^{p \rightarrow q}) \rangle){\rightarrow_{\mathcal{C}}} q\ (C2$
$q)$
using *act-in-sync-word-to-sync-step assms*(*1*,*2*) *sync-send-step-to-recv-step sync-step-output-rev*(*4*)
by *blast*

lemma *sync-lang-app*:
assumes $(u\ @\ v) \in \mathcal{L}_{\mathbf{0}}$
shows $u \in \mathcal{L}_{\mathbf{0}}$
sorry

lemma *sync-lang-sends-app*:

assumes $(u@v)\downarrow_! \in \mathcal{L}_\mathbf{0}$
shows $u\downarrow_! \in \mathcal{L}_\mathbf{0}$
by (*metis assms filter-append sync-lang-app*)

lemma *sync-run-word-configs-len-eq*:
assumes *sync-run C0 w xc*
shows *length w* = *length xc*
using *assms* **proof** (*induct rule*: *sync-run.induct*)
case (*SREmpty C*)
then show *?case* **by** *simp*
next
case (*SRComposed C0 w xc a C*)
then show *?case* **by** *simp*
qed

5.6 Mailbox System

5.6.1 Semantics and Language

lemma *initial-mbox-alt* :
shows $(\forall p.\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ p = (\mathcal{C}_{\mathcal{I}\mathbf{0}}\ p,\ \varepsilon))$
by *simp*

lemma *initial-configuration-is-mailbox-configuration*:
shows *is-mbox-config* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$
unfolding *is-mbox-config-def*
proof *clarify*
fix *p* :: *'peer*
show *fst* $(\mathcal{C}_{\mathcal{I}\mathbf{0}}\ p,\ \varepsilon) \in \mathcal{S}\ p \wedge$ *snd* $(\mathcal{C}_{\mathcal{I}\mathbf{0}}\ p,\ \varepsilon) \in \mathcal{M}^*$
using *automaton-of-peer*[*of p*] *message-alphabet Alphabet.EmptyWord*[*of* $\mathcal{M}$]
CommunicatingAutomaton.initial-state[*of p* $\mathcal{S}$ *p* $\mathcal{I}$ *p* $\mathcal{M}$ $\mathcal{R}$ *p*]
by *simp*
qed

lemma *initial-configuration-is-stable*:
shows *is-stable* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$
unfolding *is-stable-def* **using** *initial-configuration-is-mailbox-configuration*
by *simp*

lemma *sync-config-to-mbox* :
assumes *is-sync-config C*
shows $\exists\, C'.$ *is-mbox-config* $C' \wedge C' = (\lambda p.\ (C\ p,\ \varepsilon))$
using *assms initial-configuration-is-mailbox-configuration is-mbox-config-def*
is-sync-config-def **by** *auto*

lemma *mbox-step-rev*:
fixes *C1 C2* :: *'peer* ⇒ (*'state* × (*'information*, *'peer*) *message word*)

and a :: $('information, 'peer)$ *action*
and k :: *bound*
assumes *mbox-step C1 a k C2*
shows *is-mbox-config C1* **and** *is-mbox-config C2*
and $\exists\, i\ p\ q.\ a = !\langle(i^{p \to q})\rangle \vee a = ?\langle(i^{p \to q})\rangle$ **and** *get-actor a* $\neq$ *get-object a*
and *fst* (*C1* (*get-actor a*)) $-a\to_{\mathcal{C}}$(*get-actor a*) (*fst* (*C2* (*get-actor a*)))
and *is-output a* $\Longrightarrow$ *snd* (*C1* (*get-actor a*)) = *snd* (*C2* (*get-actor a*))
and *is-output a* $\Longrightarrow$ (| (*snd* (*C1* (*get-object a*))) |) $<_{\mathcal{B}}$ k
and *is-output a* $\Longrightarrow$ *C2* (*get-object a*) =
(*fst* (*C1* (*get-object a*)), (*snd* (*C1* (*get-object a*))) $\cdot$ [*get-message a*])
and *is-input a* $\Longrightarrow$ (*snd* (*C1* (*get-actor a*))) = [*get-message a*] $\cdot$ *snd* (*C2* (*get-actor a*))
and *is-output a* $\Longrightarrow \forall\, x.\ x \notin \{$*get-actor a*, *get-object a*$\} \longrightarrow$ *C1*(x) = *C2*(x)
and *is-input a* $\Longrightarrow \forall\, x.\ x \neq$ *get-actor a* $\longrightarrow$ *C1*(x) = *C2*(x)
using *assms*
proof *induct*
case (*MboxSend C1 a i p q C2 k*)
assume *A1*: *is-mbox-config C1*
thus *is-mbox-config C1* **.**
assume *A2*: $a = !\langle(i^{p \to q})\rangle$
thus $\exists\, i\ p\ q.\ a = !\langle(i^{p \to q})\rangle \vee a = ?\langle(i^{p \to q})\rangle$
by *blast*
assume *A3*: *fst* (*C1 p*) $-(!\langle(i^{p \to q})\rangle)\to_{\mathcal{C}} p$ (*fst* (*C2 p*))
with *A2* **show** *fst* (*C1* (*get-actor a*)) $-a\to_{\mathcal{C}}$(*get-actor a*) (*fst* (*C2* (*get-actor a*)))
by *simp*
have *A4*: *CommunicatingAutomaton p* ($\mathcal{S}$ *p*) ($\mathcal{I}$ *p*) $\mathcal{M}$ ($\mathcal{R}$ *p*)
using *automaton-of-peer*[*of p*]
by *simp*
with *A2 A3* **show** *get-actor a* $\neq$ *get-object a*
using *CommunicatingAutomaton.well-formed-transition*[*of p* $\mathcal{S}$ *p* $\mathcal{I}$ *p* $\mathcal{M}$ $\mathcal{R}$ *p fst* (*C1 p*) *a*
fst (*C2 p*)]
by *auto*
assume *A5*: *snd* (*C1 p*) = *snd* (*C2 p*)
with *A2* **show** *is-output a* $\Longrightarrow$ *snd* (*C1* (*get-actor a*)) = *snd* (*C2* (*get-actor a*))
by *simp*
assume (|*snd* (*C1 q*)|) $<_{\mathcal{B}}$ k
with *A2* **show** *is-output a* $\Longrightarrow$ (| (*snd* (*C1* (*get-object a*))) |) $<_{\mathcal{B}}$ k
by *simp*
assume *A6*: *C2 q* = (*fst* (*C1 q*), *snd* (*C1 q*) $\cdot$ [$i^{p \to q}$])
with *A2* **show** *is-output a* $\Longrightarrow$ *C2* (*get-object a*) =
(*fst* (*C1* (*get-object a*)), (*snd* (*C1* (*get-object a*))) $\cdot$ [*get-message a*])
by *simp*
from *A2* **show** *is-input a* $\Longrightarrow$ (*snd* (*C1* (*get-actor a*))) = [*get-message a*] $\cdot$ *snd* (*C2* (*get-actor a*))
by *simp*
assume *A7*: $\forall\, x.\ x \notin \{p,\ q\} \longrightarrow$ *C1 x* = *C2 x*

with *A2* **show** *is-output* $a \Longrightarrow \forall x.\ x \notin \{$*get-actor* a, *get-object* $a\} \longrightarrow C1(x) = C2(x)$
by *simp*
from *A2* **show** *is-input* $a \Longrightarrow \forall x.\ x \neq$ *get-actor* $a \longrightarrow C1(x) = C2(x)$
by *simp*
show *is-mbox-config C2*
unfolding *is-mbox-config-def*
proof *clarify*
fix x :: *'peer*
show *fst* $(C2\ x) \in \mathcal{S}(x) \wedge$ *snd* $(C2\ x) \in \mathcal{M}^*$
proof (*cases* $x = p$)
assume B: $x = p$
with *A3 A4* **have** *fst* $(C2\ x) \in \mathcal{S}(x)$
using *CommunicatingAutomaton.well-formed-transition*[*of* p $\mathcal{S}$ p $\mathcal{I}$ p $\mathcal{M}$ $\mathcal{R}$ p *fst* $(C1\ p)$
$!\langle(i^{p \rightarrow q})\rangle$ *fst* $(C2\ p)$]
by *simp*
moreover from *A1 A5 B* **have** *snd* $(C2\ x) \in \mathcal{M}^*$
unfolding *is-mbox-config-def*
by *metis*
ultimately show *fst* $(C2\ x) \in \mathcal{S}(x) \wedge$ *snd* $(C2\ x) \in \mathcal{M}^*$
by *simp*
next
assume B: $x \neq p$
show *fst* $(C2\ x) \in \mathcal{S}(x) \wedge$ *snd* $(C2\ x) \in \mathcal{M}^*$
proof (*cases* $x = q$)
assume $x = q$
moreover from *A1 A6* **have** *fst* $(C2\ q) \in \mathcal{S}(q)$
unfolding *is-mbox-config-def*
by *simp*
moreover from *A3 A4* **have** $i^{p \rightarrow q} \in \mathcal{M}$
using *CommunicatingAutomaton.well-formed-transition*[*of* p $\mathcal{S}$ p $\mathcal{I}$ p $\mathcal{M}$ $\mathcal{R}$ p
fst $(C1\ p)$ $!\langle(i^{p \rightarrow q})\rangle$ *fst* $(C2\ p)$]
by *simp*
with *A1 A6* **have** *snd* $(C2\ q) \in \mathcal{M}^*$
unfolding *is-mbox-config-def*
using *message-alphabet Alphabet.EmptyWord*[*of* $\mathcal{M}$] *Alphabet.Composed*[*of* $\mathcal{M}$ $i^{p \rightarrow q}$ ε]
Alphabet.concat-words-over-an-alphabet[*of* $\mathcal{M}$ *snd* $(C1\ q)$ $[i^{p \rightarrow q}]$]
by *simp*
ultimately show *fst* $(C2\ x) \in \mathcal{S}(x) \wedge$ *snd* $(C2\ x) \in \mathcal{M}^*$
by *simp*
next
assume $x \neq q$
with *A1 A7 B* **show** *fst* $(C2\ x) \in \mathcal{S}(x) \wedge$ *snd* $(C2\ x) \in \mathcal{M}^*$
unfolding *is-mbox-config-def*
by (*metis insertE singletonD*)
qed

qed
qed
next
case (*MboxRecv C1 a i p q C2 k*)
assume *A1*: *is-mbox-config C1*
thus *is-mbox-config C1* **.**
assume *A2*: $a = ?\langle(i^{p \to q})\rangle$
thus $\exists\, i\ p\ q.\ a = !\langle(i^{p \to q})\rangle \vee a = ?\langle(i^{p \to q})\rangle$
by *blast*
assume *A3*: $fst\ (C1\ q)\ {-}(?\langle(i^{p \to q})\rangle){\to_{\mathcal{C}}}\,q\ (fst\ (C2\ q))$
with *A2* **show** $fst\ (C1\ (get\text{-}actor\ a))\ {-}a{\to_{\mathcal{C}}}(get\text{-}actor\ a)\ (fst\ (C2\ (get\text{-}actor\ a)))$
by *simp*
have *A4*: *CommunicatingAutomaton q* $(\mathcal{S}\ q)\ (\mathcal{I}\ q)\ \mathcal{M}\ (\mathcal{R}\ q)$
using *automaton-of-peer*[*of q*]
by *simp*
with *A2 A3* **show** $get\text{-}actor\ a \neq get\text{-}object\ a$
using *CommunicatingAutomaton.well-formed-transition*[*of q* $\mathcal{S}$ *q* $\mathcal{I}$ *q* $\mathcal{M}$ $\mathcal{R}$ *q fst* (*C1 q*) *a*
fst (*C2 q*)]
by *auto*
from *A2* **show** $is\text{-}output\ a \Longrightarrow snd\ (C1\ (get\text{-}actor\ a)) = snd\ (C2\ (get\text{-}actor\ a))$
by *simp*
from *A2* **show** $is\text{-}output\ a \Longrightarrow (\ |\ (snd\ (C1\ (get\text{-}object\ a)))\ |\ ) <_{\mathcal{B}} k$
by *simp*
from *A2* **show** $is\text{-}output\ a \Longrightarrow C2\ (get\text{-}object\ a) =$
$(fst\ (C1\ (get\text{-}object\ a)),\ (snd\ (C1\ (get\text{-}object\ a))) \cdot [get\text{-}message\ a])$
by *simp*
assume *A5*: $snd\ (C1\ q) = [i^{p \to q}] \cdot snd\ (C2\ q)$
with *A2* **show** $is\text{-}input\ a \Longrightarrow (snd\ (C1\ (get\text{-}actor\ a))) = [get\text{-}message\ a] \cdot snd\ (C2\ (get\text{-}actor\ a))$
by *simp*
from *A2* **show** $is\text{-}output\ a \Longrightarrow \forall\, x.\ x \notin \{get\text{-}actor\ a,\ get\text{-}object\ a\} \longrightarrow C1(x) = C2(x)$
by *simp*
assume *A6*: $\forall\, x.\ x \neq q \longrightarrow C1\ x = C2\ x$
with *A2* **show** $is\text{-}input\ a \Longrightarrow \forall\, x.\ x \neq get\text{-}actor\ a \longrightarrow C1(x) = C2(x)$
by *simp*
show *is-mbox-config C2*
unfolding *is-mbox-config-def*
proof *clarify*
fix *x* :: *'peer*
show $fst\ (C2\ x) \in \mathcal{S}(x) \wedge snd\ (C2\ x) \in \mathcal{M}^*$
proof (*cases x = q*)
assume *B*: $x = q$
with *A3 A4* **have** $fst\ (C2\ x) \in \mathcal{S}(x)$
using *CommunicatingAutomaton.well-formed-transition*[*of q* $\mathcal{S}$ *q* $\mathcal{I}$ *q* $\mathcal{M}$ $\mathcal{R}$ *q fst* (*C1 q*)
$?\langle(i^{p \to q})\rangle$ *fst* (*C2 q*)]

by *simp*
moreover from *A3 A4* **have** $i^{p \rightarrow q} \in \mathcal{M}$
using *CommunicatingAutomaton.well-formed-transition*[*of q* $\mathcal{S}$ *q* $\mathcal{I}$ *q* $\mathcal{M}$ $\mathcal{R}$ *q fst* (*C1 q*)
?$\langle(i^{p \rightarrow q})\rangle$ *fst* (*C2 q*)]
by *simp*
with *A1 A5 B* **have** *snd* (*C2 x*) $\in \mathcal{M}^*$
unfolding *is-mbox-config-def*
using *message-alphabet*
Alphabet.split-a-word-over-an-alphabet(*2*)[*of* $\mathcal{M}$ [$i^{p \rightarrow q}$] *snd* (*C2 q*)]
by *metis*
ultimately show *fst* (*C2 x*) $\in \mathcal{S}(x) \wedge$ *snd* (*C2 x*) $\in \mathcal{M}^*$
by *simp*
next
assume $x \neq q$
with *A1 A6* **show** *fst* (*C2 x*) $\in \mathcal{S}(x) \wedge$ *snd* (*C2 x*) $\in \mathcal{M}^*$
unfolding *is-mbox-config-def*
by *metis*
qed
qed
qed

lemma *mbox-step-output-rev*:
fixes *C1 C2* :: *'peer* $\Rightarrow$ (*'state* $\times$ (*'information*, *'peer*) *message word*)
and *i* :: *'information*
and *p q* :: *'peer*
and *k* :: *bound*
assumes *mbox-step C1* (!$\langle(i^{p \rightarrow q})\rangle$) *k C2*
shows *is-mbox-config C1* **and** *is-mbox-config C2* **and** $p \neq q$
and *fst* (*C1 p*) $-(!\langle(i^{p \rightarrow q})\rangle) \rightarrow_{\mathcal{C}} p$ (*fst* (*C2 p*)) **and** *snd* (*C1 p*) = *snd* (*C2 p*)
and (| (*snd* (*C1 q*)) |) $<_{\mathcal{B}}$ *k*
and *C2 q* = (*fst* (*C1 q*), (*snd* (*C1 q*)) $\cdot$ [*get-message* (!$\langle(i^{p \rightarrow q})\rangle$)])
and $\forall x.\ x \notin \{p, q\} \longrightarrow C1(x) = C2(x)$
proof −
show *is-mbox-config C1*
using *assms mbox-step-rev*(*1*)[*of C1* !$\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *simp*
show *is-mbox-config C2*
using *assms mbox-step-rev*(*2*)[*of C1* !$\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *simp*
show $p \neq q$
using *assms mbox-step-rev*(*4*)[*of C1* !$\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *simp*
show *fst* (*C1 p*) $-(!\langle(i^{p \rightarrow q})\rangle) \rightarrow_{\mathcal{C}} p$ (*fst* (*C2 p*))
using *assms mbox-step-rev*(*5*)[*of C1* !$\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *simp*
show *snd* (*C1 p*) = *snd* (*C2 p*)
using *assms mbox-step-rev*(*6*)[*of C1* !$\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *simp*

show (| (*snd* (*C1* *q*)) |) $<_{\mathcal{B}}$ *k*
using *assms mbox-step-rev*(*7*)[*of C1* $!\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *fastforce*
show *C2* *q* = (*fst* (*C1* *q*), (*snd* (*C1* *q*)) · [*get-message* ($!\langle(i^{p \rightarrow q})\rangle$)])
using *assms mbox-step-rev*(*8*)[*of C1* $!\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *simp*
show $\forall x.\ x \notin \{p,\ q\} \longrightarrow$ *C1*(*x*) = *C2*(*x*)
using *assms mbox-step-rev*(*10*)[*of C1* $!\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *simp*
qed

lemma *mbox-step-input-rev*:
fixes *C1 C2* :: *'peer* ⇒ (*'state* × (*'information*, *'peer*) *message word*)
and *i* :: *'information*
and *p q* :: *'peer*
and *k* :: *bound*
assumes *mbox-step C1* ($?\langle(i^{p \rightarrow q})\rangle$) *k C2*
shows *is-mbox-config C1* **and** *is-mbox-config C2* **and** $p \neq q$
and *fst* (*C1* *q*) $-(?\langle(i^{p \rightarrow q})\rangle)\rightarrow_{\mathcal{C}} q$ (*fst* (*C2* *q*)) **and** (*snd* (*C1* *q*)) = $[i^{p \rightarrow q}]$ · *snd* (*C2* *q*)
and $\forall x.\ x \neq q \longrightarrow$ *C1*(*x*) = *C2*(*x*)
using *assms mbox-step-rev*[*of C1* $?\langle(i^{p \rightarrow q})\rangle$ *k C2*]
by *simp-all*

— if mbox can take a bounded step, it can also take an unbounded step
lemma *mbox-step-inclusion* :
assumes *mbox-step C1 a* (*Some k*) *C2*
shows *mbox-step C1 a None C2*
by (*smt* (*verit*) *MboxRecv MboxSend NetworkOfCA.mbox-step-input-rev*(*6*) *NetworkOfCA-axioms assms*
get-actor.simps(*1*,*2*) *get-message.simps*(*1*,*2*) *get-object.simps*(*1*) *get-receiver.simps*
get-sender.simps is-bounded.simps(*1*) *is-output.simps*(*1*,*2*) *mbox-step-output-rev*(*5*)
mbox-step-rev(*1*,*10*,*3*,*5*,*8*,*9*) *these-empty*)

5.6.2 mbox step conversions tofrom

lemma *send-step-to-mbox-step*:
assumes [*a*] ∈ $\mathcal{L}$ *p* **and** *is-output a*
shows $\exists C.\ \mathcal{C}_{\mathcal{I}\mathfrak{m}} -\langle a,\ \infty\rangle\rightarrow C$
using *assms*
proof −
obtain *s2* **where** *s2-def*: ($\mathcal{I}$ *p*, *a*, *s2*) ∈ $\mathcal{R}$ *p* **by** (*meson assms*(*1*) *lang-implies-run lang-implies-trans*)
then obtain *q i* **where** *a-def*: *a* = $!\langle(i^{p \rightarrow q})\rangle$
by (*metis CommunicatingAutomaton-def action.exhaust assms*(*2*) *automaton-of-peer get-actor.simps*(*1*) *get-sender.simps is-output.simps*(*2*) *message.exhaust*)
then have $p \neq q$ **by** (*metis CommunicatingAutomaton.well-formed-transition*

‹$\bigwedge$*thesis*. ($\bigwedge$*s2*. $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *p* −*a*→$_{\mathcal{C}}$*p s2* ⟹ *thesis*) ⟹ *thesis*› *automaton-of-peer get-object.simps*(*1*) *get-receiver.simps*)
let *?C0* = ($\mathcal{C}_{\mathcal{I}\mathfrak{m}}$)(*p* := (*s2*, *ε*))
let *?C* = (*?C0*)(*q* := ($\mathcal{I}$ *q*, [($i^{p \to q}$)]))
have *is-mbox-config ?C* **by** (*smt* (*verit*) *Alphabet.WordsOverAlphabet.simps CommunicatingAutomaton.well-formed-transition a-def automaton-of-peer fun-upd-apply get-message.simps*(*1*) *initial-configuration-is-synchronous-configuration is-mbox-config-def is-sync-config-def message-alphabet s2-def snd-conv split-pairs*)
then have *C-prop*: ∀ *x*. *x* ∉ {*p*, *q*} ⟶ $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$(*x*) = *?C*(*x*) **by** *simp*
then have *fst* ($\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *p*) = $\mathcal{I}$ *p* **by** *auto*
then have *fst* (*?C p*) = *s2* **by** (*simp add*: ‹*p* ≠ *q*›)
have ($\mathcal{I}$ *p*) −(!⟨($i^{p \to q}$)⟩)→$_{\mathcal{C}}$*p s2* **using** *a-def s2-def* **by** *auto*
have *is-mbox-config* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ **by** (*simp add*: *initial-configuration-is-mailbox-configuration*)
have *fst* ($\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *p*) −(!⟨($i^{p \to q}$)⟩)→$_{\mathcal{C}}$*p* (*fst* (*?C p*))
using ‹*fst* (((*λp*. ($\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *p*, *ε*)) (*p* := (*s2*, *ε*), *q* := ($\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *q*, $i^{p \to q}$ # *ε*))) *p*) = *s2*› *a-def s2-def* **by** *auto*
then have *C-prop2*: *snd* ($\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *p*) = *snd* (*?C p*) **by** (*simp add*: ‹*p* ≠ *q*›)
then have *C-prop3*: *?C q* = (*fst* ($\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *q*), (*snd* ($\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *q*)) · [($i^{p \to q}$)]) **by** *simp*
then have *mbox-step* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *a None ?C*
using *C-prop2 MboxSend*
‹*fst* (((*λp*. ($\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *p*, *ε*)) (*p* := (*s2*, *ε*), *q* := ($\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *q*, $i^{p \to q}$ # *ε*))) *p*) = *s2*› *a-def initial-configuration-is-mailbox-configuration s2-def* **by** *force*
then show *?thesis* **by** *auto*
qed

lemma *gen-send-step-to-mbox-step*:
assumes (*s1*, !⟨($i^{p \to q}$)⟩, *s2*) ∈ $\mathcal{R}$ *p* **and** *fst* (*C0 p*) = *s1* **and** *fst* (*C1 p*) = *s2* **and** *snd* (*C0 p*) = *snd* (*C1 p*) **and** *C1 q* = (*fst* (*C0 q*), (*snd* (*C0 q*)) · [($i^{p \to q}$)]) **and** *is-mbox-config C0*
and ∀ *x*. *x* ∉ {*p*, *q*} ⟶ *C0*(*x*) = *C1*(*x*)
shows *C0* −⟨!⟨($i^{p \to q}$)⟩, ∞⟩→ *C1*
using *assms*
proof *auto*
have *fst* (*C0 p*) −(!⟨($i^{p \to q}$)⟩)→$_{\mathcal{C}}$*p* (*fst* (*C1 p*)) **by** (*simp add*: *assms*(*1*,*2*,*3*))
have *all*: *is-mbox-config C0* ∧ *fst* (*C0 p*) −(!⟨($i^{p \to q}$)⟩)→$_{\mathcal{C}}$*p* (*fst* (*C1 p*)) ∧ *snd* (*C0 p*) = *snd* (*C1 p*) ∧ (| (*snd* (*C0 q*)) |) <$_{\mathcal{B}}$ *None* ∧ *C1 q* = (*fst* (*C0 q*), (*snd* (*C0 q*)) · [($i^{p \to q}$)]) ∧ (∀ *x*. *x* ∉ {*p*, *q*} ⟶ *C0*(*x*) = *C1*(*x*))
using *assms* **by** *auto*
show *?thesis* **by** (*simp add*: *NetworkOfCA.MboxSend NetworkOfCA-axioms all*)
qed

lemma *valid-send-to-p-not-q* :
assumes (*s1*, !⟨($i^{p \to q}$)⟩, *s2*) ∈ $\mathcal{R}$ *p*
shows *p* ≠ *q*
by (*metis CommunicatingAutomaton.well-formed-transition assms automaton-of-peer*

get-object.simps(*1*) *get-receiver.simps*)

lemma *valid-recv-to-p-not-q* :
assumes $(s1,\ ?\langle(i^{p\rightarrow q})\rangle,\ s2) \in \mathcal{R}\ p$
shows $p \neq q$
by (*metis CommunicatingAutomaton-def NetworkOfCA.automaton-of-peer NetworkOfCA-axioms assms*
get-object.simps(*2*) *get-sender.simps*)

— define the mbox step for a given send step (of e.g. a root)
lemma *send-trans-to-mbox-step* :
assumes $(s1,\ !\langle(i^{p\rightarrow q})\rangle,\ s2) \in \mathcal{R}\ p$ **and** *is-mbox-config C0* **and** *fst* (*C0 p*) = *s1*
shows *let p-buf* = *snd* (*C0 p*); *C1* = (*C0*)(*p* := (*s2*, *p-buf*)); *q0* = *fst* (*C0 q*); *q-buf* = *snd* (*C0 q*);
C2 = (*C1*)(*q* := (*q0*, *q-buf* · $[(i^{p\rightarrow q})]$)) *in*
mbox-step C0 $(!\langle(i^{p\rightarrow q})\rangle)$ *None C2*
using *assms*
proof −
let *?p-buf* = *snd* (*C0 p*)
let *?C1* = (*C0*)(*p* := (*s2*, *?p-buf*))
let *?q0* = *fst* (*C0 q*)
let *?q-buf* = *snd* (*C0 q*)
let *?C2* = (*?C1*)(*q* := (*?q0*, *?q-buf* · $[(i^{p\rightarrow q})]$))
have $q \neq p$ **using** *assms*(*1*) *valid-send-to-p-not-q* **by** *blast*
have *m1*: *snd* (*C0 p*) = *snd* (*?C2 p*) **using** ‹$q \neq p$› **by** *auto*
have *m2*: *fst* (*C0 p*) $-(!\langle(i^{p\rightarrow q})\rangle)\rightarrow_{\mathcal{C}} p$ (*fst* (*?C2 p*)) **using** ‹$q \neq p$› *assms*(*1*,*3*) **by** *fastforce*
have *m3*: *?C2 q* = (*fst* (*C0 q*), (*snd* (*C0 q*)) · $[(i^{p\rightarrow q})]$) **by** *simp*
have *m4*: $(\forall x.\ x \notin \{p,\ q\} \longrightarrow C0(x) = ?C2(x))$ **by** *simp*
have *m5*: (| (*snd* (*C0 q*)) |) $<_{\mathcal{B}}$ *None* **by** *simp*
have *mbox-step C0* $(!\langle(i^{p\rightarrow q})\rangle)$ *None ?C2* **using** *assms*(*2*) *gen-send-step-to-mbox-step m1 m2 m3 m4* **by** *blast*
then show *?thesis* **by** *auto*
qed

5.6.3 mbox run

lemma *mbox-run-rev-unbound* :
assumes *mbox-run C0 None* (*w*·[*a*]) (*xc*@[*C*])
shows *mbox-run C0 None w xc* ∧ *last* (*C0*#*xc*) $-\langle a,\ \infty\rangle\rightarrow$ *C*
by (*smt* (*verit*) *Nil-is-append-conv append1-eq-conv assms mbox-run.simps*
not-Cons-self2)

lemma *mbox-run-rev-bound* :
assumes *mbox-run C0* (*Some k*) (*w*·[*a*]) (*xc*@[*C*])
shows *mbox-run C0* (*Some k*) *w xc* ∧ *last* (*C0*#*xc*) $-\langle a,\ k\rangle\rightarrow$ *C*
by (*smt* (*verit*) *Nil-is-append-conv append1-eq-conv assms mbox-run.simps*
not-Cons-self2)

```
lemma run-produces-mailbox-configurations:
  fixes C C′ :: ′peer ⇒ (′state × (′information, ′peer) message word)
    and k    :: bound
    and w    :: (′information, ′peer) action word
    and xc   :: (′peer ⇒ (′state × (′information, ′peer) message word)) list
  assumes mbox-run C k w xc
    and C′ ∈ set xc
  shows is-mbox-config C′
  using assms
proof induct
  case (MREmpty C k)
  assume C′ ∈ set []
  hence False
    by simp
  thus is-mbox-config C′
    by simp
next
  case (MRComposedNat C0 k w xc a C)
  assume A1: C′ ∈ set xc ⟹ is-mbox-config C′ and A2: last (C0#xc) −⟨a, k⟩→
C
    and A3: C′ ∈ set (xc · [C])
  show is-mbox-config C′
  proof (cases C = C′)
    assume C = C′
    with A2 show is-mbox-config C′
      using mbox-step-rev(2)[of last (C0#xc) a Some k C]
      by simp
  next
    assume C ≠ C′
    with A1 A3 show is-mbox-config C′
      by simp
  qed
next
  case (MRComposedInf C0 w xc a C)
  assume A1: C′ ∈ set xc ⟹ is-mbox-config C′ and A2: last (C0#xc) −⟨a,
∞⟩→ C
    and A3: C′ ∈ set (xc · [C])
  show is-mbox-config C′
  proof (cases C = C′)
    assume C = C′
    with A2 show is-mbox-config C′
      using mbox-step-rev(2)[of last (C0#xc) a None C]
      by simp
  next
    assume C ≠ C′
    with A1 A3 show is-mbox-config C′
      by simp
  qed
qed
```

lemma *mbox-step-to-run*:
  assumes *mbox-step C0 a None C*
  shows *mbox-run C0 None* $[a]$ $[C]$
  by (*metis MRComposedInf MREmpty append.left-neutral assms last-ConsL*)

