Isabelle-FLP

Giuliano Losa

December 16, 2019

Abstract

This is a refactored version of a key lemma of the proof of the FLP theorem, reducing the size of the proof by almost 800 lines. The original Isabelle/HOL formalization appears in the Archive of Formal proofs [1] and is based on the proof of Völzer [2].

Contents

1 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 HOL−Library.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`.

1.1 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

fun unpackMessage :: ('p, 'v) message ⇒ 'v messageValue
where
  unpackMessage <p, inM b>  = Bool b
| unpackMessage <p, v>      = Value v
| unpackMessage <⊥, outM v> = Bool False

fun 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)
```

1.2 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)
```

1.3 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)
```

1.4 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.

```
fun 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:
  assumes
    Step: cfg1 ⊢ m ↦ cfg2 and Rec: ¬ isReceiverOf p m
  shows
    states cfg1 p = states cfg2 p
  using assms by (cases m, auto)

lemma ExistsMsg:
assumes
  Step: cfg1 ⊢ m ↦ cfg2
shows
  m ∈# (msgs cfg1)
```

```
using assms enabled-def by (cases m, auto)

lemma NoMessageLossStep:
assumes
  Step: cfg1 ⊢ m ↦ cfg2
shows
  msgs cfg1 ⊆# msgs cfg2 + {#m#}
  using subset-eq-diff-conv assms
  by (induct cfg1 m cfg2 rule:steps.induct)  fastforce+

lemma OutOnlyGrowing:
assumes
  cfg1 ⊢ m ↦ cfg2 isReceiverOf p m
shows
  count (msgs cfg2) <⊥, outM b>  = (count (msgs cfg1) <⊥, outM b>) +
    count (sends p (states cfg1 p) (unpackMessage m)) <⊥, outM b>
  using assms by (cases m, auto)

lemma OtherMessagesOnlyGrowing:
assumes
  Step: cfg1 ⊢ m ↦ cfg2 and m ≠ m'
shows count (msgs cfg1) m' ≤ count (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:
  assumes
    Step: cfg1 ⊢ m ↦ cfg2 and En: enabled cfg1 m' and NotEn: ¬ (enabled cfg2
m')
  shows m = m'
  using assms OtherMessagesOnlyGrowing
  apply (induct cfg1 m cfg2 rule:steps.induct; simp add:enabled-def)
   apply (metis (no-types, lifting) insert-DiffM insert-noteq-member union-iff)
  apply (metis (no-types, lifting)  insert-DiffM insert-noteq-member union-iff)
  done
```

1.5 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′
obtains
  cfg = cfg′
| *cfg1 msg p* ***where*** (*cfg* ⊢ *msg* ↦ *cfg1*) ∧ *enabled cfg msg*
  ∧ *isReceiverOf p msg* ∧ *reachable cfg1 cfg′*
using *assms*
  by (*induct rule*: *reachable.induct*, *auto*)
 (*metis asynchronousSystem.ExistsMsg asynchronousSystem.init asynchronousSystem.step enabled-def isReceiverOf.simps*(*1*) *isReceiverOf.simps*(*2*) *local.StepOutMsg message.exhaust*)

lemma *ReachableTrans*:
assumes *reachable cfg1 cfg2* ***and*** *reachable cfg2 cfg3*
shows *reachable cfg1 cfg3*
  using *assms*(*2*) *assms*(*1*) *asynchronousSystem.step* ***by*** (*induct rule*: *reachable.induct*, *auto*, *blast*)

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*:
assumes
  reachable cfg cfg′ ***and*** *enabled cfg msg* ***and*** ¬ (*enabled cfg′ msg*)
shows *stepReachable cfg msg cfg′*
 using *assms* ***by*** (*induct rule*: *reachable.induct*, *auto simp add*:*stepReachable-def*)
 (*metis asynchronousSystem.OnlyOccurenceDisables reachable.simps*)

1.6 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:
assumes
  qReachable cfg1 Q cfg2
shows
  reachable cfg1 cfg2
  using assms  apply (induct rule: qReachable.induct, auto)
  using init apply blast
  using asynchronousSystem.step apply blast
  done

lemma QReachableTrans:
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:
assumes
  qReachable cfg1 Q cfg2 and p ∉ Q
shows
  states cfg1 p = states cfg2 p
  using assms apply (induct rule: qReachable.induct, auto)
 by (metis (no-types) UniqueReceiverOf asynchronousSystem.NoReceivingNoChange)

corollary WithoutQReachablFrozenQ:
assumes
  Steps: withoutQReachable cfg1 Q cfg2 and P: p ∈ Q
shows
  states cfg1 p = states cfg2 p
using assms NotInQFrozenQReachability by simp

lemma NoActivityNoMessageLoss :
assumes
```

qReachable cfg1 Q cfg2 **and** $p \notin Q$ **and** *isReceiverOf p m'*
shows
count (msgs cfg1) m' $\leq$ *count (msgs cfg2) m'*
using *assms* **apply** (*induct rule*: *qReachable.induct*, *simp*)
by (*metis* (*no-types*, *lifting*) *OtherMessagesOnlyGrowing UniqueReceiverOf order-trans*)

