A constructive proof for FLP

Benjamin Bisping Paul-David Brodmann Tim Jungnickel
Christina Rickmann Henning Seidler Anke Stüber
Arno Wilhelm-Weidner Kirstin Peters Uwe Nestmann

March 17, 2025

Abstract

The impossibility of distributed consensus with one faulty process is a result with important consequences for real world distributed systems e.g., commits in replicated databases. Since proofs are not immune to faults and even plausible proofs with a profound formalism can conclude wrong results, we validate the fundamental result named FLP after Fischer, Lynch and Paterson by using the interactive theorem prover Isabelle/HOL. We present a formalization of distributed systems and the aforementioned consensus problem. Our proof is based on Hagen Völzer's paper *A constructive proof for FLP*. In addition to the enhanced confidence in the validity of Völzer's proof, we contribute the missing gaps to show the correctness in Isabelle/HOL. We clarify the proof details and even prove fairness of the infinite execution that contradicts consensus. Our Isabelle formalization can also be reused for further proofs of properties of distributed systems.

In the following we present the Isabelle implementation of the underlying theory as well as all proofs of the results presented in the paper *Mechanical Verification of a Constructive Proof for FLP* as submitted to the Proceedings of the *seventh conference on Interactive Theorem Proving*, ITP 2016, LNCS.

Contents

1 Multiset

Multiset contains a minimal multiset structure.

```
theory Multiset
imports Main
begin
```

1.1 A minimal multiset theory

Völzer, p. 84, does specify that messages in transit are modelled using multisets.

We decided to implement a tiny structure for multisets, just fitting our needs. These multisets allow to add new values to them, to check for elements existing in a certain multiset, filter elements according to boolean predicates, remove elements and to create a new multiset from a single element.

A multiset for a type is a mapping from the elements of the type to natural numbers. So, we record how often a message has to be processed in the future.

```
type_synonym 'a multiset = "'a ⇒ nat"

abbreviation mElem ::
  "'a ⇒ 'a multiset ⇒ bool" (‹_ ∈# _› 60)
where
  "mElem a ms ≡ 0 < ms a"
```

Hence the union of two multisets is the addition of the number of the elements and therefore the associative and the commutative laws holds for the union.

```
abbreviation mUnion ::
  "'a multiset ⇒ 'a multiset ⇒ 'a multiset" (‹_ ∪# _› 70)
where
  "mUnion msA msB v ≡ msA v + msB v"
```

Correspondingly the subtraction is defined and the commutative law holds.

```
abbreviation mRm ::
  "'a multiset ⇒ 'a ⇒ 'a multiset" (‹_ -# _› 65)
where
  "mRm ms rm v ≡ if v = rm then ms v - 1 else ms v"

abbreviation mSingleton ::
  "'a ⇒ 'a multiset"          (‹{# _ }›)
where
  "mSingleton a v ≡ if a = v then 1 else 0"
```

The lemma **AXc** adds just the fact we need for our proofs about the commutativity of the union of multisets while elements are removed.

```
lemma AXc:
assumes
  "c1 ≠ c2" and
  "c1 ∈# X" and
  "c2 ∈# X"
shows "(A1 ∪# ((A2 ∪# (X -# c2)) -# c1))
      = (A2 ∪# ((A1 ∪# (X -# c1)) -# c2))"
```

```
proof-
  have
    "(A2 ∪# ((A1 ∪# (X -# c1)) -# c2))
         = (A2 ∪# (A1 ∪# ((X -# c1) -# c2)))"
    using assms by auto
  also have
    "... = (A1 ∪# ((A2 ∪# (X -# c2)) -# c1)) "
    using assms by auto
  finally show ?thesis by auto
qed

end
```

2 AsynchronousSystem

AsynchronousSystem defines a message datatype and a transition system locale to model asynchronous distributed computation. It establishes a diamond property for a special reachability relation within such transition systems.

```
theory AsynchronousSystem
imports Multiset
begin
```

The formalization is type-parameterized over

***'p* process identifiers.** Corresponds to the set P in Völzer. Finiteness is not yet demanded, but will be in **FLPSystem**.

***'s* process states.** Corresponds to S, countability is not imposed.

***'v* message payloads.** Corresponds to the interprocess communication part of M from Völzer. The whole of M is captured by `messageValue`.

2.1 Messages

A `message` is either an initial input message telling a process which value it should introduce to the consensus negotiation, a message to the environment communicating the consensus outcome, or a message passed from one process to some other.

```
datatype ('p, 'v) message =
  InMsg 'p bool   (‹<_, inM _>›)
| OutMsg bool     (‹<⊥, outM _>›)
| Msg 'p 'v       (‹<_, _>›)
```

A message value is the content of a message, which a process may receive.

```
datatype 'v messageValue =
  Bool bool
| Value 'v

primrec unpackMessage :: "('p, 'v) message ⇒ 'v messageValue"
where
  "unpackMessage <p, inM b>  = Bool b"
```

```
| "unpackMessage <p, v>       = Value v"
| "unpackMessage <⊥, outM v> = Bool False"

primrec isReceiverOf ::
  "'p ⇒ ('p, 'v) message ⇒ bool"
where
   "isReceiverOf p1 (<p2, inM v>) = (p1 = p2)"
 | "isReceiverOf p1 (<p2, v>) =     (p1 = p2)"
 | "isReceiverOf p1 (<⊥,outM v>) =  False"

lemma UniqueReceiverOf:
fixes
  msg  :: "('p, 'v) message" and
  p q :: 'p
assumes
  "isReceiverOf q msg"
  "p ≠ q"
shows
  "¬ isReceiverOf p msg"
using assms by (cases msg, auto)
```

2.2 Configurations

Here we formalize a configuration as detailed in section 2 of Völzer's paper.

Note that Völzer imposes the finiteness of the message multiset by definition while we do not do so. In `FiniteMessages` We prove the finiteness to follow from the assumption that only finitely many messages can be sent at once.

```
record ('p, 'v, 's) configuration =
  states :: "'p ⇒ 's"
  msgs :: "(('p, 'v) message) multiset"
```

C.f. Völzer: "A step is identified with a message (p, m). A step (p, m) is enabled in a configuration c if $msgs_c$ contains the message (p, m)."

```
definition enabled ::
  "('p, 'v, 's) configuration ⇒ ('p, 'v) message ⇒ bool"
where
  "enabled cfg msg ≡ (msg ∈# msgs cfg)"
```

2.3 The system locale

The locale describing a system is derived by slight refactoring from the following passage of Völzer:

> A process p consists of an initial state $s_p \in S$ and a step transition function, which assigns to each pair (m, s) of a message value m and a process state s a follower state and a finite set of messages (the messages to be sent by p in a step).

```
locale asynchronousSystem =
fixes
  trans :: "'p ⇒ 's ⇒ 'v messageValue ⇒ 's" and
```

```
  sends :: "'p ⇒ 's ⇒ 'v messageValue ⇒ ('p, 'v) message multiset" and
  start :: "'p ⇒ 's"
begin

abbreviation Proc :: "'p set"
where "Proc ≡ (UNIV :: 'p set)"
```

2.4 The step relation

The step relation is defined analogously to Völzer:

> [If enabled, a step may] occur, resulting in a follower configuration c', where c' is obtained from c by removing (p, m) from $msgs_c$, changing p's state and adding the set of messages to $msgs_c$ according to the step transition function associated with p. We denote this by $c \xrightarrow{p,m} c'$.

There are no steps consuming output messages.

```
primrec steps ::
  "('p, 'v, 's) configuration
   ⇒ ('p, 'v) message
   ⇒ ('p, 'v, 's) configuration
   ⇒ bool"
   (‹_ ⊢ _ ↦ _› [70,70,70])
where
  StepInMsg: "cfg1 ⊢ <p, inM v> ↦ cfg2 = (
  (∀ s. ((s = p) ⟶ states cfg2 p = trans p (states cfg1 p) (Bool v))
      ∧ ((s ≠ p) ⟶ states cfg2 s = states cfg1 s))
   ∧ enabled cfg1 <p, inM v>
   ∧ msgs cfg2 = (sends p (states cfg1 p) (Bool v)
                 ∪# (msgs cfg1 -# <p, inM v>)))"
| StepMsg: "cfg1 ⊢ <p, v> ↦ cfg2 = (
  (∀ s. ((s = p) ⟶ states cfg2 p = trans p (states cfg1 p) (Value v))
      ∧ ((s ≠ p) ⟶ states cfg2 s = states cfg1 s))
   ∧ enabled cfg1 <p, v>
   ∧ msgs cfg2 = (sends p (states cfg1 p) (Value v)
                 ∪# (msgs cfg1 -# <p, v>)))"
| StepOutMsg: "cfg1 ⊢ <⊥,outM v> ↦ cfg2 =
    False"
```

The system is distributed and asynchronous in the sense that the processing of messages only affects the process the message is directed to while the rest stays unchanged.

```
lemma NoReceivingNoChange:
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  m :: "('p,'v) message" and
  p :: 'p
assumes
  Step: "cfg1 ⊢ m ↦ cfg2" and
  Rec: "¬ isReceiverOf p m"
shows
  "states cfg1 p = states cfg2 p "
proof(cases m)
```

```
  case (OutMsg b')
  thus ?thesis using Step by auto
next
  case (InMsg q b')
  assume CaseM:  "m = <q, inM b'>"
  with assms have "p ≠ q" by simp
  with Step CaseM show ?thesis by simp
next
  case (Msg q v')
  assume CaseM:  "m = <q, v'>"
  with assms have "p ≠ q" by simp
  with Step CaseM show ?thesis by simp
qed

lemma ExistsMsg:
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  m :: "('p,'v) message"
assumes
  Step: "cfg1 ⊢ m ↦ cfg2"
shows
  "m ∈# (msgs cfg1)"
using assms enabled_def by (cases m, auto)

lemma NoMessageLossStep:
fixes
  cfg1 :: "('p,'v,'s) configuration" and
  cfg2 :: "('p,'v,'s) configuration" and
  p :: 'p and
  m :: "('p,'v) message" and
  m' :: "('p,'v) message"
assumes
  Step: "cfg1 ⊢ m ↦ cfg2" and
  Rec1: "isReceiverOf p m" and
  Rec2: "¬isReceiverOf p m'"
shows
  "msgs cfg1 m' ≤ msgs cfg2 m'"
using assms by (induct m, simp_all, auto)

lemma OutOnlyGrowing:
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  b::bool and
  m::"('p, 'v) message" and
  p::'p
assumes
  "cfg1 ⊢ m ↦ cfg2"
  "isReceiverOf p m"
shows
  "msgs cfg2 <⊥, outM b>
  = (msgs cfg1 <⊥, outM b>) +
    sends p (states cfg1 p) (unpackMessage m) <⊥, outM b>"
```

```
proof(-)
  have "m = <⊥, outM b> ⟹ False" using assms proof(auto) qed
  hence MNotOut: "m ≠ <⊥, outM b>" by auto
  have MsgFunction: "msgs cfg2
                     = ((sends p (states cfg1 p) (unpackMessage m))
                       ∪# ((msgs cfg1) -# m))"
  proof(cases m)
    case (InMsg pa bool)
    then have PaIsP: "pa = p" "(unpackMessage m) = Bool bool"
      using isReceiverOf_def assms(2) by (auto simp add: UniqueReceiverOf)
    hence "cfg1 ⊢ <p, inM bool> ↦ cfg2" using assms(1) InMsg by simp
    hence "msgs cfg2 = (sends p (states cfg1 p) (Bool bool)
                        ∪# (msgs cfg1 -# <p, inM bool>))"
      by simp
    hence "msgs cfg2 = (sends p (states cfg1 p) (Bool bool)
                        ∪# (msgs cfg1 -# m))"
      using PaIsP(1) InMsg by simp
    thus ?thesis using StepInMsg assms PaIsP by simp
  next case (OutMsg b)
    hence False using assms by auto
    thus ?thesis by simp
  next case (Msg pa va)
    hence PaIsP: "pa = p" "(unpackMessage m) = Value va"
      using isReceiverOf_def assms(2) by (auto simp add: UniqueReceiverOf)
    hence "cfg1 ⊢ <p, va> ↦ cfg2" using assms(1) Msg by simp
    hence "msgs cfg2 = (sends p (states cfg1 p) (Value va)
                        ∪# (msgs cfg1 -# <p, va>))" by simp
    hence "msgs cfg2 = (sends p (states cfg1 p) (Value va)
                        ∪# (msgs cfg1 -# m))"
      using PaIsP(1) Msg by simp
    thus ?thesis using StepInMsg assms PaIsP by simp
  qed
  have "((sends p (states cfg1 p) (unpackMessage m))
         ∪# ((msgs cfg1) -# m)) <⊥, outM b>
       = ((sends p (states cfg1 p) (unpackMessage m))
         ∪# (msgs cfg1)) <⊥, outM b>"
    using MNotOut by auto
  thus "msgs cfg2 <⊥, outM b>
       = (msgs cfg1 <⊥, outM b>) +
         sends p (states cfg1 p) (unpackMessage m) <⊥, outM b>"
    using MsgFunction by simp
qed

lemma OtherMessagesOnlyGrowing:
fixes
  cfg1 :: "('p,'v,'s) configuration" and
  cfg2 :: "('p,'v,'s) configuration" and
  p :: 'p and
  m :: "('p,'v) message" and
  m' :: "('p,'v) message"
assumes
  Step: "cfg1 ⊢ m ↦ cfg2" and
```

```
  "m ≠ m'"
shows
  "msgs cfg1 m' ≤ msgs cfg2 m'"
using assms by (cases m, auto)
```

Völzer: "Note that steps are enabled persistently, i.e., an enabled step remains enabled as long as it does not occur."

```
lemma OnlyOccurenceDisables:
fixes
  cfg1 :: "('p,'v,'s) configuration" and
  cfg2 :: "('p,'v,'s) configuration" and
  p :: 'p and
  m :: "('p,'v) message" and
  m' :: "('p,'v) message"
assumes
  Step: "cfg1 ⊢ m ↦ cfg2" and
  En: "enabled cfg1 m'" and
  NotEn: "¬ (enabled cfg2 m')"
shows
  "m = m'"
using assms proof (cases m) print_cases
  case (InMsg p bool)
  with Step have "msgs cfg2 = (sends p (states cfg1 p) (Bool bool)
                              ∪# (msgs cfg1 -# <p, inM bool>))" by auto
  thus "m = m'" using InMsg En NotEn
    by (auto simp add: enabled_def, metis less_nat_zero_code)
next
  case (OutMsg bool)
  with Step show "m = m'" by auto
next
  case (Msg p v)
  with Step have "msgs cfg2 = (sends p (states cfg1 p) (Value v)
                              ∪# (msgs cfg1 -# <p, v>))" by auto
  thus "m = m'" using Msg En NotEn
    by (auto simp add: enabled_def, metis less_nat_zero_code)
qed
```

2.5 Reachability

```
inductive reachable ::
  "  ('p, 'v, 's) configuration
  ⇒ ('p, 'v, 's) configuration
  ⇒ bool"
where
  init: "reachable cfg1 cfg1"
| step: "⟦ reachable cfg1 cfg2; (cfg2 ⊢ msg ↦ cfg3) ⟧
          ⟹ reachable cfg1 cfg3"

lemma ReachableStepFirst:
assumes
  "reachable cfg cfg'"
shows
```

```
  "cfg = cfg' ∨ (∃ cfg1 msg p . (cfg ⊢ msg ↦ cfg1) ∧ enabled cfg msg
                    ∧ isReceiverOf p msg
                    ∧ reachable cfg1 cfg')"
using assms
by (induct rule: reachable.induct, auto,
    metis StepOutMsg ExistsMsg init enabled_def isReceiverOf.simps(1)
    isReceiverOf.simps(2) message.exhaust, metis asynchronousSystem.step)

lemma ReachableTrans:
fixes
  cfg1 cfg2 cfg3 :: "('p, 'v, 's ) configuration" and
  Q :: " 'p set"
assumes
  "reachable cfg1 cfg2" and
  "reachable cfg2 cfg3"
shows "reachable cfg1 cfg3"
proof -
  have "reachable cfg2 cfg3 ⟹ reachable cfg1 cfg2 ⟹ reachable cfg1 cfg3"
  proof (induct rule: reachable.induct, auto)
    fix cfg1' cfg2' msg cfg3'
    assume
      "reachable cfg1 cfg2'"
      "cfg2' ⊢ msg ↦ cfg3'"
    thus "reachable cfg1 cfg3'" using reachable.simps by metis
  qed
  thus ?thesis using assms by simp
qed

definition stepReachable ::
    "('p, 'v, 's) configuration
  ⇒ ('p ,'v) message
  ⇒ ('p, 'v, 's) configuration
  ⇒ bool"
where
  "stepReachable c1 msg c2 ≡
    ∃ c' c''. reachable c1 c' ∧ (c' ⊢ msg ↦ c'') ∧ reachable c'' c2 "

lemma StepReachable:
fixes
  cfg cfg' :: "('p,'v,'s) configuration" and
  msg :: "('p, 'v) message"
assumes
  "reachable cfg cfg'" and
  "enabled cfg msg" and
  "¬ (enabled cfg' msg)"
shows "stepReachable cfg msg cfg'"
using assms
proof(induct rule: reachable.induct, simp)
  fix cfg1 cfg2 msga cfg3
  assume Step: "cfg2 ⊢ msga ↦ cfg3" and
    ReachCfg1Cfg2: "reachable cfg1 cfg2" and
    IV: "(enabled cfg1 msg ⟹ ¬ enabled cfg2 msg
```

```
      ⟹ stepReachable cfg1 msg cfg2)" and
    AssumpInduct: "enabled cfg1 msg" "¬ enabled cfg3 msg"
  have ReachCfg2Cfg3: "reachable cfg2 cfg3" using Step
    by (metis reachable.init reachable.step)
  show "stepReachable cfg1 msg cfg3"
  proof (cases "enabled cfg2 msg")
    assume AssumpEnabled: "enabled cfg2 msg"
    hence "msga = msg" using OnlyOccurenceDisables Step AssumpInduct(2) by blast
    thus "stepReachable cfg1 msg cfg3" using ReachCfg1Cfg2 Step
      unfolding stepReachable_def by (metis init)
  next
    assume AssumpNotEnabled: "¬ enabled cfg2 msg"
    hence "stepReachable cfg1 msg cfg2" using IV AssumpInduct(1) by simp
    thus "stepReachable cfg1 msg cfg3"
      using ReachCfg2Cfg3 ReachableTrans asynchronousSystem.stepReachable_def
      by blast
  qed
qed
```

2.6 Reachability with special process activity

We say that `qReachable cfg1 Q cfg2` iff cfg2 is reachable from cfg1 only by activity of processes from Q.

```
inductive qReachable ::
  "('p,'v,'s) configuration
  ⇒ 'p set
  ⇒ ('p,'v,'s) configuration
  ⇒ bool"
where
  InitQ: "qReachable cfg1 Q cfg1"
| StepQ: "⟦ qReachable cfg1 Q cfg2; (cfg2 ⊢ msg ↦ cfg3) ;
            ∃ p ∈ Q . isReceiverOf p msg ⟧
          ⟹ qReachable cfg1 Q cfg3"
```

We say that `withoutQReachable cfg1 Q cfg2` iff cfg2 is reachable from cfg1 with no activity of processes from Q.

```
abbreviation withoutQReachable ::
   "('p,'v,'s) configuration
  ⇒ 'p set
  ⇒ ('p,'v,'s) configuration
  ⇒ bool"
where
  "withoutQReachable cfg1 Q cfg2 ≡
    qReachable cfg1 ((UNIV :: 'p set ) - Q) cfg2"
```

Obviously q-reachability (and thus also without-q-reachability) implies reachability.

```
lemma QReachImplReach:
fixes
  cfg1 cfg2::  "('p, 'v, 's ) configuration" and
  Q :: " 'p set"
assumes
  "qReachable cfg1 Q cfg2"
```

```
shows
  "reachable cfg1 cfg2"
using assms
proof (induct rule: qReachable.induct)
  case InitQ thus ?case using reachable.simps by blast
next
  case StepQ thus ?case using reachable.step by simp
qed

lemma QReachableTrans:
fixes cfg1 cfg2 cfg3 :: "('p, 'v, 's ) configuration" and
  Q :: " 'p set"
assumes "qReachable cfg2 Q cfg3" and
  "qReachable cfg1 Q cfg2"
shows "qReachable cfg1 Q cfg3"
using assms
proof (induct rule: qReachable.induct, simp)
  case (StepQ)
  thus ?case using qReachable.simps by metis
qed

lemma NotInQFrozenQReachability:
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  p :: 'p and
  Q :: "'p set"
assumes
  "qReachable cfg1 Q cfg2" and
  "p ∉ Q"
shows
  "states cfg1 p = states cfg2 p"
using assms
proof(induct rule: qReachable.induct, auto)
  fix cfg1' Q' cfg2' msg cfg3 p'
  assume "qReachable cfg1' Q' cfg2'"
  assume Step: "cfg2' ⊢ msg ↦ cfg3"
  assume Rec: "isReceiverOf p' msg"
  assume "p' ∈ Q'"  "p ∉ Q'"
  hence notEq: "p ≠ p'" by blast
  with Rec have "¬ (isReceiverOf p msg)" by (cases msg, simp_all)
  thus "states cfg2' p = states cfg3 p"
    using Step NoReceivingNoChange by simp
qed

corollary WithoutQReachablFrozenQ:
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  p :: 'p and
  Q :: "'p set"
assumes
  Steps: "withoutQReachable cfg1 Q cfg2" and
  P: "p ∈ Q"
```

```
shows
  "states cfg1 p = states cfg2 p"
using assms NotInQFrozenQReachability by simp

lemma NoActivityNoMessageLoss :
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  p :: 'p and
  Q :: "'p set" and
  m' :: "('p, 'v) message"
assumes
  "qReachable cfg1 Q cfg2" and
  "p ∉ Q" and
  "isReceiverOf p m'"
shows
  "(msgs cfg1 m') ≤ (msgs cfg2 m')"
using assms
proof (induct rule: qReachable.induct, simp)
  case (StepQ cfg1' Q' cfg2' msg cfg3)
  then obtain p' where
    P': "p' ∈ Q'" "isReceiverOf p' msg" "p ≠ p'" by blast
  with assms(3) have "¬(isReceiverOf p' m')"
    by (cases m', simp_all)
  with NoMessageLossStep StepQ(3) P'
    have "msgs cfg2' m' ≤ msgs cfg3 m'"
    by simp
  with StepQ
    show "msgs cfg1' m' ≤ msgs cfg3 m'" by simp
qed

lemma NoMessageLoss:
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  p :: 'p and
  Q :: "'p set" and
  m' :: "('p, 'v) message"
assumes
  "withoutQReachable cfg1 Q cfg2" and
  "p ∈ Q" and
  "isReceiverOf p m'"
shows
  "(msgs cfg1 m') ≤ (msgs cfg2 m')"
using assms NoActivityNoMessageLoss by simp

lemma NoOutMessageLoss:
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  v :: bool
assumes
  "reachable cfg1 cfg2"
shows
  "(msgs cfg1 <⊥, outM v>) ≤ (msgs cfg2 <⊥, outM v>)"
```

```
using assms
proof(induct rule: reachable.induct, auto)
  fix cfg1 cfg' msg cfg2
  assume AssInduct:
    "reachable cfg1 cfg'"
    "msgs cfg1 <⊥, outM v> ≤ msgs cfg' <⊥, outM v>"
    "cfg' ⊢ msg ↦ cfg2"
  from AssInduct(3) have "msgs cfg' <⊥, outM v> ≤ msgs cfg2 <⊥, outM v>"
    by (cases msg, auto)
  with AssInduct(2) show " msgs cfg1 <⊥, outM v> ≤ msgs cfg2 <⊥, outM v>"
    using le_trans by blast
qed

lemma StillEnabled:
fixes
  cfg1 cfg2:: "('p,'v,'s) configuration" and
  p :: 'p and
  msg :: "('p,'v) message" and
  Q :: "'p set"
assumes
  "withoutQReachable cfg1 Q cfg2" and
  "p ∈ Q" and
  "isReceiverOf p msg" and
  "enabled cfg1 msg"
shows
  "enabled cfg2 msg"
using assms enabled_def NoMessageLoss
  by (metis le_0_eq neq0_conv)
```

2.7 Initial reachability

```
definition initial ::
  "('p, 'v, 's) configuration ⇒ bool"
where
  "initial cfg ≡
        (∀ p::'p . (∃ v::bool . ((msgs cfg (<p, inM v>)) = 1)))
      ∧ (∀ p m1 m2 . ((m1 ∈# (msgs cfg)) ∧ (m2 ∈# (msgs cfg))
         ∧ isReceiverOf p m1 ∧ isReceiverOf p m2) ⟶ (m1 = m2))
      ∧ (∀ v::bool . (msgs cfg) (<⊥, outM v>) = 0)
      ∧ (∀ p v. (msgs cfg) (<p, v>) = 0)
      ∧ states cfg = start"

definition initReachable ::
  "('p, 'v, 's) configuration ⇒ bool"
where
  "initReachable cfg ≡ ∃cfg0 . initial cfg0 ∧ reachable cfg0 cfg"

lemma InitialIsInitReachable :
assumes "initial c"
shows "initReachable c"
  using assms reachable.init
  unfolding initReachable_def by blast
```

