Minimal CBC Casper Isabelle/HOL proofs

LayerX

March 18, 2019

Contents

1 CBC Casper **4**

2 Message Justification **12**

3 Latest Message **16**

4 Safety Proof **31**

theory *Strict-Order*

imports *Main*

begin

notation *Set.empty* (∅)

definition *strict-partial-order* $r \equiv$ *trans* $r \wedge$ *irrefl* r

definition *strict-well-order-on* $A\ r \equiv$ *strict-linear-order-on* $A\ r \wedge$ *wf* r

lemma *strict-linear-order-is-strict-partial-order* :
strict-linear-order-on $A\ r \Longrightarrow$ *strict-partial-order* r
by (*simp add*: *strict-linear-order-on-def strict-partial-order-def*)

definition *upper-bound-on* :: *'a set* ⇒ *'a rel* ⇒ *'a* ⇒ *bool*
where
upper-bound-on $A\ r\ x = (\forall\ y.\ y \in A \longrightarrow (y, x) \in r \vee x = y)$

definition *maximum-on* :: *'a set* ⇒ *'a rel* ⇒ *'a* ⇒ *bool*
where

```
    maximum-on A r x = (x ∈ A ∧ upper-bound-on A r x)

definition minimal-on :: 'a set ⇒ 'a rel ⇒ 'a ⇒ bool
  where
    minimal-on A r x = (x ∈ A ∧ (∀ y. (y, x) ∈ r ⟶ y ∉ A))

definition maximal-on :: 'a set ⇒ 'a rel ⇒ 'a ⇒ bool
  where
    maximal-on A r x = (x ∈ A ∧ (∀ y. (x, y) ∈ r ⟶ y ∉ A))

lemma maximal-and-maximum-coincide-for-strict-linear-order :
  strict-linear-order-on A r ⟹ maximal-on A r x = maximum-on A r x
  apply (simp add: strict-linear-order-on-def irrefl-def total-on-def trans-def maximal-on-def
maximum-on-def upper-bound-on-def)
  by blast

lemma strict-partial-order-on-finite-non-empty-set-has-maximal :
  strict-partial-order r ⟶ finite A ⟶ A ≠ ∅ ⟶ (∃ x. maximal-on A r x)
proof −
  have ⋀n. strict-partial-order r ⟹ (∀ A. Suc n = card A ⟶ finite A ⟶ A ≠
∅ ⟶ (∃ x. maximal-on A r x))
  proof −
    assume strict-partial-order r
    then have (∀ a. (a, a) ∉ r)
      by (simp add: strict-partial-order-def irrefl-def)
    fix n
    show ∀ A. Suc n = card A ⟶ finite A ⟶ A ≠ ∅ ⟶ (∃ x. maximal-on A r
x)
      apply (induction n)
      unfolding maximal-on-def
      using ‹(∀ a. (a, a) ∉ r)›
      apply (metis card-eq-SucD empty-iff insert-iff)
    proof −
      fix n
      assume ∀ A. Suc n = card A ⟶ finite A ⟶ A ≠ ∅ ⟶ (∃ x. x ∈ A ∧ (∀ y.
(x, y) ∈ r ⟶ y ∉ A))
      have ∀ B. Suc (Suc n) = card B ⟶ finite B ⟶ B ≠ ∅ ⟶ (∃ A' b. B =
A' ∪ {b} ∧ card A' = Suc n ∧ b ∉ A')
        by (metis Un-commute add-diff-cancel-left' card-gt-0-iff card-insert-disjoint
card-le-Suc-iff insert-is-Un not-le not-less-eq-eq plus-1-eq-Suc)
      then have ∀ B. Suc (Suc n) = card B ⟶ finite B ⟶ B ≠ ∅ ⟶ (∃ A' b.
B = A' ∪ {b} ∧ card A' = Suc n ∧ finite A' ∧ A' ≠ ∅ ∧ b ∉ A')
        by (metis card-gt-0-iff zero-less-Suc)
      then have ∀ B. Suc (Suc n) = card B ⟶ finite B ⟶ B ≠ ∅
          ⟶ (∃ A' b x. B = A' ∪ {b} ∧ b ∉ A' ∧ x ∈ A' ∧ (∀ y. (x, y) ∈ r ⟶ y
∉ A'))
        using ‹∀ A. Suc n = card A ⟶ finite A ⟶ A ≠ ∅ ⟶ (∃ x. x ∈ A ∧ (∀ y.
(x, y) ∈ r ⟶ y ∉ A))›
        by metis
      then show ∀B. Suc (Suc n) = card B ⟶ finite B ⟶ B ≠ ∅ ⟶ (∃x. x
∈ B ∧ (∀y. (x, y) ∈ r ⟶ y ∉ B))
      by (metis (no-types, lifting) Un-insert-right ‹∀ a. (a, a) ∉ r› ‹strict-partial-order
r› insertE insert-iff strict-partial-order-def sup-bot.right-neutral transE)
    qed
  qed
  then show ?thesis
    by (metis card.insert-remove finite.cases)
qed

lemma strict-partial-order-has-at-most-one-maximum :
  strict-partial-order r
  ⟶ {x. maximum-on A r x} ≠ ∅
  ⟶ is-singleton {x. maximum-on A r x}
proof (rule ccontr)
  assume ¬ (strict-partial-order r ⟶ {x. maximum-on A r x} ≠ ∅ ⟶ is-singleton
{x. maximum-on A r x})
  then have strict-partial-order r ⟶ {x. maximum-on A r x} ≠ ∅ ⟶ ¬ is-singleton
{x. maximum-on A r x}
    by simp
  then have strict-partial-order r ⟶ {x. maximum-on A r x} ≠ ∅ ⟶ (∃ x1 x2.
x1 ≠ x2 ∧ {x1, x2} ⊆ {x. maximum-on A r x})
    by (meson empty-subsetI insert-subset is-singletonI')
  then have strict-partial-order r ⟶ {x. maximum-on A r x} ≠ ∅ ⟶ (∃ x1 x2.
x1 ≠ x2 ∧ {x1, x2} ⊆ {x ∈ A. ∀ y. y ∈ A ⟶ (y, x) ∈ r ∨ x = y})
    by (simp add: maximum-on-def upper-bound-on-def)
  then have strict-partial-order r ⟶ {x. maximum-on A r x} ≠ ∅ ⟶ (∃ x1 x2.
x1 ≠ x2 ∧ {x1, x2} ⊆ A ∧ (∀ y. y ∈ A ⟶ (y, x1) ∈ r ∨ x1 = y) ∧ (∀ y. y ∈
A ⟶ (y, x2) ∈ r ∨ x2 = y))
    by auto
  then show False
    using strict-partial-order-def

      by (metis ‹¬ (strict-partial-order r ⟶ {x. maximum-on A r x} ≠ ∅ ⟶
is-singleton {x. maximum-on A r x})› insert-subset irrefl-def transE)
qed

lemma strict-linear-order-on-finite-non-empty-set-has-one-maximum :
  strict-linear-order-on A r ⟶ finite A ⟶ A ≠ ∅ ⟶ is-singleton {x. maximum-on
A r x}
  using strict-linear-order-is-strict-partial-order strict-partial-order-on-finite-non-empty-set-has-maximal

      strict-partial-order-has-at-most-one-maximum maximal-and-maximum-coincide-for-strict-linear-order
  by fastforce

end
```

1 CBC Casper

```
theory CBCCasper

imports Main HOL.Real Libraries/Strict-Order Libraries/Restricted-Predicates Libraries/LaTeXsugar

begin
```

```
notation Set.empty (∅)
```

```
typedecl validator

typedecl consensus-value
```

```
datatype message =
  Message consensus-value ∗ validator ∗ message list

type-synonym state = message set
```

```
fun sender :: message ⇒ validator
  where
    sender (Message (-, v, -)) = v

fun est :: message ⇒ consensus-value
  where
     est (Message (c, -, -)) = c

fun justification :: message ⇒ state
  where
    justification (Message (-, -, s)) = set s
```

```
fun
  Σi :: (validator set × consensus-value set × (message set ⇒ consensus-value set)) ⇒ nat ⇒ state set and
  Mi :: (validator set × consensus-value set × (message set ⇒ consensus-value set)) ⇒ nat ⇒ message set
  where
    Σi (V,C,ε) 0 = {∅}
  | Σi (V,C,ε) n = {σ ∈ Pow (Mi (V,C,ε) (n − 1)). finite σ ∧ (∀ m. m ∈ σ ⟶
justification m ⊆ σ)}
  | Mi (V,C,ε) n = {m. est m ∈ C ∧ sender m ∈ V ∧ justification m ∈ (Σi
(V,C,ε) n) ∧ est m ∈ ε (justification m)}

locale Params =
  fixes V :: validator set
  and W :: validator ⇒ real
  and t :: real
  fixes C :: consensus-value set
  and ε :: message set ⇒ consensus-value set

begin
  definition Σ = (⋃ i∈ℕ. Σi (V,C,ε) i)
  definition M = (⋃ i∈ℕ. Mi (V,C,ε) i)
  definition is-valid-estimator :: (state ⇒ consensus-value set) ⇒ bool
    where
      is-valid-estimator e = (∀σ ∈ Σ. e σ ∈ Pow C − {∅})


  lemma Σi-subset-Mi: Σi (V,C,ε) (n + 1) ⊆ Pow (Mi (V,C,ε) n)
    by force

  lemma Σi-subset-to-Mi: Σi (V,C,ε) n ⊆ Σi (V,C,ε) (n+1) ⟹ Mi (V,C,ε) n
⊆ Mi (V,C,ε) (n+1)
    by auto

  lemma Mi-subset-to-Σi: Mi (V,C,ε) n ⊆ Mi (V,C,ε) (n+1) ⟹ Σi (V,C,ε)
(n+1) ⊆ Σi (V,C,ε) (n+2)
    by auto

  lemma Σi-monotonic: Σi (V,C,ε) n ⊆ Σi (V,C,ε) (n+1)
    apply (induction n)
    apply simp
   apply (metis Mi-subset-to-Σi Suc-eq-plus1 Σi-subset-to-Mi add.commute add-2-eq-Suc)
    done

  lemma Mi-monotonic: Mi (V,C,ε) n ⊆ Mi (V,C,ε) (n+1)
    apply (induction n)
    defer
    using Σi-monotonic Σi-subset-to-Mi apply blast
    apply auto
    done

  lemma Σi-monotonicity: ∀ m ∈ ℕ. ∀ n ∈ ℕ. m ≤ n ⟶ Σi (V,C,ε) m ⊆ Σi
(V,C,ε) n
    using Σi-monotonic
    by (metis Suc-eq-plus1 lift-Suc-mono-le)

  lemma Mi-monotonicity: ∀ m ∈ ℕ. ∀ n ∈ ℕ. m ≤ n ⟶ Mi (V,C,ε) m ⊆ Mi
(V,C,ε) n
    using Mi-monotonic
    by (metis Suc-eq-plus1 lift-Suc-mono-le)

  lemma message-is-in-Mi :
   ∀ m ∈ M. ∃ n ∈ ℕ. m ∈ Mi (V, C, ε) (n − 1)
    apply (simp add: M-def Σi.elims)
    by (metis Nats-1 Nats-add One-nat-def diff-Suc-1 plus-1-eq-Suc)

  lemma state-is-in-pow-Mi :
   ∀ σ ∈ Σ. (∃ n ∈ ℕ. σ ∈ Pow (Mi (V, C, ε) (n − 1)) ∧ (∀ m ∈ σ. justification
m ⊆ σ))
    apply (simp add: Σ-def)


    apply auto
    proof −
      fix y :: nat and σ :: message set
      assume a1: σ ∈ Σi (V, C, ε) y
      assume a2: y ∈ ℕ
      have σ ⊆ Mi (V, C, ε) y
         using a1 by (meson Params.Σi-monotonic Params.Σi-subset-Mi Pow-iff
contra-subsetD)
      then have ∃n. n ∈ ℕ ∧ σ ⊆ Mi (V, C, ε) (n − 1)
        using a2 by (metis (no-types) Nats-1 Nats-add diff-Suc-1 plus-1-eq-Suc)
       then show ∃n∈ℕ. σ ⊆ {m. est m ∈ C ∧ sender m ∈ V ∧ justification m
∈ Σi (V, C, ε) (n − Suc 0) ∧ est m ∈ ε (justification m)}
        by auto
    next
       show ⋀y σ m x. y ∈ ℕ ⟹ σ ∈ Σi (V, C, ε) y ⟹ m ∈ σ ⟹ x ∈
justification m ⟹ x ∈ σ
        using Params.Σi-monotonic by fastforce
    qed

  lemma message-is-in-Mi-n :
   ∀ m ∈ M. ∃ n ∈ ℕ. m ∈ Mi (V, C, ε) n
   by (smt Mi-monotonic Suc-diff-Suc add-leE diff-add diff-le-self message-is-in-Mi
neq0-conv plus-1-eq-Suc subsetCE zero-less-diff)

  lemma message-in-state-is-valid :
   ∀ σ m. σ ∈ Σ ∧ m ∈ σ ⟶  m ∈ M
    apply (rule, rule, rule)
  proof −
    fix σ m
    assume σ ∈ Σ ∧ m ∈ σ
    have
```

```
      ∃ n ∈ ℕ. m ∈ Mi (V, C, ε) n
      ⟹ m ∈ M
      using M-def by blast
    then show
      m ∈ M
      apply (simp add: M-def)
       by (smt Mi.simps Params.Σi-monotonic PowD Suc-diff-Suc ‹σ ∈ Σ ∧ m ∈
σ› add-leE diff-add diff-le-self gr0I mem-Collect-eq plus-1-eq-Suc state-is-in-pow-Mi
subsetCE zero-less-diff)
  qed

  lemma state-is-subset-of-M : ∀ σ ∈ Σ. σ ⊆ M
    using message-in-state-is-valid by blast

  lemma state-is-finite : ∀ σ ∈ Σ. finite σ
    apply (simp add: Σ-def)
    using Params.Σi-monotonic by fastforce

  lemma justification-is-finite : ∀ m ∈ M. finite (justification m)
    apply (simp add: M-def)
    using Params.Σi-monotonic by fastforce

  lemma Σis-subseteq-of-pow-M: Σ ⊆ Pow M
    by (simp add: state-is-subset-of-M subsetI)

  lemma M-type: ⋀m. m ∈ M ⟹ est m ∈ C ∧ sender m ∈ V ∧ justification m
∈ Σ
    unfolding M-def Σ-def
    by auto

end


locale Protocol = Params +
  assumes V-type: V ≠ ∅ ∧ finite V
  and W-type: ⋀w. w ∈ range W ⟹ w > 0
  and t-type: 0 ≤ t t < Sum (W ‘ V)
  and C-type: card C > 1
  and ε-type: is-valid-estimator ε

lemma (in Protocol) estimates-are-non-empty: ⋀ σ. σ ∈ Σ ⟹ ε σ ≠ ∅
  using is-valid-estimator-def ε-type by auto

lemma (in Protocol) estimates-are-subset-of-C: ⋀ σ. σ ∈ Σ ⟹ ε σ ⊆ C
  using is-valid-estimator-def ε-type by auto

lemma (in Params) empty-set-exists-in-Σ-0: ∅ ∈ Σi (V, C, ε) 0
  by simp
```

```
lemma (in Params) empty-set-exists-in-Σ: ∅ ∈ Σ
  apply (simp add: Σ-def)
  using Nats-0 Σi.simps(1) by blast

lemma (in Params) Σi-is-non-empty: Σi (V, C, ε) n ≠ ∅
  apply (induction n)
  using empty-set-exists-in-Σ-0 by auto

lemma (in Params) Σis-non-empty: Σ ≠ ∅
  using empty-set-exists-in-Σ by blast

lemma (in Protocol) estimates-exists-for-empty-set :
  ε ∅ ≠ ∅
  by (simp add: empty-set-exists-in-Σ estimates-are-non-empty)

lemma (in Protocol) non-justifying-message-exists-in-M-0:
  ∃ m. m ∈ Mi (V, C, ε) 0 ∧ justification m = ∅
  apply auto
proof −
  have ε ∅ ⊆ C
    using Params.empty-set-exists-in-Σ ε-type is-valid-estimator-def by auto
  then show ∃m. est m ∈ C ∧ sender m ∈ V ∧ justification m = ∅ ∧ est m ∈ ε (justification m) ∧ justification m = ∅
    by (metis V-type all-not-in-conv est.simps estimates-exists-for-empty-set justification.simps sender.simps set-empty subsetCE)
qed

lemma (in Protocol) Mi-is-non-empty: Mi (V, C, ε) n ≠ ∅
  apply (induction n)
  using non-justifying-message-exists-in-M-0 apply auto
  using Mi-monotonic empty-iff empty-subsetI by fastforce

lemma (in Protocol) Mis-non-empty: M ≠ ∅
  using non-justifying-message-exists-in-M-0 M-def Nats-0 by blast

lemma (in Protocol) C-is-not-empty : C ≠ ∅
  using C-type by auto

lemma (in Params) Σi-is-subset-of-Σ :
  ∀ n ∈ ℕ. Σi (V, C, ε) n ⊆ Σ
  by (simp add: Σ-def SUP-upper)

lemma (in Protocol) message-justifying-state-in-Σ-n-exists-in-M-n :
  ∀ n ∈ ℕ. (∀ σ. σ ∈ Σi (V, C, ε) n ⟶ (∃ m. m ∈ Mi (V, C, ε) n ∧ justification m = σ))
  apply auto
proof −
  fix n σ
  assume n ∈ ℕ
  and σ ∈ Σi (V, C, ε) n
  then have σ ∈ Σ
    using Σi-is-subset-of-Σ by auto
  have ε σ ≠ ∅
    using estimates-are-non-empty ‹σ ∈ Σ› by auto
  have finite σ
    using state-is-finite ‹σ ∈ Σ› by auto
  moreover have ∃ m. sender m ∈ V ∧ est m ∈ ε σ ∧ justification m = σ
    using est.simps sender.simps justification.simps V-type ‹ε σ ≠ ∅› ‹finite σ›
    by (metis all-not-in-conv finite-list)
  moreover have ε σ ⊆ C
     using estimates-are-subset-of-C Σi-is-subset-of-Σ ‹n ∈ ℕ› ‹σ ∈ Σi (V, C, ε) n› by blast
  ultimately show ∃ m. est m ∈ C ∧ sender m ∈ V ∧ justification m ∈ Σi (V, C, ε) n ∧ est m ∈ ε (justification m) ∧ justification m = σ
    using Nats-1 One-nat-def
    using ‹σ ∈ Σi (V, C, ε) n› by blast
qed

lemma (in Protocol) Σ-type: Σ ⊂ Pow M
proof −
  obtain m where m ∈ Mi (V, C, ε) 0 ∧ justification m = ∅
    using non-justifying-message-exists-in-M-0 by auto
  then have {m} ∈ Σi (V, C, ε) (Suc 0)
    using Params.Σi-subset-Mi by auto
  then have ∃ m'. m' ∈ Mi (V, C, ε) (Suc 0) ∧ justification m' = {m}
      using message-justifying-state-in-Σ-n-exists-in-M-n Nats-1 One-nat-def by metis
  then obtain m' where m' ∈ Mi (V, C, ε) (Suc 0) ∧ justification m' = {m} by auto
  then have {m'} ∈ Pow M
    using M-def
    by (metis Nats-1 One-nat-def PowD PowI Pow-bottom UN-I insert-subset)
  moreover have {m'} ∉ Σ
    using Params.state-is-in-pow-Mi Protocol-axioms ‹m' ∈ Mi (V, C, ε) (Suc 0) ∧ justification m' = {m}› by fastforce
  ultimately show ?thesis
    using Σis-subseteq-of-pow-M by auto
qed


lemma (in Protocol) M-type-counterexample:
  (∀ σ. ε σ = C) ⟹ M = {m. est m ∈ C ∧ sender m ∈ V ∧ justification m ∈ Σ}
  apply (simp add: M-def)
  apply auto
  using Σi-is-subset-of-Σ apply blast
  by (simp add: Σ-def)
```

```
definition observed :: message set ⇒ validator set
  where
    observed σ = {sender m | m. m ∈ σ}

lemma (in Protocol) observed-type :
  ∀ σ ∈ Pow M. observed σ ∈ Pow V
  using Params.M-type Protocol-axioms observed-def by fastforce

lemma (in Protocol) observed-type-for-state :
  ∀ σ ∈ Σ. observed σ ⊆ V
 using Params.M-type Protocol-axioms observed-def state-is-subset-of-M by fastforce


fun is-future-state :: (state ∗ state) ⇒ bool
  where
    is-future-state (σ1, σ2) = (σ1 ⊆ σ2)

lemma (in Params) state-difference-is-valid-message :
  ∀ σ σ'. σ ∈ Σ ∧ σ' ∈ Σ
  ⟶ is-future-state(σ, σ')
  ⟶ σ' − σ ⊆ M
  using state-is-subset-of-M by blast


definition justified :: message ⇒ message ⇒ bool
  where
    justified m1 m2 = (m1 ∈ justification m2)




definition equivocation :: (message ∗ message) ⇒ bool
  where
    equivocation =
      (λ(m1, m2). sender m1 = sender m2 ∧ m1 ≠ m2 ∧ ¬ (justified m1 m2) ∧
¬ (justified m2 m1))


definition is-equivocating :: state ⇒ validator ⇒ bool
  where
    is-equivocating σ v = (∃ m1 ∈ σ. ∃ m2 ∈ σ. equivocation (m1, m2) ∧ sender
m1 = v)

definition equivocating-validators :: state ⇒ validator set
  where
    equivocating-validators σ = {v ∈ observed σ. is-equivocating σ v}
```

```
lemma (in Protocol) equivocating-validators-type :
  ∀ σ ∈ Σ. equivocating-validators σ ⊆ V
  using observed-type-for-state equivocating-validators-def by blast

lemma (in Protocol) equivocating-validators-is-finite :
  ∀ σ ∈ Σ. finite (equivocating-validators σ)
  using V-type equivocating-validators-type rev-finite-subset by blast

definition (in Params) equivocating-validators-paper :: state ⇒ validator set
  where
    equivocating-validators-paper σ = {v ∈ V. is-equivocating σ v}

lemma (in Protocol) equivocating-validators-is-equivalent-to-paper :
  ∀ σ ∈ Σ. equivocating-validators σ = equivocating-validators-paper σ
  by (smt Collect-cong Params.equivocating-validators-paper-def equivocating-validators-def
is-equivocating-def mem-Collect-eq observed-type-for-state observed-def subsetCE)

lemma (in Protocol) equivocation-is-monotonic :
  ∀ σ σ' v. σ ∈ Σ ∧ σ' ∈ Σ ∧ is-future-state (σ, σ') ∧ v ∈ V
  ⟶ v ∈ equivocating-validators σ
  ⟶ v ∈ equivocating-validators σ'
  apply (simp add: equivocating-validators-def is-equivocating-def)
  using observed-def by fastforce
```

```
definition (in Params) weight-measure :: validator set ⇒ real
  where

    weight-measure v-set = Sum (W ‘v-set)

lemma (in Protocol) weight-measure-comparison-strict-subset-gte :
  finite A ⟹ finite B ⟹ B ⊆ A ⟹ weight-measure A ≥ weight-measure B
  apply (simp add: weight-measure-def)
  using W-type
  by (smt Diff-iff finite-imageI subsetCE subset-UNIV subset-image-iff sum-mono2)

lemma (in Protocol) weight-measure-comparison-stritct-subset-gt :
  finite A ⟹ finite B ⟹ B ⊂ A ⟹ weight-measure A > weight-measure B
  apply (simp add: weight-measure-def)
  using W-type
  oops

lemma (in Protocol) weight-measure-gt-set-difference :
  finite A ⟹ finite B ⟹ B ≠ ∅ ⟹ weight-measure A > weight-measure (A −
B)
```