lemma *NoMessageLoss*:
assumes
withoutQReachable cfg1 Q cfg2 **and** $p \in Q$ **and** *isReceiverOf p m'*
shows
count (msgs cfg1) m' $\leq$ *count (msgs cfg2) m'*
using *assms NoActivityNoMessageLoss* **by** *simp*

lemma *NoOutMessageLoss*:
assumes
reachable cfg1 cfg2
shows
count (msgs cfg1) <⊥, outM v> $\leq$ *count (msgs cfg2) <⊥, outM v>*
using *assms*
apply (*induct rule*: *reachable.induct*, *auto*)
by (*metis* (*no-types*, *lifting*) *OtherMessagesOnlyGrowing local.StepOutMsg order-trans*)

lemma *StillEnabled*:
assumes
withoutQReachable cfg1 Q cfg2 **and** $p \in Q$ **and** *isReceiverOf p msg* **and**
enabled cfg1 msg
shows
enabled cfg2 msg
using *assms*
by (*meson NoMessageLoss count-greater-eq-one-iff dual-order.trans enabled-def*)

1.7 Initial reachability

definition *initial* ::
(*'p*, *'v*, *'s*) *configuration* $\Rightarrow$ *bool*
where
initial cfg $\equiv$
(∀ *p*::*'p* . (∃ *v*::*bool* . (*count* (*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* . *count* (*msgs cfg*) <⊥, *outM v*> = *0*)
∧ (∀ *p v*. *count* (*msgs cfg*) <*p*, *v*> = *0*)
∧ *states cfg* = *start*

definition *initReachable* ::
(*'p*, *'v*, *'s*) *configuration* $\Rightarrow$ *bool*
where
initReachable cfg $\equiv$ ∃ *cfg0* . *initial cfg0* ∧ *reachable cfg0 cfg*

lemma *InitialIsInitReachable* :
assumes *initial c*
shows *initReachable c*
  using *assms reachable.init*
  unfolding *initReachable-def* ***by*** *blast*

1.8 Diamond property of reachability

lemma *DiamondOne*:
assumes
  StepP: $c \vdash m \ \mapsto c1$ ***and***
  PNotQ: $p \neq q$ ***and***
  Rec: *isReceiverOf p m* ***and***
  Rec′: *isReceiverOf q m′* ***and***
  StepQ: $c \vdash m' \mapsto c2$
shows
  $\exists\ c'\ .\ (c1 \vdash m' \mapsto c') \wedge (c2 \vdash m \mapsto c')$
proof −

First a few auxiliary facts.

  have *enabled c m′* ***and*** *enabled c m*
    using *asynchronousSystem.ExistsMsg enabled-def local.StepQ StepP* ***by*** *blast+*
  have $m \neq m'$ ***using*** *PNotQ Rec Rec′ UniqueReceiverOf* ***by*** *fastforce*
  { ***fix*** *p q c c1* ***and*** *m m′*::(′*p*, ′*v*) *message*
    assume $p \neq q$ ***and*** *isReceiverOf p m* ***and*** $c \vdash m \mapsto c1$ ***and*** *isReceiverOf q m′*
      and *enabled c m′*
    have *states c1 q* = *states c q* ***and*** *enabled c1 m′*
    proof −
      have *withoutQReachable c* {*q*} *c1*
        by (*meson DiffI UNIV-I* ‹$c \vdash m \mapsto c1$› ‹*isReceiverOf p m*› ‹$p \neq q$› *qReachable.simps singleton-iff*)
      thus *states c1 q* = *states c q* ***using*** *WithoutQReachablFrozenQ* ***by*** *auto*
    next
      show *enabled c1 m′*
        using *UniqueReceiverOf* ‹$c \vdash m \mapsto c1$› ‹*enabled c m′*› ‹*isReceiverOf p m*› ‹*isReceiverOf q m′*› ‹$p \neq q$› *asynchronousSystem.OnlyOccurenceDisables* ***by*** *fastforce*
    qed }
  note *1* = *this*[*of p q m c c1*, *OF* ‹$p \neq q$› ‹*isReceiverOf p m*› ‹$c \vdash m \ \mapsto c1$› ‹*isReceiverOf q m′*› ‹*enabled c m′*›]
    and *2* = *this*[*of q p m′ c c2*, *OF* ‹$p \neq q$›[*symmetric*] ‹*isReceiverOf q m′*› ‹$c \vdash m' \ \mapsto c2$› ‹*isReceiverOf p m*› ‹*enabled c m*›]