5.6.4 mbox traces

lemma *Mbox-Traces-rev* :
  assumes $w \in \mathcal{T}_k$
  shows $\exists$ *xc. mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *k w xc*
  by (*metis MboxTraces.cases assms*)

lemma *mbox-run-inclusion*:
  assumes *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ (*Some k*) *w xc*
  shows *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None w xc*
  using *assms*
proof (*induct rule*: *mbox-run.induct*)
  case (*MREmpty C k*)
  then show *?case* **by** (*simp add*: *mbox-run.MREmpty*)
next
  case (*MRComposedNat C0 k w xc a C*)
  then show *?case* **by** (*simp add*: *MRComposedInf mbox-step-inclusion*)
next
  case (*MRComposedInf C0 w xc a C*)
  then show *?case* **by** (*simp add*: *mbox-run.MRComposedInf*)
qed

lemma *mbox-bounded-lang-inclusion* :
  shows $\mathcal{T}_{(Some\ k)} \subseteq \mathcal{T}_{None}$
  using *MboxTraces-def MboxTracesp.simps mbox-run-inclusion* **by** *fastforce*

lemma *execution-implies-mbox-trace* :
  assumes $w \in \mathcal{T}_k$
  shows $w{\downarrow_!} \in \mathcal{L}_k$
  using *assms* **by** *blast*
lemma *mbox-trace-implies-execution* :
  assumes $w \in \mathcal{L}_k$
  shows $\exists\, w'.\ w'{\downarrow_!} = w \wedge w' \in \mathcal{T}_k$
  using *assms* **by** *blast*

5.6.5 Language Hierarchy

theorem *sync-word-in-mbox-size-one*:
  shows $\mathcal{L}_{\mathbf{0}} \subseteq \mathcal{L}_{\mathbf{1}}$
proof *clarify*
  fix v :: (*'information*, *'peer*) *action word*
  assume $v \in \mathcal{L}_{\mathbf{0}}$
  then obtain *xcs C0* **where** *sync-run C0 v xcs* **and** $C0 = \mathcal{C}_{\mathcal{I}\mathbf{0}}$

using *SyncTracesp.simps SyncTracesp-SyncTraces-eq*
by *auto*
hence $\exists\, w\ xcm.\ mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ (\mathcal{B}\ 1)\ w\ xcm \wedge v = w{\downarrow_!} \wedge$
$(\forall\, p.\ last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ p = (last\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\#xcs)\ p,\ \varepsilon))$
proof *induct*
case (*SREmpty C*)
have $mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ (\mathcal{B}\ 1)\ \varepsilon\ []$
using $MREmpty[of\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ \mathcal{B}\ 1]$
by *simp*
moreover have $\varepsilon = \varepsilon{\downarrow_!}$
by *simp*
moreover have $\forall\, p.\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ p = (\mathcal{C}_{\mathcal{I}\mathbf{0}}\ p,\ \varepsilon)$
by *simp*
ultimately show $\exists\, w\ xcm.\ mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ (\mathcal{B}\ 1)\ w\ xcm \wedge \varepsilon = w{\downarrow_!} \wedge$
$(\forall\, p.\ last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ p = (last\ [\mathcal{C}_{\mathcal{I}\mathbf{0}}]\ p,\ \varepsilon))$
by *fastforce*
next
case (*SRComposed C0 v xc a C*)
assume $C0 = \mathcal{C}_{\mathcal{I}\mathbf{0}} \Longrightarrow \exists\, w\ xcm.\ mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ (\mathcal{B}\ 1)\ w\ xcm \wedge v = w{\downarrow_!} \wedge$
$(\forall\, p.\ last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ p = (last\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\#xc)\ p,\ \varepsilon))$
and $B1$: $C0 = \mathcal{C}_{\mathcal{I}\mathbf{0}}$
then obtain *w xcm* **where** $B2$: $mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ (\mathcal{B}\ 1)\ w\ xcm$ **and** $B3$: $v = w{\downarrow_!}$
and $B4$: $\forall\, p.\ last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ p = (last\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\#xc)\ p,\ \varepsilon)$
by *blast*
assume $last\ (C0\#xc) -\langle a,\ \mathbf{0}\rangle\rightarrow C$
with $B1$ **obtain** $C1$ **where** $B5$: $C1 = last\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\#xc)$ **and** $B6$: $C1 -\langle a,\ \mathbf{0}\rangle\rightarrow$
C
by *blast*
from $B6$ **obtain** *i p q* **where** $B7$: $a = !\langle(i^{p \rightarrow q})\rangle$ **and** $B8$: $C1\ p -a\rightarrow_{\mathcal{C}} p\ (C\ p)$
and $B9$: $C1\ q -(?\langle(i^{p \rightarrow q})\rangle)\rightarrow_{\mathcal{C}} q\ (C\ q)$ **and** $B10$: $p \neq q$
and $B11$: $\forall\, x.\ x \notin \{p,\ q\} \longrightarrow C1\ x = C\ x$
using $sync\text{-}step\text{-}rev[of\ C1\ a\ C]$
by *auto*
define $C2 :: 'peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)$ **where**
$C2\text{-}def$: $C2 \equiv \lambda x.\ if\ x = p\ then\ (C\ p,\ \varepsilon)\ else\ (C1\ x,\ if\ x = q\ then\ [i^{p \rightarrow q}]$
$else\ \varepsilon)$
define $C3 :: 'peer \Rightarrow ('state \times ('information,\ 'peer)\ message\ word)$ **where**
$C3\text{-}def$: $C3 \equiv \lambda x.\ (C\ x,\ \varepsilon)$
from $B2$ **have** $is\text{-}mbox\text{-}config\ (last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm))$
using $run\text{-}produces\text{-}mailbox\text{-}configurations[of\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ \mathcal{B}\ 1\ w\ xcm\ last\ xcm]$
initial-configuration-is-mailbox-configuration
by *simp*
moreover from $B4\ B5\ B7\ B8$ **have** $fst\ (last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ p) -(!\langle(i^{p \rightarrow q})\rangle)\rightarrow_{\mathcal{C}} p$
$(fst\ (C2\ p))$
unfolding $C2\text{-}def$
by *simp*
moreover from $B4$ **have** $snd\ (last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ p) = snd\ (C2\ p)$
unfolding $C2\text{-}def$
by *simp*

moreover from $B4$ **have** $(\mid snd\ (last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ q) \mid) <_{\mathcal{B}} \mathcal{B}\ 1$
by *simp*
moreover from $B4\ B5\ B10$
have $C2\ q = (fst\ (last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ q),\ snd\ (last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ q) \cdot [i^{p \rightarrow q}])$
unfolding *C2-def*
by *simp*
moreover from $B4\ B5$ **have** $\forall x.\ x \notin \{p,\ q\} \longrightarrow last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ x = C2\ x$
unfolding *C2-def*
by *simp*
ultimately have $B12$: $last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ -\langle a,\ 1\rangle\rightarrow\ C2$
using $B7\ MboxSend[of\ last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ !\langle(i^{p \rightarrow q})\rangle\ i\ p\ q\ C2\ \mathcal{B}\ 1]$
by *simp*
hence *is-mbox-config C2*
using $mbox\text{-}step\text{-}rev(2)[of\ last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ a\ \mathcal{B}\ 1\ C2]$
by *simp*
moreover from $B9\ B10$ **have** $fst\ (C2\ q)\ -(?\langle(i^{p \rightarrow q})\rangle)\rightarrow_{\mathcal{C}} q\ (fst\ (C3\ q))$
unfolding *C2-def C3-def*
by *simp*
moreover from $B10$ **have** $snd\ (C2\ q) = [i^{p \rightarrow q}] \cdot snd\ (C3\ q)$
unfolding *C2-def C3-def*
by *simp*
moreover from $B11$ **have** $\forall x.\ x \neq q \longrightarrow C2\ x = C3\ x$
unfolding *C2-def C3-def*
by *simp*
ultimately have $C2\ -\langle ?\langle(i^{p \rightarrow q})\rangle,\ 1\rangle\rightarrow\ C3$
using $MboxRecv[of\ C2\ ?\langle(i^{p \rightarrow q})\rangle\ i\ p\ q\ C3\ \mathcal{B}\ 1]$
by *simp*
with $B2\ B12$ **have** $mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ (\mathcal{B}\ 1)\ (w\cdot[a,\ ?\langle(i^{p \rightarrow q})\rangle])\ (xcm\cdot[C2,\ C3])$
using $MRComposedNat[of\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ 1\ w\ xcm\ a\ C2]$
$MRComposedNat[of\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ 1\ w\cdot[a]\ xcm\cdot[C2]\ ?\langle(i^{p \rightarrow q})\rangle\ C3]$
by *simp*
moreover from $B3\ B7$ **have** $v\cdot[a] = (w\cdot[a,\ ?\langle(i^{p \rightarrow q})\rangle])\downarrow_!$
using $filter\text{-}append[of\ is\text{-}output\ w\ [a,\ ?\langle(i^{p \rightarrow q})\rangle]]$
by *simp*
moreover have $\forall p.\ last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#(xcm\cdot[C2,\ C3]))\ p = (last\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\#(xc\cdot[C]))\ p,$
$\varepsilon)$
unfolding *C3-def*
by *simp*
ultimately show $\exists w\ xcm.\ mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ (\mathcal{B}\ 1)\ w\ xcm \wedge v\cdot[a] = w\downarrow_! \wedge$
$(\forall p.\ last\ (\mathcal{C}_{\mathcal{I}\mathfrak{m}}\#xcm)\ p = (last\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\#(xc\cdot[C]))\ p,\ \varepsilon))$
by *blast*
qed
then obtain $w\ xcm$ **where** $A1$: $mbox\text{-}run\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ (\mathcal{B}\ 1)\ w\ xcm$ **and** $A2$: $v = w\downarrow_!$
by *blast*
from $A1$ **have** $w \in \mathcal{T}_{\mathcal{B}\ 1}$
by (*simp add*: *MboxTraces.intros*)
with $A2$ **show** $\exists w.\ v = w\downarrow_! \wedge w \in \mathcal{T}_{\mathcal{B}\ 1}$
by *blast*
qed

lemma *mbox-sync-lang-unbounded-inclusion*:
shows $\mathcal{L}_\mathbf{0} \subseteq \mathcal{L}_\infty$
using *NetworkOfCA.mbox-bounded-lang-inclusion NetworkOfCA-axioms sync-word-in-mbox-size-one*
by *force*

— C1 ->send-> C1(p:= (C2 p)) ->recvrightarrow> C2
— shows that a sync step can be simulated with two Mbox steps
lemma *sync-step-to-mbox-steps*:
assumes $C1\ -\langle !\langle (i^{p \rightarrow q})\rangle,\ \mathbf{0}\rangle \rightarrow\ C2$
shows *let* $c1 = \lambda x.\ (C1\ x,\ \varepsilon);\ c3 = \lambda x.\ (C2\ x,\ \varepsilon);\ c2 = (c3)(q := (C1\ q,\ [(i^{p \rightarrow q})]))$ *in*
mbox-step $c1\ (!\langle (i^{p \rightarrow q})\rangle)$ *None* $c2 \wedge$ *mbox-step* $c2\ (?\langle (i^{p \rightarrow q})\rangle)$ *None* $c3$
proof − — C1 -> C2 in sync means we have c1 -> c2 -> c3 in mbox, where in c2 the message is in the mbox of the respective peer
let $?c1 = \lambda x.\ (C1\ x,\ \varepsilon)$ — C1 as mbox config
let $?c3 = \lambda x.\ (C2\ x,\ \varepsilon)$ — C2 as mbox config
let $?c2 = (?c3)(q := (C1\ q,\ [(i^{p \rightarrow q})]))$ — additional step in mbox that isnt there in sync (sequential vs synchronously)
let $?a = !\langle (i^{p \rightarrow q})\rangle$
have *O1*: $(C1\ p)\ -(!\langle (i^{p \rightarrow q})\rangle) \rightarrow_\mathcal{C} p\ (C2\ p)$ **by** (*simp add*: *assms sync-step-output-rev*(*4*))
then have $(C1\ q)\ -(?\langle (i^{p \rightarrow q})\rangle) \rightarrow_\mathcal{C} q\ (C2\ q)$ **by** (*simp add*: *assms sync-step-output-rev*(*5*))
then have $\forall x.\ x \notin \{p,\ q\} \longrightarrow C1(x) = C2(x)$ **using** *assms sync-step-output-rev*(*6*) **by** *blast*
then have *S0*: *fst* $(?c2\ p) = C2\ p$ **using** *assms sync-step-output-rev*(*3*) **by** *auto*
then have *S1*: *is-mbox-config* $?c1$ **using** *assms sync-config-to-mbox sync-step-rev*(*1*) **by** *blast*
then have *S2*: *fst* $(?c1\ p)\ -(!\langle (i^{p \rightarrow q})\rangle) \rightarrow_\mathcal{C} p$ (*fst* $(?c2\ p))$ **using** *O1 S0* **by** *auto*
then have *S3*: *snd* $(?c1\ p) =$ *snd* $(?c2\ p)$ **using** *assms sync-step-output-rev*(*3*) **by** *auto*
then have *S4*: $(\ |\ ($*snd* $(?c1\ q))\ |\ ) <_\mathcal{B}$ *None* **by** *simp*
then have *S5*: $?c2\ q = ($*fst* $(?c1\ q),\ ($*snd* $(?c1\ q)) \cdot [(i^{p \rightarrow q})])$ **by** *simp*
then have *S6*: $(\forall x.\ x \notin \{p,\ q\} \longrightarrow ?c1(x) = ?c2(x))$ **by** (*simp add*: ‹$\forall x.\ x \notin \{p,\ q\} \longrightarrow C1\ x = C2\ x$›)
then have *is-mbox-config* $?c1 \wedge ?a = !\langle (i^{p \rightarrow q})\rangle \wedge$ *fst* $(?c1\ p)\ -(!\langle (i^{p \rightarrow q})\rangle) \rightarrow_\mathcal{C} p$ (*fst* $(?c2\ p)) \wedge$
snd $(?c1\ p) =$ *snd* $(?c2\ p) \wedge (\ |\ ($*snd* $(?c1\ q))\ |\ ) <_\mathcal{B}$ *None* $\wedge$
$?c2\ q = ($*fst* $(?c1\ q),\ ($*snd* $(?c1\ q)) \cdot [(i^{p \rightarrow q})]) \wedge (\forall x.\ x \notin \{p,\ q\} \longrightarrow ?c1(x) = ?c2(x))$
using *S1 S2 S3 S4 S5* **by** *blast*
then have *mbox-step-1* : *mbox-step* $?c1\ (!\langle (i^{p \rightarrow q})\rangle)$ *None* $?c2$ **using** *MboxSend* **by** *blast*
— we have shown that mbox takes a send step from ?c1 to ?c2, now we need to show the receive step

have *R1* : *is-mbox-config* $?c2$ **using** *mbox-step-1 mbox-step-rev*(*2*) **by** *auto*
then have *R2* : *fst* $(?c2\ q) = C1\ q$ **by** *simp*
then have *R3* : *fst* $(?c3\ q) = C2\ q$ **by** *simp*
then have *R4* : *fst* $(?c2\ q)\ -(?\langle (i^{p \rightarrow q})\rangle) \rightarrow_\mathcal{C} q$ (*fst* $(?c3\ q))$ **using** *R2 R3* ‹(*C1*

$q)\ -(\ ?\langle(i^{p\rightarrow q})\rangle)\rightarrow_{\mathcal{C}}q\ (C2\ q)\rangle$ **by** *simp*
then have *R5*: $(snd\ (?c2\ q)) = [(i^{p\rightarrow q})] \cdot snd\ (?c3\ q)$ **by** *simp*
then have *R6*: $\forall\, x.\ x \neq q \longrightarrow ?c2(x) = ?c3(x)$ **by** *simp*
then have *is-mbox-config* $?c2 \wedge fst\ (?c2\ q)\ -(\ ?\langle(i^{p\rightarrow q})\rangle)\rightarrow_{\mathcal{C}}q\ (fst\ (?c3\ q)) \wedge$ $(snd\ (?c2\ q)) = [(i^{p\rightarrow q})] \cdot snd\ (?c3\ q) \wedge (\forall\, x.\ x \neq q \longrightarrow ?c2(x) = ?c3(x))$
using *R1 R4* **by** *auto*
then have *mbox-step-2* : *mbox-step* $?c2\ (?\langle(i^{p\rightarrow q})\rangle)$ *None* $?c3$ **by** (*simp add*: *MboxRecv*)
then have *mbox-step* $?c1\ (!\langle(i^{p\rightarrow q})\rangle)$ *None* $?c2 \wedge$ *mbox-step* $?c2\ (?\langle(i^{p\rightarrow q})\rangle)$ *None* $?c3$ **by** (*simp add*: *mbox-step-1*)
then have $?c1\ -\langle !\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ ?c2 \wedge ?c2\ -\langle ?\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ ?c3$ **by** *simp*
then show *?thesis* **by** *auto*
qed

— shows that sync step means mbox steps exist in general
lemma *sync-step-to-mbox-steps-existence*:
assumes $C1\ -\langle !\langle(i^{p\rightarrow q})\rangle, \mathbf{0}\rangle\rightarrow\ C2$
shows $\exists\ c1\ c2\ c3.$ *mbox-step* $c1\ (!\langle(i^{p\rightarrow q})\rangle)$ *None* $c2 \wedge$ *mbox-step* $c2\ (?\langle(i^{p\rightarrow q})\rangle)$ *None* $c3$
by (*meson assms sync-step-to-mbox-steps*)

lemma *sync-step-to-mbox-steps-alt*:
assumes $C1\ -\langle !\langle(i^{p\rightarrow q})\rangle, \mathbf{0}\rangle\rightarrow\ C2$ **and** $c1 = (\lambda x.\ (C1\ x, \varepsilon))$ **and** $c3 = (\lambda x.\ (C2\ x, \varepsilon))$ **and** $c2 = (c3)(q := (C1\ q, [(i^{p\rightarrow q})]))$
shows *mbox-step* $c1\ (!\langle(i^{p\rightarrow q})\rangle)$ *None* $c2 \wedge$ *mbox-step* $c2\ (?\langle(i^{p\rightarrow q})\rangle)$ *None* $c3$
using *assms*
proof *auto*
have *let* $c1 = \lambda x.\ (C1\ x, \varepsilon);\ c3 = \lambda x.\ (C2\ x, \varepsilon);\ c2 = (c3)(q := (C1\ q, [(i^{p\rightarrow q})]))$ *in*
mbox-step $c1\ (!\langle(i^{p\rightarrow q})\rangle)$ *None* $c2 \wedge$ *mbox-step* $c2\ (?\langle(i^{p\rightarrow q})\rangle)$ *None* $c3$
by (*simp add*: *assms*(*1*) *sync-step-to-mbox-steps*)
then show $(\lambda x.\ (C1\ x, \varepsilon))\ -\langle !\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ (\lambda x.\ (C2\ x, \varepsilon))(q := (C1\ q, i^{p\rightarrow q}\ \#\ \varepsilon))$ **by** *meson*
next
have *let* $c1 = \lambda x.\ (C1\ x, \varepsilon);\ c3 = \lambda x.\ (C2\ x, \varepsilon);\ c2 = (c3)(q := (C1\ q, [(i^{p\rightarrow q})]))$ *in*
mbox-step $c1\ (!\langle(i^{p\rightarrow q})\rangle)$ *None* $c2 \wedge$ *mbox-step* $c2\ (?\langle(i^{p\rightarrow q})\rangle)$ *None* $c3$
by (*simp add*: *assms*(*1*) *sync-step-to-mbox-steps*)
then show $(\lambda x.\ (C2\ x, \varepsilon))(q := (C1\ q, i^{p\rightarrow q}\ \#\ \varepsilon))\ -\langle ?\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ (\lambda x.\ (C2\ x, \varepsilon))$ **by** *meson*
qed

lemma *eps-in-mbox-execs*: $\varepsilon \in \mathcal{T}_{None}$ **using** *MREmpty MboxTraces.intros* **by** *blast*

6 Synchronisability

lemma *Edges-rev*:

fixes e :: $'peer \times 'peer$
assumes $e \in \mathcal{G}$
shows $\exists\, i\ p\ q.\ i^{p \to q} \in \mathcal{M} \wedge e = (p,\ q)$
proof $-$
obtain $p\ q$ **where** A: $e = (p,\ q)$
by *fastforce*
with *assms* **have** $(p,\ q) \in \mathcal{G}$
by *simp*
from *this A* **show** $\exists\, i\ p\ q.\ i^{p \to q} \in \mathcal{M} \wedge e = (p,\ q)$
by (*induct*, *blast*)
qed

lemma *w-in-peer-lang-impl-p-actor*:
assumes $w \in \mathcal{L}(p)$
shows $w = w{\downarrow_p}$
using *assms*
proof (*induct length w arbitrary*: *w*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc x*)
then obtain *v a* **where** *w-def*: $w = v\ @\ [a]$ **and** *v-len* : *length* $v = x$ **and** *v-def* : $v \in \mathcal{L}\ p$
by (*metis* (*no-types*, *lifting*) *Lang-app length-Suc-conv-rev*)
then have $v{\downarrow_p} = v$ **using** *Suc.hyps*(*1*) *Suc.prems* **by** *auto*
then obtain *s2 s3* **where** *v-run*: $(\mathcal{I}\ p)\ {-v\to^*}p\ s2$ **and** *a-run*: $s2\ {-[a]\to^*}p\ s3$
using *Lang-app-both Suc.prems*(*1*) *w-def* **by** *blast*
then have $s2\ {-a\to_{\mathcal{C}}}p\ s3$ **by** (*simp add*: *lang-implies-trans*)
then have $(s2,\ a,\ s3) \in \mathcal{R}\ p$ **by** *simp*
then have *get-actor a* $= p$ **using** *CommunicatingAutomaton.well-formed-transition automaton-of-peer* **by** *fastforce*
then show *?case*
by (*simp add*: ‹$v{\downarrow_p} = v$› *w-def*)
qed

6.1 Synchronisability is Deciable for Tree Topology in Mailbox Communication

6.1.1 Topology is a Tree

lemma *is-tree-rev*:
assumes *is-tree P E*
shows $(\exists\, p.\ P = \{p\} \wedge E = \{\}) \vee (\exists\, P'\ E'\ p\ q.\ \textit{is-tree}\ P'\ E' \wedge p \in P' \wedge q \notin P' \wedge P = \textit{insert}\ q\ P' \wedge E = \textit{insert}\ (p,\ q)\ E')$
using *assms*
proof (*induction P E rule*: *is-tree.induct*)
case (*ITRoot p*)
then show *?case* **by** *simp*
next
case (*ITNode P E p q*)

then show *?case*
  by (*intro disjI2*, *auto simp*: *insert-commute*)
qed

lemma *is-tree-rev-nonempty*:
  assumes *is-tree* P E **and** $E \neq \{\}$
  shows $(\exists P'\ E'\ p\ q.$ *is-tree* $P'\ E' \wedge p \in P' \wedge q \notin P' \wedge P =$ *insert* $q\ P' \wedge E =$ *insert* $(p,\ q)\ E')$
  using *assms*(*1*,*2*) *is-tree-rev* **by** *auto*

lemma *edge-on-peers-in-tree*:
  fixes P :: *'peer set*
    and E :: *'peer topology*
    and $p\ q$ :: *'peer*
  assumes *is-tree* P E
    and $(p,\ q) \in E$
  shows $p \in P$ **and** $q \in P$
  using *assms*
proof *induct*
  case (*ITRoot* x)
  assume $(p,\ q) \in \{\}$
  thus $p \in \{x\}$ **and** $q \in \{x\}$
    by *simp-all*
next
  case (*ITNode* P E x y)
  assume $(p,\ q) \in E \Longrightarrow p \in P$ **and** $(p,\ q) \in E \Longrightarrow q \in P$ **and** $x \in P$
    and $(p,\ q) \in$ *insert* $(x,\ y)\ E$
  thus $p \in$ *insert* $y\ P$ **and** $q \in$ *insert* $y\ P$
    by *auto*
qed

lemma *at-most-one-parent-in-tree*:
  fixes P :: *'peer set*
    and E :: *'peer topology*
    and p :: *'peer*
  assumes *is-tree* P E
  shows *card* $(E\langle\rightarrow p\rangle) \leq 1$
  using *assms*
proof *induct*
  case (*ITRoot* x)
  have $\{\}\langle\rightarrow p\rangle = \{\}$
    by *simp*
  thus *card* $(\{\}\langle\rightarrow p\rangle) \leq 1$
    by *simp*
next
  case (*ITNode* P E x y)
  assume *A1*: *is-tree* P E **and** *A2*: *card* $(E\langle\rightarrow p\rangle) \leq 1$ **and** *A3*: $y \notin P$
  show *card* (*insert* $(x,\ y)\ E\langle\rightarrow p\rangle) \leq 1$
  proof (*cases* $y = p$)

```
    assume B: y = p
    with A1 A3 have E⟨→p⟩ = {}
      using edge-on-peers-in-tree(2)[of P E - p]
      by blast
    with B have insert (x, y) E⟨→p⟩ = {x}
      by simp
    thus card (insert (x, y) E⟨→p⟩) ≤ 1
      by simp
  next
    assume y ≠ p
    hence insert (x, y) E⟨→p⟩ = E⟨→p⟩
      by simp
    with A2 show card (insert (x, y) E⟨→p⟩) ≤ 1
      by simp
  qed
qed

lemma edge-doesnt-vanish-in-growing-tree :
  assumes is-tree P E and qa ∈ P and card (E⟨→qa⟩) = 1 and is-tree (insert
q P) (insert (p, q) E)
    and qa ≠ p and qa ≠ q
  shows card (insert (p, q) E⟨→qa⟩) = 1
  using assms
proof −
  have before-le-1 : card (E⟨→qa⟩) ≤ 1  by (simp add: assms(3))
 have after-le-1 : card (insert (p, q) E⟨→qa⟩) ≤ 1 using assms(4) at-most-one-parent-in-tree
by presburger
  have at-least-1 : card (E⟨→qa⟩) = 1 by (simp add: assms(3))
  then show card (insert (p, q) E⟨→qa⟩) = 1 using assms(6) by auto
qed

lemma edge-doesnt-vanish-in-growing-tree2 :
  assumes card (E⟨→qa⟩) = 1 and p ≠ qa and q ≠ qa
  shows card (insert (p, q) E⟨→qa⟩) = 1
  using assms(1,3) by auto

lemma tree-acyclic:
  fixes P :: 'peer set
    and E :: 'peer topology
  assumes is-tree P E and (p,q) ∈ E
  shows (q,p) ∉ E
  using assms
proof(induct rule: is-tree.induct)
  case (ITRoot p)
  then show ?case by simp
next
  case (ITNode P E p q)
  then show ?case using edge-on-peers-in-tree(1) by blast
qed
```

lemma *tree-acyclic-gen*:
  fixes P :: *'peer set*
    and E :: *'peer topology*
  assumes *is-tree* P E **and** $(p,q) \in E$ **and** $E\langle\rightarrow p\rangle = \{\} \vee E\langle\rightarrow p\rangle = \{x\}$ **and** $x \neq y$
  shows $(y,p) \notin E$
  using *assms(3,4)* **by** *fastforce*

lemma *root-exists*:
  fixes P :: *'peer set*
    and E :: *'peer topology*
  assumes *is-tree* P E
  shows $\exists p.\ p \in P \wedge E\langle\rightarrow p\rangle = \{\}$
  using *assms*
proof (*induct*)
  case (*ITRoot* p)
  then show *?case* **by** *simp*
next
  case (*ITNode* P E p q)
  then obtain p' **where** p'*-def*: $p' \in P \wedge E\langle\rightarrow p'\rangle = \{\}$ **by** *blast*
  then have *new-tree*: *is-tree* (*insert* q P) (*insert* $(p,\ q)$ E) **by** (*simp add*: *ITNode.hyps(1,3,4)* *is-tree.ITNode*)
  then have p'*-not-q*: $p' \neq q$ **using** *ITNode.hyps(4)* p'*-def* **by** *auto*
  then have *is-tree* (*insert* q P) (*insert* $(p',\ q)$ E) **by** (*simp add*: *ITNode.hyps(1,4)* *is-tree.ITNode* p'*-def*)
  then have *t2*: (*insert* $(p',\ q)$ $E)\langle\rightarrow p'\rangle = \{\}$ **by** (*simp add*: p'*-def* p'*-not-q*)
  then have *t1*: $p' \in$ (*insert* q P) **using** p'*-def* **by** *auto*
  then have $p' \in$ (*insert* q P) $\wedge$ (*insert* $(p',\ q)$ $E)\langle\rightarrow p'\rangle = \{\}$ **using** *t2* **by** *auto*
  then have $p' \in$ (*insert* q P) $\wedge$ (*insert* $(p,\ q)$ $E)\langle\rightarrow p'\rangle = \{\}$ **by** *blast*
  then show *?case* **by** *blast*
qed

lemma *at-most-one-parent*:
  assumes *is-tree* P E
  shows *card* $(E\langle\rightarrow q\rangle) \leq 1$
  using *assms at-most-one-parent-in-tree* **by** *auto*

lemma *unique-root*:
  fixes P :: *'peer set*
    and E :: *'peer topology*
  assumes *is-tree* P E **and** $p \in P$ **and** $E\langle\rightarrow p\rangle = \{\}$ **and** $q \neq p$ **and** $q \in P$
  shows (*card* $(E\langle\rightarrow q\rangle)) = 1$
  using *assms*
proof (*induct* P E *rule*: *is-tree.induct*)
  case (*ITRoot* p)
  then show *?case* **by** *simp*
next
  case (*ITNode* P E p' q')

```
  then have p ∈ insert q' P ∧ insert (p', q') E⟨→p⟩ = {} by blast
  then have E⟨→p⟩ = {} by simp
  then show card (insert (p', q') E⟨→q⟩) = 1
  proof (cases card (E⟨→q⟩) = 1)
    case True
    then show ?thesis
     by (smt (verit) Collect-cong ITNode.hyps(1,4) card-1-singletonE edge-on-peers-in-tree(2)
          empty-def insert-iff insert-not-empty prod.inject)
  next
    case False
    have is-tree P E by (simp add: ITNode.hyps(1))
   then have q-le-1: card (E⟨→q⟩) ≤ 1 by (metis ‹is-tree P E› at-most-one-parent)
    then have q-0: card (E⟨→q⟩) = 0  using False by linarith
    then have q ∉ P
      using False ITNode.hyps(2) ITNode.prems(1,2) assms(4) by blast
    then have p ∈ P  using ITNode.prems(1,4) assms(4) by auto
    then have q = q'
      using ITNode.prems(4) ‹q ∉ P› by auto
    then have (insert (p', q') E⟨→q⟩) = (insert (p', q) E⟨→q⟩) by auto
    then have ({(p', q)}⟨→q⟩) = {p'} by auto
    then have card (insert (p', q) E⟨→q⟩) = card (E⟨→q⟩) + card {p'}
       by (smt (verit, ccfv-SIG) Collect-cong ITNode.hyps(1,4) ‹q = q'› add-0
edge-on-peers-in-tree(2)
          insert-iff q-0 singleton-iff)
    then have card (insert (p', q) E⟨→q⟩) = 1
      by (simp add: q-0)
    then show ?thesis
      using ‹insert (p', q') E⟨→q⟩ = insert (p', q) E⟨→q⟩› by fastforce
  qed
qed
```