2.8 Diamond property of reachability

```
lemma DiamondOne:
fixes
  cfg cfg1 cfg2 :: "('p,'v,'s) configuration" and
  p q :: 'p and
  m m' :: "('p,'v) message"
assumes
  StepP: "cfg ⊢ m  ↦ cfg1" and
  PNotQ: "p ≠ q" and
  Rec: "isReceiverOf p m" "¬ (isReceiverOf p m')" and
  Rec': "isReceiverOf q m'" "¬ (isReceiverOf q m)" and
  StepWithoutP: "cfg ⊢ m' ↦ cfg2"
shows
  "∃ cfg' :: ('p,'v,'s) configuration . (cfg1 ⊢ m' ↦ cfg')
        ∧ (cfg2 ⊢ m ↦ cfg')"
proof (cases m)
  case (InMsg p b)
  from StepWithoutP ExistsMsg have " m' ∈# (msgs cfg) " by simp
  hence "m' ∈# (msgs cfg1)"
    using StepP Rec NoMessageLossStep le_neq_implies_less le_antisym
    by (metis  gr_implies_not0 neq0_conv)
  hence EnM': "enabled cfg1 m'" using enabled_def by auto
  from StepP ExistsMsg have "m ∈# (msgs cfg) " by simp
  hence "m ∈# (msgs cfg2)"
    using StepWithoutP Rec' NoMessageLossStep
    by (metis le_0_eq neq0_conv)
  hence EnM: "enabled cfg2 m" using enabled_def by auto
  assume CaseM:  "m = <p, inM b>"

  thus ?thesis
  proof (cases m')
    case (OutMsg b')
    thus ?thesis using StepWithoutP by simp
  next
    case (InMsg q b')
    define cfg' where "cfg' = ⦇states = λs. (
      if s = q then
        trans q (states cfg q) (Bool b')
      else if s = p then
        trans p (states cfg p) (Bool b)
      else
        states cfg s),
      msgs = ((sends q (states cfg q) (Bool b'))
        ∪# (((sends p (states cfg p) (Bool b))
            ∪# ((msgs cfg)-# m)) -# m'))⦈ "
    have StepP': "(cfg1 ⊢ m' ↦ cfg')"
      using StepP  EnM' Rec
      unfolding cfg'_def InMsg CaseM by auto
    moreover from EnM have "(cfg2 ⊢ m ↦ cfg')"
      using InMsg cfg'_def StepP StepP' StepWithoutP NoReceivingNoChange
            Rec' CaseM EnM'
    proof (simp, clarify)
```

```
      assume msgCfg:
        "msgs cfg1 = (sends p (states cfg p) (Bool b)
                     ∪# (msgs cfg -# <p, inM b>))"
        "msgs cfg2 = (sends q (states cfg q) (Bool b')
                     ∪# (msgs cfg -# <q, inM b'>))"
      have "enabled cfg m" "enabled cfg m'"
        using StepP StepWithoutP CaseM InMsg
        by auto
      with msgCfg show
        "(sends q (states cfg q) (Bool b') ∪# (msgs cfg1 -# <q, inM b'>)) =
         (sends p (states cfg p) (Bool b) ∪# (msgs cfg2 -# <p, inM b>))"
        using CaseM InMsg StepP StepWithoutP Rec' AXc[of "m'" "m" "msgs cfg"
          "sends q (states cfg q) (Bool b')"
          "sends p (states cfg p) (Bool b)"]
        unfolding enabled_def
        by metis
    qed
    ultimately show ?thesis by blast
  next
    case (Msg q v')
    define cfg' where "cfg' = (|states = λs. (
      if s = q then
        trans q (states cfg q) (Value v')
      else if s = p then
        trans p (states cfg p) (Bool b)
      else
        states cfg s),
      msgs = ((sends q (states cfg q) (Value v'))
        ∪# (((sends p (states cfg p) (Bool b))
            ∪# ((msgs cfg)-# m))
            -# m'))|)"
    have StepP': "(cfg1 ⊢ m' ↦ cfg')"
      using StepP EnM' Rec
      unfolding Msg CaseM cfg'_def by auto

    moreover from EnM have "(cfg2 ⊢ m ↦ cfg')"
      using Msg cfg'_def StepP StepP' StepWithoutP NoReceivingNoChange Rec' CaseM
EnM'
    proof (simp,clarify)
      assume msgCfg1:
        "msgs cfg1 = (sends p (states cfg p) (Bool b)
                      ∪# (msgs cfg -# <p, inM b>))"
        "msgs cfg2 = (sends q (states cfg q) (Value v')
                      ∪# (msgs cfg -# <q,  v'>))"
      have "enabled cfg m" "enabled cfg m'"
        using StepP StepWithoutP CaseM Msg
        by auto
      with msgCfg1 show
        "(sends q (states cfg q) (Value v') ∪# (msgs cfg1 -# <q, v'>)) =
         (sends p (states cfg p) (Bool b) ∪# (msgs cfg2 -# <p, inM b>))"
        using CaseM Msg StepP StepWithoutP Rec' AXc[of "m'" "m" "msgs cfg"
          "sends q (states cfg q) (Value v')"
```

```
          "sends p (states cfg p) (Bool b)"]
        unfolding enabled_def by metis
    qed
    ultimately show ?thesis by blast
  qed
next
  case (OutMsg b)
  thus ?thesis using StepP by simp
next
  case (Msg p v)
  from StepWithoutP ExistsMsg have " m' ∈# (msgs cfg) " by simp
  hence "m' ∈# (msgs cfg1)"
    using StepP Rec NoMessageLossStep le_neq_implies_less le_antisym
    by (metis  gr_implies_not0 neq0_conv)
  hence EnM': "enabled cfg1 m'" using enabled_def by auto
  from StepP ExistsMsg have "m ∈# (msgs cfg) " by simp
  hence "m ∈# (msgs cfg2)"
    using StepWithoutP Rec' NoMessageLossStep
    by (metis le_0_eq neq0_conv)
  hence EnM: "enabled cfg2 m" using enabled_def by auto
  assume CaseM:  "m = <p, v>"
  thus ?thesis
  proof (cases m')
    case (OutMsg b')
    thus ?thesis using StepWithoutP by simp
  next
    case (InMsg q b')
    define cfg' where "cfg' = (|states = λs. (
      if s = q then
        trans q (states cfg q) (Bool b')
      else if s = p then
        trans p (states cfg p) (Value v)
      else
        states cfg s),
      msgs = ((sends q (states cfg q) (Bool b'))
        ∪# (((sends p (states cfg p) (Value v))
        ∪# ((msgs cfg)-# m))
        -# m'))|) "
    hence StepP': "(cfg1 ⊢ m' ↦ cfg')"
      using StepP InMsg EnM' Rec CaseM
      by auto
    moreover from EnM have "(cfg2 ⊢ m ↦ cfg')"
      using InMsg cfg'_def StepP StepP' StepWithoutP NoReceivingNoChange Rec'
            CaseM EnM'
    proof (simp, clarify)
      assume msgCfg:
        "msgs cfg1 = (sends p (states cfg p) (Value v)
          ∪# (msgs cfg -# <p, v>))"
        "msgs cfg2 = (sends q (states cfg q) (Bool b')
          ∪# (msgs cfg -# <q, inM b'>))"
      have "enabled cfg m" "enabled cfg m'"
        using StepP StepWithoutP CaseM InMsg by auto
```

```
      with msgCfg show " (sends q (states cfg q) (Bool b')
                        ∪# (msgs cfg1 -# <q, inM b'>))
                      = (sends p (states cfg p) (Value v)
                        ∪# (msgs cfg2 -# <p, v>))"
        using CaseM StepP StepWithoutP Rec' InMsg AXc[of "m'" "m" "msgs cfg"
          "sends q (states cfg q) (Bool b')"
          "sends p (states cfg p) (Value v)"]
          unfolding enabled_def by metis
    qed
    ultimately show ?thesis by blast
  next
    case (Msg q v')
    define cfg' where "cfg' = (|states = λs. (
      if s = q then
        trans q (states cfg q) (Value v')
      else if s = p then
        trans p (states cfg p) (Value v)
      else
        states cfg s),
      msgs = ((sends q (states cfg q) (Value v'))
        ∪# (((sends p (states cfg p) (Value v))
        ∪# ((msgs cfg)-# m))
        -# m'))|) "
    hence StepP': "(cfg1 ⊢ m' ↦ cfg')"
      using StepP Msg EnM' Rec CaseM by auto
    moreover from EnM have "(cfg2 ⊢ m ↦ cfg')"
      using Msg cfg'_def StepP StepP' StepWithoutP NoReceivingNoChange Rec' CaseM
EnM'
    proof (simp, clarify)
      assume msgCfg:
        "msgs cfg1 = (sends p (states cfg p) (Value v)
          ∪# (msgs cfg -# <p, v>))"
        "msgs cfg2 = (sends q (states cfg q) (Value v')
          ∪# (msgs cfg -# <q, v'>))"
      have "enabled cfg m" "enabled cfg m'"
        using StepP StepWithoutP CaseM Msg by auto

      with msgCfg show " (sends q (states cfg q) (Value v')
                        ∪# (msgs cfg1 -# <q, v'>))
                      = (sends p (states cfg p) (Value v)
                        ∪# (msgs cfg2 -# <p, v>))"
        using CaseM StepP StepWithoutP Rec' Msg
          AXc[of "m'" "m" "msgs cfg" "sends q (states cfg q) (Value v')"
          "sends p (states cfg p) (Value v)"]
        unfolding enabled_def by metis
    qed
    ultimately show ?thesis by blast
  qed
qed

lemma DiamondTwo:
fixes
```

```
  cfg cfg1 cfg2 :: "('p,'v,'s) configuration" and
  Q :: "'p set" and
  msg :: "('p, 'v) message"
assumes
  QReach: "qReachable cfg Q cfg1" and
  Step: "cfg  ⊢ msg ↦ cfg2"
        "∃p∈Proc - Q. isReceiverOf p msg"
shows
  "∃ (cfg' :: ('p,'v,'s) configuration) . (cfg1  ⊢ msg ↦ cfg')
  ∧ qReachable cfg2 Q cfg'"
using assms
proof(induct rule: qReachable.induct)
  fix cfg Q
  have "qReachable cfg2 Q cfg2" using qReachable.simps(1) by blast
  moreover assume "cfg ⊢ msg ↦ cfg2" "∃p∈UNIV - Q. isReceiverOf p msg"
  ultimately have "(cfg ⊢ msg ↦ cfg2) ∧ qReachable cfg2 Q cfg2" by blast
  thus "∃cfg'.(cfg ⊢ msg ↦ cfg') ∧ qReachable cfg2 Q cfg'" by blast
next
  fix cfg Q cfg1' msga cfg1
  assume   "(cfg ⊢ msg ↦ cfg2)"
    "(∃p∈UNIV - Q. isReceiverOf p msg)"
    "((cfg ⊢ msg ↦ cfg2) ⟹
     (∃p∈UNIV - Q. isReceiverOf p msg) ⟹
     (∃cfg'. (cfg1' ⊢ msg ↦ cfg') ∧ qReachable cfg2 Q cfg'))"
  hence "(∃cfg'. (cfg1' ⊢ msg ↦ cfg') ∧ (∃p∈UNIV - Q. isReceiverOf p msg)
        ∧ qReachable cfg2 Q cfg')" by blast
  then obtain cfg' where Cfg': "(cfg1' ⊢ msg ↦ cfg')"
    "(∃p∈UNIV - Q. isReceiverOf p msg)" "qReachable cfg2 Q cfg'" by blast
  then obtain p where P: "p∈UNIV - Q" "isReceiverOf p msg" by blast
  assume Step2: "(cfg1' ⊢ msga ↦ cfg1)"
    "(∃p∈Q. isReceiverOf p msga)"
    "(qReachable cfg Q cfg1')"
  then obtain p' where P': "p'∈Q" "isReceiverOf p' msga" by blast
  from P'(1) P(1) have notEq: "p ≠ p'" by blast
  with P(2) P'(2) have "¬ isReceiverOf p' msg" "¬ isReceiverOf p msga"
    using UniqueReceiverOf[of p' msga p] UniqueReceiverOf[of p msg p']
    by auto
  with notEq P'(2) P(2) Cfg'(1) Step2(1) have
    "∃cfg''. (cfg' ⊢ msga ↦ cfg'') ∧ (cfg1 ⊢ msg ↦ cfg'')"
    using DiamondOne by simp
  then obtain cfg'' where Cfg'': "cfg' ⊢ msga ↦ cfg''" "cfg1 ⊢ msg ↦ cfg''"
    by blast
  from Cfg''(1) Step2(2) Cfg'(3) have "qReachable cfg2 Q cfg''"
    using qReachable.simps[of cfg2 Q cfg''] by auto
  with Cfg'(2) Cfg''(2) have
    "(cfg1 ⊢ msg ↦ cfg'') ∧ qReachable cfg2 Q cfg''"  by simp
  thus "∃cfg'. (cfg1 ⊢ msg ↦ cfg') ∧ qReachable cfg2 Q cfg'" by blast
qed
```

Proposition 1 of Völzer.

```
lemma Diamond:
fixes
```

```
  cfg cfg1 cfg2 :: "('p,'v,'s) configuration" and
  Q :: "'p set"
assumes
  QReach: "qReachable cfg Q cfg1" and
  WithoutQReach: "withoutQReachable cfg Q cfg2"
shows
  "∃ cfg'. withoutQReachable cfg1 Q cfg'
     ∧ qReachable cfg2 Q cfg'"
proof -
  define notQ where "notQ ≡ UNIV - Q"
  with WithoutQReach have "qReachable cfg notQ cfg2" by simp
  thus ?thesis using QReach notQ_def
  proof (induct rule: qReachable.induct)
    fix cfg2
    assume "qReachable cfg2 Q cfg1"
    moreover have "qReachable cfg1 (UNIV - Q) cfg1"
      using qReachable.simps by blast
    ultimately show
      "∃cfg'. qReachable cfg1 (UNIV - Q) cfg'
        ∧ qReachable cfg2 Q cfg'"
      by blast
  next
    fix cfg cfg2' cfg2 msg
    assume Ass1: " qReachable cfg Q cfg1" "qReachable cfg (UNIV - Q) cfg2' "
    assume Ass2:  "cfg2' ⊢ msg ↦ cfg2" "∃p∈UNIV - Q. isReceiverOf p msg"
    assume "qReachable cfg Q cfg1 ⟹ ∃cfg'. qReachable cfg1 (UNIV - Q) cfg'
      ∧ qReachable cfg2' Q cfg'"
    with Ass1(1) have "∃cfg'. qReachable cfg1 (UNIV - Q) cfg'
      ∧ qReachable cfg2' Q cfg'" by blast
    then obtain cfg' where
      Cfg'1: "qReachable cfg2' Q cfg'" and
      Cfg':  "qReachable cfg1 (UNIV - Q) cfg'" by blast
    from Cfg'1 Ass2 have
      "∃cfg''.(cfg' ⊢ msg ↦ cfg'') ∧ qReachable cfg2 Q cfg''"
      using DiamondTwo by simp
    then obtain cfg'' where
      Cfg'': "cfg' ⊢ msg ↦ cfg''" "qReachable cfg2 Q cfg''" by blast
    from Cfg' Cfg''(1) Ass2(2) have "qReachable cfg1 (UNIV - Q) cfg''"
      using qReachable.simps[of cfg1 "UNIV-Q" cfg''] by auto
    with Cfg'' show
      "∃cfg'. qReachable cfg1 (UNIV - Q) cfg'
        ∧ qReachable cfg2 Q cfg'" by blast
  qed
qed
```

2.9 Invariant finite message count

```
lemma FiniteMessages:
fixes
  cfg  :: "('p, 'v, 's) configuration"
assumes
  FiniteProcs: "finite Proc" and
```

```
  FiniteSends: "⋀ p s m. finite {v. v ∈# (sends p s m)}" and
  InitReachable: "initReachable cfg"
shows "finite {msg . msg ∈# msgs cfg}"
proof(-)
  have "∃ init . initial init ∧ reachable init cfg" using assms
    unfolding initReachable_def by simp
  then obtain init where Init: "initial init" "reachable init cfg"
    by blast
  have InitMsgs:"{msg . msg ∈# msgs init}
    = { msg . (msg ∈# msgs init) ∧ (∃ p v. <p, v> = msg)}
      ∪ { msg . (msg ∈# msgs init) ∧ (∃ v.  <⊥, outM v> = msg)}
      ∪ { msg . (msg ∈# msgs init) ∧ (∃ p v.  <p, inM v> = msg)}"
       by (auto,metis message.exhaust)
  have A:"{ msg . (msg ∈# msgs init) ∧ (∃ p v. <p, v> = msg)} = {}"
  using initial_def[of init] Init(1)by (auto, metis less_not_refl3)
  have B:"{ msg . (msg ∈# msgs init) ∧ (∃ v.  <⊥, outM v> = msg)} = {}"
  using initial_def[of init] Init(1) by (auto, metis less_not_refl3)
  have "∀ p . finite {<p, inM True>, <p, inM False>}" by auto
  moreover have SubsetMsg:
    "∀ p. { msg . (msg ∈# msgs init)
      ∧ (∃ v::bool .  <p, inM v> = msg)}
      ⊆ {<p, inM True>, <p, inM False>}" by auto
  ultimately have AllFinite:
    "∀ p. finite { msg . (msg ∈# msgs init)
      ∧ (∃ v::bool .  <p, inM v> = msg)}"
    using finite_subset by (clarify, auto)
  have " { msg . (msg ∈# msgs init)
    ∧ (∃ p∈Proc . ∃ v::bool.  <p, inM v> = msg)}
      = (⋃ p ∈ Proc . { msg . (msg ∈# msgs init)
    ∧ (∃ v::bool .  <p, inM v> = msg)})" by auto
  hence "finite  { msg . (msg ∈# msgs init)
    ∧ (∃ p∈Proc . ∃ v::bool.  <p, inM v> = msg)}"
    using AllFinite FiniteProcs by auto
  hence InitFinite:"finite {msg . msg ∈# msgs init}"
    by (auto simp add: A B InitMsgs)
  show ?thesis using Init(2) InitFinite
  proof(induct rule: reachable.induct, simp_all)
    fix cfg1 cfg2 msg cfg3
    assume assmsInduct:"reachable cfg1 cfg2"
      "finite {msg. msg ∈# msgs cfg2}" "cfg2 ⊢ msg ↦ cfg3"
      "finite {msg. msg ∈# msgs cfg1}" "reachable init cfg"
      "finite {msg. msg ∈# msgs init}"
    from assmsInduct(3) obtain p where "isReceiverOf p msg "
      by (metis StepOutMsg isReceiverOf.simps(1) isReceiverOf.simps(2)
       message.exhaust)
    hence "msgs cfg3 = ((msgs cfg2 -# msg) ∪# (sends p (states cfg2 p)
      (unpackMessage msg) ))"
      using assmsInduct(3) by (cases msg, auto simp add: add.commute)
    hence MsgSet: "{msg. msg ∈#  msgs cfg3 }
      = {m. m ∈# ((msgs cfg2 -# msg) ∪# (sends p (states cfg2 p)
        (unpackMessage msg) )) } " by simp
    have "{v. v ∈# (msgs cfg2 -# msg)} ⊆ {msg. msg ∈# msgs cfg2}"
```

```
      by auto
    from finite_subset[OF this]
      have "finite {v. (v ∈# sends p (states cfg2 p) (unpackMessage msg))
        ∨ (v ∈# (msgs cfg2 -# msg))}"
      using FiniteSends assmsInduct(2) by auto
    thus "finite {msg. msg ∈# msgs cfg3}"
       unfolding MsgSet by auto
  qed
qed

end

end
```

3 ListUtilities

ListUtilities defines a (proper) prefix relation for lists, and proves some additional lemmata, mostly about lists.

```
theory ListUtilities
imports Main
begin
```

3.1 List Prefixes

```
inductive prefixList ::
  "'a list ⇒ 'a list ⇒ bool"
where
  "prefixList [] (x # xs)"
| "prefixList xa xb ⟹ prefixList (x # xa) (x # xb)"

lemma PrefixListHasTail:
fixes
  l1 :: "'a list" and
  l2 :: "'a list"
assumes
  "prefixList l1 l2"
shows
  "∃ l . l2 = l1 @ l ∧ l ≠ []"
using assms by (induct rule: prefixList.induct, auto)

lemma PrefixListMonotonicity:
fixes
  l1 :: "'a list" and
  l2 :: "'a list"
assumes
  "prefixList l1 l2"
shows
  "length l1 < length l2"
using assms by (induct rule: prefixList.induct, auto)

lemma TailIsPrefixList :
```

```
fixes
  l1 :: "'a list" and
  tail :: "'a list"
assumes "tail ≠ []"
shows "prefixList l1 (l1 @ tail)"
using assms
proof (induct l1, auto)
  have "∃ x xs . tail = x # xs"
    using assms by (metis neq_Nil_conv)
  thus "prefixList [] tail"
    using assms  by (metis prefixList.intros(1))
next
  fix a l1
  assume "prefixList l1 (l1 @ tail)"
  thus "prefixList (a # l1) (a # l1 @ tail)"
    by (metis prefixList.intros(2))
qed

lemma PrefixListTransitive:
fixes
  l1 :: "'a list" and
  l2 :: "'a list" and
  l3 :: "'a list"
assumes
  "prefixList l1 l2"
  "prefixList l2 l3"
shows
  "prefixList l1 l3"
using assms
proof -
  from assms(1) have "∃ l12 . l2 = l1 @ l12 ∧ l12 ≠ []"
    using PrefixListHasTail by auto
  then obtain l12 where Extend1: "l2 = l1 @ l12 ∧ l12 ≠ []" by blast
  from assms(2) have Extend2: "∃ l23 . l3 = l2 @ l23 ∧ l23 ≠ []"
    using PrefixListHasTail by auto
  then obtain l23 where Extend2: "l3 = l2 @ l23 ∧ l23 ≠ []" by blast
  have "l3 = l1 @ (l12 @ l23) ∧ (l12 @ l23) ≠ []"
    using Extend1 Extend2 by simp
  hence "∃ l . l3 = l1 @ l ∧ l ≠ []" by blast
  thus "prefixList l1 l3" using TailIsPrefixList by auto
qed
```

3.2 Lemmas for lists and nat predicates

```
lemma NatPredicateTippingPoint:
fixes
  n2 Pr
assumes
  Min:      "0 < n2" and
  Pr0:      "Pr 0" and
  NotPrN2: "¬Pr n2"
shows
```

```
  "∃n<n2. Pr n ∧ ¬Pr (Suc n)"
proof (rule classical, simp)
  assume Asm: "∀n. Pr n ⟶ n < n2 ⟶ Pr (Suc n)"
  have "⋀n. n < n2 ⟹ Pr n"
  proof-
    fix n
    show "n < n2 ⟹ Pr n"
    by (induct n, auto simp add: Pr0 Asm)
  qed
  hence False
    using Asm[rule_format, of "n2 - 1"] Min NotPrN2 by auto
  thus ?thesis by auto
qed

lemma MinPredicate:
fixes
  P::"nat ⇒ bool"
assumes
  "∃ n . P n"
shows
  "(∃ n0 . (P n0) ∧ (∀ n' . (P n') ⟶ (n' ≥ n0)))"
using assms
by (metis LeastI2_wellorder Suc_n_not_le_n)
```

The lemma MinPredicate2 describes one case of MinPredicate where the aforementioned smallest element is zero.

```
lemma MinPredicate2:
fixes
  P::"nat ⇒ bool"
assumes
 "∃ n . P n"
shows
  "∃ n0 . (P n0) ∧ (n0 = 0 ∨ ¬ P (n0 - 1))"
using assms MinPredicate
by (metis add_diff_cancel_right' diff_is_0_eq diff_mult_distrib mult_eq_if)
```