  define *c1′* ***where*** $c1' \equiv (\!|states = (states\ c1)(q := states\ c2\ q),$
    $msgs = (msgs\ c2 - (msgs\ c - \{\#m'\#\})) + (msgs\ c1 - \{\#m'\#\})|\!)$
  define *c2′* ***where*** $c2' \equiv (\!|states = (states\ c2)(p := states\ c1\ p),$
    $msgs = (msgs\ c1 - (msgs\ c - \{\#m\#\})) + (msgs\ c2 - \{\#m\#\})|\!)$

```
  have c1 ⊢ m′ ↦ c1′ using ‹c ⊢ m′ ↦ c2› 1 ‹isReceiverOf q m′›
    by (simp add:c1′-def; induct c m′ c2 rule:steps.induct)
      (auto simp add: enabled-def union-single-eq-diff add.commute)
  moreover
  have c2 ⊢ m ↦ c2′ using ‹c ⊢ m ↦ c1› 2 ‹isReceiverOf p m›
    by (simp add:c2′-def; induct c m c1 rule:steps.induct)
      (auto simp add: enabled-def union-single-eq-diff add.commute)
  moreover
  have c1′ = c2′ using 1 2 ‹p≠q› ‹enabled c m› ‹enabled c m′› ‹m ≠ m′› StepQ
StepP
      NoMessageLossStep[OF StepP] NoMessageLossStep[OF StepQ] Rec Rec′
   by (auto simp add:c1′-def c2′-def enabled-def fun-eq-iff add.commute subset-eq-diff-conv)
     (metis UniqueReceiverOf NoReceivingNoChange)
  ultimately show ?thesis by blast
qed

lemma DiamondTwo:
assumes
  QReach: qReachable c Q c1 and
  Step: c ⊢ m ↦ c2 ∃p∈Proc − Q. isReceiverOf p m
shows
  ∃ c′. (c1 ⊢ m ↦ c′) ∧ qReachable c2 Q c′
using assms
proof (induct c Q c1 rule: qReachable.induct)
  case (InitQ c Q)
  then show ?case using asynchronousSystem.InitQ by blast
next
  case (StepQ c1′ Q c2′ m2 c3)
  obtain c′ where c2′ ⊢ m ↦ c′ and qReachable c2 Q c′
    using StepQ.hyps(2)[OF StepQ.prems] by auto
  obtain c″ where c′ ⊢ m2 ↦ c″ and c3 ⊢ m ↦ c″
    using DiamondOne ‹c2′ ⊢ m ↦ c′› ‹c2′ ⊢ m2 ↦ c3›
       ‹∃p∈Q. isReceiverOf p m2› ‹∃p∈Proc − Q. isReceiverOf p m›  by (metis
DiffD2)
  moreover
  have qReachable c2 Q c″
    using ‹qReachable c2 Q c′› ‹c2′ ⊢ m2 ↦ c3› ‹c′ ⊢ m2 ↦ c″›
        ‹∃p∈Q. isReceiverOf p m2› ‹∃p∈Proc − Q. isReceiverOf p m› qReach-
able.StepQ by blast
  ultimately show ?case by blast
qed
```

Proposition 1 of Völzer.

```
lemma Diamond:
assumes
  QReach: qReachable c Q c1 and
  WithoutQReach: withoutQReachable c Q c2
shows
  ∃ c′. withoutQReachable c1 Q c′ ∧ qReachable c2 Q c′ using assms
```

```
proof (induct c Q c1 rule: qReachable.induct)
  case (InitQ c1 Q)
  then show ?case
    using asynchronousSystem.InitQ by blast
next
  case (StepQ c1 Q c2' m c3)
  obtain c' where qReachable c2' (Proc − Q) c' and qReachable c2 Q c'
    using StepQ.hyps(2) StepQ.prems by blast
  obtain c'' where qReachable c3 (Proc − Q) c'' and c' ⊢ m ↦ c''
    using ‹qReachable c2' (Proc − Q) c'› ‹c2' ⊢ m ↦ c3› ‹∃ p∈Q. isReceiverOf p
m›
    by (metis DiamondTwo DiffD2 DiffI UNIV-I)
  have qReachable c2 Q c'' using ‹qReachable c2 Q c'› ‹c' ⊢ m ↦ c''› ‹∃ p∈Q.
isReceiverOf p m›
    qReachable.StepQ by blast
  show ?case using ‹qReachable c3 (Proc − Q) c''› ‹qReachable c2 Q c''› by blast
qed

end

end
```

2 ListUtilities

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

```
theory ListUtilities
imports Main
begin

context begin
```

2.1 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
```

2.2 Lemmas for lists and nat predicates

```
lemma NatPredicateTippingPoint:
  assumes
    P0: P 0 and NotPN2: ¬P n2
  shows
    ∃n<n2. P n ∧ ¬P (Suc n)
  by (metis NotPN2 P0 dec-induct zero-le)

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.

```
private
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'==λx y . P (fst x) (snd x) (fst y) (snd y)
  hence ∀ x . ∃ y . (P' x  y) using A1 by auto
  hence A3: ∃ f . ∀ x . P' x (f x) by metis
  then obtain f where ∀ x . P' x (f x) by blast
  moreover define f' where  f'==λx1 x2. f (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 ≡ λx1 x2 . fst (f x1 x2)
  define f2 where f2 ≡ λx1 x2 . snd (f 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 set-rev-mp)

private
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
  by (metis assms in-set-conv-nth)

private
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

end

3 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
```

3.1 Locale for the FLP consensus setting

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

3.2 Decidedness and uniformity of configurations

```
abbreviation vDecided ::
  bool ⇒ ('p, 'v, 's) configuration ⇒ bool
where
  vDecided v 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 ≡
    (∃ c'::('p, 'v, 's) configuration . (withoutQReachable c {p} c')
    ∧ vDecided v c')

definition pSilentDecisionValues ::
```