— P? is defined on each automaton p, G is the topology graph
— This means there may be P?(p) = but p in> P!(q), thus (q,p) in> G> and q in> G>langle>rightarrow>prangle>, but q notin>

```
lemma sends-of-peer-subset-of-predecessors-in-topology:
  fixes p :: 'peer
  shows 𝒫?(p) ⊆ 𝒢⟨→p⟩
proof (cases 𝒫?(p) = {})
  case True
  then show ?thesis by simp
next
  case False
  show ?thesis
  proof
    fix q
    assume q ∈ 𝒫?(p)
   then have ∃ s1 a s2. (s1, a, s2) ∈ ℛ(p) ∧ is-input a using no-input-trans-no-recvs
by blast
```

then have $\exists$ *s1 a s2*. (*s1*, *a*, *s2*) $\in \mathcal{R}(p) \wedge$ *is-input a* $\wedge$ *get-object a* $= q$
using *CommunicatingAutomaton.ReceivingFromPeers-rev* ‹$q \in \mathcal{P}_?$ p› *automaton-of-peer*
by *fastforce*
then obtain *s1 s2 a* **where** (*s1*, *a*, *s2*) $\in \mathcal{R}(p) \wedge$ *is-input a* $\wedge$ *get-object a* $=$ $q \wedge$ *get-actor a* $= p$
by (*metis CommunicatingAutomaton.well-formed-transition automaton-of-peer*)
then have *get-message a* $\in \mathcal{M}$
by (*metis trans-to-edge*)
then have $\exists\, i.\ i^{q \to p}$ $=$ *get-message a*
using ‹*s1* $-a\rightarrow_{\mathcal{C}} p$ *s2* $\wedge$ *is-input a* $\wedge$ *get-object a* $= q \wedge$ *get-actor a* $= p$› *input-message-to-act-both-known*
by *blast*
then obtain i **where** $a = (?\langle(i^{q \to p})\rangle)$
by (*metis* ‹*s1* $-a\rightarrow_{\mathcal{C}} p$ *s2* $\wedge$ *is-input a* $\wedge$ *get-object a* $= q \wedge$ *get-actor a* $= p$› *action.exhaust get-message.simps*(*2*)
is-output.simps(*1*))
then have $(q, p) \in \mathcal{G}$
using *Edges.intros* ‹*get-message a* $\in \mathcal{M}$› **by** *force*
then show $q \in \mathcal{G}\langle\rightarrow p\rangle$
by *simp*
qed
qed

6.1.2 root node specifications and more tree related lemmas

lemma *local-to-global-root* :
assumes $\mathcal{P}_?(p) = \{\}$ **and** $(\forall\, q.\ p \notin \mathcal{P}_!(q))$ **and** *tree-topology*
shows $\mathcal{G}\langle\rightarrow p\rangle = \{\}$
using *assms*
proof *auto*
fix q
assume $(q, p) \in \mathcal{G}$
then show *False*
proof $-$
have $(q, p) \in \mathcal{G}$ **by** (*simp add*: ‹$(q, p) \in \mathcal{G}$›)
then obtain i **where** *i-def*: $i^{q \to p} \in \mathcal{M}$ **by** (*metis Edges.cases*)
then obtain *s1 a s2 x* **where** *trans*: (*s1*, *a*, *s2*) $\in$ *snd* (*snd* ($\mathcal{A}$ *x*)) $\wedge$
$(i^{q \to p})$ $=$ *get-message a* **using** *messages-used* **by** *blast*
then have $x = p \vee x = q$ **by** (*metis CommunicatingAutomaton.well-formed-transition NetworkOfCA.automaton-of-peer*
NetworkOfCA.output-message-to-act NetworkOfCA-axioms input-message-to-act-both-known
message.inject)
then have $x = q$ **by** (*metis CommunicatingAutomaton.SendingToPeers.intros assms*(*1*,*2*) *automaton-of-peer i-def*
local.trans message.inject no-recvs-no-input-trans output-message-to-act-both-known)
then have $a = !\langle(i^{q \to p})\rangle$ **by** (*metis CommunicatingAutomaton.well-formed-transition action.exhaust automaton-of-peer*
get-message.simps(*1*,*2*) *get-object.simps*(*2*) *get-sender.simps local.trans*)
then have $\neg\ (\forall\, q.\ p \notin \mathcal{P}_!(q))$ **using** *CommunicatingAutomaton.SendingToPeers.intros*

automaton-of-peer local.trans
      by *fastforce*
    then show *?thesis* **by** (*simp add*: *assms*(*2*))
  qed
qed

lemma *global-to-local-root*:
  assumes $\mathcal{G}\langle\to p\rangle = \{\}$ **and** *tree-topology*
  shows $\mathcal{P}_?(p) = \{\} \wedge (\forall q.\ p \notin \mathcal{P}_!(q))$
proof *auto*
  fix q
  assume $q \in \mathcal{P}_?\ p$
  then obtain *s1 i a s2* **where** *trans-def* : (*s1*, *a*, *s2*) ∈ *snd* (*snd* ($\mathcal{A}$ *p*))
   and $a = ?\langle(i^{q\to p})\rangle$ **by** (*metis* (*mono-tags*, *lifting*) *Collect-mem-eq Collect-mono-iff*
assms(*1*) *empty-Collect-eq*
        sends-of-peer-subset-of-predecessors-in-topology)
 then show *False* **using** ‹$q \in \mathcal{P}_?\ p$› *assms*(*1*) *sends-of-peer-subset-of-predecessors-in-topology*
by *force*
next
  fix q
  assume $p \in \mathcal{P}_!\ q$
  then have ∃ *s1 a s2*. (*s1*, *a*, *s2*) ∈ *snd* (*snd* ($\mathcal{A}$ *q*)) ∧ *is-output a* ∧ *get-object a*
= *p*
   by (*metis CommunicatingAutomaton.SendingToPeers.simps automaton-of-peer*)
  then obtain *s1 i a s2* **where** *trans-def* : (*s1*, *a*, *s2*) ∈ *snd* (*snd* ($\mathcal{A}$ *q*))
    and $a = !\langle(i^{q\to p})\rangle$
   by (*metis Edges.intros assms*(*1*) *empty-Collect-eq output-message-to-act-both-known*
        trans-to-edge)
  then show *False* **using** *Edges.simps assms*(*1*) *trans-to-edge* **by** *fastforce*
qed

lemma *edge-impl-psend-or-qrecv*:
  assumes $\mathcal{G}\langle\to p\rangle = \{q\}$ **and** *tree-topology*
  shows $(\mathcal{P}_?\ p = \{q\} \vee p \in \mathcal{P}_!(q))$
proof (*rule ccontr*)
  assume $\neg\ (\mathcal{P}_?\ p = \{q\} \vee p \in \mathcal{P}_!(q))$
  then show *False*
  proof −
    have $\mathcal{P}_?\ p \neq \{q\}$ **using** ‹$\neg\ (\mathcal{P}_?\ p = \{q\} \vee p \in \mathcal{P}_!\ q)$› **by** *auto*
    have $p \notin \mathcal{P}_!(q)$ **using** ‹$\neg\ (\mathcal{P}_?\ p = \{q\} \vee p \in \mathcal{P}_!\ q)$› **by** *auto*

    have $\exists i.\ i^{q\to p} \in \mathcal{M}$ **using** *Edges.simps* *assms*(*1*) **by** *auto*
    then obtain *i* **where** *m*: $i^{q\to p} \in \mathcal{M}$ **by** *auto*
    then have ∃ *s1 a s2 pp*. (*s1*, *a*, *s2*) ∈ *snd* (*snd* ($\mathcal{A}$ *pp*)) ∧
                          $(i^{q\to p})$ = *get-message a* **using** *messages-used* **by** *auto*
    then have ∃ *s1 a s2*. ((*s1*, *a*, *s2*) ∈ $\mathcal{R}$ *p* ∨ (*s1*, *a*, *s2*) ∈ $\mathcal{R}$ *q*) ∧
                          $(i^{q\to p})$ = *get-message a* **by** (*metis* (*mono-tags*, *lifting*)
CommunicatingAutomaton.well-formed-transition

NetworkOfCA.input-message-to-act-both-known NetworkOfCA-axioms automaton-of-peer message.inject
output-message-to-act-both-known)

then obtain *s1 a s2* **where** ((*s1*, *a*, *s2*) $\in \mathcal{R}$ *p* $\vee$ (*s1*, *a*, *s2*) $\in \mathcal{R}$ *q*) $\wedge$ ($i^{q \rightarrow p}$) = *get-message a* **by** *blast*
then show *?thesis*
proof (*cases is-output a*)
case *True*
then have (*s1*, *a*, *s2*) $\in \mathcal{R}$ *q* **by** (*metis CommunicatingAutomaton-def NetworkOfCA.automaton-of-peer NetworkOfCA.output-message-to-act-both-known*
NetworkOfCA-axioms ‹(*s1* $-a\rightarrow_{\mathcal{C}}$ *p s2* $\vee$ *s1* $-a\rightarrow_{\mathcal{C}}$ *q s2*) $\wedge$ $i^{q \rightarrow p}$ = *get-message a*› *message.inject*)
then show *?thesis* **by** (*metis CommunicatingAutomaton.SendingToPeers.intros True*
‹(*s1* $-a\rightarrow_{\mathcal{C}}$ *p s2* $\vee$ *s1* $-a\rightarrow_{\mathcal{C}}$ *q s2*) $\wedge$ $i^{q \rightarrow p}$ = *get-message a*› ‹*p* $\notin \mathcal{P}_!$ *q*› *automaton-of-peer m message.inject*
output-message-to-act-both-known)
next
case *False*
then have (*s1*, *a*, *s2*) $\in \mathcal{R}$ *p* **by** (*metis* ‹(*s1* $-a\rightarrow_{\mathcal{C}}$ *p s2* $\vee$ *s1* $-a\rightarrow_{\mathcal{C}}$ *q s2*) $\wedge$ $i^{q \rightarrow p}$ = *get-message a*› *empty-receiving-from-peers2*
input-message-to-act-both-known insert-absorb insert-not-empty m message.inject)
then have *is-input a* **by** (*simp add*: *False*)
then have $q \in \mathcal{P}_?(p)$ **by** (*metis CommunicatingAutomaton.ReceivingFromPeers.intros*
‹(*s1* $-a\rightarrow_{\mathcal{C}}$ *p s2* $\vee$ *s1* $-a\rightarrow_{\mathcal{C}}$ *q s2*) $\wedge$ $i^{q \rightarrow p}$ = *get-message a*› ‹*s1* $-a\rightarrow_{\mathcal{C}}$ *p s2*› *automaton-of-peer*
input-message-to-act-both-known m message.inject)
have $(\mathcal{P}_?(p)) = \{q\}$
proof
show $\{q\} \subseteq \mathcal{P}_?$ *p* **by** (*simp add*: ‹$q \in \mathcal{P}_?$ *p*›)

show $\mathcal{P}_?$ *p* $\subseteq \{q\}$
proof (*rule ccontr*)
assume $\neg$ $\mathcal{P}_?$ *p* $\subseteq \{q\}$
then obtain *pp* **where** *pp* $\in \mathcal{P}_?$ *p* **and** *pp* $\neq$ *q* **by** *auto*
then have *pp* $\in \mathcal{G}\langle\rightarrow p\rangle$ **using** *sends-of-peer-subset-of-predecessors-in-topology* **by** *auto*
then show *False* **by** (*simp add*: ‹*pp* $\neq$ *q*› *assms*(*1*))
qed
qed
then show *?thesis* **by** (*simp add*: ‹$\mathcal{P}_?$ *p* $\neq \{q\}$›)
qed
qed
qed

lemma *root-or-node*:
assumes *tree-topology*

shows *is-root p* $\vee$ ($\exists$ *q*. $\mathcal{P}_?(p)$ = {*q*} $\vee$ *p* $\in$ $\mathcal{P}_!(q)$)
using *assms*
proof (*cases* $\mathcal{G}\langle\rightarrow p\rangle$ = {})
case *True*
then show *?thesis* **by** (*simp add*: *assms*)
next
case *False*
then have *card* ($\mathcal{G}\langle\rightarrow p\rangle$) $\neq$ *0* **by** (*metis card-0-eq finite-peers finite-subset top-greatest*)
have *card* ($\mathcal{G}\langle\rightarrow p\rangle$) $\leq$ *1* **using** *assms at-most-one-parent* **by** *auto*
then have *card* ($\mathcal{G}\langle\rightarrow p\rangle$) = *1* **using** ‹*card* ($\mathcal{G}\langle\rightarrow p\rangle$) $\neq$ *0*› **by** *linarith*
then obtain *q* **where** $\mathcal{G}\langle\rightarrow p\rangle$ = {*q*} **using** *card-1-singletonE* **by** *blast*
then show *?thesis* **using** *assms edge-impl-psend-or-qrecv* **by** *blast*
qed

lemma *root-defs-eq*:
shows *is-root-from-topology p* = *is-root-from-local p*
using *global-to-local-root local-to-global-root* **by** *blast*

lemma *local-global-eq-node*:
assumes *is-node-from-topology p*
shows *is-node-from-local p*
using *assms edge-impl-psend-or-qrecv* **by** *auto*

lemma *global-local-eq-node*:
assumes *is-node-from-local p*
shows *is-node-from-topology p*
proof −
have *local-p*: *tree-topology* $\wedge$ ($\exists$ *q*. $\mathcal{P}_?(p)$ = {*q*} $\vee$ *p* $\in$ $\mathcal{P}_!(q)$) **by** (*simp add*: *assms*)
then have *t1*: *tree-topology* **by** *simp*
then show *?thesis* **using** *assms*
proof (*cases* $\exists$ *q*. $\mathcal{P}_?(p)$ = {*q*})
case *True*
then obtain *q* **where** $\mathcal{P}_?(p)$ = {*q*} **by** *auto*
then have *q* $\in$ $\mathcal{G}\langle\rightarrow p\rangle$ **using** *sends-of-peer-subset-of-predecessors-in-topology* **by** *auto*
have $\neg$ (*is-root p*) **using** ‹$\mathcal{P}_?$ *p* = {*q*}› ‹*q* $\in$ $\mathcal{G}\langle\rightarrow p\rangle$› **by** *blast*
have *card* ($\mathcal{G}\langle\rightarrow p\rangle$) $\leq$ *1* **using** *at-most-one-parent t1* **by** *auto*
then have *card* ($\mathcal{G}\langle\rightarrow p\rangle$) = *1* **by** (*smt* (*verit*) *Collect-cong* ‹*q* $\in$ $\mathcal{G}\langle\rightarrow p\rangle$› *edge-on-peers-in-tree*(*2*) *empty-Collect-eq empty-iff root-exists t1 unique-root*)
then show *?thesis* **by** (*meson is-singleton-altdef is-singleton-the-elem t1*)
next
case *False*
then obtain *q* **where** *p* $\in$ $\mathcal{P}_!(q)$ **using** *local-p* **by** *auto*
then obtain *s1 a s2* **where** *is-output a* **and** *get-actor a* = *q* **and** *get-object a* = *p* **and** (*s1*,*a*,*s2*) $\in$ $\mathcal{R}$ *q*

by (*meson CommunicatingAutomaton.SendingToPeers-rev CommunicatingAutomaton.well-formed-transition automaton-of-peer*)
then have $q \in \mathcal{G}\langle\rightarrow p\rangle$ **by** (*metis Edges.intros mem-Collect-eq output-message-to-act-both-known trans-to-edge*)
have *card* $(\mathcal{G}\langle\rightarrow p\rangle) \leq 1$ **using** *at-most-one-parent t1* **by** *auto*
then have *card* $(\mathcal{G}\langle\rightarrow p\rangle) = 1$ **by** (*smt* (*verit*) *Collect-cong* ‹$q \in \mathcal{G}\langle\rightarrow p\rangle$› *edge-on-peers-in-tree*(*2*) *empty-Collect-eq empty-iff root-exists t1 unique-root*)
then show *?thesis* **by** (*meson is-singleton-altdef is-singleton-the-elem t1*)
qed
qed

lemma *node-defs-eq*:
shows *is-node-from-topology p* = *is-node-from-local p*
using *edge-impl-psend-or-qrecv global-local-eq-node* **by** *blast*

6.1.3 parent-child relationship in tree

lemma *is-parent-of-rev*:
assumes *is-parent-of p q*
shows *is-node p* **and** $\mathcal{G}\langle\rightarrow p\rangle = \{q\}$
using *assms*
proof (*cases rule*: *is-parent-of.cases*)
case *node-parent*
then show *is-node p* **by** *simp*
next
have *is-node p* **by** (*metis assms is-parent-of.cases*)
then show $\mathcal{G}\langle\rightarrow p\rangle = \{q\}$ **by** (*metis assms is-parent-of.cases*)
qed

lemma *is-parent-of-rev2*:
assumes *is-parent-of p q*
shows *is-node p* **and** $\mathcal{P}_?(p) = \{q\} \vee p \in \mathcal{P}_!(q)$
using *assms*
proof (*cases rule*: *is-parent-of.cases*)
case *node-parent*
then show *is-node p* **by** *simp*
next
have *is-node p* **by** (*metis assms is-parent-of.cases*)
then show $\mathcal{P}_?(p) = \{q\} \vee p \in \mathcal{P}_!(q)$ **using** *assms edge-impl-psend-or-qrecv is-parent-of-rev*(*2*) **by** *blast*
qed

lemma *local-parent-to-global*:
assumes *tree-topology* **and** $\mathcal{P}_?(p) = \{q\} \vee p \in \mathcal{P}_!(q)$
shows $\mathcal{G}\langle\rightarrow p\rangle = \{q\}$
proof −
show *?thesis* **using** *assms*

proof (*cases* $\mathcal{P}_?(p) = \{q\}$)
case *True*
then have $q \in \mathcal{G}\langle\to p\rangle$ **using** *sends-of-peer-subset-of-predecessors-in-topology* **by** *auto*
have $\neg$ (*is-root p*) **using** ‹$\mathcal{P}_?$ $p = \{q\}$› ‹$q \in \mathcal{G}\langle\to p\rangle$› **by** *blast*
have *card* $(\mathcal{G}\langle\to p\rangle) \leq 1$ **using** *at-most-one-parent assms* **by** *auto*
then have *card* $(\mathcal{G}\langle\to p\rangle) = 1$ **by** (*smt* (*verit*) *Collect-cong* ‹$q \in \mathcal{G}\langle\to p\rangle$› *edge-on-peers-in-tree*(*2*) *empty-Collect-eq empty-iff root-exists assms unique-root*)
then show *?thesis* **by** (*metis* ‹$q \in \mathcal{G}\langle\to p\rangle$› *card-1-singletonE singletonD*)
next
case *False*
then have $p \in \mathcal{P}_!(q)$ **using** *assms* **by** *auto*
then obtain *s1 a s2* **where** *is-output a* **and** *get-actor* $a = q$ **and** *get-object* $a = p$ **and** $(s1,a,s2) \in \mathcal{R}\ q$
by (*meson CommunicatingAutomaton.SendingToPeers-rev CommunicatingAutomaton.well-formed-transition automaton-of-peer*)
then have *c1*: $q \in \mathcal{G}\langle\to p\rangle$ **by** (*metis Edges.intros mem-Collect-eq output-message-to-act-both-known trans-to-edge*)
have *c2*: *card* $(\mathcal{G}\langle\to p\rangle) \leq 1$ **using** *at-most-one-parent assms* **by** *auto*
have *c3*: *finite* $(\mathcal{G}\langle\to p\rangle)$ **using** *finite-peers rev-finite-subset* **by** *fastforce*
from *c3 c1 c2* **have** *card* $(\mathcal{G}\langle\to p\rangle) = 1$ **using** *assms*(*1*) *root-exists unique-root* **by** *force*
then show *?thesis* **by** (*metis c1 card-1-singletonE singleton-iff*)
qed
qed

lemma *parent-child-diff*:
assumes *is-parent-of p q*
shows $p \neq q$
proof (*rule ccontr*)
assume $\neg\ p \neq q$
then have *is-parent-of p p* **using** *assms* **by** *auto*
then have *is-node* $p \wedge \mathcal{G}\langle\to p\rangle = \{p\}$ **using** *is-parent-of-rev*(*2*) *is-parent-of-rev2*(*1*) **by** *force*
then show *False* **by** (*metis insert-iff mem-Collect-eq tree-acyclic*)
qed

lemma *child-word-filters-unique-parent*:
assumes *is-parent-of p q* **and** $w \in \mathcal{L}(p)$
shows (*filter* (λx. *get-object* $x = q$) $(w\downarrow_?)) = (w\downarrow_?)$
using *assms*
proof (*induct length w arbitrary*: *w*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc x*)
then obtain *a v* **where** *w-def*: $w = v$ @ $[a]$ **and** *length* $v = x$ **by** (*metis*

length-Suc-conv-rev)
  then have $v \in \mathcal{L}(p)$ **using** *Lang-app Suc.prems*(*2*) **by** *blast*
  then have *filter* (λx. *get-object* $x = q$) ($v\downarrow_?$) $= v\downarrow_?$ **using** *Suc.hyps*(*1*) ‹$|v| = x$› *assms*(*1*) **by** *blast*
  have (v @ [a]) $\in \mathcal{L}$ p **using** *Suc.prems*(*2*) *w-def* **by** *auto*
  then have $\exists$ *s1 s2*. (*s1*, a, *s2*) $\in \mathcal{R}$ p **using** *Lang-app-both lang-implies-trans* **by** *blast*
  then obtain *s1 s2* **where** (*s1*, a, *s2*) $\in \mathcal{R}$ p **by** *blast*
  then have *get-actor* $a = p$ **by** (*meson CommunicatingAutomaton.well-formed-transition NetworkOfCA.automaton-of-peer*
        NetworkOfCA-axioms)
  then show *?case* **using** *Suc*
  proof (*cases is-input a*)
    case *True*
    then have [a]$\downarrow_?$ = [a] **by** *simp*
    then show *?thesis* **using** *True*
    proof (*cases get-object* $a = q$)
      case *True*
      have ($w\downarrow_?$) = (v @ [a])$\downarrow_?$ **by** (*simp add*: *w-def*)
      then have (v @ [a])$\downarrow_?$ = ($v\downarrow_?$) @ [a] **using** ‹(a # ε)$\downarrow_?$ = a # ε› **by** *force*
      then have *obj-proj-decomp*: (*filter* (λx. *get-object* $x = q$) ($w\downarrow_?$)) = (*filter* (λx. *get-object* $x = q$) ($v\downarrow_?$)) @ (*filter* (λx. *get-object* $x = q$) ([a]))
        using *w-def* **by** *force*
      then show *?thesis* **using** *True* ‹*filter* (λx. *get-object* $x = q$) ($v\downarrow_?$) = $v\downarrow_?$› *w-def* **by** *fastforce*
    next
      case *False*
      then obtain *qq* **where** *get-object* a = *qq* **and** *qq* $\neq$ q **by** *simp*
      then have *qq* $\in \mathcal{G}\langle\rightarrow p\rangle$ **by** (*metis Edges.intros True* ‹*get-actor* $a = p$› ‹*s1* $-a\rightarrow_{\mathcal{C}} p$ *s2*› *input-message-to-act-both-known mem-Collect-eq*
          trans-to-edge)
      then have *qq* $\in \mathcal{P}$ **by** *auto*
      have $q \in \mathcal{G}\langle\rightarrow p\rangle$ **using** *assms*(*1*) *is-parent-of-rev*(*2*) **by** *auto*
      then have $\mathcal{G}\langle\rightarrow p\rangle \neq \{q\}$ **using** ‹*qq* $\in \mathcal{G}\langle\rightarrow p\rangle$› ‹*qq* $\neq$ q› **by** *blast*
      then show *?thesis* **using** *assms*(*1*) *is-parent-of-rev*(*2*) **by** *auto*
    qed
  next
    case *False*
    then have *is-output a* **by** *auto*
    then have [a]$\downarrow_?$ = ε **by** *simp*
    then have ($w\downarrow_?$) = ($v\downarrow_?$) **using** *w-def* **by** *fastforce*
    then show *?thesis* **using** ‹*filter* (λx. *get-object* $x = q$) ($v\downarrow_?$) = $v\downarrow_?$› **by** *presburger*
  qed
qed

lemma *pair-proj-recv-for-unique-parent*:
  assumes *is-parent-of* p q **and** $w \in \mathcal{L}(p)$
  shows ($w\downarrow_?$)$\downarrow_{\{p,q\}}$ = ($w\downarrow_?$)

proof −
have $((w)\downarrow_p) = w$ **using** *assms(2) w-in-peer-lang-impl-p-actor* **by** *auto*
then have $((w\downarrow_p)\downarrow_?) = (w\downarrow_?)$ **by** *presburger*
then have $((w\downarrow_?)\downarrow_p) = (w\downarrow_?)$ **by** (*metis filter-pair-commutative*)
then have $(w\downarrow_?)\downarrow_{\{p,q\}} = (\mathit{filter}\ (\lambda x.\ \mathit{get\text{-}object}\ x = q)\ (w\downarrow_?))$ **using** *pair-proj-to-object-proj* **by** *fastforce*
have $(\mathit{filter}\ (\lambda x.\ \mathit{get\text{-}object}\ x = q)\ (w\downarrow_?)) = (w\downarrow_?)$ **using** *assms child-word-filters-unique-parent* **by** *auto*
then show *?thesis* **using** ‹$w\downarrow_?\downarrow_{\{p,q\}} = \mathit{filter}\ (\lambda x.\ \mathit{get\text{-}object}\ x = q)\ (w\downarrow_?)$› **by** *presburger*
qed

lemma *filter-ignore-false-prop*:
assumes $\mathit{filter}\ (\lambda x.\ \mathit{False})\ w = \varepsilon$
shows $\mathit{filter}\ (\lambda x.\ \mathit{False} \lor B)\ w = \mathit{filter}\ (\lambda x.\ B)\ w$
by (*metis assms filter-False filter-True*)

lemma *recv-lang-child-pair-proj-subset1*:
assumes *is-parent-of p q*
shows $(((\mathcal{L}(p))\downharpoonleft_?)) \subseteq ((((\mathcal{L}(p))\downharpoonleft_?)\downharpoonleft_{\{p,q\}}))$
proof *auto*
fix w
show $w \in \mathcal{L}\ p \Longrightarrow \exists\, wa.\ w\downarrow_? = wa\downarrow_{\{p,q\}} \land (\exists\, w.\ wa = w\downarrow_? \land w \in \mathcal{L}\ p)$ **by** (*metis* (*no-types*, *lifting*) *assms pair-proj-recv-for-unique-parent*)
qed

lemma *child-recv-lang-inv-to-proj-with-parent*:
assumes *is-parent-of p q*
shows $(((\mathcal{L}(p))\downharpoonleft_?)) = ((((\mathcal{L}(p))\downharpoonleft_?)\downharpoonleft_{\{p,q\}}))$
proof −
have *t1*: $(((\mathcal{L}(p))\downharpoonleft_?)) \subseteq ((((\mathcal{L}(p))\downharpoonleft_?)\downharpoonleft_{\{p,q\}}))$ **using** *assms recv-lang-child-pair-proj-subset1* **by** *blast*
have *t2*: $((((\mathcal{L}(p))\downharpoonleft_?)\downharpoonleft_{\{p,q\}})) \subseteq (((\mathcal{L}(p))\downharpoonleft_?))$ **by** (*smt* (*z3*) *Collect-mono-iff filter-recursion mem-Collect-eq t1*)
from *t1 t2* **show** *?thesis* **by** *blast*
qed

6.1.4 path to root and path related lemmas

lemma *path-to-root-rev*:
assumes *path-to-root p ps* **and** $ps \neq [p]$
shows $\exists\, q\ as.\ \mathit{is\text{-}parent\text{-}of}\ p\ q \land \mathit{path\text{-}to\text{-}root}\ q\ as \land ps = (p\ \#\ as) \land \mathit{distinct}\ (p\ \#\ as)$
using *assms*
by (*meson path-to-root.simps*)

```
lemma path-to-root-rev-empty:
  assumes path-to-root p ps and ps = [p]
  shows is-root p
 by (metis (no-types, lifting) assms(1,2) list.distinct(1) list.inject path-to-root.simps)

lemma path-ends-at-root:
  assumes path-to-root p ps
  shows  is-root (last ps)
  using assms
proof (induct rule: path-to-root.induct)
  case (PTRRoot p)
  then show ?case by auto
next
  case (PTRNode p q as)
  then show ?case by (metis last-ConsR list.discI path-to-root.cases)
qed

lemma single-path-impl-root:
  assumes path-to-root p [p]
  shows is-root p
  using assms path-to-root-rev-empty by blast


lemma path-to-root-first-elem-is-peer:
  assumes  path-to-root p (x # ps)
  shows  p = x
  using assms path-to-root-rev by auto

lemma path-to-root-stepback:
  assumes path-to-root p (p # ps)
  shows ps = [] ∨ (∃ q. is-parent-of p q ∧ path-to-root q ps)
  using assms path-to-root-rev by auto

lemma path-to-root-unique:
  assumes path-to-root p ps and path-to-root p qs
  shows qs = ps
  using assms
proof (induct p ps arbitrary: qs rule: path-to-root.induct)
  case (PTRRoot p)
 then show ?case by (metis (mono-tags, lifting) ITRoot empty-iff is-parent-of.cases
local-to-global-root path-to-root.simps
        root-exists)
next
  case (PTRNode p q as)
  then have path-to-root p (p # as) using path-to-root.PTRNode by blast
  then have ∀ ys. (path-to-root q ys) ⟶ as = ys  using PTRNode.hyps(4) by
auto
  then have pq: is-parent-of p q by (simp add: PTRNode.hyps(2))
 then have as ≠ qs  by (metis PTRNode.hyps(3) PTRNode.prems ‹∀ ys. path-to-root
```