PredicatePairFunction allows to obtain functions mapping two arguments to pairs from 4-ary predicates which are left-total on their first two arguments.

```
lemma PredicatePairFunction:
fixes
  P::"'a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ bool"
assumes
  A1: "∀x1 x2 . ∃y1 y2 . (P x1 x2 y1 y2)"
shows
  "∃f . ∀x1 x2 . ∃y1 y2 .
    (f x1 x2) = (y1, y2)
    ∧ (P x1 x2 (fst (f x1 x2)) (snd (f x1 x2)))"
proof -
  define P' where "P' x y = P (fst x) (snd x) (fst y) (snd y)" for x y
  hence "∀x. ∃y. P' x  y" using A1 by auto
  hence "∃f. ∀x. P' x (f x)" by metis
  then obtain f where "∀x. P' x (f x)" by blast
```

```
  moreover define f' where "f' x1 x2 = f (x1, x2)" for x1 x2
  ultimately have "∀x. P' x (f' (fst x) (snd x))" by auto
  hence "∃f'. ∀x. P' x (f' (fst x) (snd x))" by blast
  thus ?thesis using P'_def by auto
qed

lemma PredicatePairFunctions2:
fixes
  P::"'a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ bool"
assumes
  A1: "∀x1 x2 . ∃y1 y2 . (P x1 x2 y1 y2)"
obtains f1 f2  where
  "∀x1 x2 . ∃y1 y2 .
    (f1 x1 x2) = y1 ∧ (f2 x1 x2) = y2
    ∧ (P x1 x2 (f1 x1 x2) (f2 x1 x2))"
proof (cases thesis, auto)
  assume ass: "⋀f1 f2. ∀x1 x2. P x1 x2 (f1 x1 x2) (f2 x1 x2) ⟹ False"
  obtain f where F: "∀x1 x2. ∃y1 y2. f x1 x2 = (y1, y2) ∧ P x1 x2 (fst (f x1 x2))
(snd (f x1 x2))"
    using PredicatePairFunction[OF A1] by blast
  define f1 where "f1 x1 x2 = fst (f x1 x2)" for x1 x2
  define f2 where "f2 x1 x2 = snd (f x1 x2)" for x1 x2
  show False
    using ass[of f1 f2] F unfolding f1_def f2_def by auto
qed

lemma PredicatePairFunctions2Inv:
fixes
  P::"'a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ bool"
assumes
  A1: "∀x1 x2 . ∃y1 y2 . (P x1 x2 y1 y2)"
obtains f1 f2  where
  "∀x1 x2 . (P x1 x2 (f1 x1 x2) (f2 x1 x2))"
using PredicatePairFunctions2[OF A1] by auto

lemma SmallerMultipleStepsWithLimit:
fixes
  k A limit
assumes
  "∀ n ≥ limit . (A (Suc n)) < (A n)"
shows
  "∀ n ≥ limit . (A (n + k)) ≤ (A n) - k"
proof(induct k,auto)
  fix n k
  assume IH: "∀n≥limit. A (n + k) ≤ A n - k" "limit ≤ n"
  hence "A (Suc (n + k)) < A (n + k)" using assms by simp
  hence "A (Suc (n + k)) < A n - k" using IH by auto
  thus "A (Suc (n + k)) ≤ A n - Suc k"
    by (metis Suc_lessI add_Suc_right add_diff_cancel_left'
        less_diff_conv less_or_eq_imp_le add.commute)
qed
```

```
lemma PrefixSameOnLow:
fixes
  l1 l2
assumes
  "prefixList l1 l2"
shows
  "∀ index < length l1 . l1 ! index = l2 ! index"
using assms
proof(induct rule: prefixList.induct, auto)
  fix xa xb ::"'a list" and x index
  assume AssumpProof: "prefixList xa xb"
        "∀index < length xa. xa ! index = xb ! index"
        "prefixList l1 l2" "index < Suc (length xa)"
  show "(x # xa) ! index = (x # xb) ! index" using AssumpProof
  proof(cases "index = 0", auto)
  qed
qed

lemma KeepProperty:
fixes
  P Q low
assumes
  "∀ i ≥ low . P i ⟶ (P (Suc i) ∧ Q i)" "P low"
shows
  "∀ i ≥ low . Q i"
using assms
proof(clarify)
  fix i
  assume Assump:
    "∀i≥low. P i ⟶ P (Suc i) ∧ Q i"
    "P low"
    "low ≤ i"
  hence "∀i≥low. P i ⟶ P (Suc i)" by blast
  hence "∀ i ≥ low . P i" using Assump(2) by (metis dec_induct)
  hence "P i" using Assump(3) by blast
  thus "Q i" using Assump by blast
qed

lemma ListLenDrop:
fixes
  i la lb
assumes
  "i < length lb"
  "i ≥ la"
shows
  "lb ! i ∈ set (drop la lb)"
using assms
by (metis Cons_nth_drop_Suc in_set_member member_rec(1)
      set_drop_subset_set_drop rev_subsetD)

lemma DropToShift:
fixes
```

```
  l i list
assumes
  "l + i < length list"
shows
  "(drop l list) ! i = list ! (l + i)"
using assms
by (induct l, auto)

lemma SetToIndex:
fixes
  a and liste::"'a list"
assumes
  AssumpSetToIndex: "a ∈ set liste"
shows
  "∃ index < length liste . a = liste ! index"
proof -
  have LenInduct:
    "⋀xs. ∀ys. length ys < length xs ⟶ a ∈ set ys
          ⟶ (∃index<length ys. a = ys ! index)
          ⟹ a ∈ set xs ⟶ (∃index<length xs. a = xs ! index)"
  proof(auto)
    fix xs
    assume AssumpLengthInduction:
      "∀ys. length ys < length xs ⟶ a ∈ set ys
      ⟶ (∃index<length ys. a = ys ! index)" "a ∈ set xs"
    have "∃ x xs' . xs = x#xs'" using AssumpLengthInduction(2)
      by (metis ListMem.cases ListMem_iff)
    then obtain x xs' where XSSplit: "xs = x#xs'" by blast
    hence "a ∈ insert x (set xs')" using set_simps AssumpLengthInduction
      by simp
    hence "a = x ∨ a ∈ set xs'" by simp
    thus "∃index<length xs. a = xs ! index"
    proof(cases "a = x",auto)
      show "∃index<length xs. x = xs ! index" using XSSplit by auto
    next
      assume AssumpCases: "a ∈ set xs'" "a ≠ x"
      have "length xs' < length xs" using XSSplit by simp
      hence "∃index<length xs'. a = xs' ! index"
        using AssumpLengthInduction(1) AssumpCases(1) by simp
      thus "∃index<length xs. a = xs ! index" using XSSplit by auto
    qed
  qed
  thus "∃ index < length liste . a = liste ! index"
    using length_induct[of
      "λl. a ∈ set l ⟶ (∃ index < length l . a = l ! index)" "liste"]
    AssumpSetToIndex by blast
qed

lemma DropToIndex:
fixes
  a::"'a" and l liste
assumes
```

```
  AssumpDropToIndex: "a ∈ set (drop l liste)"
shows
  "∃ i ≥ l . i < length liste ∧ a = liste ! i"
proof-
  have "∃ index < length (drop l liste) . a = (drop l liste) ! index"
    using AssumpDropToIndex SetToIndex[of "a" "drop l liste"] by blast
  then obtain index where Index: "index < length (drop l liste)"
    "a = (drop l liste) ! index" by blast
  have "l + index < length liste" using Index(1)
    by (metis length_drop less_diff_conv add.commute)
  hence "a = liste ! (l + index)"
    using DropToShift[of "l" "index"] Index(2) by blast
  thus "∃i≥l. i < length liste ∧ a = liste ! i"
    by (metis ‹l + index < length liste› le_add1)
qed

end
```

4 Execution

`Execution` introduces a locale for executions within asynchronous systems.

```
theory Execution
imports AsynchronousSystem ListUtilities
begin
```

4.1 Execution locale definition

A (finite) execution within a system is a list of configurations `exec` accompanied by a list of messages `trace` such that the first configuration is initial and every next state can be reached processing the messages in `trace`.

```
locale execution =
  asynchronousSystem trans sends start
for
  trans :: "'p ⇒ 's ⇒ 'v messageValue ⇒ 's" and
  sends :: "'p ⇒ 's ⇒ 'v messageValue ⇒ ('p, 'v) message multiset" and
  start :: "'p ⇒ 's"
+
fixes
  exec :: "('p, 'v, 's ) configuration list" and
  trace :: "('p, 'v) message list"
assumes
  notEmpty: "length exec ≥ 1" and
  length: "length exec - 1 = length trace" and
  base: "initial (hd exec)" and
  step: "⟦ i < length exec - 1 ; cfg1 = exec ! i ; cfg2 = exec ! (i + 1) ⟧
      ⟹ ((cfg1 ⊢ trace ! i ↦ cfg2)) "
begin

abbreviation execMsg ::
  "nat ⇒ ('p,'v) message"
where
```

```
  "execMsg n ≡ (trace ! n)"

abbreviation execConf ::
  "nat ⇒ ('p, 'v, 's) configuration"
where
  "execConf n  ≡ (exec ! n)"
```

4.2 Enabledness and occurrence in the execution

```
definition minimalEnabled ::
  "('p, 'v) message ⇒ bool"
where
  "minimalEnabled msg ≡ (∃ p . isReceiverOf p msg)
    ∧ (enabled (last exec) msg)
    ∧ (∃ n . n < length exec ∧ enabled (execConf n) msg
      ∧ (∀ n' ≥ n . n' < length trace ⟶ msg ≠ (execMsg n'))
    ∧ (∀ n' msg' . ((∃ p . isReceiverOf p msg')
      ∧ (enabled (last exec) msg')
      ∧ n' < length trace
      ∧ enabled (execConf n') msg'
      ∧ (∀ n'' ≥ n' . n'' < length trace ⟶ msg' ≠
                    (execMsg n''))) ⟶ n' ≥ n))"

definition firstOccurrence ::
  "('p, 'v) message ⇒ nat ⇒ bool"
where
  "firstOccurrence msg n ≡ (∃ p . isReceiverOf p msg)
    ∧ (enabled (last exec) msg) ∧ n < (length exec)
    ∧ enabled (execConf n) msg
    ∧ (∀ n' ≥ n . n' < length trace ⟶ msg ≠ (execMsg n'))
    ∧ ( n ≠ 0 ⟶ (¬ enabled (execConf (n - 1)) msg
      ∨ msg = execMsg (n - 1)))"

lemma FirstOccurrenceExists:
assumes
  "enabled (last exec) msg"
  "∃p. isReceiverOf p msg"
shows
  "∃ n . firstOccurrence msg n"
proof-
  have "length exec - 1 < length exec
    ∧ (∀ n' ≥ (length exec - 1) . n' < length trace ⟶ trace ! n' ≠ msg)"
    using length
    by (metis diff_diff_cancel leD notEmpty zero_less_diff
      zero_less_one)
  hence NNotInTrace: "∃ n < length exec .
    (∀ n'≥n . n' < length trace ⟶ trace ! n' ≠ msg)" by blast
  hence "∃ n0 < length exec .
    (∀ n'≥n0 . n' < length trace ⟶ trace ! n' ≠ msg) ∧
    ((n0 = 0)
      ∨ ¬ (∀ n' ≥ (n0 - 1) . n' < length trace ⟶ trace ! n' ≠ msg))"
    using MinPredicate2[of "λx.(x < length exec
```

```
    ∧ (∀n'≥x.(n'<length trace ⟶ trace ! n' ≠ msg)))"]
  by auto
hence "∃ n0. n0 < length exec
  ∧ (∀ n'≥n0 . n' < length trace ⟶ trace ! n' ≠ msg)
  ∧ ((n0 = 0)
    ∨ ¬ (∀ n' ≥ (n0 - 1) . n' < length trace ⟶ trace ! n' ≠ msg))"
  by simp
from this obtain n0 where N0a: "n0 < length exec
  ∧ (∀ n'≥n0 . n' < length trace ⟶ trace ! n' ≠ msg)
  ∧ ((n0 = 0)
    ∨ ¬ (∀ n' ≥ (n0 - 1) . n' < length trace ⟶ trace ! n' ≠ msg))"
  by metis
hence N0: "n0 < length exec"
  "(∀ n'≥n0 . n' < length trace ⟶ trace ! n' ≠ msg)"
  "((n0 = 0)
    ∨ ¬ (∀ n' ≥ (n0 - 1) . n' < length trace ⟶ trace ! n' ≠ msg))"
  using N0a by auto
have N0': "n0 = 0 ∨ trace ! (n0 - 1) = msg"
proof(cases "n0 = 0", auto)
  assume N0NotZero: "n0 > 0"
  hence "¬ (∀ n' ≥ (n0 - 1) . n' < length trace ⟶ trace ! n' ≠ msg)"
    using N0(3) by blast
  moreover have "n0 - 1 < length trace"
    using N0(1) length N0NotZero
    by (metis calculation le_less_trans)
  ultimately show "execMsg (n0 - Suc 0) = msg" using N0(2)
    by (metis One_nat_def Suc_diff_Suc diff_Suc_eq_diff_pred
       diff_diff_cancel diff_is_0_eq leI nat_le_linear)
qed
have "∃ n1 < length exec .
  (∀ n'≥n1 . n' < length trace ⟶ trace ! n' ≠ msg)
  ∧ enabled (exec ! n1) msg
  ∧ (n1 = 0 ∨ ¬ enabled (exec ! (n1 - 1)) msg ∨ trace ! (n1 - 1) = msg)"
proof(cases "enabled (exec ! n0) msg")
  assume "enabled (execConf n0) msg"
  hence "n0 < length exec"
    "(∀ n'≥n0 . n' < length trace ⟶ trace ! n' ≠ msg)"
    "enabled (exec ! n0) msg ∧
    (n0 = 0 ∨ ¬ enabled (exec ! (n0 - 1)) msg ∨ trace ! (n0 - 1) = msg)"
  using N0 N0' by auto
  thus "∃n1<length exec.
     (∀n'≥n1. n' < length trace ⟶ execMsg n' ≠ msg)
     ∧ enabled (execConf n1) msg
     ∧ (n1 = 0 ∨ ¬ enabled (execConf (n1 - 1)) msg
       ∨ execMsg (n1 - 1) = msg)"
    by metis
next
  assume NotEnabled: "¬ enabled (execConf n0) msg"
  have "last exec = exec ! (length exec - 1)" using last_conv_nth notEmpty
    by (metis NNotInTrace length_0_conv less_nat_zero_code)
  hence EnabledInLast: "enabled (exec ! (length exec - 1)) msg"
    using assms(1) by simp
```

```
    hence "n0 ≠ length exec - 1" using NotEnabled by auto
    hence N0Small: "n0 < length exec - 1" using N0(1) by simp
    hence "∃ k < length exec - 1 - n0  . ¬ enabled (execConf (n0 + k)) msg
      ∧ enabled (execConf (n0 + k + 1)) msg"
      using NatPredicateTippingPoint[of "length exec - 1 - n0"
        "λx.¬(enabled (exec ! (n0 + x)) msg)"]
        assms(1) NotEnabled EnabledInLast by simp
    then obtain k where K: " k < length exec - 1 - n0"
      "¬ enabled (execConf (n0 + k)) msg"
      "enabled (execConf (n0 + k + 1)) msg" by blast
    define n1 where "n1 = k + n0 + 1"
    hence N1: "n1 ≥ n0" "¬ enabled (execConf (n1 - 1)) msg"
      "enabled (execConf n1) msg" "n1 < length exec"
      unfolding n1_def using K
      by (auto simp add: add.commute)
    have "∀n'≥n1. n' < length trace ⟶ execMsg n' ≠ msg"
      using N1(1) N0(2) by (metis order_trans)
    thus "∃n1<length exec.
        (∀n'≥n1. n' < length trace ⟶ execMsg n' ≠ msg)
        ∧ enabled (execConf n1) msg
        ∧ (n1 = 0 ∨ ¬ enabled (execConf (n1 - 1)) msg
          ∨ execMsg (n1 - 1) = msg)"
    using N1 by auto
  qed
  then obtain n1 where N1: "n1 < length exec"
    "∀ n'≥n1 . n' < length trace ⟶ trace ! n' ≠ msg"
    "enabled (exec ! n1) msg"
    "n1 = 0 ∨ ¬ enabled (exec ! (n1 - 1)) msg ∨ trace ! (n1 - 1) = msg"
    by metis
  hence "firstOccurrence msg n1" using assms unfolding firstOccurrence_def
    by auto
  thus "∃n. firstOccurrence msg n" by blast
qed

lemma ReachableInExecution:
assumes
  "i < length exec"
  "j ≤ i"
shows
  "reachable (execConf j) (execConf i)"
using assms proof(induct i, auto)
  show "reachable (execConf 0) (execConf 0)"
    using reachable.simps by blast
next
  fix ia
  assume
    IH: "(j ≤ ia ⟹ reachable (execConf j) (execConf ia))"
    "Suc ia < length exec"
    "j ≤ Suc ia"
    "i < length exec"
    "j ≤ i"
  show "reachable (execConf j) (execConf (Suc ia))"
```

```
  proof(cases "j = Suc ia", auto)
    show "reachable (execConf (Suc ia)) (execConf (Suc ia))"
      using reachable.simps by metis
  next
    assume "j ≠ Suc ia"
    hence "j ≤ ia" using IH(3) by simp
    hence "reachable (execConf j) (execConf ia)" using IH(1) by simp
    moreover have "reachable (execConf ia) (execConf (Suc ia))"
      using reachable.simps
      by (metis IH(2) Suc_eq_plus1 less_diff_conv local.step)
    ultimately show "reachable (execConf j) (execConf (Suc ia))"
      using ReachableTrans by blast
  qed
qed

lemma LastPoint:
fixes
  msg::"('p, 'v) message"
assumes
  "enabled (last exec) msg"
obtains n where
  "n < length exec"
  "enabled (execConf n) msg"
  "∀ n' ≥ n .
    n' < length trace ⟶ msg ≠ (execMsg n')"
  "∀ n0 .
      n0 < length exec
    ∧ enabled (execConf n0) msg
    ∧ (∀ n' ≥ n0 .
        n' < length trace ⟶ msg ≠ (execMsg n'))
    ⟶ n0 ≥ n"
proof (cases ?thesis, simp)
  case False
  define len where "len = length exec - 1"
  have
    "len < length exec"
    "enabled (execConf len) msg"
    "∀ n' ≥ len . n' < length trace ⟶ msg ≠ (execMsg n')"
    using assms notEmpty length unfolding len_def
    by (auto, metis One_nat_def last_conv_nth list.size(3) not_one_le_zero)
  hence "∃ n . n < length exec ∧ enabled (execConf n) msg
    ∧ (∀ n' ≥ n . n' < length trace ⟶ msg ≠ (execMsg n'))"
    by blast
  from MinPredicate[OF this]
    show ?thesis using that False by blast
qed

lemma ExistImpliesMinEnabled:
fixes
  msg :: "('p, 'v) message" and
  p :: 'p
assumes
```

```
    "isReceiverOf p msg"
    "enabled (last exec) msg"
  shows
    "∃ msg' . minimalEnabled msg'"
  proof-
    have MsgHasMinTime:"∀ msg . (enabled (last exec) msg
      ∧ (∃ p . isReceiverOf p msg))
      ⟶ (∃ n . n < length exec ∧ enabled (execConf n) msg
          ∧ (∀ n' ≥ n . n' < length trace ⟶ msg ≠ (execMsg n'))
          ∧ (∀ n0 . n0 < length exec ∧ enabled (execConf n0) msg
          ∧ (∀ n' ≥ n0 . n' < length trace ⟶ msg ≠ (execMsg n'))
          ⟶ n0 ≥ n))" by (clarify, rule LastPoint, auto)
    let ?enabledTimes = "{n::nat . ∃ msg . (enabled (last exec) msg
      ∧ (∃ p . isReceiverOf p msg))
      ∧ n < length exec ∧ (enabled (execConf n) msg
      ∧ (∀ n' ≥ n . n' < length trace ⟶ msg ≠ (execMsg n')))}"
    have NotEmpty:"?enabledTimes ≠ {}" using assms MsgHasMinTime by blast
    hence "∃ n0 . n0 ∈ ?enabledTimes" by blast
    hence "∃ nMin ∈ ?enabledTimes . ∀ x ∈ ?enabledTimes . x ≥ nMin"
      using MinPredicate[of "λn.(n ∈ ?enabledTimes)"] by simp
    then obtain nMin where NMin: "nMin ∈ ?enabledTimes"
      "∀ x ∈ ?enabledTimes . x ≥ nMin" by blast
    hence "∃ msg . (enabled (last exec) msg ∧ (∃ p . isReceiverOf p msg))
      ∧ nMin < length exec ∧ (enabled (execConf nMin) msg
      ∧ (∀ n' ≥ nMin . n' < length trace ⟶ msg ≠ (execMsg n'))
      ∧ (∀ n0 . n0 < length exec ∧ enabled (execConf n0) msg
        ∧ (∀ n' ≥ n0 . n' < length trace ⟶ msg ≠ (execMsg n'))
        ⟶ n0 ≥ nMin))" by blast
    then obtain msg where "(enabled (last exec) msg
      ∧ (∃ p . isReceiverOf p msg))
      ∧ nMin < length exec ∧(enabled (execConf nMin) msg
      ∧ (∀ n' ≥ nMin . n' < length trace ⟶ msg ≠ (execMsg n'))
      ∧ (∀ n0 . n0 < length exec ∧ enabled (execConf n0) msg
        ∧ (∀ n' ≥ n0 . n' < length trace ⟶ msg ≠ (execMsg n'))
        ⟶ n0 ≥ nMin))" by blast
    moreover have "(∀ n' msg' . ((∃ p . isReceiverOf p msg')
      ∧ (enabled (last exec) msg')
      ∧ n' < length trace ∧ enabled (execConf n') msg'
      ∧ (∀ n'' ≥ n' . n'' < length trace ⟶ msg' ≠ (execMsg n'')))
        ⟶ n' ≥ nMin)"
    proof(clarify)
      fix n' msg' p
      assume Assms:
        "isReceiverOf p msg'"
        "enabled (last exec) msg'"
        "n' < length trace"
        "enabled (execConf n') msg'"
        "∀n'' ≥ n'. (n'' < length trace) ⟶ (msg' ≠ execMsg n'')"
      from Assms(3) have "n' < length exec" using length by simp
      with Assms have "n' ∈ ?enabledTimes" by auto
      thus "nMin ≤ n'" using NMin(2) by simp
    qed
```

```
  ultimately have "minimalEnabled msg"
    using minimalEnabled_def by blast
  thus ?thesis by blast
qed

lemma StaysEnabledStep:
assumes
  En: "enabled cfg msg" and
  Cfg: "cfg = exec ! n" and
  N: "n < length exec"
shows
  "enabled (exec ! (n + 1)) msg
  ∨ n = (length exec - 1)
  ∨ msg = trace ! n"
proof(cases "n = length exec - 1")
  case True
  thus ?thesis by simp
next
  case False
  with N have N: "n < length exec - 1" by simp
  with Cfg have Step:  "cfg ⊢ trace ! n ↦ (exec ! (n + 1))"
    using step by simp
  thus ?thesis proof(cases "enabled (exec ! (n + 1)) msg")
    case True
    thus ?thesis by simp
  next
    case False
    hence "¬ enabled (exec ! (n + 1)) msg" by simp
    thus ?thesis using En enabled_def Step N OnlyOccurenceDisables by blast
  qed
qed

lemma StaysEnabledHelp:
assumes
  "enabled cfg msg" and
  "cfg = exec ! n" and
  "n < length exec"
shows
  "∀ i . i ≥ n ∧ i < (length exec - 1) ∧ enabled (exec ! i) msg
  ⟶ msg = (trace ! i) ∨ (enabled (exec ! (i+1)) msg)"
proof(clarify)
  fix i
  assume "n ≤ i" "i < length exec - 1"
    "enabled (execConf i) msg" "¬ enabled (execConf (i + 1)) msg"
  thus "msg = (trace ! i)"
    using assms StaysEnabledStep
    by (metis add.right_neutral add_strict_mono le_add_diff_inverse2
        nat_neq_iff notEmpty  zero_less_one)
qed

lemma StaysEnabled:
assumes En: "enabled cfg msg" and
```

```
  "cfg = exec ! n" and
  "n < length exec"
shows "enabled (last exec) msg ∨ (∃ i . i ≥ n ∧ i < (length exec - 1)
  ∧ msg = trace ! i )"
proof(cases "enabled (last exec) msg")
  case True
  thus ?thesis by simp
next
  case False
  hence NotEnabled: "¬ enabled (last exec) msg" by simp
  have "last exec = exec ! (length exec - 1)"
    by (metis last_conv_nth list.size(3) notEmpty not_one_le_zero)
  hence "∃ l . l ≥ n ∧ last exec = exec ! l ∧ l = length exec - 1"
    using assms(3) by auto
  then obtain l where L: "l ≥ n" "last exec = exec ! l"
    "l = length exec - 1" by blast
  have "(∃ i . i ≥ n ∧ i < (length exec - 1) ∧ msg = trace ! i )"
  proof (rule ccontr)
    assume Ass: " ¬ (∃i≥n. i < length exec - 1 ∧ msg = execMsg i)"
    hence Not: "∀ i. i < n ∨ i ≥ length exec - 1 ∨ msg ≠ execMsg i"
      by (metis leI)
    have "∀ i. i ≥ n ∧ i ≤ length exec - 1 ⟶ enabled (exec ! i) msg"
    proof(clarify)
      fix i
      assume I: "n ≤ i" "i ≤ length exec - 1"
      thus "enabled (execConf i) msg"
        using StaysEnabledHelp[OF assms] assms(1,2) Ass
        by (induct i, auto, metis Suc_le_lessD le_Suc_eq)
    qed
    with NotEnabled L show False by simp
  qed
  thus ?thesis by simp
qed

end — end of locale Execution
```