oops

definition (**in** *Params*) *equivocation-fault-weight* :: *state* ⇒ *real*
where

equivocation-fault-weight σ = *weight-measure* (*equivocating-validators* σ)

lemma (**in** *Protocol*) *equivocation-fault-weight-is-monotonic* :
$\forall\ \sigma\ \sigma'.\ \sigma \in \Sigma \wedge \sigma' \in \Sigma \wedge$ *is-future-state* (σ, σ')
$\longrightarrow$ *equivocation-fault-weight* $\sigma \leq$ *equivocation-fault-weight* σ'
using *equivocation-is-monotonic weight-measure-comparison-strict-subset-gte*
by (*smt equivocating-validators-is-finite equivocating-validators-type equivocation-fault-weight-def subset-iff*)

definition (**in** *Params*) *is-faults-lt-threshold* :: *state* ⇒ *bool*
where
is-faults-lt-threshold σ = (*equivocation-fault-weight* $\sigma < t$)

definition (**in** *Protocol*) Σt :: *state set*
where
$\Sigma t = \{\sigma \in \Sigma.$ *is-faults-lt-threshold* $\sigma\}$

lemma (**in** *Protocol*) Σt*-is-subset-of-*Σ : $\Sigma t \subseteq \Sigma$
using Σt*-def* **by** *auto*

type-synonym *state-property* = *state* ⇒ *bool*

type-synonym *consensus-value-property* = *consensus-value* ⇒ *bool*

end

2 Message Justification

theory *MessageJustification*

imports *Main CBCCasper Libraries/LaTeXsugar*

begin

definition (**in** *Params*) *message-justification* :: *message rel*
where

```
  message-justification = {(m1, m2). {m1, m2} ⊆ M ∧ justified m1 m2}

lemma (in Protocol) transitivity-of-justifications :
  trans message-justification
  apply (simp add: trans-def message-justification-def justified-def)
  by (meson Params.M-type Params.state-is-in-pow-Mi Protocol-axioms contra-subsetD)

lemma (in Protocol) irreflexivity-of-justifications :
  irrefl message-justification
  apply (simp add: irrefl-def message-justification-def justified-def)
  apply (simp add: M-def)
  apply auto
proof −
  fix n m
  assume est m ∈ C
  assume sender m ∈ V
  assume justification m ∈ Σi (V, C, ε) n
  assume est m ∈ ε (justification m)
  assume m ∈ justification m
  have m ∈ Mi (V, C, ε) (n − 1)
    by (smt Mi.simps One-nat-def Params.Σi-subset-Mi Pow-iff Suc-pred ‹est m ∈
C› ‹est m ∈ ε (justification m)› ‹justification m ∈ Σi (V, C, ε) n› ‹m ∈ justification
m› ‹sender m ∈ V› add.right-neutral add-Suc-right diff-is-0-eq' diff-le-self diff-zero
mem-Collect-eq not-gr0 subsetCE)
  then have justification m ∈ Σi (V, C, ε) (n − 1)
    using Mi.simps by blast
  then have justification m ∈ Σi (V, C, ε) 0
    apply (induction n)
    apply simp
     by (smt Mi.simps One-nat-def Params.Σi-subset-Mi Pow-iff Suc-pred ‹m ∈
justification m› add.right-neutral add-Suc-right diff-Suc-1 mem-Collect-eq not-gr0
subsetCE subsetCE)
  then have justification m ∈ {∅}
    by simp
  then show False
    using ‹m ∈ justification m› by blast
qed

lemma (in Protocol) message-cannot-justify-itself :
  (∀ m ∈ M. ¬ justified m m)
proof −
  have irrefl message-justification
    using irreflexivity-of-justifications by simp
  then show ?thesis
    by (simp add: irreflexivity-of-justifications irrefl-def message-justification-def)
qed

lemma (in Protocol) justification-is-strict-partial-order-on-M :
  strict-partial-order message-justification
  apply (simp add: strict-partial-order-def)
  by (simp add: irreflexivity-of-justifications transitivity-of-justifications)

lemma (in Protocol) monotonicity-of-justifications :
  ∀ m m′ σ. m ∈ M ∧ σ ∈ Σ ∧ justified m′ m ⟶ justification m′ ⊆ justification
m
  apply simp
  by (meson M-type justified-def message-in-state-is-valid state-is-in-pow-Mi)

lemma (in Protocol) strict-monotonicity-of-justifications :
  ∀ m m′ σ. m ∈ M ∧ σ ∈ Σ ∧ justified m′ m ⟶ justification m′ ⊂ justification
m
 by (metis M-type message-cannot-justify-itself justified-def message-in-state-is-valid
monotonicity-of-justifications psubsetI)

lemma (in Protocol) justification-implies-different-messages :
  ∀ m m′. m ∈ M ∧ m′ ∈ M ⟶ justified m′ m ⟶ m ≠ m′
  using message-cannot-justify-itself by auto

lemma (in Protocol) only-valid-message-is-justified :
  ∀ m ∈ M. ∀ m′. justified m′ m ⟶ m′ ∈ M
  apply (simp add: justified-def)
  using Params.M-type message-in-state-is-valid by blast

lemma (in Protocol) justified-message-exists-in-Mi-n-minus-1 :
  ∀ n m m′. n ∈ ℕ
  ⟶ justified m′ m
  ⟶ m ∈ Mi (V, C, ε) n
  ⟶  m′ ∈ Mi (V, C, ε) (n − 1)
proof −
  have ∀ n m m′. justified m′ m
  ⟶ m ∈ Mi (V, C, ε) n
  ⟶ m ∈ M ∧ m′ ∈ M
  ⟶ m′ ∈ Mi (V, C, ε) (n − 1)
    apply (rule, rule, rule, rule, rule, rule)
  proof −
    fix n m m′
    assume justified m′ m
    assume m ∈ Mi (V, C, ε) n
    assume m ∈ M ∧ m′ ∈ M
    then have justification m ∈ Σi (V,C,ε) n
      using Mi.simps ‹m ∈ Mi (V, C, ε) n› by blast
    then have justification m ∈  Pow (Mi (V,C,ε) (n − 1))
      by (metis (no-types, lifting) Suc-diff-Suc Σi.simps(1) Σi-subset-Mi ‹justified
m′ m› add-leE diff-add diff-le-self empty-iff justified-def neq0-conv plus-1-eq-Suc
singletonD subsetCE)
    show m′ ∈ Mi (V, C, ε) (n − 1)
        using ‹justification m ∈ Pow (Mi (V, C, ε) (n − 1))› ‹justified m′ m›
justified-def by auto
```

```
  qed
  then show ?thesis
    by (metis (no-types, lifting) M-def UN-I only-valid-message-is-justified)
qed

lemma (in Protocol) monotonicity-of-card-of-justification :
  ∀ m m′. m ∈ M
  ⟶ justified m′ m
  ⟶ card (justification m′) < card (justification m)
  by (meson M-type Protocol.strict-monotonicity-of-justifications Protocol-axioms
justification-is-finite psubset-card-mono)


lemma (in Protocol) justification-is-well-founded-on-M :
  wfp-on justified M
proof (rule ccontr)
  assume ¬ wfp-on justified M
  then have ∃f. ∀i. f i ∈ M ∧ justified (f (Suc i)) (f i)
    by (simp add: wfp-on-def)
  then obtain f where ∀i. f i ∈ M ∧ justified (f (Suc i)) (f i) by auto
  have ∀ i. card (justification (f i)) ≤ card (justification (f 0)) − i
    apply (rule)
  proof −
    fix i
    have card (justification (f (Suc i))) < card (justification (f i))
   using ‹∀i. f i ∈ M ∧ justified (f (Suc i)) (f i)› by (simp add: monotonicity-of-card-of-justification)
    show card (justification (f i)) ≤ card (justification (f 0)) − i
      apply (induction i)
      apply simp
      using ‹card (justification (f (Suc i))) < card (justification (f i))›
       by (smt Suc-diff-le ‹∀i. f i ∈ M ∧ justified (f (Suc i)) (f i)› diff-Suc-Suc
diff-is-0-eq le-iff-add less-Suc-eq-le less-imp-le monotonicity-of-card-of-justification
not-less-eq-eq trans-less-add1)
  qed
  then have ∃ i. i = card (justification (f 0)) + Suc 0 ∧ card (justification (f i))
≤ card (justification (f 0)) − i
    by blast
  then show False
     using le-0-eq le-simps(2) linorder-not-le monotonicity-of-card-of-justification
nat-diff-split order-less-imp-le
   by (metis ‹∀i. f i ∈ M ∧ justified (f (Suc i)) (f i)› add.right-neutral add-Suc-right)
qed

lemma (in Protocol) subset-of-M-have-minimal-of-justification :
  ∀ S ⊆ M. S ≠ ∅ ⟶ (∃ m-min ∈ S. ∀ m. justified m m-min ⟶ m ∉ S)
 by (metis justification-is-well-founded-on-M wfp-on-imp-has-min-elt wfp-on-mono)

lemma (in Protocol) message-in-state-is-strict-subset-of-the-state :
  ∀ σ ∈ Σ. ∀ m ∈ σ. justification m ⊂ σ
```

using *justification-implies-different-messages justified-def message-in-state-is-valid state-is-in-pow-Mi* **by** *fastforce*

end

3 Latest Message

theory *LatestMessage*

imports *Main CBCCasper MessageJustification Libraries/LaTeXsugar*

begin

definition *later* :: (*message* ∗ *message set*) ⇒ *message set*
where
later = (λ(*m*, σ). {*m*′ ∈ σ. *justified m m*′})

lemma (**in** *Protocol*) *later-type* :
∀ σ *m*. σ ∈ *Pow M* ∧ *m* ∈ *M* ⟶ *later* (*m*, σ) ⊆ *M*
apply (*simp add*: *later-def*)
by *auto*

lemma (**in** *Protocol*) *later-type-for-state* :
∀ σ *m*. σ ∈ Σ ∧ *m* ∈ *M* ⟶ *later* (*m*, σ) ⊆ *M*
apply (*simp add*: *later-def*)
using *state-is-subset-of-M* **by** *auto*

definition *from-sender* :: (*validator* ∗ *message set*) ⇒ *message set*
where
from-sender = (λ(*v*, σ). {*m* ∈ σ. *sender m* = *v*})

lemma (**in** *Protocol*) *from-sender-type* :
∀ σ *v*. σ ∈ *Pow M* ∧ *v* ∈ *V* ⟶ *from-sender* (*v*, σ) ∈ *Pow M*
apply (*simp add*: *from-sender-def*)
by *auto*

lemma (**in** *Protocol*) *from-sender-type-for-state* :
∀ σ *v*. σ ∈ Σ ∧ *v* ∈ *V* ⟶ *from-sender* (*v*, σ) ⊆ *M*
apply (*simp add*: *from-sender-def*)
using *state-is-subset-of-M* **by** *auto*

lemma (**in** *Protocol*) *messages-from-observed-validator-is-non-empty* :
$\forall\ \sigma\ v.\ \sigma \in \Sigma \wedge v \in \mathit{observed}\ \sigma \longrightarrow \mathit{from\text{-}sender}\ (v,\ \sigma) \neq \emptyset$
apply (*simp add*: *observed-def from-sender-def*)
by *auto*

lemma (**in** *Protocol*) *messages-from-validator-is-finite* :
$\forall\ \sigma\ v.\ \sigma \in \Sigma \wedge v \in V\sigma \longrightarrow \mathit{finite}\ (\mathit{from\text{-}sender}\ (v,\ \sigma))$
by (*simp add*: *from-sender-def state-is-finite*)

definition *from-group* :: (*validator set* $*$ *message set*) $\Rightarrow$ *state*
where
$\mathit{from\text{-}group} = (\lambda(\mathit{v\text{-}set},\ \sigma).\ \{m \in \sigma.\ \mathit{sender}\ m \in \mathit{v\text{-}set}\})$

lemma (**in** *Protocol*) *from-group-type* :
$\forall\ \sigma\ v.\ \sigma \in \mathit{Pow}\ M \wedge \mathit{v\text{-}set} \subseteq V \longrightarrow \mathit{from\text{-}group}\ (\mathit{v\text{-}set},\ \sigma) \in \mathit{Pow}\ M$
apply (*simp add*: *from-group-def*)
by *auto*

lemma (**in** *Protocol*) *from-group-type-for-state* :
$\forall\ \sigma\ v.\ \sigma \in \Sigma \wedge \mathit{v\text{-}set} \subseteq V \longrightarrow \mathit{from\text{-}group}\ (\mathit{v\text{-}set},\ \sigma) \subseteq M$
apply (*simp add*: *from-group-def*)
using *state-is-subset-of-M* **by** *auto*

definition *later-from* :: (*message* $*$ *validator* $*$ *message set*) $\Rightarrow$ *message set*
where
$\mathit{later\text{-}from} = (\lambda(m,\ v,\ \sigma).\ \mathit{later}\ (m,\ \sigma) \cap \mathit{from\text{-}sender}\ (v,\ \sigma))$

