Coupled Similarity and Contrasimilarity,
and How to Compute Them

Benjamin Bisping* Luisa Montanari

April 5, 2024

Abstract

This theory surveys and extends characterizations of *coupled similarity* and *contrasimilarity*, and proves properties relevant for algorithms computing their simulation preorders and equivalences.

Coupled similarity and contrasimilarity are two weak forms of bisimilarity for systems with internal behavior. They have outstanding applications in contexts where internal choices must transparently be distributed in time or space, for example, in process calculi encodings or in action refinements.

Our key contribution is to characterize the coupled simulation and contrasimulation preorders by *reachability games*. We also show how preexisting definitions coincide and that they can be reformulated using *coupled delay simulations*. We moreover verify a polynomial-time coinductive fixed-point algorithm computing the coupled simulation preorder. Through reduction proofs, we establish that deciding coupled similarity is at least as complex as computing weak similarity; and that contrasimilarity checking is at least as hard as trace inclusion checking.

Contents

1 Introduction **3**
- 1.1 This Theory . . . 3
- 1.2 Coupled Similarity vs. Weak Bisimilarity vs. Contrasimilarity in a Nutshell . . . 4
- 1.3 Modal Intuition . . . 4

2 Preliminaries **4**
- 2.1 Labeled Transition Systems . . . 4
- 2.2 Transition Systems with Silent Steps . . . 7
- 2.3 Finite Transition Systems with Silent Steps . . . 16
- 2.4 Simple Games . . . 16

3 Notions of Equivalence **18**
- 3.1 Strong Simulation and Bisimulation . . . 18
- 3.2 Weak Simulation . . . 19
- 3.3 Weak Bisimulation . . . 25
- 3.4 Trace Inclusion . . . 27
- 3.5 Infinite Traces . . . 28
- 3.6 Delay Simulation . . . 29
- 3.7 Coupled Equivalences . . . 30

*TU Berlin, Germany, https://bbisping.de, benjamin.bisping@tu-berlin.de.

4 Contrasimulation **30**
4.1 Definition of Contrasimulation . . . 30
4.2 Intermediate Relation Mimicking Contrasim . . . 32
4.3 Over-Approximating Contrasimulation by a Single-Step Version . . . 34

5 Coupled Simulation **35**
5.1 Van Glabbeeks's Coupled Simulation . . . 35
5.2 Position between Weak Simulation and Weak Bisimulation . . . 35
5.3 Coupled Simulation and Silent Steps . . . 36
5.4 Closure, Preorder and Symmetry Properties . . . 37
5.5 Coinductive Coupled Simulation Preorder . . . 39
5.6 Coupled Simulation Join . . . 40
5.7 Coupled Delay Simulation . . . 41
5.8 Relationship to Contrasimulation and Weak Simulation . . . 43
5.9 τ-Reachability (and Divergence) . . . 43
5.10 On the Connection to Weak Bisimulation . . . 45
5.11 Reduction Semantics Coupled Simulation . . . 48
5.12 Coupled Simulation as Two Simulations . . . 49
5.13 S-coupled Simulation . . . 50

6 Game for Coupled Similarity with Delay Formulation **57**
6.1 The Coupled Simulation Preorder Game Using Delay Steps . . . 57
6.2 Coupled Simulation Implies Winning Strategy . . . 57
6.3 Winning Strategy Induces Coupled Simulation . . . 62

7 Fixed Point Algorithm for Coupled Similarity **64**
7.1 The Algorithm . . . 64
7.2 Correctness . . . 64

8 The Contrasimulation Preorder Word Game **66**
8.1 Contrasimulation Implies Winning Strategy in Word Game (Completeness) . . . 67
8.2 Winning Strategy Implies Contrasimulation in Word Game (Soundness) . . . 70

9 The Contrasimulation Preorder Set Game **73**
9.1 Contrasimulation Implies Winning Strategy in Set Game (Completeness) . . . 73
9.2 Winning Strategy Implies Contrasimulation in Set Game (Soundness) . . . 80

10 Infinitary Hennessy–Milner Logic **85**
10.1 Satisfaction Relation . . . 85
10.2 Distinguishing Formulas . . . 87
10.3 Weak-NOR Hennessy–Milner Logic . . . 87

11 Weak HML and the Contrasimulation Set Game **88**
11.1 Distinguishing Formulas at Winning Attacker Positions . . . 89
11.2 Attacker Wins on Pairs with Distinguishing Formulas . . . 93

12 Reductions and τ-sinks **96**
12.1 τ-Sink Properties . . . 97
12.2 Contrasimulation Equals Weak Simulation on τ-Sink Systems . . . 97
12.3 Contrasimulation Equals Weak Trace Inclusion on τ-Sink Systems . . . 98
12.4 Weak Simulation Invariant under τ-Sink Extension . . . 99
12.5 Trace Inclusion Invariant under τ-Sink Extension . . . 104

References **107**

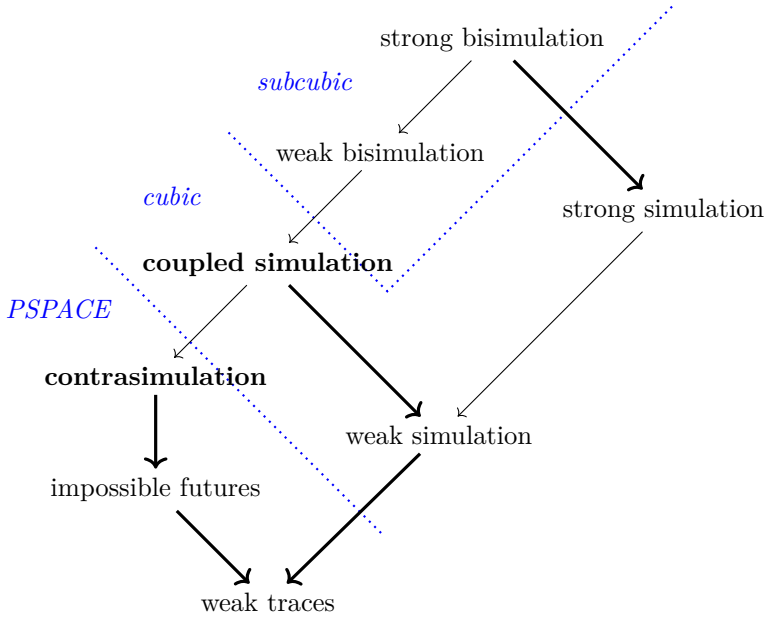

Figure 1: Hierarchy of weak behavioral preorders/equivalences. Arrows denote implication of preordering. Thinner arrows collapse into bi-implication for systems without internal steps. Blue parts indicate a slope of decision problem complexities.

1 Introduction

Coupled similarity and contrasimilarity are among the weakest abstractions of bisimilarity for systems with silent steps presented in van Glabbeek's *linear-time–branching-time spectrum* of behavioral equivalences [6]. In particular, they are *weaker than weak bisimilarity* in that they impose a weaker form of symmetry on the bisimulation link between compared states; coupled similarity implies contrasimilarity. They are weak bisimilarities, however, in the sense that, on systems with no internal behavior, they coincide with strong bisimilarity.

1.1 This Theory

This theory contains the Isabelle/HOL formalization for two related lines of publication, which present the first algorithms to check coupled similarity and contrasimilarity for pairs of states:

- *Computing coupled similarity:* Bisping and Nestmann's TACAS 2019 paper [4] and Bisping's master thesis [1] establish the first decision procedures for coupled similarity checking. This is done through a game-based approach. Also, the work introduces the idea that τ-sinks can be used to reduce from weak simulation preorder to coupled simulation preorder.
- *Game characterization of contrasimilarity:* Bisping and Montanari's EXPRESS/SOS 2021 paper [3] and Montanari's bachelor thesis provide the first game characterization of contrasimilarity. The present Isabelle theory extends this work by also showing a reduction from weak trace preorder to contrasimulation preorder and by linking the game to a modal characterization of contrasimilarity.

Combined, the results establish a slope of complexity between weak bisimilarity, coupled similarity, and contrasimilarity with the equivalence problems becoming harder for coarser equivalences. See Figure 1 for a graphical representation.

1.2 Coupled Similarity vs. Weak Bisimilarity vs. Contrasimilarity in a Nutshell

In coupled simulation semantics, the CCS process $\tau.\,a+\tau.(\tau.\,b+\tau.\,c)$ with gradual internal choice equals $\tau.\,a+\tau.\,b+\tau.\,c$, whcih has just one internal choice point. In weak bisimulation semantics, this equality does not hold as the intermediate choice point $\tau.\,b+\tau.\,c$ of the first process does not match symmetrically to any state of the second process.

The equality also holds in contrasimulation semantics. Contrasimulation moreover blurs the lines between non-determinism of visble behavior and internal non-deterministic choice by considering $a.\,b+a.\,c$ to be indistinguishable from $a.(\tau.\,b+\tau.\,c)$. This equality does not hold under coupled similarity. Therefore, contrasimilarity is strictly coarser than coupled similarity.

For a more detailed exposition about the nuances of coupled similarity and contrasimilarity we refer to our publications [4, 5, 3].

1.3 Modal Intuition

The modal characterization of contrasimilarity at the end of this theory gives a nice intuition for why contrasimilarity is a sensible weakening for bisimilarity. We show that the following Hennessy–Milner logic (with $\langle\varepsilon\rangle$ denoting places of possible internal behavior) characterizes contrasimilarity.

$$\varphi ::= \quad \langle\varepsilon\rangle\langle a\rangle\varphi \quad | \quad \langle\varepsilon\rangle\bigwedge_{i\in I}\neg\varphi_i \qquad (\text{with } a\neq\tau).$$

It is a "nice" abstraction of strong bisimilarity, since it can be obtained from the following complete fragment of Hennessy–Milner logic by inserting places for unobservable behavior in front of each constructor.

$$\varphi ::= \quad \langle a\rangle\varphi \quad | \quad \bigwedge_{i\in I}\neg\varphi_i.$$

This modal formulation is important for a unified algorithmic treatment of weak behavioral equivalences in [2].

2 Preliminaries

2.1 Labeled Transition Systems

```
theory Transition_Systems
  imports Main
begin

locale lts =
fixes
  trans :: <'s ⇒ 'a ⇒ 's ⇒ bool> ("_ ⟼_  _" [70, 70, 70] 80)

begin

abbreviation step_pred :: <'s ⇒ ('a ⇒ bool) ⇒ 's ⇒ bool>
  where
  <step_pred p af q ≡ ∃ a. af a ∧ 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 $\longmapsto^*$ `A`, and all cycles are loops, then there must be nodes which are maximal wrt. $\longmapsto^*$ `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 assms order_trans order_refl finite_has_maximal2 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
```

2.2 Transition Systems with Silent Steps

```
theory Weak_Transition_Systems
  imports Transition_Systems
begin

locale lts_tau = lts trans for
  trans :: <'s ⇒ 'a ⇒ 's ⇒ bool> ("_ ⟼_  _" [70, 70, 70] 80) + 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 <q ⇒^a q1>
proof -
  show ?thesis using assms steps_concat tau_tau by blast
qed


lemma tau_word_concat:
  assumes
    <q ⇒^τ q'>
    <q' ⇒$A q1>
  shows <q ⇒$A q1>
  using assms
proof (cases A)
  case Nil
  hence <q' ⇒^τ q1> 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  <q' ⇒^a q''> and A_step:  <q'' ⇒$A q1 > using assms by auto
  hence <q ⇒^a q''> using tau_step_concat[OF assms(1)] by auto
  then show ?thesis using Cons A_step  <q ⇒^a q''> by auto
qed

lemma strong_weak_transition_system:
  assumes
    <⋀ p q a. p ⟼ a q ⟹ ¬ tau a>
    <¬ tau a>
  shows
    <p ⇒^a p' = p ⟼ a p'>
proof
  assume <p ⇒^a  p'>
  then obtain p0 p1 where <p ⟼* tau p0> <p0 ⟼a p1> <p1 ⟼* tau p'> using assms by blast
  then have <p = p0> <p1 = p'> using assms(1) steps_no_step by blast+
  with <p0 ⟼a p1> show <p ⟼a  p'> by blast
next
  assume <p ⟼a  p'>
  thus <p ⇒^a  p'> using step_weak_step_tau by blast
qed

lemma rev_seq_split :
  assumes <q ⇒$ (xs @ [x])  q1>
  shows <∃q'. q ⇒$xs q' ∧ q' ⇒^x q1>
  using assms
proof (induct xs arbitrary: q)
  case Nil
  hence <q ⇒$ [x]  q1> by auto
  hence x_succ: <q ⇒^x q1> by blast
  have <q ⇒$[] q> by (simp add: steps.refl)
  then show ?case using x_succ by auto
next
  case (Cons a xs)
```

```
  then obtain q' where q'_def: <q ⇒^a q' ∧ q' ⇒$(xs@[x]) q1> by auto
  then obtain q'' where <q' ⇒$ xs  q'' ∧ q'' ⇒^x  q1> using Cons.hyps[of <q'>] by auto
  then show ?case using q'_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.prems(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 ⇒$xs q''> <q'' ⇒^x q'> by auto
  from snoc.prems(2) have <q' ⇒$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 ⇒ 'a list>
  where
    <taufree [] = []>
  | <taufree (a#A) = (if tau a then taufree A else a#(taufree A))>

lemma weak_step_over_tau :
  assumes
    <p ⇒$A p'>
  shows <p ⇒$(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 ⇒^a p0> <p0⇒$as p'> by auto
  then show ?case
  proof (cases <tau a>)
    case True
    hence <p ⇒$as p'> using tau_word_concat <p ⇒^a p0> <p0 ⇒$ as p'> tau_tau by blast
    hence <p ⇒$ (taufree as)  p'> using Cons by auto
    thus <p ⇒$ (taufree (a#as))  p'> using True by auto
  next
    case False
    hence rec: <taufree (a#as) = a#(taufree as)> by auto
    hence <p0 ⇒$ (taufree as)  p'> using <p0⇒$as p'> Cons by auto
    hence <p ⇒$(a#(taufree as)) p'> using  <p ⇒^a p0> by auto
    then show ?thesis using rec by auto
  qed
qed

lemma app_tau_taufree_list :
  assumes
    <∀a ∈ set A. ¬tau a>
    <b = τ>
  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 <∀a∈set xs. ¬ tau a> <¬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 <¬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
    <∀a∈set A. a ≠ τ>
    <p ⇒$ A p'>
    <filter (λa. a ≠ τ) A' = A>
  shows
    <p ⇒$ 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' = τ>)
    case True
    with Cons'.prems have <filter (λa. a ≠ τ) 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 ≠ τ) A' = A''> <∀a∈set A''. a ≠ τ > 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 ≠ τ) 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 = τ>)
    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 ≠ τ) 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 ⇒^τ 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 ⇒^τ 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
    <⋀ r1 r2. r1 ⟼* tau r2 ∧ r2 ⟼* tau r1 ⟹ r1 = r2> — contracted cycles (anti-symmetry)
    <finite {q'. q ⟼* tau q'}>
  shows
    <∃ q' . q ⟼* tau q' ∧ tau_max q'>
  using step_max_deadlock assms by blast

abbreviation stable_state :: <'s ⇒ bool> where
  <stable_state p ≡ ∄ p' . step_pred p tau p'>

lemma stable_tauclosure_only_loop:
  assumes
    <stable_state p>
  shows
    <tau_max p>
  using assms  steps_left by blast

coinductive divergent_state :: <'s ⇒ bool> where
  omega: <divergent_state p' ⟹ tau t ⟹  p ⟼t p'⟹ divergent_state p>

lemma ex_divergent:
  assumes <p ⟼a p> <tau a>
  shows <divergent_state p>
  using assms
proof (coinduct)
  case (divergent_state p)
  then show ?case using assms by auto
qed

lemma ex_not_divergent:
  assumes <∀a q. p ⟼a q ⟶ ¬ tau a> <divergent_state p>
  shows <False> using assms(2)
proof (cases rule:divergent_state.cases)
  case (omega p' t)
  thus ?thesis using assms(1) by auto
qed

lemma perpetual_instability_divergence:
  assumes
    <∀ p' . p ⟼* tau p' ⟶ ¬ stable_state p'>
  shows
    <divergent_state p>
  using assms
proof (coinduct rule: divergent_state.coinduct)
  case (divergent_state p)
```

```
  then obtain t p' where <tau t> <p ⟼t  p'> using steps.refl by blast
  then moreover have <∀p''. p' ⟼* tau  p'' ⟶ ¬ stable_state p''>
     using divergent_state step_weak_step_tau steps_concat by blast
  ultimately show ?case by blast
qed

corollary non_divergence_implies_eventual_stability:
  assumes
    <¬ divergent_state p>
  shows
    <∃ p' . p ⟼* tau p' ∧ stable_state p'>
  using assms perpetual_instability_divergence by blast

end — context lts_tau
```

2.3 Finite Transition Systems with Silent Steps

```
locale lts_tau_finite = lts_tau trans τ for
  trans :: <'s ⇒ 'a ⇒ 's ⇒ bool> and
  τ :: <'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
```

2.4 Simple Games

```
theory Simple_Game
imports
  Main
begin
```

Simple games are games where player0 wins all infinite plays.

```
locale simple_game =
fixes
  game_move :: <'s ⇒ 's ⇒ bool> ("_ ⟼♡ _" [70, 70] 80) and
  player0_position :: <'s ⇒ bool>
begin

abbreviation player1_position :: <'s ⇒ bool>
  where <player1_position s ≡ ¬ player0_position s>
```

— Plays (to be precise: play prefixes) 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>
type_synonym ('s2) posstrategy = <'s2 ⇒ 's2>

definition strategy_from_positional :: <'s posstrategy ⇒ 's strategy> where
  <strategy_from_positional pf = (λ play. pf (hd play))>
```

```
inductive_set plays :: <'s ⇒ 's play set>
  for initial :: 's where
  <[initial] ∈ plays initial> |
  <p#play ∈ plays initial ⟹ p ⟼♡ p' ⟹ p'#p#play ∈ plays initial>

definition play_continuation :: <'s play ⇒ 's play ⇒ bool>
  where <play_continuation p1 p2 ≡ (drop (length p2 - length p1) p2) = p1>

— Plays for a given player 0 strategy
inductive_set plays_for_0strategy :: <'s strategy ⇒ 's ⇒ 's play set>
  for f initial where
  init: <[initial] ∈ plays_for_0strategy f initial> |
  p0move:
    <n0#play ∈ plays_for_0strategy f initial ⟹ player0_position n0 ⟹ n0 ⟼♡ f (n0#play)
    ⟹ (f (n0#play))#n0#play ∈ plays_for_0strategy f initial> |
  p1move:
    <n1#play ∈ plays_for_0strategy f initial ⟹ player1_position n1 ⟹ n1 ⟼♡ n1'
    ⟹ n1'#n1#play ∈ plays_for_0strategy f initial>

lemma strategy0_step:
  assumes
    <n0 # n1 # rest ∈ plays_for_0strategy f initial>
    <player0_position n1>
  shows
    <f (n1 # rest) = n0>
  using assms
  by (induct rule: plays_for_0strategy.cases, auto)

— Plays for a given player 1 strategy
inductive_set plays_for_1strategy :: <'s strategy ⇒ 's ⇒ 's play set>
  for f initial where
  init: <[initial] ∈ plays_for_1strategy f initial> |
  p0move:
    <n0#play ∈ plays_for_1strategy f initial ⟹ player0_position n0 ⟹ n0 ⟼♡ n0'
    ⟹ n0'#n0#play ∈ plays_for_1strategy f initial> |
  p1move:
    <n1#play ∈ plays_for_1strategy f initial ⟹ player1_position n1 ⟹ n1 ⟼♡ f (n1#play)
    ⟹ (f (n1#play))#n1#play ∈ plays_for_1strategy f initial>

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_0strategy :: <'s strategy ⇒ 's ⇒ bool> where
  <sound_0strategy f initial ≡
    ∀ n0 play .
      n0#play ∈ plays_for_0strategy f initial ∧
      player0_position n0 ⟶ n0 ⟼♡ f (n0#play)>

definition sound_1strategy :: <'s strategy ⇒ 's ⇒ bool> where
  <sound_1strategy f initial ≡
    ∀ n1 play .
      n1#play ∈ plays_for_1strategy f initial ∧
      player1_position n1 ⟶ n1 ⟼♡ f (n1#play)>

lemma strategy0_plays_subset:
```

```
  assumes <play ∈ plays_for_0strategy f initial>
  shows <play ∈ plays initial>
  using assms by (induct rule: plays_for_0strategy.induct, auto simp add: plays.intros)
lemma strategy1_plays_subset:
  assumes <play ∈ plays_for_1strategy f initial>
  shows <play ∈ plays initial>
  using assms by (induct rule: plays_for_1strategy.induct, auto simp add: plays.intros)

lemma no_empty_plays:
  assumes <[] ∈ plays initial>
  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_immediately :: <'s play ⇒ bool> where
  <player1_wins_immediately play ≡ player0_position (hd play) ∧ (∄ p' . (hd play) ⟼♡ p')>

definition player0_winning_strategy :: <'s strategy ⇒ 's ⇒ bool> where
  <player0_winning_strategy f initial ≡ (∀ play ∈ plays_for_0strategy f initial.
    ¬ player1_wins_immediately play)>

definition player0_wins :: <'s ⇒ bool> where
  <player0_wins s ≡ (∃ f . player0_winning_strategy f s ∧ sound_0strategy f s)>

lemma stuck_player0_win:
  assumes <player1_position initial> <(∄ p' . initial ⟼♡ p')>
  shows <player0_wins initial>
proof -
  have <⋀pl. pl ∈ plays initial ⟹ pl = [initial]>
  proof -
    fix pl
    assume <pl ∈ plays initial>
    thus <pl = [initial]> using assms(2) by (induct, auto)
  qed
  thus ?thesis using assms(1)
    by (metis list.sel(1) player0_winning_strategy_def player0_wins_def player1_wins_immediately_def
        sound_0strategy_def strategy0_plays_subset)
qed

definition player0_wins_immediately :: <'s play ⇒ bool> where
  <player0_wins_immediately play ≡ player1_position (hd play) ∧ (∄ p' . (hd play) ⟼♡ p')>