4.3 Expanding executions to longer executions

```
lemma (in asynchronousSystem) expandExecutionStep:
fixes
  cfg :: "('p, 'v, 's ) configuration"
assumes
  CfgIsReachable: "(last exec') ⊢ msg ↦ cMsg" and
  ExecIsExecution: "execution trans sends start exec' trace'"
shows
  "∃ exec'' trace''. (execution trans sends start exec'' trace'')
  ∧ (prefixList exec' exec'')
  ∧ (prefixList trace' trace'')
  ∧ (last exec'') = cMsg
  ∧ (last trace'' = msg)"
proof -
  define execMsg where "execMsg = exec' @ [cMsg]"
```

```
define traceMsg where "traceMsg = trace' @ [msg]"
have ExecMsgEq: "∀ i < ((length execMsg) - 1) . execMsg ! i = exec'!i "
  using execMsg_def by (auto simp add: nth_append)
have TraceMsgEq: "∀ i < ((length traceMsg) - 1) . traceMsg!i = trace'!i"
  using traceMsg_def
  by (auto simp add: nth_append)
have ExecLen: "(length execMsg) ≥ 1" using execMsg_def by auto
have lessLessExec: "∀ i . i < (length exec') ⟶ i < (length execMsg )"
  unfolding execMsg_def by auto
have ExecLenTraceLen: "length exec'- 1 = length trace'"
  using ExecIsExecution execution.length by auto
have lessLessTrace: "∀ i . i < (length exec' - 1) ⟶ i < (length trace')"
  using ExecLenTraceLen by auto
have Exec'Len: "length exec' ≥ 1"
  using ExecIsExecution execution.notEmpty by blast
hence "hd exec' = hd execMsg " using execMsg_def
  by (metis One_nat_def hd_append2 length_0_conv not_one_le_zero)
moreover have "initial (hd exec')"
  using ExecIsExecution execution.base by blast
ultimately have ExecInit: "initial (hd execMsg)" using execMsg_def by auto
have ExecMsgLen: "length execMsg - 1 = length traceMsg"
  using ExecLenTraceLen unfolding execMsg_def traceMsg_def
  by (auto,metis Exec'Len Suc_pred length_greater_0_conv list.size(3)
     not_one_le_zero)
have ExecSteps:"∀ i < length exec' - 1 .
  ((exec' ! i)  ⊢ trace' ! i ↦ (exec' ! (i + 1)))"
  using ExecIsExecution execution.step by blast
have "∀ i < length execMsg - 1.
  ((execMsg ! i) ⊢ traceMsg ! i ↦ (execMsg ! (i + 1)))"
  unfolding execMsg_def traceMsg_def
proof auto
  fix i
  assume IlessLen:"i < (length exec')"
  show "((exec' @ [cMsg]) ! i) ⊢ ((trace' @ [msg]) ! i)
    ↦ ((exec' @ [cMsg]) ! (Suc i))"
  proof (cases "(i < (length exec') - 1)")
  case True
    hence IlessLen1: "(i < (length exec') - 1)" by auto
    hence "((exec' ! i)  ⊢ trace' ! i ↦ (exec' ! (i + 1)))"
      using ExecSteps by auto
    with IlessLen1 ExecMsgEq lessLessExec execMsg_def
    have "((exec' @ [cMsg]) ! i) ⊢ ((trace') ! i)
      ↦ ((exec' @ [cMsg]) ! (Suc i))" by auto
    thus "((exec' @ [cMsg]) ! i) ⊢ ((trace' @ [msg]) ! i)
      ↦ ((exec' @ [cMsg]) ! (Suc i))"
      using IlessLen1 TraceMsgEq lessLessTrace traceMsg_def by auto
  next
  case False
    with IlessLen have IEqLen1: "(i = (length exec') - 1)" by auto
    thus "((exec' @ [cMsg]) ! i) ⊢ ((trace' @ [msg]) ! i)
      ↦ ((exec' @ [cMsg]) ! (Suc i))"
      using  execMsg_def traceMsg_def  CfgIsReachable Exec'Len
```

```
              ExecLenTraceLen ExecMsgEq ExecMsgLen IlessLen
        by (metis One_nat_def Suc_eq_plus1 Suc_eq_plus1_left last_conv_nth
           le_add_diff_inverse length_append less_nat_zero_code list.size(3)
           list.size(4) nth_append_length)
    qed
  qed
  hence isExecution: "execution trans sends start execMsg traceMsg"
    using ExecLen ExecMsgLen ExecInit
    by (unfold_locales ,auto)
  moreover have "prefixList exec' execMsg" unfolding execMsg_def
    by (metis TailIsPrefixList not_Cons_self2)
  moreover have "prefixList trace' traceMsg" unfolding traceMsg_def
    by (metis TailIsPrefixList not_Cons_self2)
  ultimately show ?thesis using execMsg_def traceMsg_def by (metis last_snoc)
qed

lemma (in asynchronousSystem) expandExecutionReachable:
fixes
  cfg :: "('p, 'v, 's ) configuration" and
  cfgLast :: "('p, 'v, 's ) configuration"
assumes
  CfgIsReachable: "reachable (cfgLast) cfg" and
  ExecIsExecution: "execution trans sends start exec trace"  and
  ExecLast: "cfgLast = last exec"
shows
  "∃ exec' trace'. (execution trans sends start exec' trace')
  ∧ ((prefixList exec exec'
    ∧ prefixList trace trace')
    ∨ (exec = exec' ∧ trace = trace'))
  ∧ (last exec') = cfg"
using CfgIsReachable  ExecIsExecution ExecLast
proof (induct cfgLast cfg rule: reachable.induct, auto)
  fix msg cfg3 exec' trace'
  assume "(last exec') ⊢ msg ↦ cfg3"
         "execution trans sends start exec' trace'"
  hence "∃ exec'' trace''. (execution trans sends start exec'' trace'')
    ∧ (prefixList exec' exec'')
    ∧ (prefixList trace' trace'') ∧ (last exec'') = cfg3
    ∧ (last trace'') = msg" by (simp add: expandExecutionStep)
  then obtain  exec'' trace'' where
    NewExec: "execution trans sends start exec'' trace''"
             "prefixList exec' exec''" "prefixList trace' trace''"
             "last exec'' = cfg3" by blast
  assume prefixLists: "prefixList exec exec'"
                      "prefixList trace trace'"
  from prefixLists(1) NewExec(2) have "prefixList exec exec''"
    using PrefixListTransitive by auto
  moreover from prefixLists(2) NewExec(3) have
    "prefixList trace trace''"  using PrefixListTransitive by auto
  ultimately show "∃ exec'' trace'' .
          execution trans sends start exec'' trace'' ∧
          ((prefixList exec exec'' ∧ prefixList trace trace'')
```

```
          ∨ (exec = exec'' ∧ trace = trace'')) ∧
          last exec'' = cfg3" using NewExec(1) NewExec(4) by blast
next
  fix msg cfg3
  assume "(last exec) ⊢ msg ↦ cfg3" "execution trans sends start exec trace"
  then show
    "∃exec' trace'. execution trans sends start exec' trace' ∧
       (prefixList exec exec' ∧ prefixList trace trace'
        ∨ exec = exec' ∧ trace = trace') ∧ last exec' = cfg3"
    using expandExecutionStep by blast
qed

lemma (in asynchronousSystem) expandExecution:
fixes
  cfg :: "('p, 'v, 's ) configuration" and
  cfgLast :: "('p, 'v, 's ) configuration"
assumes
  CfgIsReachable: "stepReachable (last exec) msg cfg" and
  ExecIsExecution: "execution trans sends start exec trace"
shows
  "∃ exec' trace'. (execution trans sends start exec' trace')
  ∧ (prefixList exec exec')
  ∧ (prefixList trace trace') ∧ (last exec') = cfg
  ∧ (msg ∈ set (drop (length trace) trace'))"
proof -
  from CfgIsReachable obtain c' c'' where
    Step: "reachable (last exec) c'" "c' ⊢ msg ↦ c''" "reachable c'' cfg"
    by (auto simp add: stepReachable_def)
  from Step(1) ExecIsExecution have "∃ exec' trace' .
    execution trans sends start exec' trace' ∧
    ((prefixList exec exec' ∧ prefixList trace trace')
    ∨ (exec = exec' ∧ trace = trace')) ∧
    last exec' = c'" by (auto simp add: expandExecutionReachable)
  then obtain  exec' trace' where Exec':
    "execution trans sends start exec' trace'"
    "(prefixList exec exec' ∧ prefixList trace trace')
      ∨ (exec = exec' ∧ trace = trace')"
    "last exec' = c'" by blast
  from Exec'(1) Exec'(3) Step(2) have "∃ exec'' trace'' .
    execution trans sends start exec'' trace'' ∧
    prefixList exec' exec'' ∧ prefixList trace' trace''
    ∧ last exec'' = c'' ∧ last trace'' = msg"
    by (auto simp add: expandExecutionStep)
  then obtain exec'' trace'' where Exec'':
    "execution trans sends start exec'' trace''"
    "prefixList exec' exec''" "prefixList trace' trace''"
    "last exec'' = c''" "last trace'' = msg" by blast
  have PrefixLists: "prefixList exec exec'' ∧ prefixList trace trace''"
  proof(cases "exec = exec' ∧ trace = trace'")
  case True
    with Exec'' show "prefixList exec exec'' ∧ prefixList trace trace''"
    by auto
```

```
  next
  case False
    with Exec'(2) have Prefix: "prefixList exec exec'"
      "prefixList trace trace'" by auto
    from Prefix(1) Exec''(2) have "prefixList exec exec''"
      using PrefixListTransitive by auto
    with Prefix(2) Exec''(3) show  "prefixList exec exec''
      ∧ prefixList trace trace''"
      using PrefixListTransitive by auto
  qed
  with Exec''(5) have MsgInTrace'': "msg ∈ set (drop (length trace) trace'')"
    by (metis PrefixListMonotonicity drop_eq_Nil last_drop
      last_in_set not_le)
  from Step(3) Exec''(1) Exec''(4) have "∃ exec''' trace''' .
    execution trans sends start exec''' trace''' ∧
    ((prefixList exec'' exec''' ∧ prefixList trace'' trace''')
    ∨ (exec'' = exec''' ∧ trace'' = trace''')) ∧
    last exec''' = cfg"
    by (auto simp add: expandExecutionReachable)
  then obtain exec''' trace''' where Exec''':
    "execution trans sends start exec''' trace'''"
    "(prefixList exec'' exec''' ∧ prefixList trace'' trace''')
    ∨ (exec'' = exec''' ∧ trace'' = trace''')"
    "last exec''' = cfg" by blast
  have "prefixList exec exec''' ∧ prefixList trace trace'''
    ∧ msg ∈ set (drop (length trace) trace''')"
  proof(cases "exec'' = exec''' ∧ trace'' = trace'''")
  case True
    with PrefixLists MsgInTrace''
    show "prefixList exec exec''' ∧ prefixList trace trace'''
    ∧ msg ∈ set (drop (length trace) trace''')" by auto
  next
  case False
    with Exec'''(2) have Prefix: "prefixList exec'' exec'''"
      "prefixList trace'' trace'''" by auto
    from Prefix(1) PrefixLists have "prefixList exec exec'''"
      using PrefixListTransitive by auto
    with Prefix(2) PrefixLists have "prefixList exec exec'''
      ∧ prefixList trace trace'''"
      using PrefixListTransitive by auto
    moreover have "msg ∈ set (drop (length trace) trace''')"
      using Prefix(2) PrefixLists MsgInTrace''
      by (metis (opaque_lifting, no_types) PrefixListHasTail append_eq_conv_conj

         drop_take rev_subsetD set_take_subset)
    ultimately show ?thesis by auto
  qed
  with Exec'''(1) Exec'''(3) show ?thesis by blast
qed
```

4.4 Infinite and fair executions

Völzer does not give much attention to the definition of the infinite executions. We derive them from finite executions by considering infinite executions to be infinite sequence of finite executions increasing monotonically w.r.t. the list prefix relation.

```
definition (in asynchronousSystem) infiniteExecution ::
  "(nat ⇒ (('p, 'v, 's) configuration list))
  ⇒ (nat ⇒ (('p, 'v) message list)) ⇒ bool"
where
  "infiniteExecution fe ft ≡
    ∀ n . execution trans sends start (fe n) (ft n) ∧
      prefixList (fe n) (fe (n+1)) ∧
      prefixList (ft n) (ft (n+1))"

definition (in asynchronousSystem) correctInfinite ::
  "(nat ⇒ (('p, 'v, 's) configuration list))
  ⇒ (nat ⇒ (('p, 'v) message list)) ⇒ 'p ⇒ bool"
where
  "correctInfinite fe ft p ≡
    infiniteExecution fe ft
    ∧ (∀ n . ∀ n0 < length (fe n). ∀ msg .(enabled ((fe n) ! n0) msg)
    ∧ isReceiverOf p msg
    ⟶ (∃ msg' . ∃ n' ≥ n . ∃ n0' ≥ n0 .isReceiverOf p msg'
    ∧ n0' < length (fe n') ∧ (msg' = ((ft n') ! n0'))))"

definition (in asynchronousSystem) fairInfiniteExecution ::
  "(nat ⇒ (('p, 'v, 's) configuration list))
  ⇒ (nat ⇒ (('p, 'v) message list)) ⇒ bool"
where
  "fairInfiniteExecution fe ft ≡
    infiniteExecution fe ft
    ∧ (∀ n . ∀ n0 < length (fe n). ∀ p . ∀ msg .
      ((enabled ((fe n) ! n0) msg)
        ∧ isReceiverOf p msg ∧ correctInfinite fe ft p )
      ⟶ (∃ n' ≥ n . ∃ n0' ≥ n0 . n0' < length (ft n')
        ∧ (msg = ((ft n') ! n0'))))"

end
```

5 FLPSystem

`FLPSystem` extends **`AsynchronousSystem`** with concepts of consensus and decisions. It develops a concept of non-uniformity regarding pending decision possibilities, where non-uniform configurations can always reach other non-uniform ones.

```
theory FLPSystem
imports AsynchronousSystem ListUtilities
begin
```

5.1 Locale for the FLP consensus setting

```
locale flpSystem =
```

```
  asynchronousSystem trans sends start
    for trans :: "'p ⇒ 's ⇒ 'v messageValue ⇒'s"
    and sends :: "'p ⇒ 's ⇒ 'v messageValue ⇒ ('p, 'v) message multiset"
    and start :: "'p ⇒ 's" +
assumes finiteProcs: "finite Proc"
    and minimalProcs: "card Proc ≥ 2"
    and finiteSends: "finite {v. v ∈# (sends p s m)}"
    and noInSends: "sends p s m <p2, inM v> = 0"
begin
```

5.2 Decidedness and uniformity of configurations

```
abbreviation vDecided ::
  "bool ⇒ ('p, 'v, 's) configuration ⇒ bool"
where
  "vDecided v cfg ≡ initReachable cfg ∧ (<⊥, outM v> ∈# msgs cfg)"

abbreviation decided ::
  "('p, 'v, 's) configuration ⇒ bool"
where
  "decided cfg ≡ (∃v . vDecided v cfg)"

definition pSilDecVal ::
  "bool ⇒ 'p ⇒ ('p, 'v, 's) configuration ⇒ bool"
where
  "pSilDecVal v p c ≡ initReachable c ∧
    (∃ c'::('p, 'v, 's) configuration . (withoutQReachable c {p} c')
    ∧ vDecided v c')"

abbreviation pSilentDecisionValues ::
  "'p ⇒ ('p, 'v, 's) configuration ⇒ bool set" (‹val[_,_]›)
where
  "val[p, c] ≡ {v. pSilDecVal v p c}"

definition vUniform ::
  "bool ⇒ ('p, 'v, 's) configuration ⇒ bool"
where
  "vUniform v c ≡ initReachable c ∧ (∀p. val[p,c] = {v})"

abbreviation nonUniform ::
  "('p, 'v, 's) configuration ⇒ bool"
where
  "nonUniform c ≡ initReachable c ∧
   ¬(vUniform False c) ∧
    ¬(vUniform True c)"
```

5.3 Agreement, validity, termination

Völzer defines consensus in terms of the classical notions of agreement, validity, and termination. The proof then mostly applies a weakened notion of termination, which we refer to as „pseudo termination".

```
definition agreement ::
```

```
  "('p, 'v, 's) configuration ⇒ bool"
where
  "agreement c ≡
    (∀v1. (<⊥, outM v1> ∈# msgs c)
      ⟶ (∀v2. (<⊥, outM v2> ∈# msgs c)
        ⟷ v2 = v1))"

definition agreementInit ::
  "('p, 'v, 's) configuration ⇒ ('p, 'v, 's) configuration ⇒ bool"
where
  "agreementInit i c ≡
    initial i ∧ reachable i c ⟶
      (∀v1. (<⊥, outM v1> ∈# msgs c)
        ⟶ (∀v2. (<⊥, outM v2> ∈# msgs c)
          ⟷ v2 = v1))"

definition validity ::
  "('p, 'v, 's) configuration ⇒ ('p, 'v, 's) configuration ⇒ bool"
where
  "validity i c ≡
    initial i ∧ reachable i c ⟶
      (∀v. (<⊥, outM v> ∈# msgs c)
        ⟶ (∃p. (<p, inM v> ∈# msgs i)))"
```

The termination concept which is implied by the concept of "pseudo-consensus" in the paper.

```
definition terminationPseudo ::
  "nat ⇒ ('p, 'v, 's) configuration ⇒ 'p set ⇒ bool"
where
  "terminationPseudo t c Q ≡ ((initReachable c ∧ card Q + t ≥ card Proc)
    ⟶ (∃c'. qReachable c Q c' ∧ decided c'))"
```

5.4 Propositions about decisions

For every process p and every configuration that is reachable from an initial configuration (i.e. `initReachable` c) we have $val(p,c) \neq \emptyset$.

This follows directly from the definition of *val* and the definition of `terminationPseudo`, which has to be assumed to ensure that there is a reachable configuration that is decided.

This corresponds to ***Proposition 2(a)*** *in Völzer's paper.*

```
lemma DecisionValuesExist:
fixes
  c :: "('p, 'v, 's) configuration" and
  p :: "'p"
assumes
  Termination: "⋀cc Q . terminationPseudo 1 cc Q" and
  Reachable: "initReachable c"
shows
  "val[p,c] ≠ {}"
proof -
  from Termination
    have "(initReachable c ∧ card Proc ≤ card (UNIV - {p}) + 1)
```

```
      ⟶ (∃c'. qReachable c (UNIV-{p}) c' ∧ initReachable c'
      ∧ (∃v. 0 < msgs c' <⊥, outM v>))"
    unfolding terminationPseudo_def by simp
  with Reachable minimalProcs finiteProcs
    have "∃c'. qReachable c (UNIV - {p}) c' ∧ initReachable c'
    ∧ (∃v. 0 < msgs c' <⊥, outM v>)"
    unfolding terminationPseudo_def initReachable_def by simp
  thus ?thesis
    unfolding pSilDecVal_def
    using Reachable by auto
qed
```

The lemma `DecidedImpliesUniform` proves that every `vDecided` configuration c is also `vUniform`. Völzer claims that this follows directly from the definitions of `vDecided` and `vUniform`. But this is not quite enough: One must also assume `terminationPseudo` and `agreement` for all reachable configurations.

This corresponds to ***Proposition 2(b)*** *in Völzer's paper.*

```
lemma DecidedImpliesUniform:
fixes
  c :: "('p, 'v, 's) configuration" and
  v :: "bool"
assumes
  AllAgree: "∀ cfg . reachable c cfg ⟶ agreement cfg" and
  Termination: "⋀cc Q . terminationPseudo 1 cc Q" and
  VDec: "vDecided v c"
shows
  "vUniform v c"
  using AllAgree VDec unfolding agreement_def vUniform_def pSilDecVal_def
proof simp
  assume
    Agree: "∀cfg. reachable c cfg ⟶
      (∀v1. 0 < msgs cfg <⊥, outM v1>
      ⟶ (∀v2. (0 < msgs cfg <⊥, outM v2>) = (v2 = v1)))" and
    vDec: "initReachable c ∧ 0 < msgs c <⊥, outM v>"
  show
    "(∀p. {v. ∃c'. qReachable c (Proc - {p}) c' ∧ initReachable c' ∧
      0 < msgs c' <⊥, outM v>} = {v})"
  proof clarify
    fix p
    have "val[p,c] ≠ {}" using Termination DecisionValuesExist vDec
      by simp
    hence NotEmpty: "{v. ∃c'. qReachable c (UNIV - {p}) c'
      ∧ initReachable c' ∧ 0 < msgs c' <⊥, outM v>} ≠ {}"
      using pSilDecVal_def by simp
    have U: "∀ u . u ∈ {v. ∃c'. qReachable c (UNIV - {p}) c'
      ∧ initReachable c' ∧ 0 < msgs c' <⊥, outM v>} ⟶ (u = v)"
    proof clarify
      fix u c'
      assume "qReachable c (UNIV - {p}) c'" "initReachable c'"
      hence Reach: "reachable c c'" using QReachImplReach by simp
      from VDec have Msg: "0 < msgs c <⊥, outM v>" by simp
      from Reach NoOutMessageLoss have
```

```
        "msgs c <⊥, outM v> ≤ msgs c' <⊥, outM v>" by simp
      with Msg have VMsg: "0 < msgs c' <⊥, outM v>"
        using NoOutMessageLoss Reach by (metis less_le_trans)
      assume "0 < msgs c' <⊥, outM u>"
      with Agree VMsg Reach show "u = v" by simp
    qed
    thus " {v. ∃c'. qReachable c (UNIV - {p}) c' ∧ initReachable c' ∧
      0 < msgs c' <⊥, outM v>} = {v}" using NotEmpty U by auto
  qed
qed

corollary NonUniformImpliesNotDecided:
fixes
  c :: "('p, 'v, 's) configuration" and
  v :: "bool"
assumes
  "∀ cfg . reachable c cfg ⟶ agreement cfg"
  "⋀cc Q . terminationPseudo 1 cc Q"
  "nonUniform c"
  "vDecided v c"
shows
  "False"
using DecidedImpliesUniform[OF assms(1,2,4)] assms(3)
  by (cases v, simp_all)
```

All three parts of Völzer's Proposition 3 consider a single step from an arbitrary `initReachable` configuration *c* with a message *msg* to a succeeding configuration *c'*.

The silent decision values of a process which is not active in a step only decrease or stay the same.

This follows directly from the definitions and the transitivity of the reachability properties `reachable` and `qReachable`.

*This corresponds to **Proposition 3(a)** in Völzer's paper.*

```
lemma InactiveProcessSilentDecisionValuesDecrease:
fixes
  p q :: 'p and
  c c' :: "('p, 'v, 's) configuration" and
  msg :: "('p, 'v) message"
assumes
  "p ≠ q" and
  "c ⊢ msg ↦ c'" and
  "isReceiverOf p msg" and
  "initReachable c"
shows
  "val[q,c'] ⊆ val[q,c]"
proof(auto simp add: pSilDecVal_def assms(4))
  fix x cfg'
  assume
    Msg: "0 < msgs cfg' <⊥, outM x>" and
    Cfg': "qReachable c' (Proc - {q}) cfg'"
  have "initReachable c'"
    using assms initReachable_def reachable.simps
```

```
    by blast
  hence Init: "initReachable cfg'"
    using Cfg' initReachable_def QReachImplReach[of c' "(Proc - {q})" cfg']
    ReachableTrans by blast
  have "p ∈ Proc - {q}"
    using assms by blast
  hence "qReachable c (Proc - {q}) c'"
    using assms qReachable.simps by metis
  hence "qReachable c (Proc - {q}) cfg'"
    using Cfg' QReachableTrans
    by blast
  with Msg Init show
    "∃c'a. qReachable c (Proc - {q}) c'a
      ∧ initReachable c'a ∧
      0 < msgs c'a <⊥, outM x>" by blast
qed
```

...while the silent decision values of the process which is active in a step may only increase or stay the same.

This follows as stated in [1] from the *diamond property* for a reachable configuration and a single step, i.e. `DiamondTwo`, and in addition from the fact that output messages cannot get lost, i.e. `NoOutMessageLoss`.

This corresponds to ***Proposition 3(b)*** *in Völzer's paper.*

```
lemma ActiveProcessSilentDecisionValuesIncrease :
fixes
  p q :: 'p and
  c c' :: "('p, 'v, 's) configuration" and
  msg :: "('p, 'v) message"
assumes
  "p = q" and
  "c ⊢ msg ↦ c'" and
  "isReceiverOf p msg" and
  "initReachable c"
shows "val[q,c] ⊆ val[q,c']"
proof (auto simp add: pSilDecVal_def assms(4))
  from assms initReachable_def reachable.simps show "initReachable c'"
    by meson
  fix x cv
  assume Cv: "qReachable c (Proc - {q}) cv" "initReachable cv"
    "0 < msgs cv <⊥, outM x>"
  have "∃c'a. (cv ⊢ msg ↦ c'a) ∧ qReachable c' (Proc - {q}) c'a"
    using DiamondTwo Cv(1) assms
    by blast
  then obtain c'' where C'': "(cv ⊢ msg ↦ c'')"
    "qReachable c' (Proc - {q}) c''" by auto
  with Cv(2) initReachable_def reachable.simps
    have Init: "initReachable c''" by blast
  have "reachable cv c''" using C''(1) reachable.intros by blast
  hence "msgs cv <⊥, outM x> ≤ msgs c'' <⊥, outM x>" using NoOutMessageLoss
    by simp
  hence "0 < msgs c'' <⊥, outM x>" using Cv(3) by auto
  thus "∃c'a. qReachable c' (Proc - {q}) c'a
```

```
  ∧ initReachable c'a ∧ 0 < msgs c'a <⊥, outM x>"
  using C''(2) Init by blast
qed
```

As a result from the previous two propositions, the silent decision values of a process cannot go from 0 to 1 or vice versa in a step.

This is a slightly more generic version of Proposition 3 (c) from [1] since it is proven for both values, while Völzer is only interested in the situation starting with *val(q,c) = {0}*.

*This corresponds to **Proposition 3(c)** in Völzer's paper.*

```
lemma SilentDecisionValueNotInverting:
fixes
  p q :: 'p and
  c c' :: "('p, 'v, 's) configuration" and
  msg :: "('p, 'v) message" and
  v :: bool
assumes
  Val: "val[q,c] = {v}" and
  Step:  "c ⊢ msg ↦ c'" and
  Rec:  "isReceiverOf p msg" and
  Init: "initReachable c"
shows
  "val[q,c'] ≠ {¬ v}"
proof(cases "p = q")
  case False
    hence "val[q,c'] ⊆ val[q,c]"
      using Step Rec InactiveProcessSilentDecisionValuesDecrease Init by simp
    with Val show "val[q,c'] ≠ {¬ v}" by auto
  next
  case True
    hence "val[q,c] ⊆ val[q,c']"
      using Step Rec ActiveProcessSilentDecisionValuesIncrease Init by simp
    with Val show "val[q,c'] ≠ {¬ v}" by auto
qed
```

5.5 Towards a proof of FLP

There is an `initial` configuration that is `nonUniform` under the assumption of `validity`, `agreement` and `terminationPseudo`.

The lemma is used in the proof of the main theorem to construct the `nonUniform` and `initial` configuration that leads to the final contradiction.