$'p \Rightarrow ('p,\ 'v,\ 's)\ \mathit{configuration} \Rightarrow \mathit{bool\ set}\ (\mathit{val}[\text{-},\text{-}])$
where
$\mathit{pSilentDecisionValues\text{-}def}[\mathit{simp}]\text{:}\mathit{val}[p,\ c] \equiv \{v.\ \mathit{pSilDecVal}\ v\ p\ c\}$

definition *vUniform* ::
$\mathit{bool} \Rightarrow ('p,\ 'v,\ 's)\ \mathit{configuration} \Rightarrow \mathit{bool}$
where
$\mathit{vUniform}\ v\ c \equiv (\forall\, p.\ \mathit{val}[p,c] = \{v\})$

abbreviation *nonUniform* ::
$('p,\ 'v,\ 's)\ \mathit{configuration} \Rightarrow \mathit{bool}$
where
$\mathit{nonUniform}\ c \equiv$
$\quad\neg(\mathit{vUniform}\ \mathit{False}\ c) \wedge$
$\quad\neg(\mathit{vUniform}\ \mathit{True}\ c)$

3.3 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* ::
$('p,\ 'v,\ 's)\ \mathit{configuration} \Rightarrow \mathit{bool}$
where
$\mathit{agreement}\ c \equiv$
$\quad(\forall\, v1.\ (<\bot,\ \mathit{outM}\ v1> \in\#\ \mathit{msgs}\ c)$
$\qquad\longrightarrow (\forall\, v2.\ (<\bot,\ \mathit{outM}\ v2> \in\#\ \mathit{msgs}\ c)$
$\qquad\quad\longleftrightarrow v2 = v1))$

definition *agreementInit* ::
$('p,\ 'v,\ 's)\ \mathit{configuration} \Rightarrow ('p,\ 'v,\ 's)\ \mathit{configuration} \Rightarrow \mathit{bool}$
where
$\mathit{agreementInit}\ i\ c \equiv$
$\quad\mathit{initial}\ i \wedge \mathit{reachable}\ i\ c \longrightarrow$
$\qquad(\forall\, v1.\ (<\bot,\ \mathit{outM}\ v1> \in\#\ \mathit{msgs}\ c)$
$\qquad\quad\longrightarrow (\forall\, v2.\ (<\bot,\ \mathit{outM}\ v2> \in\#\ \mathit{msgs}\ c)$
$\qquad\qquad\longleftrightarrow v2 = v1))$

definition *validity* ::
$('p,\ 'v,\ 's)\ \mathit{configuration} \Rightarrow ('p,\ 'v,\ 's)\ \mathit{configuration} \Rightarrow \mathit{bool}$
where
$\mathit{validity}\ i\ c \equiv$
$\quad\mathit{initial}\ i \wedge \mathit{reachable}\ i\ c \longrightarrow$
$\qquad(\forall\, v.\ (<\bot,\ \mathit{outM}\ v> \in\#\ \mathit{msgs}\ c)$
$\qquad\quad\longrightarrow (\exists\, p.\ (<p,\ \mathit{inM}\ v> \in\#\ \mathit{msgs}\ i)))$

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
  $terminationPseudo\ t\ c\ Q \equiv ((initReachable\ c \wedge card\ Q + t \geq card\ Proc)$
    $\longrightarrow (\exists\, c'.\ qReachable\ c\ Q\ c' \wedge decided\ c'))$

