

# Coupled and Contra-Similarity

## and How to Compute Them

Benjamin Bisping      Luisa Montanari\*

June 28, 2021

### Abstract

A game characterization of coupled simulation and contrasimilarity, two weak forms of bisimilarity.

## Contents

|          |                                                                    |           |
|----------|--------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Simple Games</b>                                                | <b>1</b>  |
| <b>2</b> | <b>Preliminaries</b>                                               | <b>3</b>  |
| 2.1      | Some Utilities for Finite Partial Orders . . . . .                 | 3         |
| 2.2      | Labeled Transition Systems . . . . .                               | 4         |
| 2.3      | Transition Systems with Silent Steps . . . . .                     | 7         |
| 2.4      | Finite Transition Systems with Silent Steps . . . . .              | 16        |
| <b>3</b> | <b>Notions of Equivalence</b>                                      | <b>16</b> |
| 3.1      | Strong Simulation and Bisimulation . . . . .                       | 16        |
| 3.2      | Weak Simulation . . . . .                                          | 17        |
| 3.3      | Weak Bisimulation . . . . .                                        | 23        |
| 3.4      | Delay Simulation . . . . .                                         | 25        |
| 3.5      | Similarity ignores $\tau$ -sinks . . . . .                         | 25        |
| <b>4</b> | <b>Contrasimulation</b>                                            | <b>30</b> |
| 4.1      | Definition of Contrasimulation . . . . .                           | 30        |
| 4.2      | Intermediate Relation Mimicking Contrasim . . . . .                | 32        |
| 4.3      | The Contrasimulation Preorder Set Game . . . . .                   | 34        |
| 4.4      | Contrasimulation Implies Winning Strategy (Completeness) . . . . . | 34        |
| 4.5      | Winning Strategy Implies Contrasimulation (Soundness) . . . . .    | 40        |
| 4.6      | The Basic Contrasimulation Preorder Game . . . . .                 | 45        |
| 4.7      | Contrasimulation Implies Winning Strategy (Completeness) . . . . . | 45        |

## 1 Simple Games

```
theory Simple_Game
imports
  Main
begin
```

Simple games are games where player0 wins all infinite plays.

---

\*Technische Universität Berlin, Germany, {benjamin.bisping,luisa.montanari}@tu-berlin.de.

```

locale simple_game =
fixes
  game_move :: ⟨'s ⇒ 's ⇒ bool⟩ ("_  $\mapsto$  ♡ _" [70, 70] 80) and
  player0_position :: ⟨'s ⇒ bool⟩ and
  initial :: 's
begin

definition player1_position :: ⟨'s ⇒ bool⟩
  where ⟨player1_position s ≡ ¬ player0_position s⟩

— plays (to be precise: play p refixes) are lists. we model them with the most recent move at the
beginning. (for our purpose it's enough to consider finite plays)
type_synonym ⟨'s2⟩ play = ⟨'s2 list⟩
type_synonym ⟨'s2⟩ strategy = ⟨'s2 play ⇒ 's2⟩

inductive_set plays :: ⟨'s play set⟩ where
  ⟨[initial] ∈ plays⟩ |
  ⟨p#play ∈ plays ⇒ p  $\mapsto$  ♡ p' ⇒ p'#p#play ∈ plays⟩

— plays for a given player 0 strategy
inductive_set plays_for_strategy :: ⟨'s strategy ⇒ 's play set⟩ for f where
  init: ⟨[initial] ∈ plays_for_strategy f⟩ |
  p0move: ⟨n0#play ∈ plays_for_strategy f ⇒ player0_position n0 ⇒ n0  $\mapsto$  ♡ f (n0#play)
    ⇒ (f (n0#play))#n0#play ∈ plays_for_strategy f⟩ |
  p1move: ⟨n1#play ∈ plays_for_strategy f ⇒ player1_position n1 ⇒ n1  $\mapsto$  ♡ n1'
    ⇒ n1'#n1#play ∈ plays_for_strategy f⟩

lemma strategy_step:
  assumes
    ⟨n0 # n1 # rest ∈ plays_for_strategy f⟩
    ⟨player0_position n1⟩
  shows
    ⟨f (n1 # rest) = n0⟩
  using assms
  by (induct rule: plays_for_strategy.cases, auto simp add: simple_game.player1_position_def)

definition positional_strategy :: ⟨'s strategy ⇒ bool⟩ where
  ⟨positional_strategy f ≡ ∀ r1 r2 n. f (n # r1) = f (n # r2)⟩

  a strategy is sound if it only decides on enabled transitions.

definition sound_strategy :: ⟨'s strategy ⇒ bool⟩ where
  ⟨sound_strategy f ≡
    ∀ n0 play . n0#play ∈ plays_for_strategy f ∧ player0_position n0  $\longrightarrow$  n0  $\mapsto$  ♡ f (n0#play)⟩

lemma strategy_plays_subset:
  assumes ⟨play ∈ plays_for_strategy f⟩
  shows ⟨play ∈ plays⟩
  using assms by (induct rule: plays_for_strategy.induct, auto simp add: plays.intros)

lemma no_empty_plays:
  assumes ⟨[] ∈ plays⟩
  shows ⟨False⟩
  using assms plays.cases by blast

  player1 wins a play if the play has reached a deadlock where it's player0's turn

definition player1_wins :: ⟨'s play ⇒ bool⟩ where
  ⟨player1_wins play ≡ player0_position (hd play) ∧ (∄ p' . (hd play)  $\mapsto$  ♡ p')⟩

```

```

definition player0_winning_strategy :: ⟨'s strategy ⇒ bool⟩ where
  ⟨player0_winning_strategy f ≡ (∀ play ∈ plays_for_strategy f . ¬ player1_wins play)⟩

end

end

```

## 2 Preliminaries

### 2.1 Some Utilities for Finite Partial Orders

For some reason there seems to be no Isaebllle support for maximal elements in finite regions of incomplete partial orders (such as the transitive step relation in cycle-compressed transition systems ;).)

```

theory Finite_Partial_Order
  imports Main
begin

context preorder
begin

lemma foldl_max_inflation:
  ⟨foldl max x0 xs ≤ foldl max x0 (xs @ [x])⟩
  unfolding foldl_append foldl.simps
  by (simp add: ord.max_def)

lemma foldl_max_soundness:
  shows
    ⟨foldl max x0 (x0 # xs) ∈ set (x0 # xs)⟩
proof (induct xs rule: rev_induct)
  case Nil
  then show ?case by (auto simp add: max_def)
next
  case (snoc x xs)
  then show ?case unfolding foldl_append max_def by auto
qed

lemma foldl_max_maximal:
  shows
    ⟨∀ y ∈ set (x0 # xs). foldl max x0 (x0 # xs) ≤ y ⟶ y ≤ foldl max x0 (x0 # xs)⟩
proof (induct xs rule: rev_induct)
  case Nil
  then show ?case by (auto simp add: max_def)
next
  case (snoc x xs)
  then show ?case unfolding foldl.simps foldl_append
    by (metis Un_insert_right append_Nil2 foldl_Cons insert_iff list.simps(15) local.order_refl
      local.order_trans ord.max_def set_append snoc.hyps)
qed
end

context order — that is: partial order
begin

lemma finite_max:

```

```

fixes q
defines ⟨above_q ≡ {q'. q ≤ q'}⟩
assumes
  ⟨finite above_q⟩
shows
  ⟨∃ q_max. q_max ∈ above_q ∧ (∀ q'' ∈ above_q. q_max ≤ q'' → q'' = q_max)⟩
proof -
  have ⟨q ∈ above_q⟩ unfolding above_q_def by blast
  then obtain above_list where above_list_spec: ⟨set (q#above_list) = above_q⟩
    using ⟨finite above_q⟩ finite_list by auto
  define q_max where ⟨q_max ≡ foldl max q (q#above_list)⟩
  have ⟨q_max ∈ above_q⟩
    unfolding q_max_def above_list_spec[symmetric] using foldl_max_soundness .
  moreover have ⟨∀ q'' ∈ above_q. q_max ≤ q'' → q'' = q_max⟩
    unfolding q_max_def above_list_spec[symmetric] using foldl_max_maximal antisym by blast
  ultimately show ?thesis by blast
qed

end

end

```

## 2.2 Labeled Transition Systems

```

theory Transition_Systems
  imports Finite_Partial_Order
begin

locale lts =
fixes
  trans :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩

begin

abbreviation step_pred :: ⟨'s ⇒ ('a ⇒ bool) ⇒ 's ⇒ bool⟩
  where
  ⟨step_pred p af q ≡ ∃ a. af a ∧ trans p a q⟩

abbreviation step ::
  ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩
  ("⟶" [70, 70, 70] 80)
where
  ⟨p ⟶a q ≡ trans p a q⟩

inductive steps :: ⟨'s ⇒ ('a ⇒ bool) ⇒ 's ⇒ bool⟩
  ("⟶*" [70, 70, 70] 80)
where
  refl: ⟨p ⟶* A p⟩ | step: ⟨p ⟶* A q1 ⟹ q1 ⟶a q ⟹ A a ⟹ (p ⟶* A q)⟩

lemma steps_one_step:
  assumes
    ⟨p ⟶a p'⟩
    ⟨A a⟩
  shows
    ⟨p ⟶* A p'⟩ using steps.step[of p A p a p'] steps.refl[of p A] assms .

lemma steps_concat:

```

```

assumes
  ⟨p' ⟶* A p''⟩
  ⟨p ⟶* A p'⟩
shows
  ⟨p ⟶* A p''⟩ using assms
proof (induct arbitrary: p)
  case (refl p'' A p')
  then show ?case by auto
next
  case (step p' A p'' a pp p)
  hence ⟨p ⟶* A p''⟩ by simp
  show ?case using steps.step[OF 'p ⟶* A p''', step(3,4)] .
qed

lemma steps_left:
  assumes
    ⟨p ≠ p'⟩
    ⟨p ⟶* A p'⟩
  shows
    ⟨∃p'' a . p ⟶a p'' ∧ A a ∧ p'' ⟶* A p'⟩
  using assms(1)
  by (induct rule:steps.induct[OF assms(2)], blast, metis refl steps_concat steps_one_step)

lemma steps_no_step:
  assumes
    ⟨∧ a p' . p ⟶a p' ⟹ ¬A a⟩
    ⟨p ≠ p''⟩
    ⟨p ⟶* A p''⟩
  shows
    ⟨False⟩
  using steps_left[OF assms(2,3)] assms(1) by blast

lemma steps_no_step_pos:
  assumes
    ⟨∧ a p' . p ⟶a p' ⟹ ¬A a⟩
    ⟨p ⟶* A p'⟩
  shows
    ⟨p = p'⟩
  using assms steps_no_step by blast

lemma steps_loop:
  assumes
    ⟨∧ a p' . p ⟶a p' ⟹ p = p'⟩
    ⟨p ≠ p''⟩
    ⟨p ⟶* A p''⟩
  shows
    ⟨False⟩
  using assms(3,1,2) by (induct, auto)

corollary steps_transp:
  ⟨transp (λ p p'. p ⟶* A p')⟩
  using steps_concat unfolding transp_def by blast

lemma steps_spec:
  assumes

```

```

    ⟨p ⟶* A' p'⟩
    ⟨∧ a . A' a ⟹ A a⟩
  shows
    ⟨p ⟶* A p'⟩ using assms(1,2)
proof induct
  case (refl p)
  show ?case using steps.refl .
next
  case (step p A' pp a pp')
  hence ⟨p ⟶* A pp⟩ by simp
  then show ?case using step(3,4,5) steps.step by auto
qed

```

```

interpretation preorder ⟨(λ p p'. p ⟶* A p')⟩ ⟨λ p p'. p ⟶* A p' ∧ ¬(p' ⟶* A p)⟩
  by (standard, simp, simp add: steps.refl, metis steps_concat)

```

If one can reach only a finite portion of the graph following  $\longrightarrow^* A$ , and all cycles are loops, then there must be nodes which are maximal wrt.  $\longrightarrow^* A$ .

```

lemma step_max_deadlock:
  fixes A q
  assumes
    antiysmm: ⟨∧ r1 r2. r1 ⟶* A r2 ∧ r2 ⟶* A r1 ⟹ r1 = r2⟩ and
    finite: ⟨finite {q'. q ⟶* A q'}⟩ and
    no_max: ⟨∀ q'. q ⟶* A q' ⟹ (∃ q''. q' ⟶* A q'' ∧ q' ≠ q'')⟩
  shows
    False
proof -
  interpret order ⟨(λ p p'. p ⟶* A p')⟩ ⟨λ p p'. p ⟶* A p' ∧ ¬(p' ⟶* A p)⟩
  by (standard, simp add: assms(1))
  show ?thesis using local.finite_max assms local.order_trans mem_Collect_eq by metis
qed

end — end of lts

```

```

lemma lts_impl_steps2:
  assumes
    ⟨lts.steps step1 p1 ap p2⟩
    ⟨∧ p1 a p2 . step1 p1 a p2 ∧ P p1 a p2 ⟹ step2 p1 a p2⟩
    ⟨∧ p1 a p2 . P p1 a p2⟩
  shows
    ⟨lts.steps step2 p1 ap p2⟩
proof (induct rule: lts.steps.induct[OF assms(1)])
  case (1 p af)
  show ?case using lts.steps.refl[of step2 p af] by blast
next
  case (2 p af q1 a q)
  hence ⟨step2 q1 a q⟩ using assms(2,3) by blast
  thus ?case using lts.step[OF 2(2) _ 2(4)] by blast
qed

```

```

lemma lts_impl_steps:
  assumes
    ⟨lts.steps step1 p1 ap p2⟩
    ⟨∧ p1 a p2 . step1 p1 a p2 ⟹ step2 p1 a p2⟩
  shows
    ⟨lts.steps step2 p1 ap p2⟩
  using assms lts_impl_steps2[OF assms] by auto

```

end

## 2.3 Transition Systems with Silent Steps

```

theory Weak_Transition_Systems
  imports Transition_Systems
begin

locale lts_tau = lts trans for
  trans :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩ + fixes
  τ :: ⟨'a⟩ begin

definition tau :: ⟨'a ⇒ bool⟩ where ⟨tau a ≡ (a = τ)⟩

lemma tau_tau[simp]: ⟨tau τ⟩ unfolding tau_def by simp

abbreviation weak_step :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩
  ("_ ⇒_ _" [70, 70, 70] 80)
where
  ⟨(p ⇒a q) ≡ (∃ pq1 pq2.
    p ⟶* tau pq1 ∧
    pq1 ⟶a pq2 ∧
    pq2 ⟶* tau q)⟩

lemma step_weak_step:
  assumes ⟨p ⟶a p'⟩
  shows   ⟨p ⇒a p'⟩
  using assms steps.refl by auto

abbreviation weak_step_tau :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩
  ("_ ⇒^_ _" [70, 70, 70] 80)
where
  ⟨(p ⇒a q) ≡
    (tau a ⟶ p ⟶* tau q) ∧
    (¬tau a ⟶ p ⇒a q)⟩

abbreviation weak_step_delay :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩
  ("_ ⇒▷_ _" [70, 70, 70] 80)
where
  ⟨(p ⇒▷a q) ≡
    (tau a ⟶ p ⟶* tau q) ∧
    (¬tau a ⟶ (∃ pq.
      p ⟶* tau pq ∧
      pq ⟶a q))⟩

lemma weak_step_delay_implies_weak_tau:
  assumes ⟨p ⇒▷a p'⟩
  shows   ⟨p ⇒a p'⟩
  using assms steps.refl[of p' tau] by blast

lemma weak_step_delay_left:
  assumes
    ⟨¬ p0 ⟶a p1⟩
    ⟨p0 ⇒▷a p1⟩
    ⟨¬tau a⟩
  shows

```

```

    (∃ p0' t. tau t ∧ p0 ⟶t p0' ∧ p0' ⇒a p1)
  using assms steps_left by metis

primrec weak_step_seq :: ⟨'s ⇒ 'a list ⇒ 's ⇒ bool⟩
  ("_ ⇒$ _ _" [70, 70, 70] 80)
  where
    (weak_step_seq p0 [] p1 = p0 ⟶* tau p1)
  | (weak_step_seq p0 (a#A) p1 = (∃ p01 . p0 ⇒a p01 ∧ weak_step_seq p01 A p1))

lemma step_weak_step_tau:
  assumes ⟨p ⟶a p'⟩
  shows ⟨p ⇒a p'⟩
  using step_weak_step[OF assms] steps_one_step[OF assms]
  by blast

lemma step_tau_refl:
  shows ⟨p ⇒τ p⟩
  using steps_refl[of p tau]
  by simp

lemma weak_step_tau_weak_step[simp]:
  assumes ⟨p ⇒a p'⟩ ⟨¬ tau a⟩
  shows ⟨p ⇒a p'⟩
  using assms by auto

lemma weak_steps:
  assumes
    ⟨p ⇒a p'⟩
    ⟨∧ a . tau a ⟹ A a⟩
    ⟨A a⟩
  shows
    ⟨p ⟶* A p'⟩
proof -
  obtain pp pp' where pp:
    ⟨p ⟶* tau pp⟩ ⟨pp ⟶a pp'⟩ ⟨pp' ⟶* tau p'⟩
  using assms(1) by blast
  then have cascade:
    ⟨p ⟶* A pp⟩ ⟨pp ⟶* A pp'⟩ ⟨pp' ⟶* A p'⟩
  using steps_one_step steps_spec assms(2,3) by auto
  have ⟨p ⟶* A pp'⟩ using steps_concat[OF cascade(2) cascade(1)] .
  show ?thesis using steps_concat[OF cascade(3) 'p ⟶* A pp'''] .
qed

lemma weak_step_impl_weak_tau:
  assumes
    ⟨p ⇒a p'⟩
  shows
    ⟨p ⇒a p'⟩
  using assms weak_steps[OF assms, of tau] by auto

lemma weak_impl_strong_step:
  assumes
    ⟨p ⇒a p'⟩
  shows
    ⟨(∃ a' p' . tau a' ∧ p ⟶a p') ∨ (∃ p' . p ⟶a p')⟩
proof -
  from assms obtain pq1 pq2 where pq12:

```

```

    ⟨p ⟶* tau pq1⟩
    ⟨pq1 ⟶a pq2⟩
    ⟨pq2 ⟶* tau p'⟩ by blast
  show ?thesis
  proof (cases ⟨p = pq1⟩)
    case True
    then show ?thesis using pq12 by blast
  next
    case False
    then show ?thesis using pq12 steps_left[of p pq1 tau] by blast
  qed
qed

lemma weak_step_extend:
  assumes
    ⟨p1 ⟶* tau p2⟩
    ⟨p2 ⟶a p3⟩
    ⟨p3 ⟶* tau p4⟩
  shows
    ⟨p1 ⟶a p4⟩
  using assms steps_concat by blast

lemma weak_step_tau_tau:
  assumes
    ⟨p1 ⟶* tau p2⟩
    ⟨tau a⟩
  shows
    ⟨p1 ⟶a p2⟩
  using assms by blast

lemma weak_single_step[iff]:
  ⟨p ⟶$ [a] p' ⟷ p ⟶a p'⟩
  using steps.refl[of p' tau]
  by (meson steps_concat weak_step_seq.simps(1) weak_step_seq.simps(2))

abbreviation weak_enabled :: ⟨'s ⟶ 'a ⟶ bool⟩ where
  ⟨weak_enabled p a ≡
    ∃ pq1 pq2. p ⟶* tau pq1 ∧ pq1 ⟶a pq2⟩

lemma weak_enabled_step:
  shows ⟨weak_enabled p a = (∃ p'. p ⟶a p')⟩
  using step_weak_step steps_concat by blast

lemma step_tau_concat:
  assumes
    ⟨q ⟶a q'⟩
    ⟨q' ⟶τ q1⟩
  shows ⟨q ⟶a q1⟩
  proof -
    show ?thesis using assms steps_concat tau_tau by blast
  qed

lemma tau_step_concat:
  assumes
    ⟨q ⟶τ q'⟩
    ⟨q' ⟶a q1⟩