*This corresponds to **Lemma 1** in Völzer's paper.*

```
lemma InitialNonUniformCfg:
assumes
  Termination: "⋀cc Q . terminationPseudo 1 cc Q" and
  Validity: "∀ i c . validity i c" and
  Agreement: "∀ i c . agreementInit i c"
shows
  "∃ cfg . initial cfg ∧ nonUniform cfg"
proof-
  obtain n::nat where N: "n = card Proc" by blast
  hence "∃ procList::('p list). length procList = n ∧ set procList = Proc
```

```
    ∧ distinct procList"
    using finiteProcs
    by (metis distinct_card finite_distinct_list)
  then obtain procList where
    ProcList: "length procList = n" "set procList = Proc"
      "distinct procList" by blast
  have AllPInProclist: "∀p. ∃i<n. procList ! i = p"
    using ProcList N
  proof auto
    fix p
    assume Asm: "set procList = Proc" "length procList = card Proc"
    have "p ∈ set procList" using ProcList by auto
    with Asm in_set_conv_nth
      show "∃i<card Proc. procList ! i = p" by metis
  qed
  have NGr0: "n > 0"
    using N finiteProcs minimalProcs by auto
  define initMsg :: "nat ⇒ ('p, 'v) message ⇒ nat"
    where "initMsg ind m = (if ∃p. m = <p, inM (∃i<ind. procList!i = p)> then 1
else 0)" for ind m
  define initCfgList
    where "initCfgList = map (λind. (|states = start, msgs = initMsg ind|)) [0..<(n+1)]"
  have InitCfgLength: "length initCfgList = n + 1"
    unfolding initCfgList_def by auto
  have InitCfgNonEmpty: "initCfgList ≠ []"
    unfolding initCfgList_def by auto
  hence InitCfgStart: "(∀c ∈ set initCfgList. states c = start)"
    unfolding initCfgList_def by auto
  have InitCfgSet: "set initCfgList =
    {x. ∃ind < n+1. x = (|states = start, msgs = initMsg ind|)}"
    unfolding initCfgList_def
  proof auto
    fix ind
    assume "ind < n"
    hence "∃inda<Suc n. inda = ind ∧ initMsg ind = initMsg inda" by auto
    thus "∃inda<Suc n. initMsg ind = initMsg inda" by blast
  next
    fix ind
    assume Asm:
      "(|states = start, msgs = initMsg ind|) ∉ (λind::nat. (|states = start, msgs =
initMsg ind|)) ` {0..<n}"
      "ind < Suc n"
    hence "ind ≥ n" by auto
    with Asm have "ind = n" by auto
    thus "initMsg ind = initMsg n" by auto
  qed
  have InitInitial: "∀c ∈ set initCfgList . initial c"
    unfolding initial_def initCfgList_def initMsg_def using InitCfgStart
    by auto
  with InitCfgSet have InitInitReachable:
    "∀ c ∈ set initCfgList . initReachable c"
    using reachable.simps
```

```
  unfolding initReachable_def
  by blast

define c0 where "c0 = initCfgList ! 0"
hence "c0 = ⦇ states = start, msgs = initMsg 0 ⦈"
  using InitCfgLength nth_map_upt[of 0 "n+1" 0]
  unfolding initCfgList_def by auto
hence MsgC0: "msgs c0 = (λm. if ∃p. m = <p, inM False> then 1 else 0)"
  unfolding initMsg_def by simp

define cN where "cN = initCfgList ! n"
hence "cN = ⦇ states = start, msgs = initMsg n⦈"
  using InitCfgLength nth_map_upt[of n "n+1" 0]
  unfolding initCfgList_def
  by auto
hence MsgCN: "msgs cN = (λm. if ∃p. m = <p, inM True> then 1 else 0)"
  unfolding initMsg_def
  using AllPInProclist
  by auto

have C0NotCN: "c0 ≠ cN"
proof
  assume "c0 = cN"
  hence StrangeEq: "(λm::('p, 'v) message.
      if ∃p. m = <p, inM False> then 1 else 0 :: nat) =
    (λm. if ∃p. m = <p, inM True> then 1 else 0)"
    using InitCfgLength N minimalProcs MsgC0 MsgCN
    unfolding c0_def cN_def by auto
  thus "False"
    by (metis message.inject(1) zero_neq_one)
qed

have C0NAreUniform: "⋀ cX . (cX = c0) ∨ (cX = cN)
                      ⟹ vUniform (cX = cN) cX"
proof-
  fix cX
  assume xIs0OrN: "(cX = c0) ∨ (cX = cN)"
  have xInit: "initial cX"
    using InitCfgLength InitCfgSet set_conv_nth[of initCfgList] xIs0OrN
    unfolding c0_def cN_def
    by (auto simp add: InitInitial)
  from Validity
    have COnlyReachesOneDecision:
      "∀ c . reachable cX c ∧ decided c ⟶ (vDecided (cX = cN) c)"
    unfolding validity_def initReachable_def
  proof auto
    fix c cfg0 v
    assume
      Validity: "(∀i c. ((initial i) ∧ (reachable i c)) ⟶
        (∀v. (0 < msgs c (<⊥, outM v>))
        ⟶ (∃p. (0 < msgs i (<p, inM v>)))))" and
      OutMsg: "0 < msgs c <⊥, outM v>" and
```

```
    InitCXReachable: "reachable cX c"
  thus "0 < msgs c <⊥, outM (cX = cN)>"
    using xIs0OrN
  proof (cases "v = (cX = cN)", simp)
    assume "v ≠ (cX = cN)"
    hence vDef: "v = (cX ≠ cN)" by auto
    with Validity InitCXReachable OutMsg xInit
      have ExistMsg: "∃p. (0 < msgs cX (<p, inM (cX ≠ cN)>))" by auto
    with initMsg_def have False
      using xIs0OrN
      by (auto simp add: MsgC0 MsgCN C0NotCN)
    thus "0 < msgs c <⊥, outM cX = cN>" by simp
  qed
qed
have InitRInitC: "initReachable cX" using xInit InitialIsInitReachable
  by auto
have NoWrongDecs: "⋀ v p cc::('p, 'v, 's) configuration.
  qReachable cX (Proc - {p}) cc ∧ initReachable cc
  ∧ 0 < msgs cc <⊥, outM v>
    ⟹ v = (cX = cN)"
proof clarify
  fix v p cc
  assume Asm: "qReachable cX (Proc - {p}) cc" "initReachable cc"
  "0 < msgs cc <⊥, outM v>"
  hence "reachable cX cc" "decided cc" using QReachImplReach by auto
  hence "¬(vDecided (cX ≠ cN) cc)"
    using COnlyReachesOneDecision Agreement Asm C0NotCN xInit xIs0OrN
    unfolding agreementInit_def by metis
  with Asm C0NotCN xIs0OrN show "v = (cX = cN)"
  by (auto, metis (full_types) neq0_conv)
qed
have ExRightDecs: "⋀p. ∃cc. qReachable (cX) (Proc - {p}) cc
  ∧ initReachable cc ∧ 0 < msgs cc <⊥, outM (cX = cN)>"
proof-
  fix p
  show "∃cc::('p, 'v, 's) configuration.
      qReachable cX (Proc - {p}) cc ∧ initReachable cc
      ∧ (0::nat) < msgs cc <⊥, outM cX = cN>"
    using Termination[of "cX" "Proc - {p}"] finiteProcs minimalProcs
      InitRInitC
    unfolding terminationPseudo_def
  proof auto
    fix cc v
    assume
      Asm: "initReachable cX" "qReachable (cX) (Proc - {p}) cc"
      "initReachable cc"  "0 < msgs cc <⊥, outM v>"
    with COnlyReachesOneDecision[rule_format, of "cc"] QReachImplReach
      have "0 < msgs cc <⊥, outM cX = cN>" by auto
    with Asm
      show "∃cc::('p, 'v, 's) configuration.
        qReachable cX (Proc - {p}) cc
        ∧ initReachable cc ∧ (0::nat) < msgs cc <⊥, outM cX = cN>"
```

```
          by blast
      qed
    qed
    have ZeroinPSilent: "∀ p v . v ∈ val[p, cX] ⟷ v = (cX = cN)"
    proof clarify
      fix p v
      show "v ∈ val[p, cX] ⟷ v = (cX = cN)"
        unfolding pSilDecVal_def
        using InitRInitC xIs0OrN C0NotCN NoWrongDecs ExRightDecs by auto
    qed
    thus "vUniform (cX = cN) cX" using InitRInitC
      unfolding vUniform_def by auto
  qed
  hence
    C0Is0Uniform: "vUniform False c0" and
    CNNot0Uniform: "¬ vUniform False cN"
    using C0NAreUniform unfolding vUniform_def using C0NotCN by auto
  hence "∃ j < n. vUniform False (initCfgList ! j)
    ∧ ¬(vUniform False (initCfgList ! Suc j))"
    unfolding c0_def cN_def
    using NatPredicateTippingPoint
      [of n "λx. vUniform False (initCfgList ! x)"] NGr0 by auto
  then obtain j
    where J: "j < n"
      "vUniform False (initCfgList ! j)"
      "¬(vUniform False (initCfgList ! Suc j))" by blast
  define pJ where "pJ = procList ! j"
  define cJ where "cJ = initCfgList ! j"
  hence cJDef: "cJ = ⦇ states = start, msgs = initMsg j⦈"
    using J(1) InitCfgLength nth_map_upt[of 0 "j- 1" 1]
      nth_map_upt[of "j" "n + 1" 0]
    unfolding initCfgList_def
    by auto
  hence MsgCJ: "msgs cJ = (λm::('p, 'v) message.
    if ∃p::'p. m = <p, inM ∃i<j. procList ! i = p> then 1::nat
    else (0::nat))"
    unfolding initMsg_def
    using AllPInProclist
    by auto
  define cJ1 where "cJ1 = initCfgList ! (Suc j)"
  hence cJ1Def: "cJ1 = ⦇ states = start, msgs = initMsg (Suc j)⦈"
    using J(1) InitCfgLength nth_map_upt[of 0 "j" 1]
    nth_map_upt[of "j + 1" "n + 1" 0]
    unfolding initCfgList_def
    by auto
  hence MsgCJ1: "msgs cJ1 = (λm::('p, 'v) message.
    if ∃p::'p. m = <p, inM ∃i<(Suc j). procList ! i = p> then 1::nat
    else (0::nat))"
    unfolding initMsg_def
    using AllPInProclist
    by auto
  have CJ1Init: "initial cJ1"
```

```
    using InitInitial[rule_format, of cJ1]  J(1) InitCfgLength
    unfolding cJ1_def by auto
  hence CJ1InitR: "initReachable cJ1"
    using InitialIsInitReachable by simp
  have ValPj0: "False ∈ val[pJ, cJ]"
    using J(2) unfolding cJ_def vUniform_def by auto
  hence "∃cc. vDecided False cc ∧ withoutQReachable cJ {pJ} cc"
    unfolding pSilDecVal_def by auto
  then obtain cc
    where CC: "vDecided False cc" "withoutQReachable cJ {pJ} cc"
    by blast
    hence "∃Q. qReachable cJ Q cc ∧ Q = Proc - {pJ}" by blast
    then obtain ccQ where CCQ: "qReachable cJ ccQ cc" "ccQ = Proc - {pJ}"
      by blast
  have StatescJcJ1: "states cJ = states cJ1"
    using  cJ_def cJ1_def initCfgList_def
    by (metis InitCfgLength InitCfgStart J(1) Suc_eq_plus1 Suc_mono
      less_SucI nth_mem)
  have Distinct: "⋀ i . distinct procList ⟹ i<j
    ⟹ procList ! i = procList ! j ⟹ False"
    by (metis J(1) ProcList(1) distinct_conv_nth less_trans
      not_less_iff_gr_or_eq)
  have A: "msgs cJ (<pJ ,inM False>) = 1"
    using pJ_def ProcList J(1) MsgCJ using Distinct by auto
  have B: "msgs cJ1 (<pJ ,inM True>) = 1"
    using pJ_def ProcList J(1) MsgCJ1 by auto
  have C: "msgs cJ (<pJ ,inM True>) = 0"
    using pJ_def ProcList J(1) MsgCJ using Distinct by auto
  have D: "msgs cJ1 (<pJ ,inM False>) = 0"
    using pJ_def ProcList J(1) MsgCJ1 by auto
  define msgsCJ' where
    "msgsCJ' m = (if m = (<pJ ,inM False>) ∨ m = (<pJ ,inM True>) then 0 else (msgs
cJ) m)" for m
  have MsgsCJ': "msgsCJ' = ((msgs cJ) -# (<pJ ,inM False>))"
    using A C msgsCJ'_def by auto
  have AllOther: "∀ m . msgsCJ' m = ((msgs cJ1) -# (<pJ ,inM True>)) m"
    using B D MsgCJ1 MsgCJ  J(1) N ProcList AllPInProclist
    unfolding msgsCJ'_def pJ_def
  proof(clarify)
    fix m
    show "(if m = <procList ! j, inM False> ∨
        m = <procList ! j, inM True> then 0 else msgs cJ m)
      = (msgs cJ1 -# <procList ! j, inM True>) m"
    proof(cases "m = <procList ! j, inM False> ∨ m
      = <procList ! j, inM True>",auto)
      assume "0 < (msgs cJ1 <procList ! j, inM False>)"
      thus False using D pJ_def by (metis less_nat_zero_code)
    next
      show "msgs cJ1 <procList ! j, inM True> ≤ Suc 0"
        by (metis B One_nat_def order_refl pJ_def)
    next
      assume AssumpMJ: "m ≠ <procList ! j, inM False>"
```

```
                    "m ≠ <procList ! j, inM True>"
    hence "(if ∃p. m = <p, inM ∃i<j. procList ! i = p> then 1 else 0)
         = (if ∃p. m = <p, inM ∃i<Suc j. procList ! i = p> then 1 else 0)"
      by (induct m, auto simp add: less_Suc_eq)
    thus "msgs cJ m = msgs cJ1 m"
      using MsgCJ MsgCJ1 by auto
  qed
qed — of AllOther

with MsgsCJ' have InitMsgs: "((msgs cJ) -# (<pJ ,inM False>))
  = ((msgs cJ1) -# (<pJ, inM True>))"
  by simp
hence F: "(((msgs cJ) -# (<pJ ,inM False>)) ∪# ({#<pJ, inM True>})) =
  (((msgs cJ1) -# (<pJ, inM True>))  ∪# ({#<pJ, inM True>}))"
  by (metis (full_types))
from B have One: "(((msgs cJ1) -# (<pJ, inM True>))
  ∪# ({#<pJ, inM True>})) <pJ, inM True> = 1" by simp
with B have "∀ m :: ('p, 'v) message . (msgs cJ1) m
  = (((msgs cJ1) -# (<pJ, inM True>))  ∪#
  ({#<pJ, inM True>})) m" by simp
with B have "(((msgs cJ1) -# (<pJ, inM True>))  ∪# ({#<pJ, inM True>}))
  = (msgs cJ1)" by simp
with F have InitMsgs: "(((msgs cJ) -# (<pJ ,inM False>))
  ∪# ({#<pJ, inM True>})) = (msgs cJ1)"
  by simp
define cc' where "cc' = (|states = (states cc),
  msgs = (((msgs cc) -# (<pJ,inM False>)) ∪# {# (<pJ, inM True>)})|)"
have "[|qReachable cJ ccQ cc; ccQ = Proc - {pJ};
  (((msgs cJ) -# (<pJ ,inM False>)) ∪# ({#<pJ, inM True>}))
    = (msgs cJ1); states cJ = states cJ1|]
    ⟹ withoutQReachable cJ1 {pJ} (|states = (states cc),
    msgs = (((msgs cc) -# (<pJ,inM False>)) ∪# {# (<pJ, inM True>)}) |)"
  proof (induct rule: qReachable.induct)
    fix cfg1::"('p, 'v, 's) configuration"
    fix Q
    assume
      Assm: "(((msgs cfg1) -#(<pJ, inM False>)) ∪# {# <pJ, inM True> })
      = msgs cJ1"
      "states cfg1 = states cJ1"
    hence CJ1: "cJ1 = (|states = states cfg1,
      msgs = ((msgs cfg1) -# <pJ, inM False>) ∪# {# <pJ, inM True> }|)" by auto
    have "qReachable cJ1  (Proc - {pJ}) cJ1" using qReachable.simps
      by blast
    with Assm show "qReachable cJ1 (Proc - {pJ})
      (|states = states cfg1, msgs = ((msgs cfg1) -# <pJ, inM False>)
      ∪# {# <pJ, inM True> }|)" using CJ1 by blast
    fix cfg1 cfg2 cfg3 :: "('p, 'v, 's) configuration"
    fix msg
    assume Q: "Q = (Proc - {pJ})"
    assume "(((msgs cfg1) -# <pJ, inM False>) ∪# {# <pJ, inM True> })
      = (msgs cJ1)"
      "(states cfg1) = (states cJ1)"
```

```
  "Q = (Proc - {pJ}) ⟹
    (((msgs cfg1) -# <pJ, inM False>) ∪# {# <pJ, inM True> })
      = (msgs cJ1)
    ⟹ (states cfg1) = (states cJ1)
    ⟹ qReachable cJ1 (Proc - {pJ})
    (|states = (states cfg2),
    msgs = (((msgs cfg2) -# <pJ, inM False>) ∪# {# <pJ, inM True> })|)"
with Q have Cfg2:
  "qReachable cJ1 (Proc - {pJ}) (|states = (states cfg2),
  msgs = (((msgs cfg2) -# <pJ, inM False>) ∪# {# <pJ, inM True> })|)"
  by simp
assume "qReachable cfg1 Q cfg2"
  "cfg2 ⊢ msg ↦ cfg3"
  "∃(p::'p)∈Q. (isReceiverOf p msg)"
with Q have Step: "qReachable cfg1 (Proc - {pJ}) cfg2"
  "cfg2 ⊢ msg ↦ cfg3"
  "∃(p::'p)∈(Proc - {pJ}). (isReceiverOf p msg)" by auto
then obtain p where P: "p∈(Proc - {pJ})" "isReceiverOf p msg" by blast
hence NotEq: "pJ ≠ p" by blast
with UniqueReceiverOf[of "p" "msg" "pJ"] P(2)
  have notRec: "¬ (isReceiverOf pJ msg)" by blast
hence MsgNoIn:"msg ≠  <pJ, inM False> ∧ msg ≠  <pJ, inM True>"
  by auto
from Step(2) have "enabled cfg2 msg" using steps.simps
  by (auto, cases msg, auto)
hence MsgEnabled: "enabled (|states = (states cfg2),
  msgs = (((msgs cfg2) -# <pJ, inM False>)
  ∪# {# <pJ, inM True> })|) msg"
  using MsgNoIn by (simp add: enabled_def)
have  "(|states = (states cfg2),
        msgs = (((msgs cfg2) -# <pJ, inM False>)
        ∪# {# <pJ, inM True> })|)
          ⊢ msg ↦ (|states = (states cfg3),
          msgs = (((msgs cfg3) -# <pJ, inM False>)
          ∪# {# <pJ, inM True> })|)"
proof (cases msg)
  fix p' bool
  assume MsgIn :"(msg = <p', inM bool>)"
  with noInSends MsgIn MsgNoIn MsgEnabled
    show "(|states = (states cfg2),
    msgs = (((msgs cfg2) -# <pJ, inM False>) ∪# {# <pJ, inM True> })|)
      ⊢ msg ↦ (|states = (states cfg3),
        msgs = (((msgs cfg3) -# <pJ, inM False>)
        ∪# {# <pJ, inM True> })|)"
    using steps.simps(1) Step(2) select_convs(2) select_convs(1)
    by auto
next
  fix bool
  assume "(msg = <⊥, outM bool>)"
  thus "(|states = (states cfg2),
    msgs = (((msgs cfg2) -# <pJ, inM False>) ∪# {# <pJ, inM True> })|)
      ⊢ msg ↦ (|states = (states cfg3),
```

```
              msgs = (((msgs cfg3) -# <pJ, inM False>)
              ∪# {# <pJ, inM True> })⦈"
          using steps_def Step(2) by auto
      next
        fix p v
        assume "(msg = <p, v>)"
        with noInSends MsgNoIn MsgEnabled show "⦇states = (states cfg2),
          msgs = (((msgs cfg2) -# <pJ, inM False>) ∪# {# <pJ, inM True> })⦈
            ⊢ msg ↦ ⦇states = (states cfg3),
              msgs = (((msgs cfg3) -# <pJ, inM False>)
              ∪# {# <pJ, inM True> })⦈"
        using steps.simps(1) Step(2) select_convs(2) select_convs(1) by auto
      qed
      with Cfg2 Step(3) show "qReachable cJ1 (Proc - {pJ})
        ⦇states = (states cfg3),
          msgs = (((msgs cfg3) -# <pJ, inM False>) ∪# {# <pJ, inM True> })⦈"
        using
          qReachable.simps[of "cJ1" "(Proc - {pJ})"
          "⦇states = (states cfg3),
          msgs = (((msgs cfg3) -# <pJ, inM False>)
          ∪# {# <pJ, inM True> })⦈"] by auto
    qed
  with CCQ(1) CCQ(2) InitMsgs StatescJcJ1 have CC':
    "withoutQReachable cJ1 {pJ} ⦇states = (states cc),
      msgs = (((msgs cc) -# (<pJ,inM False>))
      ∪# {# (<pJ, inM True>)}) ⦈" by auto
  with QReachImplReach CJ1InitR initReachable_def reachable.simps
    ReachableTrans
    have "initReachable ⦇states = (states cc),
      msgs = (((msgs cc) -# (<pJ,inM False>))
      ∪# {# (<pJ, inM True>)}) ⦈" by meson
  with CC' have "False ∈ val[pJ, cJ1]"
    unfolding pSilDecVal_def
    using CJ1InitR CC(1) select_convs(2) by auto
  hence "¬(vUniform True (cJ1))"
    unfolding vUniform_def
    using CJ1InitR by blast
  hence "nonUniform cJ1"
    using J(3) CJ1InitR unfolding cJ1_def by auto
  thus ?thesis
    using CJ1Init by blast
qed
```

Völzer's Lemma 2 proves that for every process p in the consensus setting `nonUniform` configurations can reach a configuration where the silent decision values of p are True and False. This is key to the construction of non-deciding executions.

*This corresponds to **Lemma 2** in Völzer's paper.*

```
lemma NonUniformCanReachSilentBivalence:
fixes
  p:: 'p and
  c:: "('p, 'v, 's) configuration"
assumes
```

```
  NonUni: "nonUniform c" and
  PseudoTermination: "⋀cc Q . terminationPseudo 1 cc Q" and
  Agree: "⋀ cfg . reachable c cfg ⟶ agreement cfg"
shows
   "∃ c' . reachable c c' ∧ val[p,c'] = {True, False}"
proof(cases "val[p,c] = {True, False}")
  case True
  have "reachable c c" using reachable.simps by metis
  thus ?thesis
    using True by blast
next
  case False
  hence notEq: "val[p,c] ≠ {True, False}" by simp
  from NonUni PseudoTermination DecisionValuesExist
  have notE: "∀q. val[q,c] ≠ {}" by simp
  hence notEP: "val[p,c] ≠ {}" by blast
  have valType: "∀q. val[q,c] ⊆ {True, False}" using pSilDecVal_def
    by (metis (full_types) insertCI subsetI)
  hence "val[p,c] ⊆ {True, False}" by blast
  with notEq notEP  have "∃b:: bool. val[p,c] = {b}" by blast
  then obtain b where B: "val[p,c] = {b}" by auto
  from NonUni vUniform_def have
    NonUni: "(∃q. val[q,c] ≠ {True}) ∧ (∃q. val[q,c] ≠ {False})" by simp
  have Bool: "b = True ∨ b = False" by simp
  with NonUni have "∃q. val[q,c] ≠ {b}" by blast
  then obtain q where Q: "val[q,c] ≠ {b}" by auto
  from notE valType
  have "val[q,c] = {True} ∨ val[q,c] = {False} ∨ val[q,c] = {True, False}"
    by auto
  with Bool Q have "val[q,c] = {¬b} ∨ val[q,c] = {b, ¬b}" by blast
  hence "(¬b) ∈ val[q,c]" by blast
  with pSilDecVal_def
  have "∃ c'::('p, 'v, 's) configuration . (withoutQReachable c {q} c') ∧
    vDecided (¬b) c'" by simp
  then obtain c' where C': "withoutQReachable c {q} c'" "vDecided (¬b) c'"
    by auto
  hence Reach: "reachable c c'" using QReachImplReach by simp
  have "∀ cfg . reachable c' cfg ⟶ agreement cfg"
  proof clarify
    fix cfg
    assume "reachable c' cfg"
    with Reach have "reachable c cfg"
      using ReachableTrans[of c c'] by simp
    with Agree show "agreement cfg" by simp
  qed
  with PseudoTermination C'(2) DecidedImpliesUniform have "vUniform (¬b) c'"
    by simp
  hence notB: "val[p,c'] = {¬b}" using vUniform_def by simp
  with Reach B show "∃ cfg. reachable c cfg ∧ val[p,cfg] = {True, False}"
  proof(induct rule: reachable.induct, simp)
    fix cfg1 cfg2 cfg3 msg
    assume
```

```
  IV: "val[p,cfg1] = {b} ⟹
    val[p,cfg2] = {¬ b} ⟹
      ∃cfg::('p, 'v, 's) configuration. reachable cfg1 cfg
      ∧ val[p,cfg] = {True, False}" and
  Reach:  "reachable cfg1 cfg2" and
  Step: "cfg2 ⊢ msg ↦ cfg3" and
  ValCfg1: "val[p,cfg1] = {b}" and
  ValCfg3: "val[p,cfg3] = {¬ b}"
from ValCfg1 have InitCfg1: "initReachable cfg1"
  using pSilDecVal_def by auto
from Reach InitCfg1 initReachable_def reachable.simps ReachableTrans
  have InitCfg2: "initReachable cfg2" by blast
with PseudoTermination DecisionValuesExist
have notE: "∀q. val[q,cfg2] ≠ {}" by simp
have valType: "∀q. val[q,cfg2] ⊆ {True, False}" using pSilDecVal_def
  by (metis (full_types) insertCI subsetI)
from notE valType
  have "val[p,cfg2] = {True} ∨ val[p,cfg2] = {False}
    ∨ val[p,cfg2] = {True, False}"
  by auto
with Bool have Val: "val[p,cfg2] = {b} ∨ val[p,cfg2] = {¬b}
  ∨ val[p,cfg2] = {True, False}"
  by blast
show "∃cfg::('p, 'v, 's) configuration. reachable cfg1 cfg
  ∧ val[p,cfg] = {True, False}"
proof(cases "val[p,cfg2] = {b}")
  case True
  hence B: "val[p,cfg2] = {b}" by simp
  from Step have RecOrOut: "∃q. isReceiverOf q msg" by(cases msg, auto)
  then obtain q where Rec: "isReceiverOf q msg" by auto
  with B Step InitCfg2 SilentDecisionValueNotInverting
  have "val[p,cfg3] ≠ {¬b}" by simp
  with ValCfg3 have "False" by simp
  thus "∃cfg::('p, 'v, 's) configuration. reachable cfg1 cfg ∧
    val[p,cfg] = {True, False}" by simp
next
  case False
  with Val have Val: "val[p,cfg2] = {¬b} ∨ val[p,cfg2] = {True, False}"
    by simp
  show "∃cfg::('p, 'v, 's) configuration. reachable cfg1 cfg ∧
        val[p,cfg] = {True, False}"
  proof(cases "val[p,cfg2] = {¬b}")
    case True
    hence "val[p,cfg2] = {¬b}" by simp
    with ValCfg1 IV show
      "∃cfg::('p, 'v, 's) configuration.
      reachable cfg1 cfg ∧ val[p,cfg] = {True, False}"
      by simp
  next
    case False
    with Val have "val[p,cfg2] = {True, False}" by simp
    with Reach have "reachable cfg1 cfg2 ∧ val[p,cfg2] = {True, False}"
```

```
        by blast
      from this show "∃cfg::('p, 'v, 's) configuration.
        reachable cfg1 cfg ∧ val[p,cfg] = {True, False}" by blast
    qed
  qed
 qed
qed

end
end
```

6 FLPTheorem

FLPTheorem combines the results of **FLPSystem** with the concept of fair infinite executions and culminates in showing the impossibility of a consensus algorithm in the proposed setting.

```
theory FLPTheorem
imports Execution FLPSystem
begin

locale flpPseudoConsensus =
  flpSystem trans sends start
for
  trans :: "'p ⇒ 's ⇒ 'v messageValue ⇒'s" and
  sends :: "'p ⇒ 's ⇒ 'v messageValue ⇒ ('p, 'v) message multiset" and
  start :: "'p ⇒ 's" +
assumes
  Agreement: "⋀ i c . agreementInit i c" and
  PseudoTermination: "⋀cc Q . terminationPseudo 1 cc Q"
begin
```

6.1 Obtaining non-uniform executions

Executions which end with a `nonUniform` configuration can be expanded to a strictly longer execution consuming a particular message.