lemma (**in** *Protocol*) *later-from-type* :
$\forall\ \sigma\ v\ m.\ \sigma \in \mathit{Pow}\ M \wedge v \in V \wedge m \in M \longrightarrow \mathit{later\text{-}from}\ (m,\ v,\ \sigma) \in \mathit{Pow}\ M$
apply (*simp add*: *later-from-def*)
using *later-type from-sender-type* **by** *auto*

lemma (**in** *Protocol*) *later-from-type-for-state* :
$\forall\ \sigma\ v\ m.\ \sigma \in \Sigma \wedge v \in V \wedge m \in M \longrightarrow \mathit{later\text{-}from}\ (m,\ v,\ \sigma) \subseteq M$
apply (*simp add*: *later-from-def*)
using *later-type-for-state from-sender-type-for-state* **by** *auto*

definition *L-M* :: *message set* $\Rightarrow$ (*validator* $\Rightarrow$ *message set*)
where
$\mathit{L\text{-}M}\ \sigma\ v = \{m \in \mathit{from\text{-}sender}\ (v,\ \sigma).\ \mathit{later\text{-}from}\ (m,\ v,\ \sigma) = \emptyset\}$

lemma (**in** *Protocol*) *L-M-type* :
$\forall\ \sigma\ v.\ \sigma \in \mathit{Pow}\ M \wedge v \in V \longrightarrow \mathit{L\text{-}M}\ \sigma\ v \in \mathit{Pow}\ M$
apply (*simp add*: *L-M-def later-from-def*)
using *from-sender-type* **by** *auto*

```
lemma (in Protocol) L-M-type-for-state :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ⟶ L-M σ v ⊆ M
  apply (simp add: L-M-def later-from-def)
  using from-sender-type-for-state by auto

lemma (in Protocol) L-M-from-non-observed-validator-is-empty :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ∧ v ∉ observed σ ⟶ L-M σ v = ∅
  by (simp add: L-M-def observed-def later-def from-sender-def)

lemma (in Protocol) L-M-is-subset-of-the-state :
  ∀ σ ∈ Σ. ∀ v ∈ V. L-M σ v ⊆ σ
  apply (simp add: L-M-def later-from-def from-sender-def)
  by auto


definition observed-non-equivocating-validators :: state ⇒ validator set
  where
    observed-non-equivocating-validators σ = observed σ − equivocating-validators
σ

lemma (in Protocol) observed-non-equivocating-validators-type :
  ∀ σ ∈ Σ. observed-non-equivocating-validators σ ∈ Pow V
  apply (simp add: observed-non-equivocating-validators-def)
  using observed-type-for-state equivocating-validators-type by auto

lemma (in Protocol) justification-is-well-founded-on-messages-from-validator:
  ∀ σ ∈ Σ. (∀ v ∈ V. wfp-on justified (from-sender (v, σ)))
  using justification-is-well-founded-on-M from-sender-type-for-state wfp-on-subset
by blast

lemma (in Protocol) justification-is-total-on-messages-from-non-equivocating-validator:
  ∀ σ ∈ Σ. (∀ v ∈ V. v ∉ equivocating-validators σ ⟶ Relation.total-on (from-sender
(v, σ)) message-justification)
proof −
   have ∀ m1 m2 σ v. v ∈ V ∧ σ ∈ Σ ∧ {m1, m2} ⊆ from-sender (v, σ) ⟶
sender m1 = sender m2
    by (simp add: from-sender-def)
  then have ∀ σ ∈ Σ. (∀ v ∈ V. v ∉ equivocating-validators σ
       ⟶ (∀ m1 m2. {m1, m2} ⊆ from-sender (v, σ) ⟶ m1 = m2 ∨ justified
m1 m2 ∨ justified m2 m1))
    apply (simp add: equivocating-validators-def is-equivocating-def equivocation-def
from-sender-def observed-def)
    by blast
  then show ?thesis
    apply (simp add: Relation.total-on-def message-justification-def)
    using from-sender-type-for-state by blast
qed
```

```
lemma (in Protocol) justification-is-strict-linear-order-on-messages-from-non-equivocating-validator:
  ∀ σ ∈ Σ. (∀ v ∈ V. v ∉ equivocating-validators σ ⟶ strict-linear-order-on
(from-sender (v, σ)) message-justification)
 by (simp add: strict-linear-order-on-def justification-is-total-on-messages-from-non-equivocating-validator

      irreflexivity-of-justifications transitivity-of-justifications)


lemma (in Protocol) justification-is-strict-well-order-on-messages-from-non-equivocating-validator:
  ∀ σ ∈ Σ. (∀ v ∈ V. v ∉ equivocating-validators σ
   ⟶ strict-linear-order-on (from-sender (v, σ)) message-justification ∧ wfp-on
justified (from-sender (v, σ)))
  using justification-is-well-founded-on-messages-from-validator
      justification-is-strict-linear-order-on-messages-from-non-equivocating-validator

  by blast

lemma (in Protocol) latest-message-is-maximal-element-of-justification :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ⟶ L-M σ v = {m. maximal-on (from-sender (v, σ))
message-justification m}
 apply (simp add: L-M-def later-from-def later-def message-justification-def maximal-on-def)
  using from-sender-type-for-state apply auto
   apply (metis (no-types, lifting) IntI empty-iff from-sender-def mem-Collect-eq
prod.simps(2))
  by blast


lemma (in Protocol) observed-non-equivocating-validators-have-one-latest-message:
  ∀ σ ∈ Σ. (∀ v ∈ observed-non-equivocating-validators σ. is-singleton (L-M σ v))

  apply (simp add: observed-non-equivocating-validators-def)
proof −
  have ∀ σ ∈ Σ. (∀ v ∈ observed σ − equivocating-validators σ. is-singleton {m.
maximal-on (from-sender (v, σ)) message-justification m})
    using
        messages-from-observed-validator-is-non-empty
        messages-from-validator-is-finite
        observed-type-for-state
        equivocating-validators-def
      justification-is-strict-linear-order-on-messages-from-non-equivocating-validator
        strict-linear-order-on-finite-non-empty-set-has-one-maximum
        maximal-and-maximum-coincide-for-strict-linear-order
    by (smt Collect-cong DiffD1 DiffD2 set-mp)
 then show ∀σ∈Σ. ∀v∈observed σ − equivocating-validators σ. is-singleton (L-M
σ v)
    using latest-message-is-maximal-element-of-justification
       observed-non-equivocating-validators-def observed-non-equivocating-validators-type

    by fastforce
```

```
qed
```

```
definition L-E :: state ⇒ validator ⇒ consensus-value set
  where
    L-E σ v = {est m | m. m ∈ L-M σ v}

lemma (in Protocol) L-E-type :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ⟶ L-E σ v ⊆ C
  using M-type Protocol.L-M-type-for-state Protocol-axioms L-E-def by fastforce

lemma (in Protocol) L-E-from-non-observed-validator-is-empty :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ∧ v ∉ observed σ ⟶ L-E σ v = ∅
  using L-E-def L-M-from-non-observed-validator-is-empty by auto
```

```
definition L-H-M :: state ⇒ validator ⇒ message set
  where
    L-H-M σ v = (if v ∈ equivocating-validators σ then ∅ else L-M σ v)

lemma (in Protocol) L-H-M-type :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ⟶ L-H-M σ v ⊆ M
  by (simp add: L-M-type-for-state L-H-M-def)


lemma (in Protocol) L-H-M-of-observed-non-equivocating-validator-is-singleton :
  ∀ σ ∈ Σ. ∀ v ∈ observed-non-equivocating-validators σ.
      is-singleton (L-H-M σ v)
  using observed-non-equivocating-validators-have-one-latest-message
  by (simp add: L-H-M-def observed-non-equivocating-validators-def)


lemma (in Protocol) sender-of-L-H-M:
  ∀ σ ∈ Σ. ∀ v ∈ observed-non-equivocating-validators σ. sender (the-elem (L-H-M
σ v)) = v
    using L-H-M-of-observed-non-equivocating-validator-is-singleton
        L-H-M-def L-M-def from-sender-def
   by (smt Diff-iff is-singleton-the-elem mem-Collect-eq observed-non-equivocating-validators-def
prod.simps(2) singletonI)

lemma (in Protocol) L-H-M-is-in-the-state:
  ∀ σ ∈ Σ. ∀ v ∈ observed-non-equivocating-validators σ. the-elem (L-H-M σ v)
∈ σ
    using L-H-M-of-observed-non-equivocating-validator-is-singleton
        L-H-M-def L-M-is-subset-of-the-state
   by (metis Diff-iff contra-subsetD insert-subset is-singleton-the-elem observed-non-equivocating-validators-def
observed-type-for-state)




definition L-H-E :: state ⇒ validator ⇒ consensus-value set
  where

    L-H-E σ v = est ‘L-H-M σ v

lemma (in Protocol) L-H-E-type :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ⟶ L-H-E σ v ∈ Pow C
  using Protocol.L-E-type Protocol-axioms L-E-def L-H-E-def L-H-M-def
  using M-type L-H-M-type by fastforce

lemma (in Protocol) L-H-E-from-non-observed-validator-is-empty :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ∧ v ∉ observed σ ⟶ L-H-E σ v = ∅
  by (simp add: L-H-E-def L-H-M-def L-M-from-non-observed-validator-is-empty)

lemma image-of-singleton-is-singleton :
  is-singleton A ⟹ is-singleton (f ‘A)
  apply (simp add: is-singleton-def)
  by blast

lemma (in Protocol) L-H-E-of-observed-non-equivocating-validator-is-singleton :
  ∀ σ ∈ Σ. ∀ v ∈ observed-non-equivocating-validators σ.
      is-singleton (L-H-E σ v)
  using L-H-M-of-observed-non-equivocating-validator-is-singleton
  apply (simp add: L-H-E-def)
  using image-of-singleton-is-singleton
  by blast




definition L-H-J :: state ⇒ validator ⇒ state set
  where
    L-H-J σ v = justification ‘L-H-M σ v


lemma (in Protocol) L-H-J-type :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ⟶ L-H-J σ v ⊆ Σ
  using M-type L-H-M-type
      L-H-J-def by auto

lemma (in Protocol) L-H-J-of-observed-non-equivocating-validator-is-singleton :
  ∀ σ ∈ Σ. v ∈ observed-non-equivocating-validators σ
    ⟶ is-singleton (L-H-J σ v)
  using L-H-M-of-observed-non-equivocating-validator-is-singleton
  apply (simp add: L-H-J-def)
  using image-of-singleton-is-singleton
  by blast

lemma (in Protocol) L-H-J-is-subset-of-the-state :
  ∀ σ v. σ ∈ Σ ∧ v ∈ V ⟶ (∀ σ' ∈ L-H-J σ v. σ' ⊂ σ)
  apply (simp add: L-H-J-def
                  L-H-M-def)
  using L-M-is-subset-of-the-state
      message-in-state-is-strict-subset-of-the-state
  by blast


end
theory StateTransition

imports Main CBCCasper MessageJustification

begin




definition (in Params) state-transition :: state rel
  where
    state-transition = {(σ1, σ2). {σ1, σ2} ⊆ Σ ∧ is-future-state(σ1, σ2)}

lemma (in Params) reflexivity-of-state-transition :
  refl-on Σ state-transition
  apply (simp add: state-transition-def refl-on-def)
  by auto

lemma (in Params) transitivity-of-state-transition :
  trans state-transition
  apply (simp add: state-transition-def trans-def)
  by auto

lemma (in Params) state-transition-is-preorder :
  preorder-on Σ state-transition
 by (simp add: preorder-on-def reflexivity-of-state-transition transitivity-of-state-transition)
```

lemma (**in** *Params*) *antisymmetry-of-state-transition* :
 antisym state-transition
 apply (*simp add*: *state-transition-def antisym-def*)
 by *auto*

lemma (**in** *Params*) *state-transition-is-partial-order* :
 partial-order-on Σ *state-transition*
 by (*simp add*: *partial-order-on-def state-transition-is-preorder antisymmetry-of-state-transition*)

definition (**in** *Protocol*) *minimal-transitions* :: (*state* ∗ *state*) *set*
 where
 minimal-transitions ≡ {(σ, σ′) | σ σ′. σ ∈ Σt ∧ σ′ ∈ Σt ∧ *is-future-state* (σ, σ′) ∧ σ ≠ σ′
 ∧ (∄ σ′′. σ′′ ∈ Σ ∧ *is-future-state* (σ, σ′′) ∧ *is-future-state* (σ′′, σ′) ∧ σ ≠ σ′′ ∧ σ′′ ≠ σ′)}

definition *immediately-next-message* **where**
 immediately-next-message = (λ(σ,m). *justification m* ⊆ σ ∧ m ∉ σ)

lemma (**in** *Protocol*) *state-transition-by-immediately-next-message-of-same-depth-non-zero*:

 ∀ n≥*1*. ∀ σ∈Σi (V,C,ε) n. ∀ m∈Mi (V,C,ε) n. *immediately-next-message* (σ,m) ⟶ σ ∪ {m} ∈ Σi (V,C,ε) (n+*1*)
 apply (*rule*, *rule*, *rule*, *rule*, *rule*)
proof−
 fix n σ m
 assume *1* ≤ n σ ∈ Σi (V, C, ε) n m ∈ Mi (V, C, ε) n *immediately-next-message* (σ, m)

 have ∃ n′. n = *Suc* n′
 using ‹*1* ≤ n› *old.nat.exhaust* **by** *auto*
 hence *si*: Σi (V,C,ε) n = {σ ∈ *Pow* (Mi (V,C,ε) (n − *1*)). *finite* σ ∧ (∀ m. m ∈ σ ⟶ *justification* m ⊆ σ)}
 by *force*

 hence Σi (V,C,ε) (n+*1*) = {σ ∈ *Pow* (Mi (V,C,ε) n). *finite* σ ∧ (∀ m. m ∈ σ ⟶ *justification* m ⊆ σ)}
 by *force*