```

```

    shows  $\langle q \Rightarrow^a q1 \rangle$ 
proof -
  show ?thesis using assms steps_concat tau_tau by blast
qed

lemma tau_word_concat:
  assumes
     $\langle q \Rightarrow^\tau q' \rangle$ 
     $\langle q' \Rightarrow^A q1 \rangle$ 
  shows  $\langle q \Rightarrow^A q1 \rangle$  using assms
proof (cases A)
  case Nil
  hence  $\langle q' \Rightarrow^\tau q1 \rangle$  using assms by auto
  thus ?thesis using Nil assms steps_concat tau_tau weak_step_seq.simps by blast
next
  case (Cons a A)
  then obtain  $q''$  where  $\langle q' \Rightarrow^a q'' \rangle$  and A_step:  $\langle q'' \Rightarrow^A q1 \rangle$  using assms by auto
  hence  $\langle q \Rightarrow^a q'' \rangle$  using tau_step_concat[OF assms(1)] by auto
  then show ?thesis using Cons A_step  $\langle q \Rightarrow^a q'' \rangle$  by auto
qed

lemma strong_weak_transition_system:
  assumes
     $\langle \bigwedge p q a. p \mapsto a q \implies \neg \text{tau } a \rangle$ 
     $\langle \neg \text{tau } a \rangle$ 
  shows
     $\langle p \Rightarrow^a p' = p \mapsto a p' \rangle$ 
proof
  assume  $\langle p \Rightarrow^a p' \rangle$ 
  then obtain  $p0 p1$  where  $\langle p \mapsto^* \text{tau } p0 \rangle \langle p0 \mapsto a p1 \rangle \langle p1 \mapsto^* \text{tau } p' \rangle$  using assms by blast
  then have  $\langle p = p0 \rangle \langle p1 = p' \rangle$  using assms(1) steps_no_step by blast+
  with  $\langle p0 \mapsto a p1 \rangle$  show  $\langle p \mapsto a p' \rangle$  by blast
next
  assume  $\langle p \mapsto a p' \rangle$ 
  thus  $\langle p \Rightarrow^a p' \rangle$  using step_weak_step_tau by blast
qed

lemma rev_seq_split :
  assumes  $\langle q \Rightarrow^{\$} (xs @ [x]) q1 \rangle$ 
  shows  $\langle \exists q'. q \Rightarrow^{\$} xs q' \wedge q' \Rightarrow^x q1 \rangle$  using assms
proof (induct xs arbitrary: q)
  case Nil
  hence  $\langle q \Rightarrow^{\$} [x] q1 \rangle$  by auto
  hence x_succ:  $\langle q \Rightarrow^x q1 \rangle$  by blast
  have  $\langle q \Rightarrow^{\$} [] q \rangle$  by (simp add: steps.refl)
  then show ?case using x_succ by auto
next
  case (Cons a xs)
  then obtain  $q'$  where  $q'_\text{def}: \langle q \Rightarrow^a q' \wedge q' \Rightarrow^{\$} (xs @ [x]) q1 \rangle$  by auto
  then obtain  $q''$  where  $\langle q' \Rightarrow^{\$} xs q'' \wedge q'' \Rightarrow^x q1 \rangle$  using Cons.hyps[of  $\langle q' \rangle$ ] by auto
  then show ?case using  $q'_\text{def}$  by auto
qed

lemma rev_seq_concat:
  assumes

```

```

    <q ⇒ $as q'>
    <q' ⇒ $A q1>
    shows <q ⇒ $(as@A) q1> using assms
proof (induct as arbitrary: A q' rule: rev_induct)
  case Nil
  hence <q ⇒ ^τ q'> by auto
  hence <q ⇒ ^τ q' ∧ q' ⇒ $A q1> using Nil.prem1(2) by blast
  hence <q ⇒ $A q1> using tau_word_concat by auto
  then show ?case by simp
next
  case (snoc x xs)
  hence <∃ q''. q ⇒ $xs q'' ∧ q'' ⇒ ^x q'> using rev_seq_split by simp
  then obtain q'' where q_def: <q ⇒ $xs q''> <q'' ⇒ ^x q'> by auto
  from snoc.prem1(2) have <q' ⇒ $A q1> by blast
  hence <q'' ⇒ $(x#A) q1> using q_def by auto
  hence <q'' ⇒ $([x]@A) q1> by auto
  then show ?case using snoc.hyps[of <q''> <[x]@A>] q_def by auto
qed

lemma rev_seq_step_concat :
  assumes
    <q ⇒ $as q'>
    <q' ⇒ ^a q1>
  shows <q ⇒ $(as@[a]) q1>
proof -
  from assms(2) have <q' ⇒ $[a] q1> by blast
  thus ?thesis using rev_seq_concat assms(1) by auto
qed

lemma rev_seq_dstep_concat :
  assumes
    <q ⇒ $as q'>
    <q' ⇒ ▷ a q1>
  shows <q ⇒ $(as@[a]) q1>
proof -
  from assms(2) have <q' ⇒ ^a q1> using steps.refl by auto
  thus ?thesis using assms rev_seq_step_concat by auto
qed

lemma word_tau_concat:
  assumes
    <q ⇒ $A q'>
    <q' ⇒ ^τ q1>
  shows <q ⇒ $A q1>
proof -
  from assms(2) have <q' ⇒ $[] q1>
    using tau_tau weak_step_seq.simps(1) by blast
  thus ?thesis using assms(1) rev_seq_concat
    by (metis append.right_neutral)
qed

lemma list_rev_split :
  assumes <A ≠ []>
  shows <∃ as a. A = as@[a]>
proof -
  show ?thesis using assms rev_exhaust by blast
qed

```

```

primrec taufree :: ('a list  $\Rightarrow$  'a list)
  where
    (taufree [] = [])
    | (taufree (a#A) = (if tau a then taufree A else a#(taufree A)))

lemma weak_step_over_tau :
  assumes
    (p  $\Rightarrow$  $A p')
  shows (p  $\Rightarrow$  $(taufree A) p') using assms
proof (induct A arbitrary: p)
  case Nil
  thus ?case by auto
next
  case (Cons a as)
  then obtain p0 where (p  $\Rightarrow$  ^a p0) (p0  $\Rightarrow$  $as p') by auto
  then show ?case
  proof (cases (tau a))
    case True
    hence (p  $\Rightarrow$  $as p') using tau_word_concat (p  $\Rightarrow$  ^a p0) (p0  $\Rightarrow$  $ as p') tau_tau by blast
    hence (p  $\Rightarrow$  $(taufree as) p') using Cons by auto
    thus (p  $\Rightarrow$  $(taufree (a#as)) p') using True by auto
  next
    case False
    hence rec: (taufree (a#as) = a#(taufree as)) by auto
    hence (p0  $\Rightarrow$  $(taufree as) p') using (p0  $\Rightarrow$  $as p') Cons by auto
    hence (p  $\Rightarrow$  $(a#(taufree as)) p') using (p  $\Rightarrow$  ^a p0) by auto
    then show ?thesis using rec by auto
  qed
qed

lemma app_tau_taufree_list :
  assumes
    ( $\forall a \in \text{set } A. \neg \text{tau } a$ )
    (b =  $\tau$ )
  shows (A = taufree (A@[b])) using assms(1)
proof (induct A)
  case Nil
  then show ?case using assms by simp
next
  case (Cons x xs)
  have ( $\forall a \in \text{set } xs. \neg \text{tau } a$ ) ( $\neg \text{tau } x$ ) using assms(1) butlast_snoc Cons by auto
  hence last: (xs = taufree (xs @ [b])) using Cons by auto
  have (taufree (x#xs@[b]) = x#taufree (xs @ [b])) using ( $\neg \text{tau } x$ ) by auto
  hence (x # xs = x# taufree (xs @ [b])) using last by auto
  then show ?case using Cons.prems last by auto
qed

lemma word_steps_ignore_tau_addition:
  assumes
    ( $\forall a \in \text{set } A. a \neq \tau$ )
    (p  $\Rightarrow$  $ A p')
    (filter ( $\lambda a. a \neq \tau$ ) A' = A)
  shows
    (p  $\Rightarrow$  $ A' p')

```

```

    using assms
  proof (induct A' arbitrary: p A)
    case Nil': Nil
    then show ?case by simp
  next
    case Cons': (Cons a' A' p)
    show ?case proof (cases ⟨a' =  $\tau$ ⟩)
      case True
      with Cons'.prems have ⟨filter (λa. a ≠  $\tau$ ) A' = A⟩ by simp
      with Cons' have ⟨p ⇒$ A' p'⟩ by blast
      with True show ?thesis using steps.refl by fastforce
    next
      case False
      with Cons'.prems obtain A'' where
        A''_spec: ⟨A = a'#A''⟩ ⟨filter (λa. a ≠  $\tau$ ) A' = A''⟩ ⟨∀a∈set A''. a ≠  $\tau$ ⟩ by auto
      with Cons'.prems obtain p0 where
        p0_spec: ⟨p ⇒^a' p0⟩ ⟨p0 ⇒$ A'' p'⟩ by auto
      with Cons'.hyps A''_spec(2,3) have ⟨p0 ⇒$ A' p'⟩ by blast
      with p0_spec show ?thesis by auto
    qed
  qed

lemma word_steps_ignore_tau_removal:
  assumes
    ⟨p ⇒$ A p'⟩
  shows
    ⟨p ⇒$ (filter (λa. a ≠  $\tau$ ) A) p'⟩
  using assms
  proof (induct A arbitrary: p)
    case Nil
    then show ?case by simp
  next
    case (Cons a A)
    show ?case proof (cases ⟨a =  $\tau$ ⟩)
      case True
      with Cons show ?thesis using tau_word_concat by auto
    next
      case False
      with Cons.prems obtain p0 where p0_spec: ⟨p ⇒^a p0⟩ ⟨p0 ⇒$ A p'⟩ by auto
      with Cons.hyps have ⟨p0 ⇒$ (filter (λa. a ≠  $\tau$ ) A) p'⟩ by blast
      with ⟨p ⇒^a p0⟩ False show ?thesis by auto
    qed
  qed

definition weak_tau_succs :: "'s set ⇒ 's set" where
  ⟨weak_tau_succs Q = {q1. ∃q∈Q. q ⇒^ $\tau$  q1}⟩

definition dsuccs :: "'a ⇒ 's set ⇒ 's set" where
  ⟨dsuccs a Q = {q1. ∃q∈Q. q ⇒▷a q1}⟩

definition word_reachable_via_delay :: "'a list ⇒ 's ⇒ 's ⇒ 's ⇒ bool" where
  ⟨word_reachable_via_delay A p p0 p1 = (∃p00. p ⇒$(butlast A) p00 ∧ p00 ⇒▷(last A) p0 ∧ p0
  ⇒^ $\tau$  p1)⟩

primrec dsuccs_seq_rec :: "'a list ⇒ 's set ⇒ 's set" where
  ⟨dsuccs_seq_rec [] Q = Q⟩ |
  ⟨dsuccs_seq_rec (a#as) Q = dsuccs a (dsuccs_seq_rec as Q)⟩

```

```

lemma in_dsuccs_implies_word_reachable :
  assumes
    ⟨q' ∈ dsuccs_seq_rec (rev A) {q}⟩
  shows ⟨q ⇒$A q'⟩ using assms
proof (induct arbitrary: q' rule: rev_induct)
  case Nil
  hence ⟨q' = q⟩ by auto
  hence ⟨q ⇒τ q'⟩ by (simp add: steps.refl)
  thus ⟨q ⇒$[] q'⟩ by simp
next
  case (snoc a as)
  hence ⟨q' ∈ dsuccs_seq_rec (a#(rev as)) {q}⟩ by simp
  hence ⟨q' ∈ dsuccs a (dsuccs_seq_rec (rev as) {q})⟩ by simp
  then obtain q0 where ⟨q0 ∈ dsuccs_seq_rec (rev as) {q}⟩
    and ⟨q0 ⇒a q'⟩ using dsuccs_def by auto
  hence ⟨q ⇒$as q0⟩ using snoc.hyps by auto
  thus ⟨q ⇒$(as @ [a]) q'⟩
    using ⟨q0 ⇒a q'⟩ steps.refl rev_seq_step_concat by blast
qed

lemma word_reachable_implies_in_dsuccs :
  assumes
    ⟨q ⇒$A q'⟩
  shows ⟨q' ∈ weak_tau_succs (dsuccs_seq_rec (rev A) {q})⟩ using assms
proof (induct A arbitrary: q' rule: rev_induct)
  case Nil
  hence ⟨q ⇒τ q'⟩ using tau_tau weak_step_seq.simps(1) by blast
  hence ⟨q' ∈ weak_tau_succs {q}⟩ using weak_tau_succs_def by auto
  thus ?case using dsuccs_seq_rec.simps(1) by auto
next
  case (snoc a as)
  then obtain q1 where ⟨q ⇒$as q1⟩ and ⟨q1 ⇒a q'⟩ using rev_seq_split by blast
  hence in_succs: ⟨q1 ∈ weak_tau_succs (dsuccs_seq_rec (rev as) {q})⟩ using snoc.hyps by auto
  then obtain q0 where q0_def: ⟨q0 ∈ dsuccs_seq_rec (rev as) {q}⟩ ⟨q0 ⇒τ q1⟩
    using weak_tau_succs_def[of (dsuccs_seq_rec (rev as) {q})] by auto
  hence ⟨q0 ⇒a q'⟩ using ⟨q1 ⇒a q'⟩ steps_concat tau_tau by blast
  then obtain q2 where ⟨q0 ⇒a q2⟩ ⟨q2 ⇒τ q'⟩ using steps.refl by auto
  hence ⟨∃ q0 ∈ dsuccs_seq_rec (rev as) {q}. q0 ⇒a q2⟩ using q0_def by auto
  hence ⟨q2 ∈ dsuccs a (dsuccs_seq_rec (rev as) {q})⟩ using dsuccs_def by auto
  hence ⟨q2 ∈ dsuccs_seq_rec (a#(rev as)) {q}⟩ by auto
  hence ⟨q2 ∈ dsuccs_seq_rec (rev (as@[a])) {q}⟩ by auto
  hence ⟨∃ q2 ∈ dsuccs_seq_rec (rev (as@[a])) {q}. q2 ⇒τ q'⟩ using ⟨q2 ⇒τ q'⟩ by auto
  thus ?case using weak_tau_succs_def[of (dsuccs_seq_rec (rev (as@[a])) {q})] by auto
qed

lemma simp_dsuccs_seq_rev:
  assumes
    ⟨Q = dsuccs_seq_rec (rev A) {q0}⟩
  shows
    ⟨dsuccs a Q = dsuccs_seq_rec (rev (A@[a])) {q0}⟩
proof -
  show ?thesis by (simp add: assms)
qed

abbreviation tau_max :: ('s ⇒ bool) where
  ⟨tau_max p ≡ (∀ p'. p ⟶* tau p' ⟶ p = p')⟩

```

```

lemma tau_max_deadlock:
  fixes q
  assumes
     $\langle \bigwedge r1\ r2. r1 \mapsto^* \text{tau } r2 \wedge r2 \mapsto^* \text{tau } r1 \implies r1 = r2 \rangle$  — contracted cycles (anti-symmetry)
     $\langle \text{finite } \{q'. q \mapsto^* \text{tau } q'\} \rangle$ 
  shows
     $\langle \exists q'. q \mapsto^* \text{tau } q' \wedge \text{tau\_max } q' \rangle$ 
  using step_max_deadlock assms by blast

abbreviation stable_state ::  $\langle 's \Rightarrow \text{bool} \rangle$  where
   $\langle \text{stable\_state } p \equiv \# p' . \text{step\_pred } p \text{ tau } p' \rangle$ 

lemma stable_tauclosure_only_loop:
  assumes
     $\langle \text{stable\_state } p \rangle$ 
  shows
     $\langle \text{tau\_max } p \rangle$ 
  using assms steps_left by blast

coinductive divergent_state ::  $\langle 's \Rightarrow \text{bool} \rangle$  where
  omega:  $\langle \text{divergent\_state } p' \implies \text{tau } t \implies p \mapsto^t p' \implies \text{divergent\_state } p \rangle$ 

lemma ex_divergent:
  assumes  $\langle p \mapsto^a p \rangle$   $\langle \text{tau } a \rangle$ 
  shows  $\langle \text{divergent\_state } p \rangle$ 
  using assms
proof (coinduct)
  case  $\langle \text{divergent\_state } p \rangle$ 
  then show ?case using assms by auto
qed

lemma ex_not_divergent:
  assumes  $\langle \forall a\ q. p \mapsto^a q \longrightarrow \neg \text{tau } a \rangle$   $\langle \text{divergent\_state } p \rangle$ 
  shows  $\langle \text{False} \rangle$  using assms(2)
proof (cases rule:divergent_state.cases)
  case  $\langle \text{omega } p' \ t \rangle$ 
  thus ?thesis using assms(1) by auto
qed

lemma perpetual_instability_divergence:
  assumes
     $\langle \forall p'. p \mapsto^* \text{tau } p' \longrightarrow \neg \text{stable\_state } p' \rangle$ 
  shows
     $\langle \text{divergent\_state } p \rangle$ 
  using assms
proof (coinduct rule: divergent_state.coinduct)
  case  $\langle \text{divergent\_state } p \rangle$ 
  then obtain t p' where  $\langle \text{tau } t \rangle$   $\langle p \mapsto^t p' \rangle$  using steps.refl by blast
  then moreover have  $\langle \forall p''. p' \mapsto^* \text{tau } p'' \longrightarrow \neg \text{stable\_state } p'' \rangle$ 
    using divergent_state step_weak_step_tau steps_concat by blast
  ultimately show ?case by blast
qed

corollary non_divergence_implies_eventual_stability:
  assumes
     $\langle \neg \text{divergent\_state } p \rangle$ 

```

```

shows
  ⟨ $\exists p' . p \mapsto^* \text{tau } p' \wedge \text{stable\_state } p'$ ⟩
using assms perpetual_instability_divergence by blast

end — context lts_tau

```

## 2.4 Finite Transition Systems with Silent Steps

```

locale lts_tau_finite = lts_tau trans  $\tau$  for
  trans :: ⟨ $'s \Rightarrow 'a \Rightarrow 's \Rightarrow \text{bool}$ ⟩ and
   $\tau$  :: ⟨ $'a$ ⟩ +
assumes
  finite_state_set: ⟨finite (top:: $'s$  set)⟩
begin

lemma finite_state_rel: ⟨finite (top::( $'s$  rel))⟩
  using finite_state_set
  by (simp add: finite_prod)

end

end