3.4 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*:
assumes
  Termination: $\bigwedge cc\ Q\ .\ terminationPseudo\ 1\ cc\ Q$ ***and***
  Reachable: *initReachable c*
shows
  $val[p,c] \neq \{\}$
proof –
  from *Termination*
    have $(initReachable\ c \wedge card\ Proc \leq card\ (UNIV - \{p\}) + 1)$
      $\longrightarrow (\exists\, c'.\ qReachable\ c\ (UNIV - \{p\})\ c' \wedge (\exists\, v.\ <\bot, outM\ v> \in\#\ msgs\ c'\ ))$
    unfolding *terminationPseudo-def* ***by*** *simp*
  with *Reachable minimalProcs finite-UNIV*
    have $\exists\, c'.\ qReachable\ c\ (UNIV - \{p\})\ c' \wedge (\exists\, v.\ <\bot, outM\ v> \in\#\ msgs\ c')$
    unfolding *terminationPseudo-def initReachable-def* ***by*** *simp*
  thus *?thesis* ***using*** *Reachable* ***by*** (*auto simp add:pSilDecVal-def*)
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*:
assumes
  Reachable:*initReachable c* ***and***
  AllAgree: $\forall\ cfg\ .\ reachable\ c\ cfg \longrightarrow agreement\ cfg$ ***and***
  Termination: $\bigwedge cc\ Q\ .\ terminationPseudo\ 1\ cc\ Q$ ***and***
  VDec: *vDecided v c*
shows
  vUniform v c
 using *AllAgree VDec* ***unfolding*** *agreement-def vUniform-def pSilDecVal-def pSilentDecisionValues-def*
proof –

```
  assume
    Agree: ∀ cfg. reachable c cfg ⟶
      (∀ v1. <⊥, outM v1> ∈# msgs cfg
      ⟶ (∀ v2. (<⊥, outM v2> ∈# msgs cfg ) = (v2 = v1))) and
    vDec: <⊥, outM v> ∈# msgs c
  show
    (∀ p. {v. ∃ c'. qReachable c (Proc − {p}) c' ∧
      <⊥, outM v> ∈#  msgs c'} = {v})
  proof clarify
    fix p
    have val[p,c] ≠ {} using Termination DecisionValuesExist vDec Reachable by
blast
    hence NotEmpty: {v. ∃ c'. qReachable c (UNIV − {p}) c'
      ∧ initReachable c' ∧  (<⊥, outM v>) ∈# msgs c'} ≠ {}
      using pSilDecVal-def Reachable asynchronousSystem.InitQ vDec by blast
    have U: ∀  u . u ∈ {v. ∃ c'. qReachable c (UNIV − {p}) c'
      ∧  <⊥, outM v> ∈# msgs c'} ⟶ (u = v)
    proof clarify
      fix u c'
      assume qReachable c (UNIV − {p}) c'
      hence Reach: reachable c c' using QReachImplReach by simp
      from VDec have Msg: <⊥, outM v> ∈# msgs c by simp
      from Reach NoOutMessageLoss have
        count (msgs c) <⊥, outM v> ≤ count (msgs c') <⊥, outM v> by simp
      with Msg have VMsg:  <⊥, outM v> ∈# msgs c'
        using NoOutMessageLoss Reach by (metis count-eq-zero-iff le-zero-eq)
      assume  <⊥, outM u> ∈# msgs c'
      with Agree VMsg Reach show u = v by simp
    qed
    thus  {v. ∃ c'. qReachable c (UNIV − {p}) c' ∧
       <⊥, outM v> ∈# msgs c'} = {v} using NotEmpty U by blast
  qed
qed

corollary NonUniformImpliesNotDecided:
assumes
  ∀  cfg . reachable c cfg ⟶ agreement cfg
  ⋀cc Q . terminationPseudo 1 cc Q
  nonUniform c
  vDecided v c
  initReachable c
shows
  False by (metis (full-types) assms flpSystem.DecidedImpliesUniform flpSystem-axioms)
```

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:
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: <⊥, outM x> ∈# msgs cfg'  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 ∧  <⊥, outM x> ∈# msgs c'a 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 [2] 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:
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 cv
assume Cv: $qReachable\ c\ (Proc - \{q\})\ cv$
$<\bot, outM\ x> \in\#\ msgs\ cv$
have $\exists\, c'a.\ (cv \vdash msg \mapsto c'a) \wedge qReachable\ c'\ (Proc - \{q\})\ c'a$
using $DiamondTwo\ Cv(1)\ assms$ *by* $blast$
then obtain c'' *where* C'': $(cv \vdash msg \mapsto c'')$
$qReachable\ c'\ (Proc - \{q\})\ c''$ *by* $auto$
with $Cv(2)$ $initReachable\text{-}def\ reachable.simps$
have $Init$: $initReachable\ c''$ *by* $(meson\ Cv(1)\ QReachImplReach\ ReachableTrans$
$assms(4))$
have $reachable\ cv\ c''$ *using* $C''(1)$ $reachable.intros$ *by* $blast$
hence $count\ (msgs\ cv)\ <\bot, outM\ x> \leq count\ (msgs\ c'')\ <\bot, outM\ x>$ *using*
$NoOutMessageLoss$ *by* $simp$
hence $<\bot, outM\ x> \in\#\ msgs\ c''$ *using* $Cv(2)$ *by* $(metis\ count\text{-}greater\text{-}eq\text{-}one\text{-}iff$
$order\text{-}trans)$
thus $\exists\, c'a.\ qReachable\ c'\ (Proc - \{q\})\ c'a \wedge <\bot, outM\ x> \in\#\ msgs\ c'a$
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 [2] 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 $SilentDecisionValueNotInverting$:
assumes
Val: $val[q,c] = \{v\}$ *and*
$Step$: $c \vdash msg \mapsto c'$ *and*
Rec: $isReceiverOf\ p\ msg$ *and*
$Init$: $initReachable\ c$
shows
$val[q,c'] \neq \{\neg\ v\}$
proof$(cases\ p = q)$
case $False$
hence $val[q,c'] \subseteq val[q,c]$
using $Step\ Rec\ InactiveProcessSilentDecisionValuesDecrease\ Init$ *by* $simp$
with Val *show* $val[q,c'] \neq \{\neg\ v\}$ *by* $auto$
next
case $True$
hence $val[q,c] \subseteq val[q,c']$
using $Step\ Rec\ ActiveProcessSilentDecisionValuesIncrease\ Init$ *by* $simp$
with Val *show* $val[q,c'] \neq \{\neg\ v\}$ *by* $auto$
qed