 have *justification* m ⊆ σ
 using *immediately-next-message-def*
 by (*metis* (*no-types*, *lifting*) ‹*immediately-next-message* (σ, m)› *case-prod-conv*)
 hence *justification* m ⊆ σ ∪ {m}
 by *blast*
 moreover have ⋀m′. *finite* σ ∧ m′ ∈ σ ⟹ *justification* m′ ⊆ σ
 using ‹σ ∈ Σi (V, C, ε) n› *si* **by** *blast*
 hence⋀m′. *finite* σ ∧ m′ ∈ σ ⟹ *justification* m′ ⊆ σ ∪ {m}

```
    by auto
  ultimately have ⋀m'. m' ∈ σ ∪ {m} ⟹ justification m ⊆ σ
    using ‹justification m ⊆ σ› by blast

  have {m} ∈ Pow (Mi (V,C,ε) n)
    using ‹m ∈ Mi (V, C, ε) n› by auto
  moreover have σ ∈ Pow (Mi (V,C,ε) (n−1))
    using ‹σ ∈ Σi (V, C, ε) n› si by auto
  hence σ ∈ Pow (Mi (V,C,ε) n)
    using Mi-monotonic
     by (metis (full-types) PowD PowI Suc-eq-plus1 ‹∃ n'. n = Suc n'› diff-Suc-1
subset-iff)
  ultimately have σ ∪ {m} ∈ Pow (Mi (V,C,ε) n)
    by blast

  show σ ∪ {m} ∈ Σi (V, C, ε) (n + 1)
     using ‹⋀m'. finite σ ∧ m' ∈ σ ⟹ justification m' ⊆ σ ∪ {m}› ‹σ ∪ {m} ∈
Pow (Mi (V, C, ε) n)› ‹justification m ⊆ σ ∪ {m}›
    ‹σ ∈ Σi (V, C, ε) n› si by auto
qed

lemma (in Protocol) state-transition-by-immediately-next-message-of-same-depth:

  ∀σ∈Σi (V,C,ε) n. ∀m∈Mi (V,C,ε) n. immediately-next-message (σ,m) ⟶ σ
∪ {m} ∈ Σi (V,C,ε) (n+1)
  apply (cases n)
  apply auto[1]
  using state-transition-by-immediately-next-message-of-same-depth-non-zero
  by (metis le-add1 plus-1-eq-Suc)

lemma (in Params) past-state-exists-in-same-depth :
  ∀ σ σ'. σ' ∈ Σi (V,C,ε) n ⟶ σ ⊆ σ' ⟶ σ ∈ Σ ⟶ σ ∈ Σi (V,C,ε) n
  apply (rule, rule, rule, rule, rule)
proof (cases n)
  case 0
  show ⋀σ σ'. σ' ∈ Σi (V, C, ε) n ⟹ σ ⊆ σ' ⟹ σ ∈ Σ ⟹ n = 0 ⟹ σ ∈
Σi (V, C, ε) n
    by auto
next
  case (Suc nat)
  show ⋀σ σ' nat. σ' ∈ Σi (V, C, ε) n ⟹ σ ⊆ σ' ⟹ σ ∈ Σ ⟹ n = Suc nat
⟹ σ ∈ Σi (V, C, ε) n
  proof −
  fix σ σ'
  assume σ' ∈ Σi (V, C, ε) n
  and σ ⊆ σ'
  and σ ∈ Σ
  have n > 0
    by (simp add: Suc)
  have finite σ ∧ (∀ m. m ∈ σ ⟶ justification m ⊆ σ)
    using ‹σ ∈ Σ› state-is-finite state-is-in-pow-Mi by blast
  moreover have σ ∈ Pow (Mi (V, C, ε) (n − 1))
    using ‹σ ⊆ σ'›
     by (smt Pow-iff Suc-eq-plus1 Σi-monotonic Σi-subset-Mi ‹σ' ∈ Σi (V, C, ε) n› add-diff-cancel-left' add-eq-if diff-is-0-eq diff-le-self plus-1-eq-Suc subset-iff)
  ultimately have  σ ∈ {σ ∈ Pow (Mi (V,C,ε) (n − 1)). finite σ ∧ (∀ m. m ∈ σ ⟶ justification m ⊆ σ)}
    by blast
  then show σ ∈ Σi (V, C, ε) n
    by (simp add: Suc)
  qed
qed

lemma (in Protocol) immediately-next-message-exists-in-same-depth:
  ∀ σ ∈ Σ. ∀ m ∈ M. immediately-next-message (σ,m) ⟶ (∃ n ∈ ℕ. σ ∈ Σi (V,C,ε) n ∧ m ∈ Mi (V,C,ε) n)
  apply (simp add: immediately-next-message-def M-def Σ-def)
  using past-state-exists-in-same-depth
  using Σi-is-subset-of-Σ by blast

lemma (in Protocol) state-transition-by-immediately-next-message:
  ∀ σ ∈Σ. ∀ m ∈ M. immediately-next-message (σ,m) ⟶ σ ∪ {m} ∈ Σ
  apply (rule, rule, rule)
proof −
  fix σ m
  assume σ ∈ Σ
  and m ∈ M
  and immediately-next-message (σ, m)
  then have (∃ n ∈ ℕ. σ ∈ Σi (V,C,ε) n ∧ m ∈ Mi (V,C,ε) n)
    using immediately-next-message-exists-in-same-depth ‹σ ∈ Σ› ‹m ∈ M›
    by blast
  then have ∃ n ∈ ℕ. σ ∪ {m} ∈ Σi (V,C,ε) (n + 1)
    using state-transition-by-immediately-next-message-of-same-depth
    using ‹immediately-next-message (σ, m)› by blast
  show σ ∪ {m} ∈ Σ
    apply (simp add: Σ-def)
     by (metis Nats-1 Nats-add Un-insert-right ‹∃ n∈ℕ. σ ∪ {m} ∈ Σi (V, C, ε) (n + 1)› sup-bot.right-neutral)
qed

lemma (in Protocol) state-transition-imps-immediately-next-message:
  ∀ σ ∈Σ. ∀ m ∈ M. σ ∪ {m} ∈ Σ ∧ m ∉ σ ⟶ immediately-next-message (σ,m)
proof −
  have ∀ σ ∈Σ. ∀ m ∈ M. σ ∪ {m} ∈ Σ ⟶ (∀ m' ∈ σ ∪ {m}. justification m' ⊆ σ ∪ {m})
    using state-is-in-pow-Mi by blast
  then have ∀ σ ∈Σ. ∀ m ∈ M. σ ∪ {m} ∈ Σ ⟶ justification m ⊆ σ ∪ {m}
    by auto
  then have ∀ σ ∈Σ. ∀ m ∈ M. σ ∪ {m} ∈ Σ ∧ m ∉ σ ⟶ justification m ⊆ σ
    using justification-implies-different-messages justified-def by fastforce
  then show ?thesis
    by (simp add: immediately-next-message-def)
qed

lemma (in Protocol) state-transition-only-made-by-immediately-next-message:
 ∀ σ ∈Σ. ∀ m ∈ M. σ ∪ {m} ∈ Σ ∧ m ∉ σ ⟷ immediately-next-message (σ,m)
 using state-transition-imps-immediately-next-message state-transition-by-immediately-next-message
  apply (simp add: immediately-next-message-def)
  by blast

lemma (in Protocol) state-transition-is-immediately-next-message:
 ∀ σ ∈Σ. ∀ m ∈ M. σ ∪ {m} ∈ Σ  ⟷ justification m ⊆ σ
  using state-transition-only-made-by-immediately-next-message
  apply (simp add: immediately-next-message-def)
  using insert-Diff state-is-in-pow-Mi by fastforce

lemma (in Protocol) strict-subset-of-state-have-immediately-next-messages:
 ∀ σ ∈ Σ. ∀ σ'. σ' ⊂ σ ⟶ (∃ m ∈ σ − σ'. immediately-next-message (σ', m))
  apply (simp add: immediately-next-message-def)
  apply (rule, rule, rule)
proof −
  fix σ σ'
  assume σ ∈ Σ
  assume σ' ⊂ σ
  show ∃ m ∈ σ − σ'. justification m ⊆ σ'
  proof (rule ccontr)
    assume ¬ (∃ m ∈ σ − σ'. justification m ⊆ σ')
    then have ∀ m ∈ σ − σ'. ∃ m' ∈ justification m. m' ∈ σ − σ'
      using ‹¬ (∃ m∈σ − σ'. justification m ⊆ σ')› state-is-in-pow-Mi ‹σ' ⊂ σ›
      by (metis Diff-iff ‹σ ∈ Σ› subset-eq)
    then have ∀ m ∈ σ − σ'. ∃ m'. justified m' m ∧ m' ∈ σ − σ'
      using justified-def by auto
    then have ∀ m ∈ σ − σ'. ∃ m'. justified m' m ∧ m' ∈ σ − σ' ∧ m ≠ m'
     using justification-implies-different-messages  state-difference-is-valid-message
      message-in-state-is-valid  ‹σ' ⊂ σ›
      by (meson DiffD1 ‹σ ∈ Σ›)
    have σ − σ' ⊆ M
      using ‹σ ∈ Σ› ‹σ' ⊂ σ› state-is-subset-of-M by auto
    then have ∃ m-min ∈ σ − σ'. ∀ m. justified m m-min ⟶ m ∉ σ − σ'
      using subset-of-M-have-minimal-of-justification ‹σ' ⊂ σ›
      by blast
    then show False
      using ‹∀ m ∈ σ − σ'. ∃ m'. justified m' m ∧ m' ∈ σ − σ'› by blast
  qed
qed

lemma (in Protocol) union-of-two-states-is-state :
  ∀ σ1 ∈ Σ. ∀ σ2 ∈ Σ. (σ1 ∪ σ2) ∈ Σ
  apply (rule, rule)
proof −
  fix σ1 σ2
  assume σ1 ∈ Σ and σ2 ∈ Σ
  show σ1 ∪ σ2 ∈ Σ
  proof (cases σ1 ⊆ σ2)
    case True
    then show ?thesis
      by (simp add: Un-absorb1 ‹σ2 ∈ Σ›)
  next
    case False
    then have ¬ σ1 ⊆ σ2 by simp
    have ∀ σ ∈ Σ. ∀ σ' ∈ Σ. ¬ σ ⊆ σ' ⟶ (∃ m ∈ σ − (σ ∩ σ'). immediately-next-message(σ ∩ σ', m))
     by (metis Int-subset-iff psubsetI strict-subset-of-state-have-immediately-next-messages subsetI)
      then have ∀ σ ∈ Σ. ∀ σ' ∈ Σ. ¬ σ ⊆ σ' ⟶ (∃ m ∈ σ − (σ ∩ σ'). immediately-next-message(σ', m))
      apply (simp add: immediately-next-message-def)
      by blast
    then have ∀ σ ∈ Σ. ∀ σ' ∈ Σ. ¬ σ ⊆ σ' ⟶ (∃ m ∈ σ − σ'. σ' ∪ {m} ∈ Σ)
      using state-transition-by-immediately-next-message
      by (metis DiffD1 DiffD2 DiffI IntI message-in-state-is-valid)
    have ∀ σ ∈ Σ. ∀ σ' ∈ Σ. ¬ σ ⊆ σ' ⟶  σ ∪ σ' ∈ Σ
    proof −
      have ∀ σ ∈ Σ. ∀ σ' ∈ Σ. ¬ σ ⊆ σ' ⟶ card (σ − σ') > 0
        by (meson Diff-eq-empty-iff card-0-eq finite-Diff gr0I state-is-finite)
      have ∀ n. ∀ σ ∈ Σ. ∀ σ' ∈ Σ. ¬ σ ⊆ σ' ∧ Suc n = card (σ − σ')⟶  σ ∪ σ' ∈ Σ
        apply (rule)
      proof −
        fix n
        show ∀σ∈Σ. ∀σ'∈Σ. ¬ σ ⊆ σ' ∧ Suc n = card (σ − σ') ⟶ σ ∪ σ' ∈ Σ
          apply (induction n)
          apply (rule, rule, rule)
        proof −
          fix σ σ'
          assume σ ∈ Σ and σ' ∈ Σ and ¬ σ ⊆ σ' ∧ Suc 0 = card (σ − σ')
          then have is-singleton (σ − σ')
            by (simp add: is-singleton-altdef)
          then have {the-elem (σ − σ')} ∪ σ' ∈ Σ
            using ‹∀ σ ∈ Σ. ∀ σ' ∈ Σ. ¬ σ ⊆ σ' ⟶ (∃ m ∈ σ − σ'. σ' ∪ {m} ∈ Σ)› ‹σ ∈ Σ› ‹σ' ∈ Σ›
                  by (metis Un-commute ‹¬ σ ⊆ σ' ∧ Suc 0 = card (σ − σ')› is-singleton-the-elem singletonD)
          then show σ ∪ σ' ∈ Σ
            by (metis Un-Diff-cancel2 ‹is-singleton (σ − σ')› is-singleton-the-elem)
    next
     show ⋀n. ∀σ∈Σ. ∀σ'∈Σ. ¬ σ ⊆ σ' ∧ Suc n = card (σ − σ') ⟶ σ ∪ σ' ∈ Σ ⟹ ∀σ∈Σ. ∀σ'∈Σ. ¬ σ ⊆ σ' ∧ Suc (Suc n) = card (σ − σ') ⟶ σ ∪ σ' ∈ Σ
      apply (rule, rule, rule)
     proof −
      fix n σ σ'
      assume ∀σ∈Σ. ∀σ'∈Σ. ¬ σ ⊆ σ' ∧ Suc n = card (σ − σ') ⟶ σ ∪ σ' ∈ Σ and σ ∈ Σ and σ' ∈ Σ and ¬ σ ⊆ σ' ∧ Suc (Suc n) = card (σ − σ')
      have ∀ m ∈ σ − σ'. ¬ σ ⊆ σ' ∪ {m} ∧ Suc n = card (σ − (σ' ∪ {m}))
       using ‹¬ σ ⊆ σ' ∧ Suc (Suc n) = card (σ − σ')›
          by (metis Diff-eq-empty-iff Diff-insert Un-insert-right ‹σ ∈ Σ› add-diff-cancel-left' card-0-eq card-Suc-Diff1 finite-Diff nat.simps(3) plus-1-eq-Suc state-is-finite sup-bot.right-neutral)
      have ∃ m ∈ σ − σ'. σ' ∪ {m} ∈ Σ
       using ‹∀ σ ∈ Σ. ∀ σ' ∈ Σ. ¬ σ ⊆ σ' ⟶ (∃ m ∈ σ − σ'. σ' ∪ {m} ∈ Σ)› ‹σ ∈ Σ› ‹σ' ∈ Σ› ‹¬ σ ⊆ σ' ∧ Suc (Suc n) = card (σ − σ')›
       by blast
      then have ∃ m ∈ σ − σ'. σ' ∪ {m} ∈ Σ ∧ ¬ σ ⊆ σ' ∪ {m} ∧ Suc n = card (σ − (σ' ∪ {m}))
       using ‹∀ m ∈ σ − σ'. ¬ σ ⊆ σ' ∪ {m} ∧ Suc n = card (σ − (σ' ∪ {m}))›
       by simp
      then show σ ∪ σ' ∈ Σ
       using ‹∀σ∈Σ. ∀σ'∈Σ. ¬ σ ⊆ σ' ∧ Suc n = card (σ − σ') ⟶ σ ∪ σ' ∈ Σ›
          by (smt Un-Diff-cancel Un-commute Un-insert-right ‹σ ∈ Σ› insert-absorb2 mk-disjoint-insert sup-bot.right-neutral)
     qed
    qed
   qed
   then show ?thesis
    by (meson ‹∀σ∈Σ. ∀σ'∈Σ. ¬ σ ⊆ σ' ⟶ (∃m∈σ − σ'. σ' ∪ {m} ∈ Σ)› card-Suc-Diff1 finite-Diff state-is-finite)
  qed
  then show ?thesis
   using False ‹σ1 ∈ Σ› ‹σ2 ∈ Σ› by blast
 qed
qed


lemma (in Protocol) union-of-finite-set-of-states-is-state :
 ∀ σ-set ⊆ Σ. finite σ-set ⟶ ⋃ σ-set ∈ Σ
 apply auto
proof −
 have ∀ n. ∀ σ-set ⊆ Σ. n = card σ-set ⟶ finite σ-set ⟶ ⋃ σ-set ∈ Σ
  apply (rule)
 proof −
  fix n
  show ∀σ-set⊆Σ. n = card σ-set ⟶ finite σ-set ⟶ ⋃σ-set ∈ Σ
```

```
      apply (induction n)
      apply (rule, rule, rule, rule)
       apply (simp add: empty-set-exists-in-Σ)
      apply (rule, rule, rule, rule)
    proof −
      fix n σ-set
       assume ∀σ-set⊆Σ. n = card σ-set ⟶ finite σ-set ⟶ ⋃σ-set ∈ Σ and
σ-set ⊆ Σ and Suc n = card σ-set and finite σ-set
      then have ∀ σ ∈ σ-set. σ-set − {σ} ⊆ Σ ∧ ⋃ (σ-set − {σ}) ∈ Σ
          using ‹σ-set ⊆ Σ› ‹Suc n = card σ-set› ‹∀σ-set⊆Σ. n = card σ-set ⟶
finite σ-set ⟶ ⋃σ-set ∈ Σ›
        by (metis (mono-tags, lifting) Suc-inject card.remove finite-Diff insert-Diff
insert-subset)
     then have ∀ σ ∈ σ-set. σ-set − {σ} ⊆ Σ ∧ ⋃ (σ-set − {σ}) ∈ Σ ∧ ⋃ (σ-set
− {σ}) ∪ σ ∈ Σ
        using union-of-two-states-is-state ‹σ-set ⊆ Σ› by auto
      then show ⋃σ-set ∈ Σ
          by (metis Sup-bot-conv(1) Sup-insert Un-commute empty-set-exists-in-Σ
insert-Diff)
    qed
  qed
  then show ⋀σ-set. σ-set ⊆ Σ ⟹ finite σ-set ⟹ ⋃σ-set ∈ Σ
    by blast
qed


lemma (in Protocol) state-differences-have-immediately-next-messages:
 ∀ σ ∈ Σ. ∀ σ′∈ Σ. is-future-state (σ, σ′) ∧ σ ≠ σ′ ⟶ (∃ m ∈ σ′−σ. immediately-next-message
(σ, m))
  using strict-subset-of-state-have-immediately-next-messages
  by (simp add: psubsetI)

lemma non-empty-non-singleton-imps-two-elements :
  A ≠ ∅ ⟹ ¬ is-singleton A ⟹ ∃ a1 a2. a1 ≠ a2 ∧ {a1, a2} ⊆ A
  by (metis inf.orderI inf-bot-left insert-subset is-singletonI′)


lemma (in Protocol) minimal-transition-implies-recieving-single-message :
 ∀ σ σ′. (σ, σ′) ∈ minimal-transitions ⟶ is-singleton (σ′− σ)
proof (rule ccontr)
  assume ¬ (∀ σ σ′. (σ, σ′) ∈ minimal-transitions ⟶ is-singleton (σ′− σ))
  then have ∃ σ σ′. (σ, σ′) ∈ minimal-transitions ∧ ¬ is-singleton (σ′− σ)
    by blast
  have ∀ σ σ′. (σ, σ′) ∈ minimal-transitions ⟶
              (∄ σ′′. σ′′ ∈ Σ ∧ is-future-state (σ, σ′′) ∧ is-future-state (σ′′, σ′) ∧ σ
≠ σ′′ ∧ σ′′ ≠ σ′)
    by (simp add: minimal-transitions-def)
  have ∀ σ σ′. (σ, σ′) ∈ minimal-transitions ∧ ¬ is-singleton (σ′− σ)
    ⟶ (∃ m1 m2. {m1, m2} ⊆ M ∧ m1 ∈ σ′− σ ∧ m2 ∈ σ′− σ ∧ m1 ≠ m2 ∧
```

immediately-next-message (σ, *m1*))
apply (*rule*, *rule*, *rule*)
proof −
fix σ σ'
assume (σ, σ') $\in$ *minimal-transitions* $\wedge$ $\neg$ *is-singleton* (σ' − σ)
then have σ' − $\sigma \neq \emptyset$
apply (*simp add*: *minimal-transitions-def*)
by *blast*
have $\sigma' \in \Sigma \wedge \sigma \in \Sigma \wedge$ *is-future-state* (σ, σ')
using ‹(σ, σ') $\in$ *minimal-transitions* $\wedge$ $\neg$ *is-singleton* (σ' − σ)›
by (*simp add*: *minimal-transitions-def* Σ*t-def*)
then have σ' − $\sigma \subseteq M$
using *state-difference-is-valid-message* **by** *auto*
then have $\exists$ *m1 m2*. {*m1*, *m2*} $\subseteq M \wedge$ *m1* $\in \sigma'$ − $\sigma \wedge$ *m2* $\in \sigma'$ − $\sigma \wedge$ *m1* $\neq$ *m2*
using *non-empty-non-singleton-imps-two-elements*
‹(σ, σ') $\in$ *minimal-transitions* $\wedge$ $\neg$ *is-singleton* (σ' − σ)› ‹σ' − $\sigma \neq \emptyset$›
by (*metis* (*full-types*) *contra-subsetD insert-subset subsetI*)
then show $\exists$ *m1 m2*. {*m1*, *m2*} $\subseteq M \wedge$ *m1* $\in \sigma'$ − $\sigma \wedge$ *m2* $\in \sigma'$ − $\sigma \wedge$ *m1* $\neq$ *m2* $\wedge$ *immediately-next-message* (σ, *m1*)
using *state-differences-have-immediately-next-messages*
by (*metis Diff-iff* ‹$\sigma' \in \Sigma \wedge \sigma \in \Sigma \wedge$ *is-future-state* (σ, σ')› *insert-subset message-in-state-is-valid*)
qed
have $\forall$ σ σ'. (σ, σ') $\in$ *minimal-transitions* $\wedge$ $\neg$ *is-singleton* (σ'− σ) $\longrightarrow$
($\exists$ σ''. $\sigma'' \in \Sigma \wedge$ *is-future-state* (σ, σ'') $\wedge$ *is-future-state* (σ'', σ') $\wedge$ $\sigma \neq \sigma'' \wedge \sigma'' \neq \sigma'$)
apply (*rule*, *rule*, *rule*)
proof −
fix σ σ'
assume (σ, σ') $\in$ *minimal-transitions* $\wedge$ $\neg$ *is-singleton* (σ' − σ)
then have $\exists$ *m1 m2*. {*m1*, *m2*} $\subseteq M \wedge$ *m1* $\in \sigma'$− $\sigma \wedge$ *m2* $\in \sigma'$− $\sigma \wedge$ *m1* $\neq$ *m2* $\wedge$ *immediately-next-message* (σ, *m1*)
using ‹$\forall$ σ σ'. (σ, σ') $\in$ *minimal-transitions* $\wedge$ $\neg$ *is-singleton* (σ'− σ)
$\longrightarrow$ ($\exists$ *m1 m2*. {*m1*, *m2*} $\subseteq M \wedge$ *m1* $\in \sigma'$− $\sigma \wedge$ *m2* $\in \sigma'$− $\sigma \wedge$ *m1* $\neq$ *m2* $\wedge$ *immediately-next-message* (σ, *m1*))›
by *simp*
then obtain *m1 m2* **where** {*m1*, *m2*} $\subseteq M \wedge$ *m1* $\in \sigma'$− $\sigma \wedge$ *m2* $\in \sigma'$− $\sigma \wedge$ *m1* $\neq$ *m2* $\wedge$ *immediately-next-message* (σ, *m1*)
by *auto*
have $\sigma \in \Sigma \wedge \sigma' \in \Sigma$
using ‹(σ, σ') $\in$ *minimal-transitions* $\wedge$ $\neg$ *is-singleton* (σ' − σ)›
by (*simp add*: *minimal-transitions-def* Σ*t-def*)
then have $\sigma \cup$ {*m1*} $\in \Sigma$
using ‹{*m1*, *m2*} $\subseteq M \wedge$ *m1* $\in \sigma'$− $\sigma \wedge$ *m2* $\in \sigma'$− $\sigma \wedge$ *m1* $\neq$ *m2* $\wedge$ *immediately-next-message* (σ, *m1*)›
state-transition-by-immediately-next-message
by *simp*
have *is-future-state* (σ, $\sigma \cup$ {*m1*}) $\wedge$ *is-future-state* ($\sigma \cup$ {*m1*}, σ')