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 ⇒ 's ⇒ bool) ⇒ bool>
where
  <simulation R ≡ ∀ p q. R p q ⟶
    (∀ p' a. p ⟼a p' ⟶
      (∃ q'. R p' q' ∧ (q ⟼a q')))>

definition bisimulation ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <bisimulation R ≡ ∀ p q. R p q ⟶
    (∀ p' a. p ⟼a p' ⟶
      (∃ q'. R p' q' ∧ (q ⟼a q'))) ∧
    (∀ q' a. q ⟼a q' ⟶
      (∃ p'. R p' q' ∧ (p ⟼a p')))>

lemma bisim_ruleformat:
  assumes <bisimulation R>
    and <R p q>
  shows
    <p ⟼a p' ⟹ (∃ q'. R p' q' ∧ (q ⟼a q'))>
    <q ⟼a q' ⟹ (∃ p'. R p' q' ∧ (p ⟼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 $\Rightarrow\hat{}$. Therefore we use $\Rightarrow\hat{}\hat{}$ 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 ⟼a  p'>
  hence <(∀x xa. p ⟼x xa ⟶ (∃q'. q ⇒^^ x  q' ∧ greatest_weak_simulation xa q'))>
    by (meson greatest_weak_simulation.simps)
  then obtain q' where <q ⇒^^ a  q' ∧ greatest_weak_simulation p' q'> using ih by blast
  thus <∃q'. greatest_weak_simulation p' q' ∧ q ⇒^a  q'>
    unfolding weak_step_tau2_def by blast
qed

lemma weakly_sim_by_implies_gws:
  assumes <p ⊑ws q>
  shows <greatest_weak_simulation p q>
  using assms
proof coinduct
  case (greatest_weak_simulation p q)
  then obtain R where <weak_simulation R> <R p q>
    unfolding weak_simulation_def by blast
  with weak_sim_ruleformat[OF this] show ?case
    unfolding weak_step_tau2_def
    by metis
qed

lemma gws_eq_weakly_sim_by:
  shows <p ⊑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 ⟼*A  p'>
  <⋀ a . tau a ⟹ A a>
```

```
shows <∃q'. R p' q' ∧ q ⟼*A  q'>
  using assms(3,2,4) proof (induct)
  case (refl p' A)
  hence <R p' q  ∧ q ⟼* 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 q': <R p' q'> <q ⟼* A  q'> by blast
  obtain q'' where q'':
    <R p'' q''> <(¬ tau a ⟶ q' ⇒a  q'') ∧ (tau a ⟶ q' ⟼* tau  q'')>
    using `weak_simulation R` q'(1) step(3) unfolding weak_simulation_def by blast
  have <q' ⟼* A  q''>
    using q''(2) steps_spec[of q'] step(4) step(6) weak_steps[of q' a q''] by blast
  hence <q ⟼* 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 ⟶
      (∀ p' a. p ⇒^a p' ⟶ (∃ q'. R p' q' ∧ q ⇒^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 . ¬ d ⟼a p ∧ ¬ 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.simps)
  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
    <∀ p q p' A . R p q ⟶ p ⇒$A p' ⟶ (∃ q'. R p' q' ∧ q ⇒$A q')> by blast
  hence <∀ p q p' a . R p q ⟶ p ⇒$[a] p' ⟶ (∃ q'. R p' q' ∧ q ⇒$[a] q')> by blast
  thus ?thesis unfolding weak_single_step weak_sim_weak_premise by blast
qed

lemma weak_sim_word:
  <weak_simulation R =
    (∀ p q . R p q ⟶
      (∀ p' A. p ⇒$A p' ⟶ (∃ q'. R p' q' ∧ q ⇒$A q')))>
  using weak_sim_word_impl weak_sim_word_impl_contra by blast
```

3.3 Weak Bisimulation

```
definition weak_bisimulation ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <weak_bisimulation R ≡ ∀ p q. R p q ⟶
    (∀ p' a. p ⟼a p' ⟶ (∃ q'. R p' q'
      ∧ (q ⇒^a q'))) ∧
    (∀ q' a. q ⟼a q' ⟶ (∃ p'. R p' q'
      ∧ ( p ⇒^a p')))>

lemma weak_bisim_ruleformat:
assumes <weak_bisimulation 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'))>
  <q ⟼a q' ⟹ ¬tau a ⟹ (∃ p'. R p' q' ∧ (p ⇒a p'))>
  <q ⟼a q' ⟹  tau a ⟹ (∃ p'. R p' q' ∧ (p ⟼* tau p'))>
  using assms unfolding weak_bisimulation_def by (blast+)

definition tau_weak_bisimulation ::
```

```
  <('s ⇒ 's ⇒  bool) ⇒  bool>
where
  <tau_weak_bisimulation R ≡ ∀ p q. R p q ⟶
    (∀ p' a. p ⟼a p' ⟶
      (∃ q'. R p' q' ∧ (q ⇒a q'))) ∧
    (∀ q' a. q ⟼a q' ⟶
      (∃ p'. R p' q' ∧ (p ⇒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 ⟼a  p'>
  thus <∃q'. R p' q' ∧ (q ⇒^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 ⟼a  q'>
  thus <∃p'. R p' q' ∧ (p ⇒^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>
    <⋀ p q a. p ⟼ a q ⟹ ¬ tau a>
  shows
    <bisimulation R>
  using assms strong_weak_transition_system
  unfolding weak_bisimulation_def bisimulation_def
  by auto
```

3.4 Trace Inclusion

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

abbreviation weakly_trace_included_by :: <'s ⇒ 's ⇒ bool> ("_ ⊑T  _" [60, 60] 65)
  where <weakly_trace_included_by p q ≡ ∃ R . trace_inclusion R ∧ R p q>

lemma weak_trace_inlcusion_greatest:
  shows <trace_inclusion (λ p q . p ⊑T q)>
  unfolding trace_inclusion_def
  by blast
```

3.5 Infinite Traces

```
definition infinite_path ::
  <(nat ⇒ 's) ⇒ bool>
  where <infinite_path pt ≡ ∀n. ∃a. (pt n) ⇒^a (pt (Suc n))>

primrec sim_path :: <('s ⇒ 's ⇒ bool) ⇒ (nat ⇒ 's) ⇒ 's ⇒ (nat ⇒ 's)> where
    <sim_path R iPath q 0 = q> |
    <sim_path R iPath q (Suc n) = (SOME p'. ∃a.
      iPath n ⇒^a (iPath (Suc n))
    ∧ (sim_path R iPath q n ⇒^a p')
    ∧ R (iPath (Suc n)) p')>

lemma path_lifting_construction:
  assumes
    <weak_simulation R>
    <R p q>
    <infinite_path pt>
    <pt 0 = p>
  shows
    <∀n.∃a.
      ((sim_path R pt q) n) ⇒^ a ((sim_path R pt q) (Suc n)) ∧
      R (pt n) ((sim_path R pt q) n)>
proof safe
  fix n show <∃a. sim_path R pt q n ⇒^a (sim_path R pt q (Suc n)) ∧ R (pt n) (sim_path R
pt q n)>
  proof (induct n)
    case 0
    have q_def: <sim_path R pt q 0 = q> by auto
    then obtain a where p_step: <(pt 0) ⇒^a (pt (Suc 0))>
      using <infinite_path pt> unfolding infinite_path_def by blast
    then have <∃q'. q ⇒^a q' ∧ R (pt (Suc 0)) q'>
      using assms(1) <R p q>
      unfolding <pt 0 = p> weak_sim_weak_premise by blast
    hence <∃ q' a. pt 0 ⇒^a (pt (Suc 0))
      ∧ (sim_path R pt q 0 ⇒^a q')
      ∧ R (pt (Suc 0)) q'>
      using p_step
      unfolding <pt 0 = p> q_def by blast
    from someI_ex[OF this] obtain a1 where
        <sim_path R pt q 0 ⇒^a1 (sim_path R pt q (Suc 0))>
      by (auto simp add: <pt 0 = p>)
    then show ?case using assms q_def by auto
  next
    case (Suc n)
    then obtain a1 where ih:
      <sim_path R pt q n ⇒^a1 (sim_path R pt q (Suc n))> <R (pt n) (sim_path R pt q n)> by
blast
    obtain a2 a4 where p_step: <(pt n) ⇒^a2 (pt (Suc n))>  <(pt (Suc n)) ⇒^a4 (pt (Suc (Suc
n)))>
      using <infinite_path pt> unfolding infinite_path_def by blast
    then have <∃q'. (sim_path R pt q n) ⇒^a2 q' ∧ R (pt (Suc n)) q'>
      using assms(1) ih(2)
      unfolding weak_sim_weak_premise by blast
    hence <∃ q' a. pt n ⇒^a (pt (Suc n))
      ∧ (sim_path R pt q n ⇒^a q')
      ∧ R (pt (Suc n)) q'>
```

```
      using p_step by blast
    from someI_ex[OF this] obtain a3 where
      <pt n ⇒^a3 (pt (Suc n))>
      <sim_path R pt q n ⇒^a3 (SOME x. ∃a. pt n ⇒^a (pt (Suc n)) ∧ sim_path R pt q n ⇒^a
x ∧ R (pt (Suc n)) x)>
      <R (pt (Suc n)) (SOME x. ∃a. pt n ⇒^a (pt (Suc n)) ∧ sim_path R pt q n ⇒^a  x ∧ R
(pt (Suc n)) x)>
      by blast
    hence qSucN:
      <pt n ⇒^a3 (pt (Suc n))>
      <sim_path R pt q n ⇒^a3 (sim_path R pt q (Suc n))>
      <R (pt (Suc n)) (sim_path R pt q (Suc n))>
      by auto
    then have <∃q'. (sim_path R pt q (Suc n)) ⇒^a4 q' ∧ R (pt (Suc (Suc n))) q'>
      using assms(1) p_step(2)
      unfolding weak_sim_weak_premise by blast
    hence <∃ q'. ∃a4. (pt (Suc n)) ⇒^a4 (pt (Suc (Suc n)))
      ∧ (sim_path R pt q (Suc n)) ⇒^a4 q'
      ∧  R (pt (Suc (Suc n))) q'>
      using p_step by blast
    from someI_ex[OF this] have
      <∃a. sim_path R pt q (Suc n) ⇒^a (sim_path R pt q (Suc (Suc n)))> by auto
    with qSucN(3) show ?case by blast
  qed
qed

lemma path_lifting:
  assumes
    <weak_simulation R>
    <R p q>
    <infinite_path pt>
    <pt 0 = p>
  shows
    <∃qt. infinite_path qt ∧ qt 0 = q ∧ (∀n. R (pt n) (qt n))>
  using path_lifting_construction[OF assms] sim_path.simps(1)
  unfolding infinite_path_def
  by metis
```

3.6 Delay Simulation

```
definition delay_simulation ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <delay_simulation R ≡ ∀ p q. R p q ⟶
    (∀ p' a. p ⟼a p' ⟶
      (tau a ⟶ R p' q) ∧
      (¬tau a ⟶ (∃ q'. R p' q'∧ (q =▷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
```

3.7 Coupled Equivalences

```
abbreviation coupling ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
  where <coupling R ≡ ∀ p q . R p q ⟶ (∃ q'. q ⟼*tau q' ∧ R q' p)>

lemma coupling_tau_max_symm:
  assumes
    <R p q ⟶ (∃ q'. q ⟼*tau q' ∧ R q' p)>
    <tau_max q>
    <R p q>
  shows
    <R q p>
  using assms steps_no_step_pos[of q tau] by blast

corollary coupling_stability_symm:
  assumes
    <R p q ⟶ (∃ q'. q ⟼*tau q' ∧ R q' p)>
    <stable_state q>
    <R p q>
  shows
    <R q p>
  using coupling_tau_max_symm stable_tauclosure_only_loop assms by metis

end — context lts_tau
end
```

4 Contrasimulation

```
theory Contrasimulation
imports
  Weak_Relations
begin

context lts_tau
begin
```

4.1 Definition of Contrasimulation

```
definition contrasimulation ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <contrasimulation 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 <contrasimulation 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 contrasimulation_def
    using word_steps_ignore_tau_addition word_steps_ignore_tau_removal
    by metis
qed

abbreviation contrasimulated_by :: <'s ⇒ 's ⇒ bool> ("_ ⊑c  _" [60, 60] 65)
```

```
  where <contrasimulated_by p q ≡ ∃ R . contrasimulation R ∧ R p q>

lemma contrasim_preorder_is_contrasim:
  shows <contrasimulation (λ p q . p ⊑c q)>
  unfolding contrasimulation_def
  by metis

lemma contrasim_preorder_is_greatest:
  assumes <contrasimulation R>
  shows <⋀ p q. R p q ⟹ p ⊑c q>
  using assms by auto

lemma contrasim_tau_step:
  <contrasimulation (λ p1 q1 . q1 ⟼* tau p1)>
  unfolding contrasimulation_def
  using steps.simps tau_tau tau_word_concat
  by metis

lemma contrasim_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: <contrasimulation R1> <R1 p pq> and
    R2_def: <contrasimulation R2> <R2 pq q>
  shows
    <R p q> <contrasimulation R>
  using assms(2,3,4,5) unfolding R_def contrasimulation_def by metis+

lemma contrasim_trans:
  assumes
    <p ⊑c pq>
    <pq ⊑c q>
  shows
    <p ⊑c q>
  using assms contrasim_trans_constructive by blast

lemma contrasim_refl:
  shows
    <p ⊑c p>
  using contrasim_tau_step steps.refl by blast

lemma contrasimilarity_equiv:
  defines <contrasimilarity ≡ λ p q. p ⊑c q ∧ q ⊑c p>
  shows <equivp contrasimilarity>
proof -
  have <reflp contrasimilarity>
    using contrasim_refl unfolding contrasimilarity_def reflp_def by blast
  moreover have <symp contrasimilarity>
    unfolding contrasimilarity_def symp_def by blast
  moreover have <transp contrasimilarity>
    using contrasim_trans unfolding contrasimilarity_def transp_def by meson
  ultimately show ?thesis using equivpI by blast
qed

lemma contrasim_implies_trace_incl:
  assumes <contrasimulation R>
```

```
  shows <trace_inclusion R>
by (metis assms contrasim_simpler_def trace_inclusion_def)

lemma contrasim_coupled:
  assumes
    <contrasimulation R>
    <R p q>
  shows
    <∃ q'. q ⟼* tau q' ∧ 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
    <contrasimulation 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
    <contrasimulation R>
    <⋀ p q. R p q ⟹ 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
    <contrasimulation R>
    <⋀ p q a. p ⟼ a q ⟹ ¬ 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>
    <⋀ p q. R p q ⟹ R q p>
  shows
    <contrasimulation R>
  using assms unfolding contrasim_simpler_def weak_sim_word by blast
```

4.2 Intermediate Relation Mimicking Contrasim

```
definition mimicking :: "('s ⇒ 's set ⇒ bool) ⇒ 's ⇒ 's set ⇒ bool" where
<mimicking R p' Q' ≡ ∃p Q A.
  R p Q ∧ p ⇒$A p' ∧
  (∀a ∈ set A. a ≠ τ) ∧
  Q' = (dsuccs_seq_rec (rev A) Q)>

definition set_lifted :: "('s ⇒ 's ⇒ bool) ⇒ 's ⇒ 's set ⇒ bool" where
<set_lifted R p Q ≡ ∃q. R p q ∧ Q = {q}>
```

```
lemma R_is_in_mimicking_of_R :
  assumes <R p Q>
  shows <mimicking R p Q>
  using assms steps.refl lts_tau.weak_step_seq.simps(1)
  unfolding mimicking_def by fastforce

lemma mimicking_of_C_guarantees_tau_succ:
  assumes
    <contrasimulation C>
    <mimicking (set_lifted C) p Q>
    <p ⇒^τ p'>
  shows <∃q'. q' ∈ (weak_tau_succs Q) ∧ mimicking (set_lifted C) q' {p'}>
proof -
  obtain p0 Q0 A q0
      where <(set_lifted C) p0 Q0> <p0 ⇒$A p> <∀a ∈ set A. a ≠ τ> <Q0 = {q0}>
        and Q_def: <Q = (dsuccs_seq_rec (rev A) Q0)>
      using mimicking_def assms set_lifted_def by metis
  hence <C p0 q0> using set_lifted_def by auto
  have <p0 ⇒$(A@[τ]) p'> using <p0 ⇒$A p>  <p ⇒^τ  p'> rev_seq_step_concat by auto
  hence word: <p0 ⇒$A p'>
    by (metis <∀a∈set A. a ≠ τ> app_tau_taufree_list tau_def weak_step_over_tau)
  then obtain q' where <q0 ⇒$A q'> <C q' p'>
    using assms contrasimulation_def[of <C>] <C p0 q0> <∀a ∈ set A. a ≠ τ> by blast
  hence <(set_lifted C) q' {p'}> using set_lifted_def by auto
  hence in_mimicking: <mimicking (set_lifted C) q' {p'}> using R_is_in_mimicking_of_R  by
auto
  have <q' ∈ weak_tau_succs (dsuccs_seq_rec (rev A) Q0)>
    using <Q0 = {q0}> <q0 ⇒$ 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 ∧ mimicking (set_lifted C) q' {p'}> using in_mimicking
by auto
qed

lemma mimicking_of_C_guarantees_action_succ:
 assumes
    <contrasimulation C>
    <mimicking (set_lifted C) p Q>
    <p =▷a p'>
    <a ≠ τ>
  shows <mimicking (set_lifted C) p' (dsuccs a Q)>
proof -
  obtain p0 Q0 A q0
    where <(set_lifted C) p0 Q0> <p0 ⇒$A p> <Q0 = {q0}> <∀a ∈ set A. a ≠ τ >
      and Q_def: <Q = (dsuccs_seq_rec (rev A) Q0)>
    using mimicking_def assms set_lifted_def by metis
  then obtain CS where CS_def: <contrasimulation CS ∧ CS p0 q0>
    using assms set_lifted_def by (metis singleton_inject)
  have notau: <∀a ∈ set (A@[a]). a ≠ τ>
    using <a ≠ τ> <∀a ∈ set A. a ≠ τ > by auto
  have  <p ⇒a  p'> using assms(3,4) steps.refl tau_def by auto
  hence word: <p0 ⇒$(A@[a]) p'>
    using <p0 ⇒$A p>  rev_seq_step_concat
    by (meson steps.step steps_concat)
  then obtain q' where <q0 ⇒$(A@[a]) q' ∧ CS q' p'>
    using CS_def contrasimulation_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 ⇒^τ q'>
    using weak_tau_succs_def[of <dsuccs_seq_rec (rev (A@[a])) {q0}>] by auto
  thus ?thesis
    using word mimicking_def[of <(set_lifted C)>] <(set_lifted C) p0 Q0>
      <Q0 = {q0}> Q_def notau simp_dsuccs_seq_rev by meson
qed
```

4.3 Over-Approximating Contrasimulation by a Single-Step Version

```
definition contrasim_step ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <contrasim_step R ≡ ∀ p q p' a .
    R p q ∧ (p ⇒^a p') ⟶
    (∃ q'. (q ⇒^a q')
         ∧ R q' p')>

lemma contrasim_step_weaker_than_seq:
  assumes
    <contrasimulation R>
  shows
    <contrasim_step R>
  unfolding contrasim_step_def
proof ((rule allI impI)+)
  fix p q p' a
  assume
    <R p q ∧ p ⇒^a  p'>
  hence
    <R p q> <p ⇒^a  p'> by safe
  hence <p ⇒$ [a]  p'> by safe
  then obtain q' where <R q' p'> <q ⇒$ [a]  q'>
    using assms ‘R p q‘ unfolding contrasim_simpler_def by blast
  hence <q ⇒^a  q'> by blast
  thus <∃q'. q ⇒^a  q' ∧ R q' p'> using ‘R q' p'‘ by blast
qed

lemma contrasim_step_seq_coincide_for_sims:
  assumes
    <contrasim_step R>
    <weak_simulation R>
  shows
    <contrasimulation R>
  unfolding contrasimulation_def
proof (clarify)
  fix p q p' A
  assume
    <R p q>
    <p ⇒$ A  p'>
  thus <∃q'. q ⇒$ A  q' ∧ R q' p'>
  proof (induct A arbitrary: p p' q)
    case Nil
    then show ?case using assms(1) unfolding contrasim_step_def
      using tau_tau weak_step_seq.simps(1) by blast
  next
    case (Cons a A)
```

```
    then obtain p1 where p1_def: <p ⇒^a p1> <p1 ⇒$ (A)  p'> by auto
    then obtain q1 where q1_def: <q ⇒^a q1> <R p1 q1>
      using assms(2) `R p q` unfolding weak_sim_weak_premise by blast
    then obtain q' where <q1 ⇒$ (A)  q'> <R q' p'> using p1_def(2) Cons(1) by blast
    then show ?case using q1_def(1) by auto
  qed
qed

end
end
```

5 Coupled Simulation

```
theory Coupled_Simulation
  imports Contrasimulation
begin

context lts_tau
begin
```

5.1 Van Glabbeeks's Coupled Simulation

We mainly use van Glabbeek's coupled simulation from his 2017 CSP paper `cite "glabbeek2017"`. Later on, we will compare it to other definitions of coupled (delay/weak) simulations.

```
definition coupled_simulation ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <coupled_simulation R  ≡ ∀ p q .
    R p q ⟶
      (∀ p' a. p ⟼a p' ⟶
        (∃ q'. R p' q' ∧ q ⇒^a q')) ∧
      (∃ q'. q ⟼*tau q' ∧ R q' p)>

abbreviation coupled_simulated_by :: <'s ⇒ 's ⇒ bool> ("_ ⊑cs  _" [60, 60] 65)
  where <coupled_simulated_by p q ≡ ∃ R . coupled_simulation R ∧ R p q>

abbreviation coupled_similar :: <'s ⇒ 's ⇒ bool> ("_ ≡cs  _" [60, 60] 65)
  where <coupled_similar p q ≡ p ⊑cs q ∧ q ⊑cs p>
```

We call ⊑cs "coupled simulation preorder" and ≡cs coupled similarity.