3.5 Towards a proof of FLP

lemma $inM\text{-}all\text{-}eq\text{-}imp\text{-}uniform$:

```
  fixes i w
  assumes init:initial i
    and Validity: ⋀ i c . validity i c
    and inM:⋀ v . (∃ p. (<p, inM v> ∈# msgs i)) ⟹ v = w
    and Termination: ⋀cc Q . terminationPseudo 1 cc Q
  shows vUniform w i
proof –
  have 1:v = w if qReachable i (Proc – {p}) c′ and vDecided v c′ for p c′ v
    using that by (metis QReachImplReach Validity init inM validity-def)
  moreover
  have ∃ c′ . qReachable i (Proc – {p}) c′ ∧ vDecided w c′ for p
  proof –
    have val[p,i] ≠ {}
     using DecisionValuesExist Termination asynchronousSystem.InitialIsInitReachable
local.init by blast
    then obtain c′ u where 2:qReachable i (Proc – {p}) c′ ∧ vDecided u c′
      using pSilDecVal-def by auto
    hence u = w using 1 by blast
    thus ?thesis using 2 by blast
  qed
 ultimately show vUniform w i unfolding vUniform-def pSilDecVal-def pSilentDecisionValues-def
by auto
qed

lemma frozen-state-invisible:
  assumes
    withoutQReachable c Q d
    and ⋀ p . states c′ p = states c p
     and ⋀ p m . ⟦p ∉ Q; isReceiverOf p m⟧ ⟹ (count (msgs c′) m) = (count
(msgs c) m)
    and ⋀ v . count (msgs c′) <⊥,outM v> = (count (msgs c) <⊥,outM v>)
  shows ∃ d′ . withoutQReachable c′ Q d′ ∧ (∀ p . states d′ p = states d p)
   ∧ (∀ p m . p ∉ Q ∧ isReceiverOf p m ⟶ (count (msgs d′) m) = (count (msgs
d) m))
    ∧ (∀ v . count (msgs d′) <⊥,outM v> = (count (msgs d) <⊥,outM v>))
  using assms
proof (induct c Proc–Q d rule:qReachable.induct)
  case (InitQ c1)
  then show ?case using qReachable.InitQ by blast
next
  case (StepQ c1 d1 m d2)
  obtain d1′ where 1:qReachable c′ (Proc – Q) d1′ and 2:⋀ p . states d1′ p =
states d1 p
    and 3:⋀ p m . ⟦p ∉ Q; isReceiverOf p m⟧ ⟹ (count (msgs d1′) m) = (count
(msgs d1) m)
       and 4:⋀ v . count (msgs d1′) <⊥,outM v> = (count (msgs d1) <⊥,outM
v>)
    using StepQ.hyps(2) StepQ.prems by auto
  obtain p where 5:p ∉ Q and 6:isReceiverOf p m using StepQ.hyps(4) by blast
```

```
  define d2' where d2' ≡ (|states = (states d1')(p := states d2 p),
    msgs = (msgs d2 − (msgs d1 − {#m#})) + (msgs d1' − {#m#})|)

  have f1:⋀ p . states d2' p = states d2 p
    unfolding d2'-def using 2 6 ‹d1 ⊢ m ↦ d2›
   by (metis UniqueReceiverOf asynchronousSystem.NoReceivingNoChange fun-upd-apply
select-convs(1))

  define delta where delta ≡ msgs d2 − (msgs d1 − {#m#})
  have msgs':msgs d2' = delta + (msgs d1' − {#m#})
    using 2 ‹d1 ⊢ m ↦ d2› by (simp add:d2'-def delta-def; cases m; simp)
  have msgs:msgs d2 = delta + (msgs d1 − {#m#}) using ‹d1 ⊢ m ↦ d2›
   by ((simp add:delta-def)) (metis NoMessageLossStep subset-eq-diff-conv subset-mset.diff-add)
  have f2:(count (msgs d2') m2) = (count (msgs d2) m2) if p2 ∉ Q and isRe-
ceiverOf p2 m2 for p2 m2
  proof −
    have (count (msgs d1') m2) = (count (msgs d1) m2) using 3 that by blast
    with msgs msgs' show ?thesis by simp
  qed
  have f3:count (msgs d2') <⊥,outM v> = (count (msgs d2) <⊥,outM v>) for v
    unfolding d2'-def using 4 msgs by auto

  have f4:qReachable c' (Proc − Q) d2'
  proof −
    have d1' ⊢ m ↦ d2' using ‹d1 ⊢ m ↦ d2› 2 3 6
      by (cases m; simp-all add:d2'-def enabled-def) (metis 5 6 count-eq-zero-iff)+
    thus ?thesis using ‹qReachable c' (Proc − Q) d1'› 6 qReachable.StepQ 5 by
blast
  qed

  from f1 f2 f3 f4 show ?case by blast
qed
```

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−
  define n where n ≡ card Proc
```

We order the processes using a bijection to $\{0..<n\}$.

```
obtain f where f-bij:bij-betw f Proc {0..<n}
  using ex-bij-betw-finite-nat n-def finite-UNIV by blast
```

We define a family of configurations as in *This corresponds to **Lemma 1** in Völzer's paper.*.