$q\ ys \longrightarrow as = ys$› ‹*path-to-root* p (p # *as*)›
list.inject not-Cons-self2 path-to-root-rev)
have $qs \neq []$ **using** *path-to-root.cases PTRNode.prems* **by** *auto*
then obtain *x qqs* **where** *qs-decomp*: $qs = x$ # *qqs* **using** *list.exhaust* **by** *auto*
then have *path-to-root* p (x # *qqs*) **using** *PTRNode.prems* **by** *auto*
then have $x = p$ **using** *path-to-root-first-elem-is-peer* **by** *auto*
then have $qs = p$ # *qqs* **by** (*simp add*: *qs-decomp*)
then have $qqs = [] \vee (\exists\, y.$ *is-parent-of* $p\ y \wedge$ *path-to-root* y *qqs*) **using** ‹*path-to-root* p (x # *qqs*)› ‹$x = p$› *path-to-root-stepback* **by** *auto*
then have $qqs \neq []$ **using** *pq* **using** ‹*path-to-root* p (x # *qqs*)› ‹$x = p$› *is-parent-of-rev*(*2*) *root-defs-eq single-path-impl-root*
by *fastforce*
then have ($\exists\, y.$ *is-parent-of* $p\ y \wedge$ *path-to-root* y *qqs*) **using** ‹$qqs = \varepsilon \vee (\exists\, y.$ *is-parent-of* $p\ y \wedge$ *path-to-root* y *qqs*)› **by** *auto*
then obtain y **where** *is-parent-of* $p\ y \wedge$ *path-to-root* y *qqs* **by** *auto*
then have *is-parent-of* $p\ q \wedge$ *is-parent-of* $p\ y$ **by** (*simp add*: *pq*)
then have $\mathcal{G}\langle\to p\rangle = \{q\} \wedge \mathcal{G}\langle\to p\rangle = \{y\}$ **using** *is-parent-of-rev*(*2*) **by** *auto*
then have $q = y$ **by** *blast*
then have *is-parent-of* $p\ q \wedge$ *path-to-root* q *qqs* **by** (*simp add*: ‹*is-parent-of* $p\ y \wedge$ *path-to-root* y *qqs*›)
then show *?case* **by** (*simp add*: *PTRNode.hyps*(*4*) ‹$qs = p$ # *qqs*›)
qed

lemma *peer-on-path-unique*:
assumes *path-to-root p ps*
shows *distinct ps*
using *assms distinct-singleton path-to-root-rev* **by** *fastforce*

lemma *only-peer-impl-root*:
assumes *is-tree* ($\mathcal{P}$) ($\mathcal{G}$) **and** ($\mathcal{P}$) = $\{p\}$
shows *is-root p*
by (*metis assms*(*1*,*2*) *root-exists singleton-iff*)

lemma *leaf-exists*:
assumes *tree-topology*
shows $\exists\, q.\ q \in \mathcal{P} \wedge \mathcal{G}\langle q\to\rangle = \{\}$
using *assms*
proof (*induct*)
case (*ITRoot p*)
then show *?case* **by** *simp*
next
case (*ITNode P E p q*)
then show *?case* **using** *edge-on-peers-in-tree*(*1*) *prod.inject* **by** *fastforce*
qed

lemma *leaf-root-or-child*:
assumes *tree-topology* **and** $q \in \mathcal{P} \wedge \mathcal{G}\langle q\to\rangle = \{\}$
shows *is-root* $q \vee (\exists\, p.$ *is-parent-of* $q\ p$)
using *assms*(*1*) *is-parent-of.simps node-defs-eq root-or-node* **by** *presburger*

lemma *finite-set-card-union-with-singleton*:
  assumes *finite A* **and** $a \in A$ **and** *card A* $\leq$ *1*
  shows $A = \{a\}$
proof (*rule ccontr*)
  assume $A \neq \{a\}$
  have $A \neq \{\}$ **using** *assms*(*2*) **by** *auto*
  then show *False* **by** (*metis One-nat-def* ‹$A \neq \{a\}$› *assms*(*1*,*2*,*3*) *card-0-eq card-1-singleton-iff less-Suc0 linorder-le-less-linear*
    order-antisym-conv singletonD)
qed

lemma *tree-impl-finite-sets*:
  assumes *tree-topology*
  shows *finite* ($\mathcal{P}$) **and** *finite* ($\mathcal{G}$)
proof –
  show *finite* ($\mathcal{P}$) **by** (*simp add*: *finite-peers*)
  show *finite* ($\mathcal{G}$) **by** (*meson UNIV-I finite-peers finite-prod finite-subset subsetI*)
qed

lemma *leaf-ingoing-edge*:
  assumes *tree-topology* **and** *card* ($\mathcal{P}$) $\geq$ *2* **and** $q \in \mathcal{P} \wedge \mathcal{G}\langle q \rightarrow \rangle = \{\}$
  shows $\exists p.\ \mathcal{G}\langle \rightarrow q \rangle = \{p\}$
  using *assms*
proof (*induct arbitrary*: *q*)
  case (*ITRoot p*)
  then show *?case* **by** *simp*
next
  case (*ITNode P E x y*)
  then show *?case* **using** *ITNode*
  proof (*cases* $q \in P \wedge E\langle q \rightarrow \rangle = \{\}$)
    case *True*
    then have *IH-q*: *2* $\leq$ *card P* $\Longrightarrow q \in P \wedge E\langle q \rightarrow \rangle = \{\} \Longrightarrow \exists p.\ E\langle \rightarrow q \rangle = \{p\}$ **using** *ITNode.hyps*(*2*) **by** *presburger*
    have $y \neq q$ **using** *ITNode.hyps*(*4*) *True* **by** *auto*
    then show *?thesis*
    proof (*cases 2* $\leq$ *card P*)
      case *True*
      then have $\exists p.\ E\langle \rightarrow q \rangle = \{p\}$ **using** *IH-q ITNode.prems*(*2*) ‹$y \neq q$› **by** *auto*
      have *insert* (*x*, *y*) $E\langle \rightarrow q \rangle = E\langle \rightarrow q \rangle$ **using** ‹$y \neq q$› **by** *blast*
      then show *?thesis* **by** (*simp add*: ‹$\exists p.\ E\langle \rightarrow q \rangle = \{p\}$›)
    next
      case *False*
      then have *1* $\geq$ *card P* **by** *simp*
      have $q \in P$ **by** (*simp add*: *True*)
      have *is-tree P E* **by** (*simp add*: *ITNode.hyps*(*1*))
      then have *finite P* $\wedge$ *finite E* **by** (*metis UNIV-I finite-peers finite-prod finite-subset subsetI*)
      then have *finite P* **by** *blast*

then have *cq*: *card P* = *1* **by** (*metis ITNode.hyps*(*3*) ‹*card P* ≤ *1*› *finite-set-card-union-with-singleton is-singletonI is-singleton-altdef*)
then have *card P* = *1* ∧ *q* ∈ *P* **by** (*simp add*: ‹*q* ∈ *P*›)
then have {*q*} = *P* **by** (*metis* ‹*card P* ≤ *1*› ‹*finite P*› *finite-set-card-union-with-singleton*)
then show *?thesis* **using** *ITNode.hyps*(*3*) *ITNode.prems*(*2*) **by** *blast*
qed
next
case *False*
then have *y* = *q* **using** *ITNode.prems*(*2*) **by** *auto*
then have $E\langle\rightarrow q\rangle = \{\}$ **using** *ITNode.hyps*(*1*,*4*) *edge-on-peers-in-tree*(*2*) **by** *auto*
then have $\forall g.\ (g, q) \notin E$ **by** *simp*
then have *insert* (*x*, *q*) $E\langle\rightarrow q\rangle = E\langle\rightarrow q\rangle \cup \{x\}$ **by** *simp*
then have *insert* (*x*, *q*) $E\langle\rightarrow q\rangle = \{x\}$ **by** (*simp add*: ‹$E\langle\rightarrow q\rangle = \{\}$›)
then show *?thesis* **using** ‹*y* = *q*› **by** *auto*
qed
qed

lemma *app-path-peer-is-parent-or-root*:
assumes *path-to-root p* (*xs* @ [*q*] @ *ys*) **and** *xs* ≠ []
shows *is-root q* ∨ (∃ *qq*. *is-parent-of qq q*)
using *assms*
proof (*induct p xs* @ [*q*] @ *ys arbitrary*: *xs q ys*)
case (*PTRRoot p*)
then have *p* = *q* **by** (*metis* (*no-types*, *lifting*) *Nil-is-append-conv append-eq-Cons-conv list.distinct*(*1*))
then have *is-root q* **using** *PTRRoot.hyps*(*1*) **by** *auto*
then show *?case* **by** *blast*
next
case (*PTRNode x y as*)
then show *?case*
proof (*cases* ∃ *xs ys*. *as* = (*xs* • (*q* # ε • *ys*)))
case *True*
then show *?thesis* **by** (*metis Cons-eq-appendI*[*of q* ε *q* # ε ε • -] *PTRNode.hyps*(*2*,*3*) *PTRNode.hyps*(*4*)[*of* - *q*] *list.inject*[*of q* - *y*] *path-to-root.cases*[*of y as*] *self-append-conv2*[*of* - ε])
next
case *False*
then have ∀ *xs ys*. *as* ≠ (*xs* • (*q* # ε • *ys*)) **by** *simp*
then have *q* ≠ *x* **by** (*metis PTRNode.hyps*(*6*) *PTRNode.prems append-eq-Cons-conv*)
then have *q* ≠ *y* **by** (*metis Cons-eq-appendI False PTRNode.hyps*(*3*) *eq-Nil-appendI path-to-root-rev*)
then have ∀ *xs ys*. (*x*#*as*) ≠ (*xs* • (*q* # ε • *ys*)) **by** (*metis PTRNode.hyps*(*6*) *PTRNode.prems* ‹∀ *xs ys*. *as* ≠ *xs* • (*q* # ε • *ys*)› *append-eq-Cons-conv*)
then show *?thesis* **using** *PTRNode.hyps*(*6*) **by** *auto*
qed
qed

lemma *app-path-peer-is-parent-or-root2*:
  assumes *path-to-root p ps* **and** *ps*!*i* = *q* **and** *i* < *length ps*
  shows *is-root q* ∨ *is-parent-of q* (*ps*!(*Suc i*))
  using *assms*
proof (*induct p ps arbitrary*: *i q*)
  case (*PTRRoot p*)
  then show *?case* **using** *Suc-length-conv append-self-conv2* **by** *auto*
next
  case (*PTRNode x y as*)
  then show *?case*
  proof (*cases i* = *0*)
    case *True*
    then have *x* = *q* **using** *PTRNode.prems*(*1*) **by** *auto*
    then have *is-parent-of q y* **using** *PTRNode.hyps*(*2*) **by** *auto*
    then show *?thesis* **by** (*metis PTRNode.hyps*(*3*) *True nth-Cons-0 nth-Cons-Suc path-to-root.simps*)
  next
    case *False*
    then have *i* ≥ *1* **by** *auto*
    then have *as*!(*i*−*1*) = *q* **using** *PTRNode.prems*(*1*) **by** *auto*
    then have (*i*−*1*) < *length as* **by** (*metis One-nat-def PTRNode.prems*(*2*) *Suc-pred* ‹*1* ≤ *i*› *le-less-Suc-eq length-Cons less-imp-diff-less*
      less-numeral-extra(*1*) *linorder-le-less-linear order.strict-trans2*)
    then have *is-root q* ∨ *is-parent-of q* (*as*!*i*) **by** (*metis One-nat-def PTRNode.hyps*(*4*) *Suc-pred UNIV-def* ‹*1* ≤ *i*› ‹*as* ! (*i* − *1*) = *q*› *less-eq-Suc-le*
      root-defs-eq)
    then show *?thesis* **by** *simp*
  qed
qed

lemma *path-to-root-of-root-exists*:
  assumes *is-root p*
  shows *path-to-root p* [*p*]
  using *PTRRoot assms* **by** *auto*

lemma *adj-in-path-parent-child*:
  assumes *path-to-root p* (*x* # *y* # *ps*)
  shows $\mathcal{P}_?(x) = \{y\} \lor x \in \mathcal{P}_!(y)$
  by (*metis assms is-parent-of-rev2*(*2*) *neq-Nil-conv path-to-root-first-elem-is-peer*
    path-to-root-stepback)

6.1.5 path from root downwards to a node

lemma *path-to-root-downwards*:
  assumes *path-to-root q qs* **and** *is-parent-of p q*
  shows *path-to-root p* (*p* # *qs*)
  using *assms*
proof (*induct q qs arbitrary*: *p*)
  case (*PTRRoot p*)

then show *?case* **by** (*metis* (*lifting*) *NetworkOfCA.PTRNode NetworkOfCA-axioms distinct-length-2-or-more*
distinct-singleton empty-iff is-parent-of.simps local-to-global-root path-to-root-of-root-exists singletonI)
next
case (*PTRNode x y as*)
then have *path-to-root x* (*x* # *as*) **by** *blast*
then have *tree-topology* ∧ *is-parent-of p x* ∧ *path-to-root x* (*x*#*as*) **using** *PTRNode.hyps*(*1*) *PTRNode.prems* **by** *auto*
have *p* ≠ *x* **by** (*metis PTRNode.hyps*(*2*,*3*,*5*) *PTRNode.prems distinct-length-2-or-more is-parent-of-rev*(*2*) *path-to-root-rev*
singleton-inject)
have *distinct* (*p*#*x*#*as*)
proof (*rule ccontr*)
assume ¬ *distinct* (*p* # *x* # *as*)
then have ¬ *distinct* (*p* # *as*) **using** *PTRNode.hyps*(*5*) ‹*p* ≠ *x*› **by** *auto*
then have ∃ *i*. *as*!*i* = *p* ∧ *i* < *length as* **by** (*meson PTRNode.hyps*(*5*) *distinct.simps*(*2*) *in-set-conv-nth*)
then obtain *i* **where** *as*!*i* = *p* **and** *i* < *length as* **by** *blast*
then show *False*
proof (*cases last as* = *p*)
case *True*
then have *is-root p* **using** *PTRNode.hyps*(*3*) *path-ends-at-root* **by** *auto*
then show *?thesis* **using** *PTRNode.prems is-parent-of-rev*(*2*) *local-to-global-root* **by** *force*
next
case *False*
then have *path-to-root y as* ∧ *as*!*i* = *p* ∧ *i* < *length as* **by** (*simp add*: *PTRNode.hyps*(*3*) ‹*as* ! *i* = *p*› ‹*i* < |*as*|›)
then have *is-root p* ∨ *is-parent-of p* (*as*!(*Suc i*)) **using** *app-path-peer-is-parent-or-root2* **by** *blast*
then have *is-parent-of p* (*as*!(*Suc i*)) **by** (*metis PTRNode.prems insert-not-empty is-parent-of.simps is-parent-of-rev2*(*2*))
then have *c1*: *is-node p* ∧ $\mathcal{G}\langle\rightarrow p\rangle$ = {(*as*!(*Suc i*))} **using** *PTRNode.hyps*(*1*) *is-parent-of-rev*(*2*) **by** *auto*
have *x* ∉ *set as* **using** *PTRNode.hyps*(*5*) **by** *auto*
have ∀ *j*. *j* < *length as* ⟶ *as*!*j* ≠ *x* **using** ‹*x* ∉ *set as*› **by** *auto*
have *c3*: (*as*!(*Suc i*)) ≠ *x* **by** (*metis False Suc-lessI* ‹∀ *j*<|*as*|. *as* ! *j* ≠ *x*› ‹¬ *distinct* (*p* # *as*)› ‹*as* ! *i* = *p*› ‹*i* < |*as*|› *append1-eq-conv*
append-butlast-last-id distinct-singleton length-Suc-conv-rev nth-append-length)
have *is-parent-of p x* **by** (*simp add*: *PTRNode.prems*)
then have *c2*: *is-node p* ∧ $\mathcal{G}\langle\rightarrow p\rangle$ = {*x*} **using** *PTRNode.hyps*(*1*) *is-parent-of-rev*(*2*) **by** *auto*
then show *?thesis* **using** *c1 c2 c3* **by** *simp*
qed
qed
then show *?case* **using** ‹*is-tree* ($\mathcal{P}$) ($\mathcal{G}$) ∧ *is-parent-of p x* ∧ *path-to-root x* (*x* # *as*)› *path-to-root.PTRNode* **by** *blast*
qed

lemma *path-from-root-rev*:
  assumes *path-from-root p ps*
  shows *is-root p* $\vee$ ($\exists$ *q as*. *tree-topology* $\wedge$ *is-parent-of p q* $\wedge$ *path-from-root q as* $\wedge$ *distinct* (*as* @ [*p*]))
  by (*metis assms path-from-root.cases*)

lemma *path-to-from*:
  assumes *path-to-root p ps*
  shows *path-from-root p* (*rev ps*)
  using *assms*
proof (*induct*)
  case (*PTRRoot p*)
  then show *?case* **using** *PFRRoot* **by** *force*
next
  case (*PTRNode p q as*)
  then show *?case* **using** *PFRNode PTRNode.hyps*(*1*,*2*,*4*,*5*) **by** *force*
qed

lemma *path-from-to*:
  assumes *path-from-root p ps*
  shows *path-to-root p* (*rev ps*)
  using *assms*
proof (*induct*)
  case (*PFRRoot p*)
  then show *?case* **using** *PTRRoot* **by** *force*
next
  case (*PFRNode p q as*)
  then show *?case* **using** *PTRNode PFRNode.hyps*(*1*,*2*,*4*,*5*) **by** *force*
qed

lemma *paths-eq*:
  shows ($\exists$ *ps*. *path-from-root p ps*) = ($\exists$ *qs*. *path-to-root p qs*)
  using *path-from-to path-to-from* **by** *blast*

lemma *path-from-to-rev*:
  assumes *path-from-to r p r2p*
  shows (*r* = *p*) $\vee$ ($\exists$ *q qs*. *path-from-to r q qs* $\wedge$ *r2p* = (*qs*@[*p*]) $\wedge$ *is-parent-of p q*)
  by (*metis assms path-from-to.simps*)

lemma *path-from-root-2-path-from-to*:
  assumes *path-from-root p ps* **and** *is-root r*
  shows *path-from-to r p ps*
  using *assms*
proof (*induct p ps*)
  case (*PFRRoot p*)

```
  then have is-root p by auto
  then have 𝒢⟨→p⟩ = {} using root-defs-eq by auto
  have is-root r using PFRRoot.prems by auto
  then have 𝒢⟨→r⟩ = {} using root-defs-eq by auto
  have r ∈ 𝒫 by simp
  have p ∈ 𝒫 by simp
  have r = p
  proof (rule ccontr)
    assume r ≠ p
    then have is-tree (𝒫) (𝒢) ∧ p ∈ 𝒫 ∧ 𝒢⟨→p⟩ = {} ∧ r ≠ p ∧ r ∈ 𝒫  using
PFRRoot.hyps ‹𝒢⟨→p⟩ = {}› by auto
    then have card (𝒢⟨→r⟩) = 1 using unique-root by blast
    then show False by (simp add: ‹𝒢⟨→r⟩ = {}›)
  qed
  then show ?case by (metis NetworkOfCA.path-from-to.simps NetworkOfCA-axioms
PFRRoot.prems ‹p ∈ 𝒫›)
next
  case (PFRNode p q as)
  then have path-from-to r q as by simp
  then have tree-topology ∧ is-parent-of p q ∧ path-from-to r q as ∧ distinct (as
@ [p]) using PFRNode.hyps(1,2,5) by auto
  then show ?case using path-step by blast
qed

lemma p2root-down-step:
  (is-parent-of p q ∧ path-to-root q qs) ⟹ path-to-root p (p#qs)
  using path-to-root-downwards by auto


lemma path-to-root-exists:
  assumes tree-topology and p ∈ 𝒫
  shows ∃ps. path-to-root p ps
using assms proof (induct )
  case (ITRoot r)
  hence p = r
    by simp
  hence path-to-root p [p] sorry
  then show ?case by blast
next
  case (ITNode P E x q)
  assume IH: p ∈ P ⟹ ∃a. path-to-root p a
  assume a: p ∈ insert q P
  then show ?case
    proof (cases p = q)
      case True
      then show ?thesis sorry
    next
      case False
```

with *IH a* **show** *?thesis* **by** *blast*
qed
qed

lemma *edge-elem-to-edge*:
assumes $q \in \mathcal{G}\langle\rightarrow p\rangle$
shows $(q,\ p) \in \mathcal{G}$
using *assms* **by** (*meson Set.CollectD Set.CollectE*)

lemma *matching-words-to-peer-sets*:
assumes *tree-topology* **and** $((w\downarrow_{?})\downarrow_{!?}) = ((w'\downarrow_{!})\downarrow_{!?})$ **and** $w \in \mathcal{L}(p)$ **and** $w' \in \mathcal{L}(q)$ **and** *is-node p* **and** *is-parent-of p q* **and** $(w\downarrow_{?}) \neq \varepsilon$
shows $\mathcal{P}_{?}(p) = \{q\}$ **and** $p \in \mathcal{P}_{!}(q)$
using *assms*
proof −
have *t1*: *tree-topology* **using** *assms* **by** *simp*
have *pq*: *is-parent-of p q* **using** *assms* **by** *simp*
have *is-node p* **using** *assms*(*5*) **by** *blast*
then have $\mathcal{G}\langle\rightarrow p\rangle = \{q\}$ **by** (*metis is-parent-of.cases pq*)
then have *local-node*: *is-node-from-local p* **using** *edge-impl-psend-or-qrecv* **using** *t1* **by** *blast*
then have $\mathcal{P}_{?}(p) = \{q\} \vee p \in \mathcal{P}_{!}(q)$ **using** *pq* **by** (*meson edge-impl-psend-or-qrecv is-parent-of.cases*)
then have $(q,p) \in \mathcal{G}$ **using** *is-parent-of-rev*(*2*) *pq* **by** *auto*
then have *qintop*: $q \in \mathcal{G}\langle\rightarrow p\rangle$ **by** *blast*
then have $(\mathcal{G}\langle\rightarrow p\rangle) \neq \{\}$ **by** *blast*
then have *no0*: *card* $(\mathcal{G}\langle\rightarrow p\rangle) \neq 0$ **by** (*meson card-0-eq finite-peers finite-subset top-greatest*)
have *le1*: *card* $(\mathcal{G}\langle\rightarrow p\rangle) \leq 1$ **using** *at-most-one-parent t1* **by** *auto*
then have *card* $(\mathcal{G}\langle\rightarrow p\rangle) \neq 0 \wedge$ *card* $(\mathcal{G}\langle\rightarrow p\rangle) \leq 1$ **by** (*simp add*: *no0*)
have *card* $(\{q\}) = 1$ **by** *simp*
have $(\forall\, pp.\ (pp \neq q) \longrightarrow (pp,p) \notin \mathcal{G})$ **using** ‹$\mathcal{G}\langle\rightarrow p\rangle = \{q\}$› **by** *auto*
have $\exists\, a\ as\ b\ bs.\ (a\#as) = (w\downarrow_{?}) \wedge (b\#bs) = (w'\downarrow_{!})$ **by** (*metis assms*(*2*,*7*) *list.map-disc-iff neq-Nil-conv*)
then have $\exists\, a\ as\ b\ bs.\ (a\#as) = (w\downarrow_{?}) \wedge (b\#bs) = (w'\downarrow_{!}) \wedge ((a\#as)\downarrow_{!?}) = ((b\#bs)\downarrow_{!?})$ **by** (*metis assms*(*2*))
then obtain *a as b bs* **where** *as-def*: $(a\#as) = (w\downarrow_{?})$ **and** *bs-def*: $(b\#bs) = (w'\downarrow_{!})$ **and** *newt*: $((a\#as)\downarrow_{!?}) = ((b\#bs)\downarrow_{!?})$
by *blast*
then have $(([a]\downarrow_{!?})\ @\ (\ as\downarrow_{!?})) = (([b]\downarrow_{!?})\ @\ (\ bs\downarrow_{!?}))$ **by** (*metis Cons-eq-appendI append-self-conv2 map-append*)
then have $([a]\downarrow_{!?}) = ([b]\downarrow_{!?})$ **by** *simp*
have $(w\downarrow_{?}) = [a]\ @\ (as)$ **by** (*simp add*: *as-def*)
have $(w'\downarrow_{!}) = [b]\ @\ (bs)$ **by** (*simp add*: *bs-def*)
then have *is-input a*
proof *auto*
assume *a-out*: *is-output a*

then show *False*
proof –
have $(w{\downarrow_?}) = [a] \mathbin{@} as$ **by** (*simp add*: ‹$w{\downarrow_?} = a \# \varepsilon \cdot as$›)
have $(a\#as){\downarrow_?} = ([a]{\downarrow_?}) \mathbin{@} (as){\downarrow_?}$ **by** (*metis* ‹$w{\downarrow_?} = a \# \varepsilon \cdot as$› *as-def filter-append*)
then have $([a]{\downarrow_?}) = []$ **using** *a-out* **by** *auto*
then show *False* **by** (*metis Cons-eq-filterD as-def filter.simps*(*1*,*2*))
qed
qed
have *is-output b*
proof (*rule ccontr*)
assume *b-out*: *is-input b*
then show *False*
proof –
have $(w'{\downarrow_!}) = [b] \mathbin{@} bs$ **by** (*simp add*: ‹$w'{\downarrow_!} = b \# \varepsilon \cdot bs$›)
have $(b\#bs){\downarrow_!} = ([b]{\downarrow_!}) \mathbin{@} (bs){\downarrow_!}$ **by** (*metis* ‹$w'{\downarrow_!} = b \# \varepsilon \cdot bs$› *bs-def filter-append*)
then have *c1*: $([b]{\downarrow_!}) = []$ **using** *b-out* **by** *auto*
have $(w'{\downarrow_!}){\downarrow_!} = (w'{\downarrow_!})$ **by** *fastforce*
then have $([b] \mathbin{@} bs){\downarrow_!} = [b] \mathbin{@} bs$ **using** ‹$w'{\downarrow_!} = b \# \varepsilon \cdot bs$› **by** *auto*
have $([b] \mathbin{@} bs){\downarrow_!} = ([b]{\downarrow_!}) \mathbin{@} (bs){\downarrow_!}$ **using** ‹$(b \# bs){\downarrow_!} = (b \# \varepsilon){\downarrow_!} \cdot bs{\downarrow_!}$› ‹$w'{\downarrow_!} = b \# \varepsilon \cdot bs$› *bs-def* **by** *argo*
then have $([b]{\downarrow_!}) \mathbin{@} (bs){\downarrow_!} = [] \mathbin{@} (bs){\downarrow_!}$ **using** *c1* **by** *blast*
have $(w'{\downarrow_!}){\downarrow_!} = ([b] \mathbin{@} bs){\downarrow_!}$ **using** ‹$(b \# \varepsilon \cdot bs){\downarrow_!} = (b \# \varepsilon){\downarrow_!} \cdot bs{\downarrow_!}$› ‹$(b \# bs){\downarrow_!} = (b \# \varepsilon){\downarrow_!} \cdot bs{\downarrow_!}$› *bs-def* **by** *argo*
then have $(w'{\downarrow_!}){\downarrow_!} = ([] \mathbin{@} bs){\downarrow_!}$ **using** ‹$(b \# bs){\downarrow_!} = (b \# \varepsilon){\downarrow_!} \cdot bs{\downarrow_!}$› *c1* **by** *auto*
then have $([] \mathbin{@} bs) \neq (w'{\downarrow_!})$ **by** (*metis append.left-neutral bs-def not-Cons-self2*)
have $(([b] \mathbin{@} bs){\downarrow_!}){\downarrow_!} = (([b] \mathbin{@} bs){\downarrow_!})$ **by** *auto*
have $\forall c.\ length\ (c{\downarrow_!}) = length\ ((c{\downarrow_!}){\downarrow_!})$ **by** *simp*
then show *False* **by** (*metis* ‹$w'{\downarrow_!}{\downarrow_!} = (\varepsilon \cdot bs){\downarrow_!}$› *append-Nil bs-def impossible-Cons length-filter-le*)
qed
qed
then have *is-input a* $\wedge$ *is-output b* $\wedge$ *get-message a* $=$ *get-message b* **using** ‹$(a \# \varepsilon){\downarrow_{!?}} = (b \# \varepsilon){\downarrow_{!?}}$› ‹*is-input a*› **by** *auto*
then have $\exists\, s1\ s2.\ (s1, a, s2) \in \mathcal{R}\ p$ **by** (*metis NetworkOfCA.recv-proj-w-prepend-has-trans NetworkOfCA-axioms as-def assms*(*3*))
then have $\mathcal{P}_?(p) = \{q\}$
by (*metis* ‹*is-input a*› *is-parent-of-rev*(*2*) *no-recvs-no-input-trans pq sends-of-peer-subset-of-predecessors-in-topology subset-singletonD*)
then show $\mathcal{P}_?(p) = \{q\}$ **by** *blast*
have $\exists\, q1\ q2.\ (q1, b, q2) \in \mathcal{R}\ q$ **by** (*metis assms*(*4*) *bs-def send-proj-w-prepend-has-trans*)
then have $p \in \mathcal{P}_!(q)$ **by** (*metis CommunicatingAutomaton.SendingToPeers.simps CommunicatingAutomaton.well-formed-transition*
‹$\exists\, s1\ s2.\ s1\ {-a{\rightarrow}_{\mathcal{C}}}p\ s2$› ‹*is-input a* $\wedge$ *is-output b* $\wedge$ *get-message a* $=$ *get-message b*› *automaton-of-peer*
input-message-to-act-both-known message.inject output-message-to-act-both-known)
then show $p \in \mathcal{P}_!(q)$ **by** *simp*

qed

6.2 Influenced Language

lemma *is-in-infl-lang-rev-tree*:
  assumes *is-in-infl-lang p w*
  shows *tree-topology*
  using *assms is-in-infl-lang.simps* **by** *blast*

lemma *is-in-infl-lang-rev-root*:
  assumes *is-in-infl-lang p w* **and** *is-root p*
  shows $w \in \mathcal{L}(p)$
  using *assms*(*1*) *is-in-infl-lang.simps* **by** *blast*

lemma *is-in-infl-lang-rev-node*:
  assumes *is-in-infl-lang p w* **and** *is-node p*
  shows $\exists\, q\ w'.$ *is-parent-of p q* $\wedge\ w \in \mathcal{L}(p) \wedge$ *is-in-infl-lang q w'* $\wedge\ ((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$
  using *assms*
proof *induct*
  case (*IL-root r w*)
  then show *?case* **using** *root-defs-eq* **by** *fastforce*
next
  case (*IL-node p q w w'*)
  then show *?case* **by** *blast*
qed

lemma *w-in-infl-lang* : *is-in-infl-lang p w* $\Longrightarrow\ w \in \mathcal{L}(p)$ **using** *is-in-infl-lang.simps* **by** *blast*
lemma *recv-has-matching-send* : $[\![\mathcal{P}_?(p) = \{q\};\ w \in \mathcal{L}(p);$ *is-in-infl-lang q w'*; $((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})]\!] \Longrightarrow ((w{\downarrow_?}){\downarrow_{!?}}) \in ((((\mathcal{L}(q))\lfloor_{\{p,q\}})\lfloor_!)\lfloor_{!?})$
  using *w-in-infl-lang* **by** *blast*

lemma *child-matching-word-impl-in-infl-lang*:
  assumes *tree-topology* **and** *is-parent-of p q* **and** $w \in \mathcal{L}(q)$ **and** *is-in-infl-lang q w* **and** $((w'{\downarrow_?}){\downarrow_{!?}}) = (((w{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$ **and** $w' \in \mathcal{L}(p)$
  shows *is-in-infl-lang p w'*
  using *IL-node assms*(*1*,*2*,*4*,*5*,*6*) **by** *blast*