5.2 Position between Weak Simulation and Weak Bisimulation

Coupled simulations are special weak simulations, and symmetric weak bisimulations also are coupled simulations.

```
lemma coupled_simulation_weak_simulation:
  <coupled_simulation R =
    (weak_simulation R ∧ (∀ p q . R p q ⟶ (∃ q'. q ⟼*tau q' ∧ R q' p)))>
  unfolding coupled_simulation_def weak_simulation_def by blast

corollary coupled_simulation_implies_weak_simulation:
  assumes <coupled_simulation R>
  shows <weak_simulation R>
  using assms unfolding coupled_simulation_weak_simulation ..
```

```
corollary coupledsim_enabled_subs:
  assumes
    <p ⊑cs q>
    <weak_enabled p a>
    <¬ tau a>
  shows <weak_enabled q a>
  using assms weak_sim_enabled_subs coupled_simulation_implies_weak_simulation by blast

lemma coupled_simulation_implies_coupling:
  assumes
    <coupled_simulation R>
    <R p q>
  shows
    <∃ q'. q ⟼*tau q' ∧ R q' p>
  using assms unfolding coupled_simulation_weak_simulation by blast

lemma weak_bisim_implies_coupled_sim_gla17:
  assumes
    wbisim:   <weak_bisimulation R> and
    symmetry: <⋀ p q . R p q ⟹ R q p>
    — symmetry is needed here, which is alright because bisimilarity is symmetric.
  shows <coupled_simulation R>
unfolding coupled_simulation_def proof safe
  fix p q p' a
  assume <R p q> <p ⟼a  p'>
  thus <∃q'. R p' q' ∧ (q ⇒^a  q')>
    using wbisim unfolding weak_bisimulation_def by simp
next
  fix p q
  assume <R p q>
  thus <∃q'. q ⟼* tau  q' ∧ R q' p>
    using symmetry steps.refl[of q tau] by auto
qed
```

5.3 Coupled Simulation and Silent Steps

Coupled simulation shares important patterns with weak simulation when it comes to the treatment of silent steps.

```
lemma coupledsim_step_gla17:
  <coupled_simulation (λ p1 q1 . q1 ⟼* tau p1)>
  unfolding coupled_simulation_def
  using lts.steps.simps by metis

corollary coupledsim_step:
  assumes
    <p ⟼* tau  q>
  shows
    <q ⊑cs p>
  using assms coupledsim_step_gla17 by auto
```

A direct implication of this is that states on a tau loop are coupled similar.

```
corollary strongly_tau_connected_coupled_similar:
  assumes
    <p ⟼* tau  q>
    <q ⟼* tau  p>
  shows <p ≡cs q>
  using assms coupledsim_step by auto
```

```
lemma silent_steps_retain_coupled_simulation:
assumes
  <coupled_simulation R>
  <R p q>
  <p ⟼* A  p'>
  <A = tau>
shows <∃ q' . q ⟼* A q' ∧ R p' q'>
  using assms(1,3,2,4) steps_retain_weak_sim
  unfolding coupled_simulation_weak_simulation by blast

lemma coupled_simulation_weak_premise:
  <coupled_simulation R =
   (∀ p q . R p q ⟶
      (∀ p' a. p ⇒^a p' ⟶
        (∃ q'. R p' q' ∧ q ⇒^a q')) ∧
      (∃ q'. q ⟼*tau q' ∧ R q' p))>
  unfolding coupled_simulation_weak_simulation weak_sim_weak_premise by blast
```

5.4 Closure, Preorder and Symmetry Properties

The coupled simulation preorder $\sqsubseteq_{\mathtt{cs}}$ *is* a preoder and symmetric at the stable states.

```
lemma coupledsim_union:
  assumes
    <coupled_simulation R1>
    <coupled_simulation R2>
  shows
    <coupled_simulation (λ p q . R1 p q ∨ R2 p q)>
  using assms unfolding coupled_simulation_def by (blast)

lemma coupledsim_refl:
  <p ⊑cs p>
  using coupledsim_step steps.refl by auto

lemma coupledsim_trans:
  assumes
    <p ⊑cs pq>
    <pq ⊑cs q>
  shows
    <p ⊑cs q>
proof -
  obtain R1 where R1_def: <coupled_simulation R1> <R1 p pq>
    using assms(1) by blast
  obtain R2 where R2_def: <coupled_simulation R2> <R2 pq q>
    using assms(2) by blast
  define R where R_def: <R ≡ λ p q . ∃ pq . (R1 p pq ∧ R2 pq q) ∨ (R2 p pq ∧ R1 pq q)>
  have <weak_simulation R> <R p q>
    using weak_sim_trans_constructive
      R1_def(2) R2_def(2)
      coupled_simulation_implies_weak_simulation[OF R1_def(1)]
      coupled_simulation_implies_weak_simulation[OF R2_def(1)]
    unfolding R_def by auto
  moreover have <(∀ p q . R p q ⟶ (∃ q'. q ⟼*tau q' ∧ R q' p))>
    unfolding R_def
  proof safe
    fix p q pq
    assume r1r2: <R1 p pq> <R2 pq q>
```

```
    then obtain pq' where <R1 pq' p> <pq ⟼* tau  pq'>
      using r1r2 R1_def(1) unfolding coupled_simulation_weak_premise by blast
    then moreover obtain q' where <R2 pq' q'> <q ⟼* tau q'>
      using r1r2 R2_def(1) weak_step_tau_tau[OF `pq ⟼* tau  pq'`] tau_tau
      unfolding coupled_simulation_weak_premise by blast
    then moreover obtain q'' where <R2 q'' pq'> <q' ⟼* tau  q''>
      using R2_def(1) unfolding coupled_simulation_weak_premise by blast
    ultimately show <∃q'. q ⟼* tau  q' ∧ (∃pq. R1 q' pq ∧ R2 pq p ∨ R2 q' pq ∧ R1 pq p)>
      using steps_concat by blast
  next — analogous with R2 and R1 swapped
    fix p q pq
    assume r2r1: <R2 p pq> <R1 pq q>
    then obtain pq' where <R2 pq' p> <pq ⟼* tau  pq'>
      using r2r1 R2_def(1) unfolding coupled_simulation_weak_premise by blast
    then moreover obtain q' where <R1 pq' q'> <q ⟼* tau q'>
      using r2r1 R1_def(1) weak_step_tau_tau[OF `pq ⟼* tau  pq'`] tau_tau
      unfolding coupled_simulation_weak_premise by blast
    then moreover obtain q'' where <R1 q'' pq'> <q' ⟼* tau  q''>
      using R1_def(1) unfolding coupled_simulation_weak_premise by blast
    ultimately show <∃q'. q ⟼* tau  q' ∧ (∃pq. R1 q' pq ∧ R2 pq p ∨ R2 q' pq ∧ R1 pq p)>
      using steps_concat by blast
  qed
  ultimately have <R p q> <coupled_simulation R>
    using coupled_simulation_weak_simulation by auto
  thus ?thesis by blast
qed

interpretation preorder <λ p q. p ⊑cs q> <λ p q. p ⊑cs q ∧ ¬(q ⊑cs p)>
  by (standard, blast, fact coupledsim_refl, fact coupledsim_trans)

lemma coupled_similarity_equivalence:
  <equivp (λ p q. p ≡cs q)>
proof (rule equivpI)
  show <reflp coupled_similar>
    unfolding reflp_def by blast
next
  show <symp coupled_similar>
    unfolding symp_def by blast
next
  show <transp coupled_similar>
    unfolding transp_def using coupledsim_trans by meson
qed

lemma coupledsim_tau_max_eq:
  assumes
    <p ⊑cs q>
    <tau_max q>
  shows  <p ≡cs q>
  using assms using coupled_simulation_weak_simulation coupling_tau_max_symm by metis

corollary coupledsim_stable_eq:
  assumes
    <p ⊑cs q>
    <stable_state q>
  shows  <p ≡cs q>
  using assms using coupled_simulation_weak_simulation coupling_stability_symm by metis
```

5.5 Coinductive Coupled Simulation Preorder

⊑cs can also be characterized coinductively. ⊑cs is the greatest coupled simulation.

```
coinductive greatest_coupled_simulation :: <'s ⇒ 's ⇒ bool>
  where gcs:
    <⟦⋀ a p' . p ⟼a p' ⟹ ∃q'. q ⇒^^ a q' ∧ greatest_coupled_simulation p' q';
      ∃ q' . q ⟼* tau q' ∧ greatest_coupled_simulation q' p⟧
  ⟹ greatest_coupled_simulation p q>

lemma gcs_implies_gws:
  assumes <greatest_coupled_simulation p q>
  shows <greatest_weak_simulation p q>
  using assms by (metis greatest_coupled_simulation.cases greatest_weak_simulation.coinduct)

lemma gcs_is_coupled_simulation:
  shows <coupled_simulation greatest_coupled_simulation>
  unfolding coupled_simulation_def
proof safe
  — identical to ws
  fix p q p' a
  assume ih:
    <greatest_coupled_simulation p q>
    <p ⟼a  p'>
  hence <(∀x xa. p ⟼x xa ⟶ (∃q'. q ⇒^^ x  q' ∧ greatest_coupled_simulation xa q'))>
    by (meson greatest_coupled_simulation.simps)
  then obtain q' where <q ⇒^^ a  q' ∧ greatest_coupled_simulation p' q'> using ih by blast
  thus <∃q'. greatest_coupled_simulation p' q' ∧ q ⇒^a  q'>
    unfolding weak_step_tau2_def by blast
next
  fix p q
  assume
    <greatest_coupled_simulation p q>
  thus <∃q'. q ⟼* tau  q' ∧ greatest_coupled_simulation q' p>
    by (meson greatest_coupled_simulation.simps)
qed

lemma coupled_similarity_implies_gcs:
  assumes <p ⊑cs q>
  shows <greatest_coupled_simulation p q>
  using assms
proof (coinduct)
  case (greatest_coupled_simulation p1 q1)
  then obtain R where <coupled_simulation R> <R p1 q1>
    <weak_simulation R> using coupled_simulation_implies_weak_simulation by blast
  then have <(∀x xa. p1 ⟼x  xa ⟶
          (∃q'. q1 ⇒^x  q' ∧ (xa ⊑cs  q' ∨ greatest_coupled_simulation xa q'))) ∧
        (∃q'. q1 ⟼* tau  q' ∧
          (q' ⊑cs  p1 ∨ greatest_coupled_simulation q' p1))>
    unfolding weak_step_tau2_def
    using coupled_simulation_implies_coupling
       weak_sim_ruleformat[OF <weak_simulation R>]
    by (metis (no_types, lifting))
  thus ?case by simp
qed

lemma gcs_eq_coupled_sim_by:
  shows <p ⊑cs q = greatest_coupled_simulation p q>
```

```
  using coupled_similarity_implies_gcs gcs_is_coupled_simulation by blast

lemma coupled_sim_by_is_coupled_sim:
  shows
    <coupled_simulation (λ p q . p ⊑cs q)>
  unfolding gcs_eq_coupled_sim_by using gcs_is_coupled_simulation .

lemma coupledsim_unfold:
  shows <p ⊑cs q =
    ((∀a p'. p ⟼a  p' ⟶ (∃q'. q ⇒^a  q' ∧ p' ⊑cs q')) ∧
       (∃q'. q ⟼* tau  q' ∧ q' ⊑cs p))>
  unfolding gcs_eq_coupled_sim_by weak_step_tau2_def[symmetric]
  by (metis lts_tau.greatest_coupled_simulation.simps)
```

5.6 Coupled Simulation Join

The following lemmas reproduce Proposition 3 from `cite glabbeek2017` that internal choice acts as a least upper bound within the semi-lattice of CSP terms related by ⊑cs taking ≡cs as equality.

```
lemma coupledsim_choice_1:
  assumes
    <p ⊑cs q>
    <⋀ pq a . pqc ⟼a pq ⟷ (a = τ ∧ (pq = p ∨ pq = q))>
  shows
    <pqc ⊑cs q>
    <q ⊑cs pqc>
proof -
  define R1 where <R1 ≡ (λp1 q1. q1 ⟼* tau  p1)>
  have <R1 q pqc>
    using assms(2) steps_one_step R1_def by simp
  moreover have <coupled_simulation R1>
    unfolding R1_def using coupledsim_step_gla17 .
  ultimately show q_pqc: <q ⊑cs pqc> by blast
— next case
  define R where <R ≡ λ p0 q0 . p0 = q ∧ q0 = pqc ∨ p0 = pqc ∧ q0 = q ∨ p0 = p ∧ q0 = q>
  hence <R pqc q> by blast
  thus <pqc ⊑cs  q>
    unfolding gcs_eq_coupled_sim_by
  proof (coinduct)
    case (greatest_coupled_simulation p1 q1)
    then show ?case
      unfolding weak_step_tau2_def R_def
    proof safe
      assume <q1 = pqc> <p1 = q>
      thus <∃pa qa.
        q = pa ∧ pqc = qa ∧
        (∀x xa. pa ⟼x  xa ⟶
          (∃q'. qa ⇒^x  q' ∧ ((xa = q ∧ q' = pqc ∨ xa = pqc ∧ q' = q ∨ xa = p ∧ q' = q)
            ∨ greatest_coupled_simulation xa q'))) ∧
        (∃q'. qa ⟼* tau  q' ∧
            ((q' = q ∧ pa = pqc ∨ q' = pqc ∧ pa = q ∨ q' = p ∧ pa = q)
            ∨ greatest_coupled_simulation q' pa))>
        using ‘q ⊑cs pqc‘ step_tau_refl weak_sim_ruleformat tau_def
          coupled_simulation_implies_weak_simulation[OF gcs_is_coupled_simulation]
        unfolding gcs_eq_coupled_sim_by by fastforce
    next
      assume <q1 = q> <p1 = pqc>
      thus <∃pa qa.
```

```
        pqc = pa ∧ q = qa ∧
        (∀x xa. pa ⟼x  xa ⟶
          (∃q'. qa ⇒^x  q' ∧ ((xa = q ∧ q' = pqc ∨ xa = pqc ∧ q' = q ∨ xa = p ∧ q' = q)
            ∨ greatest_coupled_simulation xa q'))) ∧
        (∃q'. qa ⟼* tau  q' ∧
          ((q' = q ∧ pa = pqc ∨ q' = pqc ∧ pa = q ∨ q' = p ∧ pa = q)
            ∨ greatest_coupled_simulation q' pa))>
        using R1_def <coupled_simulation R1> assms(2)
          coupled_similarity_implies_gcs step_tau_refl by fastforce
    next
      assume <q1 = q> <p1 = p>
      thus <∃pa qa.
       p = pa ∧ q = qa ∧
       (∀x xa. pa ⟼x  xa ⟶ (∃q'. qa ⇒^x  q' ∧ ((xa = q ∧ q' = pqc ∨ xa = pqc ∧ q' =
q ∨ xa = p ∧ q' = q) ∨ greatest_coupled_simulation xa q'))) ∧
       (∃q'. qa ⟼* tau  q' ∧ ((q' = q ∧ pa = pqc ∨ q' = pqc ∧ pa = q ∨ q' = p ∧ pa = q)
∨ greatest_coupled_simulation q' pa))>
        using ‘p ⊑cs q‘ weak_sim_ruleformat
          coupled_simulation_implies_weak_simulation[OF gcs_is_coupled_simulation]
          coupled_simulation_implies_coupling[OF gcs_is_coupled_simulation]
        unfolding gcs_eq_coupled_sim_by
        by (auto, metis+)
    qed
  qed
qed

lemma coupledsim_choice_2:
  assumes
    <pqc ⊑cs q>
    <⋀ pq a . pqc ⟼a pq ⟷ (a = τ ∧ (pq = p ∨ pq = q))>
  shows
    <p ⊑cs q>
proof -
  have <pqc ⟼τ p> using assms(2) by blast
  then obtain q' where <q ⟼* tau q'> <p ⊑cs q'>
    using assms(1) tau_tau unfolding coupled_simulation_def by blast
  then moreover have <q' ⊑cs q> using coupledsim_step_gla17 by blast
  ultimately show ?thesis using coupledsim_trans tau_tau by blast
qed

lemma coupledsim_choice_join:
  assumes
    <⋀ pq a . pqc ⟼a pq ⟷ (a = τ ∧ (pq = p ∨ pq = q))>
  shows
    <p ⊑cs q ⟷ pqc ≡cs q>
  using  coupledsim_choice_1[OF _ assms] coupledsim_choice_2[OF _ assms] by blast
```

5.7 Coupled Delay Simulation

⊑cs can also be characterized in terms of coupled delay simulations, which are conceptionally simpler than van Glabbeek's coupled simulation definition.