```
define initMsgs :: nat ⇒ (('p, 'v) message) multiset
  where initMsgs ≡ (λ i . mset-set {m . ∃ p . m = <p, inM (f p < i)>})
define initCfg :: nat ⇒ ('p, 'v, 's) configuration where
  initCfg ≡ λ i . ⦇ states = start, msgs = initMsgs i ⦈
have count-initMsgs[simp]:count (initMsgs i) m = (if (∃ p . m = <p, inM f p
< i>) then 1 else 0) for i m
proof –
  have finite-initMsgs-set:
    finite {m . ∃ p . m = <p, inM (f p < i)>} (is finite ?S) for i
  proof –
   have ?S = (λ p . <p, inM (f p < i)>) ` UNIV by (simp add: full-SetCompr-eq)
    thus ?thesis by simp
  qed
  thus ?thesis
   using count-mset-set(1)[OF finite-initMsgs-set] count-mset-set(3) initMsgs-def
by auto
qed
hence in-initMsgs[iff]:m ∈# initMsgs i ⟷ (∃ p . m = <p, inM f p < i>) for
m i
  by (metis count-eq-zero-iff zero-neq-one)
```

All the configurations in the family are initial.

```
have InitInitial: initial c if 1:c ∈ initCfg ` {0..m} for c m
  using that unfolding initial-def initCfg-def
  by (cases c, auto simp add: split:if-splits message.splits)
```

Now we obtain an index j where the configuration j is uniform, but not the configuration $j + (1::'a)$

```
define P::nat ⇒ bool where P ≡ λ i . vUniform False (initCfg i)
obtain j where j∈{0..<(n+1)} and P j and ¬ (P (j+1))
proof –
  have P 0
  proof –
    have ⋀ v . (∃ p. (<p, inM v> ∈# msgs (initCfg 0))) ⟹ v = False
      unfolding initCfg-def by (auto split!:message.splits if-splits)
    moreover from InitInitial have initial (initCfg 0)
      by (simp add: finite-UNIV finite-UNIV-card-ge-0 n-def)
   ultimately show ?thesis using inM-all-eq-imp-uniform Validity Termination
P-def
      by blast
  qed
  have ¬ P (n+1)
```

```
  proof –
    have ⋀ v . (∃ p. (<p, inM v> ∈# msgs (initCfg (n+1)))) ⟹ v = True
      unfolding initCfg-def n-def by (auto split!:message.splits if-splits)
       (metis atLeastLessThan-iff bij-betw-imp-surj-on f-bij less-SucI n-def rangeI)
    moreover from InitInitial have initial (initCfg (n+1))
      by (meson atLeastAtMost-iff image-iff le0 order-refl)
  ultimately have vUniform True (initCfg (n+1)) using inM-all-eq-imp-uniform
Validity Termination
      by blast
    thus ?thesis unfolding P-def vUniform-def by auto
  qed
  from ‹P 0› and ‹¬ (P (n+1))› show ?thesis using that NatPredicateTipping-
Point by moura
qed
```

Now we show that the configuration $j + 1$ is non-uniform.

```
consider (a) vUniform True (initCfg (j+1)) | (b) nonUniform (initCfg (j+1))
  using ‹¬ (P (j+1))› P-def by blast
thus ?thesis
proof (cases)
  case a
```

We obtain an execution where False is decided, leading to a contradiction.

```
  define pj where pj ≡ (inv f) j
  obtain c where qReachable (initCfg (j+1)) (Proc−{pj}) c and vDecided False
c
  proof –
   obtain cj where 1:withoutQReachable (initCfg j) {pj} cj and vDecided False
cj
    using ‹P j› that unfolding P-def vUniform-def pSilDecVal-def pSilentDecisionValues-def
by auto
    have 2:⋀ p . states (initCfg j) p = states (initCfg (j+1)) p
      unfolding initCfg-def by auto
    have 3:count (msgs (initCfg j)) m = (count (msgs (initCfg (j+1))) m)
        if p ∉ {pj} and isReceiverOf p m for p m using that f-bij unfolding
initCfg-def pj-def
      by auto (metis UNIV-I bij-betw-def inv-into-f-f less-antisym)+
    have 4:count (msgs (initCfg (j+1))) <⊥,outM v> = (count (msgs (initCfg
j)) <⊥,outM v>) for v
      using initCfg-def by auto
    obtain cSucJ where withoutQReachable (initCfg (j+1)) {pj} cSucJ
      and ⋀ v . count (msgs cSucJ) <⊥,outM v> = (count (msgs cj) <⊥,outM
v>)
      using frozen-state-invisible[OF 1] 2 3 4 by simp blast
    have vDecided False cSucJ using ‹vDecided False cj›
        ‹⋀ v . count (msgs cSucJ) <⊥,outM v> = (count (msgs cj) <⊥,outM
v>)›
      by (metis count-eq-zero-iff)
    show ?thesis using that ‹vDecided False cSucJ› ‹withoutQReachable (initCfg
```

```
(j+1)) {pj} cSucJ›
        by blast
    qed
   with a have False unfolding vUniform-def pSilDecVal-def pSilentDecisionValues-def
      by blast
    thus ?thesis by auto
  next
    case b
    then show ?thesis using InitInitial atLeastAtMost-iff by blast
  qed
qed

lemma bool-set-cases:
  obtains bs = {} | bs = {True} | bs = {False} | bs = {True,False}
 by (cases bs = {}; cases bs = {True}; cases bs = {False}; cases bs = {True,False})
    (auto, (metis (full-types))+)
```

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:
assumes
  Init: initReachable c and
  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
```