lemma *is-in-infl-lang-rev2*:
  assumes $w \in \mathcal{L}^*\ p$ **and** *is-node p*
  shows $w \in \mathcal{L}(p)$ **and** $\exists\, q\ w'.$ *is-parent-of p q* $\wedge\ w \in \mathcal{L}(p) \wedge w' \in \mathcal{L}^*\ q \wedge ((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$
  using *assms*
proof −
  show $w \in \mathcal{L}(p)$ **using** *assms*(*1*) *is-in-infl-lang.simps* **by** *blast*
  have *is-in-infl-lang p w* $\wedge$ *is-node p* **using** *assms*(*1*,*2*) **by** *auto*
  then have $\exists\, q\ w'.$ *is-parent-of p q* $\wedge\ w \in \mathcal{L}(p) \wedge$ *is-in-infl-lang q w'* $\wedge\ ((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$ **using** *is-in-infl-lang-rev-node* **by** *auto*

then show $\exists\, q\ w'.\ \textit{is-parent-of}\ p\ q \wedge w \in \mathcal{L}(p) \wedge w' \in \mathcal{L}^*\ q \wedge ((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$ **by** *blast*
qed

lemma *infl-lang-subset-of-lang*:
shows $(\mathcal{L}^*\ p) \subseteq (\mathcal{L}\ p)$
using *w-in-infl-lang* **by** *fastforce*

lemma *lang-subset-infl-lang*:
assumes *is-root p*
shows $(\mathcal{L}\ p) \subseteq (\mathcal{L}^*\ p)$
proof *auto*
fix x
assume $x \in \mathcal{L}\ p$
show *is-in-infl-lang p x* **using** *IL-root* ‹$x \in \mathcal{L}\ p$› *assms* **by** *presburger*
qed

lemma *root-lang-is-infl-lang*:
assumes *is-root p* **and** $w \in \mathcal{L}(p)$
shows $w \in \mathcal{L}^*(p)$
using *IL-root assms*(*1*,*2*) **by** *blast*

lemma *eps-in-infl*:
assumes *tree-topology* **and** $p \in \mathcal{P}$
shows $\varepsilon \in \mathcal{L}^*(p)$
proof $-$
have *a1*: $\forall\, q.\ ((\varepsilon{\downarrow_?}){\downarrow_{!?}}) = (((\varepsilon{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$ **by** *simp*
have *a2*: $\varepsilon \in \mathcal{L}(p)$ **by** (*meson CommunicatingAutomaton.REmpty2 CommunicatingAutomaton.Traces.simps automaton-of-peer*)
have $\exists\, ps.$ *path-to-root p ps* **by** (*simp add*: *assms*(*1*) *path-to-root-exists*)
then obtain *ps* **where** *path-to-root p ps* **by** *blast*
from *this a2* **show** *?thesis*
proof (*induct arbitrary*: *ps*)
case (*PTRRoot p*)
then show *?case* **using** *root-lang-is-infl-lang* **by** *blast*
next
case (*PTRNode p q as*)
have $\varepsilon \in \mathcal{L}\ q$ **by** (*meson CommunicatingAutomaton.REmpty2 CommunicatingAutomaton.Traces.simps automaton-of-peer*)
then have $\varepsilon \in \mathcal{L}^*\ q$ **using** *PTRNode.hyps*(*4*) **by** *auto*
then have *is-parent-of* $p\ q \wedge \varepsilon \in \mathcal{L}(p) \wedge$ *is-in-infl-lang* $q\ \varepsilon \wedge ((\varepsilon{\downarrow_?}){\downarrow_{!?}}) = (((\varepsilon{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$ **by** (*simp add*: *PTRNode.hyps*(*2*) *PTRNode.prems*)
then show *?case* **using** *IL-node assms*(*1*) **by** *blast*
qed
qed

lemma *infl-lang-has-tree-topology*:
assumes $w \in \mathcal{L}^*(p)$

shows *tree-topology*
using *assms is-in-infl-lang.simps* **by** *blast*

lemma *infl-parent-child-matching-ws* :
fixes w :: ($'information$, $'peer$) *action word*
assumes $w \in \mathcal{L}^*(p)$ **and** *is-parent-of p q*
shows $\exists\, w'.\ \ w' \in \mathcal{L}^*(q) \wedge ((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$
proof −
have $\exists\, q\ w'.$ *is-parent-of* $p\ q \wedge w \in \mathcal{L}(p) \wedge w' \in \mathcal{L}^*\ q \wedge ((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$
using *assms*(*1*,*2*) *is-in-infl-lang-rev2*(*2*) *is-parent-of.simps* **by** *blast*
then show *?thesis* **by** (*metis* (*mono-tags*, *lifting*) *assms*(*2*) *is-parent-of-rev*(*2*) *mem-Collect-eq singleton-conv*)
qed

lemma *infl-parent-child-matching-ws2* :
fixes w :: ($'information$, $'peer$) *action word*
assumes $w \in \mathcal{L}^*(q)$ **and** *is-parent-of p q* **and** $((w'{\downarrow_?}){\downarrow_{!?}}) = (((w{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$ **and** $w' \in \mathcal{L}(p)$
shows $w' \in \mathcal{L}^*(p)$
using *IL-node assms*(*1*,*2*,*3*,*4*) *is-parent-of-rev2*(*1*) **by** *blast*

6.2.1 influenced language and its shuffles

lemma *word-in-shuffled-infl-lang* :
fixes w :: ($'information$, $'peer$) *action word*
assumes $w \in \mathcal{L}^*(p)$
shows $w \in \mathcal{L}^*{}_{\sqcup\sqcup}(p)$
by (*meson assms shuffle-id*)

lemma *language-shuffle-subset* :
shows $\mathcal{L}^*(p) \subseteq \mathcal{L}^*{}_{\sqcup\sqcup}(p)$
using *word-in-shuffled-infl-lang* **by** *auto*

lemma *shuffled-infl-lang-rev* :
assumes $v \in \mathcal{L}^*(p)$
shows $\exists\, v'.\ (\ v' \sqcup\sqcup_?\ v \wedge v' \in \mathcal{L}^*{}_{\sqcup\sqcup}(p))$
using *assms* **by** (*rule valid-input-shuffles-of-lang*)

lemma *shuffled-infl-lang-impl-valid-shuffle* :
assumes $v \in \mathcal{L}^*{}_{\sqcup\sqcup}(p)$
shows $\exists\, v'.\ (\ v \sqcup\sqcup_?\ v' \wedge v' \in \mathcal{L}^*(p))$
using *assms shuffled-lang-impl-valid-shuffle* **by** *auto*

lemma *shuffle-prepend*:
assumes $y \sqcup\sqcup_?\ x$
shows $(w \cdot y) \sqcup\sqcup_?\ (w \cdot x)$
using *assms* **proof** (*induct x y rule*: *shuffled.induct*)

case (*refl w*)
then show *?case* **using** *shuffled.refl* **by** *blast*
next
case (*swap a b w xs ys*)
then show *?case* **by** (*metis append.assoc shuffled.swap*)
next
case (*trans w w′ w′′*)
then show *?case* **using** *shuffled.trans* **by** *blast*
qed

lemma *shuffle-append*:
assumes $y \sqcup\sqcup_? x$
shows $(y \cdot w) \sqcup\sqcup_? (x \cdot w)$
using *assms* **proof** (*induct x y rule*: *shuffled.induct*)
case (*refl w*)
then show *?case* **using** *shuffled.refl* **by** *blast*
next
case (*swap a b w xs ys*)
then show *?case* **by** (*simp add*: *shuffled.swap*)
next
case (*trans w w′ w′′*)
then show *?case* **using** *shuffled.trans* **by** *blast*
qed

lemma *full-shuffle-of*:
shows $\exists$ *xs ys*. $(xs \cdot ys) \sqcup\sqcup_? x \wedge xs{\downarrow_?} = xs \wedge ys{\downarrow_!} = ys$
proof (*induct x*)
case *Nil*
then show *?case* **by** (*metis append.right-neutral filter.simps*(*1*) *shuffled.refl*)
next
case (*Cons a as*)
then obtain *xs ys* **where** *shuf*: $xs \cdot ys \sqcup\sqcup_? as$ **and** *xs-def*: $xs{\downarrow_?} = xs$ **and** *ys-def*: $ys{\downarrow_!} = ys$ **by** *blast*
then show *?case* **proof** (*cases is-input a*)
case *True*
then have $([a] \cdot xs){\downarrow_?} = ([a] \cdot xs)$ **by** (*simp add*: *xs-def*)
have *new-shuf*: $[a] \cdot xs \cdot ys \sqcup\sqcup_? ([a] \cdot as)$ **by** (*simp add*: *shuf shuffled-prepend-inductive*)
then show *?thesis* **by** (*metis* ‹$(a \# \varepsilon \cdot xs){\downarrow_?} = a \# \varepsilon \cdot xs$› *append-eq-Cons-conv self-append-conv2 ys-def*)
next
case *False*
then have *a-ys-def*: $([a] \cdot ys){\downarrow_!} = ([a] \cdot ys)$ **by** (*simp add*: *ys-def*)
have $xs \cdot [a] \sqcup\sqcup_? ([a] \cdot xs)$ **using** *fully-shuffled-implies-output-right* **by** (*metis False xs-def*)
then have $xs \cdot [a] \cdot ys \sqcup\sqcup_? ([a] \cdot xs \cdot ys)$ **using** *shuffle-append* **by** *blast*
then have *new-shuf*: $xs \cdot [a] \cdot ys \sqcup\sqcup_? ([a] \cdot as)$ **by** (*metis* (*no-types*, *lifting*)

append.assoc shuf shuffle-prepend shuffled.trans)
    then show *?thesis* **using** *a-ys-def xs-def* **by** *fastforce*
  qed
qed

lemma *full-shuffle-of-concrete*:
  shows $((x{\downarrow_?}) \cdot (x{\downarrow_!})) \sqcup\sqcup_? x$
proof (*induct x*)
  case *Nil*
  then show *?case* **by** (*metis append.right-neutral filter.simps*(*1*) *shuffled.refl*)
next
  case (*Cons a as*)
  then show *?case* **using** *Cons* **proof** (*cases is-input a*)
    case *True*
    have $(a \# as){\downarrow_?} = ([a]{\downarrow_?} \cdot as{\downarrow_?})$ **by** *simp*
    moreover have $[a]{\downarrow_?} = [a]$ **by** (*simp add*: *True*)
    then show *?thesis* **by** (*metis Cons-eq-appendI filter.simps*(*1*,*2*) *filter-head-helper local.Cons shuffled-prepend-inductive*)
  next
    case *False*
    have $(a \# as){\downarrow_!} = ([a]{\downarrow_!} \cdot as{\downarrow_!})$ **by** *simp*
    moreover have $[a]{\downarrow_!} = [a]$ **by** (*simp add*: *False*)
    moreover have $(a \# as){\downarrow_?} = as{\downarrow_?}$ **using** *False* **by** *auto*
    moreover have *is-output a* **using** *False* **by** *auto*
    ultimately show *?thesis* **by** (*metis* (*mono-tags*, *lifting*) *append.right-neutral append-Nil filter-append full-shuffle-of input-proj-output-yields-eps output-proj-input-yields-eps shuffled-keeps-recv-order shuffled-keeps-send-order*)
  qed
qed

lemma *shuffle-keeps-outputs-right*:
  assumes $w' \sqcup\sqcup_? (w)$ **and** *is-output* (*last w*)
  shows *is-output* (*last w'*)
  using *assms shuffle-keeps-outputs-right-shuffled* **by** *metis*

lemma *root-graph*:
  assumes $\mathcal{P} = \{p\}$ **and** *tree-topology*
  shows $\mathcal{G}\langle \rightarrow p\rangle = \{\}$
  by (*metis* (*full-types*, *lifting*) *UNIV-I assms*(*1*,*2*) *empty-Collect-eq singleton-iff tree-acyclic*)

lemma *p-root*:

assumes *path-to-root* p $[p]$ **and** *tree-topology*
shows $\mathcal{G}\langle\rightarrow p\rangle = \{\}$
proof *auto*
fix q
assume $(q, p) \in \mathcal{G}$
then show *False*
by (*smt* (*verit*, *ccfv-threshold*) *CommunicatingAutomaton.SendingToPeers.intros*
CommunicatingAutomaton.well-formed-transition Edges-rev NetworkOfCA.no-input-trans-root NetworkOfCA-axioms
assms(*1*) *automaton-of-peer get-receiver.simps global-to-local-root input-message-to-act messages-used*
output-message-to-act-both-known prod.inject single-path-impl-root)
qed

lemma *root-lang-word-facts*:
assumes $\mathcal{P}_?(q) = \{\}$ **and** $(\forall p.\ q \notin \mathcal{P}_!(p))$ **and** $w \in \mathcal{L}^*(q)$ **and** *tree-topology*
shows $w = w{\downarrow_q} \wedge w = w{\downarrow_!} \wedge w \in \mathcal{L}(q)$
using *assms*(*1*,*3*) *no-inputs-implies-only-sends-alt w-in-infl-lang w-in-peer-lang-impl-p-actor*
by *auto*

lemma *root-lang-is-mbox*:
assumes *is-root* p **and** $w \in \mathcal{L}(p)$
shows $w \in \mathcal{T}_{None}$
sorry

lemma *parent-in-infl-has-matching-sends*:
assumes $w \in \mathcal{L}^*(p)$ **and** *path-to-root* p $(p\#q\#ps)$
shows $\exists\, w'.\ w' \in \mathcal{L}^*(q) \wedge ((w{\downarrow_?}){\downarrow_{!?}}) = (((w'{\downarrow_{\{p,q\}}}){\downarrow_!}){\downarrow_{!?}})$
using *assms*(*1*,*2*) *infl-parent-child-matching-ws path-to-root-first-elem-is-peer path-to-root-stepback*
by *blast*

lemma *send-proj-on-infl-word*:
assumes $v \in ((\mathcal{L}_!{}^*(p)))$
shows $v = v{\downarrow_!}$
using *assms*
proof (*induct* v)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a as*)
then show *?case* **by** *force*
qed

lemma *v-in-send-infl-to-send-L*:
assumes $v \in (\mathcal{L}_!{}^*(p))$
shows $v \in (\mathcal{L}_!(p))$
using *assms w-in-infl-lang* **by** (*induct*, *auto*)

lemma *send-infl-subset-send-lang*: $(\mathcal{L}_!{}^*(p)) \subseteq (\mathcal{L}_!(p))$ **using** *v-in-send-infl-to-send-L*

by *blast*

lemma *pair-proj-comm*: $v\downarrow_{\{p,q\}} = v\downarrow_{\{q,p\}}$ **by** *meson*

lemma *pair-proj-inv-with-send-proj*:
assumes $v = v\downarrow_!$
shows $(v\downarrow_{\{p,q\}}) = (v\downarrow_{\{p,q\}})\downarrow_!$
using *assms*
proof (*induct v*)
case *Nil*
then show *?case* **using** *eps-always-in-lang* **by** *auto*
next
case (*Cons a as*)
then show *?case* **by** (*metis* (*no-types*, *lifting*) *filter.simps*(*2*) *list.distinct*(*1*) *list.inject*
output-proj-input-yields-eps)
qed

lemma *send-infl-lang-pair-proj-inv-with-send*:
assumes $v \in ((\mathcal{L}_!{}^*(q))|_{\{p,q\}})$
shows $v = v\downarrow_!$
using *assms*
proof (*induct v*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a as*)
obtain v' **where** $(a\#as) = (v'\downarrow_{\{p,q\}})$ **and** $v' \in (\mathcal{L}_!{}^*(q))$ **using** *Cons.prems* **by** *blast*
then have $(v') = (v')\downarrow_!$ **by** *force*
then have $(v'\downarrow_{\{p,q\}}) = (v'\downarrow_{\{p,q\}})\downarrow_!$ **using** *pair-proj-inv-with-send-proj* **by** *fastforce*
then show *?case* **using** ‹$a \# as = v'\downarrow_{\{p,q\}}$› **by** *presburger*
qed

lemma *projs-on-peer-eq-if-in-peer-lang*:
assumes $v \in ((\mathcal{L}_!{}^*(q))|_{\{p,q\}})$ **and** *is-parent-of p q*
shows $v = (v)\downarrow_q$
proof –
have $v \in ((\mathcal{L}_!(q))|_{\{p,q\}})$ **using** *assms*(*1*) *w-in-infl-lang* **by** *auto*
then have $v \in (((\mathcal{L}(q))|_!)|_{\{p,q\}})$ **by** *blast*
have $\forall x. (x \in (\mathcal{L}(q))) \longrightarrow (x = (x\downarrow_q))$ **by** (*simp add*: *w-in-peer-lang-impl-p-actor*)

then have $\forall v'. ((((v')\downarrow_!)\downarrow_{\{p,q\}}) = v \wedge v' \in (\mathcal{L}(q))) \longrightarrow (v' = (v'\downarrow_q))$ **by** *simp*
then have $\forall v'. ((((v')\downarrow_!)\downarrow_{\{p,q\}}) = v \wedge v' \in (\mathcal{L}(q))) \longrightarrow (((((v')\downarrow_!)\downarrow_{\{p,q\}})) = (((((v')\downarrow_!)\downarrow_{\{p,q\}}))\downarrow_q))$ **by** (*metis* (*mono-tags*, *lifting*) *filter-recursion proj-trio-inv proj-trio-inv2*)
then show *?thesis* **using** ‹$v \in (\mathcal{L}\ q)|_!|_{\{p,q\}}$› **by** *blast*
qed

lemma *is-in-infl-lang-app*:
  assumes *is-in-infl-lang p* $(u\ @\ v)$
  shows *is-in-infl-lang p u*
  using *assms*
proof (*induct p* $(u\ @\ v)$ *arbitrary*: $u\ v$)
  case (*IL-root r w*)
  then show *?case* **using** *Lang-app is-in-infl-lang.IL-root* **by** *blast*
next
  case (*IL-node p q w w'*)
  then have *is-in-infl-lang p* $(w' \cdot v)$ **using** *is-in-infl-lang.IL-node* **by** *blast*
  then have $w \in \mathcal{L}^*(q) \wedge (((w' \cdot v)\downarrow_?)\downarrow_{!?}) = (((w\downarrow_{\{p,q\}})\downarrow_!)\downarrow_{!?})$ **using** *IL-node.hyps*(*4*,*6*) **by** *blast*
  then have *p-w-match*: $(((w' \cdot v)\downarrow_?)\downarrow_{!?}) = (((w\downarrow_{\{p,q\}})\downarrow_!)\downarrow_{!?})$ **by** *blast*
  have *p-decomp*: $(((w' \cdot v)\downarrow_?)\downarrow_{!?}) = (((w')\downarrow_?)\downarrow_{!?})\ @\ (((v)\downarrow_?)\downarrow_{!?})$ **by** *simp*

  have $\exists\, w''\ w'''.\ w = (w''\ @\ w''') \wedge (((w')\downarrow_?)\downarrow_{!?}) = (((w''\downarrow_{\{p,q\}})\downarrow_!)\downarrow_{!?})$
  proof (*induct length w' arbitrary*: w')
    case *0*
    then show *?case* **by** *fastforce*
  next
    case (*Suc x*)
    then obtain *a as* **where** $x = |as|$ **and** $w' = as\ @\ [a]$ **by** (*metis length-Suc-conv-rev*)
    then have $\exists\, w''\ w'''.\ w = w'' \cdot w''' \wedge as\downarrow_?\downarrow_{!?} = w''\downarrow_{\{p,q\}}\downarrow_!\downarrow_{!?}$ **using** *Suc.hyps*(*1*) **by** *presburger*
    then obtain $w''\ w'''$ **where** $w = w'' \cdot w'''$ **and** $as\downarrow_?\downarrow_{!?} = w''\downarrow_{\{p,q\}}\downarrow_!\downarrow_{!?}$ **by** *blast*
    then have *is-in-infl-lang q* (w'') **using** *IL-node.hyps*(*5*) **by** *blast*
    then show *?case* **sorry**
  qed

  then obtain $w''\ w'''$ **where** $w = (w''\ @\ w''')$ **and** $(((w')\downarrow_?)\downarrow_{!?}) = (((w''\downarrow_{\{p,q\}})\downarrow_!)\downarrow_{!?})$ **by** *blast*

  then have *is-in-infl-lang q* w'' **by** (*meson IL-node.hyps*(*5*))
  have $w' \in \mathcal{L}\ p$ **using** *IL-node.hyps*(*3*) *Lang-app* **by** *blast*
  then have *tree-topology* $\wedge$ *is-parent-of p q* $\wedge$ $w' \in \mathcal{L}(p) \wedge$ *is-in-infl-lang q* w'' $\wedge ((w'\downarrow_?)\downarrow_{!?}) = (((w''\downarrow_{\{p,q\}})\downarrow_!)\downarrow_{!?})$
    using *IL-node.hyps*(*1*,*2*) ‹*is-in-infl-lang q* w''› ‹$w'\downarrow_?\downarrow_{!?} = w''\downarrow_{\{p,q\}}\downarrow_!\downarrow_{!?}$› **by** *blast*
  then have *is-in-infl-lang p* w' **using** *is-in-infl-lang.IL-node*[*of p q* $w'\ w''$] **by** *blast*
  then show *?case* **by** *simp*
qed

lemma *infl-word-impl-prefix-valid*:
  assumes $(u\ @\ v) \in \mathcal{L}^*\ p$
  shows $u \in \mathcal{L}^*\ p$
  using *assms is-in-infl-lang-app* **by** *blast*

lemma *peer-pair-infl-send-nosymb-comm*: $(((\mathcal{L}_!{}^*(q))\downharpoonleft_{\{q,p\}})\downharpoonleft_{!?}) = (((\mathcal{L}_!{}^*(q))\downharpoonleft_{\{p,q\}})\downharpoonleft_{!?})$
proof −
have $(((\mathcal{L}_!{}^*(q))\downharpoonleft_{\{q,p\}})) = (((\mathcal{L}_!{}^*(q))\downharpoonleft_{\{p,q\}}))$ **by** (*simp add*: *pair-proj-comm*)
then show *?thesis* **by** *presburger*
qed

lemma *child-send-is-from-parent*:
assumes *is-input a* **and** *is-parent-of p q* **and** *get-actor a = p* **and** $(s1, a, s2) \in (\mathcal{R}\ p)$
shows *get-object a = q*
proof (*rule ccontr*)
assume *get-object a* $\neq q$
then obtain *qq* **where** $qq \neq q$ **and** *get-object a = qq* **and** $qq \in \mathcal{P}$ **by** *simp*
then have $qq \in \mathcal{P}_?\ p$ **by** (*metis CommunicatingAutomaton.empty-receiving-from-peers assms*(*1*,*4*) *automaton-of-peer*)
have *card* $(\mathcal{P}_?\ p) \leq 1$ **using** ‹*get-object a = qq*› ‹*get-object a* $\neq q$› ‹$qq \in \mathcal{P}_?\ p$› *assms*(*2*) *is-parent-of-rev*(*2*)
sends-of-peer-subset-of-predecessors-in-topology **by** *fastforce*
then have $\mathcal{P}_?\ p = \{qq\}$ **by** (*meson* ‹$qq \in \mathcal{P}_?\ p$› *finite-peers finite-set-card-union-with-singleton finite-subset subset-UNIV*)
then show *False* **using** ‹$\mathcal{P}_?\ p = \{qq\}$› ‹$qq \neq q$› *assms*(*2*) *insert-subset is-parent-of-rev*(*2*) *sends-of-peer-subset-of-predecessors-in-topology singleton-iff* **by** *metis*
qed

lemma *infl-word-actor-app*:
assumes $(w\ @\ xs) \in (\mathcal{L}^*(q))$
shows $(w{\downarrow_q} = w) \wedge (xs{\downarrow_q} = xs)$
using *assms* **proof** −
have $(w\ @\ xs) \in (\mathcal{L}(q))$ **using** *assms w-in-infl-lang* **by** *auto*
then have $(w\ @\ xs){\downarrow_q} = (w\ @\ xs)$ **using** *w-in-peer-lang-impl-p-actor* **by** *presburger*
then show *?thesis* **by** (*metis actor-proj-app-inv*)
qed

6.2.2 simulate sync with mbox word

lemma *add-matching-recvs-app* :
shows *add-matching-recvs* $(xs \cdot ys)$ = (*add-matching-recvs xs*) $\cdot$ (*add-matching-recvs ys*)
proof (*induct xs arbitrary*: *ys rule*: *add-matching-recvs.induct*)
case *1*
then show *?case* **by** *simp*
next
case (*2 a w*)
then show *?case* **by** *simp*
qed

lemma *adding-recvs-keeps-send-order* :
  shows $w{\downarrow_!} = (\textit{add-matching-recvs } w){\downarrow_!}$
proof (*induct w*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a w'*)
  then show *?case* **using** *Cons*
  proof (*cases is-input a*)
    case *True*
    then show *?thesis* **by** (*simp add*: *local.Cons*)
  next
    case *False*
    then show *?thesis* **by** (*simp add*: *local.Cons*)
  qed
qed

lemma *simulate-sync-step-with-matching-recvs-helper2*:
  assumes $c1 \ -\langle(!\langle(i^{p\rightarrow q})\rangle), \infty\rangle\rightarrow\ c2 \wedge c2 \ -\langle ?\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ c3$
  shows *mbox-run c1 None* $[!\langle(i^{p\rightarrow q})\rangle,\ ?\langle(i^{p\rightarrow q})\rangle]$ $[c2,c3]$
  using *assms*
proof −
  have *mbox-run c1 None* [] [] **by** (*simp add*: *MREmpty*)
  have *last* $(c1 \ \# \ [])$ $-\langle!\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ c2$ **by** (*simp add*: *assms*)
  have *mbox-run c1 None* $[!\langle(i^{p\rightarrow q})\rangle]$ $[c2]$ **by** (*metis MRComposedInf* ‹*last* $(c1 \ \# \ \varepsilon)$ $-\langle!\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ c2$› ‹*mbox-run c1 None* ε ε›
    self-append-conv2)
  have *last* $(c1 \ \# \ [c2])$ $-\langle ?\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ c3$ **by** (*simp add*: *assms*)
  have *mbox-run c1 None* $[!\langle(i^{p\rightarrow q})\rangle,\ ?\langle(i^{p\rightarrow q})\rangle]$ $[c2,\ c3]$ **using** *MRComposedInf* ‹*last* $(c1 \ \# \ c2 \ \# \ \varepsilon)$ $-\langle ?\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ c3$›
    ‹*mbox-run c1 None* $(!\langle(i^{p\rightarrow q})\rangle \ \# \ \varepsilon)$ $(c2 \ \# \ \varepsilon)$› **by** *fastforce*
  show *?thesis* **by** (*simp add*: ‹*mbox-run c1 None* $(!\langle(i^{p\rightarrow q})\rangle \ \# \ ?\langle(i^{p\rightarrow q})\rangle \ \# \ \varepsilon)$ $(c2 \ \# \ c3 \ \# \ \varepsilon)$›)
qed

lemma *simulate-sync-step-with-matching-recvs*:
  assumes $c1 \ -\langle(!\langle(i^{p\rightarrow q})\rangle), \infty\rangle\rightarrow\ c2 \wedge c2 \ -\langle ?\langle(i^{p\rightarrow q})\rangle, \infty\rangle\rightarrow\ c3$
  shows *mbox-run c1 None* (*add-matching-recvs* $[!\langle(i^{p\rightarrow q})\rangle]$) $[c2,c3]$
  by (*simp add*: *assms simulate-sync-step-with-matching-recvs-helper2*)

