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*⟩

```
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"
⟨proof⟩
```

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⟩

lemma ExistsMsg:
fixes
  cfg1 cfg2 :: "('p,'v,'s) configuration" and
  m :: "('p,'v) message"
assumes
  Step: "cfg1 ⊢ m ↦ cfg2"
shows
  "m ∈# (msgs cfg1)"
⟨proof⟩
```

```
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'"
⟨proof⟩

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⟩

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'"
⟨proof⟩
```

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’"
⟨proof⟩
```

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’)"
⟨proof⟩

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⟩

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'"
⟨proof⟩
```

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"
⟨proof⟩

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"
⟨proof⟩
```

```
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"
⟨proof⟩

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"
⟨proof⟩

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')"
⟨proof⟩

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')"
⟨proof⟩
```

```
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>)"
⟨proof⟩

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"
⟨proof⟩
```

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"
  ⟨proof⟩
```

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⟩

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'"
⟨proof⟩
```

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⟩
```

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⟩

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 ≠ []"
⟨proof⟩

lemma PrefixListMonotonicity:
fixes
  l1 :: "'a list" and
  l2 :: "'a list"
assumes
  "prefixList l1 l2"
shows
  "length l1 < length l2"
⟨proof⟩

lemma TailIsPrefixList :
fixes
  l1 :: "'a list" and
  tail :: "'a list"
assumes "tail ≠ []"
shows "prefixList l1 (l1 @ tail)"
⟨proof⟩

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"
```
⟨*proof*⟩

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*⟩

```
lemma MinPredicate:
fixes
  P::"nat ⇒ bool"
assumes
  "∃ n . P n"
shows
  "(∃ n0 . (P n0) ∧ (∀ n' . (P n') ⟶ (n' ≥ n0)))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

`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*⟩

```
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*⟩

```
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))"
```
⟨*proof*⟩

```
lemma SmallerMultipleStepsWithLimit:
fixes
  k A limit
assumes
  "∀ n ≥ limit . (A (Suc n)) < (A n)"
shows
  "∀ n ≥ limit . (A (n + k)) ≤ (A n) - k"
```
⟨*proof*⟩

```
lemma PrefixSameOnLow:
fixes
  l1 l2
assumes
  "prefixList l1 l2"
shows
  "∀ index < length l1 . l1 ! index = l2 ! index"
```
⟨*proof*⟩

```
lemma KeepProperty:
fixes
  P Q low
assumes
  "∀ i ≥ low . P i ⟶ (P (Suc i) ∧ Q i)" "P low"
shows
  "∀ i ≥ low . Q i"
```
⟨*proof*⟩

```
lemma ListLenDrop:
fixes
  i la lb
```

```
assumes
  "i < length lb"
  "i ≥ la"
shows
  "lb ! i ∈ set (drop la lb)"
⟨proof⟩

lemma DropToShift:
fixes
  l i list
assumes
  "l + i < length list"
shows
  "(drop l list) ! i = list ! (l + i)"
⟨proof⟩

lemma SetToIndex:
fixes
  a and liste::"'a list"
assumes
  AssumpSetToIndex: "a ∈ set liste"
shows
  "∃ index < length liste . a = liste ! index"
⟨proof⟩

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⟩

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⟩

lemma ReachableInExecution:
assumes
  "i < length exec"
  "j ≤ i"
shows
  "reachable (execConf j) (execConf i)"
⟨proof⟩

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⟩

lemma ExistImpliesMinEnabled:
fixes
  msg :: "('p, 'v) message" and
  p :: 'p
assumes
  "isReceiverOf p msg"
  "enabled (last exec) msg"
shows
  "∃ msg' . minimalEnabled msg'"
⟨proof⟩

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⟩

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⟩

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⟩

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⟩

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"
⟨proof⟩

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⟩
```

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⟩
```

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"
  ⟨proof⟩

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"
⟨proof⟩
```

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⟩
```

...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⟩
```

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⟩
```

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⟩
```

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⟩

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⟩

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*⟩

```
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*⟩

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⟩
```

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⟩

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}"
  ⟨proof⟩
```

7.2 Interpretation as FLP Locale

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

interpretation FLPExec: execution trans sends start exec "[]"
⟨proof⟩

end
```

References

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