Since the configuration is non-uniform, we obtain p with $val[p,c] = \{b\}$ and q with $(\neg\ b) \in val[q,c]$

```
  have 2:val[q,c] ≠ {} for q
    using DecisionValuesExist Init PseudoTermination by blast
  obtain b where  val[p,c] = {b} using 2 False
    by (cases val[p,c] rule:bool-set-cases; auto)
  obtain q where (¬ b) ∈ val[q,c]
  proof –
    obtain p2 where 4:val[p2,c] ≠ {b} using False that ‹nonUniform c›
      by (simp add:pSilDecVal-def vUniform-def) (metis (mono-tags, lifting))
    moreover
```

have $val[p2,c] \neq \{\}$ *using* 2 *by* auto
ultimately show ?thesis
using that *by* (cases $val[p2,c]$ rule:bool-set-cases; auto)
qed

Then we reach a configuration *cNotB* in which $val[p,cNotB] = \{\neg\ b\}$ by letting the system run without q and reach a $\neg\ b$ decision.

obtain cNotB *where* vDecided ($\neg$ b) cNotB *and* withoutQReachable c {q} cNotB
using ‹($\neg$ b) ∈ val[q,c]› pSilDecVal-def *by* auto
hence $val[p,cNotB] = \{\neg\ b\}$
by (meson Agree DecidedImpliesUniform Init PseudoTermination Reachable-Trans asynchronousSystem.QReachImplReach initReachable-def vUniform-def)

We obtain two configuration *cB* and *cNotB'*, on the way to *cNotB* where the set of silent decision values of p changes to include $\neg\ b$ or is {*True*, *False*} already.

obtain cB cNotB' m q' *where* $val[p,cB] = \{b\} \lor val[p,cB] = \{True,False\}$ *and* $(\neg b) \in val[p,cNotB']$ *and*
cB ⊢ m ↦ cNotB' *and* isReceiverOf q' m *and* withoutQReachable c {q} cB
using ‹withoutQReachable c {q} cNotB› ‹val[p,cNotB] = {¬ b}› ‹val[p,c] = {b}› ‹initReachable c›
proof (induct c Proc − {q} cNotB rule:qReachable.induct)
case (InitQ c1)
then show ?case *by* simp
next
case (StepQ c1 c2 msg c3)
then show ?case
proof (cases $val[p,c2] = \{\neg\ b\}$)
case True

Immediate by induction hypothesis.

then show ?thesis
using StepQ.hyps(2) StepQ.prems(1) StepQ.prems(3) StepQ.prems(4) *by* blast
next
case False
have $val[p,c2] \neq \{\}$
by (meson DecisionValuesExist PseudoTermination ReachableTrans StepQ.hyps(1) StepQ.prems(4) asynchronousSystem.QReachImplReach initReachable-def)
with False *have* $val[p,c2] = \{b\} \lor val[p,c2] = \{True,False\}$
by (cases $val[p,c2]$ rule:bool-set-cases) auto
then show ?thesis
by (metis StepQ.hyps(1) StepQ.hyps(3) StepQ.hyps(4) StepQ.prems(1) StepQ.prems(2) singletonI)
qed
qed

Trivial facts

```
  have initReachable cB
   using ‹withoutQReachable c {q} cB› ‹initReachable c›  QReachImplReach Reach-
ableTrans initReachable-def
    by blast
  have reachable c cNotB' using ‹cB ⊢ m ↦ cNotB'› ‹withoutQReachable c {q}
cB›
    using QReachImplReach reachable.step by blast
```

Now either $val[p,cB] = \{True,\ False\}$ or, using $[\![val[?q,?c] = \{?v\};\ ?c \vdash ?msg \mapsto ?c';\ isReceiverOf\ ?p\ ?msg;\ initReachable\ ?c]\!] \Longrightarrow val[?q,?c'] \neq \{\neg ?v\}$, $val[p,cNotB'] = \{True,\ False\}$

```
  consider val[p,cB] = {True,False} | val[p,cNotB'] = {True,False}
  proof −
    have val[p,cNotB'] = {True,False} if val[p,cB] ≠ {True,False}
    proof −
      from that and ‹val[p,cB] = {b} ∨ val[p,cB] = {True,False}›
      have val[p,cB] = {b}
        by linarith
      have val[p,cNotB'] ≠ {¬b}
        using  SilentDecisionValueNotInverting[OF ‹val[p,cB] = {b}› ‹cB ⊢ m ↦
cNotB'› ‹isReceiverOf q' m›
          ‹initReachable cB›] by simp
      with ‹(¬b) ∈ val[p,cNotB']› and 2
      show val[p,cNotB'] = {True,False} by fastforce
    qed
    thus ?thesis using that by blast
  qed
```

And in both cases we have found our c'

```
  hence ∃ c' . reachable c c' ∧ val[p,c'] = {True, False}
   using ‹reachable c cNotB'› ‹withoutQReachable c {q} cB› by (meson QReachIm-
plReach)
  with False 2 show ?thesis by auto
qed

end

end
```

References

[1] B. Bisping, P.-D. Brodmann, T. Jungnickel, C. Rickmann, H. Seidler, A. Stüber, A. Wilhelm-Weidner, K. Peters, and U. Nestmann. A constructive proof for flp. *Archive of Formal Proofs*, May 2016. http://isa-afp.org/entries/FLP.html, Formal proof development.

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