```

## 3 Notions of Equivalence

### 3.1 Strong Simulation and Bisimulation

```

theory Strong_Relations
  imports Transition_Systems
begin

context lts
begin

definition simulation ::
  ⟨( $'s \Rightarrow 's \Rightarrow \text{bool}$ )  $\Rightarrow \text{bool}$ ⟩
where
  ⟨simulation R  $\equiv \forall p q. R p q \longrightarrow$ 
    ( $\forall p' a. p \mapsto^a p' \longrightarrow$ 
      ( $\exists q'. R p' q' \wedge (q \mapsto^a q')$ )⟩)⟩

definition bisimulation ::
  ⟨( $'s \Rightarrow 's \Rightarrow \text{bool}$ )  $\Rightarrow \text{bool}$ ⟩
where
  ⟨bisimulation R  $\equiv \forall p q. R p q \longrightarrow$ 
    ( $\forall p' a. p \mapsto^a p' \longrightarrow$ 
      ( $\exists q'. R p' q' \wedge (q \mapsto^a q')$ )⟩)  $\wedge$ 
    ( $\forall q' a. q \mapsto^a q' \longrightarrow$ 
      ( $\exists p'. R p' q' \wedge (p \mapsto^a p')$ )⟩)⟩

lemma bisim_ruleformat:
  assumes ⟨bisimulation R⟩
  and ⟨R p q⟩
  shows
    ⟨ $p \mapsto^a p' \implies (\exists q'. R p' q' \wedge (q \mapsto^a q'))$ ⟩
    ⟨ $q \mapsto^a q' \implies (\exists p'. R p' q' \wedge (p \mapsto^a p'))$ ⟩
  using assms unfolding bisimulation_def by auto

```

```
end — context lts
```

```
end
```

## 3.2 Weak Simulation

```
theory Weak_Relations
imports
  Weak_Transition_Systems
  Strong_Relations
begin
```

```
context lts_tau
begin
```

```
definition weak_simulation ::
  ⟨'s ⇒ 's ⇒ bool⟩ ⇒ bool
where
  weak_simulation R ≡ ∀ p q. R p q ⟶
    (∀ p' a. p ⟶a p' ⟶ (∃ q'. R p' q'
      ∧ (q ⇒a q')))
```

Note: Isabelle won't finish the proofs needed for the introduction of the following coinductive predicate if it unfolds the abbreviation of  $\Rightarrow^a$ . Therefore we use  $\Rightarrow^{\wedge}$  as a barrier. There is no mathematical purpose in this.

```
definition weak_step_tau2 :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩
  ("_ ⇒∧ _ _" [70, 70, 70] 80)
```

```
where [simp]:
  ⟨(p ⇒∧ a q) ≡ p ⇒a q⟩
```

```
coinductive greatest_weak_simulation ::
  ⟨'s ⇒ 's ⇒ bool⟩
```

```
where
  ⟨(∀ p' a. p ⟶a p' ⟶ (∃ q'. greatest_weak_simulation p' q' ∧ (q ⇒∧ a q'))
    ⟹ greatest_weak_simulation p q⟩
```

```
lemma weak_sim_ruleformat:
```

```
assumes ⟨weak_simulation R⟩
```

```
and ⟨R p q⟩
```

```
shows
```

```
⟨p ⟶a p' ⟹ ¬tau a ⟹ (∃ q'. R p' q' ∧ (q ⇒a q'))⟩
```

```
⟨p ⟶a p' ⟹ tau a ⟹ (∃ q'. R p' q' ∧ (q ⟶* tau q'))⟩
```

```
using assms unfolding weak_simulation_def by (blast+)
```

```
abbreviation weakly_simulated_by :: ⟨'s ⇒ 's ⇒ bool⟩ ("_ ⊆ws _" [60, 60] 65)
```

```
where ⟨weakly_simulated_by p q ≡ ∃ R . weak_simulation R ∧ R p q⟩
```

```
lemma weaksim_greatest:
```

```
shows ⟨weak_simulation (λ p q . p ⊆ws q)⟩
```

```
unfolding weak_simulation_def
```

```
by (metis (no_types, lifting))
```

```
lemma gws_is_weak_simulation:
```

```
shows ⟨weak_simulation greatest_weak_simulation⟩
```

```
unfolding weak_simulation_def
```

```

proof safe
  fix p q p' a
  assume ih:
    (greatest_weak_simulation p q)
    (p  $\mapsto^a$  p')
  hence (⟨∀ x xa. p  $\mapsto^x$  xa  $\longrightarrow$  (∃ q'. q  $\Rightarrow^{^x}$  q' ∧ greatest_weak_simulation xa q')⟩)
    by (meson greatest_weak_simulation.simps)
  then obtain q' where (q  $\Rightarrow^{^a}$  q' ∧ greatest_weak_simulation p' q') using ih by blast
  thus (∃ q'. greatest_weak_simulation p' q' ∧ q  $\Rightarrow^{^a}$  q')
    unfolding weak_step_tau2_def by blast
qed

lemma weakly_sim_by_implies_gws:
  assumes (p  $\sqsubseteq_{ws}$  q)
  shows (greatest_weak_simulation p q)
  using assms
proof (coinduct, simp del: weak_step_tau2_def, safe)
  fix x1 x2 R a xa
  assume ih: (weak_simulation R) (R x1 x2) (x1  $\mapsto^a$  xa)
  then obtain q' where (x2  $\Rightarrow^{^a}$  q') (R xa q')
    unfolding weak_simulation_def weak_step_tau2_def by blast
  thus (∃ q'. (xa  $\sqsubseteq_{ws}$  q' ∨ greatest_weak_simulation xa q') ∧ x2  $\Rightarrow^{^a}$  q')
    using ih by blast
qed

lemma gws_eq_weakly_sim_by:
  shows (p  $\sqsubseteq_{ws}$  q = greatest_weak_simulation p q)
  using weakly_sim_by_implies_gws gws_is_weak_simulation by blast

lemma steps_retain_weak_sim:
  assumes
    (weak_simulation R)
    (R p q)
    (p  $\mapsto^{*A}$  p')
    (⟨∧ a . tau a  $\Longrightarrow$  A a)
  shows (∃ q'. R p' q' ∧ q  $\mapsto^{*A}$  q')
  using assms(3,2,4) proof (induct)
  case (refl p' A)
  hence (R p' q ∧ q  $\mapsto^{*A}$  q) using assms(2) by (simp add: steps.refl)
  then show ?case by blast
next
  case (step p A p' a p'')
  then obtain q' where (R p' q') (q  $\mapsto^{*A}$  q') by blast
  obtain q'' where q'':
    (R p'' q'') (⟨¬ tau a  $\longrightarrow$  q'  $\Rightarrow^a$  q''⟩ ∧ (tau a  $\longrightarrow$  q'  $\mapsto^{*A}$  q''))
    using 'weak_simulation R' q'(1) step(3) unfolding weak_simulation_def by blast
  have (q'  $\mapsto^{*A}$  q'')
    using q''(2) steps_spec[of q'] step(4) step(6) weak_steps[of q' a q''] by blast
  hence (q  $\mapsto^{*A}$  q'') using steps_concat[OF _ q'(2)] by blast
  thus ?case using q''(1) by blast
qed

lemma weak_sim_weak_premise:
  (weak_simulation R =
    (∀ p q . R p q  $\longrightarrow$ 
      (∀ p' a. p  $\Rightarrow^a$  p'  $\longrightarrow$  (∃ q'. R p' q' ∧ q  $\Rightarrow^a$  q'))))
proof

```

```

assume ⟨∀ p q . R p q ⟶ (∀ p' a. p ⇒a p' ⟶ (∃ q'. R p' q' ∧ q ⇒a q'))⟩
thus ⟨weak_simulation R⟩
  unfolding weak_simulation_def using step_weak_step_tau by simp
next
assume ws: ⟨weak_simulation R⟩
show ⟨∀ p q. R p q ⟶ (∀ p' a. p ⇒a p' ⟶ (∃ q'. R p' q' ∧ q ⇒a q'))⟩
proof safe
  fix p q p' a pq1 pq2
  assume case_assms:
    ⟨R p q⟩
    ⟨p ⟶* tau pq1⟩
    ⟨pq1 ⟶a pq2⟩
    ⟨pq2 ⟶* tau p'⟩
  then obtain q' where q'_def: ⟨q ⟶* tau q'⟩ ⟨R pq1 q'⟩
    using steps_retain_weak_sim[OF ws] by blast
  then moreover obtain q'' where q''_def: ⟨R pq2 q''⟩ ⟨q' ⇒a q''⟩
    using ws case_assms(3) unfolding weak_simulation_def by blast
  then moreover obtain q''' where q'''_def: ⟨R p' q'''⟩ ⟨q'' ⟶* tau q'''⟩
    using case_assms(4) steps_retain_weak_sim[OF ws] by blast
  ultimately show ⟨∃ q'''. R p' q''' ∧ q ⇒a q'''⟩ using weak_step_extend by blast
next
fix p q p' a
assume
  ⟨R p q⟩
  ⟨p ⟶* tau p'⟩
  ⟨¬ q'. R p' q' ∧ q ⇒a q'⟩
  ⟨tau a⟩
thus ⟨False⟩
  using steps_retain_weak_sim[OF ws] by blast
next
— case identical to first case
fix p q p' a pq1 pq2
assume case_assms:
  ⟨R p q⟩
  ⟨p ⟶* tau pq1⟩
  ⟨pq1 ⟶a pq2⟩
  ⟨pq2 ⟶* tau p'⟩
then obtain q' where q'_def: ⟨q ⟶* tau q'⟩ ⟨R pq1 q'⟩
  using steps_retain_weak_sim[OF ws] by blast
then moreover obtain q'' where q''_def: ⟨R pq2 q''⟩ ⟨q' ⇒a q''⟩
  using ws case_assms(3) unfolding weak_simulation_def by blast
then moreover obtain q''' where q'''_def: ⟨R p' q'''⟩ ⟨q'' ⟶* tau q'''⟩
  using case_assms(4) steps_retain_weak_sim[OF ws] by blast
ultimately show ⟨∃ q'''. R p' q''' ∧ q ⇒a q'''⟩ using weak_step_extend by blast
qed
qed

lemma weak_sim_enabled_subs:
  assumes
    ⟨p ⊆ws q⟩
    ⟨weak_enabled p a⟩
    ⟨¬ tau a⟩
  shows ⟨weak_enabled q a⟩
proof-
  obtain p' where p'_spec: ⟨p ⇒ a p'⟩
    using ⟨weak_enabled p a⟩ weak_enabled_step by blast
  obtain R where ⟨R p q⟩ ⟨weak_simulation R⟩ using assms(1) by blast

```

```

then obtain q' where ⟨q ⇒a q'⟩
  unfolding weak_sim_weak_premise using weak_step_impl_weak_tau[OF p'_spec] by blast
thus ?thesis using weak_enabled_step assms(3) by blast
qed

lemma weak_sim_union_cl:
  assumes
    ⟨weak_simulation RA⟩
    ⟨weak_simulation RB⟩
  shows
    ⟨weak_simulation (λ p q. RA p q ∨ RB p q)⟩
  using assms unfolding weak_simulation_def by blast

lemma weak_sim_remove_dead_state:
  assumes
    ⟨weak_simulation R⟩
    ⟨∧ a p . ¬ step d a p ∧ ¬ step p a d⟩
  shows
    ⟨weak_simulation (λ p q . R p q ∧ q ≠ d)⟩
  unfolding weak_simulation_def
proof safe
  fix p q p' a
  assume
    ⟨R p q⟩
    ⟨q ≠ d⟩
    ⟨p ⟶a p'⟩
  then obtain q' where ⟨R p' q'⟩ ⟨q ⇒a q'⟩
    using assms(1) unfolding weak_simulation_def by blast
  moreover hence ⟨q' ≠ d⟩
    using assms(2) 'q ≠ d' by (metis steps.cases)
  ultimately show ⟨∃ q'. (R p' q' ∧ q' ≠ d) ∧ q ⇒a q'⟩ by blast
qed

lemma weak_sim_tau_step:
  ⟨weak_simulation (λ p1 q1 . q1 ⟶* tau p1)⟩
  unfolding weak_simulation_def
  using lts.steps.simps by metis

lemma weak_sim_trans_constructive:
  fixes R1 R2
  defines
    ⟨R ≡ λ p q . ∃ pq . (R1 p pq ∧ R2 pq q) ∨ (R2 p pq ∧ R1 pq q)⟩
  assumes
    R1_def: ⟨weak_simulation R1⟩ ⟨R1 p pq⟩ and
    R2_def: ⟨weak_simulation R2⟩ ⟨R2 pq q⟩
  shows
    ⟨R p q⟩ ⟨weak_simulation R⟩
proof-
  show ⟨R p q⟩ unfolding R_def using R1_def(2) R2_def(2) by blast
next
  show ⟨weak_simulation R⟩
    unfolding weak_sim_weak_premise R_def
  proof (safe)
    fix p q pq p' a pq1 pq2
    assume
      ⟨R1 p pq⟩
      ⟨R2 pq q⟩

```

```

    <¬ tau a>
    <p ⟶* tau pq1>
    <pq1 ⟶a pq2>
    <pq2 ⟶* tau p'>
  thus (∃ q'. (∃ pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q')) ∧ q ⇒a q'
    using R1_def(1) R2_def(1) unfolding weak_sim_weak_premise by blast
next
  fix p q pq p' a
  assume
    <R1 p pq>
    <R2 pq q>
    <p ⟶* tau p'>
    <¬ q'. (∃ pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q') ∧ q ⇒a q'>
    <tau a>
  thus (False)
    using R1_def(1) R2_def(1) unfolding weak_sim_weak_premise by blast
next
  fix p q pq p' a pq1 pq2
  assume
    <R1 p pq>
    <R2 pq q>
    <p ⟶* tau p'>
    <p ⟶* tau pq1>
    <pq1 ⟶a pq2>
    <pq2 ⟶* tau p'>
  then obtain pq' q' where <R1 p' pq'> <pq ⇒a pq'> <R2 pq' q'> <q ⇒a q'>
    using R1_def(1) R2_def(1) assms(3) unfolding weak_sim_weak_premise by blast
  thus (∃ q'. (∃ pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q')) ∧ q ⇒a q'
    by blast
next
  fix p q pq p' a pq1 pq2
  assume sa:
    <R2 p pq>
    <R1 pq q>
    <¬ tau a>
    <p ⟶* tau pq1>
    <pq1 ⟶a pq2>
    <pq2 ⟶* tau p'>
  then obtain pq' q' where <R2 p' pq'> <pq ⇒a pq'> <R1 pq' q'> <q ⇒a q'>
    using R2_def(1) R1_def(1) unfolding weak_sim_weak_premise by blast
  thus (∃ q'. (∃ pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q')) ∧ q ⇒a q'
    by blast
next
  fix p q pq p' a
  assume
    <R2 p pq>
    <R1 pq q>
    <p ⟶* tau p'>
    <¬ q'. (∃ pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q') ∧ q ⇒a q'>
    <tau a>
  thus (False)
    using R1_def(1) R2_def(1) weak_step_tau_tau[OF 'p ⟶* tau p'' tau_tau]
    unfolding weak_sim_weak_premise by (metis (no_types, lifting))
next
  fix p q pq p' a pq1 pq2
  assume sa:
    <R2 p pq>

```

```

    ⟨R1 pq q⟩
    ⟨p ⟶* tau p'⟩
    ⟨p ⟶* tau pq1⟩
    ⟨pq1 ⟶a pq2⟩
    ⟨pq2 ⟶* tau p'⟩
  then obtain pq' where ⟨R2 p' pq'⟩ ⟨pq ⟶^a pq'⟩
    using R1_def(1) R2_def(1) weak_step_impl_weak_tau[of p a p']
    unfolding weak_sim_weak_premise by blast
  moreover then obtain q' where ⟨R1 pq' q'⟩ ⟨q ⟶^a q'⟩
    using R1_def(1) sa(2) unfolding weak_sim_weak_premise by blast
  ultimately show ⟨∃q'. (∃pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q') ∧ q ⟶^a q'⟩
    by blast
qed
qed

lemma weak_sim_trans:
  assumes
    ⟨p ⊆ws pq⟩
    ⟨pq ⊆ws q⟩
  shows
    ⟨p ⊆ws q⟩
  using assms(1,2)
proof -
  obtain R1 R2 where ⟨weak_simulation R1⟩ ⟨R1 p pq⟩ ⟨weak_simulation R2⟩ ⟨R2 pq q⟩
    using assms(1,2) by blast
  thus ?thesis
    using weak_sim_trans_constructive tau_tau
    by blast
qed

lemma weak_sim_word_impl:
  fixes
    p q p' A
  assumes
    ⟨weak_simulation R⟩ ⟨R p q⟩ ⟨p ⟶$ A p'⟩
  shows
    ⟨∃q'. R p' q' ∧ q ⟶$ A q'⟩
  using assms(2,3) proof (induct A arbitrary: p q)
    case Nil
    then show ?case
      using assms(1) steps_retain_weak_sim by auto
  next
    case (Cons a A)
    then obtain p'' where p''_spec: ⟨p ⟶^a p''⟩ ⟨p'' ⟶$ A p'⟩ by auto
    with Cons(2) assms(1) obtain q'' where q''_spec: ⟨q ⟶^a q''⟩ ⟨R p'' q''⟩
      unfolding weak_sim_weak_premise by blast
    then show ?case using Cons(1) p''_spec(2)
      using weak_step_seq.simps(2) by blast
  qed

lemma weak_sim_word_impl_contra:
  assumes
    ⟨∀ p q . R p q ⟶
      (∀ p' A. p ⟶$ A p' ⟶ (∃ q'. R p' q' ∧ q ⟶$ A q'))⟩
  shows
    ⟨weak_simulation R⟩
proof -

```

```

from assms have
  ⟨ $\forall p q p'. A. R p q \longrightarrow p \Rightarrow A p' \longrightarrow (\exists q'. R p' q' \wedge q \Rightarrow A q')$ ⟩ by blast
hence ⟨ $\forall p q p' a. R p q \longrightarrow p \Rightarrow [a] p' \longrightarrow (\exists q'. R p' q' \wedge q \Rightarrow [a] q')$ ⟩ by blast
thus ?thesis unfolding weak_single_step weak_sim_weak_premise by blast
qed

```

```

lemma weak_sim_word:
  ⟨weak_simulation R =
    (⟨ $\forall p q. R p q \longrightarrow$ 
      (⟨ $\forall p'. A. p \Rightarrow A p' \longrightarrow (\exists q'. R p' q' \wedge q \Rightarrow A q')$ ⟩)⟩)⟩
  using weak_sim_word_impl weak_sim_word_impl_contra by blast