In the greatest coupled simulation, τ-challenges can be answered by stuttering.

```
lemma coupledsim_tau_challenge_trivial:
  assumes
    <p ⊑cs q>
    <p ⟼* tau p'>
```

```
  shows
    <p' ⊑cs q>
  using assms coupledsim_trans coupledsim_step by blast

lemma coupled_similarity_s_delay_simulation:
  <delay_simulation (λ p q. p ⊑cs q)>
  unfolding delay_simulation_def
proof safe
  fix p q R p' a
  assume assms:
    <coupled_simulation R>
    <R p q>
    <p ⟼a p'>
  {
    assume <tau a>
    then show <p' ⊑cs  q>
      using assms coupledsim_tau_challenge_trivial steps_one_step by blast
  } {
    show <∃q'. p' ⊑cs  q' ∧ q =▷a  q'>
    proof -
      obtain q''' where q'''_spec: <q ⇒^a q'''> <p' ⊑cs q'''>
        using assms coupled_simulation_implies_weak_simulation weak_sim_ruleformat by metis
      show ?thesis
      proof (cases <tau a>)
        case True
        then have <q ⟼* tau q'''> using q'''_spec by blast
        thus ?thesis using q'''_spec(2) True assms(1) steps.refl by blast
      next
        case False
        then obtain q' q'' where q'q''_spec:
            <q ⟼* tau q'> <q' ⟼a q''> <q'' ⟼* tau q'''>
          using q'''_spec by blast
        hence <q''' ⊑cs q''> using coupledsim_step by blast
        hence <p' ⊑cs q''> using q'''_spec(2) coupledsim_trans by blast
        thus ?thesis using q'q''_spec(1,2) False by blast
      qed
    qed
  }
qed

definition coupled_delay_simulation ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
  where
  <coupled_delay_simulation R  ≡
    delay_simulation R ∧ coupling R>

lemma coupled_sim_by_eq_coupled_delay_simulation:
  <(p ⊑cs q) = (∃R. R p q ∧ coupled_delay_simulation R)>
  unfolding coupled_delay_simulation_def
proof
  assume <p ⊑cs q>
  define R where <R ≡ coupled_simulated_by>
  hence <R p q ∧ delay_simulation R ∧ coupling R>
    using coupled_similarity_s_delay_simulation coupled_sim_by_is_coupled_sim
        coupled_simulation_implies_coupling <p ⊑cs q> by blast
  thus <∃R. R p q ∧ delay_simulation R ∧ coupling R> by blast
next
```

```
  assume <∃R. R p q ∧ delay_simulation R ∧ coupling R>
  then obtain R where <R p q> <delay_simulation R> <coupling R> by blast
  hence <coupled_simulation R>
    using delay_simulation_implies_weak_simulation coupled_simulation_weak_simulation by blast
  thus <p ⊑cs q> using <R p q> by blast
qed
```

5.8 Relationship to Contrasimulation and Weak Simulation

Coupled simulation is precisely the intersection of contrasimulation and weak simulation.

```
lemma weak_sim_and_contrasim_implies_coupled_sim:
  assumes
    <contrasimulation R>
    <weak_simulation R>
  shows
    <coupled_simulation R>
  unfolding coupled_simulation_weak_simulation
  using contrasim_coupled assms by blast

lemma coupledsim_implies_contrasim:
  assumes
    <coupled_simulation R>
  shows
    <contrasimulation R>
proof -
  have <contrasim_step R>
  unfolding contrasim_step_def
  proof (rule allI impI)+
    fix p q p' a
    assume
      <R p q ∧ p ⇒^a  p'>
    then obtain q' where q'_def: <R p' q'> <q ⇒^a  q'>
      using assms unfolding coupled_simulation_weak_premise by blast
    then obtain q'' where q''_def: <R q'' p'> <q' ⟼* tau  q''>
      using assms unfolding coupled_simulation_weak_premise by blast
    then have <q ⇒^a  q''> using q'_def(2) steps_concat by blast
    thus <∃q'. q ⇒^a  q' ∧ R q' p'>
      using q''_def(1) by blast
  qed
  thus <contrasimulation R> using contrasim_step_seq_coincide_for_sims
      coupled_simulation_implies_weak_simulation[OF assms] by blast
qed

lemma coupled_simulation_iff_weak_sim_and_contrasim:
  shows <coupled_simulation R ⟷ contrasimulation R ∧ weak_simulation R>
  using weak_sim_and_contrasim_implies_coupled_sim
    coupledsim_implies_contrasim coupled_simulation_weak_simulation by blast
```

5.9 τ-Reachability (and Divergence)

Coupled similarity comes close to (weak) bisimilarity in two respects:

- If there are no τ transitions, coupled similarity coincides with bisimilarity.
- If there are only finite τ reachable portions, then coupled similarity contains a bisimilarity on the τ-maximal states. (For this, τ-cycles have to be ruled out, which, as we show, is no problem because their removal is transparent to coupled similarity.)

```
lemma taufree_coupledsim_symm:
  assumes
    <⋀ p1 a p2 . (p1 ⟼a p2 ⟹ ¬ tau a)>
    <coupled_simulation R>
    <R p q>
  shows <R q p>
  using assms(1,3) coupledsim_implies_contrasim[OF assms(2)] contrasim_taufree_symm
  by blast

lemma taufree_coupledsim_weak_bisim:
  assumes
    <⋀ p1 a p2 . (p1 ⟼a p2 ⟹ ¬ tau a)>
    <coupled_simulation R>
  shows <weak_bisimulation R>
  using assms contrasim_taufree_symm symm_contrasim_is_weak_bisim coupledsim_implies_contrasim[OF
assms(2)]
  by blast

lemma coupledsim_stable_state_symm:
  assumes
    <coupled_simulation R>
    <R p q>
    <stable_state q>
  shows
    <R q p>
  using assms steps_left unfolding coupled_simulation_def by metis
```

In finite systems, coupling is guaranteed to happen through τ-maximal states.

```
lemma coupledsim_max_coupled:
  assumes
    <p ⊑cs q>
    <⋀ r1 r2 . r1 ⟼* tau r2 ∧ r2 ⟼* tau r1 ⟹ r1 = r2> — contracted tau cycles
    <⋀ r. finite {r'. r ⟼* tau r'}>
  shows
    <∃ q' . q ⟼* tau q' ∧ q' ⊑cs p ∧ tau_max q'>
proof -
  obtain q1 where q1_spec: <q ⟼* tau q1> <q1 ⊑cs p>
    using assms(1) coupled_simulation_implies_coupling coupledsim_implies_contrasim by blast
  then obtain q' where <q1 ⟼* tau q'> <(∀q''. q' ⟼* tau q'' ⟶ q' = q'')>
    using tau_max_deadlock assms(2,3) by blast
  then moreover have <q' ⊑cs p> <q ⟼* tau q'>
    using q1_spec coupledsim_trans coupledsim_step steps_concat[of q1 tau q' q]
    by blast+
  ultimately show ?thesis by blast
qed
```

In the greatest coupled simulation, a-challenges can be answered by a weak move without trailing τ-steps. (This property is what bridges the gap between weak and delay simulation for coupled simulation.)

```
lemma coupledsim_step_challenge_short_answer:
  assumes
    <p ⊑cs q>
    <p ⟼a p'>
    <¬ tau a>
  shows
    <∃ q' q1. p' ⊑cs q' ∧ q ⟼* tau q1 ∧ q1 ⟼a q'>
  using assms
```

```
  unfolding coupled_sim_by_eq_coupled_delay_simulation
    coupled_delay_simulation_def delay_simulation_def by blast
```

If two states share the same outgoing edges with except for one τ-loop, then they cannot be distinguished by coupled similarity.

```
lemma coupledsim_tau_loop_ignorance:
  assumes
    <⋀ a p'. p ⟼a p' ∨ p' = pp ∧ a = τ ⟷ pp ⟼a p'>
  shows
    <pp ≡cs p>
proof -
  define R where <R ≡ λ p1 q1. p1 = q1 ∨ p1 = pp ∧ q1 = p ∨ p1 = p ∧ q1 = pp>
  have <coupled_simulation R>
    unfolding coupled_simulation_def R_def
  proof (safe)
    fix pa q p' a
    assume
      <q ⟼a  p'>
    thus <∃q'. (p' = q' ∨ p' = pp ∧ q' = p ∨ p' = p ∧ q' = pp) ∧ q ⇒^a  q'>
      using assms step_weak_step_tau by auto
  next
    fix pa q
    show <∃q'. q ⟼* tau  q' ∧ (q' = q ∨ q' = pp ∧ q = p ∨ q' = p ∧ q = pp)>
      using steps.refl by blast
  next
    fix pa q p' a
    assume
      <pp ⟼a  p'>
    thus <∃q'. (p' = q' ∨ p' = pp ∧ q' = p ∨ p' = p ∧ q' = pp) ∧ p ⇒^a  q'>
      using assms by (metis lts.steps.simps tau_def)
  next
    fix pa q
    show <∃q'. p ⟼* tau  q' ∧ (q' = pp ∨ q' = pp ∧ pp = p ∨ q' = p ∧ pp = pp)>
      using steps.refl[of p tau] by blast
  next
    fix pa q p' a
    assume
      <p ⟼a  p'>
    thus <∃q'. (p' = q' ∨ p' = pp ∧ q' = p ∨ p' = p ∧ q' = pp) ∧ pp ⇒^a  q'>
      using assms step_weak_step_tau by fastforce
  next
    fix pa q
    show <∃q'. pp ⟼* tau  q' ∧ (q' = p ∨ q' = pp ∧ p = p ∨ q' = p ∧ p = pp)>
      using steps.refl[of pp tau] by blast
  qed
  moreover have <R p pp> <R pp p> unfolding R_def by auto
  ultimately show ?thesis by blast
qed
```

5.10 On the Connection to Weak Bisimulation

When one only considers steps leading to τ-maximal states in a system without infinite τ-reachable regions (e.g. a finite system), then $\equiv$cs on these steps is a bisimulation.

This lemma yields a neat argument why one can use a signature refinement algorithm to pre-select the tuples which come into question for further checking of coupled simulation by contraposition.

```
lemma coupledsim_eventual_symmetry:
  assumes
    contracted_cycles: <⋀ r1 r2 . r1 ⟼* tau r2 ∧ r2 ⟼* tau r1 ⟹ r1 = r2> and
    finite_taus: <⋀ r. finite {r'. r ⟼* tau r'}> and
    cs: <p ⊑cs q> and
    step: <p ⇒^a p'> and
    tau_max_p': <tau_max p'>
  shows
    <∃ q'. tau_max q' ∧ q ⇒^a q' ∧ p' ≡cs q'>
proof-
  obtain q' where q'_spec: <q ⇒^a q'> <p' ⊑cs q'>
    using cs step unfolding coupled_simulation_weak_premise by blast
  then obtain q'' where q''_spec: <q' ⟼* tau q''> <q'' ⊑cs p'>
    using cs unfolding coupled_simulation_weak_simulation by blast
  then obtain q_max where q_max_spec: <q'' ⟼* tau q_max> <tau_max q_max>
    using tau_max_deadlock contracted_cycles finite_taus by force
  hence <q_max ⊑cs p'> using q''_spec coupledsim_tau_challenge_trivial by blast
  hence <q_max ≡cs p'> using tau_max_p' coupledsim_tau_max_eq by blast
  moreover have <q ⇒^a q_max> using q_max_spec q'_spec q''_spec steps_concat by blast
  ultimately show ?thesis using q_max_spec(2) by blast
qed
```

Even without the assumption that the left-hand-side step `p ⇒^a p'` ends in a τ-maximal state, a situation resembling bismulation can be set up – with the drawback that it only refers to a τ-maximal sibling of `p'`.

```
lemma coupledsim_eventuality_2:
  assumes
    contracted_cycles: <⋀ r1 r2 . r1 ⟼* tau r2 ∧ r2 ⟼* tau r1 ⟹ r1 = r2> and
    finite_taus: <⋀ r. finite {r'. r ⟼* tau r'}> and
    cbisim: <p ≡cs q> and
    step: <p ⇒^a p'>
  shows
    <∃ p'' q'. tau_max p'' ∧ tau_max q' ∧ p ⇒^a p'' ∧ q ⇒^a q' ∧ p'' ≡cs q'>
proof-
  obtain q' where q'_spec: <q ⇒^a q'>
    using cbisim step unfolding coupled_simulation_weak_premise by blast
  then obtain q_max where q_max_spec: <q' ⟼* tau q_max> <tau_max q_max>
    using tau_max_deadlock contracted_cycles finite_taus by force
  hence <q ⇒^a q_max> using q'_spec steps_concat by blast
  then obtain p'' where p''_spec: <p ⇒^a p''> <q_max ⊑cs p''>
    using cbisim unfolding coupled_simulation_weak_premise by blast
  then obtain p''' where p'''_spec: <p'' ⟼* tau p'''> <p''' ⊑cs q_max>
    using cbisim unfolding coupled_simulation_weak_simulation  by blast
  then obtain p_max where p_max_spec: <p''' ⟼* tau p_max> <tau_max p_max>
    using tau_max_deadlock contracted_cycles finite_taus by force
  hence <p_max ⊑cs p'''> using coupledsim_step by blast
  hence <p_max ⊑cs q_max> using p'''_spec coupledsim_trans by blast
  hence <q_max ≡cs p_max> using coupledsim_tau_max_eq q_max_spec by blast
  moreover have  <p ⇒^a p_max>
    using  p''_spec(1) steps_concat[OF p_max_spec(1) p'''_spec(1)] steps_concat by blast
  ultimately show ?thesis using p_max_spec(2) q_max_spec(2) ‘q ⇒^a q_max‘ by blast
qed

lemma coupledsim_eq_reducible_1:
  assumes
    contracted_cycles: <⋀ r1 r2 . r1 ⟼* tau r2 ∧ r2 ⟼* tau r1 ⟹ r1 = r2> and
    finite_taus: <⋀ r. finite {r'. r ⟼* tau r'}> and
```

```
    tau_shortcuts:
      <⋀r a r'. r ⟼* tau r' ⟹ ∃r''. tau_max r'' ∧ r ⟼τ r'' ∧ r' ⊑cs r''>  and
    sim_vis_p:
      <⋀p' a. ¬tau a ⟹ p ⇒^a p' ⟹ ∃p'' q'. q ⇒^a q' ∧ p' ⊑cs q'> and
    sim_tau_max_p:
      <⋀p'. tau_max p' ⟹ p ⟼* tau p' ⟹ ∃q'. tau_max q' ∧ q ⟼* tau q' ∧ p' ≡cs q'>
  shows
    <p ⊑cs q>
proof-
  have
    <((∀a p'. p ⟼a  p' ⟶ (∃q'. q ⇒^a  q' ∧ p' ⊑cs q')) ∧
    (∃q'. q ⟼* tau  q' ∧ q' ⊑cs p))>
  proof safe
    fix a p'
    assume
      step: <p ⟼a  p'>
    thus <∃q'. q ⇒^a  q' ∧ p' ⊑cs  q'>
    proof (cases <tau a>)
      case True
      then obtain p'' where p''_spec: <p ⟼τ p''> <tau_max p''> <p' ⊑cs p''>
        using tau_shortcuts step tau_def steps_one_step[of p τ p']
        by (metis (no_types, lifting))
      then obtain q' where q'_spec: <q ⟼* tau q'> <p'' ≡cs q'>
        using sim_tau_max_p steps_one_step[OF step, of tau, OF ‘tau a‘]
          steps_one_step[of p τ p''] tau_def
        by metis
      then show ?thesis using ‘tau a‘ p''_spec(3) using coupledsim_trans by blast
    next
      case False
      then show ?thesis using sim_vis_p step_weak_step_tau[OF step] by blast
    qed
  next
    obtain p_max where <p ⟼* tau p_max> <tau_max p_max>
      using tau_max_deadlock contracted_cycles finite_taus by blast
    then obtain q_max where <q ⟼* tau  q_max> <q_max ⊑cs p_max>
      using sim_tau_max_p[of p_max] by force
    moreover have <p_max ⊑cs  p> using ‘p ⟼* tau p_max‘ coupledsim_step by blast
    ultimately show <∃q'. q ⟼* tau  q' ∧ q' ⊑cs  p>
      using coupledsim_trans by blast
  qed
  thus <p ⊑cs q> using coupledsim_unfold[symmetric] by auto
qed

lemma coupledsim_eq_reducible_2:
  assumes
    cs: <p ⊑cs q> and
    contracted_cycles: <⋀r1 r2 . r1 ⟼* tau r2 ∧ r2 ⟼* tau r1 ⟹ r1 = r2> and
    finite_taus: <⋀r. finite {r'. r ⟼* tau r'}>
  shows
    sim_vis_p:
      <⋀p' a. ¬tau a ⟹ p ⇒^a p' ⟹ ∃q'. q ⇒^a q' ∧ p' ⊑cs q'> and
    sim_tau_max_p:
      <⋀p'. tau_max p' ⟹ p ⟼* tau p' ⟹ ∃q'. tau_max q' ∧ q ⟼* tau q' ∧ p' ≡cs q'>
proof-
  fix p' a
  assume
    <¬ tau a>
```

```
    <p ⇒^a  p'>
  thus <∃q'. q ⇒^a  q' ∧ p' ⊑cs  q'>
    using cs unfolding coupled_simulation_weak_premise by blast
next
  fix p'
  assume step:
    <p ⟼* tau p'>
    <tau_max p'>
  hence <p ⇒^τ  p'>  by auto
  hence <∃ q'. tau_max q' ∧ q ⇒^τ q' ∧ p' ≡cs q'>
    using coupledsim_eventual_symmetry[OF _ finite_taus, of p q τ p']
      contracted_cycles cs step(2) by blast
  thus <∃ q'. tau_max q' ∧ q ⟼* tau q' ∧ p' ≡cs q'>
    by auto
qed
```

5.11 Reduction Semantics Coupled Simulation

The tradition to describe coupled simulation as special delay/weak simulation is quite common for coupled simulations on reduction semantics as in `cite "gp15" and "Fournet2005"`, of which `cite "gp15"` can also be found in the AFP `cite "Encodability_Process_Calculi-AFP"`. The notions coincide (for systems just with τ-transitions).

```
definition coupled_simulation_gp15 ::
  <('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <coupled_simulation_gp15 R ≡ ∀ p q p'. R p q ∧ (p ⟼* (λa. True) p') ⟶
    (∃ q'. (q ⟼* (λa. True) q') ∧ R p' q') ∧
    (∃ q'. (q ⟼* (λa. True) q') ∧ R q' p')>

lemma weak_bisim_implies_coupled_sim_gp15:
  assumes
    wbisim: <weak_bisimulation R> and
    symmetry: <⋀ p q . R p q ⟹ R q p>
  shows <coupled_simulation_gp15 R>
unfolding coupled_simulation_gp15_def proof safe
  fix p q p'
  assume Rpq: <R p q> <p ⟼* (λa. True)  p'>
  have always_tau: <⋀a. tau a ⟹ (λa. True) a> by simp
  hence <∃q'. q ⟼* (λa. True)  q' ∧ R p' q'>
    using steps_retain_weak_bisim[OF wbisim Rpq] by auto
  moreover hence <∃q'. q ⟼* (λa. True)  q' ∧ R q' p'>
    using symmetry by auto
  ultimately show
    <(∃q'. q ⟼* (λa. True)  q' ∧ R p' q')>
    <(∃q'. q ⟼* (λa. True)  q' ∧ R q' p')> .
qed

lemma coupledsim_gla17_implies_gp15:
  assumes
    <coupled_simulation R>
  shows
    <coupled_simulation_gp15 R>
  unfolding coupled_simulation_gp15_def
proof safe
  fix p q p'
  assume challenge:
```

```
    <R p q>
    <p ⟼*(λa. True)p'>
  have tau_true: <⋀a. tau a ⟹ (λa. True) a> by simp
  thus <∃q'. q ⟼* (λa. True)  q' ∧ R p' q'>
    using steps_retain_weak_sim assms challenge
    unfolding coupled_simulation_weak_simulation by meson
  then obtain q' where q'_def: <q ⟼* (λa. True)  q'> <R p' q'> by blast
  then obtain q'' where <q' ⟼* tau  q''> <R q'' p'>
    using assms unfolding coupled_simulation_weak_simulation by blast
  moreover hence <q ⟼* (λa. True)  q''>
    using q'_def(1) steps_concat steps_spec tau_true by meson
  ultimately show <∃q'. q ⟼* (λa. True)  q' ∧ R q' p'> by blast
qed

lemma coupledsim_gp15_implies_gla17_on_tau_systems:
  assumes
    <coupled_simulation_gp15 R>
    <⋀ a . tau a>
  shows
    <coupled_simulation R>
  unfolding coupled_simulation_def
proof safe
  fix p q p' a
  assume challenge:
    <R p q>
    <p ⟼a  p'>
  hence <p ⟼* (λa. True)  p'> using steps_one_step by metis
  then obtain q' where <q ⟼* (λa. True)  q'> <R p' q'>
    using challenge(1) assms(1) unfolding coupled_simulation_gp15_def by blast
  hence <q ⇒^a  q'> using assms(2) steps_concat steps_spec by meson
  thus <∃q'. R p' q' ∧ q ⇒^a  q'> using `R p' q'` by blast
next
  fix p q
  assume
    <R p q>
  moreover have <p ⟼* (λa. True) p> using steps.refl by blast
  ultimately have <∃q'. q ⟼* (λa. True)  q' ∧ R q' p>
    using assms(1) unfolding coupled_simulation_gp15_def by blast
  thus <∃q'. q ⟼* tau  q' ∧ R q' p> using assms(2) steps_spec by blast
qed
```

5.12 Coupled Simulation as Two Simulations

Historically, coupled similarity has been defined in terms of `emph <two>` weak simulations coupled in some way `cite "sangiorgi2012" and "ps1994"`. We reproduce these (more well-known) formulations and show that they are equivalent to the coupled (delay) simulations we are using.

```
definition coupled_simulation_san12 ::
  <('s ⇒ 's ⇒ bool) ⇒ ('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <coupled_simulation_san12 R1 R2 ≡
    weak_simulation R1 ∧ weak_simulation (λ p q . R2 q p)
  ∧ (∀ p q . R1 p q ⟶ (∃ q' . q ⟼* tau q' ∧ R2 p q'))
  ∧ (∀ p q . R2 p q ⟶ (∃ p' . p ⟼* tau p' ∧ R1 p' q))>

lemma weak_bisim_implies_coupled_sim_san12:
  assumes <weak_bisimulation R>
  shows <coupled_simulation_san12 R R>
```

```
  using assms weak_bisim_weak_sim steps.refl[of _ tau]
  unfolding coupled_simulation_san12_def
  by blast

lemma coupledsim_gla17_resembles_san12:
  shows
    <coupled_simulation R1 =
    coupled_simulation_san12 R1 (λ p q . R1 q p)>
  unfolding coupled_simulation_weak_simulation coupled_simulation_san12_def by blast

lemma coupledsim_san12_impl_gla17:
  assumes
    <coupled_simulation_san12 R1 R2>
  shows
    <coupled_simulation (λ p q. R1 p q ∨ R2 q p)>
  unfolding coupled_simulation_weak_simulation
proof safe
  have <weak_simulation R1> <weak_simulation (λp q. R2 q p)>
    using assms unfolding coupled_simulation_san12_def by auto
  thus <weak_simulation (λp q. R1 p q ∨ R2 q p)>
    using weak_sim_union_cl by blast
next
  fix p q
  assume
    <R1 p q>
  hence <∃q'. q ⟼* tau  q' ∧ R2 p q'>
    using assms unfolding coupled_simulation_san12_def by auto
  thus <∃q'. q ⟼* tau  q' ∧ (R1 q' p ∨ R2 p q')> by blast
next
  fix p q
  assume
    <R2 q p>
  hence <∃q'. q ⟼* tau  q' ∧ R1 q' p>
    using assms unfolding coupled_simulation_san12_def by auto
  thus <∃q'. q ⟼* tau  q' ∧ (R1 q' p ∨ R2 p q')> by blast
qed
```