using ‹$(\sigma, \sigma') \in$ *minimal-transitions* $\wedge \neg$ *is-singleton* $(\sigma' - \sigma)$› ‹$\{m1, m2\} \subseteq M \wedge m1 \in \sigma' - \sigma \wedge m2 \in \sigma' - \sigma \wedge m1 \neq m2 \wedge$ *immediately-next-message* $(\sigma, m1)$› *minimal-transitions-def* **by** *auto*

have $\sigma \neq \sigma \cup \{m1\} \wedge \sigma \cup \{m1\} \neq \sigma'$

using ‹$\{m1, m2\} \subseteq M \wedge m1 \in \sigma' - \sigma \wedge m2 \in \sigma' - \sigma \wedge m1 \neq m2 \wedge$ *immediately-next-message* $(\sigma, m1)$› **by** *auto*

then show $\exists \sigma''. \sigma'' \in \Sigma \wedge$ *is-future-state* $(\sigma, \sigma'') \wedge$ *is-future-state* $(\sigma'', \sigma') \wedge \sigma \neq \sigma'' \wedge \sigma'' \neq \sigma'$

using ‹$\sigma \cup \{m1\} \in \Sigma$› ‹*is-future-state* $(\sigma, \sigma \cup \{m1\}) \wedge$ *is-future-state* $(\sigma \cup \{m1\}, \sigma')$›

by *auto*

qed

then show *False*

using ‹$\forall \sigma\ \sigma'. (\sigma, \sigma') \in$ *minimal-transitions* $\longrightarrow (\nexists \sigma''. \sigma'' \in \Sigma \wedge$ *is-future-state* $(\sigma, \sigma'') \wedge$ *is-future-state* $(\sigma'', \sigma') \wedge \sigma \neq \sigma'' \wedge \sigma'' \neq \sigma')$› ‹$\neg (\forall \sigma\ \sigma'. (\sigma, \sigma') \in$ *minimal-transitions* $\longrightarrow$ *is-singleton* $(\sigma' - \sigma))$› **by** *blast*

qed

lemma (**in** *Protocol*) *minimal-transitions-reconstruction* :

$\forall\ \sigma\ \sigma'. (\sigma, \sigma') \in$ *minimal-transitions* $\longrightarrow \sigma \cup \{$*the-elem* $(\sigma' - \sigma)\} = \sigma'$

apply (*rule*, *rule*, *rule*)

proof −

fix $\sigma\ \sigma'$

assume $(\sigma, \sigma') \in$ *minimal-transitions*

then have *is-singleton* $(\sigma' - \sigma)$

using *minimal-transitions-def minimal-transition-implies-recieving-single-message* **by** *auto*

then have $\sigma \subseteq \sigma'$

using ‹$(\sigma, \sigma') \in$ *minimal-transitions*› *minimal-transitions-def* **by** *auto*

then show $\sigma \cup \{$*the-elem* $(\sigma' - \sigma)\} = \sigma'$

by (*metis Diff-partition* ‹*is-singleton* $(\sigma' - \sigma)$› *is-singleton-the-elem*)

qed

lemma (**in** *Protocol*) *road-to-future-state* :

$\forall\ \sigma\ \sigma'. \sigma \in \Sigma \wedge \sigma' \in \Sigma \wedge$ *is-future-state*(σ, σ')

$\longrightarrow n =$ *card* $(\sigma' - \sigma)$

$\longrightarrow (\exists\ f. f\ 0 = \sigma \wedge f\ n = \sigma' \wedge (\forall\ \ i.\ 0 \leq i \wedge i \leq n - 1 \longrightarrow f\ i \in \Sigma \wedge (\exists\ \ m \in M. f\ i \cup \{m\} = f\ (Suc\ i))))$

apply (*rule*, *rule*, *rule*, *rule*)

oops

end

4 Safety Proof

theory *ConsensusSafety*

imports *Main CBCCasper MessageJustification StateTransition Libraries/LaTeXsugar*

```
begin



definition (in Protocol) futures :: state ⇒ state set
  where
    futures σ = {σ' ∈ Σt. is-future-state (σ, σ')}


lemma (in Protocol) monotonic-futures :
  ∀ σ' σ. σ' ∈ Σt ∧ σ ∈ Σt
   ⟶ σ' ∈ futures σ ⟷ futures σ' ⊆ futures σ
  apply (simp add: futures-def) by auto


theorem (in Protocol) two-party-common-futures :
  ∀ σ1 σ2. σ1 ∈ Σt ∧ σ2 ∈ Σt
  ⟶ is-faults-lt-threshold (σ1 ∪ σ2)
  ⟶ futures σ1 ∩ futures σ2 ≠ ∅
  apply (simp add: futures-def Σt-def) using union-of-two-states-is-state
  by blast


theorem (in Protocol) n-party-common-futures :
  ∀ σ-set. σ-set ⊆ Σt
  ⟶ finite σ-set
  ⟶ is-faults-lt-threshold (⋃ σ-set)
  ⟶ ⋂ {futures σ | σ. σ ∈ σ-set} ≠ ∅
  apply (simp add: futures-def Σt-def) using union-of-finite-set-of-states-is-state
  by blast

lemma (in Protocol) n-party-common-futures-exists :
  ∀ σ-set. σ-set ⊆ Σt
  ⟶ finite σ-set
  ⟶ is-faults-lt-threshold (⋃ σ-set)
  ⟶ (∃ σ ∈Σt. σ ∈ ⋂ {futures σ | σ. σ ∈ σ-set})
  apply (simp add: futures-def Σt-def) using union-of-finite-set-of-states-is-state
  by blast



definition (in Protocol) state-property-is-decided :: (state-property ∗ state) ⇒ bool
  where
    state-property-is-decided = (λ(p, σ). (∀ σ' ∈ futures σ . p σ'))
```

```
lemma (in Protocol) forward-consistency :
  ∀ σ' σ. σ' ∈ Σt ∧ σ ∈ Σt
  ⟶ σ' ∈ futures σ
  ⟶ state-property-is-decided (p, σ)
  ⟶ state-property-is-decided (p, σ')
  apply (simp add: futures-def state-property-is-decided-def)
  by auto
```

```
fun state-property-not :: state-property ⇒ state-property
  where
    state-property-not p = (λσ. (¬ p σ))
```

```
lemma (in Protocol) backword-consistency :
  ∀ σ' σ. σ' ∈ Σt ∧ σ ∈ Σt
  ⟶ σ' ∈ futures σ
  ⟶ state-property-is-decided (p, σ')
  ⟶ ¬state-property-is-decided (state-property-not p, σ)
  apply (simp add: futures-def state-property-is-decided-def)
  by auto
```

```
theorem (in Protocol) two-party-consensus-safety-for-state-property :
  ∀ σ1 σ2. σ1 ∈ Σt ∧ σ2 ∈ Σt
  ⟶ is-faults-lt-threshold (σ1 ∪ σ2)
 ⟶ ¬(state-property-is-decided (p, σ1) ∧ state-property-is-decided (state-property-not p, σ2))
  apply (simp add: state-property-is-decided-def)
  using two-party-common-futures
  by (metis Int-emptyI)
```

```
definition (in Protocol) state-properties-are-inconsistent :: state-property set ⇒ bool
  where
    state-properties-are-inconsistent p-set = (∀ σ ∈ Σ. ¬ (∀ p ∈ p-set. p σ))
```

```
definition (in Protocol) state-properties-are-consistent :: state-property set ⇒ bool
  where
    state-properties-are-consistent p-set = (∃ σ ∈ Σ. ∀ p ∈ p-set. p σ)
```

```
definition (in Protocol) state-property-decisions :: state ⇒ state-property set
  where
    state-property-decisions σ = {p. state-property-is-decided (p, σ)}
```

```
theorem (in Protocol) n-party-safety-for-state-properties :
  ∀ σ-set. σ-set ⊆ Σt
  ⟶ finite σ-set
  ⟶ is-faults-lt-threshold (⋃ σ-set)
  ⟶ state-properties-are-consistent (⋃ {state-property-decisions σ | σ. σ ∈ σ-set})
  apply rule+
proof−
  fix σ-set
  assume σ-set: σ-set ⊆ Σt
  and finite σ-set
  and is-faults-lt-threshold (⋃ σ-set)
  hence ∃ σ∈Σt. σ ∈ ⋂ {futures σ | σ. σ ∈ σ-set}
    using n-party-common-futures-exists by simp
  hence ∃ σ∈Σt. ∀ s∈σ-set. σ ∈ futures s
    by blast
  hence ∃ σ∈Σt. (∀ s∈σ-set. σ ∈ futures s) ∧ (∀ s∈σ-set. σ ∈ futures s ⟶ (∀ p.
state-property-is-decided (p,s) ⟶ state-property-is-decided (p,σ)))
    by (simp add: subset-eq state-property-is-decided-def futures-def)
 hence ∃ σ∈Σt. ∀ s∈σ-set. (∀ p. state-property-is-decided (p,s) ⟶ state-property-is-decided
(p,σ))
    by blast
 hence ∃ σ∈Σt. ∀ s∈σ-set. (∀ p ∈ state-property-decisions s. state-property-is-decided
(p,σ))
    by (simp add: state-property-decisions-def)
 hence ∃ σ∈Σt. ∀ p∈⋃ {state-property-decisions σ | σ. σ ∈ σ-set}. state-property-is-decided
(p,σ)
  proof−
   obtain σ where σ ∈ Σt ∀ s∈σ-set. (∀ p ∈ state-property-decisions s. state-property-is-decided
(p,σ))
     using ‹∃ σ∈Σt. ∀ s∈σ-set. ∀ p∈state-property-decisions s. state-property-is-decided
(p, σ)› by blast
   have ∀ p∈⋃ {state-property-decisions σ | σ. σ ∈ σ-set}. state-property-is-decided
(p,σ)
      using ‹∀ s∈σ-set. ∀ p∈state-property-decisions s. state-property-is-decided (p,
σ)› by fastforce
    thus ?thesis
      using ‹σ ∈ Σt› by blast
  qed
  hence ∃ σ∈Σt. ∀ p∈⋃ {state-property-decisions σ | σ. σ ∈ σ-set}. ∀ σ'∈futures
σ. p σ'
   by (simp add: state-property-decisions-def futures-def state-property-is-decided-def)
 show state-properties-are-consistent (⋃ {state-property-decisions σ |σ. σ ∈ σ-set})
    unfolding state-properties-are-consistent-def
    by (metis (mono-tags, lifting) Σt-def ‹∃ σ∈Σt. ∀ p∈⋃ {state-property-decisions
σ |σ. σ ∈ σ-set}. ∀ σ'∈futures σ. p σ'› mem-Collect-eq monotonic-futures order-refl)
qed
```

```
definition (in Protocol) naturally-corresponding-state-property :: consensus-value-property
⇒ state-property
  where
    naturally-corresponding-state-property q = (λσ. ∀ c ∈ ε σ. q c)


definition (in Protocol) consensus-value-properties-are-consistent :: consensus-value-property
set ⇒ bool
  where
    consensus-value-properties-are-consistent q-set = (∃ c ∈ C. ∀ q ∈ q-set. q c)


lemma (in Protocol) naturally-corresponding-consistency :
  ∀ q-set. state-properties-are-consistent {naturally-corresponding-state-property q
| q. q ∈ q-set}
  ⟶ consensus-value-properties-are-consistent q-set
  apply (rule, rule)
proof −
  fix q-set
  have
     state-properties-are-consistent {naturally-corresponding-state-property q | q. q
∈ q-set}
    ⟶ (∃ σ ∈ Σ. ∀ p ∈ {λσ'. ∀ c ∈ ε σ'. q c | q. q ∈ q-set}. p σ)
   by (simp add: naturally-corresponding-state-property-def state-properties-are-consistent-def)
  moreover have
    (∃ σ ∈ Σ. ∀ p ∈ {λσ'. ∀ c ∈ ε σ'. q c | q. q ∈ q-set}. p σ)
    ⟶ (∃ σ ∈ Σ. ∀ q' ∈ q-set. (λσ'. ∀ c ∈ ε σ'. q' c) σ)
    by (metis (mono-tags, lifting) mem-Collect-eq)
  moreover have
    (∃ σ ∈ Σ. ∀ q ∈ q-set. (λσ'. ∀ c ∈ ε σ'. q c) σ)
    ⟶ (∃ σ ∈ Σ. ∀ q' ∈ q-set. ∀ c ∈ ε σ. q' c)
    by blast
  moreover have
    (∃ σ ∈ Σ. ∀ q ∈ q-set. ∀ c ∈ ε σ. q c)
    ⟶ (∃ σ ∈ Σ. ∀ c ∈ ε σ. ∀ q' ∈ q-set. q' c)
    by blast
  moreover have
    (∃ σ ∈ Σ. ∀ c ∈ ε σ. ∀ q ∈ q-set. q c)
    ⟶ (∃ σ ∈ Σ. ∃ c ∈ ε σ. ∀ q' ∈ q-set. q' c)
    by (meson all-not-in-conv estimates-are-non-empty)
  moreover have
    (∃ σ ∈ Σ. ∃ c ∈ ε σ. ∀ q ∈ q-set. q c)
    ⟶ (∃ c ∈ C. ∀ q' ∈ q-set. q' c)
    using is-valid-estimator-def ε-type by fastforce
  ultimately show
    state-properties-are-consistent {naturally-corresponding-state-property q |q. q ∈
q-set}
    ⟹ consensus-value-properties-are-consistent q-set
```

by (*simp add*: *consensus-value-properties-are-consistent-def*)
qed

definition (**in** *Protocol*) *consensus-value-property-is-decided* :: (*consensus-value-property* $*$ *state*) $\Rightarrow$ *bool*
where
consensus-value-property-is-decided
$=$ (λ(q, σ). *state-property-is-decided* (*naturally-corresponding-state-property* q, σ))

definition (**in** *Protocol*) *consensus-value-property-decisions* :: *state* $\Rightarrow$ *consensus-value-property set*
where
consensus-value-property-decisions σ $=$ $\{q$. *consensus-value-property-is-decided* (q, σ)$\}$