```

### 3.3 Weak Bisimulation

```

definition weak_bisimulation ::
  ⟨('s  $\Rightarrow$  's  $\Rightarrow$  bool)  $\Rightarrow$  bool⟩
where
  ⟨weak_bisimulation R  $\equiv \forall p q. R p q \longrightarrow$ 
    (⟨ $\forall p' a. p \mapsto a p' \longrightarrow (\exists q'. R p' q' \wedge (q \Rightarrow a q'))$ ⟩  $\wedge$ 
      (⟨ $\forall q' a. q \mapsto a q' \longrightarrow (\exists p'. R p' q' \wedge (p \Rightarrow a p'))$ ⟩)⟩)⟩

lemma weak_bisim_ruleformat:
  assumes ⟨weak_bisimulation R⟩
  and ⟨R p q⟩
  shows
    ⟨ $p \mapsto a p' \Longrightarrow \neg \text{tau } a \Longrightarrow (\exists q'. R p' q' \wedge (q \Rightarrow a q'))$ ⟩
    ⟨ $p \mapsto a p' \Longrightarrow \text{tau } a \Longrightarrow (\exists q'. R p' q' \wedge (q \mapsto^* \text{tau } q'))$ ⟩
    ⟨ $q \mapsto a q' \Longrightarrow \neg \text{tau } a \Longrightarrow (\exists p'. R p' q' \wedge (p \Rightarrow a p'))$ ⟩
    ⟨ $q \mapsto a q' \Longrightarrow \text{tau } a \Longrightarrow (\exists p'. R p' q' \wedge (p \mapsto^* \text{tau } p'))$ ⟩
  using assms unfolding weak_bisimulation_def by (blast+)

```

```

definition tau_weak_bisimulation ::
  ⟨('s  $\Rightarrow$  's  $\Rightarrow$  bool)  $\Rightarrow$  bool⟩
where
  ⟨tau_weak_bisimulation R  $\equiv \forall p q. R p q \longrightarrow$ 
    (⟨ $\forall p' a. p \mapsto a p' \longrightarrow$ 
      (⟨ $\exists q'. R p' q' \wedge (q \Rightarrow a q')$ ⟩)  $\wedge$ 
      (⟨ $\forall q' a. q \mapsto a q' \longrightarrow$ 
        (⟨ $\exists p'. R p' q' \wedge (p \Rightarrow a p')$ ⟩)⟩)⟩)⟩

```

```

lemma weak_bisim_implies_tau_weak_bisim:
  assumes
    ⟨tau_weak_bisimulation R⟩
  shows
    ⟨weak_bisimulation R⟩
  unfolding weak_bisimulation_def proof (safe)
    fix p q p' a
    assume ⟨R p q⟩ ⟨ $p \mapsto a p'$ ⟩
    thus ⟨ $\exists q'. R p' q' \wedge (q \Rightarrow a q')$ ⟩
      using assms weak_steps[of q a _ tau] unfolding tau_weak_bisimulation_def by blast
  next
    fix p q q' a
    assume ⟨R p q⟩ ⟨ $q \mapsto a q'$ ⟩
    thus ⟨ $\exists p'. R p' q' \wedge (p \Rightarrow a p')$ ⟩
      using assms weak_steps[of p a _ tau] unfolding tau_weak_bisimulation_def by blast

```

```

qed

lemma weak_bisim_invert:
  assumes
    ⟨weak_bisimulation R⟩
  shows
    ⟨weak_bisimulation (λ p q. R q p)⟩
using assms unfolding weak_bisimulation_def by auto

lemma bisim_weak_bisim:
  assumes ⟨bisimulation R⟩
  shows ⟨weak_bisimulation R⟩
  unfolding weak_bisimulation_def
proof (clarify, rule)
  fix p q
  assume R: ⟨R p q⟩
  show ⟨∀ p' a. p ⟶a p' ⟶ (∃ q'. R p' q' ∧ (q ⟶a q'))⟩
  proof (clarify)
    fix p' a
    assume p'a: ⟨p ⟶a p'⟩
    have
      ⟨¬ tau a ⟶ (∃ q'. R p' q' ∧ q ⟶a q')⟩
      ⟨(tau a ⟶ (∃ q'. R p' q' ∧ q ⟶* tau q'))⟩
      using bisim_ruleformat(1)[OF assms R p'a] step_weak_step step_weak_step_tau by auto
    thus ⟨∃ q'. R p' q' ∧ (q ⟶a q')⟩ by blast
  qed
next
  fix p q
  assume R: ⟨R p q⟩
  have ⟨∀ q' a. q ⟶a q' ⟶ (¬ tau a ⟶ (∃ p'. R p' q' ∧ p ⟶a p'))
    ∧ (tau a ⟶ (∃ p'. R p' q' ∧ p ⟶* tau p'))⟩
  proof (clarify)
    fix q' a
    assume q'a: ⟨q ⟶a q'⟩
    show
      ⟨(¬ tau a ⟶ (∃ p'. R p' q' ∧ p ⟶a p')) ∧
      (tau a ⟶ (∃ p'. R p' q' ∧ p ⟶* tau p'))⟩
      using bisim_ruleformat(2)[OF assms R q'a] step_weak_step
      step_weak_step_tau steps_one_step by auto
  qed
  thus ⟨∀ q' a. q ⟶a q' ⟶ (∃ p'. R p' q' ∧ (p ⟶a p'))⟩ by blast
qed

lemma weak_bisim_weak_sim:
  shows
    ⟨weak_bisimulation R = (weak_simulation R ∧ weak_simulation (λ p q . R q p))⟩
  unfolding weak_bisimulation_def weak_simulation_def by auto

lemma steps_retain_weak_bisim:
  assumes
    ⟨weak_bisimulation R⟩
    ⟨R p q⟩
    ⟨p ⟶*A p'⟩
    ⟨⋀ a . tau a ⟹ A a⟩
  shows ⟨∃ q'. R p' q' ∧ q ⟶*A q'⟩
  using assms weak_bisim_weak_sim steps_retain_weak_sim
  by auto

```

```

lemma weak_bisim_union:
  assumes
    ⟨weak_bisimulation R1⟩
    ⟨weak_bisimulation R2⟩
  shows
    ⟨weak_bisimulation (λ p q . R1 p q ∨ R2 p q)⟩
  using assms unfolding weak_bisimulation_def by blast

lemma weak_bisim_taufree_strong:
  assumes
    ⟨weak_bisimulation R⟩
    ⟨ $\bigwedge p q a. p \mapsto a q \implies \neg \text{tau } a$ ⟩
  shows
    ⟨bisimulation R⟩
  using assms strong_weak_transition_system
  unfolding weak_bisimulation_def bisimulation_def
  by auto

```

### 3.4 Delay Simulation

```

definition delay_simulation ::
  ⟨('s  $\Rightarrow$  's  $\Rightarrow$  bool)  $\Rightarrow$  bool⟩
where
  ⟨delay_simulation R  $\equiv \forall p q. R p q \longrightarrow$ 
    ⟨ $\forall p' a. p \mapsto a p' \longrightarrow$ 
      ⟨tau a  $\longrightarrow R p' q$ ⟩  $\wedge$ 
      ⟨ $\neg \text{tau } a \longrightarrow (\exists q'. R p' q' \wedge (q \Rightarrow a q'))$ ⟩⟩⟩⟩

lemma delay_simulation_implies_weak_simulation:
  assumes
    ⟨delay_simulation R⟩
  shows
    ⟨weak_simulation R⟩
  using assms weak_step_delay_implies_weak_tau steps.refl
  unfolding delay_simulation_def weak_simulation_def by blast

end — context lts_tau

```

### 3.5 Similarity ignores $\tau$ -sinks

```

lemma simulation_tau_sink_1:
  fixes
    step sink  $\tau$  R
  defines
    ⟨step2  $\equiv \lambda p1 a p2. (p1 \neq \text{sink} \wedge a = \tau \wedge p2 = \text{sink}) \vee \text{step } p1 a p2$ ⟩
  assumes
    ⟨ $\bigwedge a p. \neg \text{step sink } a p$ ⟩
    ⟨lts_tau.weak_simulation step  $\tau$  R⟩
  shows
    ⟨lts_tau.weak_simulation step2  $\tau$  (λ p q. p = sink  $\vee R p q$ )⟩
proof -
  let ?tau = ⟨lts_tau.tau  $\tau$ ⟩
  let ?tauEx = ⟨ $\tau$ ⟩
  show ?thesis unfolding lts_tau.weak_simulation_def
proof safe
  fix p q p' a

```

```

    assume ⟨step2 sink a p'⟩
    hence ⟨p' = sink⟩ ⟨a = τ⟩
      using assms(2) unfolding step2_def by auto
    thus ⟨∃q'. (p' = sink ∨ R p' q') ∧
      (?tau a ⟶ lts.steps step2 q ?tau q') ∧
      (¬ ?tau a ⟶ (∃pq1 pq2. lts.steps step2 q ?tau pq1 ∧ step2 pq1 a pq2
        ∧ lts.steps step2 pq2 ?tau q'))⟩
      using lts_tau.step_tau_refl[of τ step2 q] by auto
  next
    fix p q p' a
    assume ⟨step2 p a p'⟩ ⟨R p q⟩
    have step_impl_step2: ⟨∧ p1 a p2 . step p1 a p2 ⟹ step2 p1 a p2⟩
      unfolding step2_def by blast
    have ⟨(p ≠ sink ∧ a = ?tauEx ∧ p' = sink) ∨ step p a p'⟩
      using 'step2 p a p'' unfolding step2_def .
    thus ⟨∃q'. (p' = sink ∨ R p' q') ∧
      (?tau a ⟶ lts.steps step2 q ?tau q') ∧
      (¬ ?tau a ⟶ (∃pq1 pq2. lts.steps step2 q ?tau pq1 ∧ step2 pq1 a pq2
        ∧ lts.steps step2 pq2 ?tau q'))⟩
    proof safe
      show ⟨∃q'. (sink = sink ∨ R sink q') ∧
        (?tau ?tauEx ⟶ lts.steps step2 q ?tau q') ∧
        (¬ ?tau ?tauEx ⟶ (∃pq1 pq2. lts.steps step2 q ?tau pq1
          ∧ step2 pq1 ?tauEx pq2 ∧ lts.steps step2 pq2 ?tau q'))⟩
        using lts.steps.refl[of step2 q ?tau] assms(1) by (meson lts_tau.tau_tau)
    next
      assume ⟨step p a p'⟩
      then obtain q' where q'_def:
        ⟨R p' q' ∧
        (?tau a ⟶ lts.steps step q ?tau q') ∧
        (¬ ?tau a ⟶ (∃pq1 pq2. lts.steps step q ?tau pq1 ∧ step pq1 a pq2
          ∧ lts.steps step pq2 ?tau q'))⟩
        using assms(3) 'R p q' unfolding lts_tau.weak_simulation_def by blast
      hence ⟨(p' = sink ∨ R p' q') ∧
        (?tau a ⟶ lts.steps step2 q ?tau q') ∧
        (¬ ?tau a ⟶ (∃pq1 pq2. lts.steps step2 q ?tau pq1 ∧ step2 pq1 a pq2
          ∧ lts.steps step2 pq2 ?tau q'))⟩
        using lts_impl_steps[of step _ _ step2] step_impl_step2 by blast
      thus ⟨∃q'. (p' = sink ∨ R p' q') ∧
        (?tau a ⟶ lts.steps step2 q ?tau q') ∧
        (¬ ?tau a ⟶ (∃pq1 pq2. lts.steps step2 q ?tau pq1 ∧ step2 pq1 a pq2
          ∧ lts.steps step2 pq2 ?tau q'))⟩
        by blast
    qed
  qed
qed

lemma simulation_tau_sink_2:
  fixes
    step sink R τ
  defines
    (step2 ≡ λ p1 a p2 . (p1 ≠ sink ∧ a = τ ∧ p2 = sink) ∨ step p1 a p2)
  assumes
    ⟨∧ a p . ¬ step sink a p ∧ ¬ step p a sink⟩
    ⟨lts_tau.weak_simulation step2 τ (λ p q. p = sink ∨ R p q)⟩
    ⟨∧ p' q' q . (p' = sink ∨ R p' q')
      ∧ lts.steps step2 q (lts_tau.tau τ) q' ⟶ (p' = sink ∨ R p' q)⟩

```

```

shows
  (lts_tau.weak_simulation step  $\tau$  ( $\lambda p q. p = \text{sink} \vee R p q$ ))
proof -
  let ?tau = (lts_tau.tau  $\tau$ )
  let ?tauEx = ( $\tau$ )
  let ?steps = (lts.steps)
  show ?thesis
    unfolding lts_tau.weak_simulation_def
  proof safe
    fix p q p' a
    assume
      (step sink a p')
    hence False using assms(2) by blast
    thus  $\langle \exists q'. (p' = \text{sink} \vee R p' q') \wedge$ 
      ( $?tau a \longrightarrow ?steps \text{ step } q ?tau q'$ )  $\wedge$ 
      ( $\neg ?tau a \longrightarrow (\exists pq1 pq2. ?steps \text{ step } q ?tau pq1 \wedge \text{step } pq1 a pq2$ 
         $\wedge ?steps \text{ step } pq2 ?tau q') \rangle$  ..
  next
    fix p q p' a
    assume  $\langle R p q \rangle \langle \text{step } p a p' \rangle$ 
    hence not_sink:  $\langle p \neq \text{sink} \rangle \langle p' \neq \text{sink} \rangle$ 
      using assms(2)[of a p] assms(2)[of a p'] by auto
    have  $\langle \text{step2 } p a p' \rangle$  using 'step p a p' ' unfolding step2_def by blast
    then obtain q' where q'_def:
       $\langle p' = \text{sink} \vee R p' q' \rangle$ 
       $\langle ?tau a \longrightarrow ?steps \text{ step2 } q ?tau q' \rangle$ 
       $\langle \neg ?tau a \longrightarrow (\exists pq1 pq2. ?steps \text{ step2 } q ?tau pq1 \wedge \text{step2 } pq1 a pq2$ 
         $\wedge ?steps \text{ step2 } pq2 ?tau q') \rangle$ 
      using assms(3) 'R p q' unfolding lts_tau.weak_simulation_def by blast
    hence outer_goal_a:  $\langle R p' q' \rangle$  using not_sink by blast
    show  $\langle \exists q'. (p' = \text{sink} \vee R p' q') \wedge$ 
      ( $?tau a \longrightarrow ?steps \text{ step } q ?tau q'$ )  $\wedge$ 
      ( $\neg ?tau a \longrightarrow (\exists pq1 pq2. ?steps \text{ step } q ?tau pq1 \wedge \text{step } pq1 a pq2$ 
         $\wedge ?steps \text{ step } pq2 ?tau q') \rangle$ 
    proof (cases  $\langle q' = \text{sink} \rangle$ )
      assume  $\langle q' = \text{sink} \rangle$ 
      then obtain q'' where q''_def:
         $\langle ?tau a \longrightarrow (?steps \text{ step } q ?tau q'' \wedge ?steps \text{ step2 } q'' ?tau q') \rangle$ 
         $\langle \neg ?tau a \longrightarrow (\exists pq1. ?steps \text{ step2 } q ?tau pq1 \wedge \text{step } pq1 a q''$ 
           $\wedge ?steps \text{ step2 } q'' ?tau q') \rangle$ 
        using q'_def(2,3) assms(1) step2_def lts_tau.step_tau_refl[of  $\tau$ ]
          lts_tau.tau_tau[of  $\tau$ ] by metis
      hence  $\langle q'' = \text{sink} \longrightarrow q = \text{sink} \rangle$ 
        using assms(2) unfolding step2_def by (metis lts.steps.cases)
      have  $\langle ?steps \text{ step2 } q'' ?tau q' \rangle$  using q''_def by auto
      hence  $\langle p' = \text{sink} \vee R p' q' \rangle$  using q'_def(1) assms(4)[of p' q' q''] by blast
      moreover have  $\langle \neg ?tau a \longrightarrow (\exists pq1 pq2. ?steps \text{ step } q ?tau pq1 \wedge \text{step } pq1 a pq2$ 
         $\wedge ?steps \text{ step } pq2 ?tau q') \rangle$ 
    proof
      assume  $\langle \neg ?tau a \rangle$ 
      hence  $\langle q \neq \text{sink} \rangle$  using q'_def by (metis assms(2) lts.steps_left step2_def)
      hence  $\langle q'' \neq \text{sink} \rangle$  using 'q'' = sink  $\longrightarrow q = \text{sink}$ ' by simp
      obtain pq1 where pq1_def:
         $\langle ?steps \text{ step2 } q ?tau pq1 \rangle \langle \text{step } pq1 a q'' \rangle \langle ?steps \text{ step2 } q'' ?tau q' \rangle$ 
        using q''_def(2) ' $\neg ?tau a$ ' by blast
      hence  $\langle pq1 \neq \text{sink} \rangle$  using 'q''  $\neq \text{sink}$ ' assms(2) by blast
      hence  $\langle ?steps \text{ step } q ?tau pq1 \rangle$  using 'q  $\neq \text{sink}$ ' ' $?steps \text{ step2 } q ?tau pq1$ '

```

```

proof (induct rule: lts.steps.induct[OF '?steps step2 q ?tau pq1'])
  case (1 p af)
  then show ?case using lts.steps.refl[of step p af] by blast
next
  case (2 p af q1 a q)
  hence <q1 ≠ sink> <step q1 a q> using assms(2) unfolding step2_def by auto
  moreover hence <?steps step p af q1> using 2 by blast
  ultimately show ?case using 2(4) by (meson lts.step)
qed
thus
  <∃pq1 pq2. ?steps step q ?tau pq1 ∧ step pq1 a pq2 ∧ ?steps step pq2 ?tau q''>
  using pq1_def(2) lts.steps.refl[of step q'' ?tau] by blast
qed
ultimately show <∃q''. (p' = sink ∨ R p' q'') ∧
  (?tau a ⟶ ?steps step q ?tau q'') ∧
  (¬ ?tau a ⟶ (∃pq1 pq2. ?steps step q ?tau pq1 ∧ step pq1 a pq2
    ∧ ?steps step pq2 ?tau q''))>
  using q''_def(1) q'_def(1) by auto
next
  assume not_sink_q': <q' ≠ sink>
  have outer_goal_b: <?tau a ⟶ ?steps step q ?tau q'>
    using q'_def(2) not_sink_q' unfolding step2_def
  proof (safe)
    assume i:
      <q' ≠ sink> <?tau a>
      <?steps (λp1 a p2. p1 ≠ sink ∧ a = ?tauEx ∧ p2 = sink ∨ step p1 a p2) q ?tau q'>
    thus <?steps step q ?tau q'>
  proof (induct rule: lts.steps.induct[OF i(3)])
    case (1 p af)
    then show ?case using lts.steps.refl[of _ p af] by auto
  next
    case (2 p af q1 a q)
    hence <step q1 a q> by blast
    moreover have <?steps step p af q1> using 2 assms(2) by blast
    ultimately show ?case using 'af a' lts.step[of step p af q1 a q] by blast
  qed
qed
have outer_goal_c:
  <¬ ?tau a ⟶ (∃pq1 pq2. ?steps step q ?tau pq1 ∧ step pq1 a pq2
    ∧ ?steps step pq2 ?tau q'')>
  using q'_def(3)
proof safe
  fix pq1 pq2
  assume subassms:
    <¬ ?tau a>
    <?steps step2 q ?tau pq1>
    <step2 pq1 a pq2>
    <?steps step2 pq2 ?tau q'>
  have <pq2 ≠ sink>
    using subassms(4) assms(2) not_sink_q' lts.steps_loop
    unfolding step2_def by (metis (mono_tags, lifting))
  have goal_c: <?steps step pq2 ?tau q'>
    using subassms(4) not_sink_q' unfolding step2_def
  proof (induct rule: lts.steps.induct[OF subassms(4)])
    case (1 p af) show ?case using lts.steps.refl by assumption
  next
    case (2 p af q1 a q)