— shows that we can simulate a synchronous run by adding the matching receives after each send
— this also shows that both the first config and the last config of the mbox run are the same as in sync run
lemma *sync-run-to-mbox-run* :
  assumes *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *w xcs* **and** $xcs \neq []$
  shows $\exists\, xcm.$ *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs w*) *xcm* $\wedge$ $(\forall\, p.$ (*last xcm* p) = ((*last xcs*) p, ε))

using *assms*
proof (*induct length w arbitrary*: *w xcs*)
case *0*
then have *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *w xcs* = *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ [] *xcs* **by** *simp*
then have *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *w xcs* = *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ [] []
by (*simp add*: *0.prems*(*1*) *SREmpty*)
then show *?case*
by (*metis 0.prems*(*2*) ‹*sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *w xcs* = *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ ε *xcs*› *append-is-Nil-conv not-Cons-self2 sync-run.simps*)
next
case (*Suc x*)
then have *fact1*: *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *w xcs* **by** *auto*
then have *fact2*: *Suc x* = |*w*| **using** *Suc.hyps*(*2*) **by** *auto*
then obtain *v a xc s-a* **where** *w* = *v* @ [*a*] **and** *v-sync*: *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *v xc* **and** *xc-def* : *xcs* = *xc* @ [*s-a*]
by (*metis Suc.prems*(*2*) *fact1 sync-run.simps*)
then have *length v* = *x*
by (*simp add*: *Suc-inject fact2*)
then show *?case* **using** *assms Suc*
proof (*cases xc* $\neq$ ε)
case *True*

have $\exists$ *xcm*. *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs v*) *xcm* $\wedge$ ($\forall$ *p*. (*last xcm p*) = ((*last xc*) *p*, ε))
by (*simp add*: *Suc.hyps*(*1*) *True* ‹|*v*| = *x*› *v-sync*)
then obtain *xcm* **where** *v-mbox*: *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs v*) *xcm*
and *v-state* :($\forall$ *p*. (*last xcm p*) = ((*last xc*) *p*, ε)) **by** *auto*
then obtain *s-1* **where** *s-1-1*: *sync-step s-1 a s-a* **and** *s-1-2*: *s-1* = *last xc*
by (*metis* ‹*w* = *v* · *a* # ε› ‹*xc* $\neq$ ε› *fact1 last-ConsR sync-run-rev xc-def*)
then obtain *i p q* **where** *a-decomp*: *a* = $!\langle(i^{p \rightarrow q})\rangle$ **using** *sync-step-rev*(*3*) **by** *blast*
let *?c1* = (λx. (*s-1 x*, ε))
let *?c3* = (λx. (*s-a x*, ε))
let *?c2* = (*?c3*)(*q* := ((*s-1*) *q*, [$(i^{p \rightarrow q})$]))
have *c1-def* : *?c1* = (λx. (*s-1 x*, ε)) **by** *simp*
have *c3-def* : *?c3* = (λx. (*s-a x*, ε)) **by** *simp*
have *c2-def* : *?c2* = (*?c3*)(*q* := ((*s-1*) *q*, [$(i^{p \rightarrow q})$])) **by** *simp*
have *sync-step s-1* ($!\langle(i^{p \rightarrow q})\rangle$) *s-a* **using** *a-decomp s-1-1* **by** *auto*
then have *sync-abb* : *s-1* $-\langle !\langle(i^{p \rightarrow q})\rangle, \mathbf{0}\rangle\rightarrow$ *s-a* **by** *simp*
then have *mbox-steps*: *let c1* = λx. (*s-1 x*, ε); *c3* = λx. (*s-a x*, ε); *c2* = (*c3*)(*q* := (*s-1 q*, [$(i^{p \rightarrow q})$])) *in*
mbox-step c1 ($!\langle(i^{p \rightarrow q})\rangle$) *None c2* $\wedge$ *mbox-step c2* ($?\langle(i^{p \rightarrow q})\rangle$) *None c3* **by** (*simp add*: *sync-step-to-mbox-steps*)
then have *mbox-steps-init* : *mbox-step ?c1* ($!\langle(i^{p \rightarrow q})\rangle$) *None ?c2* $\wedge$ *mbox-step ?c2* ($?\langle(i^{p \rightarrow q})\rangle$) *None ?c3* **by** *metis*
then have *a-mbox-run*: *mbox-run ?c1 None* (*add-matching-recvs* ([*a*])) ([*?c2*, *?c3*]) **using** *a-decomp simulate-sync-step-with-matching-recvs* **by** *blast*
then have ($\forall$ *p*. *fst* (*last xcm p*) = (*s-1*) *p*) **by** (*simp add*: *s-1-2 v-state*)

then have $(\forall\, p.\ (\mathit{last\ xcm\ p}\ ) = \mathit{?c1\ p})$ **by** (*simp add*: *v-state*)
then have *last-config-xcm* : *last xcm* = *?c1* **by** *auto*
then have $(\mathit{last\ xcm})\ -\langle !\langle (i^{p\rightarrow q})\rangle,\ \infty\rangle\rightarrow\ \mathit{?c2}$ **by** (*metis mbox-steps*)
then have *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs v*) *xcm* **by** (*simp add*: *v-mbox*)
then have *mbox-inter* : *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* $((\mathit{add\text{-}matching\text{-}recvs\ v})@\ [!\langle (i^{p\rightarrow q})\rangle])$ $(\mathit{xcm}@[\mathit{?c2}])$
by (*smt* (*verit*) *Nil-is-append-conv*
‹$\mathit{last\ xcm}\ -\langle !\langle (i^{p\rightarrow q})\rangle,\ \infty\rangle\rightarrow\ (\lambda x.\ (\mathit{s\text{-}a}\ x,\ \varepsilon))\ (q := (\mathit{s\text{-}1}\ q,\ i^{p\rightarrow q}\ \#\ \varepsilon))$› ‹$\mathit{xc} \neq \varepsilon$›
add-matching-recvs.elims last-ConsR list.distinct(*1*) *mbox-run.simps sync-run.cases*
v-sync)
then have $(\mathit{last}\ (\mathit{xcm}@[\mathit{?c2}]))\ -\langle\ ?\langle (i^{p\rightarrow q})\rangle,\ \infty\rangle\rightarrow\ \mathit{?c3}$ **by** (*simp add*: *mbox-steps-init*)
then have *mbox-inter2* : *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* $((\mathit{add\text{-}matching\text{-}recvs\ v})@\ [!\langle (i^{p\rightarrow q})\rangle]@[?\langle (i^{p\rightarrow q})\rangle])$ $(\mathit{xcm}@[\mathit{?c2}]@[\mathit{?c3}])$
using *MRComposedInf mbox-inter* **by** *fastforce*
— found existing run when xc not empty
then have *mbox-run-final*: *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* $((\mathit{add\text{-}matching\text{-}recvs}\ (v@[a])))$ $(\mathit{xcm}@[\mathit{?c2},\mathit{?c3}])$
using *NetworkOfCA.add-matching-recvs-app NetworkOfCA-axioms a-decomp append-Cons* **by** *fastforce*
then have *xc-nonempty-thesis* : *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* $((\mathit{add\text{-}matching\text{-}recvs}\ (v@[a])))\ (\mathit{xcm}@[\mathit{?c2},\mathit{?c3}]) \wedge (\forall\, p.\ (\mathit{last}\ (\mathit{xcm}@[\mathit{?c2},\mathit{?c3}])\ p\ ) = ((\mathit{last\ xcs})\ p,\ \varepsilon\ ))$
by (*simp add*: *xc-def*)
then show *?thesis* **using** ‹$w = v \cdot a\ \#\ \varepsilon$› **by** *blast*
next
case *False*
then have *xc-empty*: $\mathit{xc} = \varepsilon$ **by** *simp*
then have *w-a* : $w = [a]$ **using** *NetworkOfCA.sync-run.cases NetworkOfCA-axioms* ‹$w = v \cdot a\ \#\ \varepsilon$› *v-sync* **by** *blast*
then have *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *w xcs* = *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ $[a]$ *xcs* **by** (*simp add*: *SREmpty fact1*)
then obtain *i p q C* **where** *C-def*: *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ $[a]$ $[C]$ **and** *C-def2*: $\mathit{xcs} = [C]$ **and** *a-def*: $a = !\langle (i^{p\rightarrow q})\rangle$
by (*metis fact1 self-append-conv2 sync-run-rev sync-step-rev*(*3*) *xc-def xc-empty*)
let $\mathit{?c1} = (\lambda p.\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\ p,\ \varepsilon))$
let $\mathit{?c3} = (\lambda x.\ (C\ x,\ \varepsilon))$
let $\mathit{?c2} = (\mathit{?c3})(q := ((\mathcal{C}_{\mathcal{I}\mathbf{0}})\ q,\ [(i^{p\rightarrow q})]))$
have *c1-def* : $\mathit{?c1} = (\lambda x.\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\ x,\ \varepsilon))$ **by** *simp*
have *c3-def* : $\mathit{?c3} = (\lambda x.\ (C\ x,\ \varepsilon))$ **by** *simp*
have *c2-def* : $\mathit{?c2} = (\mathit{?c3})(q := ((\mathcal{C}_{\mathcal{I}\mathbf{0}})\ q,\ [(i^{p\rightarrow q})]))$ **by** *simp*
have $(\forall\, p.\ \mathcal{C}_{\mathcal{I}\mathfrak{m}}\ p = (\mathcal{C}_{\mathcal{I}\mathbf{0}}\ p,\ \varepsilon))$ **by** *simp*
then have $\mathcal{C}_{\mathcal{I}\mathfrak{m}} = (\lambda p.\ (\mathcal{C}_{\mathcal{I}\mathbf{0}}\ p,\ \varepsilon))$ **by** *simp*
then have $\mathit{?c1} = \mathcal{C}_{\mathcal{I}\mathfrak{m}}$ **by** *simp*
have *sync-step* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *a C* **by** (*metis C-def2* ‹$w = v \cdot a\ \#\ \varepsilon$› *fact1 last-ConsL self-append-conv2 sync-run-rev*)
then have $\mathcal{C}_{\mathcal{I}\mathbf{0}}\ -\langle !\langle (i^{p\rightarrow q})\rangle,\ \mathbf{0}\rangle\rightarrow\ C$ **by** (*simp add*: *a-def*)
then have *steps* : *mbox-step* *?c1* $(!\langle (i^{p\rightarrow q})\rangle)$ *None* *?c2* $\wedge$ *mbox-step* *?c2*

$(?\langle(i^{p\rightarrow q})\rangle)$ *None ?c3*
by (*metis sync-step-to-mbox-steps*)
then have *mbox-run ?c1 None* (*add-matching-recvs* ([*a*])) [*?c2*, *?c3*]
using *a-def simulate-sync-step-with-matching-recvs* **by** *blast*
then have *mbox-run ?c1 None* (*add-matching-recvs w*) [*?c2*, *?c3*] **by** (*simp add*: *w-a*)
then have *mbox-run ?c1 None* (*add-matching-recvs w*) [*?c2*, *?c3*] **by** *simp*
then have *mbox-run* ($\lambda p.$ ($\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *p*, ε)) *None* (*add-matching-recvs w*) [*?c2*, *?c3*] **by** *simp*
then show *?thesis* **using** *C-def2* **by** *auto*
qed
qed

lemma *empty-sync-run-to-mbox-run* :
assumes *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *w xcs* **and** *xcs* = []
shows *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs w*) []
using *assms* **by** (*metis* (*no-types*, *lifting*) *MREmpty Nil-is-append-conv add-matching-recvs.simps*(*1*) *not-Cons-self2 sync-run.simps*)

6.2.3 Lemma 4.4 and preparations

lemma *concat-infl-path-rev* :
assumes *concat-infl p w* (*q*#*ps*) *w'*
shows *path-to-root q* (*q*#*ps*)
using *assms*
proof(*induct* (*q*#*ps*) *w' arbitrary*: *q ps rule*: *concat-infl.induct*)
case *at-p*
then show *?case* **using** *path-to-root-first-elem-is-peer* **by** *blast*
next
case (*reach-root q qw x w-acc*)
then show *?case* **using** *path-to-root-first-elem-is-peer path-to-root-stepback* **by** *blast*
next
case (*node-step x q ps qw w-acc*)
then show *?case* **by** (*metis list.discI path-to-root-first-elem-is-peer path-to-root-stepback*)
qed

lemma *concat-infl-tree-rev* :
assumes *concat-infl p w ps w'*
shows *tree-topology*
using *assms concat-infl.cases* **by** *blast*

lemma *concat-infl-p-first-or-not-exists*:
assumes *concat-infl p w ps w'*
shows ($\exists$ *qs*. *ps* = *p*#*qs*) $\vee$ ($\forall$ *xs ys*. *ps* $\neq$ *xs* @ [*p*] @ *ys*)
using *assms*
sorry

lemma *concat-infl-actor-consistent*:
assumes *concat-infl p w ps w-acc*
shows $w\text{-}acc{\downarrow}_p = w$
using *assms*
proof (*induct ps w-acc rule*: *concat-infl.induct*)
case (*at-p ps*)
then show *?case* **using** *w-in-infl-lang w-in-peer-lang-impl-p-actor* **by** *force*
next
case (*reach-root q qw x w-acc′*)
then have $qw \in \mathcal{L}\ q$ **by** (*simp add*: *w-in-infl-lang*)
then have $qw{\downarrow}_q = qw$ **using** *w-in-peer-lang-impl-p-actor* **by** *fastforce*
then show *?case*
proof (*cases q = p*) — can't be the case because then concat$_i nfl_{isnottrue}$
case True
then have qw$\downarrow_p$ = qw **using** ‹qw$\downarrow_q$ = qw› **by** blast
then have qw ∈ $\mathcal{L}$ p **using** True ‹qw ∈ $\mathcal{L}$ q› **by** blast
then have is-root p **using** True reach-root.hyps(1) **by** auto
then have ¬ path-to-root p (x # q # ε) **by** (metis True list.distinct(1) list.inject path-to-root-first-elem-is-peer path-to-root-stepback
path-to-root-unique)
have concat-infl p w (x # q # ε) w-acc′ **by** (simp add: reach-root.hyps(5))
then have path-to-root x (x # q # ε) **by** (simp add: reach-root.hyps(3))
then have x ≠ q **using** True ‹¬ path-to-root p (x # q # ε)› **by** auto
have (∀ xs ys. (x # q # ε) ≠ xs @ [p] @ ys) **using** True ‹x ≠ q› concat-infl-p-first-or-not-exists reach-root.hyps(5) **by** blast
have (x # q # ε) = [x] @ [p] @ [] **using** True **by** auto
then show ?thesis **using** ‹∀ xs ys. x # q # ε ≠ xs · (p # ε · ys)› **by** blast
next
case False
then have qw$\downarrow_p$ = ε **by** (metis ‹qw$\downarrow_q$ = qw› only-one-actor-proj)
then show ?thesis **by** (simp add: reach-root.hyps(6))
qed
next
case (node-step x q w-acc′ ps qw)
then have qw ∈ $\mathcal{L}$ q **by** (meson mem-Collect-eq w-in-infl-lang)
then have qw$\downarrow_q$ = qw **using** w-in-peer-lang-impl-p-actor **by** fastforce
then show ?case
proof (cases q = p) — can't be the case because then concat$_i nfl_{isnottrue}$
case True
then have qw$\downarrow_p$ = qw **using** ‹qw$\downarrow_q$ = qw› **by** blast
then have qw ∈ $\mathcal{L}$ p **using** True ‹qw ∈ $\mathcal{L}$ q› **by** blast
have concat-infl p w (x # q # ps) w-acc′ **by** (simp add: node-step.hyps(6))
then have path-to-root x (x # q # ps) **by** (simp add: node-step.hyps(4))
then have x ≠ q **by** (metis insert-subset mem-Collect-eq node-step.hyps(1,2) sends-of-peer-subset-of-predecessors-in-topology
tree-acyclic)
have (∀ xs ys. (x # q # ps) ≠ xs @ [p] @ ys) **using** True ‹x ≠ q› concat-infl-p-first-or-not-exists node-step.hyps(6) **by** blast
have (x # q # ps) = [x] @ [p] @ ps **using** True **by** auto

```
    then show ?thesis using ‹∀xs ys. x # q # ps ≠ xs · (p # ε · ys)› by blast
  next
    case False
    then have qw↓p = ε  by (metis ‹qw↓q = qw› only-one-actor-proj)
    then show ?thesis  by (simp add: node-step.hyps(7))
  qed
qed


lemma concat-infl-word-exists:
  assumes concat-infl p w ps w and is-root r
  shows ∃w'. concat-infl p w [r] w'
  sorry

lemma concat-infl-mbox:
  assumes concat-infl p w [q] w-acc
  shows w-acc ∈ 𝒯None
proof −
  define xp where xp-def: xp = [q]
  with assms  have concat-infl p w xp w-acc
    by simp
  from this xp-def show w-acc ∈ 𝒯None
  proof (induct)
    case (at-p ps)
    then show ?case sorry
  next
    case (reach-root q qw x w-acc)
    then show ?case sorry
  next
    case (node-step x q w-acc ps qw)
    then show ?case sorry
  qed
qed


lemma concat-infl-children-not-included:
  assumes concat-infl p w ps w-acc and is-parent-of q p
  shows w-acc↓q = ε

  using assms
proof (induct)
  case (at-p ps)
  then show ?case sorry
next
  case (reach-root q qw x w-acc)
  then show ?case sorry
next
  case (node-step x q w-acc ps qw)
  then show ?case sorry
```

qed

lemma concat-infl-w-in-w-acc:
assumes concat-infl p w ps w-acc
shows ∃ xs. w-acc = xs @ w
using assms
proof induct
case (at-p ps)
then show ?case **by** simp
next
case (reach-root q qw x w-acc)
then show ?case **by** (metis append.assoc)
next
case (node-step x q w-acc ps qw)
then show ?case **by** (metis append.assoc)
qed

6.2.4 sync and infl lang relations

7 new formalization

lemma prefix-mbox-trace-valid:
assumes (w@v) $\in \mathcal{L}_\infty$
shows w $\in \mathcal{L}_\infty$
sorry

lemma mbox-exec-to-peer-act:
assumes w $\in \mathcal{T}_{None}\downharpoonleft_!$ **and** (!⟨(i$^{q \rightarrow p}$)⟩) ∈ set w **and** tree-topology
shows ∃ s1 s2 . (s1, !⟨(i$^{q \rightarrow p}$)⟩, s2) $\in \mathcal{R}$ q
sorry

lemma mbox-exec-to-infl-peer-word:
assumes w $\in \mathcal{T}_{None}$
shows w$\downarrow_p \in \mathcal{L}^*$ p
sorry

lemma peer-recvs-in-exec-is-prefix-of-parent-sends:
assumes e $\in \mathcal{T}_{None}$ **and** is-parent-of p q
shows prefix (((e$\downarrow_p$)$\downarrow_?$)$\downarrow_{!?}$) ((((e$\downarrow_q$)$\downarrow_!$)$\downarrow_{\{p,q\}}$)$\downarrow_{!?}$)
sorry

lemma root-infl-word-no-recvs:
assumes is-root p **and** w $\in \mathcal{L}^*$ p

shows $w{\downarrow_!} = w$
proof (rule ccontr)
assume $w{\downarrow_!} \neq w$
then have $\exists$x. x $\in$ set w $\wedge$ is-input x **by** (simp add: not-only-sends-impl-recv)
then obtain x **where** x $\in$ set w **and** is-input x **by** auto
with assms **show** False **sorry**
qed

lemma matching-recvs-word-matches-sends-explicit:
assumes e $\in \mathcal{T}_{None}$ **and** is-parent-of p q
shows $(((e{\downarrow_!}){\downarrow_q}){\downarrow_{\{p,q\}}}){\downarrow_{!?}} = (((\text{add-matching-recvs } (e{\downarrow_!}){\downarrow_?}){\downarrow_p}){\downarrow_{!?}})$
sorry

lemma mbox-exec-recv-append:
assumes $(w \cdot [!\langle(i^{q \rightarrow p})\rangle]) \in \mathcal{T}_{None}$ **and** $w{\downarrow_p} \cdot [?\langle(i^{q \rightarrow p})\rangle] \in \mathcal{L}^*$ p
and $(((((w){\downarrow_q}){\downarrow_!}){\downarrow_{\{p,q\}}}){\downarrow_{!?}}) = ((((w){\downarrow_p}){\downarrow_?}){\downarrow_{!?}})$ **and** is-parent-of p q
shows $w \cdot [!\langle(i^{q \rightarrow p})\rangle] \cdot [?\langle(i^{q \rightarrow p})\rangle] \in \mathcal{T}_{None}$
sorry

lemma no-sign-recv-prefix-to-sign-inv:
assumes prefix $(w{\downarrow_{!?}})$ $(w'{\downarrow_{!?}})$ **and** $w{\downarrow_?} = w$ **and** $w'{\downarrow_?} = w'$
shows prefix w w'
using assms
apply (induct w)
apply auto
sorry

lemma match-exec-and-child-prefix-to-parent-match:
assumes $(((((v'){\downarrow_r}){\downarrow_!}){\downarrow_{\{q,r\}}}){\downarrow_{!?}}) = ((((v'){\downarrow_q}){\downarrow_?}){\downarrow_{!?}})$ **and** prefix $(wq{\downarrow_?})$ $(((v'){\downarrow_q}){\downarrow_?})$
and is-parent-of q r
and $v' \in \mathcal{T}_{None}$
shows $\exists$wr$'$. prefix wr$'$ $((v'){\downarrow_r}) \wedge (((wr'{\downarrow_!}){\downarrow_{\{q,r\}}}){\downarrow_{!?}}) = (((wq){\downarrow_?}){\downarrow_{!?}}) \wedge wr' \in \mathcal{L}^*$
r
sorry

lemma subset-cond-from-child-prefix-and-parent:
assumes subset-condition q r **and** wq $\in \mathcal{L}^*$ q **and** $wr' \cdot x' \in \mathcal{L}^*$ r **and** $(((wr'{\downarrow_!}){\downarrow_{\{q,r\}}}){\downarrow_{!?}})$

$= (((\text{wq})\downarrow_?)\downarrow_{!?})$
shows $\exists$x. (wq $\cdot$ x) $\in \mathcal{L}^*$ q $\wedge$ $(((\text{wq} \cdot \text{x})\downarrow_?)\downarrow_{!?}) = ((((\text{wr}' \cdot \text{x}')\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$
apply (rule ccontr)
sorry

lemma mbox-exec-app-send:
assumes $(\text{e}\downarrow_q \cdot [\text{a}]) \in (\mathcal{L}^*(\text{q}))$ **and** (e) $\in \mathcal{T}_{None}$ **and** is-output a
shows (e $\cdot$ [a]) $\in \mathcal{T}_{None}$
sorry

lemma mbox-trace-to-root-word:
assumes $(\text{v} \cdot [!\langle(\text{i}^{q \rightarrow p})\rangle]) \in \mathcal{T}_{None}\lfloor_!$ **and** is-root q
shows $(\text{v}\downarrow_q \cdot [!\langle(\text{i}^{q \rightarrow p})\rangle]) \in (\mathcal{L}^*(\text{q}))$
sorry

lemma no-shuffle-implies-output-input-exists:
assumes $\neg(\text{w}' \sqcup\!\sqcup_? \text{w})$ **and** $\text{w}\downarrow_? = \text{w}'\downarrow_?$ **and** $\text{w}\downarrow_! = \text{w}'\downarrow_!$
shows $\exists$ xs a ys b zs xs$'$ ys$'$ zs$'$. is-input a $\wedge$ is-output b $\wedge$ w = (xs @ [a] @ ys @ [b] @ zs) $\wedge$
w$'$ = (xs$'$ @ [b] @ ys$'$ @ [a] @ zs$'$)
sorry

lemma exec-append-missing-recvs:
assumes $(((\text{wq} \cdot \text{xs})\downarrow_?)\downarrow_{!?}) = (((((\text{v} \cdot [\text{a}])\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?})$
and (wq $\cdot$ xs) $\in \mathcal{L}^*$ q **and** (v $\cdot$ [a]) $\in \mathcal{T}_{None}\lfloor_!$ **and** e $\in \mathcal{T}_{None}$ **and** $\text{e}\downarrow_q$ = wq
and $\text{e}\downarrow_!$ = (v $\cdot$ [a])
shows (e $\cdot$ xs) $\in \mathcal{T}_{None}$
sorry

lemma diff-peer-word-impl-diff-trace:
assumes $\text{wq}\downarrow_? = (\text{v}'\downarrow_q \cdot [\text{a}])\downarrow_?$ **and** $\text{wq}\downarrow_! = (\text{v}'\downarrow_q \cdot [\text{a}])\downarrow_!$
and $\neg((\text{v}'\downarrow_q \cdot [\text{a}]) \sqcup\!\sqcup_? \text{wq})$ **and** wq $\neq$ $(\text{v}'\downarrow_q \cdot [\text{a}])$
and e $\in \mathcal{T}_{None}$ **and** $\text{e}\downarrow_q$ = wq **and** v$'$ $\in \mathcal{T}_{None}$ **and** (v $\cdot$ [a]) $\in \mathcal{T}_{None}\lfloor_!$ **and** v$'$ = (add-matching-recvs v)
and $\text{v}'\downarrow_q \in \mathcal{L}^*$ q **and** wq $\in \mathcal{L}^*$ q

shows $e\downarrow_! \neq (v' \cdot [a])\downarrow_!$
sorry

lemma subset-cond-from-child-prefix-and-parent-act:
assumes subset-condition q r **and** wq $\in \mathcal{L}^*$ q **and** wr$'$ $\cdot$ $[!\langle(i^{r \to q})\rangle] \in \mathcal{L}^*$ r **and**
$(((wr'\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?}) = (((wq)\downarrow_?)\downarrow_{!?})$
and is-parent-of q r **and** $((\mathcal{L}^*(q)) = (\mathcal{L}^*_{\sqcup\sqcup}(q)))$
shows $(wq \cdot [?\langle(i^{r \to q})\rangle]) \in \mathcal{L}^*$ q $\wedge$ $(((wq \cdot [?\langle(i^{r \to q})\rangle])\downarrow_?)\downarrow_{!?}) = ((((wr' \cdot [!\langle(i^{r \to q})\rangle])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$
proof −
have $\exists$x. $(wq \cdot x) \in \mathcal{L}^*$ q $\wedge$ $(((wq \cdot x)\downarrow_?)\downarrow_{!?}) = ((((wr' \cdot [!\langle(i^{r \to q})\rangle])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$
using
subset-cond-from-child-prefix-and-parent assms **by** blast
then obtain x **where** wqx-def: $(wq \cdot x) \in \mathcal{L}^*$ q **and** wqx-match: $(((wq \cdot x)\downarrow_?)\downarrow_{!?})$
$= ((((wr' \cdot [!\langle(i^{r \to q})\rangle])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$ **by** auto

then obtain xs ys **where** x-shuf: $(xs \cdot ys) \sqcup\sqcup_?$ x **and** $xs\downarrow_? = xs$ **and** $ys\downarrow_! = ys$
using full-shuffle-of **by** blast
then have xsys-recvs: $(((wq \cdot (xs \cdot ys))\downarrow_?)\downarrow_{!?}) = ((((wr' \cdot [!\langle(i^{r \to q})\rangle])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$
using shuffled-keeps-recv-order wqx-match **by** force
have $(wq \cdot xs \cdot ys) \sqcup\sqcup_? (wq \cdot x)$ **using** x-shuf shuffle-prepend **by** auto
then have wq $\cdot$ xs $\cdot$ ys $\in \mathcal{L}^*$ q **by** (metis assms(6) input-shuffle-implies-shuffled-lang
mem-Collect-eq wqx-def)
then have wqxs-L: wq $\cdot$ xs $\in \mathcal{L}^*$ q **using** local.infl-word-impl-prefix-valid **by**
simp
have $(wq \cdot xs)\downarrow_! = wq\downarrow_!$ **by** (simp add: ‹$xs\downarrow_? = xs$› input-proj-output-yields-eps)
have $xs\downarrow_? = (xs \cdot ys)\downarrow_?$ **by** (simp add: ‹$ys\downarrow_! = ys$› output-proj-input-yields-eps)
have $(xs \cdot ys)\downarrow_? = (x)\downarrow_?$ **using** x-shuf **by** (metis shuffled-keeps-recv-order)
then have $xs\downarrow_? = (x)\downarrow_?$ **using** ‹$xs\downarrow_? = (xs \cdot ys)\downarrow_?$› **by** presburger
have $(((wq \cdot x)\downarrow_?)\downarrow_{!?}) = (((wq \cdot xs)\downarrow_?)\downarrow_{!?})$ **by** (simp add: ‹$xs\downarrow_? = x\downarrow_?$›)
then have t0: $((((wr' \cdot [!\langle(i^{r \to q})\rangle])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?}) = (((wq \cdot xs)\downarrow_?)\downarrow_{!?})$ **using**
wqx-match **by** presburger
then have t1: $(wq \cdot xs) \in \mathcal{L}^*$ q $\wedge$ $(((wq \cdot xs)\downarrow_?)\downarrow_{!?}) = ((((wr' \cdot [!\langle(i^{r \to q})\rangle])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$
using wqxs-L **by** presburger
have $xs = [?\langle(i^{r \to q})\rangle]$ **sorry**
then show ?thesis **using** t0 wqxs-L **by** argo
qed

lemma matched-mbox-run-to-sync-run :
assumes mbox-run $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ None (add-matching-recvs w) xcm **and** w $\in \mathcal{T}_{None}\downarrow_!$
shows sync-run $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ w xcs
sorry

7.1 theorem 4.5=>2.

lemma decompose-vq-given-decomposed-vp:
assumes vq$\downarrow_!\downarrow_{\{p,q\}}\downarrow_{!?}$ = v$\downarrow_?\downarrow_{!?}$ **and** v′ ∈ $\mathcal{L}^*{}_{\sqcup\sqcup}$(p) **and** v′ $\sqcup\sqcup_?$ v **and** v ∈ $\mathcal{L}^*$(p) **and** vq ∈ $\mathcal{L}^*$(q)
and is-output b **and** is-input a **and** v = xs · b # a # ys
shows ∃ as bs. vq$\downarrow_!\downarrow_{\{p,q\}}$ = (as$\downarrow_!\downarrow_{\{p,q\}}$ · (!⟨get-message a⟩) # bs$\downarrow_!\downarrow_{\{p,q\}}$)
sorry

end
end

theory *Express*
imports *Defs CommunicatingAutomaton*

begin

context *NetworkOfCA*
begin

8 Synchronizability of Trees

lemma *lemma4-4* :
fixes *w* :: (*′information*, *′peer*) *action word*
and *q* :: *′peer*
assumes *tree-topology* **and** $w \in \mathcal{L}^*(q)$ **and** $q \in \mathcal{P}$
shows ∃ w'. ($w' \in \mathcal{T}_{None} \wedge w'\downarrow_q = w \wedge ((\textit{is-parent-of } p\ q) \longrightarrow\ w'\downarrow_p = \varepsilon)$)

using *assms*
proof (*cases is-root q*)
case *True* — q = r
then have $w \in \mathcal{L}(q)$ **using** *assms*(*2*) *is-in-infl-lang.cases* **by** *blast*
then have $w = w\downarrow_!$ **by** (*meson NetworkOfCA.no-inputs-implies-only-sends-alt NetworkOfCA-axioms True assms*(*1*) *global-to-local-root p-root*)
then have $w\downarrow_? = \varepsilon$ **by** (*simp add*: *output-proj-input-yields-eps*)
then have *t2*: $w = w\downarrow_q$ **by** (*simp add*: ‹$w \in \mathcal{L}\ q$› *w-in-peer-lang-impl-p-actor*)
then have ∀ $p.\ p \neq q \longrightarrow w\downarrow_p = \varepsilon$ **by** (*metis only-one-actor-proj*)
then have *t3*: ($\forall p.\ (p \in \mathcal{P} \wedge \mathcal{P}_?(p) = \{q\}) \longrightarrow\ w\downarrow_p = \varepsilon$) **by** (*metis True assms*(*1*) *global-to-local-root insert-not-empty*)
— need to prove lemma that if w is w of root r, then mbox (unbounded) has a run for it basically construct the configs, where it starts with (I>(r), epsilon>) and each step appends a send to the buffer of the respective receiver
then have $w \in \mathcal{L}(q)$ **by** (*simp add*: ‹$w \in \mathcal{L}\ q$›)
then have *is-root q* **using** *True* **by** *auto*
then have $w \in \mathcal{T}_{None}$ **using** ‹$w \in \mathcal{L}\ q$› *root-lang-is-mbox* **by** *auto*
have $w\downarrow_q = w$ **using** *t2* **by** *auto*

then have (*is-parent-of* p q $\longrightarrow$ $w\downarrow_p = \varepsilon$) **by** (*metis True is-parent-of-rev*(*2*) *iso-tuple-UNIV-I only-one-actor-proj root-defs-eq t3*)
then show *?thesis* **by** (*metis* ‹$w \in \mathcal{T}_{None}$› *t2*)
next
case *False*
then obtain p **where** *q-parent*: *is-parent-of* q p **by** (*metis UNIV-I assms*(*1*) *path-to-root.cases path-to-root-exists*)
then obtain *ps* **where** *p2root*: *path-to-root* p (p # *ps*) **by** (*metis UNIV-I assms*(*1*) *path-to-root-exists path-to-root-rev*)
then have *is-node* q **by** (*metis is-parent-of.cases q-parent*)
have $w \in \mathcal{L}^*(q)$ **using** *assms*(*2*) **by** *auto*
then have *is-parent-of* q p **by** (*simp add*: *q-parent*)

then have $\exists\, w'.\ w' \in \mathcal{L}^*\ p \wedge ((w\downarrow_?)\downarrow_{!?}) = (((w'\downarrow_{\{q,p\}})\downarrow_!)\downarrow_{!?})$ **using** *assms*(*2*) *infl-parent-child-matching-ws* **by** *blast*
then obtain w' **where** *w′-w*: $((w\downarrow_?)\downarrow_{!?}) = (((w'\downarrow_{\{q,p\}})\downarrow_!)\downarrow_{!?})$ **and** *w′-Lp*: $w' \in \mathcal{L}^*\ p$ **by** *blast*
then have $w' \in \mathcal{L}\ p$ **by** (*meson mem-Collect-eq w-in-infl-lang*)
have *tree-topology* **using** *assms*(*1*) **by** *auto*
have *c1*: $((w\downarrow_?)\downarrow_{!?}) = (((w'\downarrow_{\{q,p\}})\downarrow_!)\downarrow_{!?}) \wedge w \in \mathcal{L}(q) \wedge w' \in \mathcal{L}(p) \wedge$ *is-node* q **using** ‹*is-parent-of* q p›
‹*is-tree* ($\mathcal{P}$) ($\mathcal{G}$) $\wedge$ ($\exists\, qa.\ \mathcal{G}\langle\rightarrow q\rangle = \{qa\}$) $\vee$ *is-tree* ($\mathcal{P}$) ($\mathcal{G}$) $\wedge$ ($\exists\, qa.\ \mathcal{P}_?\ q = \{qa\} \vee q \in \mathcal{P}_!\ qa$)› ‹$w' \in \mathcal{L}\ p$›
assms(*2*) *is-in-infl-lang-rev2*(*1*) *w′-w* **by** *blast*