This lemma connects the previous results to the world of executions, thereby paving the way to the final contradiction. It covers a big part of the original proof of the theorem, i.e. finding the expansion to a longer execution where the decision for both values is still possible. *This corresponds to **constructing executions using Lemma 2** in Völzer's paper.*

```
lemma NonUniformExecutionsConstructable:
fixes
  exec :: "('p, 'v, 's ) configuration list " and
  trace :: "('p, 'v) message list" and
  msg :: "('p, 'v) message" and
  p :: 'p
assumes
  MsgEnabled: "enabled (last exec) msg" and
  PisReceiverOf: "isReceiverOf p msg" and
  ExecIsExecution: "execution trans sends start exec trace" and
```

```
  NonUniformLexec: "nonUniform (last exec)" and
  Agree: "⋀ cfg . reachable (last exec) cfg ⟶ agreement cfg"
shows
  "∃ exec' trace' . (execution trans sends start exec' trace')
    ∧ nonUniform (last exec')
    ∧ prefixList exec exec' ∧ prefixList trace trace'
    ∧ (∀ cfg . reachable (last exec') cfg ⟶ agreement cfg)
    ∧ stepReachable (last exec) msg (last exec')
    ∧ (msg ∈ set (drop (length trace) trace'))"
proof -
  from NonUniformCanReachSilentBivalence[OF NonUniformLexec PseudoTermination Agree]
    obtain c' where C':
      "reachable (last exec) c'"
      "val[p,c'] = {True, False}"
    by blast
  show ?thesis
  proof (cases "stepReachable (last exec) msg c'")
    case True
    hence IsStepReachable: "stepReachable (last exec) msg c'" by simp
    hence "∃ exec' trace'. (execution trans sends start exec' trace')
      ∧ prefixList exec exec'
      ∧ prefixList trace trace' ∧ (last exec') = c'
      ∧ msg ∈ set (drop (length trace) trace')"
      using ExecIsExecution expandExecution
      by auto
    then obtain exec' trace' where NewExec:
      "(execution trans sends start exec' trace')"
      "prefixList exec exec'" "(last exec') = c'" "prefixList trace trace'"
      "msg ∈ set (drop (length trace) trace')" by blast
    hence lastExecExec'Reachable: "reachable (last exec) (last exec')"
      using C'(1) by simp
    hence InitReachLastExec':  "initReachable (last exec')"
      using NonUniformLexec
      by (metis ReachableTrans initReachable_def)
    hence nonUniformC': "nonUniform (last exec')" using C'(2) NewExec(3)
      by (auto simp add: vUniform_def)
    hence isAgreementPreventing:
      "(∀ cfg . reachable (last exec') cfg ⟶ agreement cfg)"
      using lastExecExec'Reachable Agree by (metis ReachableTrans)
    with NewExec nonUniformC' IsStepReachable show ?thesis by auto
  next
    case False
    hence NotStepReachable: "¬ (stepReachable (last exec) msg c')" by simp
    from C'(1) obtain exec' trace' where NewExec:
      "execution trans sends start exec' trace'"
      "(prefixList exec exec' ∧ prefixList trace trace')
      ∨ (exec = exec' ∧ trace = trace')"
      "last exec' = c'"
      using ExecIsExecution expandExecutionReachable by blast
    have lastExecExec'Reachable: "reachable (last exec) (last exec')"
      using C'(1) NewExec(3) by simp
    with NonUniformLexec have InitReachLastExec':
```

```
    "initReachable (last exec')"
    by (metis ReachableTrans initReachable_def)
  with C'(2) NewExec(3) have nonUniformC': "nonUniform (last exec')"
    by (auto simp add: vUniform_def)
  show "∃ exec1 trace1 . (execution trans sends start exec1 trace1)
    ∧ nonUniform (last exec1)
    ∧ prefixList exec exec1 ∧ prefixList trace trace1
    ∧ (∀ cfg . reachable (last exec1) cfg ⟶ agreement cfg)
    ∧ stepReachable (last exec) msg (last exec1)
    ∧ (msg ∈ set (drop (length trace) trace1))"
  proof (cases "enabled (last exec') msg")
    case True
    hence EnabledMsg: "enabled (last exec') msg" by auto
    hence "∃ cMsg . ((last exec') ⊢ msg ↦ cMsg )"
    proof (cases msg)
      case (InMsg p' b)
      with PisReceiverOf have MsgIsInMsg: "(msg = <p, inM b>)" by auto
      define cfgInM where "cfgInM = ⦇states = λproc. (
      if proc = p then
        trans p (states (last exec') p) (Bool b)
      else states (last exec') proc),
      msgs = (((sends p (states (last exec') p) (Bool b))
            ∪# (msgs (last exec')-# msg)))⦈ "
        with UniqueReceiverOf MsgIsInMsg EnabledMsg have
        "((last exec') ⊢ msg ↦ cfgInM)" by auto
      thus "∃ cMsg . ((last exec') ⊢ msg ↦ cMsg )" by blast
    next
      case (OutMsg b)
      thus "∃ cMsg . ((last exec') ⊢ msg ↦ cMsg )" using PisReceiverOf
        by auto
    next
      case (Msg p' v')
      with PisReceiverOf have MsgIsVMsg: "(msg = <p, v'>)" by auto
      define cfgVMsg where "cfgVMsg =
        ⦇states = λproc. (
            if proc = p then
             trans p (states (last exec') p) (Value v')
             else states (last exec') proc),
          msgs = (((sends p (states (last exec') p) (Value v'))
              ∪# (msgs (last exec') -# msg )))⦈ "
      with UniqueReceiverOf MsgIsVMsg EnabledMsg noInSends have
        "((last exec') ⊢ msg ↦ cfgVMsg)" by auto
      thus "∃ cMsg . ((last exec') ⊢ msg ↦ cMsg )" by blast
    qed
    then obtain cMsg where  CMsg:"((last exec') ⊢ msg ↦ cMsg )" by auto
    define execMsg where "execMsg = exec' @ [cMsg]"
    define traceMsg where "traceMsg = trace' @ [msg]"
    from NewExec(1) CMsg obtain execMsg traceMsg where isExecution:
      "execution trans sends start execMsg traceMsg"
      and ExecMsg: "prefixList exec' execMsg" "prefixList trace' traceMsg"
      "last execMsg = cMsg" "last traceMsg = msg"
      using expandExecutionStep by blast
```

```
      have isPrefixListExec: "prefixList exec execMsg"
        using PrefixListTransitive NewExec(2) ExecMsg(1) by auto
      have isPrefixListTrace: "prefixList trace traceMsg"
        using PrefixListTransitive NewExec(2) ExecMsg(2) by auto
      have cMsgLastReachable: "reachable cMsg (last execMsg)"
        by (auto simp add: ExecMsg reachable.init)
      hence isStepReachable: "stepReachable (last exec) msg (last execMsg)"
        using CMsg lastExecExec'Reachable
        by (auto simp add: stepReachable_def)
      have InitReachLastExecMsg: "initReachable (last execMsg)"
        using CMsg InitReachLastExec' cMsgLastReachable
        by (metis ReachableTrans initReachable_def step)
      have "val[p, (last exec')] ⊆ val[p, cMsg]"
        using CMsg PisReceiverOf InitReachLastExec'
          ActiveProcessSilentDecisionValuesIncrease[of p p "last exec'" msg cMsg]
        by auto
      with ExecMsg C'(2) NewExec(3) have
        "val[p, (last execMsg)] = {True, False}" by auto
      with InitReachLastExecMsg have isNonUniform:
        "nonUniform (last execMsg)" by (auto simp add: vUniform_def)
      have "reachable (last exec) (last execMsg)"
        using lastExecExec'Reachable cMsgLastReachable CMsg
        by (metis ReachableTrans step)
      hence isAgreementPreventing:
        "(∀ cfg . reachable (last execMsg) cfg ⟶ agreement cfg)"
        using Agree by (metis ReachableTrans)
      have "msg ∈ set (drop (length trace) traceMsg)" using ExecMsg(4)
        isPrefixListTrace
        by (metis (full_types) PrefixListMonotonicity last_drop last_in_set
          length_0_conv length_drop less_zeroE zero_less_diff)
      thus ?thesis using isExecution isNonUniform isPrefixListExec
        isPrefixListTrace isAgreementPreventing isStepReachable by blast
    next
      case False
      hence notEnabled: "¬ (enabled (last exec') msg)" by auto
      have isStepReachable: "stepReachable (last exec) msg (last exec')"
        using MsgEnabled notEnabled lastExecExec'Reachable StepReachable
          by auto
        with NotStepReachable NewExec(3) show ?thesis by simp
    qed
  qed
qed

lemma NonUniformExecutionBase:
fixes
  cfg
assumes
  Cfg: "initial cfg" "nonUniform cfg"
shows
  "execution trans sends start [cfg] []
  ∧ nonUniform (last [cfg])
  ∧ (∃ cfgList' msgList'.  nonUniform (last cfgList')
```

```
    ∧ prefixList [cfg] cfgList'
    ∧ prefixList [] msgList'
    ∧ (execution trans sends start cfgList' msgList')
     ∧ (∃ msg'. execution.minimalEnabled [cfg] [] msg'
         ∧ msg' ∈ set msgList'))"
proof -
  have NonUniListCfg: "nonUniform (last [cfg])" using Cfg(2) by auto
  have AgreeCfg': "∀ cfg' .
    reachable (last [cfg]) cfg' ⟶ agreement cfg'"
    using Agreement Cfg(1)
    by (auto simp add: agreementInit_def reachable.init agreement_def)
  have StartExec: "execution trans sends start [cfg] []"
    using Cfg(1) by (unfold_locales, auto)
  hence "∃ msg . execution.minimalEnabled [cfg] [] msg"
    using Cfg execution.ExistImpliesMinEnabled
    by (metis enabled_def initial_def isReceiverOf.simps(1)
       last.simps zero_less_one)
  then obtain msg where MinEnabledMsg:
    "execution.minimalEnabled [cfg] [] msg" by blast
  hence "∃ pMin . isReceiverOf pMin msg" using StartExec
    by (auto simp add: execution.minimalEnabled_def)
  then obtain pMin where PMin: "isReceiverOf pMin msg" by blast
  hence "enabled (last [cfg]) msg ∧ isReceiverOf pMin msg"
    using MinEnabledMsg StartExec
    by (auto simp add: execution.minimalEnabled_def)
  hence Enabled: "enabled (last [cfg]) msg" "isReceiverOf pMin msg"
    by auto
  from Enabled StartExec NonUniListCfg PseudoTermination AgreeCfg'
  have "∃ exec' trace' . (execution trans sends start exec' trace')
    ∧ nonUniform (last exec')
    ∧ prefixList [cfg] exec' ∧ prefixList [] trace'
    ∧ (∀ cfg' . reachable (last exec') cfg' ⟶ agreement cfg')
    ∧ stepReachable (last [cfg]) msg (last exec')
    ∧ (msg ∈ set (drop (length []) trace'))"
  using NonUniformExecutionsConstructable[of "[cfg]" "msg" "pMin"
        "[]::('p,'v) message list"]
    by simp
  with StartExec NonUniListCfg MinEnabledMsg show ?thesis by auto
qed

lemma NonUniformExecutionStep:
fixes
 cfgList msgList
assumes
  Init: "initial (hd cfgList)" and
  NonUni: "nonUniform (last cfgList)" and
  Execution: "execution trans sends start cfgList msgList"
shows
  "(∃ cfgList' msgList' .
      nonUniform (last cfgList')
      ∧ prefixList cfgList cfgList'
      ∧ prefixList msgList msgList'
```

```
      ∧ (execution trans sends start cfgList' msgList')
      ∧ (initial (hd cfgList'))
      ∧ (∃ msg'. execution.minimalEnabled cfgList msgList msg'
        ∧ msg' ∈ (set (drop (length msgList ) msgList')) ))"
proof -
  have ReachImplAgree: "∀ cfg . reachable (last cfgList) cfg
    ⟶ agreement cfg"
    using Agreement Init NonUni ReachableTrans
    unfolding agreementInit_def agreement_def initReachable_def
    by (metis (full_types))
  have "∃ msg p. enabled (last cfgList) msg ∧ isReceiverOf p msg"
  proof -
    from PseudoTermination NonUni have
      "∃c'. qReachable (last cfgList) Proc c' ∧ decided c'"
      using terminationPseudo_def by auto
      then obtain c' where C': "reachable (last cfgList) c'"
        "decided c'"
        using QReachImplReach by blast
    have NoOut:
      "0 = msgs (last cfgList) <⊥, outM False>"
      "0 = msgs (last cfgList) <⊥, outM True>"
      using NonUni ReachImplAgree PseudoTermination
      by (metis NonUniformImpliesNotDecided neq0_conv)+
    with C'(2) have "(last cfgList) ≠ c'"
      by (metis (full_types) less_zeroE)
    thus ?thesis using C'(1) ReachableStepFirst by blast
  qed
  then obtain msg p where Enabled:
    "enabled (last cfgList) msg" "isReceiverOf p msg" by blast
  hence "∃ msg . execution.minimalEnabled cfgList msgList msg"
    using Init execution.ExistImpliesMinEnabled[OF Execution] by auto
  then obtain msg' where MinEnabledMsg:
    "execution.minimalEnabled cfgList msgList msg'" by blast
  hence "∃ p' . isReceiverOf p' msg'"
    using Execution
    by (auto simp add: execution.minimalEnabled_def)
  then obtain p' where
    P': "isReceiverOf p' msg'" by blast
  hence Enabled':
    "enabled (last cfgList) msg'" "isReceiverOf p' msg'"
    using MinEnabledMsg Execution
    by (auto simp add: execution.minimalEnabled_def)
  have "∃ exec' trace' . (execution trans sends start exec' trace')
    ∧ nonUniform (last exec')
    ∧ prefixList cfgList exec' ∧ prefixList msgList trace'
    ∧ (∀ cfg . reachable (last exec') cfg ⟶ agreement cfg)
    ∧ stepReachable (last cfgList) msg' (last exec')
    ∧ (msg' ∈ set (drop (length msgList) trace')) "
    using NonUniformExecutionsConstructable[OF Enabled' Execution
          NonUni] ReachImplAgree by auto
  thus ?thesis
    using MinEnabledMsg by (metis execution.base)
```

```
qed
```

6.2 Non-uniformity even when demanding fairness

Using NonUniformExecutionBase and NonUniformExecutionStep one can obtain non-uniform executions which are fair.

Proving the fairness turned out quite cumbersome.

These two functions construct infinite series of configurations lists and message lists from two extension functions.

```
fun infiniteExecutionCfg ::
  "('p, 'v, 's) configuration ⇒
   (('p, 'v, 's) configuration list ⇒ ('p, 'v) message list
    ⇒ ('p, 'v, 's) configuration list) ⇒
   (('p, 'v, 's) configuration list ⇒ ('p, 'v) message list
    ⇒('p, 'v) message list)
  ⇒ nat
  ⇒ (('p, 'v, 's) configuration list)"
and  infiniteExecutionMsg ::
  "('p, 'v, 's) configuration ⇒
   (('p, 'v, 's) configuration list ⇒ ('p, 'v) message list
    ⇒ ('p, 'v, 's) configuration list) ⇒
   (('p, 'v, 's) configuration list ⇒ ('p, 'v) message list
    ⇒('p, 'v) message list)
  ⇒ nat
  ⇒ ('p, 'v) message list"
where
  "infiniteExecutionCfg cfg fStepCfg fStepMsg 0 = [cfg]"
| "infiniteExecutionCfg cfg fStepCfg fStepMsg (Suc n) =
    fStepCfg (infiniteExecutionCfg cfg fStepCfg fStepMsg n)
             (infiniteExecutionMsg cfg fStepCfg fStepMsg n)"
| "infiniteExecutionMsg cfg fStepCfg fStepMsg 0 = []"
| "infiniteExecutionMsg cfg fStepCfg fStepMsg (Suc n) =
    fStepMsg (infiniteExecutionCfg cfg fStepCfg fStepMsg n)
             (infiniteExecutionMsg cfg fStepCfg fStepMsg n)"

lemma FairNonUniformExecution:
fixes
  cfg
assumes
  Cfg: "initial cfg" "nonUniform cfg"
shows "∃ fe ft.
  (fe 0) = [cfg]
  ∧ fairInfiniteExecution fe ft
  ∧ (∀ n . nonUniform (last (fe n))
       ∧ prefixList (fe n) (fe (n+1))
       ∧ prefixList (ft n) (ft (n+1))
       ∧ (execution trans sends start (fe n) (ft n)))"
proof -
  have BC:
    "execution trans sends start [cfg] []
    ∧ nonUniform (last [cfg])
```

```
  ∧ (∃ cfgList' msgList'.  nonUniform (last cfgList')
  ∧ prefixList [cfg] cfgList'
  ∧ prefixList [] msgList'
  ∧ (execution trans sends start cfgList' msgList')
  ∧ (∃ msg'. execution.minimalEnabled [cfg] [] msg'
     ∧ msg' ∈ set msgList'))"
  using NonUniformExecutionBase[OF assms] .
```