```

```

    hence ⟨step q1 a q⟩ unfolding step2_def by simp
    moreover hence ⟨q1 ≠ sink⟩ using assms(2) by blast
    ultimately have ⟨?steps step p af q1⟩ using 2 unfolding step2_def by auto
    thus ?case using 'step q1 a q' 2(4) lts.step[of step p af q1 a q] by blast
qed
have goal_b: ⟨step pq1 a pq2⟩
  using 'pq2 ≠ sink' subassms(3) unfolding step2_def by blast
hence ⟨pq1 ≠ sink⟩ using assms(2) by blast
hence goal_a: ⟨?steps step q ?tau pq1⟩
  using subassms(4) unfolding step2_def
proof (induct rule:lts.steps.induct[OF subassms(2)])
  case (1 p af) show ?case using lts.steps.refl by assumption
next
  case (2 p af q1 a q)
  hence ⟨step q1 a q⟩ unfolding step2_def by simp
  moreover hence ⟨q1 ≠ sink⟩ using assms(2) by blast
  ultimately have ⟨?steps step p af q1⟩ using 2 unfolding step2_def by auto
  thus ?case using 'step q1 a q' 2(4) lts.step[of step p af q1 a q] by blast
qed
thus
  ⟨∃pq1 pq2. ?steps step q ?tau pq1 ∧ step pq1 a pq2 ∧ ?steps step pq2 ?tau q'⟩
  using goal_b goal_c by auto
qed
thus ⟨∃q'. (p' = sink ∨ R p' q') ∧ (?tau a ⟶ ?steps step q ?tau q') ∧
  (¬ ?tau a ⟶ (∃pq1 pq2. ?steps step q ?tau pq1 ∧ step pq1 a pq2
    ∧ ?steps step pq2 ?tau q'))⟩
  using outer_goal_a outer_goal_b by auto
qed
qed
qed

```

lemma simulation\_sink\_invariant:

```

  fixes
    step sink τ R
  defines
    ⟨step2 ≡ λ p1 a p2 . (p1 ≠ sink ∧ a = τ ∧ p2 = sink) ∨ step p1 a p2⟩
  assumes
    ⟨∧ a p . ¬ step sink a p ∧ ¬ step p a sink⟩
  shows ⟨lts_tau.weakly_simulated_by step2 τ p q = lts_tau.weakly_simulated_by step τ p q⟩
proof (rule)
  have sink_sim_min: ⟨lts_tau.weak_simulation step2 τ (λ p q. p = sink)⟩
    unfolding lts_tau.weak_simulation_def step2_def using assms(2)
    by (meson lts.steps.simps)
  define R where ⟨R ≡ lts_tau.weakly_simulated_by step2 τ⟩
  have weak_sim_R: ⟨lts_tau.weak_simulation step2 τ R⟩
    using lts_tau.weaksim_greatest[of step2 τ] unfolding R_def by blast
  have R_contains_inv_tau_closure:
    ⟨R = (λp1 q1. R p1 q1 ∨ lts.steps step2 q1 (lts_tau.tau τ) p1)⟩ unfolding R_def
  proof (rule, rule, rule, simp)
    fix p q
    assume
      ⟨(∃R. lts_tau.weak_simulation step2 τ R ∧ R p q) ∨
        (lts.steps step2 q (lts_tau.tau τ) p)⟩
    thus ⟨∃R. lts_tau.weak_simulation step2 τ R ∧ R p q⟩
      using weak_sim_R
        lts_tau.weak_sim_tau_step[of step2 ⟨τ⟩]
        lts_tau.weak_sim_union_cl[of step2 ⟨τ⟩] by blast
  
```

```

qed
assume Rpq: (R p q)
have (⟨∧ p' q' q . R p' q' ∧ lts.steps step2 q (lts_tau.tau τ) q' ⟶ R p' q⟩
  using R_contains_inv_tau_closure lts_tau.weak_sim_trans[of step2 ⟨τ⟩ _ _ _] R_def assms(2)
  by meson)
hence closed_R:
  (⟨∧ p' q' q . (p' = sink ∨ R p' q') ∧ lts.steps step2 q (lts_tau.tau τ) q'
    ⟶ (p' = sink ∨ R p' q)⟩
  using weak_sim_R sink_sim_min lts_tau.weak_sim_union_cl by blast)
have (⟨lts_tau.weak_simulation step2 τ (λp q. p = sink ∨ R p q)⟩
  using weak_sim_R sink_sim_min lts_tau.weak_sim_union_cl[of step2 τ] by blast)
hence (⟨lts_tau.weak_simulation step τ (λp q. p = sink ∨ R p q)⟩
  using simulation_tau_sink_2[of step sink τ R] assms(2) closed_R
  unfolding step2_def by blast)
thus (⟨∃R. lts_tau.weak_simulation step τ R ∧ R p q⟩
  using Rpq weak_sim_R by blast)
next
show (⟨∃R. lts_tau.weak_simulation step τ R ∧ R p q ⟶
  ∃R. lts_tau.weak_simulation step2 τ R ∧ R p q⟩)
proof clarify
  fix R
  assume
    (lts_tau.weak_simulation step τ R)
    (R p q)
  hence (lts_tau.weak_simulation
    (λp1 a p2. p1 ≠ sink ∧ a = τ ∧ p2 = sink ∨ step p1 a p2) τ (λp q. p = sink ∨ R p q))
    using simulation_tau_sink_1[of step sink τ R] assms(2) unfolding step2_def by auto
  thus (⟨∃R. lts_tau.weak_simulation step2 τ R ∧ R p q⟩
    using 'R p q' unfolding step2_def by blast)
qed
qed
end

```

## 4 Contrsimulation

```

theory Contrsimulation
imports
  Weak_Relations
begin

```

```

context lts_tau
begin

```

### 4.1 Definition of Contrsimulation

```

definition contrasim ::
  ((s ⇒ s ⇒ bool) ⇒ bool)
where
  (contrasim R ≡ ∀ p q p' A . (∀ a ∈ set(A). a ≠ τ) ∧ R p q ∧ (p ⇒$ A p') ⟶
    (∃ q'. (q ⇒$ A q') ∧ R q' p'))

lemma contrasim_simpler_def:
  shows (contrasim R =
    (∀ p q p' A . R p q ∧ (p ⇒$ A p') ⟶ (∃ q'. (q ⇒$ A q') ∧ R q' p)))
proof -
  have (⟨∧A. ∀a∈set (filter (λa. a ≠ τ) A). a ≠ τ⟩ by auto)

```

```

then show ?thesis
  unfolding contrasim_def
  using word_steps_ignore_tau_addition word_steps_ignore_tau_removal
  by metis
qed
abbreviation contrasimulated_by :: ⟨'s ⇒ 's ⇒ bool⟩ ("_  $\sqsubseteq_c$  _" [60, 60] 65)
  where ⟨contrasimulated_by p q  $\equiv \exists R . \text{contrasim } R \wedge R p q$ ⟩

lemma contrasim_preorder_is_contrasim:
  shows ⟨contrasim ( $\lambda p q . p \sqsubseteq_c q$ )⟩
  unfolding contrasim_def
  by metis

lemma contrasim_preorder_is_greatest:
  assumes ⟨contrasim R⟩
  shows ⟨ $\bigwedge p q . R p q \implies p \sqsubseteq_c q$ ⟩
  using assms by auto

lemma contrasim_tau_step:
  ⟨contrasim ( $\lambda p1 q1 . q1 \mapsto^* \text{tau } p1$ )⟩
  unfolding contrasim_def
  using steps.simps tau_tau tau_word_concat
  by metis

lemma contrasim_trans_constructive:
  fixes R1 R2
  defines
    ⟨ $R \equiv \lambda p q . \exists pq . (R1 p pq \wedge R2 pq q) \vee (R2 p pq \wedge R1 pq q)$ ⟩
  assumes
    R1_def: ⟨contrasim R1⟩ ⟨R1 p pq⟩ and
    R2_def: ⟨contrasim R2⟩ ⟨R2 pq q⟩
  shows
    ⟨R p q⟩ ⟨contrasim R⟩
  using assms(2,3,4,5) unfolding R_def contrasim_def by metis+

lemma contrasim_trans:
  assumes
    ⟨p  $\sqsubseteq_c$  pq⟩
    ⟨pq  $\sqsubseteq_c$  q⟩
  shows
    ⟨p  $\sqsubseteq_c$  q⟩
  using assms contrasim_trans_constructive by blast

lemma contrasim_refl:
  shows
    ⟨p  $\sqsubseteq_c$  p⟩
  using contrasim_tau_step steps.refl by blast

lemma contrasim_coupled:
  assumes
    ⟨contrasim R⟩
    ⟨R p q⟩
  shows
    ⟨ $\exists q' . q \mapsto^* \text{tau } q' \wedge R q' p$ ⟩
  using assms steps.refl[of p tau] weak_step_seq.simps(1)
  unfolding contrasim_simpler_def by blast

```

```

lemma contrasim_taufree_symm:
  assumes
    ⟨contrasim R⟩
    ⟨R p q⟩
    ⟨stable_state q⟩
  shows
    ⟨R q p⟩
  using contrasim_coupled assms stable_tauclosure_only_loop by blast

lemma symm_contrasim_is_weak_bisim:
  assumes
    ⟨contrasim R⟩
    ⟨ $\bigwedge p q. R p q \implies R q p$ ⟩
  shows
    ⟨weak_bisimulation R⟩
  using assms unfolding contrasim_simpler_def weak_sim_word weak_bisim_weak_sim by blast

lemma contrasim_weakest_bisim:
  assumes
    ⟨contrasim R⟩
    ⟨ $\bigwedge p q a. p \mapsto a q \implies \neg \text{tau } a$ ⟩
  shows
    ⟨bisimulation R⟩
  using assms contrasim_taufree_symm symm_contrasim_is_weak_bisim weak_bisim_taufree_strong
  by blast

lemma symm_weak_sim_is_contrasim:
  assumes
    ⟨weak_simulation R⟩
    ⟨ $\bigwedge p q. R p q \implies R q p$ ⟩
  shows
    ⟨contrasim R⟩
  using assms unfolding contrasim_simpler_def weak_sim_word by blast

```

## 4.2 Intermediate Relation Mimicking Contrasim

```

definition F :: "(( $'s \Rightarrow ('s \text{ set}) \Rightarrow \text{bool}$ )  $\Rightarrow ('s \Rightarrow ('s \text{ set}) \Rightarrow \text{bool})$ )" where
  ⟨F R p' Q'  $\equiv \exists p Q A. R p Q \wedge p \Rightarrow \$A p' \wedge (\forall a \in \text{set } A. a \neq \tau) \wedge Q' = (\text{dsuccs\_seq\_rec } (\text{rev } A) Q)$ ⟩

```

```

definition set_type :: "(' $s \Rightarrow 's \Rightarrow \text{bool}$ )  $\Rightarrow ('s \Rightarrow ('s \text{ set}) \Rightarrow \text{bool})$ " where
  ⟨set_type R p Q  $\equiv \exists q. R p q \wedge Q = \{q\}$ ⟩

```

```

lemma R_is_in_F_of_R :
  assumes ⟨R p Q⟩
  shows ⟨F R p Q⟩
proof -
  have Q_cond: ⟨Q = dsuccs_seq_rec (rev []) Q⟩ by simp
  have p_cond: ⟨p  $\Rightarrow \$[] p$ ⟩ using steps.simps weak_step_seq.simps(1) by blast
  thus ⟨F R p Q⟩ using Q_cond F_def ⟨R p Q⟩
  by (metis bex_empty empty_set tau_tau)
qed

```

```

lemma F_of_C_guarantees_tau_succ:
  assumes
    ⟨contrasim C⟩
    ⟨F (set_type C) p Q⟩

```

```

    (p  $\Rightarrow^{\tau}$  p')
  shows (∃ q'. q' ∈ (weak_tau_succs Q) ∧ F (set_type C) q' {p'})
proof -
  obtain p0 Q0 A q0
    where (set_type C) p0 Q0 (p0  $\Rightarrow$  $A p) (∀ a ∈ set A. a ≠ τ) (Q0 = {q0})
      and Q_def: (Q = (dsuccs_seq_rec (rev A) Q0))
    using F_def assms set_type_def by metis
  hence (C p0 q0) using set_type_def by auto
  have (p0  $\Rightarrow$  $(A@[τ]) p') using (p0  $\Rightarrow$  $A p) (p  $\Rightarrow^{\tau}$  p') rev_seq_step_concat by auto
  hence word: (p0  $\Rightarrow$  $A p')
    by (metis (∀ a ∈ set A. a ≠ τ) app_tau_taufree_list tau_def weak_step_over_tau)
  then obtain q' where (q0  $\Rightarrow$  $A q') (C q' p')
    using assms contrasim_def[of (C)] (C p0 q0) (∀ a ∈ set A. a ≠ τ) by blast
  hence ((set_type C) q' {p'}) using set_type_def by auto
  hence inF: (F (set_type C) q' {p'}) using R_is_in_F_of_R by auto
  have (q' ∈ weak_tau_succs (dsuccs_seq_rec (rev A) Q0))
    using (Q0 = {q0}) (q0  $\Rightarrow$  $A q')
    by (simp add: word_reachable_implies_in_dsuccs)
  hence (q' ∈ weak_tau_succs Q) using Q_def by simp
  thus (∃ q'. q' ∈ weak_tau_succs Q ∧ F (set_type C) q' {p'}) using inF by auto
qed

lemma F_of_C_guarantees_action_succ :
  assumes
    (contrasim C)
    (F (set_type C) p Q)
    (p  $\Rightarrow$  a p')
    (a ≠ τ)
  shows (F (set_type C) p' (dsuccs a Q))
proof -
  obtain p0 Q0 A q0
    where (set_type C) p0 Q0 (p0  $\Rightarrow$  $A p) (Q0 = {q0}) (∀ a ∈ set A. a ≠ τ)
      and Q_def: (Q = (dsuccs_seq_rec (rev A) Q0))
    using F_def assms set_type_def by metis
  then obtain CS where CS_def: (contrasim CS ∧ CS p0 q0)
    using assms set_type_def by (metis singleton_inject)
  have notau: (∀ a ∈ set (A@[a]). a ≠ τ)
    using (a ≠ τ) (∀ a ∈ set A. a ≠ τ) by auto
  have (p  $\Rightarrow$  a p') using assms(3,4) steps.refl tau_def by auto
  hence word: (p0  $\Rightarrow$  $(A@[a]) p')
    using (p0  $\Rightarrow$  $A p) rev_seq_step_concat
    by (meson steps.step steps_concat)
  then obtain q' where (q0  $\Rightarrow$  $(A@[a]) q') ∧ CS q' p'
    using CS_def contrasim_def[of (CS)] notau
    by fastforce
  hence (q' ∈ weak_tau_succs (dsuccs_seq_rec (rev (A@[a])) {q0}))
    using word_reachable_implies_in_dsuccs by blast
  then obtain q1 where (q1 ∈ dsuccs_seq_rec (rev (A@[a])) {q0}) (q1  $\Rightarrow^{\tau}$  q')
    using weak_tau_succs_def[of (dsuccs_seq_rec (rev (A@[a])) {q0})] by auto
  thus ?thesis
    using word F_def[of ((set_type C))] ((set_type C) p0 Q0)
    (Q0 = {q0}) Q_def notau simp_dsuccs_seq_rev by meson
qed

end
end
theory Contrasim_Set_Game

```

```

imports
  Simple_Game
  Contrsimulation
begin

```

### 4.3 The Contrsimulation Preorder Set Game

```

datatype ('s, 'a) c_set_game_node =
  AttackerNode 's "'s set" |
  DefenderSimNode 'a 's "'s set" |
  DefenderSwapNode 's "'s set"

fun (in lts_tau) c_set_game_moves ::
  ⟨('s, 'a) c_set_game_node ⇒ ('s, 'a) c_set_game_node ⇒ bool⟩ where

  simulation_challenge:
    ⟨c_set_game_moves (AttackerNode p Q) (DefenderSimNode a p1 Q0) =
      (p ⇒a p1 ∧ Q = Q0 ∧ ¬ tau a)⟩ |

  simulation_answer:
    ⟨c_set_game_moves (DefenderSimNode a p1 Q) (AttackerNode p10 Q1) =
      (p1 = p10 ∧ Q1 = dsuccs a Q)⟩ |

  swap_challenge:
    ⟨c_set_game_moves (AttackerNode p Q) (DefenderSwapNode p1 Q0) =
      (p ⇒τ p1 ∧ Q = Q0)⟩ |

  swap_answer:
    ⟨c_set_game_moves (DefenderSwapNode p1 Q) (AttackerNode q1 P1) =
      (q1 ∈ weak_tau_succs Q ∧ P1 = {p1})⟩ |

  c_set_game_moves_no_step:
    ⟨c_set_game_moves _ _ = False⟩

fun c_set_game_defender_node :: ⟨('s, 'a) c_set_game_node ⇒ bool⟩ where
  ⟨c_set_game_defender_node (AttackerNode _ _) = False⟩ |
  ⟨c_set_game_defender_node (DefenderSimNode _ _ _) = True⟩ |
  ⟨c_set_game_defender_node (DefenderSwapNode _ _) = True⟩

```

### 4.4 Contrsimulation Implies Winning Strategy (Completeness)

```

locale c_set_game =
  lts_tau trans τ +
  simple_game c_set_game_moves c_set_game_defender_node initial
for
  trans :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩ and
  τ :: ⟨'a⟩ and
  initial :: ⟨('s, 'a) c_set_game_node⟩
begin

  fun strategy_from_F_of_C :: ⟨('s ⇒ ('s set) ⇒ bool) ⇒ ('s, 'a) c_set_game_node strategy⟩
  where

    ⟨strategy_from_F_of_C R ((DefenderSwapNode p1 Q)#play) =
      (AttackerNode (SOME q1 . (∃ q. (q ∈ Q ∧ q ⇒τ q1)) ∧ R q1 {p1}) {p1})⟩ |

    ⟨strategy_from_F_of_C R ((DefenderSimNode a p1 Q)#play) =