obtain r **where** *is-root* r **using** *assms*(*1*) *root-exists* **by** *blast*
have *path-to-root* q (q # p # *ps*) **using** *p2root p2root-down-step q-parent* **by** *auto*
then have *concat-infl* q w (q # p # *ps*) w **using** *assms*(*1*,*2*) *at-p* **by** *auto*
have $w \in \mathcal{L}(q)$ **by** (*simp add*: *c1*)
then have $w\downarrow_q = w$ **using** *w-in-peer-lang-impl-p-actor* **by** *auto*

obtain *w-acc* **where** *concat-infl* q w [r] *w-acc* **by** (*meson* ‹*concat-infl* q w (q # p # *ps*) w›
‹*is-tree* ($\mathcal{P}$) ($\mathcal{G}$) $\wedge$ $\mathcal{P}_?\ r = \{\}$ $\wedge$ ($\forall\, q.\ r \notin \mathcal{P}_!\ q$) $\vee$ *is-tree* ($\mathcal{P}$) ($\mathcal{G}$) $\wedge$ $\mathcal{G}\langle\rightarrow r\rangle = \{\}$›
concat-infl-word-exists)
then have *w-acc* $\in \mathcal{T}_{None}$ **by** (*simp add*: *concat-infl-mbox*)
have *w-acc*$\downarrow_q = w$ **using** ‹*concat-infl* q w (r # ε) *w-acc*› *concat-infl-actor-consistent* **by** *blast*
then have ($\forall\, p.$ (*is-parent-of* p q) $\longrightarrow$ *w-acc*$\downarrow_p = \varepsilon$) **using** ‹*concat-infl* q w (r # ε) *w-acc*› *concat-infl-children-not-included* **by** *blast*
then show *?thesis* **using** ‹*w-acc* $\in \mathcal{T}_{None}$› ‹*w-acc*$\downarrow_q = w$› **by** *blast*
qed

lemma *lemm4-4-extra*:
fixes w :: ('*information*, '*peer*) *action word*
and q :: '*peer*
assumes *tree-topology* **and** $w \in \mathcal{L}^*(q)$ **and** $q \in \mathcal{P}$
shows $\exists\ w'.\ (w' \in \mathcal{T}_{None} \wedge w'\downarrow_q = w \wedge ((\textit{is-parent-of}\ p\ q) \longrightarrow\ w'\downarrow_p = \varepsilon)) \wedge (\exists\ xs.\ (xs\ @\ w) = w')$
using *assms*
proof (*cases is-root q*)
case *True* — q = r
then have $w \in \mathcal{L}(q)$ **using** *assms*(*2*) *is-in-infl-lang.cases* **by** *blast*
then have $w = w\downarrow_!$ **by** (*meson NetworkOfCA.no-inputs-implies-only-sends-alt NetworkOfCA-axioms True assms*(*1*) *global-to-local-root p-root*)
then have $w\downarrow_? = \varepsilon$ **by** (*simp add*: *output-proj-input-yields-eps*)
then have *t2*: $w = w\downarrow_q$ **by** (*simp add*: ‹$w \in \mathcal{L}\ q$› *w-in-peer-lang-impl-p-actor*)
then have $\forall\ p.\ p \neq q \longrightarrow w\downarrow_p = \varepsilon$ **by** (*metis only-one-actor-proj*)
then have *t3*: $(\forall p.\ (p \in \mathcal{P} \wedge \mathcal{P}_?(p) = \{q\}) \longrightarrow\ w\downarrow_p = \varepsilon)$ **by** (*metis True assms*(*1*) *global-to-local-root insert-not-empty*)
— need to prove lemma that if w is w of root r, then mbox (unbounded) has a run for it basically construct the configs, where it starts with (I>(r), epsilon>) and each step appends a send to the buffer of the respective receiver
then have $w \in \mathcal{L}(q)$ **by** (*simp add*: ‹$w \in \mathcal{L}\ q$›)
then have *is-root q* **using** *True* **by** *auto*
then have $w \in \mathcal{T}_{None}$ **using** ‹$w \in \mathcal{L}\ q$› *root-lang-is-mbox* **by** *auto*
have $w\downarrow_q = w$ **using** *t2* **by** *auto*
then have $(\textit{is-parent-of}\ p\ q \longrightarrow w\downarrow_p = \varepsilon)$ **by** (*metis True is-parent-of-rev*(*2*) *iso-tuple-UNIV-I only-one-actor-proj root-defs-eq t3*)
then show *?thesis* **by** (*metis* ‹$w \in \mathcal{T}_{None}$› *append-self-conv2 t2*)
next
case *False*
then obtain p **where** *q-parent*: *is-parent-of q p* **by** (*metis UNIV-I assms*(*1*) *path-to-root.cases path-to-root-exists*)
then obtain ps **where** *p2root*: *path-to-root* p $(p\ \#\ ps)$ **by** (*metis UNIV-I assms*(*1*) *path-to-root-exists path-to-root-rev*)
then have *is-node q* **by** (*metis is-parent-of.cases q-parent*)
have $w \in \mathcal{L}^*(q)$ **using** *assms*(*2*) **by** *auto*
then have *is-parent-of q p* **by** (*simp add*: *q-parent*)

then have $\exists w'.\ w' \in \mathcal{L}^*\ p \wedge ((w\downarrow_?)\downarrow_{!?}) = (((w'\downarrow_{\{q,p\}})\downarrow_!)\downarrow_{!?})$ **using** *assms*(*2*) *infl-parent-child-matching-ws* **by** *blast*
then obtain w' **where** *w'-w*: $((w\downarrow_?)\downarrow_{!?}) = (((w'\downarrow_{\{q,p\}})\downarrow_!)\downarrow_{!?})$ **and** *w'-Lp*: $w' \in \mathcal{L}^*\ p$ **by** *blast*
then have $w' \in \mathcal{L}\ p$ **by** (*meson mem-Collect-eq w-in-infl-lang*)
have *tree-topology* **using** *assms*(*1*) **by** *auto*
have *c1*: $((w\downarrow_?)\downarrow_{!?}) = (((w'\downarrow_{\{q,p\}})\downarrow_!)\downarrow_{!?}) \wedge w \in \mathcal{L}(q) \wedge w' \in \mathcal{L}(p) \wedge$ *is-node q* **using** ‹*is-parent-of q p*›
‹*is-tree* $(\mathcal{P})$ $(\mathcal{G}) \wedge (\exists\, qa.\ \mathcal{G}\langle\rightarrow q\rangle = \{qa\}) \vee$ *is-tree* $(\mathcal{P})$ $(\mathcal{G}) \wedge (\exists\, qa.\ \mathcal{P}_?\ q = \{qa\} \vee q \in \mathcal{P}_!\ qa)$› ‹$w' \in \mathcal{L}\ p$›

assms(*2*) *is-in-infl-lang-rev2*(*1*) *w′-w* **by** *blast*

obtain *r* **where** *is-root r* **using** *assms*(*1*) *root-exists* **by** *blast*
have *path-to-root q* (*q* # *p* # *ps*) **using** *p2root p2root-down-step q-parent* **by** *auto*
then have *concat-infl q w* (*q* # *p* # *ps*) *w* **using** *assms*(*1*,*2*) *at-p* **by** *auto*
have $w \in \mathcal{L}(q)$ **by** (*simp add*: *c1*)
then have $w{\downarrow}_q = w$ **using** *w-in-peer-lang-impl-p-actor* **by** *auto*

obtain *w-acc* **where** *concat*: *concat-infl q w* [*r*] *w-acc* **by** (*meson* ‹*concat-infl q w* (*q* # *p* # *ps*) *w*›
‹*is-tree* ($\mathcal{P}$) ($\mathcal{G}$) $\wedge$ $\mathcal{P}_?$ *r* = {} $\wedge$ ($\forall$ *q*. *r* $\notin$ $\mathcal{P}_!$ *q*) $\vee$ *is-tree* ($\mathcal{P}$) ($\mathcal{G}$) $\wedge$ $\mathcal{G}\langle\rightarrow r\rangle$ = {}›
concat-infl-word-exists)
then have *w-acc* $\in \mathcal{T}_{None}$ **by** (*simp add*: *concat-infl-mbox*)
have $w\text{-}acc{\downarrow}_q = w$ **using** ‹*concat-infl q w* (*r* # ε) *w-acc*› *concat-infl-actor-consistent* **by** *blast*
then have ($\forall$ *p*. (*is-parent-of p q*) $\longrightarrow$ $w\text{-}acc{\downarrow}_p = \varepsilon$) **using** ‹*concat-infl q w* (*r* # ε) *w-acc*› *concat-infl-children-not-included* **by** *blast*
then have *t1*: *w-acc* $\in \mathcal{T}_{None}$ $\wedge$ $w\text{-}acc{\downarrow}_q = w$ $\wedge$ ((*is-parent-of p q*) $\longrightarrow$ $w\text{-}acc{\downarrow}_p = \varepsilon$) **using** ‹*w-acc* $\in \mathcal{T}_{None}$› ‹$w\text{-}acc{\downarrow}_q = w$› **by** *blast*
have $\exists$ *es*. *w-acc* = *es* @ *w* **using** *concat* **by** (*simp add*: *concat-infl-w-in-w-acc*)
then show *?thesis* **using** *t1* **using** ‹$\forall$ *p*. *is-parent-of p q* $\longrightarrow$ $w\text{-}acc{\downarrow}_p = \varepsilon$› **by** *blast*
qed

lemma *mbox-trace-with-matching-recvs-is-mbox-exec*:
assumes $w \in \mathcal{T}_{None}\lfloor_!$ **and** *tree-topology* **and** *theorem-rightside*
shows (*add-matching-recvs w*) $\in \mathcal{T}_{None}$
using *assms*
proof (*induct length w arbitrary*: *w*)
case *0*
then show *?case* **by** (*simp add*: *eps-in-mbox-execs*)
next
case (*Suc n*)

then obtain *v a* **where** *w-def*: *w* = *v* · [*a*] **and** *v-len*: *length v* = *n* **by** (*metis length-Suc-conv-rev*)
then have $v \in \mathcal{T}_{None}\lfloor_!$ **using** *Suc.prems*(*1*) *prefix-mbox-trace-valid* **by** *blast*
then have *v-IH-prems*: *n* = |*v*| $\wedge$ $v \in \mathcal{T}_{None}\lfloor_!$ $\wedge$ *is-tree* ($\mathcal{P}$) ($\mathcal{G}$) $\wedge$ *theorem-rightside* **using** *Suc.prems*(*3*) *assms*(*2*) *v-len* **by** *blast*

let *?v'* = *add-matching-recvs v*
have *v-IH*: *?v'* $\in \mathcal{T}_{None}$ **using** *v-IH-prems Suc* **by** *blast*
have $(v \cdot [a]) = (v \cdot [a])\downarrow_!$ **using** *Suc.prems(1) w-def* **by** *fastforce*
then obtain *i p q* **where** *a-def*: $a = (!\langle(i^{q \rightarrow p})\rangle)$ **by** (*metis Nil-is-append-conv append1-eq-conv decompose-send neq-Nil-conv*)
then have $\exists$ *s1 s2* . (*s1*, $!\langle(i^{q \rightarrow p})\rangle$, *s2*) $\in \mathcal{R}$ *q* **using** *Suc.prems(1) assms(2) mbox-exec-to-peer-act w-def* **by** *auto*
then have $p \in \mathcal{P}_!(q)$ **by** (*metis CommunicatingAutomaton.SendingToPeers.intros automaton-of-peer get-message.simps(1)*
is-output.simps(1) message.inject output-message-to-act-both-known trans-to-edge)
then have $\mathcal{G}\langle\rightarrow p\rangle = \{q\}$ **by** (*simp add*: *assms(2) local-parent-to-global*)
then have *pq*: *is-parent-of p q* **using** *assms* **by** (*simp add*: *node-parent*)
have $(?v')\downarrow_q \in \mathcal{L}^*$ *q* **using** *mbox-exec-to-infl-peer-word v-IH* **by** *auto*
have *w-sends-0*: $w = ((?v') \cdot [a])\downarrow_!$ **by** (*metis* ‹$v \cdot a \# \varepsilon = (v \cdot a \# \varepsilon)\downarrow_!$› *adding-recvs-keeps-send-order filter-append w-def*)
then have *w-sends-1*: $w = (?v')\downarrow_! \cdot [a]$ **using** ‹$v \in \mathcal{T}_{None}\downarrow_!$› *adding-recvs-keeps-send-order w-def* **by** *fastforce*
have *a-facts*: *is-output a* $\wedge$ *get-actor a* = *q* $\wedge$ *get-object a* = *p* $\wedge$ $p \neq q$ **using** *a-def is-output.simps(1)* **by** (*simp add*: ‹*is-parent-of p q*› *parent-child-diff*)
then have $[a]\downarrow_q = [a]$ **by** *simp*
have $[a]\downarrow_? = \varepsilon$ **using** *a-def a-facts* **by** *simp*
have *v'-q-recvs-inv-to-a*: $(?v'\downarrow_q)\downarrow_? = ((?v' \cdot [a])\downarrow_q)\downarrow_?$ **using** ‹$(a \# \varepsilon)\downarrow_? = \varepsilon$› **by** *auto*

have $p \in \mathcal{P} \wedge q \in \mathcal{P}$ **by** *simp*
then have (*is-parent-of p q*) $\longrightarrow$ ((*subset-condition p q*) $\wedge$ $((\mathcal{L}^*(p)) = (\mathcal{L}^*_{\sqcup\sqcup}(p))))$ **using** *assms(3) theorem-rightside-def* **by** *blast*
then have *theorem-right-pq*: ((*subset-condition p q*) $\wedge$ $((\mathcal{L}^*(p)) = (\mathcal{L}^*_{\sqcup\sqcup}(p))))$ **using** *pq* **by** *auto*

then have *a-send-ok*: $(?v' \cdot [a]) \in \mathcal{T}_{None}$ **using** *a-def Suc assms*
proof (*cases is-root q*)
case *True*
then have $(v\downarrow_q \cdot [!\langle(i^{q \rightarrow p})\rangle]) \in (\mathcal{L}^*(q))$ **using** *mbox-trace-to-root-word*[*of v i q p*] **using** *Suc.prems(1) a-def w-def* **by** *fastforce*

have $?v'\downarrow_q = (?v'\downarrow_q)\downarrow_!$ **using** *root-infl-word-no-recvs*[*of q* $?v'\downarrow_q$] **using** *True* ‹*add-matching-recvs* $v\downarrow_q \in \mathcal{L}^*$ *q*› **by** *presburger*
then have $?v'\downarrow_q \cdot [a] \in \mathcal{L}^*$ *q* **by** (*metis* (*no-types*, *lifting*) ‹$v\downarrow_q \cdot !\langle(i^{q \rightarrow p})\rangle \# \varepsilon \in \mathcal{L}^*$ *q*› ‹w = *add-matching-recvs* $v\downarrow_! \cdot a \# \varepsilon$› *a-def*
append1-eq-conv filter-pair-commutative w-def)
show *?thesis* **using** *mbox-exec-app-send*[*of q ?v' a*] **using** ‹*add-matching-recvs* $v\downarrow_q \cdot a \# \varepsilon \in \mathcal{L}^*$ *q*› *a-facts v-IH* **by** *linarith*
next
case *False*

obtain *e* **where** *e-def*: $e \in \mathcal{T}_{None}$ **and** *e-trace*: $e\downarrow_! = w$ **using** *Suc.prems(1)* **by** *blast*
then obtain *wq* **where** *wq-def*: $wq = e\downarrow_q$ **and** *wq-in-q*: $wq \in \mathcal{L}^*$ *q* **using**

mbox-exec-to-infl-peer-word **by** *presburger*

have *v'a0*: $((?v')\downarrow_q \cdot [a])\downarrow_! = ((?v')\downarrow_q)\downarrow_! \cdot [a]\downarrow_!$ **by** *simp*
have *v'a1*: $((?v')\downarrow_q)\downarrow_! \cdot [a]\downarrow_! = ((?v')\downarrow_q)\downarrow_! \cdot [a]$ **using** *a-facts* **by** *simp*
then have *v'a2*: $((?v')\downarrow_q)\downarrow_! \cdot [a] = v\downarrow_q \cdot [a]$ **by** (*smt* (*verit*, *ccfv-threshold*) ‹$v \cdot a \# \varepsilon = (v \cdot a \# \varepsilon)\downarrow_!$› *adding-recvs-keeps-send-order append1-eq-conv filter-append filter-pair-commutative same-append-eq*)
have $wq\downarrow_! = w\downarrow_q$ **using** *e-trace filter-pair-commutative wq-def* **by** *blast*
have *wq-v'-sends*: $wq\downarrow_! = ((?v')\downarrow_! \cdot [a])\downarrow_q$ **using** ‹$w =$ *add-matching-recvs* $v\downarrow_! \cdot a \# \varepsilon$› ‹$wq\downarrow_! = w\downarrow_q$› **by** *fastforce*
have *v'a3*: $((?v')\downarrow_! \cdot [a])\downarrow_q = ((?v')\downarrow_!)\downarrow_q \cdot [a]\downarrow_q$ **by** *simp*
have *v'a4*: $((?v')\downarrow_!)\downarrow_q \cdot [a]\downarrow_q = ((?v')\downarrow_q)\downarrow_! \cdot [a]\downarrow_q$ **using** *filter-pair-commutative* **by** *blast*
have $[a]\downarrow_q = [a]$ **using** *a-def* **by** *simp*
have *wq-to-v'a-trace*: $wq\downarrow_! = ((?v')\downarrow_q)\downarrow_! \cdot [a]$ **using** ‹$(a \# \varepsilon)\downarrow_q = a \# \varepsilon$› *v'a3 v'a4 wq-v'-sends* **by** *argo*

have *is-node q* **by** (*metis False NetworkOfCA.root-or-node NetworkOfCA-axioms assms*(*2*))
then obtain *r* **where** *is-parent-of q r* **by** (*metis False UNIV-I path-to-root.cases path-to-root-exists*)

have *v'-recvs-match*: $(((?v'\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?} = (((\textit{add-matching-recvs}\ ((?v'\downarrow_!))\downarrow_?)\downarrow_q)\downarrow_{!?})$ **using** *matching-recvs-word-matches-sends-explicit*[*of ?v' q r*] **using** ‹*is-parent-of q r*› *v-IH* **by** *simp*
then have $(((?v'\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?} = (((?v'\downarrow_?)\downarrow_q)\downarrow_{!?})$ **using** ‹$w =$ *add-matching-recvs* $v\downarrow_! \cdot a \# \varepsilon$› *w-def* **by** *fastforce*
then have *wr-0*: $(((?v'\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?} = (((?v'\downarrow_q)\downarrow_?)\downarrow_{!?})$ **by** (*metis filter-pair-commutative*)
then have *e-pref*:*prefix* $(((e\downarrow_q)\downarrow_?)\downarrow_{!?})$ $((((e\downarrow_r)\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** *peer-recvs-in-exec-is-prefix-of-parent-send* *e q r*] **using** ‹*is-parent-of q r*› *e-def* **by** *linarith*
then have *wq-e-pref*: *prefix* $(((wq)\downarrow_?)\downarrow_{!?})$ $((((e\downarrow_r)\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** *wq-def* **by** *fastforce*
have *e-trace2*: $(e\downarrow_!) = ((?v' \cdot [a])\downarrow_!)$ **using** ‹$w = (\textit{add-matching-recvs}\ v \cdot a \# \varepsilon)\downarrow_!$› *e-trace* **by** *blast*
then have *prefix* $(((wq)\downarrow_?)\downarrow_{!?})$ $(((((?v' \cdot [a])\downarrow_r)\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$ **by** (*metis* (*no-types*, *lifting*) *e-pref filter-pair-commutative*
wq-def)
have $((((?v' \cdot [a])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?}) = ((((?v')\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?}) \cdot ((((([a])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$ **by** *simp*
have $(((([a])\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?}) = ((([a])\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** *a-facts* **by** *simp*
have $r \neq q$ **using** ‹*is-parent-of q r*› *parent-child-diff* **by** *blast*
have $p \neq q$ **by** (*simp add*: *a-facts*)
have $r \neq p$ **proof** (*rule ccontr*)
assume $\neg\ r \neq p$
then have $r = p$ **by** *simp*
then have *is-parent-of q p* **using** ‹*is-parent-of q r*› **by** *auto*
then have *g1*: $\mathcal{G}\langle\rightarrow q\rangle = \{p\}$ **using** *is-parent-of-rev* **by** *simp*
then have *e1*: $(p,\ q) \in \mathcal{G}$ **by** *auto*
have *g2*: $\mathcal{G}\langle\rightarrow p\rangle = \{q\}$ **using** *pq is-parent-of-rev* **by** *simp*

then have *e2*: $(q,\ p) \in \mathcal{G}$ **by** *auto*
show *False* **using** *tree-acyclic*[*of* $\mathcal{P}$ $\mathcal{G}$ *p q*] **using** *assms*(*2*) *e1 e2* **by** *auto*
qed
have $[a]\downarrow_{\{q,r\}} = \varepsilon$ **using** *a-facts* **using** ‹$r \neq p$› **by** *auto*
then have $((([a])\downarrow_{\{q,r\}})\downarrow_{!?}) = (\varepsilon)\downarrow_{!?}$ **using** *a-facts* **by** *simp*
then have $((((?v' \cdot [a])\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?}) = ((((?v')\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?})$ **by** *simp*
have $((((?v' \cdot [a])\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?}) = (((e\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** ‹$e\downarrow_{!} = ($*add-matching-recvs* $v \cdot a \ \# \ \varepsilon)\downarrow_{!}$› **by** *presburger*
then have $(((e\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?}) = ((((?v')\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** ‹(*add-matching-recvs* $v \cdot a \ \# \ \varepsilon)\downarrow_{!}\downarrow_{\{q,r\}}\downarrow_{!?} =$ *add-matching-recvs* $v\downarrow_{!}\downarrow_{\{q,r\}}\downarrow_{!?}$›
by *argo*
have *v′-match*: $(((((?v')\downarrow_{!})\downarrow_{r})\downarrow_{\{q,r\}})\downarrow_{!?}) = ((((?v')\downarrow_{?})\downarrow_{q})\downarrow_{!?})$ **using** ‹$w =$ *add-matching-recvs* $v\downarrow_{!} \cdot a \ \# \ \varepsilon$› *v′-recvs-match w-def* **by** *force*
then have *e-v′-match*:$((((e\downarrow_{!})\downarrow_{r})\downarrow_{\{q,r\}})\downarrow_{!?}) = ((((?v')\downarrow_{?})\downarrow_{q})\downarrow_{!?})$ **using** ‹$(a \ \# \ \varepsilon)\downarrow_{\{q,r\}} = \varepsilon$› ‹$w =$ *add-matching-recvs* $v\downarrow_{!} \cdot a \ \# \ \varepsilon$› *e-trace* **by** *force*
then have *wq-recvs-pref*: *prefix* $(((wq)\downarrow_{?})\downarrow_{!?})$ $((((?v')\downarrow_{?})\downarrow_{q})\downarrow_{!?})$ **by** (*metis filter-pair-commutative wq-e-pref*)
have *v′-proj-inv*: $((((?v')\downarrow_{?})\downarrow_{q})\downarrow_{!?}) = ((((?v')\downarrow_{q})\downarrow_{?})\downarrow_{!?})$ **by** (*metis filter-pair-commutative*)

then have *wq-recvs-prefix*: *prefix* $(wq\downarrow_{?})$ $(((?v')\downarrow_{q})\downarrow_{?})$ **by** (*metis wq-recvs-pref filter-recursion no-sign-recv-prefix-to-sign-inv*)

have $(((((?v' \cdot [a])\downarrow_{!})\downarrow_{r})\downarrow_{\{q,r\}})\downarrow_{!?}) = ((((?v' \cdot [a])\downarrow_{?})\downarrow_{q})\downarrow_{!?})$ **by** (*metis* (*no-types*, *lifting*) *e-trace2 e-v′-match filter-pair-commutative v′-q-recvs-inv-to-a*)
have *prefix* $(wq\downarrow_{?})$ $(((?v' \cdot [a])\downarrow_{q})\downarrow_{?})$ **using** *v′-q-recvs-inv-to-a wq-recvs-prefix* **by** *presburger*
have *wq-pref-of-rq-sends*: *prefix* $(((wq)\downarrow_{?})\downarrow_{!?})$ $(((((?v')\downarrow_{!})\downarrow_{r})\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** *v′-match wq-recvs-pref* **by** *argo*
then have *prefix* $(((wq)\downarrow_{?})\downarrow_{!?})$ $((((?v'\downarrow_{r})\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?})$ **by** (*metis filter-pair-commutative*)
have *v′-match-alt*: $(((((?v')\downarrow_{r})\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?}) = ((((?v')\downarrow_{q})\downarrow_{?})\downarrow_{!?})$ **by** (*metis* (*no-types*, *lifting*) *filter-pair-commutative v′-match*)
then have $\exists\, wr'.$ *prefix* wr' $((?v')\downarrow_{r}) \wedge (((wr'\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?}) = (((wq)\downarrow_{?})\downarrow_{!?}) \wedge wr' \in \mathcal{L}^{*}\ r$
using *match-exec-and-child-prefix-to-parent-match*[*of r q ?v′ wq*] ‹*is-parent-of q r*› *v-IH wq-recvs-prefix* **by** *blast*
then obtain wr' x' **where** *v′r-def*: $((?v')\downarrow_{r}) = wr' \cdot x'$ **and** *wr′-match*: $(((wr'\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?}) = (((wq)\downarrow_{?})\downarrow_{!?})$ **and** $wr' \in \mathcal{L}^{*}\ r$ **by** (*meson prefixE*)
have $((?v')\downarrow_{r}) \in \mathcal{L}^{*}\ r$ **using** *mbox-exec-to-infl-peer-word*[*of ?v′ r*] **using** *v-IH* **by** *blast*
then have $wr' \cdot x' \in \mathcal{L}^{*}\ r$ **by** (*simp add*: *v′r-def*)

have $q \in \mathcal{P} \wedge r \in \mathcal{P}$ **by** *simp*
then have (*is-parent-of q r*) $\longrightarrow$ ((*subset-condition q r*) $\wedge$ $((\mathcal{L}^{*}(q)) = (\mathcal{L}^{*}{}_{\sqcup\sqcup}(q))))$ **using** *assms*(*3*) *theorem-rightside-def* **by** *blast*
then have *theorem-right-qr*: ((*subset-condition q r*) $\wedge$ $((\mathcal{L}^{*}(q)) = (\mathcal{L}^{*}{}_{\sqcup\sqcup}(q))))$ **by** (*simp add*: ‹*is-parent-of q r*›)

have $\exists\, x.\ (wq \cdot x) \in \mathcal{L}^{*}\ q \wedge (((wq \cdot x)\downarrow_{?})\downarrow_{!?}) = ((((wr' \cdot x')\downarrow_{!})\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** *subset-cond-from-child-prefix-and-parent*[

of q r wq wr' x'] **using** ‹$wr' \cdot x' \in \mathcal{L}^*$ *r*› *theorem-right-qr wq-in-q wr'-match* **by** *fastforce*

then obtain *x* **where** *wqx-def*: $(wq \cdot x) \in \mathcal{L}^*$ *q* **and** *wqx-match*: $(((wq \cdot x)\downarrow_?)\downarrow_{!?}) = ((((wr' \cdot x')\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{!?})$ **by** *auto*

then have *wqx-match-v'*: $(((wq \cdot x)\downarrow_?)\downarrow_{!?}) = (((((?v' \cdot [a])\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** *e-trace2 e-v'-match v'-match-alt v'-proj-inv v'r-def* **by** *argo*

then obtain *xs ys* **where** *x-shuf*: $(xs \cdot ys) \sqcup\sqcup_? x$ **and** $xs\downarrow_? = xs$ **and** $ys\downarrow_! = ys$ **using** *full-shuffle-of* **by** *blast*

then have *xsys-recvs*: $(((wq \cdot (xs \cdot ys))\downarrow_?)\downarrow_{!?}) = (((((?v' \cdot [a])\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?})$ **by** (*metis* (*mono-tags*, *lifting*) *filter-append shuffled-keeps-recv-order wqx-match-v'*)

have $(wq \cdot xs \cdot ys) \sqcup\sqcup_? (wq \cdot x)$ **using** *x-shuf shuffle-prepend* **by** *auto*

then have $wq \cdot xs \cdot ys \in \mathcal{L}^*$ *q* **by** (*metis UNIV-def* ‹*is-parent-of q r*› ‹$wq \cdot x \in \mathcal{L}^*$ *q*› *assms*(*3*) *input-shuffle-implies-shuffled-lang*
mem-Collect-eq theorem-rightside-def)

then have *wqxs-L*: $wq \cdot xs \in \mathcal{L}^*$ *q* **using** *local.infl-word-impl-prefix-valid* **by** *simp*

have $(wq \cdot xs)\downarrow_! = wq\downarrow_!$ **by** (*simp add*: ‹$xs\downarrow_? = xs$› *input-proj-output-yields-eps*)

have *wqx-match-v'a*: $((((?v' \cdot [a])\downarrow_q)\downarrow_?)\downarrow_{!?}) = (((wq \cdot x)\downarrow_?)\downarrow_{!?})$ **using** *e-trace2 e-v'-match v'-proj-inv v'-q-recvs-inv-to-a wqx-match-v'* **by** *presburger*

have $xs\downarrow_? = (xs \cdot ys)\downarrow_?$ **by** (*simp add*: ‹$ys\downarrow_! = ys$› *output-proj-input-yields-eps*)

have $(xs \cdot ys)\downarrow_? = (x)\downarrow_?$ **using** *x-shuf* **by** (*metis shuffled-keeps-recv-order*)

then have $xs\downarrow_? = (x)\downarrow_?$ **using** ‹$xs\downarrow_? = (xs \cdot ys)\downarrow_?$› **by** *presburger*

have $(((wq \cdot x)\downarrow_?)\downarrow_{!?}) = (((wq \cdot xs)\downarrow_?)\downarrow_{!?})$ **by** (*simp add*: ‹$xs\downarrow_? = x\downarrow_?$›)

then have *xs-recvs*: $(((wq \cdot xs)\downarrow_?)\downarrow_{!?}) = (((((?v' \cdot [a])\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** *wqx-match-v' wqx-match-v'a* **by** *argo*

have *v'-eq*: $(((((?v' \cdot [a])\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?}) = (((?v' \cdot [a])\downarrow_q)\downarrow_?)\downarrow_{!?}$ **using** *e-trace2 e-v'-match v'-proj-inv v'-q-recvs-inv-to-a* **by** *presburger*

then have $(((wq \cdot xs)\downarrow_?)\downarrow_{!?}) = (((?v' \cdot [a])\downarrow_q)\downarrow_?)\downarrow_{!?}$ **using** *xs-recvs* **by** *presburger*

then have $(wq \cdot xs)\downarrow_? = (((?v' \cdot [a])\downarrow_q)\downarrow_?)$ **using** *no-sign-recv-prefix-to-sign-inv*[*of* $(wq \cdot xs)\downarrow_?$ $(((?v' \cdot [a])\downarrow_q)\downarrow_?)$] **by** (*metis filter-recursion no-sign-recv-prefix-to-sign-inv prefix-order.dual-order.antisym*
prefix-order.dual-order.refl)

then have *wqxs-order*: $(wq \cdot xs)\downarrow_? = (((?v' \cdot [a])\downarrow_q)\downarrow_?) \wedge (wq \cdot xs)\downarrow_! = ((?v' \cdot [a])\downarrow_q)\downarrow_!$ **using** ‹$(a \# \varepsilon)\downarrow_q = a \# \varepsilon$› ‹$(wq \cdot xs)\downarrow_! = wq\downarrow_!$› *w-sends-0 w-sends-1 wq-to-v'a-trace* **by** *force*

have *wqxs-trace-match*: $(((wq \cdot xs)\downarrow_?)\downarrow_{!?}) = (((((v \cdot [a])\downarrow_!)\downarrow_r)\downarrow_{\{q,r\}})\downarrow_{!?})$ **using** ‹$v \cdot a \# \varepsilon = (v \cdot a \# \varepsilon)\downarrow_!$› *e-trace e-trace2 w-def xs-recvs* **by** *presburger*

let *?wq* = $wq \cdot xs$

show *?thesis* **using** *wqxs-order*

proof (*cases* $(?v'\downarrow_q \cdot [a]) \sqcup\sqcup_? (?wq)$)

case *True*

then have $(?v'\downarrow_q \cdot [a]) \in (\mathcal{L}^*{}_{\sqcup\sqcup}(q))$ **using** *input-shuffle-implies-shuffled-lang wqxs-L* **by** *blast*

then have $(?v'\downarrow_q \cdot [a]) \in (\mathcal{L}^*(q))$ **using** *theorem-right-qr* **by** *simp*

then show *?thesis* **using** *mbox-exec-app-send*[*of q ?v' a*] **using** *a-facts v-IH* **by** *blast*

next

case *False*
then have $(?v'\downarrow_q \cdot [a]) \neq (?wq)$ **by** (*metis shuffled.refl*)

then have $\neg ((?v'\downarrow_q \cdot [a]) \sqcup\sqcup_? (?wq))$ **using** *False* **by** *blast*
then have $\neg ((?v'\downarrow_q \cdot [a]) \sqcup\sqcup_? (?wq)) \wedge (wq \cdot xs)\downarrow_? = (((?v' \cdot [a])\downarrow_q)\downarrow_?) \wedge (wq \cdot xs)\downarrow_! = ((?v' \cdot [a])\downarrow_q)\downarrow_!$
using *wqxs-order* **by** *blast*

then have $\exists\ xs'\ a'\ ys'\ b'\ zs'\ xs''\ ys''\ zs''.$ *is-input* $a' \wedge$ *is-output* $b' \wedge (?wq) = (xs' @ [a'] @ ys' @ [b'] @ zs') \wedge (?v'\downarrow_q \cdot [a]) = (xs'' @ [b'] @ ys'' @ [a'] @ zs'')$ **using** *no-shuffle-implies-output-input-exists*[*of*