— fStep ... a step leading to a fair execution.

```
obtain fStepCfg fStepMsg where FStep: "∀ cfgList msgList . ∃cfgList' msgList'
.
        fStepCfg cfgList msgList = cfgList' ∧
        fStepMsg cfgList msgList = msgList' ∧
        (initial (hd cfgList) ∧
        nonUniform (last cfgList) ∧
        execution trans sends start cfgList msgList ⟶
        (nonUniform (last (fStepCfg cfgList msgList))
        ∧ prefixList cfgList (fStepCfg cfgList msgList)
        ∧ prefixList msgList (fStepMsg cfgList msgList)
        ∧ execution trans sends start (fStepCfg cfgList msgList)
            (fStepMsg cfgList msgList)
        ∧ (initial (hd (fStepCfg cfgList msgList)))
        ∧ (∃ msg'. execution.minimalEnabled cfgList msgList msg'
          ∧ msg' ∈ (set (drop (length msgList)
                              (fStepMsg cfgList msgList))))))"
 using NonUniformExecutionStep
    PredicatePairFunctions2[of
      "λ cfgList msgList cfgList' msgList'.
        (initial (hd cfgList)
        ∧ nonUniform (last cfgList)
        ∧ execution trans sends start cfgList msgList
          ⟶ (nonUniform (last cfgList')
          ∧ prefixList cfgList cfgList'
          ∧ prefixList msgList msgList'
          ∧ execution trans sends start cfgList' msgList'
          ∧ (initial (hd cfgList'))
          ∧ (∃ msg'. execution.minimalEnabled cfgList msgList msg'
            ∧ msg' ∈ (set (drop (length msgList ) msgList')))))" "False"] by auto
define fe ft
  where "fe = infiniteExecutionCfg cfg fStepCfg fStepMsg"
    and "ft = infiniteExecutionMsg cfg fStepCfg fStepMsg"

have BasicProperties: "(∀n. nonUniform (last (fe n))
  ∧ prefixList (fe n) (fe (n + 1)) ∧ prefixList (ft n) (ft (n + 1))
  ∧ execution trans sends start (fe n) (ft n)
  ∧ initial (hd (fe (n + 1))))"
proof (clarify)
  fix n
  show "nonUniform (last (fe n)) ∧
        prefixList (fe n) (fe (n + (1::nat)))
        ∧ prefixList (ft n) (ft (n + (1::nat)))
        ∧ execution trans sends start (fe n) (ft n)
        ∧ initial (hd (fe (n + 1)))"
```

```
  proof(induct n)
    case 0
    hence "fe 0 = [cfg]" "ft 0 = []" "fe 1 = fStepCfg (fe 0) (ft 0)"
      "ft 1 = fStepMsg (fe 0) (ft 0)"
      using fe_def ft_def
      by simp_all
    thus ?case
      using BC FStep
      by (simp, metis execution.base)
  next
    case (Suc n)
      thus ?case
      using fe_def ft_def
      by (auto, (metis FStep execution.base)+)
  qed
qed
have Fair: "fairInfiniteExecution fe ft"
  using BasicProperties
  unfolding fairInfiniteExecution_def infiniteExecution_def
    execution_def flpSystem_def
proof(auto simp add: finiteProcs minimalProcs finiteSends noInSends)
  fix n n0 p msg
  assume AssumptionFair: "∀n. initReachable (last (fe n)) ∧
     ¬ vUniform False (last (fe n)) ∧
     ¬ vUniform True (last (fe n)) ∧
     prefixList (fe n) (fe (Suc n)) ∧
     prefixList (ft n) (ft (Suc n)) ∧
     Suc 0 ≤ length (fe n) ∧
     length (fe n) - Suc 0 = length (ft n) ∧
     initial (hd (fe n)) ∧
     (∀i<length (fe n) - Suc 0. ((fe n ! i) ⊢ (ft n ! i)
     ↦ (fe n ! Suc i))) ∧ initial (hd (fe (Suc n)))"
    "n0 < length (fe n)"
    "enabled (fe n ! n0) msg"
    "isReceiverOf p msg"
    "correctInfinite fe ft p"
  have MessageStaysOrConsumed: "⋀ n n1 n2 msg.
    (n1 ≤ n2 ∧ n2 < length (fe n) ∧ (enabled (fe n ! n1) msg))
    ⟶ (enabled (fe n ! n2) msg)
        ∨ (∃ n0' ≥ n1. n0' < length (ft n) ∧ ft n ! n0' = msg)"
  proof(auto)
    fix n n1 n2 msg
    assume Ass: "n1 ≤ n2" "n2 < length (fe n)" "enabled (fe n ! n1) msg"
      "∀index<length (ft n). n1 ≤ index ⟶ ft n ! index ≠ msg"
    have "∀ k ≤ n2 - n1 .
      msgs (fe n ! n1) msg ≤ msgs (fe n ! (n1 + k)) msg"
    proof(auto)
      fix k
      show "k ≤ n2 - n1 ⟹
        msgs (fe n ! n1) msg ≤ msgs (fe n ! (n1 + k)) msg"
      proof(induct k, auto)
        fix k
```

```
      assume IV: "msgs (fe n ! n1) msg ≤ msgs (fe n ! (n1 + k)) msg"
        "Suc k ≤ n2 - n1"
      from BasicProperties have Exec:
        "execution trans sends start (fe n) (ft n)" by blast
      have "n2 ≤ length (ft n)"
        using Exec Ass(2)
        execution.length[of trans sends start "fe n" "ft n"]
        by simp
      hence RightIndex: "n1 + k ≥ n1 ∧ n1 + k < length (ft n)"
        using IV(2) by simp
      have  Step: "(fe n ! (n1 + k)) ⊢ (ft n ! (n1 + k))
               ↦ (fe n ! Suc (n1 + k))"
        using Exec execution.step[of trans sends start "fe n" "ft n"
          "n1 + k" "fe n ! (n1 + k)" "fe n ! (n1 + k + 1)"] IV(2)
          Ass(2)
        by simp
      hence "msg ≠ (ft n ! (n1 + k))"
        using Ass(4) Ass(2) IV(2) RightIndex Exec
        execution.length[of trans sends start "fe n" "ft n"]
        by blast
      thus "msgs (fe n ! n1) msg ≤ msgs (fe n ! Suc (n1 + k)) msg"
        using Step OtherMessagesOnlyGrowing[of "(fe n ! (n1 + k))"
          "(ft n ! (n1 + k))" "(fe n ! Suc (n1 + k))" "msg"] IV(1)
        by simp
    qed
  qed
  hence "msgs (fe n ! n1) msg ≤ msgs (fe n ! n2) msg"
    by (metis Ass(1) le_add_diff_inverse order_refl)
  thus "enabled (fe n ! n2) msg" using Ass(3) enabled_def
    by (metis gr0I leD)
qed
have EnabledOrConsumed: "enabled (fe n ! (length (fe n) - 1)) msg
  ∨ (∃n0'≥n0. n0' < length (ft n) ∧ ft n ! n0' = msg)"
  using AssumptionFair(3) AssumptionFair(2)
    MessageStaysOrConsumed[of "n0" "length (fe n) - 1" "n" "msg"]
  by auto
have EnabledOrConsumedAtLast: "enabled (last (fe n)) msg ∨
  (∃ n0' . n0' ≥ n0 ∧ n0' < length (ft n) ∧ (ft n) ! n0' = msg )"
  using EnabledOrConsumed last_conv_nth AssumptionFair(2)
  by (metis length_0_conv less_nat_zero_code)
have Case2ImplThesis: "(∃ n0' . n0' ≥ n0 ∧ n0' < length (ft n)
  ∧ ft n ! n0' = msg)
  ⟹ (∃n'≥n. ∃n0'≥n0. n0' < length (ft n') ∧ msg = ft n' ! n0')"
  by auto
have Case1ImplThesis': "enabled (last (fe n)) msg
  ⟶ (∃n'≥n. ∃n0'≥ (length (ft n)). n0' < length (ft n')
      ∧ msg = ft n' ! n0')"
proof(clarify)
  assume AssumptionCase1ImplThesis': "enabled (last (fe n)) msg"
  show "∃n'≥n. ∃n0'≥length (ft n). n0' < length (ft n')
    ∧ msg = ft n' ! n0'"
  proof(rule ccontr,simp)
```

```
assume AssumptionFairContr: "∀n'≥n. ∀n0'<length (ft n').
  length (ft n) ≤ n0' ⟶ msg ≠ ft n' ! n0'"
define firstOccSet where "firstOccSet n = { msg1 . ∃ nMsg .
  ∃ n1 ≤ nMsg .
  execution.firstOccurrence (fe n) (ft n) msg1 n1
  ∧ execution.firstOccurrence (fe n) (ft n) msg nMsg }" for n
have NotEmpty: "fe n ≠ []" using  AssumptionFair(2)
  by (metis less_nat_zero_code list.size(3))
have FirstToLast':
  "∀ n . reachable ((fe n) ! 0) ((fe n) ! (length (fe n) - 1))"
  using execution.ReachableInExecution BasicProperties execution.notEmpty
  by (metis diff_less less_or_eq_imp_le not_gr0 not_one_le_zero)
hence FirstToLast: "∀ n . reachable (hd (fe n)) (last (fe n))"
  using NotEmpty hd_conv_nth last_conv_nth AssumptionFair(1)
  by (metis (full_types) One_nat_def length_0_conv
    not_one_le_zero)
hence InitToLast: "∀ n . initReachable (last (fe n))"
  using BasicProperties by auto
have "⋀ msg n0 . ∀ n .
  (execution.firstOccurrence (fe n) (ft n) msg n0)
  ⟶  0 < msgs (last (fe n)) msg"
  using BasicProperties execution.firstOccurrence_def
    enabled_def
  by metis
hence "∀ n . ∀ msg' ∈ (firstOccSet n) .
  0 < msgs (last (fe n)) msg'" using firstOccSet_def by blast
hence "∀ n . firstOccSet n ⊆ {msg. 0 < msgs (last (fe n)) msg}"
  by (metis (lifting, full_types) mem_Collect_eq subsetI)
hence FiniteMsgs: "∀ n . finite (firstOccSet n)"
  using FiniteMessages[OF finiteProcs finiteSends] InitToLast
  by (metis rev_finite_subset)
have FirstOccSetDecrOrConsumed: "∀ index .
  (enabled (last (fe index)) msg)
  ⟶ (firstOccSet (Suc index) ⊂ firstOccSet index
  ∧ (enabled (last (fe (Suc index))) msg)
    ∨ msg ∈ (set (drop (length (ft index)) (ft (Suc index)))))"
proof(clarify)
  fix index
  assume AssumptionFirstOccSetDecrOrConsumed:
    "enabled (last (fe index)) msg"
    "msg ∉ set (drop (length (ft index)) (ft (Suc index)))"
  have NotEmpty: "fe (Suc index) ≠ []" "fe index ≠ []"
    using BasicProperties
    by (metis AssumptionFair(1) One_nat_def list.size(3)
      not_one_le_zero)+
  have LengthStep: "length (ft (Suc index)) > length (ft index)"
    using AssumptionFair(1)
    by (metis PrefixListMonotonicity)
  have IPrefixList:
    "∀ i::nat . prefixList (ft i) (ft (Suc i))"
    using AssumptionFair(1) by auto
  have IPrefixListEx:
```

```
  "∀ i::nat . prefixList (fe i) (fe (Suc i))"
  using AssumptionFair(1) by auto
have LastOfIndex:
  "(fe (Suc index) ! (length (fe index) - Suc 0))
  = (last (fe index))"
  using PrefixSameOnLow[of "fe index" "fe (Suc index)"]
    IPrefixListEx[rule_format, of index]
    NotEmpty LengthStep
  by (auto simp add: last_conv_nth)
have NotConsumedIntermediate:
  "∀ i::nat < length (ft (Suc index)) .
    (i ≥ length (ft index)
    ⟶ ft (Suc index) ! i ≠ msg)"
  using AssumptionFirstOccSetDecrOrConsumed(2) ListLenDrop
  by auto
hence
  "¬(∃i. i < length (ft (Suc index)) ∧ i ≥ length (ft index)
  ∧ msg = (ft (Suc index)) ! i)"
  using execution.length BasicProperties
  by auto
hence "¬(∃i. i < length (fe (Suc index)) - 1
  ∧ i ≥ length (fe index) - 1
  ∧ msg = (ft (Suc index)) ! i)"
  using BasicProperties[rule_format, of "Suc index"]
    BasicProperties[rule_format, of "index"]
    execution.length[of trans sends start]
  by auto
hence EnabledIntermediate:
  "∀ i < length (fe (Suc index)) . (i ≥ length (fe index) - 1
    ⟶ enabled (fe (Suc index) ! i) msg)"
  using BasicProperties[rule_format, of "Suc index"]
    BasicProperties[rule_format, of "index"]
    execution.StaysEnabled[of trans sends start
    "fe (Suc index)" "ft (Suc index)" "last (fe index)" msg
    "length (fe index) - 1 "]
    AssumptionFirstOccSetDecrOrConsumed(1)
  by (auto, metis AssumptionFair(1) LastOfIndex
    MessageStaysOrConsumed)
have "length (fe (Suc index)) - 1 ≥ length (fe index) - 1"
  using PrefixListMonotonicity NotEmpty BasicProperties
  by (metis AssumptionFair(1) diff_le_mono less_imp_le)
hence "enabled (fe (Suc index)
  ! (length (fe (Suc index)) - 1)) msg"
  using EnabledIntermediate NotEmpty(1)
  by (metis diff_less length_greater_0_conv zero_less_one)
hence EnabledInSuc: "enabled (last (fe (Suc index))) msg"
  using NotEmpty last_conv_nth[of "fe (Suc index)"] by simp
have IndexIsExec:
  "execution trans sends start (fe index) (ft index)"
  using BasicProperties by blast
have SucIndexIsExec:
  "execution trans sends start (fe (Suc index))
```

```
    (ft (Suc index))"
  using BasicProperties by blast
have SameCfgOnLow: "∀ i < length (fe index) . (fe index) ! i
  = (fe (Suc index)) ! i"
  using BasicProperties PrefixSameOnLow by auto
have SameMsgOnLow: "∀ i < length (ft index) . (ft index) ! i
  = (ft (Suc index)) ! i"
  using BasicProperties PrefixSameOnLow by auto
have SmallIndex: "⋀ nMsg . execution.firstOccurrence
  (fe (Suc index)) (ft (Suc index)) msg nMsg
  ⟹ nMsg < length (fe index)"
proof(-)
  fix nMsg
  assume "execution.firstOccurrence (fe (Suc index))
    (ft (Suc index)) msg nMsg"
  hence AssumptionSubset3:
    "∃p. isReceiverOf p msg"
      "enabled (last (fe (Suc index))) msg"
      "nMsg < length (fe (Suc index))"
      "enabled (fe (Suc index) ! nMsg) msg"
      "∀n'≥nMsg. n' < length (ft (Suc index))
      ⟶ msg ≠ ft (Suc index) ! n'"
      "nMsg ≠ 0 ⟶ ¬ enabled (fe (Suc index) ! (nMsg - 1))
      msg ∨ msg = ft (Suc index) ! (nMsg - 1)"
    using execution.firstOccurrence_def[of "trans" "sends"
      "start" "fe (Suc index)" "ft (Suc index)" "msg" "nMsg"]
      SucIndexIsExec by auto
  show "nMsg < length (fe index)"
  proof(rule ccontr)
    assume AssumpSmallIndex: "¬ nMsg < length (fe index)"
    have "fe index ≠ []" using BasicProperties
      AssumptionFair(1)
      by (metis One_nat_def list.size(3) not_one_le_zero)
    hence "length (fe index) > 0"
      by (metis length_greater_0_conv)
    hence nMsgNotZero: "nMsg ≠ 0"
      using AssumpSmallIndex by metis
    hence SucCases: "¬ enabled ((fe (Suc index)) ! (nMsg - 1))
      msg ∨ msg = (ft (Suc index)) ! (nMsg - 1)"
      using AssumptionSubset3(6) by blast
    have Cond1: "nMsg - 1 ≥ length (fe index) - 1"
      using AssumpSmallIndex by (metis diff_le_mono leI)
    hence Enabled: "enabled (fe (Suc index) ! (nMsg - 1)) msg"
      using EnabledIntermediate AssumptionSubset3(3)
      by (metis less_imp_diff_less)
    have Cond2: "nMsg - 1 ≥ length (ft index) ∧ nMsg - 1
      < length (ft (Suc index))"
      using Cond1 execution.length[of "trans" "sends" "start"
        "fe index" "ft index"]
        IndexIsExec AssumptionSubset3(3)
        by (simp, metis AssumptionFair(1) One_nat_def Suc_diff_1
          Suc_eq_plus1 less_diff_conv nMsgNotZero neq0_conv)
```

```
    hence NotConsumed: "ft (Suc index) ! (nMsg - 1) ≠ msg"
      using NotConsumedIntermediate by simp
    show False using SucCases Enabled NotConsumed
    by blast
  qed
qed
have Subset: "⋀ msgInSet . msgInSet ∈ firstOccSet (Suc index)
  ⟹ msgInSet ∈ firstOccSet index"
unfolding firstOccSet_def
proof(auto)
  fix msgInSet nMsg n1
  assume AssumptionSubset: "n1 ≤ nMsg"
    "execution.firstOccurrence (fe (Suc index))
      (ft (Suc index)) msgInSet n1"
    "execution.firstOccurrence (fe (Suc index))
      (ft (Suc index)) msg nMsg"
  have AssumptionSubset2:
    "∃p. isReceiverOf p msgInSet"
      "enabled (last (fe (Suc index))) msgInSet"
      "n1 < length (fe (Suc index))"
      "enabled (fe (Suc index) ! n1) msgInSet"
      "∀n'≥n1. n' < length (ft (Suc index))
        ⟶ msgInSet ≠ ft (Suc index) ! n'"
      "n1 ≠ 0 ⟶ ¬ enabled (fe (Suc index) ! (n1 - 1))
        msgInSet ∨ msgInSet = ft (Suc index) ! (n1 - 1)"
    using execution.firstOccurrence_def[of "trans" "sends"
      "start" "fe (Suc index)" "ft (Suc index)" "msgInSet"
      "n1"] AssumptionSubset(2) SucIndexIsExec by auto
  have AssumptionSubset3:
    "∃p. isReceiverOf p msg"
      "enabled (last (fe (Suc index))) msg"
      "nMsg < length (fe (Suc index))"
      "enabled (fe (Suc index) ! nMsg) msg"
      "∀n'≥nMsg. n' < length (ft (Suc index))
        ⟶ msg ≠ ft (Suc index) ! n'"
      "nMsg ≠ 0 ⟶ ¬ enabled (fe (Suc index) ! (nMsg - 1))
        msg ∨ msg = ft (Suc index) ! (nMsg - 1)"
    using execution.firstOccurrence_def[of "trans" "sends"
      "start" "fe (Suc index)" "ft (Suc index)" "msg" "nMsg"]
      AssumptionSubset(3) SucIndexIsExec by auto
  have ShorterTrace: "length (ft index)
                      < length (ft (Suc index))"
    using PrefixListMonotonicity BasicProperties by auto

  have FirstOccurrenceMsg: "execution.firstOccurrence
    (fe index) (ft index) msg nMsg"
  proof-
    have Occ1: "∃ p . isReceiverOf p msg"
      using AssumptionSubset3(1) by blast
    have Occ2: "enabled (last (fe index)) msg"
      using AssumptionFirstOccSetDecrOrConsumed by blast
```

```
    have "(fe index) ! nMsg = (fe (Suc index)) ! nMsg"
      using SmallIndex AssumptionSubset(3)
        PrefixSameOnLow[of "fe index" "fe (Suc index)"]
        BasicProperties
      by simp
    hence Occ4: "enabled ((fe index) ! nMsg) msg"
      using AssumptionSubset3(4) by simp
    have OccSameMsg: "∀ n' ≥ nMsg . n' < length (ft index)
      ⟶ (ft index) ! n' = (ft (Suc index)) ! n'"
      using PrefixSameOnLow BasicProperties by auto
    hence Occ5: "∀ n' ≥ nMsg . n' < length (ft index)
      ⟶ msg ≠ ((ft index) ! n')"
      using AssumptionSubset3(5) ShorterTrace by simp

    have Occ6: "nMsg ≠ 0 ⟶ (¬ enabled ((fe index) !
      (nMsg - 1)) msg ∨ msg = (ft index ) ! (nMsg - 1))"
    proof(clarify)
      assume AssumpOcc6: "0 < nMsg" "msg ≠ ft index !
        (nMsg - 1)" "enabled (fe index ! (nMsg - 1)) msg"
      have "nMsg - (Suc 0) < length (fe index) - (Suc 0)"
        using SmallIndex AssumptionSubset(3) AssumpOcc6(1)
        by (metis Suc_le_eq diff_less_mono)
      hence SmallIndexTrace: "nMsg - 1 < length (ft index)"
        using IndexIsExec execution.length
        by (metis One_nat_def)
      have "¬ enabled (fe (Suc index) ! (nMsg - 1)) msg
        ∨ msg = ft (Suc index) ! (nMsg - 1)"
        using AssumptionSubset3(6) AssumpOcc6(1) by blast
      moreover have "fe (Suc index) ! (nMsg - 1)
        = fe index ! (nMsg - 1)"
        using SameCfgOnLow SmallIndex AssumptionSubset(3)
        by (metis less_imp_diff_less)
      moreover have "ft (Suc index) ! (nMsg - 1)
        = ft index ! (nMsg - 1)"
        using SameMsgOnLow SmallIndexTrace by metis
      ultimately have "¬ enabled (fe index ! (nMsg - 1)) msg
        ∨ msg = ft index ! (nMsg - 1)"
        by simp
      thus False using AssumpOcc6 by blast
    qed

    show ?thesis using IndexIsExec Occ1 Occ2 SmallIndex
      AssumptionSubset(3) Occ4 Occ5 Occ6
      execution.firstOccurrence_def[of "trans" "sends" "start"
        "fe index" "ft index"]
      by simp
  qed

  have "execution.firstOccurrence (fe index) (ft index)
    msgInSet n1"
    using AssumptionSubset2 AssumptionSubset(1)
  proof-
```

```
have Occ1': "∃p. isReceiverOf p msgInSet"
  using AssumptionSubset2(1) by blast
have Occ3': "n1 < length (fe index)"
  using SmallIndex AssumptionSubset(3) AssumptionSubset(1)
  by (metis le_less_trans)
have "(fe index) ! n1 = (fe (Suc index)) ! n1"
  using Occ3' PrefixSameOnLow[of "fe index"
    "fe (Suc index)"] BasicProperties by simp
hence Occ4': "enabled (fe index ! n1) msgInSet"
  using AssumptionSubset2(4) by simp
have OccSameMsg': "∀ n' ≥ n1 . n' < length (ft index)
  ⟶ (ft index) ! n' = (ft (Suc index)) ! n'"
  using PrefixSameOnLow BasicProperties by auto
hence Occ5': "∀n' ≥ n1. n' < length (ft index)
  ⟶ msgInSet ≠ ft index ! n'"
  using AssumptionSubset2(5) ShorterTrace by simp
have "length (fe index) > 0" using NotEmpty(2)
  by (metis length_greater_0_conv)
hence "length (fe index) - 1 < length (fe index)"
  by (metis One_nat_def diff_Suc_less)
hence
  "enabled (fe index ! (length (fe index) - 1)) msgInSet
  ∨ (∃n0'≥n1. n0' < length (ft index) ∧ ft index ! n0'
    = msgInSet)"
  using Occ4' Occ3' MessageStaysOrConsumed[of "n1"
    "length (fe index) - 1" "index" "msgInSet"]
  by (metis Suc_pred' ‹0 < length (fe index)›
    not_le not_less_eq_eq)
hence "enabled ((fe index) ! (length (fe index) - 1))
  msgInSet"
  using Occ5' by auto
hence Occ2': "enabled (last (fe index)) msgInSet"
  using last_conv_nth[of "fe index"] NotEmpty(2) by simp

have Occ6': "n1 ≠ 0 ⟶ ¬ enabled (fe index ! (n1 - 1))
  msgInSet ∨ msgInSet = ft index ! (n1 - 1)"
proof(clarify)
  assume AssumpOcc6': "0 < n1" "msgInSet ≠ ft index !
    (n1 - 1)" "enabled (fe index ! (n1 - 1)) msgInSet"
  have "n1 - (Suc 0) < length (fe index) - (Suc 0)"
    using Occ3' AssumpOcc6'(1)
    by (metis Suc_le_eq diff_less_mono)
  hence SmallIndexTrace': "n1 - 1 < length (ft index)"
    using IndexIsExec execution.length
    by (metis One_nat_def)
  have "¬ enabled (fe (Suc index) ! (n1 - 1)) msgInSet
    ∨ msgInSet = ft (Suc index) ! (n1 - 1)"
    using AssumptionSubset2(6) AssumpOcc6'(1) by blast
  moreover have "fe (Suc index) ! (n1 - 1)
    = fe index ! (n1 - 1)"
    using SameCfgOnLow Occ3' by (metis less_imp_diff_less)
  moreover have "ft (Suc index) ! (n1 - 1)
```

```
        = ft index ! (n1 - 1)"
        using SameMsgOnLow SmallIndexTrace' by metis
      ultimately have "¬ enabled (fe index !
        (n1 - 1)) msgInSet ∨ msgInSet = ft index ! (n1 - 1)"
        by simp
      thus False using AssumpOcc6' by blast
    qed

    show ?thesis using IndexIsExec Occ1' Occ2' Occ3' Occ4'
      Occ5' Occ6'
      execution.firstOccurrence_def[of "trans" "sends"
        "start" "fe index" "ft index"]
      by simp
  qed

  thus "∃nMsg' n1'. n1' ≤ nMsg'
    ∧ execution.firstOccurrence (fe index) (ft index)
      msgInSet n1'
    ∧ execution.firstOccurrence (fe index) (ft index)
      msg nMsg'"
    using FirstOccurrenceMsg AssumptionSubset(1) by blast
qed

have ProperSubset: "∃ msg' .msg' ∈ firstOccSet index
  ∧ msg' ∉ firstOccSet (Suc index)"
proof-
  have "initial (hd (fe index))" using AssumptionFair(1)
    by blast
  hence "∃msg'. execution.minimalEnabled (fe index) (ft index)
    msg' ∧  msg' ∈ set (drop (length (ft index))
      (fStepMsg (fe index) (ft index)))"
    using FStep fe_def ft_def
      BasicProperties by simp
  then obtain consumedMsg where ConsumedMsg:
    "execution.minimalEnabled (fe index) (ft index)
      consumedMsg"
    "consumedMsg ∈ set (drop (length (ft index))
      (fStepMsg (fe index) (ft index)))" by blast
  hence ConsumedIsInDrop:
    "consumedMsg ∈ set (drop (length (ft index)) (ft (Suc index)))"
    using fe_def ft_def FStep
      BasicProperties[rule_format, of index]
    by auto

  have MinImplAllBigger: "⋀ msg' . execution.minimalEnabled
  (fe index) (ft index) msg'
   ⟶ (∃ OccM' . (execution.firstOccurrence (fe index)
    (ft index) msg' OccM' )
      ∧ (∀ msg . ∀ OccM . execution.firstOccurrence (fe index)
        (ft index) msg OccM
      ⟶ OccM' ≤ OccM))"
  proof(auto)
```

```
fix msg'
assume AssumpMinImplAllBigger: "execution.minimalEnabled
  (fe index) (ft index) msg'"
have IsExecIndex: "execution trans sends start
  (fe index) (ft index)"
  using  BasicProperties[rule_format, of index] by simp
have "(∃ p . isReceiverOf p msg') ∧
  (enabled (last (fe index)) msg')
  ∧ (∃ n .  n < length (fe index)
    ∧ enabled ( (fe index) ! n) msg'
    ∧ (∀ n' ≥ n . n' < length (ft index)
    ⟶ msg' ≠ ((ft index)! n'))
    ∧ (∀ n' msg' . ((∃ p . isReceiverOf p msg')
      ∧ (enabled (last (fe index)) msg')
      ∧ n' < length (ft index)
      ∧ enabled ((fe index)! n') msg'
      ∧ (∀ n'' ≥ n' . n'' < length (ft index)
      ⟶ msg' ≠ ((ft index) ! n''))) ⟶ n' ≥ n))"
using execution.minimalEnabled_def[of trans sends start
  "(fe index)" "(ft index)" msg']
  AssumpMinImplAllBigger IsExecIndex by auto
then obtain OccM' where OccM':
  "(∃ p . isReceiverOf p msg')"
  "(enabled (last (fe index)) msg')"
  "OccM' < length (fe index)"
  "enabled ( (fe index) ! OccM') msg'"
  "(∀ n' ≥ OccM' . n' < length (ft index)
  ⟶ msg' ≠ ((ft index)! n'))"
  "(∀ n' msg' . ((∃ p . isReceiverOf p msg')
    ∧ (enabled (last (fe index)) msg')
    ∧ n' < length (ft index)
    ∧ enabled ((fe index)! n') msg'
    ∧ (∀ n'' ≥ n' . n'' < length (ft index)
    ⟶ msg' ≠ ((ft index) ! n''))) ⟶ n' ≥ OccM')"
  by blast
have "0 < OccM' ⟹ enabled (fe index ! (OccM' - Suc 0)) msg'
        ⟹ msg' ≠ ft index ! (OccM' - Suc 0) ⟹ False"
proof(-)
  fix p
  assume AssumpContr:
    "0 < OccM'"
    "enabled (fe index ! (OccM' - Suc 0)) msg'"
    "msg' ≠ ft index ! (OccM' - Suc 0)"
  have LengthOccM': "(OccM' - 1) < length (ft index)"
  using OccM'(3) IndexIsExec AssumpContr(1)
    AssumptionFair(1)
    by (metis  One_nat_def Suc_diff_1 Suc_eq_plus1_left
      Suc_less_eq le_add_diff_inverse)
  have BiggerIndices: "(∀n''≥(OccM' - 1).
    n'' < length (ft index) ⟶ msg' ≠ ft index ! n'')"
    using OccM'(5) by (metis AssumpContr(3) One_nat_def
        Suc_eq_plus1 diff_Suc_1 le_SucE le_diff_conv)
```

```
  have "(∃p. isReceiverOf p msg') ∧ enabled (last
    (fe index)) msg' ∧ (OccM' - 1) < length (ft index)
      ∧ enabled (fe index ! (OccM' - 1)) msg'
      ∧ (∀n''≥(OccM' - 1). n'' < length (ft index)
        ⟶ msg' ≠ ft index ! n'')"
    using OccM' LengthOccM' AssumpContr BiggerIndices
    by simp
  hence "OccM' ≤ OccM' - 1" using OccM'(6) by blast
  thus False using AssumpContr(1) diff_less leD zero_less_one by blast
qed
hence FirstOccMsg': "execution.firstOccurrence (fe index)
    (ft index) msg' OccM'"
  unfolding execution_def
    execution.firstOccurrence_def[OF IsExecIndex, of msg' OccM']
  by (auto simp add: OccM'(1,2,3,4,5))
have "∀msg OccM. execution.firstOccurrence (fe index)
  (ft index) msg OccM ⟶ OccM' ≤ OccM"
proof clarify
  fix msg OccM
  assume  "execution.firstOccurrence (fe index)
    (ft index) msg OccM"
  hence AssumpOccMFirstOccurrence:
    "∃ p . isReceiverOf p msg"
    "enabled (last (fe index)) msg"
    "OccM < (length (fe index))"
    "enabled ((fe index) ! OccM) msg"
    "(∀ n' ≥ OccM . n' < length (ft index)
    ⟶ msg ≠ ((ft index) ! n'))"
    "(OccM ≠ 0 ⟶ (¬ enabled ((fe index) ! (OccM - 1))
    msg ∨ msg = (ft index)!(OccM - 1)))"
    by (auto simp add: execution.firstOccurrence_def[of
        trans sends start "(fe index)" "(ft index)"
        msg OccM] IsExecIndex)
  hence "(∃p. isReceiverOf p msg) ∧
    enabled (last (fe index)) msg ∧
    enabled (fe index ! OccM) msg ∧
    (∀n''≥ OccM. n'' < length (ft index)
      ⟶ msg ≠ ft index ! n'')"
   by simp
  thus "OccM' ≤ OccM" using OccM'
  proof(cases "OccM < length (ft index)",auto)
    assume "¬ OccM < length (ft index)"
    hence "OccM ≥ length (fe index) - 1"
      using AssumptionFair(1) by (metis One_nat_def leI)
    hence "OccM = length (fe index) - 1"
      using AssumpOccMFirstOccurrence(3) by simp
    thus "OccM' ≤ OccM" using OccM'(3) by simp
  qed
qed
with FirstOccMsg' show "∃OccM'.
  execution.firstOccurrence (fe index) (ft index)
    msg' OccM'
```

```
    ∧ (∀msg OccM. execution.firstOccurrence (fe index)
      (ft index) msg OccM ⟶ OccM' ≤ OccM)" by blast
qed

have MinImplFirstOcc: "⋀ msg' . execution.minimalEnabled
  (fe index) (ft index) msg'
  ⟹ msg' ∈ firstOccSet index"
proof -
  fix msg'
  assume AssumpMinImplFirstOcc:
    "execution.minimalEnabled (fe index) (ft index) msg'"
  then obtain OccM' where OccM':
    "execution.firstOccurrence (fe index) (ft index)
      msg' OccM'"
    "∀ msg . ∀ OccM . execution.firstOccurrence
    (fe index) (ft index) msg OccM
  ⟶ OccM' ≤ OccM" using MinImplAllBigger by blast
  thus "msg' ∈ firstOccSet index" using OccM'
  proof (auto simp add: firstOccSet_def)
    have "enabled (last (fe index)) msg"
      using AssumptionFirstOccSetDecrOrConsumed(1) by blast
    hence "∃nMsg .  execution.firstOccurrence (fe index)
      (ft index) msg nMsg"
      using execution.FirstOccurrenceExists IndexIsExec
      AssumptionFair(4) by blast
    then obtain nMsg where NMsg: "execution.firstOccurrence
      (fe index) (ft index) msg nMsg" by blast
    hence "OccM' ≤ nMsg" using OccM' by simp
    hence "∃nMsg . OccM' ≤ nMsg ∧
      execution.firstOccurrence (fe index) (ft index) msg'
        OccM' ∧
      execution.firstOccurrence (fe index) (ft index) msg
        nMsg"
      using OccM'(1) NMsg by blast
    thus "∃nMsg n1 . n1 ≤ nMsg ∧
      execution.firstOccurrence (fe index) (ft index)
        msg' n1 ∧
      execution.firstOccurrence (fe index) (ft index)
        msg nMsg" by blast
  qed
qed
hence ConsumedInSet: "consumedMsg ∈ firstOccSet index"
  using ConsumedMsg by simp
have GreaterOccurrence: "⋀ nMsg n1 .
    execution.firstOccurrence (fe (Suc index))
      (ft (Suc index)) consumedMsg n1 ∧
    execution.firstOccurrence (fe (Suc index))
      (ft (Suc index)) msg nMsg
    ⟹ nMsg < n1"
proof(rule ccontr,auto)
  fix nMsg n1
  assume AssumpGreaterOccurrence: "¬ nMsg < n1"
```

```
        "execution.firstOccurrence (fe (Suc index))
          (ft (Suc index)) consumedMsg n1"
        "execution.firstOccurrence (fe (Suc index))
          (ft (Suc index)) msg nMsg"
      have "nMsg < length (fe index)"
        using SmallIndex AssumpGreaterOccurrence(3) by simp
      hence "n1 < length (fe index)"
        using AssumpGreaterOccurrence(1)
        by (metis less_trans nat_neq_iff)
      hence N1Small: "n1 ≤ length (ft index)"
        using IndexIsExec AssumptionFair(1)
        by (metis  One_nat_def Suc_eq_plus1 le_diff_conv2
          not_le not_less_eq_eq)
      have NotConsumed: "∀ i ≥ n1 . i < length (ft (Suc index))
        ⟶ consumedMsg ≠ (ft (Suc index)) ! i"
        using execution.firstOccurrence_def[of "trans" "sends"
          "start" "fe (Suc index)" "ft (Suc index)"
          "consumedMsg" "n1"]
          AssumpGreaterOccurrence(2) SucIndexIsExec by auto
      have "∃ i ≥ length (ft index) .
        i < length (ft (Suc index))
        ∧ consumedMsg = (ft (Suc index)) ! i"
        using DropToIndex[of "consumedMsg" "length (ft index)"]
        ConsumedIsInDrop by simp
      then obtain i where IDef: "i ≥ length (ft index)"
        "i < length (ft (Suc index))"
        "consumedMsg = (ft (Suc index)) ! i" by blast
      thus False using NotConsumed N1Small by simp
    qed
    have "consumedMsg ∉ firstOccSet (Suc index)"
    proof(clarify)
      assume AssumpConsumedInSucSet:
        "consumedMsg ∈ firstOccSet (Suc index)"
      hence "∃nMsg n1. n1 ≤ nMsg ∧
          execution.firstOccurrence (fe (Suc index))
            (ft (Suc index)) consumedMsg n1 ∧
          execution.firstOccurrence (fe (Suc index))
            (ft (Suc index)) msg nMsg"
        using firstOccSet_def by blast
      thus False using GreaterOccurrence
        by (metis less_le_trans less_not_refl3)
    qed
    thus ?thesis using ConsumedInSet by blast
  qed

  hence "firstOccSet (Suc index) ⊂ firstOccSet index"
    using Subset by blast
  thus "firstOccSet (Suc index) ⊂ firstOccSet index
    ∧ enabled (last (fe (Suc index))) msg"
    using EnabledInSuc by blast
qed
```

```
have NotConsumed: "∀ index ≥ n . ¬ msg ∈
  (set (drop (length (ft index)) (ft (Suc index))))"
proof(clarify)
  fix index
  assume AssumpMsgNotConsumed: "n ≤ index"
    "msg ∈ set (drop (length (ft index)) (ft (Suc index)))"

  have "∃ n0' ≥ length (ft index) .
    n0' < length (ft (Suc index))
    ∧ msg = (ft (Suc index)) ! n0'"
    using AssumpMsgNotConsumed(2) DropToIndex[of "msg"
      "length (ft index)" "ft (Suc index)"] by auto
  then obtain n0' where MessageIndex: "n0' ≥ length (ft index)"
    "n0' < length (ft (Suc index))"
    "msg = (ft (Suc index)) ! n0'" by blast
  have LengthIncreasing: "length (ft n) ≤ length (ft index)"
    using AssumpMsgNotConsumed(1)
  proof(induct index,auto)
    fix indexa
    assume AssumpLengthIncreasing:
      "n ≤ indexa ⟹ length (ft n) ≤ length (ft indexa)"
      "n ≤ Suc indexa" "n ≤ index"
    show "length (ft n) ≤ length (ft (Suc indexa))"
    proof(cases "n = Suc indexa",auto)
      assume "n ≠ Suc indexa"
      hence "n ≤ indexa" using AssumpLengthIncreasing(2)
        by (metis le_SucE)
      hence LengthNA: "length (ft n) ≤ length (ft indexa)"
        using AssumpLengthIncreasing(1) by blast
      have PrefixIndexA: "prefixList (ft indexa) (ft (Suc indexa))"
        using BasicProperties by simp
      show "length (ft n) ≤ length (ft (Suc indexa))"
        using LengthNA PrefixListMonotonicity[OF PrefixIndexA]
        by (metis (opaque_lifting, no_types) antisym le_cases
          less_imp_le less_le_trans)
    qed
  qed
  thus False using AssumptionFairContr MessageIndex
    AssumpMsgNotConsumed(1)
    by (metis ‹length (ft index) ≤ n0'› le_SucI le_trans)
qed

hence FirstOccSetDecrImpl:
  "∀ index ≥ n . (enabled (last (fe index)) msg)
  ⟶ firstOccSet (Suc index) ⊂ firstOccSet index
    ∧ (enabled (last (fe (Suc index))) msg)"
  using FirstOccSetDecrOrConsumed by blast
hence FirstOccSetDecrImpl: "∀ index ≥ n . firstOccSet
  (Suc index) ⊂ firstOccSet index"
  using KeepProperty[of "n" "λx.(enabled (last (fe x)) msg)"
    "λx.(firstOccSet (Suc x) ⊂ firstOccSet x)"]
    AssumptionCase1ImplThesis' by blast
```

```
      hence FirstOccSetDecr': "∀ index ≥ n .
        card (firstOccSet (Suc index)) < card (firstOccSet index)"
        using FiniteMsgs psubset_card_mono by metis
      hence "card (firstOccSet (n + (card (firstOccSet n) + 1)))
        ≤ card (firstOccSet n) - (card (firstOccSet n) + 1)"
        using SmallerMultipleStepsWithLimit[of "n"
          "λx. card (firstOccSet x)" "card (firstOccSet n) + 1"]
        by blast
      hence IsNegative:"card (firstOccSet (n + (card
        (firstOccSet n) + 1))) < 0"
        by (metis FirstOccSetDecr' diff_add_zero leD le_add1
          less_nat_zero_code neq0_conv)
      thus False by (metis less_nat_zero_code)
    qed
  qed

  hence Case1ImplThesis: "enabled (last (fe n)) msg
    ⟹ (∃n'≥n. ∃n0'≥n0. n0' < length (ft n') ∧ msg = ft n' ! n0')"
    using AssumptionFair(2) execution.length[of trans sends start
      "fe n" "ft n"] BasicProperties
    by (metis One_nat_def Suc_eq_plus1 Suc_lessI leI le_less_trans
      less_asym less_diff_conv)

  show "∃n'≥n. ∃n0'≥n0. n0' < length (ft n') ∧ msg = ft n' ! n0'"
  using disjE[OF EnabledOrConsumedAtLast Case1ImplThesis Case2ImplThesis] .
qed
show ?thesis proof (rule exI[of _ fe], rule exI[of _ ft])
  show "fe 0 = [cfg] ∧ fairInfiniteExecution fe ft
    ∧ (∀n. nonUniform (last (fe n)) ∧ prefixList (fe n) (fe (n + 1))
        ∧ prefixList (ft n) (ft (n + 1))
        ∧ execution trans sends start (fe n) (ft n))"
  using Fair fe_def FStep BasicProperties by auto
qed
qed
```