theorem (**in** *Protocol*) *n-party-safety-for-consensus-value-properties* :
$\forall$ σ*-set*. σ*-set* $\subseteq$ Σt
$\longrightarrow$ *finite* σ*-set*
$\longrightarrow$ *is-faults-lt-threshold* ($\bigcup$ σ*-set*)
$\longrightarrow$ *consensus-value-properties-are-consistent* ($\bigcup$ $\{$*consensus-value-property-decisions* σ $\mid$ σ. σ $\in$ σ*-set*$\}$)
apply (*rule*, *rule*, *rule*, *rule*)
proof $-$
fix σ*-set*
assume σ*-set* $\subseteq$ Σt
and *finite* σ*-set*
and *is-faults-lt-threshold* ($\bigcup$ σ*-set*)
hence *state-properties-are-consistent* ($\bigcup$ $\{$*state-property-decisions* σ $\mid$ σ. σ $\in$ σ*-set*$\}$)
using ‹σ*-set* $\subseteq$ Σt› *n-party-safety-for-state-properties* **by** *auto*
hence *state-properties-are-consistent* $\{p \in \bigcup$ $\{$*state-property-decisions* σ $\mid$ σ. σ $\in$ σ*-set*$\}$. $\exists$ q. p $=$ *naturally-corresponding-state-property* $q\}$
unfolding *naturally-corresponding-state-property-def state-properties-are-consistent-def*
apply (*simp*)
by *meson*
hence *state-properties-are-consistent* $\{$*naturally-corresponding-state-property* q $\mid$ q. *naturally-corresponding-state-property* q $\in$ $\bigcup$ $\{$*state-property-decisions* σ $\mid$ σ. σ $\in$ σ*-set*$\}\}$
by (*smt Collect-cong*)
hence *consensus-value-properties-are-consistent* $\{q$. *naturally-corresponding-state-property* q $\in$ $\bigcup$ $\{$*state-property-decisions* σ $\mid$ σ. σ $\in$ σ*-set*$\}\}$
using *naturally-corresponding-consistency*
proof $-$
show *?thesis*
by (*metis* (*no-types*) *Setcompr-eq-image* ‹$\forall$ *q-set*. *state-properties-are-consistent*

```
{naturally-corresponding-state-property q |q. q ∈ q-set} ⟶ consensus-value-properties-are-consistent
q-set› ‹state-properties-are-consistent {naturally-corresponding-state-property q |q.
naturally-corresponding-state-property q ∈ ⋃{state-property-decisions σ |σ. σ ∈
σ-set}}› setcompr-eq-image)
  qed
 hence consensus-value-properties-are-consistent (⋃ {consensus-value-property-decisions
σ | σ. σ ∈ σ-set})
   apply (simp add: consensus-value-property-decisions-def consensus-value-property-is-decided-def
state-property-decisions-def consensus-value-properties-are-consistent-def)
    by (metis mem-Collect-eq)
  thus
   consensus-value-properties-are-consistent (⋃ {consensus-value-property-decisions
σ | σ. σ ∈ σ-set})
    by simp
qed

fun consensus-value-property-not :: consensus-value-property ⇒ consensus-value-property
  where
    consensus-value-property-not p = (λc. (¬ p c))

lemma (in Protocol) negation-is-not-decided-by-other-validator :
  ∀ σ-set. σ-set ⊆ Σt
  ⟶ finite σ-set
  ⟶ is-faults-lt-threshold (⋃ σ-set)
  ⟶ (∀ σ σ' p. {σ, σ'} ⊆ σ-set ∧ p ∈ consensus-value-property-decisions σ
          ⟶ consensus-value-property-not p ∉ consensus-value-property-decisions
σ')
  apply (rule, rule, rule, rule, rule, rule, rule, rule)
proof –
  fix σ-set σ σ' p
  assume σ-set ⊆ Σt and finite σ-set and is-faults-lt-threshold (⋃σ-set) and {σ,
σ'} ⊆ σ-set ∧ p ∈ consensus-value-property-decisions σ
  hence ∃ σ. σ ∈ Σt ∧ σ ∈ ⋂ {futures σ | σ. σ ∈ σ-set}
    using n-party-common-futures-exists by meson
  then obtain σ'' where σ'' ∈ Σt ∧ σ'' ∈ ⋂ {futures σ | σ. σ ∈ σ-set} by auto
  hence state-property-is-decided (naturally-corresponding-state-property p, σ'')
   using ‹{σ, σ'} ⊆ σ-set ∧ p ∈ consensus-value-property-decisions σ› consensus-value-property-decisions-def
consensus-value-property-is-decided-def
    using ‹σ-set ⊆ Σt› forward-consistency by fastforce
  have σ'' ∈ futures σ'
    using ‹σ'' ∈ Σt ∧ σ'' ∈ ⋂ {futures σ | σ. σ ∈ σ-set}› ‹{σ, σ'} ⊆ σ-set ∧ p ∈
consensus-value-property-decisions σ›
    by auto
 hence ¬ state-property-is-decided (state-property-not (naturally-corresponding-state-property
p), σ')

    using backword-consistency ‹state-property-is-decided (naturally-corresponding-state-property
p, σ'')›
     using ‹σ'' ∈ Σt ∧ σ'' ∈ ⋂-Collect (futures σ) (σ ∈ σ-set)› ‹σ-set ⊆ Σt› ‹{σ,
σ'} ⊆ σ-set ∧ p ∈ consensus-value-property-decisions σ› by auto
  then show consensus-value-property-not p ∉ consensus-value-property-decisions
σ'
   apply (simp add: consensus-value-property-decisions-def consensus-value-property-is-decided-def
naturally-corresponding-state-property-def state-property-is-decided-def)
    using Σt-def estimates-are-non-empty futures-def by fastforce
qed


lemma (in Protocol) n-party-consensus-safety :
  ∀ σ-set. σ-set ⊆ Σt
  ⟶ finite σ-set
  ⟶ is-faults-lt-threshold (⋃ σ-set)
  ⟶ (∀ p ∈ ⋃ {consensus-value-property-decisions σ' | σ'. σ' ∈ σ-set}.
        (λc. (¬ p c)) ∉ ⋃ {consensus-value-property-decisions σ' | σ'. σ' ∈ σ-set})
  apply (rule, rule, rule, rule, rule, rule)
proof −
  fix σ-set p
  assume σ-set ⊆ Σt and finite σ-set and is-faults-lt-threshold (⋃σ-set) and p
∈ ⋃ {consensus-value-property-decisions σ' | σ'. σ' ∈ σ-set}
  and (λc. (¬ p c)) ∈ ⋃ {consensus-value-property-decisions σ' | σ'. σ' ∈ σ-set}
  hence ∃ σ. σ ∈ Σt ∧ σ ∈ ⋂ {futures σ | σ. σ ∈ σ-set}
    using n-party-common-futures-exists by meson
  then obtain σ'' where σ'' ∈ Σt ∧ σ'' ∈ ⋂ {futures σ | σ. σ ∈ σ-set} by auto
  hence state-property-is-decided (naturally-corresponding-state-property p, σ'')
   using ‹p ∈ ⋃ {consensus-value-property-decisions σ' | σ'. σ' ∈ σ-set}› consensus-value-property-decisions-de
consensus-value-property-is-decided-def
    using ‹σ-set ⊆ Σt› forward-consistency by fastforce
  have state-property-is-decided (naturally-corresponding-state-property (λc. (¬ p
c)), σ'')
     using ‹(λc. (¬ p c)) ∈ ⋃ {consensus-value-property-decisions σ' | σ'. σ' ∈
σ-set}› consensus-value-property-decisions-def consensus-value-property-is-decided-def

    using ‹σ-set ⊆ Σt› forward-consistency ‹σ'' ∈ Σt ∧ σ'' ∈ ⋂ {futures σ | σ. σ
∈ σ-set}› by fastforce
  then show False
    using ‹state-property-is-decided (naturally-corresponding-state-property p, σ'')›
   apply (simp add: state-property-is-decided-def naturally-corresponding-state-property-def)
     by (meson Σt-is-subset-of-Σ ‹σ'' ∈ Σt ∧ σ'' ∈ ⋂-Collect (futures σ) (σ ∈
σ-set)› estimates-are-non-empty monotonic-futures order-refl subsetCE)
qed


lemma (in Protocol) two-party-consensus-safety-for-consensus-value-property :
  ∀ σ1 σ2. σ1 ∈ Σt ∧ σ2 ∈ Σt
  ⟶ is-faults-lt-threshold (σ1 ∪ σ2)
  ⟶ consensus-value-property-is-decided (p, σ1)
  ⟶ ¬ consensus-value-property-is-decided (consensus-value-property-not p, σ2)
  apply (rule, rule, rule, rule, rule)
```

```
proof –
  fix σ1 σ2
  have two-party: ∀ σ1 σ2. {σ1, σ2} ⊆ Σt
        ⟶ is-faults-lt-threshold (⋃ {σ1, σ2})
        ⟶ p ∈ consensus-value-property-decisions σ1
           ⟶ consensus-value-property-not p ∉ consensus-value-property-decisions
σ2
    using negation-is-not-decided-by-other-validator
    by (meson finite.emptyI finite.insertI order-refl)
  assume σ1 ∈ Σt ∧ σ2 ∈ Σt and is-faults-lt-threshold (σ1 ∪ σ2) and consensus-value-property-is-decided
(p, σ1)
  then show ¬ consensus-value-property-is-decided (consensus-value-property-not
p, σ2)
    using two-party
    apply (simp add: consensus-value-property-decisions-def)
    by blast
qed

lemma (in Protocol) n-party-consensus-safety-for-power-set-of-decisions :
  ∀ σ-set. σ-set ⊆ Σt
  ⟶ finite σ-set
  ⟶ is-faults-lt-threshold (⋃ σ-set)
  ⟶ (∀ σ p-set. σ ∈ σ-set ∧ p-set ∈ Pow (⋃ {consensus-value-property-decisions
σ′ | σ′. σ′ ∈ σ-set}) – {∅}
      ⟶ (λc. ¬ (∀ p ∈ p-set. p c)) ∉ consensus-value-property-decisions σ)
  apply (rule, rule, rule, rule, rule, rule, rule, rule)
proof –
  fix σ-set σ p-set
  assume σ-set ⊆ Σt and finite σ-set and is-faults-lt-threshold (⋃σ-set)
  and σ ∈ σ-set ∧ p-set ∈ Pow (⋃ {consensus-value-property-decisions σ′ | σ′. σ′
∈ σ-set}) – {∅}
  and (λc. ¬ (∀ p ∈ p-set. p c)) ∈ consensus-value-property-decisions σ
  hence ∃ σ. σ ∈ Σt ∧ σ ∈ ⋂ {futures σ | σ. σ ∈ σ-set}
    using n-party-common-futures-exists by meson
  then obtain σ′ where σ′ ∈ Σt ∧ σ′ ∈ ⋂ {futures σ | σ. σ ∈ σ-set} by auto
  hence ∀ p ∈ p-set. ∃ σ″ ∈ σ-set. state-property-is-decided (naturally-corresponding-state-property
p, σ″)
    using ‹σ ∈ σ-set ∧ p-set ∈ Pow (⋃ {consensus-value-property-decisions σ′ |
σ′. σ′ ∈ σ-set}) – {∅}›
   apply (simp add: consensus-value-property-decisions-def consensus-value-property-is-decided-def)
    by blast
  have ∀ σ″ ∈ σ-set. σ′ ∈ futures σ″
    using ‹σ′ ∈ Σt ∧ σ′ ∈ ⋂-Collect (futures σ) (σ ∈ σ-set)› by blast
  hence ∀ p ∈ p-set. state-property-is-decided (naturally-corresponding-state-property
p, σ′)
    using forward-consistency ‹∀ p ∈ p-set. ∃ σ″ ∈ σ-set. state-property-is-decided
(naturally-corresponding-state-property p, σ″)›
    by (meson ‹σ′ ∈ Σt ∧ σ′ ∈ ⋂-Collect (futures σ) (σ ∈ σ-set)› ‹σ-set ⊆ Σt›
subsetCE)
```

hence *state-property-is-decided* (*naturally-corresponding-state-property* (*λc.* ∀ *p* ∈ *p-set. p c*), *σ′*)
apply (*simp add*: *naturally-corresponding-state-property-def state-property-is-decided-def*)
by *auto*
then show *False*
using ‹(*λc.* ¬ (∀ *p* ∈ *p-set. p c*)) ∈ *consensus-value-property-decisions σ*›
apply (*simp add*: *consensus-value-property-decisions-def consensus-value-property-is-decided-def naturally-corresponding-state-property-def state-property-is-decided-def*)
using Σ*t-is-subset-of-*Σ ‹*σ* ∈ *σ-set* ∧ *p-set* ∈ *Pow* (⋃*-Collect* (*consensus-value-property-decisions σ′*) (*σ′* ∈ *σ-set*)) − {∅}› ‹*σ′* ∈ Σ*t* ∧ *σ′* ∈ ⋂*-Collect* (*futures σ*) (*σ* ∈ *σ-set*)› *estimates-are-non-empty monotonic-futures* **by** *fastforce*
qed

end
theory *SafetyOracle*

imports *Main CBCCasper LatestMessage StateTransition ConsensusSafety*

begin

definition *agreeing-validators* :: (*consensus-value-property* ∗ *state*) ⇒ *validator set*
where
agreeing-validators = (*λ*(*p*, *σ*).{*v* ∈ *observed-non-equivocating-validators σ*. ∀ *c* ∈ *L-H-E σ v. p c*})

definition *is-agreeing* :: (*consensus-value-property* ∗ *state* ∗ *validator*) ⇒ *bool*
where
is-agreeing = (*λ*(*p*, *σ*, *v*). ∀ *c* ∈ *L-H-E σ v. p c*)

lemma (**in** *Protocol*) *agreeing-validators-type* :

$\forall\ \sigma \in \Sigma.$ *agreeing-validators* $(p,\ \sigma) \subseteq V$
apply (*simp add*: *observed-non-equivocating-validators-def agreeing-validators-def*)
using *observed-type-for-state* **by** *auto*

lemma (**in** *Protocol*) *agreeing-validators-finite* :
$\forall\ \sigma \in \Sigma.$ *finite* (*agreeing-validators* $(p,\ \sigma)$)
by (*meson V-type agreeing-validators-type rev-finite-subset*)

definition *disagreeing-validators* :: (*consensus-value-property* $*$ *state*) $\Rightarrow$ *validator set*
where
disagreeing-validators $=$ ($\lambda(p,\ \sigma)$. $\{v \in$ *observed-non-equivocating-validators* $\sigma.$ $\exists\ c \in$ *L-H-E* $\sigma\ v.\ \neg\ p\ c\}$)

lemma (**in** *Protocol*) *disagreeing-validators-type* :
$\forall\ \sigma \in \Sigma.$ *disagreeing-validators* $(p,\ \sigma) \subseteq V$
apply (*simp add*: *observed-non-equivocating-validators-def disagreeing-validators-def*)
using *observed-type-for-state* **by** *auto*

definition (**in** *Params*) *is-majority* :: (*validator set* $*$ *state*) $\Rightarrow$ *bool*
where
is-majority $=$ (λ(*v-set*, σ). (*weight-measure v-set* $>$ (*weight-measure* ($V -$ *equivocating-validators* σ)) *div 2*))

definition (**in** *Protocol*) *is-majority-driven* :: *consensus-value-property* $\Rightarrow$ *bool*
where
is-majority-driven $p = (\forall\ \sigma\ c.\ \sigma \in \Sigma \wedge c \in C \wedge$ *is-majority* (*agreeing-validators* $(p,\ \sigma),\ \sigma) \longrightarrow (\forall\ c \in \varepsilon\ \sigma.\ p\ c))$

definition (**in** *Protocol*) *is-max-driven* :: *consensus-value-property* $\Rightarrow$ *bool*
where
is-max-driven $p =$
$(\forall\ \sigma\ c.\ \sigma \in \Sigma \wedge c \in C \wedge$ *weight-measure* (*agreeing-validators* $(p,\ \sigma)) >$ *weight-measure* (*disagreeing-validators* $(p,\ \sigma)) \longrightarrow c \in \varepsilon\ \sigma \wedge p\ c)$

definition *later-disagreeing-messages* :: (*consensus-value-property* $*$ *message* $*$ *validator* $*$ *state*) $\Rightarrow$ *message set*
where
later-disagreeing-messages $=$ ($\lambda(p,\ m,\ v,\ \sigma)$.$\{m' \in$ *later-from* $(m,\ v,\ \sigma).\ \neg\ p$ (*est* m')$\}$)

lemma (**in** *Protocol*) *later-disagreeing-messages-type* :
$\forall\ p\ \sigma\ v\ m.\ \sigma \in \Sigma \wedge v \in V \wedge m \in M \longrightarrow$ *later-disagreeing-messages* $(p,\ m,\ v,$

$\sigma) \subseteq M$
unfolding *later-disagreeing-messages-def*
using *later-from-type-for-state* **by** *auto*

definition $\textit{is-clique} :: (\textit{validator set} * \textit{consensus-value-property} * \textit{state}) \Rightarrow \textit{bool}$
where
$\textit{is-clique} = (\lambda(\textit{v-set}, p, \sigma).$
$(\forall\ v \in \textit{v-set}.\ v \in \textit{observed-non-equivocating-validators}\ \sigma$
$\wedge (\forall\ \ v' \in \textit{v-set}.$
$\textit{is-agreeing}\ (p, (\textit{the-elem}\ (\textit{L-H-J}\ \sigma\ v)), v')$
$\wedge\ \textit{later-disagreeing-messages}\ (p, \textit{the-elem}\ (\textit{L-H-M}\ (\textit{the-elem}\ (\textit{L-H-J}\ \sigma$
$v))\ v'), v', \sigma) = \emptyset)))$

lemma (**in** *Protocol*) *non-equivocating-validator-is-non-equivocating-in-past* :
$\forall\ \sigma\ v\ \sigma'.\ v \in V \wedge \{\sigma, \sigma'\} \subseteq \Sigma \wedge \textit{is-future-state}\ (\sigma', \sigma)$
$\longrightarrow v \notin \textit{equivocating-validators}\ \sigma$
$\longrightarrow v \notin \textit{equivocating-validators}\ \sigma'$
oops

lemma (**in** *Protocol*) *validator-in-clique-see-L-H-M-of-others-is-singleton* :
$\forall\ \textit{v-set}\ p\ \sigma.\ \textit{v-set} \subseteq V \wedge \sigma \in \Sigma$
$\longrightarrow \textit{is-clique}\ (\textit{v-set}, p, \sigma)$
$\longrightarrow (\forall\ v\ v'.\ \{v, v'\} \subseteq \textit{v-set} \longrightarrow \textit{is-singleton}\ (\textit{L-H-M}\ (\textit{the-elem}\ (\textit{L-H-J}\ \sigma\ v))$
$v'))$
sorry

lemma (**in** *Protocol*) *later-from-of-non-sender-not-affected-by-minimal-transitions*
:
$\forall\ \sigma\ \sigma'\ m\ m'\ v.\ (\sigma, \sigma') \in \textit{minimal-transitions} \wedge m \in M$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow v \in V - \{\textit{sender}\ m'\}$
$\longrightarrow \textit{later-from}\ (m, v, \sigma) = \textit{later-from}\ (m, v, \sigma')$
apply (*rule*, *rule*, *rule*, *rule*, *rule*, *rule*, *rule*, *rule*)
proof−
fix $\sigma\ \sigma'\ m\ m'\ v$
assume $(\sigma, \sigma') \in \textit{minimal-transitions} \wedge m \in M$
assume $m' = \textit{the-elem}\ (\sigma' - \sigma)$
assume $v \in V - \{\textit{sender}\ m'\}$

```
  have later-from (m,v,σ) = {m'' ∈ σ. sender m'' = v ∧ justified m m''}
    apply (simp add: later-from-def from-sender-def later-def)
    by auto
  also have ... = {m'' ∈ σ. sender m'' = v ∧ justified m m''} ∪ ∅
    by auto
  also have ... = {m'' ∈ σ. sender m'' = v ∧ justified m m''} ∪ {m'' ∈ {m'}.
sender m'' = v}
  proof−
    have {m'' ∈ {m'}. sender m'' = v} = ∅
      using ‹v ∈ V − {sender m'}› by auto
    thus ?thesis
      by blast
  qed
  also have ... = {m'' ∈ σ. sender m'' = v ∧ justified m m''} ∪ {m'' ∈ {m'}.
sender m'' = v ∧ justified m m''}
  proof−
    have sender m' = v ⟹ justified m m'
      using ‹v ∈ V − {sender m'}› by auto
    thus ?thesis
      by blast
  qed
  also have ... = {m'' ∈ σ ∪ {m'}. sender m'' = v ∧ justified m m''}
    by auto
  also have ... = {m'' ∈ σ'. sender m'' = v ∧ justified m m''}
  proof −
    have σ' =  σ ∪ {m'}
       using ‹(σ, σ') ∈ minimal-transitions ∧ m ∈ M› ‹m' = the-elem (σ' − σ)›
minimal-transitions-reconstruction by auto
    then show ?thesis
      by auto
  qed
  then have ... = later-from (m,v,σ')
    apply (simp add: later-from-def from-sender-def later-def)
    by auto
  then show later-from (m, v, σ) = later-from (m, v, σ')
   using ‹{m'' ∈ σ ∪ {m'}. sender m'' = v ∧ justified m m''} = {m'' ∈ σ'. sender
m'' = v ∧ justified m m''}› calculation by auto
qed


lemma (in Protocol) equivocation-status-of-non-sender-not-affected-by-minimal-transitions
:
  ∀ σ σ' m' v. (σ, σ') ∈ minimal-transitions
  ⟶ m' = the-elem (σ' − σ)
  ⟶ v ∈ V − {sender m'}
  ⟶ v ∈ equivocating-validators σ ⟷ v ∈ equivocating-validators σ'
  oops
```