```

```

    (AttackerNode p1 (SOME Q1 . Q1 = dsuccs a Q ∧ R p1 Q1))> |

    (strategy_from_F_of_C _ _ = undefined)

lemma set_defender_pred_is_attacker:
  assumes
    ⟨c_set_game_defender_node n0⟩
    ⟨(n0#play) ∈ plays⟩
    ⟨initial = AttackerNode p0 Q0⟩
  shows ⟨∃p Q. (hd play) = AttackerNode p Q ∧ c_set_game_moves (hd play) n0⟩
proof -
  have ⟨n0 ≠ initial⟩ using assms (1, 3) by auto
  hence mov: ⟨c_set_game_moves (hd play) n0⟩ using assms(2)
    by (metis list.inject list.sel(1) plays.cases)
  from assms(1) have def_cases:
    ⟨∃p1 Q. (∃a. n0 = DefenderSimNode a p1 Q) ∨ n0 = DefenderSwapNode p1 Q⟩
    using c_set_game_defender_node.elims(2) by blast
  then obtain p1 Q where pQ_def: ⟨(∃a. n0 = DefenderSimNode a p1 Q) ∨ n0 = DefenderSwapNode
p1 Q⟩
    by auto
  hence ⟨∃p. (hd play) = AttackerNode p Q⟩
  proof (rule disjE)
    assume ⟨∃a. n0 = DefenderSimNode a p1 Q⟩
    then obtain a where a_def: ⟨n0 = DefenderSimNode a p1 Q⟩ ..
    thus ?thesis using c_set_game_moves.elims(2) [OF mov] c_set_game_node.distinct(5) by auto
  next
    assume ⟨n0 = DefenderSwapNode p1 Q⟩
    thus ?thesis using c_set_game_moves.elims(2) [OF mov] c_set_game_node.distinct(5) by auto
  qed
  thus ?thesis using mov by auto
qed

lemma second_elem_in_play_set :
  assumes
    ⟨(n0#play) ∈ plays⟩
    ⟨initial = AttackerNode p0 Q0⟩
    ⟨c_set_game_defender_node n0⟩
  shows ⟨hd play ∈ set (n0 # play)⟩
proof -
  from assms(2, 3) have ⟨n0 ≠ initial⟩ by auto
  hence ⟨play ∈ plays⟩ using assms(1) plays.cases no_empty_plays by blast
  hence play_split: ⟨∃x xs. play = x#xs⟩ using no_empty_plays
    using plays.cases by blast
  then obtain x where x_def: ⟨∃xs. play = x#xs⟩ ..
  have x_in_set: ⟨x ∈ set (n0#play)⟩ using x_def by auto
  have x_head: ⟨x = hd play⟩ using x_def by auto
  from x_in_set x_head show ⟨hd play ∈ set (n0 # play)⟩ by auto
qed

lemma def_sim_or_swap_before_atkNode:
  assumes
    ⟨c_set_game_moves n0 n1⟩
    ⟨n1 = AttackerNode p Q⟩
  shows
    ⟨(∃Qpred a. n0 = (DefenderSimNode a p Qpred)) ∨ (∃q Ppred. n0 = (DefenderSwapNode q Ppred)
  ∧ Q = {q})⟩
proof (cases n0 rule: c_set_game_node.exhaust)

```

```

    case (AttackerNode s T)
  hence ⟨c_set_game_moves (AttackerNode s T) (AttackerNode p Q)⟩ using assms by auto
  hence ⟨False⟩ by simp
  then show ?thesis by auto
next
  case (DefenderSimNode a s T)
  then show ?thesis using assms by auto
next
  case (DefenderSwapNode s T)
  hence ⟨c_set_game_moves (DefenderSwapNode s T) (AttackerNode p Q)⟩ using assms by auto
  then show ?thesis using DefenderSwapNode by auto
qed

lemma set_game_strategy_retains_F:
  assumes
    ⟨contrasim C⟩
    ⟨C p0 q0⟩
    ⟨initial = AttackerNode p0 {q0}⟩
    ⟨play ∈ plays_for_strategy (strategy_from_F_of_C (F (set_type C)))⟩
  shows ⟨n = AttackerNode p Q ⟹ n ∈ set play ⟹ F (set_type C) p Q ⟩
proof (induct arbitrary: n p Q rule: plays_for_strategy.induct[OF assms(4)])
  case init: 1
  hence ⟨n = initial⟩ by auto
  hence ⟨p = p0 ∧ Q = {q0}⟩ using assms(3) init.prem(1) by auto
  thus ⟨F (set_type C) p Q⟩ using assms R_is_in_F_of_R set_type_def by simp
next
  case p0moved: (2 n0 play)
  hence ⟨(n = strategy_from_F_of_C (F (set_type C)) (n0 # play)) ∨ (n ∈ set (n0#play))⟩ by
  auto
  thus ?case
  proof (rule disjE)
    assume ⟨n ∈ set (n0#play)⟩
    thus ?thesis using p0moved.prem(1) p0moved.hyps(1,2) by blast
  next
    assume strat: ⟨n = strategy_from_F_of_C (F (set_type C)) (n0 # play)⟩
    hence ⟨(∃ a Qpred. n0 = DefenderSimNode a p Qpred) ∨ (∃ q Ppred. n0 = DefenderSwapNode
    q Ppred ∧ Q = {q})⟩
    using def_sim_or_swap_before_atkNode[OF p0moved.hyps(4), of ⟨p⟩ ⟨Q⟩] p0moved.prem(1)
  by blast
    thus ?case
    proof (rule disjE)
      assume ⟨∃ a Qpred. n0 = DefenderSimNode a p Qpred⟩
      then obtain a Qpred where n0_def: ⟨n0 = DefenderSimNode a p Qpred⟩ by auto
      hence ⟨strategy_from_F_of_C (F (set_type C)) (n0#play)
      = AttackerNode p (SOME Q1. Q1 = dsuccs a Qpred ∧ (F (set_type C)) p Q1)⟩
      using strategy_from_F_of_C.simps(2) by auto
      hence Q_def: ⟨Q = (SOME Q1. Q1 = dsuccs a Qpred ∧ (F (set_type C)) p Q1)⟩
      using strat by (simp add: p0moved.prem(1))
      have ⟨∃ ppred. hd play = (AttackerNode ppred Qpred) ∧ c_set_game_moves (hd play) n0⟩

      using set_defender_pred_is_attacker strategy_plays_subset[OF p0moved.hyps(1)]
      assms(3, 4) n0_def by force
      then obtain ppred where ppred_def: ⟨hd play = (AttackerNode ppred Qpred)⟩
      and ⟨c_set_game_moves (hd play) n0⟩ by auto
      hence ⟨ppred =>a p⟩ ⟨a ≠ τ⟩ using n0_def by auto
      hence ⟨hd play ∈ set (n0 # play)⟩
      using second_elem_in_play_set strategy_plays_subset[OF p0moved.hyps(1)] assms(3) n0_def

```

```

    by (simp add: assms(4))
  hence ⟨F (set_type C) ppred Qpred⟩ using p0moved.hyps(2) ppred_def by blast
  hence ⟨F (set_type C) p (dsuccs a Qpred)⟩
    using ⟨ppred  $\Rightarrow$  a p⟩ assms(1,2,4) F_of_C_guarantees_action_succ ⟨a  $\neq$   $\tau$ ⟩ by auto
  hence ⟨ $\exists$  Q. Q = (dsuccs a Qpred)  $\wedge$  F (set_type C) p Q⟩ by auto
  from someI_ex[OF this] show ⟨F (set_type C) p Q⟩
    unfolding Q_def
    using n0_def p0moved.hyps(4) by auto
next
  assume ⟨ $\exists$  q Ppred. n0 = DefenderSwapNode q Ppred  $\wedge$  Q = {q}⟩
  then obtain q Ppred where n0_def: ⟨n0 = DefenderSwapNode q Ppred⟩ and Q_def: ⟨Q = {q}⟩

  by auto
  hence ⟨strategy_from_F_of_C (F (set_type C)) (n0#play)
    = AttackerNode (SOME p1. ( $\exists$  p. p  $\in$  Ppred  $\wedge$  p  $\Rightarrow$   $\tau$  p1)  $\wedge$  (F (set_type C)) p1 {q})⟩
{q}
    using strategy_from_F_of_C.simps(1) by auto
  hence p_def: ⟨p = (SOME p1. ( $\exists$  p. p  $\in$  Ppred  $\wedge$  p  $\Rightarrow$   $\tau$  p1)  $\wedge$  (F (set_type C)) p1 {q})⟩

  using strat p0moved.prem by auto
  have ⟨ $\exists$  qpred. hd play = (AttackerNode qpred Ppred)  $\wedge$  c_set_game_moves (hd play) n0⟩

  using set_defender_pred_is_attacker strategy_plays_subset[OF p0moved.hyps(1)]
    assms(3, 4) n0_def
  by force
  then obtain qpred where qpred_def: ⟨hd play = (AttackerNode qpred Ppred)⟩
    and qpred_move: ⟨c_set_game_moves (hd play) n0⟩ by auto
  hence p1: ⟨player1_position (hd play)⟩ by (simp add: player1_position_def)
  have qpred_q_move: ⟨qpred  $\Rightarrow$   $\tau$  q⟩ using qpred_def qpred_move n0_def by simp
  have ⟨hd play  $\in$  set (n0 # play)⟩
    using second_elem_in_play_set strategy_plays_subset[OF p0moved.hyps(1)] assms(3) n0_def
    by (simp add: assms(4))
  hence ⟨F (set_type C) qpred Ppred⟩ using p0moved.hyps(2) qpred_def by blast
  hence ⟨ $\exists$  p. p  $\in$  weak_tau_succs Ppred  $\wedge$  F (set_type C) p {q}⟩
    using qpred_q_move assms(1,2) F_of_C_guarantees_tau_succ by blast
  hence ⟨ $\exists$  p. ( $\exists$  p0. p0  $\in$  Ppred  $\wedge$  p0  $\Rightarrow$   $\tau$  p)  $\wedge$  F (set_type C) p {q}⟩
    using weak_tau_succs_def[of ⟨Ppred⟩] by blast
  from someI_ex[OF this] p_def have ⟨F (set_type C) p {q}⟩ by simp
  thus ⟨F (set_type C) p Q⟩ using Q_def by blast
qed
qed
next
  case p1moved: (3 n1 play n1')
  hence ⟨(n = n1')  $\vee$  (n  $\in$  set (n1#play))⟩ by auto
  thus ?case
  proof (rule disjE)
    assume ⟨n  $\in$  set (n1#play)⟩
    thus ?case using p1moved.prem p1moved.hyps(1,2) by blast
  next
    assume A1: ⟨n = n1'⟩
    hence ⟨c_set_game_defender_node n1'⟩
      using def_sim_or_swap_before_atkNode p1moved.hyps(3, 4) p1moved.prem(1)
      player1_position_def by fastforce
    hence ⟨False⟩ using A1 p1moved.prem(1) by auto
    thus ?case by auto
  qed
qed

```

```

lemma set_contrasim_game_complete:
  assumes
    ⟨contrasim C⟩
    ⟨C p0 q0⟩
    ⟨initial = AttackerNode p0 {q0}⟩
  shows ⟨player0_winning_strategy (strategy_from_F_of_C (F (set_type C)))⟩
  unfolding player0_winning_strategy_def
  proof (safe)
    fix play
    assume A1: ⟨play ∈ (plays_for_strategy (strategy_from_F_of_C (F (set_type C))))⟩
    thus ⟨player1_wins play ⟹ False⟩
      unfolding player1_wins_def
    proof -
      assume A: ⟨c_set_game_defender_node (hd play) ∧ (∄p'. c_set_game_moves (hd play) p')⟩

      have player0_has_succ_node: ⟨c_set_game_defender_node (hd play) ⟹ ∃p'. c_set_game_moves
        (hd play) p'⟩
      proof (induct rule: simple_game.plays_for_strategy.induct[OF A1])
        case init: 1
        have ⟨¬c_set_game_defender_node (hd [initial])⟩ by (simp add: assms)
        hence ⟨False⟩ using init.premis by simp
        then show ?case ..
      next
        case p0moved: (2 n0 play)
        from p0moved.hyps have ⟨c_set_game_defender_node n0⟩ by simp
        hence ⟨(∃a p1 q. n0 = (DefenderSimNode a p1 q)) ∨ (∃q P. n0 = DefenderSwapNode q P)⟩
          by (meson c_set_game_defender_node.elims(2))
        hence ⟨¬c_set_game_defender_node (strategy_from_F_of_C (F (set_type C)) (n0#play))⟩
          using p0moved.hyps(4)
          c_set_game_moves.elims(2)[of ⟨n0⟩ ⟨strategy_from_F_of_C (F (set_type C)) (n0#play)⟩]
          by force
        hence ⟨¬c_set_game_defender_node (hd (strategy_from_F_of_C (F (set_type C)) (n0 # play)
          # n0 # play))⟩
          by simp
        hence ⟨False⟩ using p0moved.premis ..
        then show ?case ..
      next
        case p1moved: (3 n1 play n1')
        hence ⟨¬c_set_game_defender_node n1⟩ using p1moved.hyps and player1_position_def by
      simp
      then obtain p Q where n1_def: ⟨n1 = AttackerNode p Q⟩
        using c_set_game_defender_node.elims(3) by auto
      hence in_F: ⟨F (set_type C) p Q⟩
        using set_game_strategy_retains_F[OF assms(1-3), of ⟨n1#play⟩, OF p1moved.hyps(1)]
        by auto
      have ⟨(∃a p1. n1' = DefenderSimNode a p1 Q) ∨ (∃p1. n1' = DefenderSwapNode p1 Q)⟩
        using p1moved.premis n1_def p1moved.hyps(4)
        by (metis c_set_game_defender_node.elims(2) list.sel(1) local.simulation_challenge
          local.swap_challenge)
      thus ?case
      proof (rule disjE)
        assume A: ⟨∃a p1. n1' = DefenderSimNode a p1 Q⟩
        then obtain a p1 where n1'_def: ⟨n1' = DefenderSimNode a p1 Q⟩ by auto
        have move: ⟨c_set_game_moves (AttackerNode p Q) (DefenderSimNode a p1 Q)⟩
          using p1moved.hyps n1_def n1'_def by auto
      qed
    qed
  qed

```

```

    hence ⟨p ⇒a p1⟩ by auto
    hence ⟨p ⇒τ a p1⟩ using steps.refl by blast
    hence ⟨F (set_type C) p1 (dsuccs a Q)⟩
      using F_of_C_guarantees_action_succ move
      by (metis in_F assms(1) simulation_challenge tau_tau)
    then obtain Q1 where ⟨Q1 = dsuccs a Q ∧ F (set_type C) p1 Q1⟩ by blast
    hence ⟨c_set_game_moves n1' (AttackerNode p1 Q1)⟩
      using A n1'_def by auto
    thus ⟨∃a. c_set_game_moves (hd (n1' # n1 # play)) a⟩ by auto
  next
    assume ⟨∃p1. n1' = DefenderSwapNode p1 Q⟩
    then obtain p1 where n1'_def: ⟨n1' = DefenderSwapNode p1 Q⟩ ..
    hence ⟨c_set_game_moves (AttackerNode p Q) (DefenderSwapNode p1 Q)⟩
      using plmoved.hyps(4) n1'_def by auto
    hence p_succ: ⟨p ⇒τ p1⟩ by auto
    hence ⟨∃q'. q' ∈ weak_tau_succs Q ∧ F (set_type C) q' {p1}⟩
      using in_F F_of_C_guarantees_tau_succ assms(1) by auto
    hence ⟨∃q1. q1 ∈ weak_tau_succs Q ∧ F (set_type C) q1 {p1}⟩ by auto
    hence ⟨∃q1 P1. c_set_game_moves n1' (AttackerNode q1 P1)⟩ using n1'_def by auto
    thus ⟨∃a. c_set_game_moves (hd (n1' # n1 # play)) a⟩ by auto
  qed
qed
hence ⟨False⟩ using A by auto
thus ?thesis by auto
qed
qed

lemma strategy_from_F_of_C_sound :
  assumes
    ⟨contrasim C⟩
    ⟨C p0 q0⟩
    ⟨initial = AttackerNode p0 {q0}⟩
  shows
    ⟨sound_strategy (strategy_from_F_of_C (F (set_type C)))⟩
  unfolding sound_strategy_def
proof (safe)
  fix n0 play
  assume A:
    ⟨n0 # play ∈ plays_for_strategy (strategy_from_F_of_C (F (set_type C)))⟩
    ⟨c_set_game_defender_node n0⟩
  hence ⟨(∃a p' Q. n0 = DefenderSimNode a p' Q) ∨ (∃p' Q. n0 = DefenderSwapNode p' Q)⟩
    by (meson c_set_game_defender_node.elims(2))
  thus ⟨c_set_game_moves n0 (strategy_from_F_of_C (F (set_type C)) (n0 # play))⟩
  proof (rule disjE)
    assume ⟨∃a p' Q. n0 = DefenderSimNode a p' Q⟩
    then obtain a p' Q where n0_def: ⟨n0 = DefenderSimNode a p' Q⟩ by auto
    then obtain p where p_def: ⟨hd play = AttackerNode p Q⟩ using A
      by (metis assms(3) set_defender_pred_is_attacker simulation_challenge strategy_plays_subset)

    hence ⟨c_set_game_moves (AttackerNode p Q) (DefenderSimNode a p' Q)⟩
      by (metis A n0_def assms(3) set_defender_pred_is_attacker strategy_plays_subset)
    hence ⟨p ⇒a p'⟩ ⟨¬ tau a⟩ by auto
    hence ⟨F (set_type C) p Q⟩
      using set_game_strategy_retains_F[OF assms] A p_def
      assms(3) second_elem_in_play_set strategy_plays_subset
      by fastforce
    hence ⟨F (set_type C) p' (dsuccs a Q)⟩

```

```

    using F_of_C_guarantees_action_succ (¬ tau a) (p => a p') assms(1) tau_tau by blast
  hence Q1_ex: (∃ Q'. Q' = dsuccs a Q ∧ F (set_type C) p' Q') by auto
  from n0_def have st: (strategy_from_F_of_C (F (set_type C)) (n0#play) =
    (AttackerNode p' (SOME Q1 . Q1 = dsuccs a Q ∧ (F (set_type C)) p' Q1))) by auto
  then obtain Q1 where
    (AttackerNode p' (SOME Q1 . Q1 = dsuccs a Q ∧ (F (set_type C)) p' Q1) = AttackerNode
  p' Q1)
    by blast
  hence Q1_def: (Q1 = (SOME Q1 . Q1 = dsuccs a Q ∧ (F (set_type C)) p' Q1))
    by auto
  have next_is_atk: (strategy_from_F_of_C (F (set_type C)) (n0#play) = (AttackerNode p'
  Q1))
    using st Q1_def by auto
  with someI_ex[OF Q1_ex] Q1_def have mov_cond: (Q1 = dsuccs a Q ∧ F (set_type C) p' Q1)
    by blast
  have (c_set_game_moves n0 (AttackerNode p' Q1)) using n0_def mov_cond by auto
  thus ?thesis using next_is_atk by auto
next
  assume (∃ p' Q. n0 = DefenderSwapNode p' Q)
  then obtain p' Q where n0_def: (n0 = DefenderSwapNode p' Q) by auto
  then obtain p where p_def: (hd play = AttackerNode p Q) using A
    by (metis assms(3) set_defender_pred_is_attacker swap_challenge strategy_plays_subset)

  hence (c_set_game_moves (AttackerNode p Q) (DefenderSwapNode p' Q))
    by (metis A n0_def assms(3) set_defender_pred_is_attacker strategy_plays_subset)
  hence (p =>^tau p') by auto
  hence (F (set_type C) p Q)
    using set_game_strategy_retains_F[OF assms] A p_def
      assms(3) second_elem_in_play_set strategy_plays_subset
    by fastforce
  hence (∃ q'. q' ∈ weak_tau_succs Q ∧ F (set_type C) q' {p'})
    using F_of_C_guarantees_tau_succ (p =>^tau p') assms(1) by auto
  hence q1_ex: (∃ q1. (∃ q. (q ∈ Q ∧ q =>^tau q1)) ∧ F (set_type C) q1 {p'}) using weak_tau_succs_def
  by auto
  hence strat: (strategy_from_F_of_C (F (set_type C)) (n0#play) =
    AttackerNode (SOME q1 . (∃ q. (q ∈ Q ∧ q =>^tau q1)) ∧ (F (set_type C)) q1 {p'}) {p'})
    using n0_def by auto
  then obtain q1 where
    (AttackerNode (SOME q1 . (∃ q. (q ∈ Q ∧ q =>^tau q1)) ∧ (F (set_type C)) q1 {p'}) {p'})
    = AttackerNode q1 {p'} by blast
  hence q1_def: (q1 = (SOME q1 . (∃ q. (q ∈ Q ∧ q =>^tau q1)) ∧ (F (set_type C)) q1 {p'}))
    by auto
  with someI_ex[OF q1_ex] have X: (∃ q. (q ∈ Q ∧ q =>^tau q1) ∧ F (set_type C) q1 {p'})
    by blast
  hence (q1 ∈ weak_tau_succs Q ∧ {p'} = {p'}) using weak_tau_succs_def by auto
  thus ?thesis using n0_def strat q1_def by auto
qed
qed