6.3 Contradiction

An infinite execution is said to be a terminating FLP execution if each process at some point sends a decision message or if it stops, which is expressed by the process not processing any further messages.

```
definition (in flpSystem) terminationFLP::
  "(nat ⇒ ('p, 'v, 's) configuration list)
  ⇒ (nat ⇒ ('p, 'v) message list) ⇒ bool"
where
  "terminationFLP fe ft ≡ infiniteExecution fe ft ⟶
  (∀ p . ∃ n .
     (∃ i0 < length (ft n). ∃ b .
      (<⊥, outM b> ∈# sends p (states ((fe n) ! i0) p) (unpackMessage ((ft n) !
i0)))
      ∧ isReceiverOf p ((ft n) ! i0))
  ∨ (∀ n1 > n . ∀ m ∈ set (drop (length (ft n)) (ft n1)) . ¬ isReceiverOf p m))"
```

```
theorem ConsensusFails:
assumes
  Termination:
    "⋀ fe ft . (fairInfiniteExecution fe ft ⟹ terminationFLP fe ft)" and
  Validity: "∀ i c . validity i c" and
  Agreement: "∀ i c . agreementInit i c"
shows
  "False"
proof -
  obtain cfg where Cfg: "initial cfg" "nonUniform cfg"
    using InitialNonUniformCfg[OF PseudoTermination Validity Agreement]
        by blast
  obtain fe:: "nat ⇒ ('p, 'v, 's) configuration list" and
              ft:: "nat ⇒ ('p, 'v) message list"
    where FE: "(fe 0) = [cfg]" "fairInfiniteExecution fe ft"
        "(∀(n::nat) . nonUniform (last (fe n))
          ∧ prefixList (fe n) (fe (n+1))
          ∧ prefixList (ft n) (ft (n+1))
          ∧ (execution trans sends start (fe n) (ft n)))"
    using FairNonUniformExecution[OF Cfg]
    by blast

  have AllArePrefixesExec: "∀ m . ∀ n > m . prefixList (fe m) (fe n)"
  proof(clarify)
    fix m::nat and n::nat
    assume MLessN: "m < n"
    have "prefixList (fe m) (fe n)" using MLessN
    proof(induct n, simp)
      fix n
      assume IA: "(m < n) ⟹ (prefixList (fe m) (fe n))" "m < (Suc n)"
      have "m = n ∨ m < n" using IA(2) by (metis less_SucE)
      thus "prefixList (fe m) (fe (Suc n))"
      proof(cases "m = n", auto)
        show "prefixList (fe n) (fe (Suc n))" using FE by simp
      next
        assume "m < n"
        hence IA2: "prefixList (fe m) (fe n)" using IA(1) by simp
        have "prefixList (fe n) (fe (n+1))" using FE by simp
        thus "prefixList (fe m) (fe (Suc n))" using PrefixListTransitive
          IA2 by simp
      qed
    qed
    thus "prefixList (fe m) (fe n)" by simp
  qed

  have AllArePrefixesTrace: "∀ m . ∀ n > m . prefixList (ft m) (ft n)"
  proof(clarify)
    fix m::nat and n::nat
    assume MLessN: "m < n"
    have "prefixList (ft m) (ft n)" using MLessN
    proof(induct n, simp)
      fix n
```

```
    assume IA: "(m < n) ⟹ (prefixList (ft m) (ft n))" "m < (Suc n)"
    have "m = n ∨ m < n" using IA(2) by (metis less_SucE)
    thus "prefixList (ft m) (ft (Suc n))"
    proof(cases "m = n", auto)
      show "prefixList (ft n) (ft (Suc n))" using FE by simp
    next
      assume "m < n"
      hence IA2: "prefixList (ft m) (ft n)" using IA(1) by simp
      have "prefixList (ft n) (ft (n+1))" using FE by simp
      thus "prefixList (ft m) (ft (Suc n))" using PrefixListTransitive
        IA2 by simp
    qed
  qed
  thus "prefixList (ft m) (ft n)" by simp
qed

have Length: "∀ n . length (fe n) ≥ n + 1"
proof(clarify)
  fix n
  show "length (fe n) ≥ n + 1"
  proof(induct n, simp add: FE(1))
    fix n
    assume IH: "(n + (1::nat)) ≤ (length (fe n))"
    have "length (fe (n+1)) ≥ length (fe n) + 1" using FE(3)
      PrefixListMonotonicity
      by (metis Suc_eq_plus1 Suc_le_eq)
    thus "(Suc n) + (1::nat) ≤ (length (fe (Suc n)))" using IH by auto
  qed
qed

have AllExecsFromInit: "∀ n . ∀ n0 < length (fe n) .
  reachable cfg ((fe n) ! n0)"
proof(clarify)
  fix n::nat and n0::nat
  assume "n0 < length (fe n)"
  thus "reachable cfg ((fe n) ! n0)"
  proof(cases "0 = n", auto)
    assume N0Less: "n0 < length (fe 0)"
    have NoStep: "reachable cfg cfg" using reachable.simps by blast
    have "length (fe 0) = 1" using FE(1) by simp
    hence N0Zero: "n0 = 0" using N0Less FE by simp
    hence "(fe 0) ! n0 = cfg" using FE(1) by simp
    thus "reachable cfg ((fe 0) ! n0)" using FE(1) NoStep N0Zero by simp
  next
    assume NNotZero: "0 < n" "n0 < (length (fe n))"
    have ZeroCfg: "(fe 0) = [cfg]" using FE by simp
    have "prefixList (fe 0) (fe n)" using AllArePrefixesExec NNotZero
      by simp
    hence PrList: "prefixList [cfg] (fe n)" using ZeroCfg by simp
    have CfgFirst: "cfg = (fe n) ! 0"
      using prefixList.cases[OF PrList]
      by (metis (full_types) ZeroCfg list.distinct(1) nth_Cons_0)
```

```
    have "reachable ((fe n) ! 0) ((fe n) ! n0)"
      using execution.ReachableInExecution FE NNotZero(2) by (metis le0)
    thus "(reachable cfg ((fe n) ! n0))" using assms CfgFirst by simp
  qed
qed

have NoDecided: "(∀ n n0 v . (n0 < length (fe n))
                ⟶ ¬ vDecided v ((fe n) ! n0))"
proof(clarify)
  fix n n0 v
  assume AssmNoDecided: "n0 < length (fe n)"
    "initReachable ((fe n) ! n0)"
    "0 < (msgs ((fe n) ! n0) <⊥, outM v>)"
  have LastNonUniform: "nonUniform (last (fe n))" using FE by simp
  have LastIsLastIndex: "⋀ l . l ≠ [] ⟶ last l = l ! ((length l) - 1)"
    by (metis last_conv_nth)
  have Fou: "n0 ≤ length (fe n) - 1" using AssmNoDecided by simp
  have FeNNotEmpty:"fe n ≠ []" using FE(1) AllArePrefixesExec
    by (metis AssmNoDecided(1) less_nat_zero_code list.size(3))
  hence Fou2: "length (fe n) - 1 < length (fe n)" by simp
  have "last (fe n) = (fe n) ! (length (fe n) - 1)"
    using LastIsLastIndex FeNNotEmpty by auto
  have LastNonUniform: "nonUniform (last (fe n))" using FE by simp
  have "reachable ((fe n) ! n0) ((fe n) ! (length (fe n) - 1))"
    using FE execution.ReachableInExecution Fou Fou2 by metis
  hence NOToLast: "reachable ((fe n) ! n0) (last (fe n))"
    using LastIsLastIndex[of "fe n"] FeNNotEmpty by simp
  hence LastVDecided: "vDecided v (last (fe n))"
    using NoOutMessageLoss[of "((fe n) ! n0)" "(last (fe n))"]
      AssmNoDecided
    by (simp,
      metis LastNonUniform le_neq_implies_less less_nat_zero_code neq0_conv)

  have AllAgree: "∀ cfg' . reachable (last (fe n)) cfg'
    ⟶ agreement cfg'"
  proof(clarify)
    fix cfg'
    assume LastToNext: "reachable (last (fe n)) cfg'"
    hence "reachable cfg  ((fe n) ! (length (fe n) - 1))"
      using AllExecsFromInit AssmNoDecided(1) by auto
    hence "reachable cfg (last (fe n))" using LastIsLastIndex[of "fe n"]
      FeNNotEmpty by simp
    hence FirstToLast: "reachable cfg cfg'" using initReachable_def Cfg
      LastToNext ReachableTrans by blast
    hence "agreementInit cfg cfg'" using Agreement by simp
    hence "∀v1. (<⊥, outM v1> ∈# msgs cfg') ⟶ (∀v2. (<⊥, outM v2> ∈#
      msgs cfg') ⟷ v2 = v1)"
      using Cfg FirstToLast
      by (simp add: agreementInit_def)
    thus "agreement cfg'" by (simp add: agreement_def)
  qed
  thus "False" using NonUniformImpliesNotDecided LastNonUniform
```

```
    PseudoTermination LastVDecided by simp
qed

have Termination: "terminationFLP fe ft" using assms(1)[OF FE(2)] .

hence AllDecideOrCrash:
  "∀p. ∃n .
     (∃ i0 < length (ft n) . ∃b.
        (<⊥, outM b> ∈#
          sends p (states (fe n ! i0) p) (unpackMessage (ft n ! i0)))
        ∧ isReceiverOf p (ft n ! i0))
    ∨ (∀ n1 > n . ∀ m ∈ (set (drop (length (ft n)) (ft n1))) .
        ¬ isReceiverOf p m)"
  using FE(2)
  unfolding terminationFLP_def fairInfiniteExecution_def
  by blast

have "∀ p . ∃ n . (∀ n1 > n . ∀ m ∈ (set (drop (length (ft n)) (ft n1))) .
  ¬ isReceiverOf p m)"
proof(clarify)
  fix p
  from AllDecideOrCrash have
  "∃ n .
     (∃ i0 < length (ft n) . ∃b.
      (<⊥, outM b> ∈# sends p (states (fe n ! i0) p) (unpackMessage (ft n ! i0)))

     ∧ isReceiverOf p (ft n ! i0))
    ∨ (∀ n1 > n . ∀ m ∈ (set (drop (length (ft n)) (ft n1))).
      ¬ isReceiverOf p m)" by simp
  hence "(∃ n . ∃ i0 < length (ft n) .
       (∃b. (<⊥, outM b> ∈#
          sends p (states (fe n ! i0) p) (unpackMessage (ft n ! i0)))
        ∧ isReceiverOf p (ft n ! i0)))
      ∨ (∃ n .∀ n1 > n . ∀ m ∈ (set (drop (length (ft n)) (ft n1))) .
        ¬ isReceiverOf p m)" by blast
  thus "∃n. (∀n1>n. (∀ m ∈ (set (drop (length (ft n)) (ft n1))).
    (¬ (isReceiverOf p m))))"
  proof(elim disjE, auto)
    fix n i0 b
    assume DecidingPoint:
      "i0 < length (ft n)"
      "isReceiverOf p (ft n ! i0)"
      "<⊥, outM b> ∈# sends p (states (fe n ! i0) p) (unpackMessage (ft n ! i0))"
    have "i0 < length (fe n) - 1"
      using DecidingPoint(1)
      by (metis (no_types) FE(3) execution.length)
    hence StepN0: "((fe n) ! i0) ⊢ ((ft n) ! i0) ↦ ((fe n) ! (i0 + 1))"
      using FE by (metis execution.step)
    hence "msgs ((fe n) ! (i0 + 1)) <⊥, outM b>
      = (msgs ((fe n) ! i0) <⊥, outM b>) +
      (sends p (states ((fe n) ! i0) p)
      (unpackMessage ((ft n) ! i0)) <⊥, outM b>)"
```

```
      using DecidingPoint(2) OutOnlyGrowing[of "(fe n) ! i0" "(ft n) ! i0"
        "(fe n) ! (i0 + 1)" "p"]
      by auto
    hence "(sends p (states ((fe n) ! i0) p)
      (unpackMessage ((ft n) ! i0)) <⊥, outM b>)
      ≤ msgs ((fe n) ! (i0 + 1)) <⊥, outM b>"
      using asynchronousSystem.steps_def by auto
    hence OutMsgEx: "0 < msgs ((fe n) ! (i0 + 1)) <⊥, outM b>"
      using asynchronousSystem.steps_def DecidingPoint(3) by auto
    have "(i0 + 1) < length (fe n)"
      using DecidingPoint(1) ‹i0 < length (fe n) - 1› by auto
    hence "initReachable ((fe n) ! (i0 + 1))"
      using AllExecsFromInit Cfg(1)
      by (metis asynchronousSystem.initReachable_def)
    hence Decided: "vDecided b ((fe n) ! (i0 + 1))" using OutMsgEx
      by auto
    have "i0 + 1 < length (fe n)" using DecidingPoint(1)
      by (metis ‹(((i0::nat) + (1::nat)) < (length (
        (fe::(nat ⇒ ('p, 'v, 's) configuration list)) (n::nat))))›)
    hence "¬ vDecided b ((fe n) ! (i0 + 1))" using NoDecided by auto
    hence "False" using Decided by auto
    thus "∃n. (∀n1>n. (∀ m ∈ (set (drop (length (ft n)) (ft n1))).
      (¬ (isReceiverOf p m))))" by simp
  qed
qed
hence "∃ (crashPoint::'p ⇒ nat) .
  ∀ p . ∃  n . crashPoint p = n ∧ (∀ n1 > n . ∀ m ∈ (set (drop
  (length (ft n)) (ft n1))) . (¬ isReceiverOf p m))" by metis
then obtain crashPoint where CrashPoint:
  "∀ p . (∀ n1 > (crashPoint p) . ∀ m ∈ (set (drop (length
  (ft (crashPoint p))) (ft n1))) . (¬ isReceiverOf p m))"
  by blast
define limitSet where "limitSet = {crashPoint p | p . p ∈ Proc}"
have "finite {p. p ∈ Proc}" using finiteProcs by simp
hence "finite limitSet" using limitSet_def finite_image_set[] by blast
hence "∃ limit . ∀ l ∈ limitSet . l < limit" using
  finite_nat_set_iff_bounded by auto
hence "∃ limit . ∀ p . (crashPoint p) < limit" using limitSet_def by auto
then obtain limit where Limit: "∀ p . (crashPoint p) < limit" by blast
define lengthLimit where "lengthLimit = length (ft limit) - 1"
define lateMessage where "lateMessage = last (ft limit)"
hence "lateMessage = (ft limit) ! (length (ft limit) - 1)"
  by (metis AllArePrefixesTrace Limit last_conv_nth less_nat_zero_code
    list.size(3) PrefixListMonotonicity)
hence LateIsLast: "lateMessage = (ft limit) ! lengthLimit"
using lateMessage_def lengthLimit_def by auto

have "∃ p . isReceiverOf p lateMessage"
proof(rule ccontr)
  assume "¬ (∃(p::'p). (isReceiverOf p lateMessage))"
  hence IsOutMsg: "∃ v . lateMessage = <⊥, outM v>"
    by (metis isReceiverOf.simps(1) isReceiverOf.simps(2) message.exhaust)
```

```
    have "execution trans sends start (fe limit) (ft limit)" using FE
      by auto
    hence "length (fe limit) - 1 = length (ft limit)"
      using execution.length by simp
    hence "lengthLimit < length (fe limit) - 1"
      using lengthLimit_def
      by (metis (opaque_lifting, no_types) Length Limit One_nat_def Suc_eq_plus1
        Suc_le_eq diff_less
        diffs0_imp_equal gr_implies_not0 less_Suc0 neq0_conv)
    hence "((fe limit) ! lengthLimit) ⊢ ((ft limit) ! lengthLimit)
      ↦ ((fe limit) ! (lengthLimit + 1))"
      using FE by (metis execution.step)
    hence "((fe limit) ! lengthLimit) ⊢ lateMessage ↦ ((fe limit) !
      (lengthLimit + 1))"
      using LateIsLast by auto
    thus False using IsOutMsg steps_def by auto
  qed

  then obtain p where ReceiverOfLate: "isReceiverOf p lateMessage" by blast
  have "∀ n1 > (crashPoint p) .
    ∀ m ∈ (set (drop (length (ft (crashPoint p))) (ft n1))) .
      (¬ isReceiverOf p m)"
    using CrashPoint
    by simp
  hence NoMsgAfterLimit: "∀ m ∈ (set (drop (length (ft (crashPoint p)))
    (ft limit))) . (¬ isReceiverOf p m)"
    using Limit
    by auto
  have "lateMessage ∈ set (drop (length(ft (crashPoint p))) (ft limit))"
  proof-
    have "crashPoint p < limit" using Limit by simp
    hence "prefixList (ft (crashPoint p)) (ft limit)"
      using AllArePrefixesTrace by auto
    hence CrashShorterLimit: "length (ft (crashPoint p))
      < length (ft limit)" using PrefixListMonotonicity by auto
    hence "last (drop (length (ft (crashPoint p))) (ft limit))
      = last (ft limit)" by (metis last_drop)
    hence "lateMessage = last (drop (length (ft (crashPoint p)))
      (ft limit))" using lateMessage_def by auto
    thus "lateMessage ∈ set (drop (length(ft (crashPoint p))) (ft limit))"
      by (metis CrashShorterLimit drop_eq_Nil last_in_set not_le)
  qed

  hence "¬ isReceiverOf p lateMessage" using NoMsgAfterLimit by auto
  thus "False" using ReceiverOfLate by simp
qed

end

end
```

7 An Existing FLPSystem

```
theory FLPExistingSystem
imports FLPTheorem
begin
```

We define an example FLPSystem with some example execution to show that the locales employed are not void. (If they were, the consensus impossibility result would be trivial.)

7.1 System definition

```
datatype proc = p0 | p1
datatype state = s0 | s1
datatype val = v0 | v1

primrec trans :: "proc ⇒ state ⇒ val messageValue ⇒ state"
where
  "trans p s0 v = s1"
| "trans p s1 v = s0"

primrec sends ::
  "proc ⇒ state ⇒ val messageValue ⇒ (proc, val) message multiset"
where
  "sends p s0 v = {# <p0, v1> }"
| "sends p s1 v = {# <p1, v0> }"

definition start :: "proc ⇒ state"
where "start p  ≡ s0"
```

— An example execution

```
definition exec ::
  "(proc, val, state ) configuration list"
where
  exec_def: "exec ≡ [ (|
    states = (λp. s0),
    msgs = ({# <p0, inM True> } ∪# {# <p1, inM True> }) |) ]"

lemma ProcUniv: "(UNIV :: proc set) = {p0, p1}"
  by (metis UNIV_eq_I insert_iff proc.exhaust)
```

7.2 Interpretation as FLP Locale

```
interpretation FLPSys: flpSystem trans sends start
proof
```

  — finiteProcs

```
  show "finite (UNIV :: proc set)"
    unfolding ProcUniv by simp
next
```

  — minimalProcs

```
  have "card (UNIV :: proc set) = 2"
    unfolding ProcUniv by simp
  thus "2 ≤ card (UNIV :: proc set)" by simp
```

```
next
  — finiteSends
  fix p s m
  have FinExplSends: "finite {<p0, v1>, <p1, v0>}" by auto
  have "{v. 0 < sends p s m v} ⊆ {<p0, v1>, <p1, v0>}"
  proof auto
    fix x
    assume "x ≠ <p0, v1>" "0 < sends p s m x"
    thus "x = <p1, v0>"
      by (metis (full_types) neq0_conv sends.simps(1,2) state.exhaust)
  qed
  thus "finite {v. 0 < sends p s m v}"
    using FinExplSends finite_subset by blast
next
  — noInSends
  fix p s m p2 v
  show "sends p s m <p2, inM v> = 0" by (induct s, auto)
qed

interpretation FLPExec: execution trans sends start exec "[]"
proof
  — notEmpty
  show "1 ≤ length exec"
    by (simp add:exec_def)
next
  — length
  show "length exec - 1 = length []"
    by (simp add:exec_def)
next
  — base
  show "asynchronousSystem.initial start (hd exec)"
    unfolding asynchronousSystem.initial_def isReceiverOf_def
    by (auto simp add: start_def exec_def, metis proc.exhaust)
next
  — step
  fix i cfg1 cfg2
  assume "i < length exec - 1"
  hence "False" by (simp add:exec_def)
  thus "asynchronousSystem.steps FLPExistingSystem.trans sends cfg1 ([] ! i) cfg2"
    by rule
qed

end
```

References

[1] H. Völzer. A Constructive Proof for FLP. *Inf. Process. Lett.*, 92(2):83–87, Oct. 2004.