5.13 S-coupled Simulation

Originally coupled simulation was introduced as two weak simulations coupled at the stable states. We give the definitions from `cite "parrow1992" and "ps1994"` and a proof connecting this notion to our coupled similarity in the absence of divergences following `cite "sangiorgi2012"`.

```
definition coupled_simulation_p92 ::
  <('s ⇒ 's ⇒ bool) ⇒ ('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <coupled_simulation_p92 R1 R2 ≡ ∀ p q .
    (R1 p q ⟶
      ((∀ p' a. p ⟼a p' ⟶
        (∃ q'. R1 p' q' ∧
          (q ⇒^a q'))) ∧
      (stable_state p ⟶ R2 p q))) ∧
    (R2 p q ⟶
      ((∀ q' a. q ⟼a q' ⟶
        (∃ p'. R2 p' q' ∧
          (p ⇒^a p'))) ∧
      (stable_state q ⟶ R1 p q)))>
```

```
lemma weak_bisim_implies_coupled_sim_p92:
  assumes <weak_bisimulation R>
  shows <coupled_simulation_p92 R R>
using assms unfolding weak_bisimulation_def coupled_simulation_p92_def by blast

lemma coupled_sim_p92_symm:
  assumes <coupled_simulation_p92 R1 R2>
  shows <coupled_simulation_p92 (λ p q. R2 q p) (λ p q. R1 q p)>
using assms unfolding coupled_simulation_p92_def by blast

definition s_coupled_simulation_san12 ::
  <('s ⇒ 's ⇒ bool) ⇒ ('s ⇒ 's ⇒ bool) ⇒ bool>
where
  <s_coupled_simulation_san12 R1 R2 ≡
    weak_simulation R1 ∧ weak_simulation (λ p q . R2 q p)
  ∧ (∀ p q . R1 p q ⟶ stable_state p ⟶ R2 p q)
  ∧ (∀ p q . R2 p q ⟶ stable_state q ⟶ R1 p q)>

abbreviation s_coupled_simulated_by :: <'s ⇒ 's ⇒ bool> ("_ ⊑scs  _" [60, 60] 65)
  where <s_coupled_simulated_by p q ≡
    ∃ R1 R2 . s_coupled_simulation_san12 R1 R2 ∧ R1 p q>

abbreviation s_coupled_similar :: <'s ⇒ 's ⇒ bool> ("_ ≡scs  _" [60, 60] 65)
  where <s_coupled_similar p q ≡
    ∃ R1 R2 . s_coupled_simulation_san12 R1 R2 ∧ R1 p q ∧ R2 p q>

lemma s_coupled_sim_is_original_coupled:
  <s_coupled_simulation_san12 = coupled_simulation_p92>
unfolding coupled_simulation_p92_def
  s_coupled_simulation_san12_def weak_simulation_def by blast

corollary weak_bisim_implies_s_coupled_sim:
  assumes <weak_bisimulation R>
  shows <s_coupled_simulation_san12 R R>
  using assms s_coupled_sim_is_original_coupled weak_bisim_implies_coupled_sim_p92 by simp

corollary s_coupled_sim_symm:
  assumes <s_coupled_simulation_san12 R1 R2>
  shows <s_coupled_simulation_san12 (λ p q. R2 q p) (λ p q. R1 q p)>
  using assms coupled_sim_p92_symm s_coupled_sim_is_original_coupled by simp

corollary s_coupled_sim_union_cl:
  assumes
    <s_coupled_simulation_san12 RA1 RA2>
    <s_coupled_simulation_san12 RB1 RB2>
  shows
    <s_coupled_simulation_san12 (λ p q. RA1 p q ∨ RB1 p q) (λ p q. RA2 p q ∨ RB2 p q)>
  using assms weak_sim_union_cl unfolding s_coupled_simulation_san12_def by auto

corollary s_coupled_sim_symm_union:
  assumes <s_coupled_simulation_san12 R1 R2>
  shows <s_coupled_simulation_san12 (λ p q. R1 p q ∨ R2 q p) (λ p q. R2 p q ∨ R1 q p)>
  using s_coupled_sim_union_cl[OF assms s_coupled_sim_symm[OF assms]] .

lemma s_coupledsim_stable_eq:
  assumes
    <p ⊑scs q>
```

```
    <stable_state p>
  shows  <p ≡scs q>
proof -
  obtain R1 R2 where
      <R1 p q>
      <weak_simulation R1>
      <weak_simulation (λp q. R2 q p)>
      <∀p q. R1 p q ⟶ stable_state p ⟶ R2 p q>
      <∀p q. R2 p q ⟶ stable_state q ⟶ R1 p q>
    using assms(1)  unfolding s_coupled_simulation_san12_def by blast
  moreover hence <R2 p q> using assms(2) by blast
  ultimately show ?thesis unfolding s_coupled_simulation_san12_def by blast
qed

lemma s_coupledsim_symm:
  assumes
    <p ≡scs q>
  shows
    <q ≡scs p>
  using assms s_coupled_sim_symm by blast

lemma s_coupledsim_eq_parts:
  assumes
    <p ≡scs q>
  shows
    <p ⊑scs q>
    <q ⊑scs p>
  using assms s_coupledsim_symm by metis+
```

— From `cite "sangiorgi2012"`, p. 226

```
lemma divergence_free_coupledsims_coincidence_1:
  defines
    <R1 ≡ (λ p q . p ⊑cs q ∧ (stable_state p ⟶ stable_state q))> and
    <R2 ≡ (λ p q . q ⊑cs p ∧ (stable_state q ⟶ stable_state p))>
  assumes
    non_divergent_system: <⋀ p . ¬ divergent_state p>
  shows
    <s_coupled_simulation_san12 R1 R2>
  unfolding s_coupled_simulation_san12_def
proof safe
  show <weak_simulation R1> unfolding weak_simulation_def
  proof safe
    fix p q p' a
    assume sub_assms:
      <R1 p q>
      <p ⟼a  p'>
    then obtain q' where q'_def: <q ⇒^a q'> <p' ⊑cs q'>
      using coupled_sim_by_is_coupled_sim unfolding R1_def coupled_simulation_def by blast
    show <∃q'. R1 p' q' ∧ q ⇒^a  q'>
    proof (cases <stable_state p'>)
      case True
      obtain  q'' where q''_def: <q' ⟼* tau q''> <q'' ⊑cs  p'>
        using coupled_sim_by_is_coupled_sim q'_def(2)
        unfolding coupled_simulation_weak_simulation by blast
      then obtain q''' where q'''_def: <q'' ⟼* tau q'''> <stable_state q'''>
        using non_divergence_implies_eventual_stability non_divergent_system by blast
      hence <q''' ⊑cs p'>
```

```
        using coupledsim_step_gla17 coupledsim_trans[OF _ q''_def(2)] by blast
      hence <p' ⊑cs q'''>
        using ‘stable_state p''‘ coupled_sim_by_is_coupled_sim coupledsim_stable_state_symm
        by blast
      moreover have <q ⇒^a q'''> using q'_def(1) q''_def(1) q'''_def(1) steps_concat by blast
      ultimately show ?thesis using q'''_def(2) unfolding R1_def by blast
    next
      case False
      then show ?thesis using q'_def unfolding R1_def by blast
    qed
  qed
  — analogous to previous case
  then show <weak_simulation (λp q. R2 q p)> unfolding R1_def R2_def .
next
  fix p q
  assume
    <R1 p q>
    <stable_state p>
  thus <R2 p q>
    unfolding R1_def R2_def
    using coupled_sim_by_is_coupled_sim coupledsim_stable_state_symm by blast
next — analogous
  fix p q
  assume
    <R2 p q>
    <stable_state q>
  thus <R1 p q>
    unfolding R1_def R2_def
    using coupled_sim_by_is_coupled_sim coupledsim_stable_state_symm by blast
qed

— From cite "sangiorgi2012", p. 227
lemma divergence_free_coupledsims_coincidence_2:
  defines
    <R ≡ (λ p q . p ⊑scs q ∨ (∃ q' . q ⟼* tau q' ∧ p ≡scs q'))>
  assumes
    non_divergent_system: <⋀ p . ¬ divergent_state p>
  shows
    <coupled_simulation R>
  unfolding coupled_simulation_weak_simulation
proof safe
  show <weak_simulation R>
    unfolding weak_simulation_def
  proof safe
    fix p q p' a
    assume sub_assms:
      <R p q>
      <p ⟼a  p'>
    thus <∃q'. R p' q' ∧ q ⇒^a  q'>
      unfolding R_def
    proof (cases <p ⊑scs q>)
      case True
      then obtain q' where <p' ⊑scs  q'> <q ⇒^a  q'>
        using s_coupled_simulation_san12_def sub_assms(2) weak_sim_ruleformat by metis
      thus <∃q'. (p' ⊑scs  q' ∨ (∃q'a. q' ⟼* tau  q'a ∧ p' ≡scs  q'a)) ∧ q ⇒^a  q'>
        by blast
    next
```

```
      case False
      then obtain q' where <q ⟼* tau  q'> <p ≡scs  q'>
          using sub_assms(1) unfolding R_def by blast
      then obtain q'' where <q' ⇒^a  q''> <p' ⊑scs  q''>
        using s_coupled_simulation_san12_def sub_assms(2) weak_sim_ruleformat by metis
      hence <p' ⊑scs  q'' ∧ q ⇒^a  q''> using steps_concat `q ⟼* tau  q'` by blast
      thus <∃q'. (p' ⊑scs  q' ∨ (∃q'a. q' ⟼* tau  q'a ∧ p' ≡scs  q'a)) ∧ q ⇒^a  q'>
        by blast
    qed
  qed
next
  fix p q
  assume
    <R p q>
  thus <∃q'. q ⟼* tau  q' ∧ R q' p> unfolding R_def
  proof safe
    fix R1 R2
    assume sub_assms:
      <s_coupled_simulation_san12 R1 R2>
      <R1 p q>
    thus <∃q'. q ⟼* tau  q' ∧ (q' ⊑scs  p ∨ (∃q'a. p ⟼* tau  q'a ∧ q' ≡scs  q'a))>
    proof -
      — dropped a superfluous case distinction from @citesangiorgi2012
      obtain p' where <stable_state p'> <p ⟼* tau p'>
        using non_divergent_system non_divergence_implies_eventual_stability by blast
      hence <p ⇒^τ  p'> using tau_tau by blast
      then obtain q' where <q ⟼* tau q'> <p' ⊑scs q'>
        using s_coupled_simulation_san12_def weak_sim_weak_premise sub_assms tau_tau
        by metis
      moreover hence <p' ≡scs q'> using `stable_state p'` s_coupledsim_stable_eq by blast
      ultimately show ?thesis using `p ⟼* tau p'` s_coupledsim_symm by blast
    qed
  qed (metis s_coupledsim_symm)
qed
```

While this proof follows `cite "sangiorgi2012"`, we needed to deviate from them by also requiring rootedness (shared stability) for the compared states.

```
theorem divergence_free_coupledsims_coincidence:
  assumes
    non_divergent_system: <⋀ p . ¬ divergent_state p> and
    stability_rooted: <stable_state p ⟷ stable_state q>
  shows
    <(p ≡cs q) = (p ≡scs q)>
proof rule
  assume <p ≡cs q>
  hence <p ⊑cs q> <q ⊑cs p> by auto
  thus <p ≡scs q>
    using stability_rooted divergence_free_coupledsims_coincidence_1[OF non_divergent_system]
    by blast
next
  assume <p ≡scs q>
  thus <p ≡cs q>
    using stability_rooted divergence_free_coupledsims_coincidence_2[OF non_divergent_system]
      s_coupledsim_eq_parts by blast
qed

end — context lts_tau
```

The following example shows that a system might be related by s-coupled-simulation without being connected by coupled-simulation.

```
datatype ex_state = a0 | a1 | a2 | a3 | b0 | b1 | b2

locale ex_lts = lts_tau trans τ
  for trans :: <ex_state ⇒ nat ⇒ ex_state ⇒ bool> ("_ ⟼_  _" [70, 70, 70] 80) and τ +
  assumes
    sys:
  <trans = (λ p act q .
     1 = act ∧ (p = a0 ∧ q = a1
               ∨ p = a0 ∧ q = a2
               ∨ p = a2 ∧ q = a3
               ∨ p = b0 ∧ q = b1
               ∨ p = b1 ∧ q = b2) ∨
     0 = act ∧ (p = a1 ∧ q = a1))>
   <τ = 0>
begin

lemma no_root_coupled_sim:
  fixes R1 R2
  assumes
    coupled:
      <coupled_simulation_san12 R1 R2> and
    root:
      <R1 a0 b0> <R2 a0 b0>
  shows
    False
proof -
  have
    R1sim:
      <weak_simulation R1> and
    R1coupling:
      <∀p q. R1 p q ⟶ (∃q'. q ⟼* tau  q' ∧ R2 p q')> and
    R2sim:
      <weak_simulation (λp q. R2 q p)>
    using coupled unfolding coupled_simulation_san12_def by auto
  hence R1sim_rf:
      <⋀ p q. R1 p q ⟹
        (∀p' a. p ⟼a  p' ⟶
          (∃q'. R1 p' q' ∧ (¬ tau a ⟶ q ⇒a  q') ∧
          (tau a ⟶ q ⟼* tau  q')))>
    unfolding weak_simulation_def by blast
  have <a0 ⟼1 a1> using sys by auto
  hence <∃q'. R1 a1 q' ∧ b0 ⇒1 q'>
    using R1sim_rf[OF root(1), rule_format, of 1 a1] tau_def
    by (auto simp add: sys)
  then obtain q' where q': <R1 a1 q'> <b0 ⇒1 q'> by blast
  have b0_quasi_stable: <∀ q' . b0 ⟼*tau q' ⟶ b0 = q'>
    using steps_no_step[of b0 tau] tau_def by (auto simp add: sys)
  have b0_only_b1: <∀ q' . b0 ⟼1 q' ⟶ q' = b1> by (auto simp add: sys)
  have b1_quasi_stable: <∀ q' . b1 ⟼*tau q' ⟶ b1 = q'>
    using steps_no_step[of b1 tau] tau_def by (auto simp add: sys)
  have <∀ q' . b0 ⇒1 q' ⟶ q' = b1>
    using b0_quasi_stable b0_only_b1 b1_quasi_stable by auto
  hence <q' = b1> using q'(2) by blast
  hence <R1 a1 b1> using q'(1) by simp
  hence <R2 a1 b1>
```

```
    using b1_quasi_stable R1coupling by auto
  have b1_b2: <b1 ⟼1 b2>
    by (auto simp add: sys)
  hence a1_sim: <∃ q' . R2 q' b2 ∧ a1 ⇒1  q'>
    using ‘R2 a1 b1‘ R2sim b1_b2
    unfolding weak_simulation_def tau_def by (auto simp add: sys)
  have a1_quasi_stable: <∀ q' . a1 ⟼*tau q' ⟶ a1 = q'>
    using steps_loop[of a1] by (auto simp add: sys)
  hence a1_stuck: <∀ q' . ¬ a1 ⇒1 q'>
    by (auto simp add: sys)
  show ?thesis using a1_sim  a1_stuck by blast
qed

lemma root_s_coupled_sim:
  defines
    <R1 ≡ λ a b .
      a = a0 ∧ b = b0 ∨
      a = a1 ∧ b = b1 ∨
      a = a2 ∧ b = b1 ∨
      a = a3 ∧ b = b2>
  and
    <R2 ≡ λ a b .
      a = a0 ∧ b = b0 ∨
      a = a2 ∧ b = b1 ∨
      a = a3 ∧ b = b2>
  shows
    coupled:
      <s_coupled_simulation_san12 R1 R2>
  unfolding s_coupled_simulation_san12_def
proof safe
  show <weak_simulation R1>
    unfolding weak_simulation_def proof (clarify)
    fix p q p' a
    show <R1 p q ⟹ p ⟼a  p' ⟹ ∃q'. R1 p' q' ∧ (q ⇒^a  q')>
      using step_tau_refl unfolding sys assms tau_def using sys(2) tau_def by (cases p, auto)
  qed
next
  show <weak_simulation (λp q. R2 q p)>
    unfolding weak_simulation_def proof (clarify)
    fix p q p' a
    show <R2 q p ⟹ p ⟼a  p' ⟹ ∃q'. R2 q' p' ∧ (q ⇒^a  q')>
      using steps.refl[of _ tau] tau_def unfolding assms sys
      using sys(2) tau_def by (cases p, auto)
  qed
next
  fix p q
  assume <R1 p q> <stable_state p>
  thus <R2 p q> unfolding assms sys using sys(2) tau_def by auto
next
  fix p q
  assume <R2 p q> <stable_state q>
  thus <R1 p q> unfolding assms sys using tau_def by auto
qed

end — ex_lts// example lts

end
```

6 Game for Coupled Similarity with Delay Formulation

```
theory Coupledsim_Game_Delay
imports
  Coupled_Simulation
  Simple_Game
begin
```

6.1 The Coupled Simulation Preorder Game Using Delay Steps

```
datatype ('s, 'a) cs_game_node =
  AttackerNode 's 's |
  DefenderStepNode 'a 's 's |
  DefenderCouplingNode 's 's

fun (in lts_tau) cs_game_moves ::
  <('s, 'a) cs_game_node ⇒ ('s, 'a) cs_game_node ⇒ bool> where
  simulation_visible_challenge:
    <cs_game_moves (AttackerNode p q) (DefenderStepNode a p1 q0) =
      (¬tau a ∧ p ⟼a p1 ∧ q = q0)> |
  simulation_internal_attacker_move:
    <cs_game_moves (AttackerNode p q) (AttackerNode p1 q0) =
      (∃a. tau a ∧ p ⟼a p1 ∧ q = q0)> |
  simulation_answer:
    <cs_game_moves (DefenderStepNode a p1 q0) (AttackerNode p11 q1) =
      (q0 =▷a q1 ∧ p1 = p11)> |
  coupling_challenge:
    <cs_game_moves (AttackerNode p q) (DefenderCouplingNode p0 q0) =
      (p = p0 ∧ q = q0)> |
  coupling_answer:
    <cs_game_moves (DefenderCouplingNode p0 q0) (AttackerNode q1 p00) =
      (p0 = p00 ∧ q0 ⟼* tau q1)> |
  cs_game_moves_no_step:
    <cs_game_moves _ _ = False>

fun cs_game_defender_node :: <('s, 'a) cs_game_node ⇒ bool> where
  <cs_game_defender_node (AttackerNode _ _) = False> |
  <cs_game_defender_node (DefenderStepNode _ _ _) = True> |
  <cs_game_defender_node (DefenderCouplingNode _ _) = True>

locale cs_game =
  lts_tau trans τ +
  simple_game cs_game_moves cs_game_defender_node
for
  trans :: <'s ⇒ 'a ⇒ 's ⇒ bool> ("_ ⟼_  _" [70, 70, 70] 80) and
  τ :: <'a>
begin
```