$?wq\ (?v'\downarrow_q \cdot [a])$] **by** (*metis* ‹$(a \# \varepsilon)\downarrow_q = a \# \varepsilon$› *filter-append*)

have *diff-trace-prems*: $?wq\downarrow_? = (?v'\downarrow_q \cdot [a])\downarrow_? \wedge ?wq\downarrow_! = (?v'\downarrow_q \cdot [a])\downarrow_! \wedge \neg((?v'\downarrow_q \cdot [a]) \sqcup\sqcup_? ?wq) \wedge ?wq \neq (?v'\downarrow_q \cdot [a]) \wedge e \in \mathcal{T}_{None} \wedge ?v' \in \mathcal{T}_{None} \wedge (v \cdot [a]) \in \mathcal{T}_{None}\downarrow_! \wedge ?v' =$ (*add-matching-recvs* $v) \wedge ?v'\downarrow_q \in \mathcal{L}^*\ q \wedge ?wq \in \mathcal{L}^*\ q$
by (*metis* (*no-types*, *lifting*) *False Suc.prems*(*1*) ‹$(a \# \varepsilon)\downarrow_q = a \# \varepsilon$› ‹$(wq \cdot xs)\downarrow_! = wq\downarrow_!$›
‹$((wq \cdot xs)\downarrow_?) = (($*add-matching-recvs* $v \cdot a \# \varepsilon)\downarrow_q)\downarrow_?$› ‹*add-matching-recvs* $v\downarrow_q \cdot a \# \varepsilon \neq wq \cdot xs$›
‹*add-matching-recvs* $v\downarrow_q \in \mathcal{L}^*\ q$› *e-def filter-append v'a1 v-IH w-def wq-to-v'a-trace wqxs-L*)

have $(e \cdot xs) \in \mathcal{T}_{None}$ **using** *exec-append-missing-recvs*[*of wq xs r q v a e*] **using** *diff-trace-prems wq-def wqxs-trace-match*
e-trace w-def **by** *blast*
have $(e \cdot xs)\downarrow_q = e\downarrow_q \cdot xs\downarrow_q$ **by** *simp*
have $xs\downarrow_q = xs$ **using** *infl-word-actor-app* **by** (*meson wqxs-L*)
then have $(e \cdot xs)\downarrow_q = ?wq$ **using** ‹$(e \cdot xs)\downarrow_q = e\downarrow_q \cdot xs\downarrow_q$› *wq-def* **by** *presburger*
have $(e \cdot xs)\downarrow_! = e\downarrow_!$ **by** (*simp add*: ‹$xs\downarrow_? = xs$› *input-proj-output-yields-eps*)
have *diff-trace-prems2*: $?wq\downarrow_? = (?v'\downarrow_q \cdot [a])\downarrow_? \wedge ?wq\downarrow_! = (?v'\downarrow_q \cdot [a])\downarrow_! \wedge \neg((?v'\downarrow_q \cdot [a]) \sqcup\sqcup_? ?wq) \wedge ?wq \neq (?v'\downarrow_q \cdot [a]) \wedge (e \cdot xs) \in \mathcal{T}_{None} \wedge (e \cdot xs)\downarrow_q = ?wq \wedge ?v' \in \mathcal{T}_{None} \wedge (v \cdot [a]) \in \mathcal{T}_{None}\downarrow_! \wedge ?v' =$ (*add-matching-recvs* $v) \wedge ?v'\downarrow_q \in \mathcal{L}^*\ q \wedge ?wq \in \mathcal{L}^*\ q$ **using** ‹$(e \cdot xs)\downarrow_q = wq \cdot xs$› ‹$e \cdot xs \in \mathcal{T}_{None}$› *diff-trace-prems* **by** *blast*
then have $(e \cdot xs)\downarrow_! \neq (?v' \cdot [a])\downarrow_!$ **using** *diff-peer-word-impl-diff-trace*
[*of* $?wq\ q\ ?v'\ a\ (e \cdot xs)\ v$] **by** *simp*
then show *?thesis* **using** ‹$(e \cdot xs)\downarrow_! = e\downarrow_!$› *e-trace2* **by** *argo*
qed
qed

then have $((\textit{add-matching-recvs } v)\downarrow_q \ @\ [a]\downarrow_q\ ) \in \mathcal{L}^*\ q$ **using** *mbox-exec-to-infl-peer-word* **by** *fastforce*
then have *q-full*: $((\textit{add-matching-recvs } v)\downarrow_q\ @\ [!\langle(i^{q \rightarrow p})\rangle]) \in \mathcal{L}^*\ q$ **using** *a-def* **by** *simp*
have *v'p-in-L*: $(\textit{add-matching-recvs } v)\downarrow_p \in \mathcal{L}^*\ p$ **using** *mbox-exec-to-infl-peer-word* *v-IH* **by** *blast*

have *v'-recvs-match-pq*: $(((?v'\downarrow_!)\downarrow_q)\downarrow_{\{p,q\}})\downarrow_{!?} = (((\textit{add-matching-recvs } ((?v'\downarrow_!))\downarrow_?)\downarrow_p)\downarrow_{!?})$

using *matching-recvs-word-matches-sends-explicit*[*of ?v' p q*] **using** ‹*is-parent-of p q*› *v-IH* **by** *simp*
then have *v'-recvs-match-pq2*: $(((?v'\downarrow_!)\downarrow_q)\downarrow_{\{p,q\}})\downarrow_{!?} = (((?v'\downarrow_?)\downarrow_p)\downarrow_{!?})$ **using** ‹$w = \textit{add-matching-recvs } v\downarrow_! \cdot a \# \varepsilon$› *w-def* **by** *fastforce*
let $?wp = (?v'\downarrow_p)$
let $?wq\text{-}full = ((\textit{add-matching-recvs } v)\downarrow_q\ @\ [!\langle(i^{q \rightarrow p})\rangle])$

have $(?wp \cdot [?\langle(i^{q \rightarrow p})\rangle]) \in \mathcal{L}^*\ p \wedge (((?wp \cdot [?\langle(i^{q \rightarrow p})\rangle])\downarrow_?)\downarrow_{!?}) = ((((?wq\text{-}full)\downarrow_!)\downarrow_{\{p,q\}})\downarrow_{!?})$
using *subset-cond-from-child-prefix-and-parent-act*[*of p q ?wp* $(?v'\downarrow_q)$ *i*] **by** (*smt* (*verit*, *ccfv-SIG*) *filter-pair-commutative pq q-full theorem-right-pq v'-recvs-match-pq2 v'p-in-L*)
then have $(((?v')\downarrow_p \cdot [(?\langle(i^{q \rightarrow p})\rangle)])) \in \mathcal{L}^*\ p$ **by** *simp*

then have *a-ok0*: $(?v' \cdot ([!\langle(i^{q \rightarrow p})\rangle] \cdot [?\langle(i^{q \rightarrow p})\rangle])) \in \mathcal{T}_{None}$
using *mbox-exec-recv-append*[*of ?v' i q p*] **using** *a-def a-send-ok* **by** (*metis* (*no-types*, *lifting*) *append1-eq-conv append-assoc filter-pair-commutative pq v'-recvs-match-pq w-def w-sends-1*)
have *a-match*: $(\textit{add-matching-recvs } [a]) = ([!\langle(i^{q \rightarrow p})\rangle] \cdot [?\langle(i^{q \rightarrow p})\rangle])$ **using** *a-def* **by** *force*
then have *a-ok*: $((\textit{add-matching-recvs } v) \cdot (\textit{add-matching-recvs } [a])) \in \mathcal{T}_{None}$ **using** *a-ok0* **by** *auto*

then show *?case* **by** (*simp add*: *add-matching-recvs-app w-def*)
qed

theorem *synchronisability-for-trees*:
assumes *tree-topology*
shows *is-synchronisable* $\longleftrightarrow ((\forall\, p \in \mathcal{P}.\ \forall\, q \in \mathcal{P}.\ ((\textit{is-parent-of } p\ q) \longrightarrow ((\textit{subset-condition } p\ q) \wedge ((\mathcal{L}^*(p)) = (\mathcal{L}^*{}_{\sqcup\sqcup}(p))))\ )))$ (**is** $?L \longleftrightarrow ?R$)

proof

assume *?L*

show *?R*
proof *clarify*
fix *p q*
assume *q-parent*: *is-parent-of p q*
have *sync-def*: $\mathcal{T}_{None}\downharpoonleft_{!} = \mathcal{L}_{\mathbf{0}}$ **using** ‹*?L*› **by** *simp*
show *subset-condition p q* $\wedge$ $\mathcal{L}^*$ *p* $=$ $\mathcal{L}^*{}_{\sqcup\sqcup}$ *p*
proof (*rule conjI*)

show *subset-condition p q* **unfolding** *subset-condition-def*
proof *auto*

fix *w w′ x′*
assume *w-Lp*: *is-in-infl-lang p w*
and *w′-Lq*: *is-in-infl-lang q w′*
and *w′-w-match*: *filter* (λx. *is-output x* $\wedge$ (*get-object x* $=$ *q* $\wedge$ *get-actor x* $=$ *p*
$\vee$ *get-object x* $=$ *p* $\wedge$ *get-actor x* $=$ *q*)) $w'\downarrow_{!?} = w\downarrow_{?}\downarrow_{!?}$
and *w′x′-Lq*: *is-in-infl-lang q* $(w' \cdot x')$
then show $\exists$ *wa*. *filter* (λx. *is-output x* $\wedge$ (*get-object x* $=$ *q* $\wedge$ *get-actor x* $=$ *p* $\vee$ *get-object x* $=$ *p* $\wedge$
get-actor x $=$ *q*)) $x'\downarrow_{!?} = wa\downarrow_{!?}$ $\wedge$ ($\exists x$. $wa = x\downarrow_{?}$ $\wedge$ *is-in-infl-lang p* $(w \cdot x)$)
using *w-Lp w′-Lq w′-w-match w′x′-Lq*
proof (*cases is-root q*)
case *True*
then have $(w' \cdot x') \in \mathcal{L}$ *q* **using** *w′x′-Lq w-in-infl-lang* **by** *auto*

then have $(w' \cdot x') \in \mathcal{T}_{None}$ **using** *root-lang-is-mbox True* **by** *blast*

have $w'\downarrow_{!}\downarrow_{\{p,q\}}\downarrow_{!?} = w\downarrow_{?}\downarrow_{!?}$ **using** *w′-w-match* **by** *force*

let *?mix* $=$ (*mix-pair w′ w* [])
have *?mix* $\cdot\, x' \in \mathcal{T}_{None}$ **sorry**
then obtain *t* **where** $t \in \mathcal{L}_{\mathbf{0}} \wedge t \in \mathcal{T}_{None}\downharpoonleft_{!} \wedge t = (?mix \cdot x')\downarrow_{!}$ **using** *sync-def* **by** *fastforce*
then obtain *xc* **where** *t-sync-run* : *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *t xc* **using** *Sync-Traces.simps* **by** *auto*

then have $\exists$ *xcm*. *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs t*) *xcm* **using** *empty-sync-run-to-mbox-run sync-run-to-mbox-run* **by** *blast*

then have *sync-exec*: (*add-matching-recvs t*) $\in \mathcal{T}_{None}$ **using** *Mbox-Traces.intros* **by** *auto*

then have $\exists x$. (*add-matching-recvs t*)$\downarrow_{p} = w \cdot x$ **sorry**
then obtain *x* **where** *x-def*: (*add-matching-recvs t*)$\downarrow_{p} = w \cdot x$ **by** *blast*
then have *w′x′-wx-match*: $(w' \cdot x')\downarrow_{!}\downarrow_{\{p,q\}}\downarrow_{!?} = (w \cdot x)\downarrow_{?}\downarrow_{!?}$ **sorry**

have $(w \cdot x) \in \mathcal{L}^*\ p$ **using** *sync-exec x-def* **by** (*metis mbox-exec-to-infl-peer-word*)
have $\exists\, wa.\ x'{\downarrow_!}{\downarrow_{\{p,q\}}}{\downarrow_{!?}} = wa{\downarrow_{!?}} \wedge (\exists\, x.\ wa = x{\downarrow_?} \wedge$ *is-in-infl-lang* $p\ (w \cdot x))$ **using** $\langle w \cdot x \in \mathcal{L}^*\ p\rangle$ $\langle w'{\downarrow_!}{\downarrow_{\{p,q\}}}{\downarrow_{!?}} = w{\downarrow_?}{\downarrow_{!?}}\rangle$ *w′x′-wx-match* **by** *auto*
then show *?thesis* **by** *simp*
next
case *False*
then have *is-node q* **by** (*metis NetworkOfCA.root-or-node NetworkOfCA-axioms assms*)
then obtain r **where** *qr*: *is-parent-of q r* **by** (*metis False UNIV-I path-from-root.simps path-to-root-exists paths-eq*)

have $(w' \cdot x') \in \mathcal{L}^*\ q$ **by** (*simp add*: *w′x′-Lq*)
then have $\exists\, w''.\ w'' \in \mathcal{L}^*(r) \wedge (((w' \cdot x'){\downarrow_?}){\downarrow_{!?}}) = (((w''{\downarrow_{\{q,r\}}}){\downarrow_!}){\downarrow_{!?}})$
using *infl-parent-child-matching-ws*[*of* $(w' \cdot x')$ $q\ r$] **using** *qr* **by** *blast*
then obtain w'' **where** *w″-w′-match*: $w''{\downarrow_!}{\downarrow_{\{q,r\}}}{\downarrow_{!?}} = (w' \cdot x'){\downarrow_?}{\downarrow_{!?}}$ **and** *w″-def*: $w'' \in \mathcal{L}^*\ r$ **by** (*metis* (*no-types, lifting*) *filter-pair-commutative*)

have $\exists\ e.\ (e \in \mathcal{T}_{None} \wedge e{\downarrow_r} = w'' \wedge ((\textit{is-parent-of}\ q\ r) \longrightarrow e{\downarrow_q} = \varepsilon))$ **using** *lemma4-4*[*of* $w''\ r\ q$] **using** $\langle w'' \in \mathcal{L}^*\ r\rangle$ *assms* **by** *blast*
then obtain e **where** *e-def*: $e \in \mathcal{T}_{None}$ **and** *e-proj-r*: $e{\downarrow_r} = w''$
and *e-proj-q*: $e{\downarrow_q} = \varepsilon$ **using** *qr* **by** *blast*

let *?mix* = (*mix-pair* $w'\ w$ [])

have $e \cdot \textit{?mix} \cdot x' \in \mathcal{T}_{None}$ **sorry**

then obtain t **where** $t \in \mathcal{L}_{\mathbf{0}} \wedge t \in \mathcal{T}_{None}|_! \wedge t = (e \cdot \textit{?mix} \cdot x'){\downarrow_!}$ **using** *sync-def* **by** *fastforce*
then obtain *xc* **where** *t-sync-run* : *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ t *xc* **using** *Sync-Traces.simps* **by** *auto*

then have $\exists\, xcm.$ *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs* t) *xcm* **using** *empty-sync-run-to-mbox-run sync-run-to-mbox-run* **by** *blast*

then have *sync-exec*: (*add-matching-recvs* t) $\in \mathcal{T}_{None}$ **using** *Mbox-Traces.intros* **by** *auto*

then have $\exists\, x.$ (*add-matching-recvs* $t){\downarrow_p} = w \cdot x$ **sorry**
then obtain x **where** *x-def*: (*add-matching-recvs* $t){\downarrow_p} = w \cdot x$ **by** *blast*
then have *w′x′-wx-match*: $(w' \cdot x'){\downarrow_!}{\downarrow_{\{p,q\}}}{\downarrow_{!?}} = (w \cdot x){\downarrow_?}{\downarrow_{!?}}$ **sorry**

have $(w \cdot x) \in \mathcal{L}^*\ p$ **using** *sync-exec x-def* **by** (*metis mbox-exec-to-infl-peer-word*)
have $w'{\downarrow_!}{\downarrow_{\{p,q\}}}{\downarrow_{!?}} = w{\downarrow_?}{\downarrow_{!?}}$ **using** *w′-w-match* **by** *force*
have $\exists\, wa.\ x'{\downarrow_!}{\downarrow_{\{p,q\}}}{\downarrow_{!?}} = wa{\downarrow_{!?}} \wedge (\exists\, x.\ wa = x{\downarrow_?} \wedge$ *is-in-infl-lang* $p\ (w \cdot x))$ **using** $\langle w \cdot x \in \mathcal{L}^*\ p\rangle$ $\langle w'{\downarrow_!}{\downarrow_{\{p,q\}}}{\downarrow_{!?}} = w{\downarrow_?}{\downarrow_{!?}}\rangle$ *w′x′-wx-match* **by** *auto*
then show *?thesis* **by** *simp*
qed
qed

show $\mathcal{L}^*(p) = \mathcal{L}^*{}_{\sqcup\sqcup}(p)$
proof

show $\mathcal{L}^*(p) \subseteq \mathcal{L}^*{}_{\sqcup\sqcup}(p)$ **using** *language-shuffle-subset* **by** *auto*

show $\mathcal{L}^*{}_{\sqcup\sqcup}(p) \subseteq \mathcal{L}^*(p)$
proof
fix v'

assume $v' \in \mathcal{L}^*{}_{\sqcup\sqcup}(p)$
then obtain v **where** *v-orig*: $v' \sqcup\sqcup_{?}\ v$ **and** *orig-in-L*: $v \in \mathcal{L}^*(p)$ **using** *shuffled-infl-lang-impl-valid-shuffle* **by** *auto*
then show $v' \in \mathcal{L}^*(p)$
proof (*induct v v'*)
case (*refl w*)
then show *?case* **by** *simp*
next
case (*swap b a w xs ys*)

then have $\exists\, vq.\ \ vq \in \mathcal{L}^*(q) \wedge ((w{\downarrow_{?}}){\downarrow_{!?}}) = (((vq{\downarrow_{\{p,q\}}}){\downarrow_{!}}){\downarrow_{!?}})$ **using** *infl-parent-child-matching-ws*[*of w p q*] *orig-in-L q-parent* **by** *blast*
then obtain *vq* **where** *vq-v-match*: $((w{\downarrow_{?}}){\downarrow_{!?}}) = (((vq{\downarrow_{\{p,q\}}}){\downarrow_{!}}){\downarrow_{!?}})$ **and** *vq-def*: $vq \in \mathcal{L}^*\ q$ **by** *auto*
have *lem4-4-prems*: *tree-topology* $\wedge\ w \in \mathcal{L}^*(p) \wedge p \in \mathcal{P}$ **using** *assms swap.prems* **by** *auto*
then show *?case* **using** *assms swap vq-v-match vq-def lem4-4-prems*
proof (*cases is-root q*)
case *True*
have $vq \in \mathcal{L}\ q$ **using** *vq-def w-in-infl-lang* **by** *auto*
then have $vq \in \mathcal{T}_{None}$ **using** *root-lang-is-mbox True* **by** *simp*

let $?w' = xs \cdot a \# b \# ys$
have $\exists$ *acc. mix-shuf vq v v' acc* **sorry**
then obtain *mix* **where** *mix-shuf vq v v' mix* **by** *blast*
let *?mix* = *mix*
have $?mix \in \mathcal{T}_{None}$ **sorry**
then obtain t **where** $t \in \mathcal{L}_{\mathbf{0}} \wedge t \in \mathcal{T}_{None}{\downarrow_{!}} \wedge t = (?mix){\downarrow_{!}}$ **using** *sync-def* **by** *fastforce*
then obtain *xc* **where** *t-sync-run* : *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *t xc* **using** *Sync-Traces.simps* **by** *auto*

then have $\exists\, xcm.$ *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs t*) *xcm* **using** *empty-sync-run-to-mbox-run sync-run-to-mbox-run* **by** *blast*

then have *sync-exec*: (*add-matching-recvs t*) $\in \mathcal{T}_{None}$ **using** *Mbox-Traces.intros* **by** *auto*

then have $(\textit{add-matching-recvs } t){\downarrow_p} = \textit{?w}'$ **sorry**
then have $\textit{?w}' \in \mathcal{L}^*\ p$ **using** *sync-exec mbox-exec-to-infl-peer-word* **by** *metis*
then show *?thesis* **by** *simp*
next
case *False*
then have *is-node q* **by** (*metis NetworkOfCA.root-or-node NetworkOfCA-axioms assms*)
then obtain r **where** *qr*: *is-parent-of q r* **by** (*metis False UNIV-I path-from-root.simps path-to-root-exists paths-eq*)
then have $\exists\, vr.\ \ vr \in \mathcal{L}^*(r) \wedge ((vq{\downarrow_?}){\downarrow_{!?}}) = (((vr{\downarrow_{\{q,r\}}}){\downarrow_!}){\downarrow_{!?}})$
using *infl-parent-child-matching-ws*[*of vq q r*] *orig-in-L vq-def* **by** *blast*

then obtain *vr* **where** *vr-def*: $vr \in \mathcal{L}^*(r)$ **and** *vr-vq-match*: $((vq{\downarrow_?}){\downarrow_{!?}}) = (((vr{\downarrow_{\{q,r\}}}){\downarrow_!}){\downarrow_{!?}})$ **by** *blast*

have $\exists\ e.\ (e \in \mathcal{T}_{None} \wedge e{\downarrow_r} = vr \wedge ((\textit{is-parent-of } q\ r) \longrightarrow\ e{\downarrow_q} = \varepsilon))$ **using** *lemma4-4*[*of vr r q*] **using** ‹$vr \in \mathcal{L}^*\ r$› *assms* **by** *blast*
then obtain e **where** *e-def*: $e \in \mathcal{T}_{None}$ **and** *e-proj-r*: $e{\downarrow_r} = vr$
and *e-proj-q*: $e{\downarrow_q} = \varepsilon$ **using** *qr* **by** *blast*

let $\textit{?w}' = xs \cdot a\ \#\ b\ \#\ ys$
have $\exists\ acc.$ *mix-shuf vq v v′ acc* **sorry**
then obtain *mix* **where** *mix-shuf vq v v′ mix* **by** *blast*
let $\textit{?mix} = mix$

have $e \cdot \textit{?mix} \in \mathcal{T}_{None}$ **sorry**

then obtain t **where** $t \in \mathcal{L}_\mathbf{0} \wedge t \in \mathcal{T}_{None}\lfloor_! \wedge t = (e \cdot \textit{?mix}){\downarrow_!}$ **using** *sync-def* **by** *fastforce*
then obtain *xc* **where** *t-sync-run* : *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *t xc* **using** *Sync-Traces.simps* **by** *auto*

then have $\exists\, xcm.$ *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs t*) *xcm* **using** *empty-sync-run-to-mbox-run sync-run-to-mbox-run* **by** *blast*

then have *sync-exec*: (*add-matching-recvs t*) $\in \mathcal{T}_{None}$ **using** *Mbox-Traces.intros* **by** *auto*

then have $(\textit{add-matching-recvs } t){\downarrow_p} = \textit{?w}'$ **sorry**
then have $\textit{?w}' \in \mathcal{L}^*\ p$ **using** *sync-exec mbox-exec-to-infl-peer-word* **by** *metis*
then show *?thesis* **by** *simp*
qed
next
case (*trans w w′ w″*)
then show *?case* **by** *simp*

qed
qed
qed
qed
qed

next

assume *?R*
show *?L* — show that TMbox = TSync, i.e. L> = (i.e. the sends are equal
proof *auto* — cases: w in TMbox, w in TSync
fix w
show $w \in \mathcal{T}_{None} \Longrightarrow w{\downarrow_!} \in \mathcal{L}_{\mathbf{0}}$
proof –
assume $w \in \mathcal{T}_{None}$
then have $(w{\downarrow_!}) \in \mathcal{T}_{None}\lfloor_!$ **by** *blast*

then have *match-exec*: *add-matching-recvs* $(w{\downarrow_!}) \in \mathcal{T}_{None}$
using *mbox-trace-with-matching-recvs-is-mbox-exec* ‹$\forall\, p{\in}\mathcal{P}.\ \forall\, q{\in}\mathcal{P}.$ *is-parent-of* $p\ q \longrightarrow$ *subset-condition* $p\ q \wedge \mathcal{L}^*\ p = \mathcal{L}^*{}_{\sqcup\sqcup}\ p$› *assms theorem-rightside-def*
by *blast*
then obtain *xcm* **where** *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (*add-matching-recvs* $(w{\downarrow_!})$) *xcm* **by** (*metis MboxTraces.cases*)
then show $(w{\downarrow_!}) \in \mathcal{L}_{\mathbf{0}}$ **using** *SyncTraces.simps* ‹$w{\downarrow_!} \in \mathcal{T}_{None}\lfloor_!$› *matched-mbox-run-to-sync-run* **by** *blast*
qed
next — w in TSync –> show that w' (= w with recvs added) is in EMbox
fix w
show $w \in \mathcal{L}_{\mathbf{0}} \Longrightarrow \exists\, w'.\ w = w'{\downarrow_!} \wedge w' \in \mathcal{T}_{None}$
proof –
assume $w \in \mathcal{L}_{\mathbf{0}}$
— For every output in w, Nsync was able to send the respective message and directly receive it
then have $w = w{\downarrow_!}$ **by** (*metis SyncTraces.cases run-produces-no-inputs*(*1*))
then obtain *xc* **where** *w-sync-run* : *sync-run* $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ *w xc* **using** *SyncTraces.simps* ‹$w \in \mathcal{L}_{\mathbf{0}}$› **by** *auto*
then have $w \in \mathcal{L}_\infty$ **using** ‹$w \in \mathcal{L}_{\mathbf{0}}$› *mbox-sync-lang-unbounded-inclusion* **by** *blast*
obtain w' **where** $w' =$ *add-matching-recvs w* **by** *simp*
— then Nmbox can simulate the run of w in Nsync by sending every message first to the mailbox of the receiver and then receiving this message
then show *?thesis*
proof (*cases* $xc = []$) — this case distinction isn't in the paper but i need it here to properly get the simulated run
case *True*
then have *mbox-run* $\mathcal{C}_{\mathcal{I}\mathfrak{m}}$ *None* (w') [] **using** ‹$w' =$ *add-matching-recvs* w› *empty-sync-run-to-mbox-run w-sync-run* **by** *auto*
then show *?thesis* **using** ‹$w \in \mathcal{T}_{None}\lfloor_!$› **by** *blast*
next

case *False*
then obtain *xcm* **where** *sim-run*: *mbox-run* $\mathcal{C}_{\mathcal{I}\mathtt{m}}$ *None* w' *xcm* $\wedge$ ($\forall$ *p*. (*last xcm p*) = ((*last xc*) *p*, ε))
using ‹w' = *add-matching-recvs w*› *sync-run-to-mbox-run w-sync-run* **by** *blast*
then have $w' \in \mathcal{T}_{None}$ **using** *MboxTraces.intros* **by** *auto*
then have $w = w'\downarrow_!$ **using** ‹$w = w\downarrow_!$› ‹w' = *add-matching-recvs w*› *adding-recvs-keeps-send-order* **by** *auto*
then have $(w'\downarrow_!) \in \mathcal{L}_\infty$ **using** ‹$w' \in \mathcal{T}_{None}$› **by** *blast*
then show *?thesis* **using** ‹$w = w'\downarrow_!$› ‹$w' \in \mathcal{T}_{None}$› **by** *blast*
qed
qed
qed
qed

end
end