lemma (**in** *Protocol*) *L-M-of-non-sender-not-affected-by-minimal-transitions* :
$\forall\ \sigma\ \sigma'\ m'\ v.\ (\sigma, \sigma') \in \mathit{minimal\text{-}transitions}$
$\longrightarrow m' = \mathit{the\text{-}elem}\ (\sigma' - \sigma)$
$\longrightarrow v \in V - \{\mathit{sender}\ m'\}$
$\longrightarrow \mathit{L\text{-}H\text{-}M}\ \sigma\ v = \mathit{L\text{-}H\text{-}M}\ \sigma'\ v$
oops

lemma (**in** *Protocol*) *latest-justificationss-of-non-sender-not-affected-by-minimal-transitions* :
$\forall\ \sigma\ \sigma'\ m'\ v.\ (\sigma, \sigma') \in \mathit{minimal\text{-}transitions}$
$\longrightarrow m' = \mathit{the\text{-}elem}\ (\sigma' - \sigma)$
$\longrightarrow v \in V - \{\mathit{sender}\ m'\}$
$\longrightarrow \mathit{L\text{-}H\text{-}J}\ \sigma\ v = \mathit{L\text{-}H\text{-}J}\ \sigma'\ v$
oops

lemma (**in** *Protocol*) *later-disagreeing-of-non-sender-not-affected-by-minimal-transitions* :
$\forall\ \sigma\ \sigma'\ m\ m'\ v.\ (\sigma, \sigma') \in \mathit{minimal\text{-}transitions} \wedge m \in M$
$\longrightarrow m' = \mathit{the\text{-}elem}\ (\sigma' - \sigma)$
$\longrightarrow v \in V - \{\mathit{sender}\ m'\}$
$\longrightarrow \mathit{later\text{-}disagreeing\text{-}messages}\ (p, m, v, \sigma) = \mathit{later\text{-}disagreeing\text{-}messages}\ (p, m, v, \sigma')$
oops

lemma (**in** *Protocol*) *clique-not-affected-by-minimal-transitions-outside-clique* :
$\forall\ \sigma\ \sigma'\ m'\ \mathit{v\text{-}set}.\ (\sigma, \sigma') \in \mathit{minimal\text{-}transitions} \wedge \mathit{v\text{-}set} \subseteq V$
$\longrightarrow m' = \mathit{the\text{-}elem}\ (\sigma' - \sigma)$
$\longrightarrow \mathit{is\text{-}clique}\ (\mathit{v\text{-}set}, p, \sigma) = \mathit{is\text{-}clique}\ (\mathit{v\text{-}set}, p, \sigma')$
oops

lemma (**in** *Protocol*) *free-sub-clique* :
$\forall\ \sigma\ \sigma'\ m'\ \mathit{v\text{-}set}.\ (\sigma, \sigma') \in \mathit{minimal\text{-}transitions} \wedge \mathit{v\text{-}set} \subseteq V$
$\longrightarrow m' = \mathit{the\text{-}elem}\ (\sigma' - \sigma)$
$\longrightarrow \mathit{is\text{-}clique}\ (\mathit{v\text{-}set}, p, \sigma) = \mathit{is\text{-}clique}\ (\mathit{v\text{-}set} - \{\mathit{sender}\ m'\}, p, \sigma')$
oops

```
lemma (in Protocol) later-messages-from-non-equivocating-validator-include-all-earlier-messages
:
  ∀ v σ σ1 σ2. σ ∈ Σ ∧ σ1 ∈ Σ ∧ σ1 ⊆ σ ∧ σ2 ⊆ σ ∧ σ1 ∩ σ2 = ∅
  ⟶ (∀ m1 ∈ σ1. sender(m1) = v ⟶ (∀ m2 ∈ σ2. sender(m2) = v ⟶ m1
∈ justification(m2)))
  oops


lemma (in Protocol) message-between-minimal-transition-is-latest-message :
  ∀ σ σ' m' v. (σ, σ') ∈ minimal-transitions
  ⟶ m' = the-elem (σ' − σ)
  ⟶ v ∉ equivocating-validators σ'
  ⟶ m' = the-elem (L-H-M σ' v)
  oops


lemma (in Protocol) latest-message-from-non-equivocating-validator-is-previous-latest-or-later:
  ∀ σ σ' m' v. (σ, σ') ∈ minimal-transitions
  ⟶ m' = the-elem (σ' − σ)
  ⟶ sender m' ∉ equivocating-validators σ ∧ v ∉ equivocating-validators σ'
  ⟶ the-elem (L-H-M (justification m') v)
       = the-elem (L-H-M (the-elem (L-H-J σ (sender m'))) v)
     ∨ justified (the-elem (L-H-M (the-elem (L-H-J σ (sender m'))) v))
                 (the-elem (L-H-M (justification m') v))
  oops


lemma (in Protocol) justified-message-exists-in-later-from:
  ∀ σ m1 m2. σ ∈ Σ ∧ {m1, m2} ⊆ σ
  ⟶ justified m1 m2 ⟶ m2 ∈ later-from (m1, sender m1, σ)
  apply (simp add: later-from-def later-def from-sender-def)
  oops


lemma (in Protocol) non-equivocating-message-from-clique-see-clique-agreeing :
  ∀ σ σ' m' v-set. (σ, σ') ∈ minimal-transitions ∧ v-set ⊆ V
  ⟶ m' = the-elem (σ' − σ)
 ⟶ is-clique (v-set, p, σ) ∧ sender m' ∈ v-set ∧ sender m' ∉ equivocating-validators
σ'
  ⟶ v-set ⊆ agreeing-validators (p, justification m')
  oops


lemma (in Protocol) new-message-from-majority-clique-see-members-agreeing :
  ∀ σ σ' m' v-set. (σ, σ') ∈ minimal-transitions ∧ v-set ⊆ V
```

$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{is-clique}\ (\textit{v-set}, p, \sigma) \wedge \textit{sender}\ m' \in \textit{v-set} \wedge \textit{sender}\ m' \notin \textit{equivocating-validators}\ \sigma'$
$\wedge\ (\forall\ v \in \textit{v-set}.\ \textit{is-majority}\ (\textit{v-set}, \textit{the-elem}\ (\textit{L-H-J}\ \sigma\ v)))$
$\longrightarrow \textit{sender}\ m' \in \textit{agreeing-validators}\ (p, \textit{justification}\ m')$
oops

lemma (**in** *Protocol*) *latest-message-in-justification-of-new-message-is-latest-message* :
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}.\ (\sigma, \sigma') \in \textit{minimal-transitions} \wedge \textit{v-set} \subseteq V$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{sender}\ m' \notin \textit{equivocating-validators}\ \sigma'$
$\longrightarrow \textit{the-elem}\ (\textit{L-H-M}\ (\textit{justification}\ m')\ (\textit{sender}\ m')) = \textit{the-elem}\ (\textit{L-H-M}\ \sigma\ (\textit{sender}\ m'))$
oops

lemma (**in** *Protocol*) *latest-message-justified-by-new-message* :
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}.\ (\sigma, \sigma') \in \textit{minimal-transitions} \wedge \textit{v-set} \subseteq V$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{sender}\ m' \notin \textit{equivocating-validators}\ \sigma'$
$\longrightarrow \textit{justified}\ (\textit{the-elem}\ (\textit{L-H-M}\ \sigma\ (\textit{sender}\ m')))\ m'$
oops

lemma (**in** *Protocol*) *nothing-later-than-latest-honest-message* :
$\forall\ v\ \sigma\ m.\ v \in V \wedge \sigma \in \Sigma \wedge m \in M$
$\longrightarrow v \notin \textit{equivocating-validators}\ \sigma'$
$\longrightarrow \textit{later-from}\ (\textit{the-elem}\ (\textit{L-H-M}\ \sigma\ v), v, \sigma) = \emptyset$
oops

lemma (**in** *Protocol*) *later-messages-for-sender-is-new-message* :
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}.\ (\sigma, \sigma') \in \textit{minimal-transitions} \wedge \textit{v-set} \subseteq V$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{sender}\ m' \notin \textit{equivocating-validators}\ \sigma'$
$\longrightarrow \textit{later-from}\ (\textit{the-elem}\ (\textit{L-H-M}\ \sigma\ (\textit{sender}\ m')), \textit{sender}\ m', \sigma') = \{m'\}$
oops

lemma (**in** *Protocol*) *later-disagreeing-is-monotonic*:
$\forall\ v\ \sigma\ \textit{m1}\ \textit{m2}.\ v \in V \wedge \sigma \in \Sigma \wedge \{\textit{m1}, \textit{m2}\} \subseteq M$
$\longrightarrow \textit{justified}\ \textit{m1}\ \textit{m2}$
$\longrightarrow \textit{later-disagreeing-messages}\ (p, \textit{m2}, v, \sigma) \subseteq \textit{later-disagreeing-messages}\ (p, \textit{m1}, v, \sigma)$

oops

lemma (**in** *Protocol*) *empty-later-disagreeing-messages-in-new-message* :
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}\ v\ p.\ (\sigma, \sigma') \in \textit{minimal-transitions} \wedge \textit{v-set} \subseteq V \wedge v \in V$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{sender}\ m' \notin \textit{equivocating-validators}\ \sigma'$
$\longrightarrow v \notin \textit{equivocating-validators}\ \sigma$
$\longrightarrow \textit{later-disagreeing-messages}\ (p, (\textit{the-elem}\ (\textit{L-H-M}\ (\textit{the-elem}\ (\textit{L-H-J}\ \sigma\ (\textit{sender}\ m')))\ v)), v, \sigma) = \emptyset$
$\longrightarrow \textit{later-disagreeing-messages}\ (p, (\textit{the-elem}\ (\textit{L-H-M}\ (\textit{justification}\ m')\ v)), v, \sigma) = \emptyset$
oops

lemma (**in** *Protocol*) *clique-not-affected-by-minimal-transitions-outside-clique* :
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}\ p.\ (\sigma, \sigma') \in \textit{minimal-transitions} \wedge \textit{v-set} \subseteq V$
$\longrightarrow \textit{is-majority-driven}\ p$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{is-clique}\ (\textit{v-set}, p, \sigma) \wedge \textit{sender}\ m' \in \textit{v-set} \wedge \textit{sender}\ m' \notin \textit{equivocating-validators}\ \sigma'$
$\wedge\ (\forall\ v \in \textit{v-set}.\ \textit{is-majority}\ (\textit{v-set}, \textit{the-elem}\ (\textit{L-H-J}\ \sigma\ v)))$
$\longrightarrow \textit{is-clique}\ (\textit{v-set}, p, \sigma')$
oops

definition (**in** *Params*) *gt-threshold* :: (*validator set* $*$ *state*) $\Rightarrow$ *bool*
where
gt-threshold
$= (\lambda(\textit{v-set}, \sigma).(\textit{weight-measure}\ \textit{v-set} > (\textit{weight-measure}\ V)\ \textit{div}\ 2 + t\ - \textit{weight-measure}\ (\textit{equivocating-validators}\ \sigma)))$

lemma (**in** *Protocol*) *gt-threshold-imps-majority-for-any-validator* :
$\forall\ \sigma\ \textit{v-set}\ p.\ \sigma \in \Sigma \wedge \textit{v-set} \subseteq V$
$\longrightarrow \textit{gt-threshold}\ (\textit{v-set}, \sigma)$
$\longrightarrow (\forall\ v \in \textit{v-set}.\ \textit{is-majority}\ (\textit{v-set}, \textit{the-elem}\ (\textit{L-H-J}\ \sigma\ v)))$
oops

definition (**in** *Params*) *is-clique-oracle* :: (*validator set* $*$ *state* $*$ *consensus-value-property*) $\Rightarrow$ *bool*
where
is-clique-oracle
$= (\lambda(\textit{v-set}, \sigma, p).\ (\textit{is-clique}\ (\textit{v-set} - (\textit{equivocating-validators}\ \sigma), p, \sigma) \wedge \textit{gt-threshold}\ (\textit{v-set} - (\textit{equivocating-validators}\ \sigma), \sigma)))$

lemma (**in** *Protocol*) *clique-oracles-preserved-over-minimal-transitions-from-validators-not-in-clique*
:
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}\ p.\ (\sigma, \sigma') \in \textit{minimal-transitions} \land \textit{v-set} \subseteq V$
$\longrightarrow \textit{is-majority-driven}\ p$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{sender}\ m' \notin \textit{v-set} - \textit{equivocating-validators}\ \sigma$
$\quad \land \textit{is-clique-oracle}\ (\textit{v-set}, \sigma, p)$
$\longrightarrow \textit{is-clique-oracle}\ (\textit{v-set}, \sigma', p)$
oops

lemma (**in** *Protocol*) *clique-oracles-preserved-over-minimal-transitions-from-non-equivocating-validator*
:
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}\ p.\ (\sigma, \sigma') \in \textit{minimal-transitions} \land \textit{v-set} \subseteq V$
$\longrightarrow \textit{is-majority-driven}\ p$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{sender}\ m' \in \textit{v-set} - \textit{equivocating-validators}\ \sigma \land \textit{sender}\ m' \notin \textit{equivocating-validators}$
σ'
$\quad \land \textit{is-clique-oracle}\ (\textit{v-set}, \sigma, p)$
$\longrightarrow \textit{is-clique-oracle}\ (\textit{v-set}, \sigma', p)$
oops

lemma (**in** *Protocol*) *clique-oracles-preserved-over-minimal-transitions-from-equivocating-validator*
:
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}\ p.\ (\sigma, \sigma') \in \textit{minimal-transitions} \land \textit{v-set} \subseteq V$
$\longrightarrow \textit{is-majority-driven}\ p$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{sender}\ m' \in \textit{v-set} - \textit{equivocating-validators}\ \sigma \land \textit{sender}\ m' \in \textit{equivocating-validators}$
σ'
$\quad \land \textit{is-clique-oracle}\ (\textit{v-set}, \sigma, p)$
$\longrightarrow \textit{is-clique-oracle}\ (\textit{v-set}, \sigma', p)$
oops

lemma (**in** *Protocol*) *clique-oracles-preserved-over-minimal-transitions* :
$\forall\ \sigma\ \sigma'\ m'\ \textit{v-set}\ p.\ (\sigma, \sigma') \in \textit{minimal-transitions} \land \textit{v-set} \subseteq V$
$\longrightarrow \textit{is-majority-driven}\ p$
$\longrightarrow m' = \textit{the-elem}\ (\sigma' - \sigma)$
$\longrightarrow \textit{is-clique-oracle}\ (\textit{v-set}, \sigma, p)$
$\longrightarrow \textit{is-clique-oracle}\ (\textit{v-set}, \sigma', p)$
sorry