6.2 Coupled Simulation Implies Winning Strategy

```
fun strategy_from_coupleddsim :: <('s ⇒ 's ⇒ bool) ⇒ ('s, 'a) cs_game_node strategy> where
  <strategy_from_coupleddsim R ((DefenderStepNode a p1 q0)#play) =
    (AttackerNode p1 (SOME q1 . R p1 q1 ∧ q0 =▷a q1))> |
  <strategy_from_coupleddsim R ((DefenderCouplingNode p0 q0)#play) =
    (AttackerNode (SOME q1 . R q1 p0 ∧ q0 ⟼* tau q1) p0)> |
  <strategy_from_coupleddsim _ _ = undefined>
```

```
lemma defender_preceded_by_attacker:
  assumes
    <n0 # play ∈ plays (AttackerNode p0 q0)>
    <cs_game_defender_node n0>
  shows
    <∃ p q . hd play = AttackerNode p q ∧ cs_game_moves (AttackerNode p q) n0>
    <play ≠ []>
proof -
  have n0_not_init: <n0 ≠ (AttackerNode p0 q0)> using assms(2) by auto
  hence <cs_game_moves (hd play) n0> using assms(1)
    by (metis list.sel(1) list.sel(3) plays.cases)
  thus <∃p q. hd play = AttackerNode p q ∧ cs_game_moves (AttackerNode p q) n0> using assms(2)
    by (metis cs_game_defender_node.elims(2,3) local.cs_game_moves_no_step(1,2,3,6))
  show <play ≠ []> using n0_not_init assms(1) plays.cases by auto
qed

lemma defender_only_challenged_by_visible_actions:
  assumes
    <(DefenderStepNode a p q) # play ∈ plays (AttackerNode p0 q0)>
  shows
    <¬tau a>
  using assms defender_preceded_by_attacker
  by fastforce

lemma strategy_from_coupleddsim_retains_coupledsim:
  assumes
    <R p0 q0>
    <coupled_delay_simulation R>
    <initial = AttackerNode p0 q0>
    <play ∈ plays_for_0strategy (strategy_from_coupleddsim R) initial>
  shows
    <hd play = AttackerNode p q ⟹ R p q>
    <length play > 1 ⟹ hd (tl play) = AttackerNode p q ⟹ R p q>
  using assms(4)
proof (induct arbitrary: p q rule: plays_for_0strategy.induct[OF assms(4)])
  case 1
  fix p q
  assume <hd [initial] = AttackerNode p q>
  hence <p = p0> <q = q0> using assms(3) by auto
  thus <R p q> using assms(1) by simp
next
  case 1
  fix p q
  assume <1 < length [initial]>
  hence False by auto
  thus <R p q>  ..
next
  case (2 n0 play)
  hence n0play_is_play: <n0 # play ∈ plays initial> using strategy0_plays_subset by blast
  fix p q
  assume subassms:
    <hd (strategy_from_coupleddsim R (n0 # play) # n0 # play) = AttackerNode p q>
    <strategy_from_coupleddsim R (n0 # play) # n0 # play
      ∈ plays_for_0strategy (strategy_from_coupleddsim R) initial>
  then obtain pi qi where
      piqi_def: <hd (play) = AttackerNode pi qi>
        <cs_game_moves (AttackerNode pi qi) n0> <play ≠ []>
```

```
    using defender_preceded_by_attacker n0play_is_play ‘cs_game_defender_node n0‘ assms(3)
by blast
  hence <R pi qi> using 2(1,3) by simp
  have <(∃ a . n0 = (DefenderStepNode a p qi) ∧ ¬ tau a ∧ pi ⟼a p)
    ∨ (n0 = (DefenderCouplingNode pi qi))>
    using piqi_def(2) 2(4,5) subassms(1)
    using cs_game_defender_node.elims(2) cs_game_moves.simps(1,3)
      cs_game_moves.simps(4) list.sel(1)
    by metis
  thus <R p q>
  proof safe
    fix a
    assume n0_def: <n0 = DefenderStepNode a p qi> <¬ tau a> <pi ⟼a p>
    have <strategy_from_coupleddsim R (n0 # play) =
        (AttackerNode p (SOME q1 . R p q1 ∧ qi =▷a q1))>
      unfolding n0_def(1) by auto
    with subassms(1) have q_def: <q = (SOME q1. R p q1 ∧ qi =▷a  q1)> by auto
    have <∃ qii . R p qii ∧ qi =▷a qii>
      using n0_def(2,3) ‘R pi qi‘ ‘coupled_delay_simulation R‘
      unfolding coupled_delay_simulation_def delay_simulation_def by blast
    from someI_ex[OF this] show <R p q> unfolding q_def by blast
  next
    assume n0_def: <n0 = DefenderCouplingNode pi qi>
    have <strategy_from_coupleddsim R (n0 # play) =
        (AttackerNode (SOME q1 . R q1 pi ∧ qi ⟼* tau q1) pi)>
      unfolding n0_def(1) by auto
    with subassms(1) have qp_def:
      <p = (SOME q1 . R q1 pi ∧ qi ⟼* tau q1)> <q = pi> by auto
    have <∃ q1 . R q1 pi ∧ qi ⟼* tau q1>
      using n0_def ‘R pi qi‘ ‘coupled_delay_simulation R‘
      unfolding coupled_delay_simulation_def by blast
    from someI_ex[OF this] show <R p q> unfolding qp_def by blast
  qed
next
  case (2 n0 play)
  fix p q
  assume <hd (tl (strategy_from_coupleddsim R (n0 # play) # n0 # play)) = AttackerNode p q>
  hence False using 2(4) by auto
  thus <R p q> ..
next
  case (3 n1 play n1')
  fix p q
  assume <hd (n1' # n1 # play) = AttackerNode p q>
  then obtain p1 a where n1_spec: <n1 = AttackerNode p1 q> <p1 ⟼a p> <tau a>
   using 3 list.sel(1)
   by (metis cs_game_defender_node.elims(3) simulation_internal_attacker_move)
  then have <R p1 q> using 3 by auto
  thus <R p q>
    using  n1_spec(2,3) <coupled_delay_simulation R>
    unfolding coupled_delay_simulation_def delay_simulation_def by auto
next
  case (3 n1 play n1')
  fix p q
  assume <hd (tl (n1' # n1 # play)) = AttackerNode p q>
  thus <R p q> using 3(1,2) by auto
qed
```

```
lemma strategy_from_coupleddsim_sound:
  assumes
    <R p0 q0>
    <coupled_delay_simulation R>
    <initial = AttackerNode p0 q0>
  shows
    <sound_0strategy (strategy_from_coupleddsim R) initial>
  unfolding sound_0strategy_def
proof clarify
  fix n0 play
  assume subassms:
    <n0 # play ∈ plays_for_0strategy(strategy_from_coupleddsim R) initial>
    <cs_game_defender_node n0>
  then obtain pi qi where
      piqi_def: <hd (play) = AttackerNode pi qi>
        <cs_game_moves (AttackerNode pi qi) n0> <play ≠ []>
    using defender_preceded_by_attacker `cs_game_defender_node n0` assms(3)
      strategy0_plays_subset by blast
  hence <R pi qi>
    using strategy_from_coupleddsim_retains_coupledsim[OF assms] list.sel subassms by auto
  have <(∃ a p . n0 = (DefenderStepNode a p qi) ∧ pi ⟼a p)
    ∨ (n0 = (DefenderCouplingNode pi qi))>
    by (metis cs_game_defender_node.elims(2)
        coupling_challenge simulation_visible_challenge piqi_def(2) subassms(2))
  thus <cs_game_moves n0 (strategy_from_coupleddsim R (n0 # play))>
  proof safe
    fix a p
    assume dsn:
      <pi ⟼a  p>
      <n0 = DefenderStepNode a p qi>
    hence qi_spec:
      <(strategy_from_coupleddsim R (n0 # play)) =
        AttackerNode p (SOME q1 . R p q1 ∧ qi =▷a q1)>
      by simp
    then obtain qii where qii_spec:
      <AttackerNode p (SOME q1 . R p q1 ∧ qi =▷a q1) = AttackerNode p qii> by blast
    have <∃ qii . R p qii ∧ qi =▷a qii>
      using dsn `R pi qi` `coupled_delay_simulation R` steps.refl
      unfolding coupled_delay_simulation_def delay_simulation_def by blast
    from someI_ex[OF this] have <R p qii ∧ qi =▷a qii> using qii_spec by blast
    thus <cs_game_moves (DefenderStepNode a p qi)
          (strategy_from_coupleddsim R (DefenderStepNode a p qi # play))>
      using qi_spec qii_spec unfolding dsn(2) by auto
  next — coupling quite analogous.
    assume dcn:
      <n0 = DefenderCouplingNode pi qi>
    hence qi_spec:
      <(strategy_from_coupleddsim R (n0 # play)) =
      AttackerNode (SOME q1 . R q1 pi ∧ qi ⟼* tau q1) pi>
      by simp
    then obtain qii where qii_spec:
      <AttackerNode (SOME q1 . R q1 pi ∧ qi ⟼* tau q1) pi = AttackerNode qii pi> by blast
    have <∃ qii . R qii pi ∧ qi ⟼* tau qii>
      using dcn `R pi qi` `coupled_delay_simulation R`
      unfolding coupled_delay_simulation_def by blast
    from someI_ex[OF this] have <R qii pi ∧ qi ⟼* tau qii> using qii_spec by blast
    thus <cs_game_moves (DefenderCouplingNode pi qi)
```

```
        (strategy_from_coupleddsim R (DefenderCouplingNode pi qi # play))>
      using qi_spec qii_spec unfolding dcn by auto
  qed
qed

lemma coupleddsim_implies_winning_strategy:
  assumes
    <R p q>
    <coupled_delay_simulation R>
    <initial = AttackerNode p q>
  shows
    <player0_winning_strategy (strategy_from_coupleddsim R) initial>
  unfolding player0_winning_strategy_def
proof (clarify)
  fix play
  assume subassms:
    <play ∈ plays_for_0strategy (strategy_from_coupleddsim R) initial>
    <player1_wins_immediately play>
  show <False> using subassms
  proof (induct rule: simple_game.plays_for_0strategy.induct[OF subassms(1)])
    case 1
    then show ?case unfolding player1_wins_immediately_def using assms(3) by auto
  next
    case (2 n0 play)
    hence <¬ cs_game_defender_node (strategy_from_coupleddsim R (n0 # play))>
      using cs_game_moves_no_step cs_game_defender_node.elims(2) by metis
    hence <¬  player1_wins_immediately (strategy_from_coupleddsim R (n0 # play) # n0 # play)>
      unfolding player1_wins_immediately_def by auto
    thus ?case using 2(6) by auto
  next
    case (3 n1 play n1')
    then obtain p q where n1_def: <n1 = AttackerNode p q>
      using cs_game_defender_node.elims(3) by blast
    hence <R p q>
      using strategy_from_coupleddsim_retains_coupledsim[OF assms, of <n1 # play>] 3(1)
      by auto
    have <(∃ p1 a . n1' = (DefenderStepNode a p1 q) ∧ (p ⟼a p1))
        ∨ n1' = (DefenderCouplingNode p q)>
      using n1_def ‘cs_game_moves n1 n1'‘ coupling_challenge cs_game_moves_no_step(5)
        simulation_visible_challenge "3.prems"(2) cs_game_defender_node.elims(1) list.sel(1)
      unfolding player1_wins_immediately_def
      by metis
    then show ?case
    proof
      assume <(∃ p1 a . n1' = (DefenderStepNode a p1 q) ∧ (p ⟼a p1))>
      then obtain p1 a where
        n1'_def: <n1' = (DefenderStepNode a p1 q)> <p ⟼a p1>
        by blast
      hence <∃ q1 . q =▷a q1>
        using ‘R p q‘ ‘coupled_delay_simulation R‘
        unfolding coupled_delay_simulation_def delay_simulation_def by blast
      hence <∃ q1 . cs_game_moves (DefenderStepNode a p1 q) (AttackerNode p1 q1)>
        by auto
      with ‘player1_wins_immediately (n1' # n1 # play)‘ show False
        unfolding player1_wins_immediately_def n1'_def
        by (metis list.sel(1))
    next
```

```
      assume n1'_def: <n1' = DefenderCouplingNode p q>
      have <∃ q1 . q ⟼*tau q1>
        using `coupled_delay_simulation R` `R p q`
        unfolding coupled_delay_simulation_def by blast
      hence <∃ q1 . cs_game_moves (DefenderCouplingNode p q) (AttackerNode q1 p)>
        by auto
      with `player1_wins_immediately (n1' # n1 # play)` show False
        unfolding player1_wins_immediately_def n1'_def
        by (metis list.sel(1))
    qed
  qed
qed
```

6.3 Winning Strategy Induces Coupled Simulation

```
lemma winning_strategy_implies_coupleddsim:
  assumes
    <player0_winning_strategy f initial>
    <sound_0strategy f initial>
  defines
    <R == λ p q . (∃ play ∈ plays_for_0strategy f initial. hd play = AttackerNode p q)>
  shows
    <coupled_delay_simulation R>
  unfolding coupled_delay_simulation_def delay_simulation_def
proof safe
  fix p q p' a
  assume challenge:
    <R p q>
    <p ⟼a  p'>
    <tau a >
  hence game_move: <cs_game_moves (AttackerNode p q) (AttackerNode p' q)> by auto
  have <(∃ play ∈ plays_for_0strategy f initial. hd play = AttackerNode p q)>
    using challenge(1) assms by blast
  then obtain play where
      play_spec: <AttackerNode p q # play ∈ plays_for_0strategy f initial>
    by (metis list.sel(1) simple_game.plays.cases strategy0_plays_subset)
  hence interplay:
      <(AttackerNode p' q) # AttackerNode p q # play ∈ plays_for_0strategy f initial>
    using game_move by (simp add: simple_game.plays_for_0strategy.p1move)
  then show <R p' q>
    unfolding R_def list.sel by force
next
  fix p q p' a
  assume challenge:
    <R p q>
    <p ⟼a  p'>
    <¬ tau a >
  hence game_move: <cs_game_moves (AttackerNode p q) (DefenderStepNode a p' q)> by auto
  have <(∃ play ∈ plays_for_0strategy f initial. hd play = AttackerNode p q)>
    using challenge(1) assms by blast
  then obtain play where
      play_spec: <AttackerNode p q # play ∈ plays_for_0strategy f initial>
    by (metis list.sel(1) simple_game.plays.cases strategy0_plays_subset)
  hence interplay:
      <(DefenderStepNode a p' q) # AttackerNode p q # play ∈ plays_for_0strategy f initial>
    using game_move by (simp add: simple_game.plays_for_0strategy.p1move)
  hence <¬ player1_wins_immediately ((DefenderStepNode a p' q) # AttackerNode p q # play)>
```

```
    using assms(1) unfolding player0_winning_strategy_def by blast
  then obtain n1 where n1_def:
      ‹n1 = f (DefenderStepNode a p' q # AttackerNode p q # play)›
      ‹cs_game_moves (DefenderStepNode a p' q) n1›
    using interplay assms(2) unfolding player0_winning_strategy_def sound_0strategy_def by
simp
  obtain q' where q'_spec:
      ‹(AttackerNode p' q') = n1› ‹q =▷a q'›
    using n1_def(2) by (cases n1, auto)
  hence ‹(AttackerNode p' q') # (DefenderStepNode a p' q) # AttackerNode p q # play
      ∈ plays_for_0strategy f initial›
    using interplay n1_def by (simp add: simple_game.plays_for_0strategy.p0move)
  hence ‹R p' q'› unfolding R_def by (meson list.sel(1))
  thus ‹∃q'. R p' q' ∧ q =▷a  q'› using q'_spec(2) by blast
next
  fix p q
  assume challenge:
    ‹R p q›
  hence game_move: ‹cs_game_moves (AttackerNode p q) (DefenderCouplingNode p q)› by auto
  have ‹(∃ play ∈ plays_for_0strategy f initial . hd play = AttackerNode p q)›
    using challenge assms by blast
  then obtain play where
      play_spec: ‹AttackerNode p q # play ∈ plays_for_0strategy f initial›
    by (metis list.sel(1) simple_game.plays.cases strategy0_plays_subset)
  hence interplay: ‹(DefenderCouplingNode p q) # AttackerNode p q # play
      ∈ plays_for_0strategy f initial›
    using game_move by (simp add: simple_game.plays_for_0strategy.p1move)
  hence ‹¬ player1_wins_immediately ((DefenderCouplingNode p q) # AttackerNode p q # play)›
    using assms(1) unfolding player0_winning_strategy_def by blast
  then obtain n1 where n1_def:
      ‹n1 = f (DefenderCouplingNode p q # AttackerNode p q # play)›
      ‹cs_game_moves (DefenderCouplingNode p q) n1›
    using interplay assms(2)
    unfolding player0_winning_strategy_def sound_0strategy_def by simp
  obtain q' where q'_spec:
      ‹(AttackerNode q' p) = n1› ‹q ⟼* tau q'›
    using n1_def(2) by (cases n1, auto)
  hence ‹(AttackerNode q' p) # (DefenderCouplingNode p q) # AttackerNode p q # play
      ∈ plays_for_0strategy f initial›
    using interplay n1_def by (simp add: simple_game.plays_for_0strategy.p0move)
  hence ‹R q' p› unfolding R_def by (meson list.sel(1))
  thus ‹∃q'. q ⟼* tau  q' ∧ R q' p› using q'_spec(2) by blast
qed

theorem winning_strategy_iff_coupledsim:
  assumes
    ‹initial = AttackerNode p q›
  shows
    ‹(∃ f . player0_winning_strategy f initial ∧ sound_0strategy f initial)
    = p ⊑cs q›
proof (rule)
  assume
    ‹(∃f. player0_winning_strategy f initial ∧ sound_0strategy f initial)›
  then obtain f where
    ‹coupled_delay_simulation (λp q. ∃play∈plays_for_0strategy f initial. hd play = AttackerNode
p q)›
    using winning_strategy_implies_coupleddsim by blast
```

```
  moreover have <(λp q. ∃play∈plays_for_0strategy f initial. hd play = AttackerNode p q)
p q>
    using assms plays_for_0strategy.init by force
  ultimately show <p ⊑cs q>
    unfolding coupled_sim_by_eq_coupled_delay_simulation
    by (metis (mono_tags, lifting))
next
  assume
    <p ⊑cs  q>
  thus <(∃f. player0_winning_strategy f initial ∧ sound_0strategy f initial)>
    unfolding coupled_sim_by_eq_coupled_delay_simulation
    using coupleddsim_implies_winning_strategy[OF _ _ assms]
          strategy_from_coupleddsim_sound[OF _ _ assms] by blast
qed

end
end
```

7 Fixed Point Algorithm for Coupled Similarity

7.1 The Algorithm

```
theory Coupledsim_Fixpoint_Algo_Delay
imports
  Coupled_Simulation
  "HOL-Library.While_Combinator"
  "HOL-Library.Finite_Lattice"
begin

context lts_tau
begin

definition fp_step ::
  <'s rel ⇒ 's rel>
where
  <fp_step R1 ≡ { (p,q)∈R1.
    (∀ p' a. p ⟼a p' ⟶
      (tau a ⟶ (p',q)∈R1) ∧
      (¬tau a ⟶ (∃ q'. ((p',q')∈R1) ∧ (q =▷a q')))) ∧
    (∃ q'. q ⟼*tau q' ∧ ((q',p)∈R1)) }>

definition fp_compute_cs :: <'s rel>
where <fp_compute_cs ≡ while (λR. fp_step R ≠ R) fp_step top>
```

7.2 Correctness

```
lemma mono_fp_step:
  <mono fp_step>
proof (rule, safe)
  fix x y::<'s rel> and p q
  assume
    <x ⊆ y>
    <(p, q) ∈ fp_step x>
  thus  <(p, q) ∈ fp_step y>
    unfolding fp_step_def
    by (auto, blast)
qed
```

```
lemma fp_fp_step:
  assumes
    <R = fp_step R>
  shows
    <coupled_delay_simulation (λ p q. (p, q) ∈ R)>
  using assms unfolding fp_step_def coupled_delay_simulation_def delay_simulation_def
  by (auto, blast, fastforce+)

lemma gfp_fp_step_subset_gcs:
  shows <(gfp fp_step) ⊆ { (p,q) . greatest_coupled_simulation p q }>
  unfolding  gcs_eq_coupled_sim_by[symmetric]
proof clarify
  fix a b
  assume
    <(a, b) ∈ gfp fp_step>
  thus <a ⊑cs  b>
    unfolding coupled_sim_by_eq_coupled_delay_simulation
    using fp_fp_step mono_fp_step gfp_unfold
    by metis
qed

lemma fp_fp_step_gcs:
  assumes
    <R = { (p,q) . greatest_coupled_simulation p q }>
  shows
    <fp_step R = R>
  unfolding assms
proof safe
  fix p q
  assume
    <(p, q) ∈ fp_step {(x, y). greatest_coupled_simulation x y}>
  hence
    <(∀p' a. p ⟼a  p' ⟶
      (tau a ⟶ greatest_coupled_simulation p' q) ∧
      (¬tau a ⟶  (∃q'. greatest_coupled_simulation p' q' ∧ q =▷a  q'))) ∧
    (∃q'. q ⟼* tau  q' ∧ greatest_coupled_simulation q' p)>
    unfolding fp_step_def by auto
  hence <(∀p' a. p ⟼a  p' ⟶ (∃q'. greatest_coupled_simulation p' q' ∧ q ⇒^a  q')) ∧
    (∃q'. q ⟼* tau  q' ∧ greatest_coupled_simulation q' p)>
    unfolding fp_step_def using weak_step_delay_implies_weak_tau steps.refl by blast
  hence <(∀p' a. p ⟼a  p' ⟶ (∃q'. greatest_coupled_simulation p' q' ∧ q ⇒^^a  q'))
∧
    (∃q'. q ⟼* tau  q' ∧ greatest_coupled_simulation q' p)>
    using weak_step_tau2_def by simp
  thus <greatest_coupled_simulation p q>
    using lts_tau.gcs by metis
next
  fix p q
  assume asm:
    <greatest_coupled_simulation p q>
  then have <(p, q) ∈ {(x, y). greatest_coupled_simulation x y}> by blast
  moreover from asm have <∃ R. R p q ∧ coupled_delay_simulation R>
    unfolding gcs_eq_coupled_sim_by[symmetric] coupled_sim_by_eq_coupled_delay_simulation.
  then obtain R where <R p q> <coupled_delay_simulation R> by blast
  moreover then have  <∀ p' a. p ⟼a p' ∧ ¬tau a ⟶
      (∃ q'. (greatest_coupled_simulation p' q') ∧ (q =▷a q'))>
```

```
    using coupled_delay_simulation_def delay_simulation_def
    by (metis coupled_similarity_implies_gcs coupled_simulation_weak_simulation
        delay_simulation_implies_weak_simulation)
  moreover from asm have  <∀ p' a. p ⟼a p' ∧ tau a ⟶ greatest_coupled_simulation p' q>
    unfolding  gcs_eq_coupled_sim_by[symmetric] coupled_sim_by_eq_coupled_delay_simulation
    by (metis coupled_delay_simulation_def delay_simulation_def)
  moreover have  <(∃ q'. q ⟼*tau q' ∧ (greatest_coupled_simulation q' p))>
    using asm gcs_is_coupled_simulation coupled_simulation_implies_coupling by blast
  ultimately show <(p, q) ∈ fp_step {(x, y). greatest_coupled_simulation x y}>
    unfolding fp_step_def by blast
qed

lemma gfp_fp_step_gcs: <gfp fp_step = { (p,q) . greatest_coupled_simulation p q }>
  using fp_fp_step_gcs gfp_fp_step_subset_gcs
  by (simp add: equalityI gfp_upperbound)

end

context lts_tau_finite
begin
lemma gfp_fp_step_while:
  shows
    <gfp fp_step = fp_compute_cs>
  unfolding fp_compute_cs_def
  using gfp_while_lattice[OF mono_fp_step] finite_state_rel Finite_Set.finite_set by blast

theorem coupled_sim_fp_step_while:
  shows <fp_compute_cs = { (p,q) . greatest_coupled_simulation p q }>
  using gfp_fp_step_while gfp_fp_step_gcs by blast

end

end
```