```

## 4.5 Winning Strategy Implies Contrasimulation (Soundness)

```

lemma move_DefSim_to_AtkNode :
  assumes
    (c_set_game_moves (DefenderSimNode a p Q) n0)
  shows (n0 = AttackerNode p (dsuccs a Q))
proof -
  have (∃ p1 Q1. n0 = AttackerNode p1 Q1)

```

```

    by (metis assms c_set_game_defender_node.elims(2) c_set_game_defender_node.elims(3) local.c_set_game_moves_no_step(6))
  then obtain p1 Q1 where n0_def: (n0 = AttackerNode p1 Q1) by auto
  hence (p = p1) using assms local.simulation_answer by blast
  from n0_def have (Q1 = dsuccs a Q)
    using assms local.simulation_answer by blast
  thus ?thesis using (p = p1) n0_def by auto
qed

lemma def_never_stuck_on_sim_pos:
  assumes (n0 = DefenderSimNode a p Q)
  shows (∃ Q'. c_set_game_moves n0 (AttackerNode p Q'))
proof -
  have (c_set_game_moves (DefenderSimNode a p Q) (AttackerNode p (dsuccs a Q))) by auto
  thus ?thesis using assms by auto
qed

lemma def_sim_pos_with_prefix_in_play:
  assumes
    (A ≠ [])
    (p ⇒ $A p1)
    (∀ a ∈ set A. a ≠ τ)
    (sound_strategy f)
    (play ∈ plays_for_strategy f)
    (hd play = AttackerNode p {q})
  shows (∃ play p0. ((DefenderSimNode (last A) p0 (dsuccs_seq_rec (rev (butlast A)) {q}))#play)
    ∈ plays_for_strategy f ∧ word_reachable_via_delay A p p0 p1) using assms(1-3)
proof (induct arbitrary: p1 rule: rev_nonempty_induct[OF assms(1)])
  case single: (1 a)
  hence (¬tau a) using (∀ a ∈ set A. a ≠ τ) by (simp add: tau_def)
  hence (p ⇒ $[a] p1) using single by auto
  hence p_step: (p ⇒ ^a p1) by blast
  then obtain p0 where (p ⇒▷ a p0) (p0 ⇒ ^τ p1) using Cons (¬tau a) steps.refl by auto
  hence (∃ n0. n0 = DefenderSimNode a p0 {q} ∧ c_set_game_moves (AttackerNode p {q}) n0)
    using assms(4) (¬tau a) by simp
  hence ((DefenderSimNode a p0 {q})#play) ∈ plays_for_strategy f using assms(5,6)
    by (metis c_set_game_defender_node.simps(1) list.collapse no_empty_plays
      player1_position_def plays_for_strategy.p1move strategy_plays_subset)
  hence inplay: ((DefenderSimNode (last [a]) p0 (dsuccs_seq_rec (rev (butlast [a])) {q}))#play)
    ∈ plays_for_strategy f
  by auto
  have (p ⇒ $(butlast [a]) p) by (simp add: steps.refl)
  hence (word_reachable_via_delay [a] p p0 p1) using word_reachable_via_delay_def (p ⇒▷ a p0)
    (p0 ⇒ ^τ p1) by auto
  then show ?case using inplay by auto
next
  case snoc: (2 a as)
  hence (¬tau a) using (∀ a ∈ set A. a ≠ τ) by (simp add: tau_def)
  then obtain a2 as2 where as_def: (as = as2@[a2]) using list_rev_split[OF snoc.hyps(1)] by
    auto
  have (∃ p'. p ⇒ $ as p' ∧ p' ⇒ $[a] p1) using rev_seq_split[OF snoc.prem(2)] by blast
  hence (∃ p'. p ⇒ $ as p' ∧ p' ⇒ ^a p1) by blast
  hence (∃ p'. p ⇒ $ as p' ∧ p' ⇒ a p1) using (¬tau a) by simp
  then obtain p' where p'_def: (p ⇒ $ as p') and p'_step: (p' ⇒ a p1) by auto
  then obtain p11 where (p' ⇒▷ a p11) (p11 ⇒ ^τ p1) using steps.refl (¬tau a) tau_tau by

```

```

blast
  hence ⟨∃ play p0.
    DefenderSimNode (last as) p0 (dsuccs_seq_rec (rev (butlast as)) {q}) # play
    ∈ plays_for_strategy f ∧ word_reachable_via_delay as p p0 p'⟩ using p'_def snoc by auto
  then obtain play p0
    where play_def: ⟨DefenderSimNode (last as) p0 (dsuccs_seq_rec (rev (butlast as)) {q})
# play
  ∈ plays_for_strategy f⟩ ⟨word_reachable_via_delay as p p0 p'⟩ by auto

  hence ⟨DefenderSimNode a2 p0 (dsuccs_seq_rec (rev as2) {q}) # play
    ∈ plays_for_strategy f⟩ using as_def by auto
  then obtain n0 where n0_def: ⟨n0 = DefenderSimNode a2 p0 (dsuccs_seq_rec (rev as2) {q})⟩

    and n0_in_play: ⟨n0#play ∈ plays_for_strategy f⟩ by auto
  then obtain n1
    where n1_def: ⟨c_set_game_moves (DefenderSimNode a2 p0 (dsuccs_seq_rec (rev as2) {q}))
n1)
  using def_never_stuck_on_sim_pos by meson
  hence n1_atk: ⟨n1 = AttackerNode p0 (dsuccs a2 ((dsuccs_seq_rec (rev as2) {q})))⟩
  using move_DefSim_to_AtkNode[OF n1_def] by auto
  have n1_in_play: ⟨n1#n0#play ∈ plays_for_strategy f⟩ using n1_def n0_in_play n0_def
  by (metis assms(4) move_DefSim_to_AtkNode c_set_game_defender_node.simps(2)
    plays_for_strategy.simps sound_strategy_def)
  then obtain n0'
    where n0'_def : ⟨n0' = DefenderSimNode a p11 (dsuccs a2 ((dsuccs_seq_rec (rev as2) {q})))⟩

    and n0'_mov: ⟨c_set_game_moves n1 n0'⟩ using p'_step n1_atk
  by (metis (no_types, lifting) ⟨¬ tau a⟩ ⟨p' ⇒ a p11⟩ word_reachable_via_delay_def simulation_challenge
play_def(2) steps_concat tau_tau)
  hence in_play: ⟨n0'#n1#n0#play ∈ plays_for_strategy f⟩ using n1_in_play
  by (simp add: n1_atk player1_position_def plays_for_strategy.p1move)
  hence ⟨n0' = DefenderSimNode a p11 (dsuccs_seq_rec (rev (as2@[a2])) {q})⟩ using n0'_def by
auto
  hence n0'_is_defSimNode: ⟨n0' = DefenderSimNode a p11 (dsuccs_seq_rec (rev (as)) {q})⟩ using
as_def by auto
  from ⟨p ⇒$ as p'⟩ ⟨p' ⇒ a p11⟩ ⟨p11 ⇒ τ p1⟩ have ⟨word_reachable_via_delay (as@[a]) p
p11 p1⟩
  using word_reachable_via_delay_def by auto
  then show ?case using n0'_is_defSimNode in_play by auto
qed

lemma set_contrasim_game_sound:
  assumes
    ⟨player0_winning_strategy f⟩
    ⟨sound_strategy f⟩
  defines
    C == λ p q . (∃ play ∈ plays_for_strategy f . hd play = AttackerNode p {q} ∧ (hd play
= initial ∨ (∃ P. hd (tl play) = DefenderSwapNode q P)))
  shows
    ⟨contrasim C⟩ unfolding contrasim_def
proof (safe)
  fix p q p1 A
  assume ⟨∀ a ∈ set A. a ≠ τ⟩ ⟨p ⇒$A p1⟩ and ⟨C p q⟩
  hence ⟨p ⇒$(taufree A) p1⟩ by (simp add: weak_step_over_tau)
  hence ⟨(∃ play ∈ plays_for_strategy f . hd play = AttackerNode p {q} ∧ (hd play = initial
∨ (∃ P. hd (tl play) = DefenderSwapNode q P)))⟩
  using C_def ⟨p ⇒$A p1⟩ and ⟨C p q⟩ by auto

```

```

from this obtain play where play_def: ⟨play ∈ plays_for_strategy f⟩
  and play_hd: ⟨hd play = AttackerNode p {q}⟩
  and ⟨hd play = initial ∨ (∃P. hd (tl play) = DefenderSwapNode q P)⟩
  by auto
hence ⟨¬player1_wins play⟩ using assms(1) player0_winning_strategy_def by auto
hence ⟨(c_set_game_defender_node (hd play) ∧ (∃p'. c_set_game_moves (hd play) p')) ⟹ False⟩

  using player1_wins_def by auto
hence Def_not_stuck: ⟨c_set_game_defender_node (hd play) ⟹ (∃p'. c_set_game_moves (hd
play) p')⟩ by auto
from ⟨p ⟹ $A p1⟩ ⟨p ⟹ $(taufree A) p1⟩ ⟨C p q⟩ show ⟨∃q'. q ⟹ $ A q' ∧ C q' p1⟩
proof (cases A rule: rev_cases)
  case Nil
  hence ⟨p ⟹ ^τ p1⟩ using ⟨p ⟹ $A p1⟩ by auto
  hence ⟨∃n0. n0 = DefenderSwapNode p1 {q} ∧ c_set_game_moves (AttackerNode p {q}) n0⟩ by
simp
  from this obtain n0 where n0_def: ⟨n0 = DefenderSwapNode p1 {q}⟩
  and n0_move: ⟨c_set_game_moves (AttackerNode p {q}) n0⟩ by auto
  have ⟨play = (hd play)#(tl play)⟩
  by (metis hd_Cons_tl no_empty_plays play_def strategy_plays_subset)
  hence ⟨n0#play ∈ plays_for_strategy f⟩ using n0_def n0_move play_def play_hd
  by (metis c_set_game_defender_node.simps(1) play_def
    player1_position_def plays_for_strategy.p1move)
  hence ⟨∃n1'. c_set_game_moves n0 n1' ∧ n1'#n0#play ∈ plays_for_strategy f⟩
  using assms(2) n0_def sound_strategy_def
  by (meson c_set_game_defender_node.simps(3) plays_for_strategy.p0move)
  then obtain n1' where n1'_mov: ⟨c_set_game_moves n0 n1'⟩
  and in_play: ⟨n1'#n0#play ∈ plays_for_strategy f⟩ by auto
  hence ⟨∃q1. n1' = AttackerNode q1 {p1} ∧ (q1 ∈ weak_tau_succs {q})⟩
  by (metis c_set_game_defender_node.elims(2, 3) c_set_game_moves_no_step(3, 4) swap_answer
n0_def)
  then obtain q1 where q1_def: ⟨n1' = AttackerNode q1 {p1}⟩
  and q_succ: ⟨q1 ∈ weak_tau_succs {q}⟩ by auto
  hence q_tau: ⟨q ⟹ ^τ q1⟩ using weak_tau_succs_def by auto
  from in_play q1_def n0_def have ⟨C q1 p1⟩ unfolding C_def by force
  then show ?thesis using q_tau Nil by auto
next
case (snoc as a)
hence ⟨A ≠ []⟩ by auto
hence ⟨¬tau a⟩ using ⟨∀a∈set A. a ≠ τ⟩ snoc by (simp add: tau_def)
then obtain A_play p0 where
  gotoA: ⟨DefenderSimNode (last A) p0 (dsuccs_seq_rec (rev (butlast A)) {q}) # A_play
    ∈ plays_for_strategy f⟩ ⟨word_reachable_via_delay A p p0 p1⟩
  using def_sim_pos_with_prefix_in_play ⟨p ⟹ $A p1⟩
  ⟨∀a∈set A. a ≠ τ⟩ ⟨A ≠ []⟩ assms(2) play_def play_hd by meson
  then obtain Q where ⟨Q = dsuccs_seq_rec (rev (butlast A)) {q}⟩ by auto
  hence ⟨∀q' ∈ Q. q ⟹ $(butlast A) q'⟩ using in_dsuccs_implies_word_reachable by auto
  then obtain n0 where n0_def: ⟨n0 = DefenderSimNode a p0 (dsuccs_seq_rec (rev as) {q})⟩
  by auto
  hence A_play_def: ⟨n0#A_play ∈ plays_for_strategy f⟩ using gotoA snoc by auto
  then obtain n1 where n1_move: ⟨c_set_game_moves n0 n1⟩
  using n0_def by (meson assms(2) c_set_game_defender_node.simps(2) sound_strategy_def)
  hence ⟨n1 = AttackerNode p0 (dsuccs a (dsuccs_seq_rec (rev as) {q}))⟩
  using move_DefSim_to_AtkNode n0_def by blast
  hence ⟨n1 = AttackerNode p0 (dsuccs_seq_rec (a#(rev as)) {q})⟩
  using dsuccs_seq_rec.simps(2) by auto
  hence ⟨n1 = AttackerNode p0 (dsuccs_seq_rec (rev (as@[a])) {q})⟩ by auto

```

```

hence n1_def: ⟨n1 = AttackerNode p0 (dsuccs_seq_rec (rev A) {q})⟩ using snoc by auto
hence n1_in_play: ⟨n1#n0#A_play ∈ plays_for_strategy f⟩
  using n0_def A_play_def n1_move assms(2) move_DefSim_to_AtkNode
    plays_for_strategy.p0move sound_strategy_def
  by fastforce
from ⟨word_reachable_via_delay A p p0 p1⟩ have ⟨p0 ⇒τ p1⟩
  using word_reachable_via_delay_def by auto
then obtain n0' where n0'_move: ⟨c_set_game_moves n1 n0'⟩
  and n0'_def: ⟨n0' = DefenderSwapNode p1 (dsuccs_seq_rec (rev A) {q})⟩
  using swap_challenge tau_tau n1_def by blast
hence n0'_in_play: ⟨n0'#n1#n0#A_play ∈ plays_for_strategy f⟩
  using n1_in_play by (simp add: n1_def player1_position_def plays_for_strategy.p1move)

then obtain n1' where n1'_move: ⟨c_set_game_moves n0' n1'⟩
  and in_strat: ⟨n1' = f(n0'#n1#n0#A_play)⟩
  using Def_not_stuck n0'_def assms(2) sound_strategy_def by auto
then obtain q1 where q1_def: ⟨q1 ∈ weak_tau_succs (dsuccs_seq_rec (rev A) {q})⟩
  and n1'_def: ⟨n1' = AttackerNode q1 {p1}⟩ using n0'_def
  by (metis c_set_game_defender_node.cases c_set_game_moves_no_step(3, 7) swap_answer)

hence ⟨q1 ∈ {q1. ∃q0 ∈ (dsuccs_seq_rec (rev A) {q}). q0 ⇒τ q1}⟩
  using weak_tau_succs_def by auto
also have ⟨... = {q1. ∃q0 ∈ (dsuccs_seq_rec (rev A) {q}). q ⇒$A q0 ∧ q0 ⇒τ q1}⟩
  using in_dsuccs_implies_word_reachable by auto
also have ⟨... ⊆ {q1. ∃q0 ∈ (dsuccs_seq_rec (rev A) {q}). q ⇒$A q1}⟩
  using word_tau_concat by auto
also have ⟨... ⊆ {q1. q ⇒$A q1}⟩ by auto
finally have ⟨q1 ∈ {q1. q ⇒$A q1}⟩ .
hence q_goal: ⟨q ⇒$A q1⟩ by auto
from n1'_move in_strat have move_f: ⟨c_set_game_moves n0' (f(n0'#n1#n0#A_play))⟩ by auto
hence ⟨n1'#n0'#n1#n0#A_play ∈ plays_for_strategy f⟩
  using in_strat plays_for_strategy.p0move[OF n0'_in_play _ move_f] n0'_def by auto
hence ⟨C q1 p1⟩ unfolding C_def using n1'_def n0'_def by force
thus ?thesis using q_goal by auto
qed
qed

theorem winning_strategy_in_set_game_iff_contrasim:
  assumes
    ⟨initial = AttackerNode p {q}⟩
  shows
    ⟨(∃ f . player0_winning_strategy f ∧ sound_strategy f) = (∃ C. contrasim C ∧ C p q)⟩
proof
  assume
    ⟨(∃ f. player0_winning_strategy f ∧ sound_strategy f)⟩
  then obtain f where
    ⟨contrasim
      (λp q. ∃play∈plays_for_strategy f.
        hd play = AttackerNode p {q} ∧
        (hd play = initial ∨ (∃P. hd (tl play) = DefenderSwapNode q P)))⟩
  using set_contrasim_game_sound by blast
moreover have ⟨(λp q. ∃play∈plays_for_strategy f.
  hd play = AttackerNode p {q} ∧
  (hd play = initial ∨ (∃P. hd (tl play) = DefenderSwapNode q P))) p q⟩
  using assms plays_for_strategy.init[of f] by (meson list.sel(1))
ultimately show ⟨∃ C. contrasim C ∧ C p q⟩ by blast
next

```

```

    assume
      (∃ C. contrasim C ∧ C p q)
    thus (∃ f. player0_winning_strategy f ∧ sound_strategy f)
      using set_contrasim_game_complete[OF _ _ assms]
        strategy_from_F_of_C_sound[OF _ _ assms] by blast
qed

end
end
theory Basic_Contrasim_Game
imports
  Simple_Game
  Contrasimulation
begin