lemma (**in** *Protocol*) *clique-oracles-preserved-over-nice-message* :
$\forall\ \sigma\ m'\ \textit{v-set}\ p.\ \sigma \in \Sigma t \land \textit{v-set} \subseteq V$
$\longrightarrow \textit{is-majority-driven}\ p$
$\longrightarrow \sigma \cup \{m'\} \in \Sigma t$

```
  ⟶ is-clique-oracle (v-set, σ, p)
  ⟶ is-clique-oracle (v-set, σ ∪ {m'}, p)
  sorry
```

```
lemma (in Protocol) clique-imps-everyone-agreeing :
  ∀ σ v-set p. σ ∈ Σ ∧ v-set ⊆ V
  ⟶ is-clique (v-set, p, σ)
  ⟶ v-set ⊆ agreeing-validators (p, σ)
  apply (rule, rule, rule, rule, rule)
proof−
  fix σ v-set p assume σ ∈ Σ ∧ v-set ⊆ V  and is-clique (v-set, p, σ)
  then have clique: ∀ v ∈ v-set. v ∈ observed-non-equivocating-validators σ
          ∧ later-disagreeing-messages (p,
                                        the-elem (L-H-M
                                           (the-elem (L-H-J σ v)) v)
                                       , v, σ) = ∅
    by (simp add: is-clique-def)
  then have p-on-est : ∀ v ∈ v-set. (∀ m ∈ {m' ∈ σ. sender m' = v
                                      ∧ justified (the-elem (L-H-M
                                                   (the-elem (L-H-J σ v)) v))
                                                 m'}.
                                       p(est m))
   by (simp add: later-disagreeing-messages-def later-from-def later-def from-sender-def)
  have ∀ v ∈ v-set. v ∈ observed-non-equivocating-validators σ
    using clique by simp
  then have ∀ v ∈ v-set. the-elem (L-H-J σ v)
                  = justification (the-elem (L-H-M σ v))
    apply (simp add: L-H-J-def)
   by (metis ‹σ ∈ Σ ∧ v-set ⊆ V› empty-iff is-singleton-the-elem L-H-M-of-observed-non-equivocating-validator-
singletonD singletonI the-elem-image-unique)
  then have justified-ok: ∀ v ∈ v-set. justified (the-elem (L-H-M
                                                   (the-elem (L-H-J σ v)) v))
                                    (the-elem (L-H-M σ v))
    using validator-in-clique-see-L-H-M-of-others-is-singleton
   by (smt Diff-iff L-H-M-def L-H-M-is-in-the-state L-M-from-non-observed-validator-is-empty
M-type ‹∀ v∈v-set. v ∈ observed-non-equivocating-validators σ› ‹σ ∈ Σ ∧ v-set ⊆ V›
‹is-clique (v-set, p, σ)› empty-subsetI insert-subset is-singleton-the-elem justified-def
observed-non-equivocating-validators-def state-is-subset-of-M subsetCE)
  have sender-ok: ∀ v ∈ v-set. sender (the-elem (L-H-M σ v)) = v
   using ‹∀ v ∈ v-set. v ∈ observed-non-equivocating-validators σ› sender-of-L-H-M
    using ‹σ ∈ Σ ∧ v-set ⊆ V› by blast
  have ∀ v ∈ v-set.  the-elem (L-H-M σ v) ∈ σ
   using ‹∀ v ∈ v-set. v ∈ observed-non-equivocating-validators σ› L-H-M-is-in-the-state
    using ‹σ ∈ Σ ∧ v-set ⊆ V› by blast
  then have ∀ v ∈ v-set. p (est (the-elem (L-H-M σ v)))
    using p-on-est sender-ok justified-ok
    by blast
  then have ∀ v ∈ v-set. p (the-elem (L-H-E σ v))
    apply (simp add: L-H-E-def)
   by (metis (no-types, lifting) ‹∀ v∈v-set. v ∈ observed-non-equivocating-validators
σ› ‹σ ∈ Σ ∧ v-set ⊆ V› empty-iff is-singleton-the-elem L-H-M-of-observed-non-equivocating-validator-is-singleto
singletonD singletonI the-elem-image-unique)
  then show v-set ⊆ agreeing-validators (p, σ)
    unfolding agreeing-validators-def
   by (smt ‹∀ v∈v-set. v ∈ observed-non-equivocating-validators σ› ‹σ ∈ Σ ∧ v-set ⊆
V› is-singleton-the-elem mem-Collect-eq L-H-E-of-observed-non-equivocating-validator-is-singleton
old.prod.case singletonD subsetI)
qed


lemma (in Protocol) threshold-sized-clique-imps-estimator-agreeing :
  ∀ σ v-set p. σ ∈ Σt ∧ v-set ⊆ V
  ⟶ finite v-set
  ⟶ is-majority-driven p
  ⟶ is-clique (v-set − equivocating-validators σ, p, σ) ∧ gt-threshold (v-set −
equivocating-validators σ, σ)
  ⟶ (∀ c ∈ ε σ. p c)
  apply (rule, rule, rule, rule, rule, rule, rule, rule)
proof −
  fix σ v-set p c
  assume σ ∈ Σt ∧ v-set ⊆ V
  and finite v-set
  and is-majority-driven p
  and is-clique (v-set − equivocating-validators σ, p, σ) ∧ gt-threshold (v-set −
equivocating-validators σ, σ)
  and c ∈ ε σ
  then have v-set − equivocating-validators σ ⊆ agreeing-validators (p, σ)
    using clique-imps-everyone-agreeing
    by (meson Diff-subset Σt-is-subset-of-Σ subsetCE subset-trans)
  then have weight-measure (v-set − equivocating-validators σ) ≤ weight-measure
(agreeing-validators (p, σ))
   using agreeing-validators-finite equivocating-validators-def weight-measure-comparison-strict-subset-gte
        Σt-is-subset-of-Σ ‹σ ∈ Σt ∧ v-set ⊆ V› ‹finite v-set› by auto
  have weight-measure (v-set − equivocating-validators σ) > (weight-measure V)
div 2 + t − weight-measure (equivocating-validators σ)
    using ‹is-clique (v-set − equivocating-validators σ, p, σ) ∧ gt-threshold (v-set
− equivocating-validators σ, σ)›
    unfolding gt-threshold-def by simp
  then have weight-measure (v-set − equivocating-validators σ) > (weight-measure
V) div 2
   using Σt-def ‹σ ∈ Σt ∧ v-set ⊆ V› equivocation-fault-weight-def is-faults-lt-threshold-def

    by auto
  then have weight-measure (v-set − equivocating-validators σ) > (weight-measure
(V − equivocating-validators σ)) div 2
```

```
  proof -
    have finite (V - equivocating-validators σ)
      using  V-type equivocating-validators-is-finite
      by simp
    moreover have V - equivocating-validators σ ⊆ V
      by (simp add: Diff-subset)
   ultimately have (weight-measure V) div 2 ≥ (weight-measure (V - equivocating-validators
σ)) div 2
      using weight-measure-comparison-strict-subset-gte
      by (simp add: V-type)
    then show ?thesis
     using ‹weight-measure V / 2 < weight-measure (v-set - equivocating-validators
σ)› by linarith
  qed
  then have weight-measure (agreeing-validators (p, σ)) > weight-measure (V -
equivocating-validators σ) div 2
     using ‹weight-measure (v-set - equivocating-validators σ) ≤ weight-measure
(agreeing-validators (p, σ))›
    by linarith
  then show p c
     using ‹is-majority-driven p› unfolding is-majority-driven-def is-majority-def
gt-threshold-def
    using ‹c ∈ ε σ›
   using Mi.simps Σt-is-subset-of-Σ ‹σ ∈ Σt ∧ v-set ⊆ V› non-justifying-message-exists-in-M-0
by blast
qed


lemma (in Protocol) clique-oracle-for-all-futures :
  ∀ σ v-set p. σ ∈ Σt ∧ v-set ⊆ V
  ⟶ is-majority-driven p
  ⟶ is-clique-oracle (v-set, σ, p)
  ⟶ (∀ σ' ∈ futures σ. is-clique-oracle (v-set, σ', p))
  apply (rule+)
proof -
  fix σ v-set p σ'
 assume σ ∈ Σt ∧ v-set ⊆ V and is-majority-driven p and is-clique-oracle (v-set,
σ, p) and σ' ∈ futures σ
  show is-clique-oracle (v-set, σ', p)
    using clique-oracles-preserved-over-minimal-transitions
  sorry
qed


lemma (in Protocol) clique-oracle-is-safety-oracle :
  ∀ σ v-set p. σ ∈ Σt ∧ v-set ⊆ V
  ⟶ finite v-set
  ⟶ is-majority-driven p
  ⟶ is-clique-oracle (v-set, σ, p)
  ⟶ (∀ σ' ∈ futures σ. naturally-corresponding-state-property p σ')
  using clique-oracle-for-all-futures threshold-sized-clique-imps-estimator-agreeing
  apply (simp add: is-clique-oracle-def naturally-corresponding-state-property-def)
  by (metis (mono-tags, lifting) futures-def mem-Collect-eq)

end
theory TFGCasper

imports Main HOL.Real CBCCasper LatestMessage SafetyOracle ConsensusSafety

begin
```

```
type-synonym block = consensus-value

locale BlockchainParams = Params +

  fixes B :: block set
  fixes genesis :: block

  and prev :: block ⇒ block


fun (in BlockchainParams) n-cestor :: block ∗ nat ⇒ block
  where
    n-cestor (b, 0) = b
  | n-cestor (b, n) = n-cestor (prev b, n−1)


definition (in BlockchainParams) blockchain-membership :: block ⇒ block ⇒ bool
(infixl ⇂ 70)
  where
    b1 ⇂ b2 = (∃ n. n ∈ ℕ ∧ b1 = n-cestor (b2, n))

notation (ASCII)
  comp  (infixl blockchain-membership 70)


definition (in BlockchainParams) score :: state ⇒ block ⇒ real
  where
    score σ b = sum W {v ∈ observed σ. ∃ b' ∈ B. b' ∈ (L-H-E σ v) ∧ (b ⇂ b')}


definition (in BlockchainParams) children :: block ∗ state ⇒ block set
```

where
$\mathit{children} = (\lambda(b,\ \sigma).\ \{b' \in \mathit{est}\ `\sigma.\ b = \mathit{prev}\ b'\})$

definition (**in** *BlockchainParams*) *best-children* :: *block* $*$ *state* $\Rightarrow$ *block set*
where
best-children $= (\lambda\ (b,\ \sigma).\ \{$*arg-max-on* $(\mathit{score}\ \sigma)\ (\mathit{children}\ (b,\ \sigma))\})$

function (**in** *BlockchainParams*) *GHOST* :: (*block set* $*$ *state*) $=>$ *block set*
where
GHOST (*b-set*, σ) $=$
$(\bigcup\ b \in \{b \in$ *b-set*. *children* $(b,\ \sigma) \neq \emptyset\}$. *GHOST* (*best-children* $(b,\ \sigma),\ \sigma))$
$\cup\ \{b \in$ *b-set*. *children* $(b,\ \sigma) = \emptyset\}$
by *auto*

definition (**in** *BlockchainParams*) *GHOST-estimator* :: *state* $\Rightarrow$ *block set*
where
GHOST-estimator $\sigma =$ *GHOST* $(\{\mathit{genesis}\},\ \sigma) \cup (\bigcup\ b \in$ *GHOST* $(\{\mathit{genesis}\},$
$\sigma)$. *children* $(b,\ \sigma))$

abbreviation (**in** *BlockchainParams*) P :: *consensus-value-property set*
where
$P \equiv \{p.\ \exists!b \in B.\ \forall\, b' \in B.\ (b \downharpoonleft b' \longrightarrow p\ b' = \mathit{True}) \wedge \neg\ (b \downharpoonleft b' \longrightarrow p\ b' =$
$\mathit{False})\}$

locale *Blockchain* $=$ *BlockchainParams* $+$ *Protocol* $+$
assumes *blockchain-type* : $\forall\ b\ b'\ b''.\ \{b,\ b',\ b''\} \subseteq B \longrightarrow b' \downharpoonleft b \wedge b'' \downharpoonleft b \longrightarrow$
$(b' \downharpoonleft b'' \vee b'' \downharpoonleft b')$
and *block-is-consensus-value* : $B = C$

definition (**in** *BlockchainParams*) *block-membership-property* :: *block* $\Rightarrow$ *consensus-value-property*
where
block-membership-property $b = (\lambda b'.\ b \downharpoonleft b')$

definition (**in** *BlockchainParams*) *block-conflicting* :: (*block* $*$ *block*) $\Rightarrow$ *bool*
where
block-conflicting $= (\lambda(b1,\ b2).\ \neg\ (b1 \downharpoonleft b2 \vee b2 \downharpoonleft b1))$

lemma (**in** *Blockchain*) *conflicting-blocks-imps-conflicting-decision* :
$\forall\ b1\ b2\ \sigma.\ \{b1,\ b2\} \subseteq B \wedge \sigma \in \Sigma$
$\longrightarrow$ *block-conflicting* $(b1,\ b2)$
$\longrightarrow$ *consensus-value-property-is-decided* (*block-membership-property* $b1,\ \sigma$)
$\longrightarrow$ *consensus-value-property-is-decided* (*consensus-value-property-not* (*block-membership-property*
$b2),\ \sigma)$

```
  apply (simp add: block-membership-property-def consensus-value-property-is-decided-def
          naturally-corresponding-state-property-def  state-property-is-decided-def)
   apply (rule, rule, rule, rule, rule, rule)
proof −
  fix b1 b2 σ
  assume b1 ∈ B ∧ b2 ∈ B ∧ σ ∈ Σ and block-conflicting (b1, b2) and ∀σ∈futures
σ. ∀b′∈ε σ. b1 ⇂ b′
  show  ∀σ∈futures σ. ∀c∈ε σ. ¬ b2 ⇂ c
  proof (rule ccontr)
    assume ¬ (∀σ∈futures σ. ∀c∈ε σ. ¬ b2 ⇂ c)
    hence ∃ σ ∈futures σ. ∃ c ∈ ε σ. b2 ⇂ c
      by blast
    hence ∃ σ ∈futures σ. ∃ c ∈ ε σ. b2 ⇂ c ∧ b1 ⇂ c
      using ‹∀σ∈futures σ. ∀b′∈ε σ. b1 ⇂ b′› by simp
    hence b1 ⇂ b2 ∨ b2 ⇂ b1
      using blockchain-type
      apply (simp)
      using Σt-is-subset-of-Σ ‹b1 ∈ B ∧ b2 ∈ B ∧ σ ∈ Σ› block-is-consensus-value
estimates-are-subset-of-C futures-def by blast
    then show False
      using ‹block-conflicting (b1, b2)›
      by (simp add: block-conflicting-def)
  qed
qed

theorem (in Blockchain) blockchain-safety :
  ∀ σ-set. σ-set ⊆ Σt
  ⟶ finite σ-set
  ⟶ is-faults-lt-threshold (⋃ σ-set)
  ⟶ (∀ σ σ′ b1 b2. {σ, σ′} ⊆ σ-set ∧ {b1, b2} ⊆ B ∧ block-conflicting (b1, b2)
∧ block-membership-property b1 ∈ consensus-value-property-decisions σ
      ⟶ block-membership-property b2 ∉ consensus-value-property-decisions σ′)
  apply (rule, rule, rule, rule, rule, rule, rule, rule, rule, rule)
proof −
  fix σ-set σ σ′ b1 b2
   assume σ-set ⊆ Σt and finite σ-set and is-faults-lt-threshold (⋃σ-set)
  and {σ, σ′} ⊆ σ-set ∧ {b1, b2} ⊆ B ∧ block-conflicting (b1, b2) ∧ block-membership-property
b1 ∈ consensus-value-property-decisions σ
   and block-membership-property b2 ∈ consensus-value-property-decisions σ′
  hence ¬ consensus-value-property-is-decided (consensus-value-property-not (block-membership-property
b1), σ′)
      using negation-is-not-decided-by-other-validator ‹σ-set ⊆ Σt› ‹finite σ-set›
‹is-faults-lt-threshold (⋃σ-set)› apply (simp add: consensus-value-property-decisions-def)

       using ‹{σ, σ′} ⊆ σ-set ∧ {b1, b2} ⊆ B ∧ block-conflicting (b1, b2) ∧
block-membership-property b1 ∈ consensus-value-property-decisions σ› by auto
   have {b1, b2} ⊆ B ∧ σ ∈ Σ ∧ block-conflicting (b1, b2)
     using Σt-is-subset-of-Σ ‹σ-set ⊆ Σt› ‹{σ, σ′} ⊆ σ-set ∧ {b1, b2} ⊆ B ∧
block-conflicting (b1, b2) ∧ block-membership-property b1 ∈ consensus-value-property-decisions
```

```
σ› by auto
  hence consensus-value-property-is-decided (consensus-value-property-not (block-membership-property
b1), σ')
     using ‹block-membership-property b2 ∈ consensus-value-property-decisions σ'›
conflicting-blocks-imps-conflicting-decision
     apply (simp add: consensus-value-property-decisions-def)
    by (metis ‹σ-set ⊆ Σt› ‹finite σ-set› ‹is-faults-lt-threshold (⋃σ-set)› ‹{σ, σ'} ⊆
σ-set ∧ {b1, b2} ⊆ B ∧ block-conflicting (b1, b2) ∧ block-membership-property b1
∈ consensus-value-property-decisions σ› conflicting-blocks-imps-conflicting-decision
consensus-value-property-decisions-def insert-subset mem-Collect-eq negation-is-not-decided-by-other-validator)

   then show False
      using ‹¬ consensus-value-property-is-decided (consensus-value-property-not
(block-membership-property b1), σ')› by blast
 qed


theorem (in Blockchain) no-decision-on-conflicting-blocks :
  ∀ σ1 σ2. {σ1, σ2} ⊆ Σt
  ⟶ is-faults-lt-threshold (σ1 ∪ σ2)
  ⟶ (∀ b1 b2. {b1, b2} ⊆ C ∧ block-conflicting (b1, b2)
      ⟶ block-membership-property b1 ∈ consensus-value-property-decisions σ1
      ⟶ block-membership-property b2 ∉ consensus-value-property-decisions σ2)
  apply (rule, rule, rule, rule, rule, rule, rule, rule, rule)
proof -
  fix σ1 σ2 b1 b2
  assume {σ1, σ2} ⊆ Σt and is-faults-lt-threshold (σ1 ∪ σ2) and {b1, b2} ⊆ C
∧ block-conflicting (b1, b2)
  and block-membership-property b1 ∈ consensus-value-property-decisions σ1
  and block-membership-property b2 ∈ consensus-value-property-decisions σ2
  hence consensus-value-property-is-decided (block-membership-property b1, σ1)
    by (simp add: consensus-value-property-decisions-def)
 hence ¬ consensus-value-property-is-decided (consensus-value-property-not (block-membership-property
b1), σ2)
   using two-party-consensus-safety-for-consensus-value-property ‹is-faults-lt-threshold
(σ1 ∪ σ2)› ‹{σ1, σ2} ⊆ Σt› by blast
  have block-membership-property b2 ∈ consensus-value-property-decisions σ2
    using ‹block-membership-property b2 ∈ consensus-value-property-decisions σ2›

    by (simp add: consensus-value-property-decisions-def)
  have σ2 ∈ Σt ∧ {b2, b1} ⊆ B ∧ block-conflicting (b2, b1)
   using block-is-consensus-value ‹{σ1, σ2} ⊆ Σt› ‹{b1, b2} ⊆ C ∧ block-conflicting
(b1, b2)› by (simp add: block-conflicting-def)
 hence consensus-value-property-is-decided (consensus-value-property-not (block-membership-property
b1), σ2)
     using  conflicting-blocks-imps-conflicting-decision ‹block-membership-property
b2 ∈ consensus-value-property-decisions σ2›
    using Σt-is-subset-of-Σ consensus-value-property-decisions-def by auto
  then show False
```

```
    using ‹¬ consensus-value-property-is-decided (consensus-value-property-not
(block-membership-property b1), σ2)› by blast
 qed



locale Ghost = BlockchainParams + Protocol +
  assumes block-type : ∀ b. b ∈ B ⟷ prev b ∈ B
  and block-is-consensus-value : B = C
  and ghost-is-estimator : ε = GHOST-estimator
  and genesis-type : genesis ∈ C

lemma (in Ghost) children-type :
  ∀ b σ. b ∈ B ∧ σ ∈ Σ ⟶ children (b, σ) ⊆ B
  apply (simp add: children-def)
  using Ghost-axioms Ghost-axioms-def Ghost-def by auto

lemma argmax-type :
  S ⊆ A ⟹ arg-max-on f S ∈ A
  apply (simp add: arg-max-on-def arg-max-def is-arg-max-def)
  oops

lemma (in Ghost) best-children-type :
  ∀ b σ. b ∈ B ∧ σ ∈ Σ ⟶ best-children (b, σ) ⊆ B
  apply (simp add: best-children-def arg-max-on-def arg-max-def is-arg-max-def)
  using children-type
  apply auto
  oops

lemma (in Ghost) GHSOT-type :
  ∀ σ b-set. σ ∈ Σ ∧ b-set ⊆ B ⟶ GHOST(b-set, σ) ⊆ B
  oops

lemma (in BlockchainParams) GHOST-is-valid-estimator :
  (∀ b. b ∈ B ⟷ prev b ∈ B) ∧ B = C ∧ genesis ∈ C
  ⟹ is-valid-estimator GHOST-estimator
 apply (simp add: is-valid-estimator-def BlockchainParams.GHOST-estimator-def)
  oops

lemma (in Ghost) block-membership-property-is-majority-driven :
  ∀ p ∈ P. is-majority-driven p
  apply (simp add: is-majority-driven-def)

  oops

lemma (in Ghost) block-membership-property-is-max-driven :
  ∀ p ∈ P. is-max-driven p
  apply (simp add: is-max-driven-def)
  oops

end
```