8 The Contrasimulation Preorder Word Game

```
theory Contrasim_Word_Game
imports
  Simple_Game
  Contrasimulation
begin

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

fun (in lts_tau) c_word_game_moves ::
  <('s, 'a) c_word_game_node ⇒ ('s, 'a) c_word_game_node ⇒ bool> where

  simulation_challenge:
   <c_word_game_moves (AttackerNode p q) (DefenderNode A p1 q0) =
     (p ⇒$A p1 ∧ q = q0 ∧ (∀a∈set A. a ≠ τ))> |

  simulation_answer:
    <c_word_game_moves (DefenderNode A p1 q0) (AttackerNode q1 p10) =
      (q0 ⇒$A q1 ∧ p1 = p10)> |
```

```
  c_word_game_moves_no_step:
    ‹c_word_game_moves _ _ = False›

fun c_word_game_defender_node :: ‹('s, 'a) c_word_game_node ⇒ bool› where
  ‹c_word_game_defender_node (AttackerNode _ _) = False› |
  ‹c_word_game_defender_node (DefenderNode _ _ _) = True›
```

8.1 Contrasimulation Implies Winning Strategy in Word Game (Completeness)

```
locale c_word_game =
  lts_tau trans τ +
  simple_game c_word_game_moves c_word_game_defender_node
for
  trans :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› and
  τ :: ‹'a› and
  initial :: ‹('s, 'a) c_word_game_node›
begin

fun strategy_from_contrasim:: ‹('s ⇒ 's ⇒ bool) ⇒ ('s, 'a) c_word_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 cwg_atknodes_precede_defnodes_in_plays:
  assumes
    ‹c_word_game_defender_node n0›
    ‹n0 = DefenderNode A p' q›
    ‹(n0#play) ∈ plays initial›
    ‹initial = AttackerNode p0 q0›
  shows ‹∃p. (hd play) = AttackerNode p q ∧ c_word_game_moves (hd play) n0›
proof -
  have ‹n0 ≠ initial› using assms (2, 4) by auto
  hence mov: ‹c_word_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_word_game_node.inject(2) c_word_game_defender_node.simps(1) c_word_game_moves.elims(2))
  thus ?thesis using mov by auto
qed

lemma cwg_second_play_elem_in_play_set :
  assumes
    ‹(n0#play) ∈ plays initial›
    ‹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 initial› 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 cwg_contrasim_contains_all_strat_consistent_atknodes:
  assumes
    <contrasimulation R>
    <R p0 q0>
    <initial = AttackerNode p0 q0>
    <play ∈ plays_for_0strategy (strategy_from_contrasim R) initial>
shows <((AttackerNode p q) ∈ set play) ⟹ R p q>
  using assms(4)
proof (induct arbitrary: p q rule: plays_for_0strategy.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.prems 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_word_game_node.inject(1)
          c_word_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_word_game.cwg_atknodes_precede_defnodes_in_plays
          simulation_challenge p0moved.hyps(1, 3) strategy0_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_word_game_moves (hd play) n0>
      using cwg_atknodes_precede_defnodes_in_plays strategy0_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_word_game_moves (hd play) n0> by auto
    have qpred_q_move: <qpred ⇒$A q> using qpred_def qpred_move n0_def by simp
    have <hd play ∈ set (n0 # play)>
      using cwg_second_play_elem_in_play_set strategy0_plays_subset[OF p0moved.hyps(1)] assms(3)
      by (auto simp add: n0_def)
    hence pred_R: <R qpred ppred>
      by (simp add: qpred_def p0moved.hyps(1) p0moved.hyps(2))
    have  <∃ p1 . R p1 q ∧ ppred ⇒$ A p1>
     using qpred_q_move pred_R assms(1) <∀a∈set A. a ≠ τ>
     unfolding contrasimulation_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_word_game_defender_node n1'>
     by (metis c_word_game_defender_node.simps(2) c_word_game_moves.elims(2) p1moved.hyps(4))
   hence <False> using A by auto
   thus <R p q> ..
 qed
qed

lemma contrasim_word_game_complete:
  assumes
    <contrasimulation R>
    <R p q>
    <initial = AttackerNode p q>
  shows <player0_winning_strategy (strategy_from_contrasim R) initial>
    unfolding player0_winning_strategy_def
proof (safe)
  fix play
  assume A1: <play ∈ plays_for_0strategy (strategy_from_contrasim R) initial>
  thus <player1_wins_immediately play ⟹ False>
    unfolding player1_wins_immediately_def
  proof -
    assume A: <c_word_game_defender_node (hd play) ∧ (∄p'. c_word_game_moves (hd play) p')>
    have player0_has_succ_node: <c_word_game_defender_node (hd play) ⟹
      ∃p'. c_word_game_moves (hd play) p'>
    proof (induct rule: simple_game.plays_for_0strategy.induct[OF A1])
      case init: 1
      from assms(3) have <¬c_word_game_defender_node (hd [initial])> by simp
      hence <False> using init.prems by simp
      then show ?case ..
    next
      case p0moved: (2 n0 play)
      from p0moved.hyps have <c_word_game_defender_node n0> by simp
      hence <∃A p1 q. n0 = (DefenderNode A p1 q)>
        by (meson c_word_game_defender_node.elims(2))
      hence <¬c_word_game_defender_node (strategy_from_contrasim R (n0#play))>
        using p0moved.hyps(4) c_word_game_moves.elims(2)
          [of <n0> <strategy_from_contrasim R (n0#play)>]
        by force
      hence <False> using p0moved.prems by simp
      then show ?case ..
    next
      case p1moved: (3 n1 play n1')
      hence <¬c_word_game_defender_node n1> using p1moved.hyps by simp
      then obtain p q where n1_def: <n1 = AttackerNode p q>
        using c_word_game_defender_node.elims(3) by auto
      hence pq_in_R: <R p q>
```

```
      using cwg_contrasim_contains_all_strat_consistent_atknodes[OF assms,
              of <n1#play>, OF p1moved.hyps(1)]
      by auto
    have is_def: <c_word_game_defender_node n1'> using p1moved.prems by auto
    then obtain A p1 q0 where n1'_def: <n1' = DefenderNode A p1 q0>
      using c_word_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_word_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 contrasimulation_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_word_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
```

8.2 Winning Strategy Implies Contrasimulation in Word Game (Soundness)

```
lemma cwg_strategy_from_contrasim_sound:
 assumes
   <R p0 q0>
   <contrasimulation R>
   <initial = AttackerNode p0 q0>
 shows
   <sound_0strategy (strategy_from_contrasim R) initial>
 unfolding sound_0strategy_def
proof (safe)
 fix n0 play
 assume A:
   <n0 # play ∈ plays_for_0strategy (strategy_from_contrasim R) initial>
   <c_word_game_defender_node n0>
 then obtain A p1 q where n0_def: <n0 = DefenderNode A p1 q>
   using c_word_game_defender_node.elims(2) by blast
 then obtain p where p_def: <hd play = AttackerNode p q>
   using n0_def A cwg_atknodes_precede_defnodes_in_plays assms(3) strategy0_plays_subset

   by blast
 hence <c_word_game_moves (AttackerNode p q) (DefenderNode A p1 q)>
   using A n0_def
   by (metis assms(3) cwg_atknodes_precede_defnodes_in_plays strategy0_plays_subset)
 hence mov_p_p1: <p ⇒$A p1> <∀a∈set A. a ≠ τ> by auto
 from p_def have <R p q>
   using cwg_contrasim_contains_all_strat_consistent_atknodes A assms
     cwg_second_play_elem_in_play_set n0_def strategy0_plays_subset
   by fastforce
 with mov_p_p1 have q1_def: <∃q1. R q1 p1 ∧ q ⇒$A q1>
   using assms(2) unfolding contrasimulation_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_word_game_moves n0 (strategy_from_contrasim R (n0 # play))>
    using q'_def by (simp add: n0_def)
qed

lemma contrasim_word_game_sound:
  assumes
    <player0_winning_strategy f initial>
    <sound_0strategy f initial>
  defines
    <R == λ p q . (∃ play ∈ plays_for_0strategy f initial. hd play = AttackerNode p q)>
  shows
    <contrasimulation R>  unfolding contrasimulation_def
proof (safe)
  fix p q p1 A
  assume A1: <p ⇒$A p1>
  assume A2: <R p q> <∀a∈set A. a ≠ τ>
  hence <∃ play . play ∈ plays_for_0strategy f initial ∧ hd play = AttackerNode p q>
    using R_def by auto
  from this obtain play where play_def:
    <play ∈ plays_for_0strategy f initial ∧ hd play = AttackerNode p q> ..
  from assms(1) have <¬player1_wins_immediately play>
    using player0_winning_strategy_def play_def by auto
  hence <(c_word_game_defender_node (hd play) ∧ (∄p'. c_word_game_moves (hd play) p'))
    ⟹ False>
    using player1_wins_immediately_def by auto
  hence Def_not_stuck:
    <c_word_game_defender_node (hd play) ⟹ (∄p'. c_word_game_moves (hd play) p')
    ⟹ False>
    by auto
  have <(p ⇒$A p1) ⟹ (((DefenderNode A p1 q)#play) ∈ plays_for_0strategy f initial)>
  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 strategy0_plays_subset no_empty_plays)
    hence inF:<(hd play)#(tl play) ∈ plays_for_0strategy f initial> by (simp add: play_def)
    have pl1: <player1_position (hd play)> by (simp add: play_def)
    have mov0:<c_word_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_0strategy f initial>
      using plays_for_0strategy.p1move[OF inF pl1 mov0] .
    thus <DefenderNode A p1 q#play ∈ plays_for_0strategy f initial>
      by (simp add: sym[OF play_split])
  qed
  hence def_in_f: <(DefenderNode A p1 q)#play ∈ plays_for_0strategy f initial>
    by (simp add: A1)
  hence <¬(player1_wins_immediately (DefenderNode A p1 q#play))>
    using assms(1) player0_winning_strategy_def by auto
  hence <∃n1. c_word_game_moves (DefenderNode A p1 q) n1>
```

```
    using player1_wins_immediately_def by auto
  have move_ex: <c_word_game_moves (DefenderNode A p1 q) (f (DefenderNode A p1 q # play))>
    using assms(2) def_in_f sound_0strategy_def by auto
  hence in_f: <f ((DefenderNode A p1 q) # play) # (DefenderNode A p1 q) # play
      ∈ plays_for_0strategy f initial>
    using plays_for_0strategy.p0move[OF def_in_f] by auto
  obtain n1 where
    n1_def: <n1 = f (DefenderNode A p1 q # play)> and
    n1_move: <c_word_game_moves (DefenderNode A p1 q) n1>
    using move_ex by auto
  hence <∃q1. n1 = (AttackerNode q1 p1)>
    using c_word_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_word_game_moves (DefenderNode A p1 q) (AttackerNode q1 p1)>
    using q1_def move_ex n1_def by auto
  hence q1_succ: <q ⇒$A q1> using c_word_game_moves.simps(2) by auto
  have def: <c_word_game_defender_node (DefenderNode A p1 q)> by simp
  hence <(AttackerNode q1 p1)#(DefenderNode A p1 q)#play ∈ plays_for_0strategy f initial>
    using q1_def n1_def in_f by auto
  then obtain R_play where
    R_play_def: <R_play = (AttackerNode q1 p1)#(DefenderNode A p1 q)#play> and
    R_play_in_f: <R_play ∈ plays_for_0strategy f initial> by simp
  hence <(hd R_play) = AttackerNode q1 p1> by (simp add: R_play_def)
  hence <R q1 p1> unfolding R_def using R_play_in_f by auto
  thus <R p q ⟹ p ⇒$ A  p1 ⟹ ∃q1. q ⇒$ A  q1 ∧ R q1 p1> using q1_succ by auto
qed

theorem winning_strategy_in_c_word_game_iff_contrasim:
  assumes
    <initial = AttackerNode p q>
  shows
    <(∃ f . player0_winning_strategy f initial ∧ sound_0strategy f initial)
    = (∃ C. contrasimulation C ∧ C p q)>
proof
  assume
    <(∃f. player0_winning_strategy f initial ∧ sound_0strategy f initial)>
  then obtain f where
    <contrasimulation (λp q.
      ∃play∈plays_for_0strategy f initial. hd play = AttackerNode p q)>
    using contrasim_word_game_sound by blast
  moreover have
    <(λp q. ∃play∈plays_for_0strategy f initial. hd play = AttackerNode p q) p q>
    using assms plays_for_0strategy.init[of _ f] by (meson list.sel(1))
  ultimately show <∃ C. contrasimulation C ∧ C p q> by blast
next
  assume
    <∃ C. contrasimulation C ∧ C p q>
  thus <(∃f. player0_winning_strategy f initial ∧ sound_0strategy f initial)>
    using contrasim_word_game_complete[OF _ _ assms]
        cwg_strategy_from_contrasim_sound[OF _ _ assms] by blast
qed

end
end
```

9 The Contrasimulation Preorder Set Game

```
theory Contrasim_Set_Game
imports
  Simple_Game
  Contrasimulation
begin

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>
```