```

## 4.6 The Basic Contrasimulation Preorder Game

```

datatype ('s, 'a) c_basic_game_node =
  AttackerNode 's 's |
  DefenderNode "'a list" 's 's

fun (in lts_tau) c_game_moves ::
  ⟨('s, 'a) c_basic_game_node ⇒ ('s, 'a) c_basic_game_node ⇒ bool⟩ where

  simulation_challenge:
    ⟨c_game_moves (AttackerNode p q) (DefenderNode A p1 q0) =
      (p ⇒$A p1 ∧ q = q0 ∧ (∀a∈set A. a ≠ τ))⟩ |

  simulation_answer:
    ⟨c_game_moves (DefenderNode A p1 q0) (AttackerNode q1 p10) =
      (q0 ⇒$A q1 ∧ p1 = p10)⟩ |

  c_game_moves_no_step:
    ⟨c_game_moves _ _ = False⟩

fun c_game_defender_node :: ⟨('s, 'a) c_basic_game_node ⇒ bool⟩ where
  ⟨c_game_defender_node (AttackerNode _ _) = False⟩ |
  ⟨c_game_defender_node (DefenderNode _ _ _) = True⟩

```

## 4.7 Contrasimulation Implies Winning Strategy (Completeness)

```

locale c_basic_game =
  lts_tau trans τ +
  simple_game c_game_moves c_game_defender_node initial
for
  trans :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩ and
  τ :: ⟨'a⟩ and
  initial :: ⟨('s, 'a) c_basic_game_node⟩
begin

fun strategy_from_contrasim :: ⟨('s ⇒ 's ⇒ bool) ⇒ ('s, 'a) c_basic_game_node strategy⟩
where
  ⟨strategy_from_contrasim R ((DefenderNode A p1 q0)#play) =
    (AttackerNode (SOME q1 . R q1 p1 ∧ q0 ⇒$A q1) p1)⟩ |
  ⟨strategy_from_contrasim _ _ = undefined⟩

lemma basic_defender_pred_is_attacker:

```

```

assumes
  <c_game_defender_node n0>
  <n0 = DefenderNode A p' q>
  <(n0#play) ∈ plays>
  <initial = AttackerNode p0 q0>
shows <∃p. (hd play) = AttackerNode p q ∧ c_game_moves (hd play) n0>
proof -
  have <n0 ≠ initial> using assms (2, 4) by auto
  hence mov: <c_game_moves (hd play) n0> using assms(3)
    by (metis list.inject list.sel(1) plays.cases)
  hence <∃p. (hd play) = AttackerNode p q> using assms(1 - 3)
    by (metis c_basic_game_node.inject(2) c_game_defender_node.simps(1) c_game_moves.elims(2))
  thus ?thesis using mov by auto
qed

lemma second_elem_in_play_set :
  assumes
    <(n0#play) ∈ plays>
    <initial = AttackerNode p0 q0>
    <n0 = DefenderNode A p q>
  shows <hd play ∈ set (n0 # play)>
proof -
  from assms(2, 3) have <n0 ≠ initial> by auto
  hence <play ∈ plays> using assms(1) plays.cases no_empty_plays by blast
  hence play_split: <∃x xs. play = x#xs> using no_empty_plays
    using plays.cases by blast
  then obtain x where x_def: <∃xs. play = x#xs> ..
  have x_in_set: <x ∈ set (n0#play)> using x_def by auto
  have <x = hd play> using x_def by auto
  with x_in_set show <hd play ∈ set (n0 # play)> by auto
qed

lemma basic_game_all_f_consistent_atk_pos_in_R:
  assumes
    <contrasim R>
    <R p0 q0>
    <initial = AttackerNode p0 q0>
    <play ∈ plays_for_strategy (strategy_from_contrasim R)>
  shows <((AttackerNode p q) ∈ set play) ⇒ R p q>
  using assms(4)
proof (induct arbitrary: p q rule: plays_for_strategy.induct[OF assms(4)])
  case 1
  fix p q
  assume <((AttackerNode p q) ∈ (set [initial]))>
  thus <R p q> using assms(3, 2) by simp
next
  case p0moved: (2 n0 play)
  hence IH: <AttackerNode p q ∈ set (n0#play) ⇒ R p q> by simp
  from p0moved.prem1 have <((AttackerNode p q) ∈ set ((strategy_from_contrasim R (n0 # play))#n0#play))>
    by simp
  hence <((AttackerNode p q = (strategy_from_contrasim R (n0 # play))) ∨ (AttackerNode p q
    ∈ set (n0#play)))>
    by simp
  thus <R p q>
proof (rule disjE)

```

```

    assume ⟨AttackerNode p q ∈ set (n0#play)⟩
    thus ⟨R p q⟩ using IH by simp
next
  assume A: ⟨AttackerNode p q = (strategy_from_contrasim R (n0 # play))⟩
  have ⟨∃A ppred. n0 = (DefenderNode A q ppred)⟩ using p0moved.hyps(3)
    strategy_from_contrasim.simps(1)[of ⟨R⟩]
    by (metis (no_types, lifting) A c_basic_game_node.inject(1) c_game_defender_node.elims(2))
  then obtain A ppred where n0_def: ⟨n0 = (DefenderNode A q ppred)⟩ ⟨∀a∈set A. a ≠ τ⟩
    by (metis assms(3) c_basic_game.basic_defender_pred_is_attacker simulation_challenge
p0moved.hyps(1, 3) strategy_plays_subset)
  hence ⟨strategy_from_contrasim R (n0#play) = AttackerNode (SOME q1. R q1 q ∧ ppred ⇒$
A q1) q⟩
    using n0_def strategy_from_contrasim.simps(1)[of ⟨R⟩ ⟨A⟩ ⟨q⟩ ⟨ppred⟩ ⟨play⟩] by auto
  hence p_def: ⟨p = (SOME p1. R p1 q ∧ ppred ⇒$ A p1)⟩ using A by auto
  have ⟨∃qpred. hd play = (AttackerNode qpred ppred) ∧ c_game_moves (hd play) n0⟩
    using basic_defender_pred_is_attacker strategy_plays_subset[OF p0moved.hyps(1)]
    by (simp add: assms(3) n0_def p0moved.hyps(3))
  then obtain qpred where qpred_def: ⟨hd play = (AttackerNode qpred ppred)⟩
    and qpred_move: ⟨c_game_moves (hd play) n0⟩ by auto
  have qpred_q_move: ⟨qpred ⇒$A q⟩ using qpred_def qpred_move n0_def by simp
  have ⟨hd play ∈ set (n0 # play)⟩
    using second_elem_in_play_set strategy_plays_subset[OF p0moved.hyps(1)] assms(3)
    by (auto simp add: n0_def)
  hence pred_R: ⟨R qpred ppred⟩
    by (simp add: qpred_def p0moved.hyps(1) p0moved.hyps(2))
  have ⟨∃ p1 . R p1 q ∧ ppred ⇒$ A p1⟩
    using qpred_q_move pred_R assms(1) ⟨∀a∈set A. a ≠ τ⟩
    unfolding contrasim_def by blast
  from someI_ex[OF this] show ⟨R p q⟩ unfolding p_def by blast
qed
next
  case p1moved: (3 n1 play n1')
  from p1moved.hyps have IH: ⟨AttackerNode p q ∈ set (n1#play) ⟹ R p q⟩ by simp
  assume ⟨(AttackerNode p q) ∈ (set (n1'#n1#play))⟩
  hence ⟨(AttackerNode p q = n1') ∨ (AttackerNode p q ∈ set (n1#play))⟩ by auto
  thus ⟨R p q⟩
  proof (rule disjE)
    assume ⟨(AttackerNode p q ∈ set (n1#play))⟩
    thus ⟨R p q⟩ using p1moved.hyps by auto
  next
    assume A: ⟨AttackerNode p q = n1'⟩
    from p1moved.hyps have ⟨player1_position n1⟩ by simp
    hence ⟨c_game_defender_node n1'⟩
      by (metis c_game_defender_node.simps(2) c_game_moves.elims(2) p1moved.hyps(4) player1_position_def)
    hence ⟨False⟩ using A by auto
    thus ⟨R p q⟩ ..
  qed
qed

```

lemma basic\_contrasim\_game\_complete:

```

  assumes
    ⟨contrasim R⟩
    ⟨R p q⟩
    ⟨initial = AttackerNode p q⟩
  shows ⟨player0_winning_strategy (strategy_from_contrasim R)⟩
    unfolding player0_winning_strategy_def

```

```

proof (safe)
  fix play
  assume A1: (play ∈ (plays_for_strategy (strategy_from_contrasim R)))
  thus (player1_wins play ⇒ False)
    unfolding player1_wins_def
  proof -
    assume A: (c_game_defender_node (hd play) ∧ (∄p'. c_game_moves (hd play) p'))
    have player0_has_succ_node: (c_game_defender_node (hd play) ⇒ ∃p'. c_game_moves (hd
play) p')
    proof (induct rule: simple_game.plays_for_strategy.induct[OF A1])
      case init: 1
      from assms(3) have (¬c_game_defender_node (hd [initial])) by simp
      hence (False) using init.premis by simp
      then show ?case ..
    next
      case p0moved: (2 n0 play)
      from p0moved.hyps have (c_game_defender_node n0) by simp
      hence (∃A p1 q. n0 = (DefenderNode A p1 q))
        by (meson c_game_defender_node.elims(2))
      hence (¬c_game_defender_node (strategy_from_contrasim R (n0#play)))
        using p0moved.hyps(4) c_game_moves.elims(2)[of (n0) (strategy_from_contrasim R (n0#play))]
        by force
      hence (¬c_game_defender_node (hd (strategy_from_contrasim R (n0 # play) # n0 # play)))

        by simp
      hence (False) using p0moved.premis ..
      then show ?case ..
    next
      case p1moved: (3 n1 play n1')
      hence (¬c_game_defender_node n1) using p1moved.hyps and player1_position_def by simp
      then obtain p q where n1_def: (n1 = AttackerNode p q)
        using c_game_defender_node.elims(3) by auto
      hence pq_in_R: (R p q)
        using basic_game_all_f_consistent_atk_pos_in_R[OF assms, of (n1#play), OF p1moved.hyps(1)]

        by auto
      have is_def: (c_game_defender_node n1') using p1moved.premis by auto
      then obtain A p1 q0 where n1'_def: (n1' = DefenderNode A p1 q0)
        using c_game_defender_node.elims(2)[OF is_def] by auto
      hence (∀a∈set A. a ≠ τ) using p1moved.hyps(4) n1_def by simp
      have Move_n1_n1': (c_game_moves (AttackerNode p q) (DefenderNode A p1 q0))
        using p1moved.hyps n1_def n1'_def by auto
      hence same_q: (q0 = q) by auto
      from Move_n1_n1' have p_succ: (p ⇒ $A p1) by auto
      from assms(1) have Contra: (∧p q p'. A. (∀a∈set A. a ≠ τ) ⇒ R p q ⇒ p ⇒ $ A p'
⇒ (∃q'. q ⇒ $ A q' ∧ R q' p'))
        unfolding contrasim_def by auto
      hence (∃ q'. (q ⇒ $ A q') ∧ R q' p1) using Contra[OF (∀a∈set A. a ≠ τ) pq_in_R p_succ]
        by auto
      hence (∃ p1 q1. c_game_moves n1' (AttackerNode p1 q1)) using same_q n1'_def by auto
      then show ?case by auto
    qed
    thus (False) using A by auto
  qed
qed
lemma strategy_from_contrasim_sound :

```

```

assumes
  ⟨R p0 q0⟩
  ⟨contrasim R⟩
  ⟨initial = AttackerNode p0 q0⟩
shows
  ⟨sound_strategy (strategy_from_contrasim R)⟩
unfolding sound_strategy_def
proof (safe)
  fix n0 play
  assume A:
    ⟨n0 # play ∈ plays_for_strategy (strategy_from_contrasim R)⟩
    ⟨c_game_defender_node n0⟩
  then obtain A p1 q where n0_def: ⟨n0 = DefenderNode A p1 q⟩
    using c_game_defender_node.elims(2) by blast
  then obtain p where p_def: ⟨hd play = AttackerNode p q⟩
    using n0_def A basic_defender_pred_is_attacker assms(3) strategy_plays_subset by blast
  hence ⟨c_game_moves (AttackerNode p q) (DefenderNode A p1 q)⟩
    using A n0_def by (metis assms(3) basic_defender_pred_is_attacker strategy_plays_subset)
  hence mov_p_p1: ⟨p ⇒$A p1⟩ ⟨∀a∈set A. a ≠ τ⟩ by auto
  from p_def have ⟨R p q⟩
    using basic_game_all_f_consistent_atk_pos_in_R A assms
    second_elem_in_play_set n0_def strategy_plays_subset
    by fastforce
  with mov_p_p1 have q1_def: ⟨∃q1. R q1 p1 ∧ q ⇒$A q1⟩
    using assms(2) unfolding contrasim_def by blast
  from n0_def have ⟨strategy_from_contrasim R (n0 # play) = (AttackerNode (SOME q1 . R q1
p1 ∧ q ⇒$A q1) p1)⟩
    by auto
  then obtain q' where ⟨AttackerNode (SOME q1 . R q1 p1 ∧ q ⇒$A q1) p1 = AttackerNode q'
p1) by blast
  hence q'_def: ⟨q' = (SOME q1 . R q1 p1 ∧ q ⇒$A q1)⟩ by auto
  with someI_ex[OF q1_def] have ⟨R q' p1 ∧ q ⇒$A q'⟩ by blast
  thus ⟨c_game_moves n0 (strategy_from_contrasim R (n0 # play))⟩ using q'_def by (simp add:
n0_def)
qed

lemma basic_contrasim_game_sound:
  assumes
    ⟨player0_winning_strategy f⟩
    ⟨sound_strategy f⟩
  defines
    ⟨R == λ p q . (∃ play ∈ plays_for_strategy f . hd play = AttackerNode p q)⟩
  shows
    ⟨contrasim R⟩
  unfolding contrasim_def
proof (safe)
  fix p q p1 A
  assume A1: ⟨p ⇒$A p1⟩
  assume A2: ⟨R p q⟩ ⟨∀a∈set A. a ≠ τ⟩
  hence ⟨∃ play . play ∈ plays_for_strategy f ∧ hd play = AttackerNode p q⟩ using R_def by
auto
  from this obtain play where play_def: ⟨play ∈ plays_for_strategy f ∧ hd play = AttackerNode
p q⟩ ..
  from assms(1) have ⟨¬player1_wins play⟩ using player0_winning_strategy_def play_def by
auto
  hence ⟨(c_game_defender_node (hd play) ∧ (∃p'. c_game_moves (hd play) p')) ⇒ False⟩
    using player1_wins_def by auto
  hence Def_not_stuck: ⟨c_game_defender_node (hd play) ⇒ (∃p'. c_game_moves (hd play) p')⟩

```

```

⇒ False
  by auto
have ⟨(p ⇒ $A p1) ⇒ (((DefenderNode A p1 q)#play) ∈ plays_for_strategy f)⟩
proof -
  have ⟨∃ n0. n0 = DefenderNode A p1 q⟩ by auto
  from this obtain n0 where n0_def: ⟨n0 = DefenderNode A p1 q⟩ ..
  have play_split: ⟨play = (hd play)#(tl play)⟩
    by (metis hd_Cons_tl play_def strategy_plays_subset no_empty_plays)
  hence inF: ⟨(hd play)#(tl play) ∈ plays_for_strategy f⟩ by (simp add: play_def)
  have pl1: ⟨player1_position (hd play)⟩ by (simp add: play_def player1_position_def)
  have mov0: ⟨c_game_moves (hd play) (DefenderNode A p1 q)⟩
    using ⟨∀ a ∈ set A. a ≠ τ⟩ by (auto simp add: play_def A1)
  have ⟨(DefenderNode A p1 q)#(hd play)#(tl play) ∈ plays_for_strategy f⟩
    using plays_for_strategy.p1move[OF inF pl1 mov0] .
  thus ⟨DefenderNode A p1 q#play ∈ plays_for_strategy f⟩ by (simp add: sym[OF play_split])
qed

hence def_in_f: ⟨(DefenderNode A p1 q)#play ∈ plays_for_strategy f⟩ by (simp add: A1)
hence ⟨¬(player1_wins (DefenderNode A p1 q#play))⟩
  using assms(1) player0_winning_strategy_def by auto
hence ⟨∃ n1. c_game_moves (DefenderNode A p1 q) n1⟩ using player1_wins_def by auto
have move_ex: ⟨c_game_moves (DefenderNode A p1 q) (f (DefenderNode A p1 q # play))⟩
  using assms(2) def_in_f sound_strategy_def by auto
hence in_f: ⟨f ((DefenderNode A p1 q) # play) # (DefenderNode A p1 q) # play ∈ plays_for_strategy
f)⟩
  using plays_for_strategy.p0move[OF def_in_f] by auto
obtain n1 where n1_def: ⟨n1 = f (DefenderNode A p1 q # play)⟩
  and n1_move: ⟨c_game_moves (DefenderNode A p1 q) n1⟩ using move_ex by auto
hence ⟨∃ q1. n1 = (AttackerNode q1 p1)⟩
  using c_game_moves.elims(2)[of ⟨DefenderNode A p1 q⟩ n1] by auto
from this obtain q1 where q1_def: ⟨n1 = (AttackerNode q1 p1)⟩ ..
have ⟨c_game_moves (DefenderNode A p1 q) (AttackerNode q1 p1)⟩ using q1_def move_ex n1_def
by auto
hence q1_succ: ⟨q ⇒ $A q1⟩ using c_game_moves.simps(2) by auto
have def: ⟨c_game_defender_node (DefenderNode A p1 q)⟩ by simp
hence ⟨(AttackerNode q1 p1)#(DefenderNode A p1 q)#play ∈ plays_for_strategy f⟩
  using q1_def n1_def in_f by auto
then obtain R_play where R_play_def: ⟨R_play = (AttackerNode q1 p1)#(DefenderNode A p1 q)#play⟩

  and R_play_in_f: ⟨R_play ∈ plays_for_strategy f⟩ by simp
hence ⟨(hd R_play) = AttackerNode q1 p1⟩ by (simp add: R_play_def)
hence ⟨R q1 p1⟩ unfolding R_def using R_play_in_f by auto
thus ⟨R p q ⇒ p ⇒ $A p1 ⇒ ∃ q1. q ⇒ $A q1 ∧ R q1 p1⟩ using q1_succ by auto
qed

theorem winning_strategy_in_basic_game_iff_contrasim:
  assumes
    ⟨initial = AttackerNode p q⟩
  shows
    ⟨(∃ f . player0_winning_strategy f ∧ sound_strategy f) = (∃ C. contrasim C ∧ C p q)⟩
proof
  assume
    ⟨(∃ f. player0_winning_strategy f ∧ sound_strategy f)⟩
  then obtain f where
    ⟨contrasim (λ p q. ∃ play ∈ plays_for_strategy f. hd play = AttackerNode p q)⟩
    using basic_contrasim_game_sound by blast
  moreover have ⟨(λ p q. ∃ play ∈ plays_for_strategy f. hd play = AttackerNode p q) p q⟩

```

```

      using assms plays_for_strategy.init[of f] by (meson list.sel(1))
    ultimately show  $\langle \exists C. \text{contrasim } C \wedge C \text{ p } q \rangle$  by blast
  next
    assume
       $\langle \exists C. \text{contrasim } C \wedge C \text{ p } q \rangle$ 
    thus  $\langle (\exists f. \text{player0\_winning\_strategy } f \wedge \text{sound\_strategy } f) \rangle$ 
      using basic_contrasim_game_complete[OF _ _ assms]
        strategy_from_contrasim_sound[OF _ _ assms] by blast
  qed

end
end

```