9.1 Contrasimulation Implies Winning Strategy in Set Game (Completeness)

```
locale c_set_game =
  lts_tau trans τ +
  simple_game c_set_game_moves c_set_game_defender_node
for
  trans :: <'s ⇒ 'a ⇒ 's ⇒ bool> and
  τ :: <'a>
begin

fun strategy_from_mimicking_of_C ::
    <('s ⇒ ('s set) ⇒ bool) ⇒ ('s, 'a) c_set_game_node strategy>
  where

    <strategy_from_mimicking_of_C R ((DefenderSwapNode p1 Q)#play) =
      (AttackerNode (SOME q1 . (∃q. (q ∈ Q ∧ q  ⇒^τ q1)) ∧ R q1 {p1}) {p1})> |
```

```
    <strategy_from_mimicking_of_C R ((DefenderSimNode a p1 Q)#play) =
      (AttackerNode p1 (SOME Q1 . Q1 = dsuccs a Q ∧ R p1 Q1))> |

    <strategy_from_mimicking_of_C _ _ = undefined>

lemma csg_atknodes_precede_defnodes_in_plays:
  assumes
    <c_set_game_defender_node n0>
    <(n0#play) ∈ plays (AttackerNode p0 Q0)>
  shows <∃p Q. (hd play) = AttackerNode p Q ∧ c_set_game_moves (hd play) n0>
proof -
  have <n0 ≠ AttackerNode p0 Q0> using assms 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 csg_second_play_elem_in_play_set:
  assumes
    <(n0#play) ∈ plays (AttackerNode p0 Q0)>
    <c_set_game_defender_node n0>
  shows
    <hd play ∈ set (n0 # play)>
proof -
  from assms have <n0 ≠ AttackerNode p0 Q0> by auto
  hence <play ∈ plays (AttackerNode p0 Q0)>
    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 csg_only_defnodes_move_to_atknodes:
  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 c_set_game_strategy_retains_mimicking:
  assumes
    <contrasimulation C>
    <C p0 q0>
    <play ∈ plays_for_0strategy
      (strategy_from_mimicking_of_C (mimicking (set_lifted C))) (AttackerNode p0 {q0})>
  shows
    <n = AttackerNode p Q ⟹ n ∈ set play ⟹ mimicking (set_lifted C) p Q >
proof (induct arbitrary: n p Q rule: plays_for_0strategy.induct[OF assms(3)])
  case init: 1
  hence <p = p0 ∧ Q = {q0}> using init.prems(1) by auto
  thus <mimicking (set_lifted C) p Q>
    using assms R_is_in_mimicking_of_R set_lifted_def by simp
next
  case p0moved: (2 n0 play)
  hence <(n = strategy_from_mimicking_of_C
    (mimicking (set_lifted C)) (n0 # play)) ∨ (n ∈ set (n0#play))> by auto
  thus ?case
  proof(rule disjE)
    assume <n ∈ set (n0#play)>
    thus ?thesis using p0moved.prems p0moved.hyps(1,2) by blast
  next
    assume strat: <n = strategy_from_mimicking_of_C
      (mimicking (set_lifted C)) (n0 # play)>
    hence <(∃a Qpred. n0 = DefenderSimNode a p Qpred) ∨
        (∃q Ppred. n0 = DefenderSwapNode q Ppred ∧ Q = {q})>
      using csg_only_defnodes_move_to_atknodes[OF p0moved.hyps(4), of <p> <Q>]
        p0moved.prems(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_mimicking_of_C (mimicking (set_lifted C)) (n0#play)
          = AttackerNode p (SOME Q1. Q1 = dsuccs a Qpred ∧ (mimicking (set_lifted C)) p Q1)>
        using strategy_from_mimicking_of_C.simps(2) by auto
      hence Q_def: <Q = (SOME Q1. Q1 = dsuccs a Qpred ∧ (mimicking (set_lifted C)) p Q1)>
        using strat by (simp add: p0moved.prems(1))
      have <∃ppred. hd play = (AttackerNode ppred Qpred) ∧ c_set_game_moves (hd play) n0>
        using csg_atknodes_precede_defnodes_in_plays
              strategy0_plays_subset[OF p0moved.hyps(1)] assms(2,3) 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 csg_second_play_elem_in_play_set strategy0_plays_subset[OF p0moved.hyps(1)]
          assms(3) n0_def
        by (simp add: assms(3))
      hence <mimicking (set_lifted C) ppred Qpred>
        using p0moved.hyps(2) ppred_def by blast
      hence <mimicking (set_lifted C) p (dsuccs a Qpred)>
        using <ppred =▷a p> assms(1,2) mimicking_of_C_guarantees_action_succ <a ≠ τ>
        by auto
      hence <∃Q. Q = (dsuccs a Qpred) ∧ mimicking (set_lifted C) p Q> by auto
      from someI_ex[OF this] show <mimicking (set_lifted C) p Q>
        unfolding Q_def
        using n0_def p0moved.hyps(4) by auto
    next
      assume <(∃q Ppred. n0 = DefenderSwapNode q Ppred ∧ Q = {q})>
      then obtain q Ppred where
          n0_def: <n0 = DefenderSwapNode q Ppred> and
          Q_def: <Q = {q}>
        by auto
      hence <strategy_from_mimicking_of_C (mimicking (set_lifted C)) (n0#play)
         = AttackerNode (SOME p1.
            (∃p. p ∈ Ppred ∧ p ⇒^τ p1) ∧ (mimicking (set_lifted C)) p1 {q}) {q}>
        using strategy_from_mimicking_of_C.simps(1) by auto
      hence p_def: <p = (SOME p1.
            (∃p. p ∈ Ppred ∧ p ⇒^τ p1) ∧ (mimicking (set_lifted C)) p1 {q})>
        using strat p0moved.prems by auto
      have <∃qpred. hd play = (AttackerNode qpred Ppred) ∧ c_set_game_moves (hd play) n0>
        using csg_atknodes_precede_defnodes_in_plays
          strategy0_plays_subset[OF p0moved.hyps(1)] assms(3) 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
      have qpred_q_move: <qpred ⇒^τ q> using qpred_def qpred_move n0_def by simp
      have <hd play ∈ set (n0 # play)>
        using csg_second_play_elem_in_play_set strategy0_plays_subset[OF p0moved.hyps(1)]
          assms(3) n0_def
        by simp
      hence <mimicking (set_lifted C) qpred Ppred>
        using p0moved.hyps(2) qpred_def by blast
      hence <∃p. p ∈ weak_tau_succs Ppred ∧ mimicking (set_lifted C) p {q}>
        using qpred_q_move assms(1,2) mimicking_of_C_guarantees_tau_succ by blast
      hence <∃p. (∃p0. p0 ∈ Ppred ∧ p0 ⇒^τ  p) ∧ mimicking (set_lifted C) p {q}>
        using weak_tau_succs_def[of <Ppred>] by blast
      from someI_ex[OF this] p_def have <mimicking (set_lifted C) p {q}> by simp
      thus <mimicking (set_lifted C) p Q> using Q_def by blast
    qed
  qed
next
  case p1moved: (3 n1 play n1')
  hence <(n = n1') ∨ (n ∈ set (n1#play))> by auto
  thus ?case
  proof (rule disjE)
    assume <n ∈ set (n1#play)>
    thus ?case using p1moved.prems p1moved.hyps(1,2) by blast
```

```
  next
    assume A1: <n = n1'>
    hence <c_set_game_defender_node n1'>
      using csg_only_defnodes_move_to_atknodes p1moved.hyps(3, 4) p1moved.prems(1)
        by fastforce
    hence <False> using A1 p1moved.prems(1) by auto
    thus ?case by auto
  qed
qed

lemma contrasim_set_game_complete:
  assumes
    <contrasimulation C>
    <C p0 q0>
  shows
    <player0_winning_strategy (strategy_from_mimicking_of_C
      (mimicking (set_lifted C))) (AttackerNode p0 {q0})>
  unfolding player0_winning_strategy_def
proof (safe)
  fix play
  assume A1: <play ∈ (plays_for_0strategy
    (strategy_from_mimicking_of_C (mimicking (set_lifted C))) (AttackerNode p0 {q0}))>
  thus <player1_wins_immediately play ⟹ False>
    unfolding player1_wins_immediately_def
  proof clarify
    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_0strategy.induct[OF A1])
      case init: 1
      have <¬c_set_game_defender_node (AttackerNode p0 {q0})> by (simp add: assms)
      hence <False> using init.prems 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_mimicking_of_C
          (mimicking (set_lifted C)) (n0#play))>
        using p0moved.hyps(4)
          c_set_game_moves.elims(2)[of <n0>
            <strategy_from_mimicking_of_C (mimicking (set_lifted C)) (n0#play)>]
        by force
      hence <¬c_set_game_defender_node (hd (strategy_from_mimicking_of_C
          (mimicking (set_lifted C)) (n0 # play) # n0 # play))>
        by simp
      hence <False> using p0moved.prems ..
      then show ?case ..
    next
      case p1moved: (3 n1 play n1')
      hence <¬c_set_game_defender_node n1>
        using p1moved.hyps 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_mimicking: <mimicking (set_lifted C) p Q>
        using c_set_game_strategy_retains_mimicking[OF assms, of <n1#play>, OF p1moved.hyps(1)]
        by auto
      have <(∃a p1. n1' = DefenderSimNode a p1 Q) ∨ (∃p1. n1' = DefenderSwapNode p1 Q)>
        using p1moved.prems 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
        hence <p =▷a p1> by auto
        hence <p ⇒^a p1> using steps.refl by blast
        hence <mimicking (set_lifted C) p1 (dsuccs a Q)>
          using mimicking_of_C_guarantees_action_succ move
          by (metis in_mimicking assms(1) simulation_challenge tau_tau)
        then obtain Q1 where <Q1 = dsuccs a Q ∧ mimicking (set_lifted 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 p1moved.hyps(4) n1_def by auto
        hence p_succ: <p ⇒^τ p1> by auto
        hence <∃q'. q' ∈ weak_tau_succs Q ∧ mimicking (set_lifted C) q' {p1}>
          using in_mimicking mimicking_of_C_guarantees_tau_succ assms(1) by auto
        hence <∃q1. q1 ∈ weak_tau_succs Q ∧ mimicking (set_lifted 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 csg_strategy_from_mimicking_of_C_sound:
  assumes
    <contrasimulation C>
    <C p0 q0>
  shows
    <sound_0strategy
      (strategy_from_mimicking_of_C (mimicking (set_lifted C)))
      (AttackerNode p0 {q0})>
  unfolding sound_0strategy_def
proof (safe)
  fix n0 play
  assume A:
    <n0 # play ∈ plays_for_0strategy
      (strategy_from_mimicking_of_C (mimicking (set_lifted C))) (AttackerNode p0 {q0})>
    <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_mimicking_of_C (mimicking (set_lifted 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 csg_atknodes_precede_defnodes_in_plays simulation_challenge strategy0_plays_subset)
  hence <c_set_game_moves (AttackerNode p Q) (DefenderSimNode a p' Q)>
    by (metis A n0_def csg_atknodes_precede_defnodes_in_plays strategy0_plays_subset)
  hence <p =▷a p'> <¬ tau a> by auto
  hence <mimicking (set_lifted C) p Q>
    using c_set_game_strategy_retains_mimicking[OF assms] A p_def
      assms(2) csg_second_play_elem_in_play_set strategy0_plays_subset
    by fastforce
  hence  <mimicking (set_lifted C) p' (dsuccs a Q)>
    using mimicking_of_C_guarantees_action_succ <¬ tau a> <p =▷a p'> assms(1) tau_tau
    by blast
  hence Q1_ex: <∃Q'. Q' = dsuccs a Q ∧ mimicking (set_lifted C) p' Q'> by auto
  from n0_def have strat_succ:
    <strategy_from_mimicking_of_C (mimicking (set_lifted C)) (n0#play)
    = (AttackerNode p'
        (SOME Q1 . Q1 = dsuccs a Q ∧ (mimicking (set_lifted C)) p' Q1))>
    by auto
  then obtain Q1 where
    <AttackerNode p' (SOME Q1 . Q1 = dsuccs a Q ∧ (mimicking (set_lifted C)) p' Q1)
      = AttackerNode p' Q1>
    by blast
  hence Q1_def: <Q1 = (SOME Q1 . Q1 = dsuccs a Q ∧ (mimicking (set_lifted C)) p' Q1)>
    by auto
  have next_is_atk:
    <strategy_from_mimicking_of_C (mimicking (set_lifted C)) (n0#play)
    = (AttackerNode p' Q1)>
    using strat_succ Q1_def by auto
  with someI_ex[OF Q1_ex] Q1_def
    have mov_cond: <Q1 = dsuccs a Q ∧ mimicking (set_lifted 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 csg_atknodes_precede_defnodes_in_plays swap_challenge strategy0_plays_subset)

  hence <c_set_game_moves (AttackerNode p Q) (DefenderSwapNode p' Q)>
    by (metis A n0_def csg_atknodes_precede_defnodes_in_plays strategy0_plays_subset)
  hence <p ⇒^τ p'> by auto
  hence <mimicking (set_lifted C) p Q>
    using c_set_game_strategy_retains_mimicking[OF assms] A p_def
      csg_second_play_elem_in_play_set strategy0_plays_subset
    by fastforce
  hence  <∃q'. q' ∈ weak_tau_succs Q ∧ mimicking (set_lifted C) q' {p'}>
    using mimicking_of_C_guarantees_tau_succ <p ⇒^τ p'> assms(1) by auto
  hence q1_ex: <∃q1. (∃q.(q ∈ Q ∧ q  ⇒^τ q1)) ∧ mimicking (set_lifted C) q1 {p'}>
    using weak_tau_succs_def by auto
```

```
    hence strat: <strategy_from_mimicking_of_C (mimicking (set_lifted C)) (n0#play)
      = AttackerNode (SOME q1.
        (∃q. (q ∈ Q ∧ q  ⇒^τ q1)) ∧ (mimicking (set_lifted C)) q1 {p'}) {p'}>
      using n0_def by auto
    then obtain q1 where
      <AttackerNode (SOME q1.
        (∃q. (q ∈ Q ∧ q  ⇒^τ q1)) ∧ (mimicking (set_lifted C)) q1 {p'}) {p'}
      = AttackerNode q1 {p'}> by blast
    hence q1_def:
      <q1 = (SOME q1 . (∃q. (q ∈ Q ∧ q  ⇒^τ q1)) ∧ (mimicking (set_lifted C)) q1 {p'})>
      by auto
    with someI_ex[OF q1_ex] have
      <∃q. (q ∈ Q ∧ q  ⇒^τ q1) ∧ mimicking (set_lifted 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
```

9.2 Winning Strategy Implies Contrasimulation in Set Game (Soundness)

```
lemma csg_move_defsimnode_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, 3) c_set_game_moves_no_step(1, 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 csg_move_defswapnode_to_atknode:
  assumes
    <c_set_game_moves (DefenderSwapNode p' Q) n0>
  shows
    <∃q'. n0 = AttackerNode q' {p'} ∧ q' ∈ weak_tau_succs Q>
proof -
  have <¬c_set_game_defender_node n0>
    using assms c_set_game_moves_no_step(3, 4) c_set_game_defender_node.elims(2)
    by metis
  hence <∃q1 P1. n0 = AttackerNode q1 P1>
    by (meson c_set_game_defender_node.elims(3))
  then obtain q1 P1 where n0_def: <n0 = AttackerNode q1 P1> by auto
  hence <P1 = {p'}> using assms local.swap_answer by blast
  from n0_def have <q1 ∈  weak_tau_succs Q> using assms by auto
  thus ?thesis using  <P1 = {p'}> n0_def by simp
qed

lemma csg_defsimnode_never_stuck:
  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 csg_defender_can_simulate_prefix:
  assumes
    <A ≠ []>
    <p ⇒$A p1>
    <∀a∈set A. a ≠ τ>
    <sound_0strategy f (AttackerNode p00 {q00})>
    <play ∈ plays_for_0strategy f (AttackerNode p00 {q00})>
    <hd play = AttackerNode p {q}>
  shows
    <∃play p0.
      ((DefenderSimNode (last A) p0 (dsuccs_seq_rec (rev (butlast A)) {q}))#play)
        ∈ plays_for_0strategy f (AttackerNode p00 {q00})
      ∧ word_reachable_via_delay A p p0 p1>
  using assms(1-3)
proof (induct arbitrary: p1 rule: rev_nonempty_induct[OF assms(1)])
  case single: (1 a)
  hence <¬tau a> using <∀a∈set A. a ≠ τ> by (simp add: tau_def)
  hence <p ⇒$[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_0strategy f (AttackerNode p00 {q00})>
    using assms(5,6)
    by (metis c_set_game_defender_node.simps(1) list.collapse no_empty_plays
          plays_for_0strategy.p1move strategy0_plays_subset)
  hence inplay:
    <(DefenderSimNode (last [a]) p0 (dsuccs_seq_rec (rev (butlast [a])) {q}))#play
      ∈ plays_for_0strategy f (AttackerNode p00 {q00})>
    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' ⇒$[a]  p1>
    using rev_seq_split[OF snoc.prems(2)] by blast
  hence <∃p'. p ⇒$ 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_0strategy f (AttackerNode p00 {q00})
    ∧ 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_0strategy f (AttackerNode p00 {q00})>
      <word_reachable_via_delay as p p0 p'>
    by auto
  hence
    <DefenderSimNode a2 p0 (dsuccs_seq_rec (rev as2) {q}) # play
      ∈ plays_for_0strategy f (AttackerNode p00 {q00})>
    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_0strategy f (AttackerNode p00 {q00})>
    by auto
  then obtain n1 where
    n1_def: <c_set_game_moves (DefenderSimNode a2 p0 (dsuccs_seq_rec (rev as2) {q})) n1>
    using csg_defsimnode_never_stuck by meson
  hence n1_atk: <n1 = AttackerNode p0 (dsuccs a2 ((dsuccs_seq_rec (rev as2) {q})))>
    using csg_move_defsimnode_to_atknode[OF n1_def] by auto
  have n1_in_play: <n1#n0#play ∈ plays_for_0strategy f (AttackerNode p00 {q00})>
    using n1_def n0_in_play n0_def
    by (metis assms(4) csg_move_defsimnode_to_atknode c_set_game_defender_node.simps(2)
        plays_for_0strategy.simps sound_0strategy_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_0strategy f (AttackerNode p00 {q00})>
    using n1_in_play
    by (simp add: n1_atk plays_for_0strategy.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 contrasim_set_game_sound:
  assumes
    <player0_winning_strategy f (AttackerNode p00 {q00})>
    <sound_0strategy f (AttackerNode p00 {q00})>
  defines
    <C == λ p q . (∃ play ∈ plays_for_0strategy f (AttackerNode p00 {q00}) .
      hd play = AttackerNode p {q} ∧ (hd play = (AttackerNode p00 {q00})
      ∨ (∃P. hd (tl play) = DefenderSwapNode q P)))>
  shows
    <contrasimulation C>
  unfolding contrasimulation_def
proof (safe)
  fix p q p1 A
  assume <∀a∈set A. a ≠ τ> <p ⇒$A p1> <C p q>
```

```
hence <p ⇒$(taufree A) p1> by (simp add: weak_step_over_tau)
hence <∃ play ∈ plays_for_0strategy f (AttackerNode p00 {q00}).
    hd play = AttackerNode p {q}
    ∧ (hd play = (AttackerNode p00 {q00})
    ∨ (∃P. hd (tl play) = DefenderSwapNode q P))>
  using C_def <p ⇒$A p1> <C p q> by auto
from this obtain play where
  play_def: <play ∈ plays_for_0strategy f (AttackerNode p00 {q00})> and
  play_hd: <hd play = AttackerNode p {q}> and
  <hd play = (AttackerNode p00 {q00}) ∨ (∃P. hd (tl play) = DefenderSwapNode q P)>
  by auto
hence <¬player1_wins_immediately 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_immediately_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 ⇒$A p1> by auto
  hence <∃n0. n0 = DefenderSwapNode p1 {q}
    ∧ c_set_game_moves (AttackerNode p {q}) n0> by simp
  from this obtain n0 where n0_def: <n0 = DefenderSwapNode p1 {q}>
    and n0_move: <c_set_game_moves (AttackerNode p {q}) n0> by auto
  have <play = (hd play)#(tl play)>
    by (metis hd_Cons_tl no_empty_plays play_def strategy0_plays_subset)
  hence <n0#play ∈ plays_for_0strategy f (AttackerNode p00 {q00})>
    using n0_def n0_move play_def play_hd
    by (metis c_set_game_defender_node.simps(1) play_def
        plays_for_0strategy.p1move)
  hence <∃n1'. c_set_game_moves n0 n1'
      ∧ n1'#n0#play ∈ plays_for_0strategy f (AttackerNode p00 {q00})>
    using assms(2) n0_def sound_0strategy_def
    by (meson c_set_game_defender_node.simps(3) plays_for_0strategy.p0move)
  then obtain n1' where n1'_mov: <c_set_game_moves n0 n1'>
    and in_play: <n1'#n0#play ∈ plays_for_0strategy f (AttackerNode p00 {q00})> by auto
  hence <∃q1. n1' = AttackerNode q1 {p1} ∧ (q1 ∈ weak_tau_succs {q})>
    by (metis c_set_game_defender_node.elims(2, 3) c_set_game_moves_no_step(3, 4)
        swap_answer n0_def)
  then obtain q1 where q1_def: <n1' = AttackerNode q1 {p1}>
    and q_succ: <q1 ∈ weak_tau_succs {q}> by auto
  hence q_tau: <q ⇒^τ q1> using weak_tau_succs_def by auto
  from in_play q1_def n0_def have <C q1 p1> unfolding C_def by force
  then show ?thesis using q_tau Nil by auto
next
  case (snoc as a)
  hence <A ≠ []> by auto
  hence <¬tau a> using <∀a∈set A. a ≠ τ> snoc  by (simp add: tau_def)
  then obtain A_play p0 where gotoA:
    <DefenderSimNode (last A) p0 (dsuccs_seq_rec (rev (butlast A)) {q}) # A_play
      ∈ plays_for_0strategy f (AttackerNode p00 {q00})>
    <word_reachable_via_delay A p p0 p1>
    using csg_defender_can_simulate_prefix <p ⇒$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_0strategy f (AttackerNode p00 {q00})>
      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_0strategy_def)
    hence <n1 = AttackerNode p0 (dsuccs a (dsuccs_seq_rec (rev as) {q}))>
      using csg_move_defsimnode_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_0strategy f (AttackerNode p00 {q00})>
      using n0_def A_play_def n1_move assms(2) csg_move_defsimnode_to_atknode
        plays_for_0strategy.p0move sound_0strategy_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_0strategy f (AttackerNode p00 {q00})>
      using n1_in_play by (simp add: n1_def plays_for_0strategy.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_0strategy_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 swap_answer
      by (metis c_set_game_defender_node.cases c_set_game_moves_no_step(3, 7))
    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_0strategy f (AttackerNode p00 {q00})>
      using in_strat plays_for_0strategy.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_c_set_game_iff_contrasim:
  shows
    <(∃ f . player0_winning_strategy f (AttackerNode p0 {q0})
```

```
      ∧ sound_0strategy f (AttackerNode p0 {q0}))
    = p0 ⊑c q0>
proof safe
  fix f
  assume
    <player0_winning_strategy f (AttackerNode p0 {q0})>
    <sound_0strategy f (AttackerNode p0 {q0})>
  hence <contrasimulation (λp q. ∃play ∈ plays_for_0strategy f (AttackerNode p0 {q0}).
    hd play = AttackerNode p {q} ∧
    (hd play = (AttackerNode p0 {q0}) ∨ (∃P. hd (tl play) = DefenderSwapNode q P)))>
    using contrasim_set_game_sound by blast
  thus <p0 ⊑c q0>
    using plays_for_0strategy.init[of <AttackerNode p0 {q0}> f] list.sel(1) by force
next
  fix C
  assume <contrasimulation C> <C p0 q0>
  thus
    <(∃f. player0_winning_strategy f (AttackerNode p0 {q0})
      ∧ sound_0strategy f (AttackerNode p0 {q0}))>
    using contrasim_set_game_complete[OF _ _]
      csg_strategy_from_mimicking_of_C_sound[OF _ _]
    by blast
qed

end
end
```

10 Infinitary Hennessy–Milner Logic

```
theory HM_Logic_Infinitary
  imports
    Weak_Relations
begin

datatype ('a,'x)HML_formula =
  HML_true
| HML_conj <'x set> <'x ⇒ ('a,'x)HML_formula>  (<AND _ _>)
| HML_neg <('a,'x)HML_formula>                  (<~_> [20] 60)
| HML_poss <'a> <('a,'x)HML_formula>            (<⟨_⟩_> [60] 60)
```

— The HML formulation is derived from that by Max Pohlmann `cite pohlmann2021reactivebisim`.

10.1 Satisfaction Relation

```
context lts_tau
begin

function satisfies :: <'s ⇒ ('a, 's) HML_formula ⇒ bool>
  (<_ ⊨ _> [50, 50] 50)
  where
    <(p ⊨ HML_true) = True>
  | <(p ⊨ HML_conj I F) = (∀ i ∈ I. p ⊨ (F i))>
  | <(p ⊨ HML_neg φ) = (¬ p ⊨ φ)>
  | <(p ⊨ HML_poss α φ) =
      (∃ p'. ((tau α ∧ p ⟼* tau p') ∨ (¬ tau α ∧ p ⟼α p')) ∧ p' ⊨ φ)>
  using HML_formula.exhaust by (auto, blast)
```

```
inductive_set HML_wf_rel :: <('s × ('a, 's) HML_formula) rel>
  where
    <φ = F i ∧ i ∈ I ⟹ ((p, φ), (p, HML_conj I F)) ∈ HML_wf_rel>
  | <((p, φ), (p, HML_neg φ)) ∈ HML_wf_rel>
  | <((p, φ), (p', HML_poss α φ)) ∈ HML_wf_rel>

lemma HML_wf_rel_is_wf: <wf HML_wf_rel>
  unfolding wf_def
proof (safe)
  fix P::<'s × ('a, 's) HML_formula ⇒ bool> and t::<'s × ('a, 's) HML_formula>
  obtain p φ where <t = (p, φ)> by force
  assume <∀x. (∀y. (y, x) ∈ HML_wf_rel ⟶ P y) ⟶ P x>
  hence <P (p, φ)>
  proof (induct φ arbitrary: p)
    case HML_true
    then show ?case
      by (metis HML_formula.distinct(1,3,5) HML_wf_rel.cases old.prod.exhaust)
  next
    case (HML_conj I F)
    thus ?case
      by (smt (verit) HML_formula.distinct(7,9) HML_formula.inject(1) HML_wf_rel.cases
         case_prodD case_prodE' lts_tau.HML_wf_rel_def mem_Collect_eq range_eqI)
  next
    case (HML_neg φ)
    thus ?case
      by (metis HML_formula.distinct(11,7) HML_formula.inject(2) HML_wf_rel.cases surj_pair)
  next
    case (HML_poss a φ)
    thus ?case
      by (smt (verit, del_insts) HML_formula.distinct(3,5,9,11)  HML_formula.inject(3)
        HML_wf_rel.cases case_prodD case_prodE' HML_wf_rel_def mem_Collect_eq)
  qed
  thus <P t> using <t = (p, φ)> by simp
qed

termination satisfies using HML_wf_rel_is_wf
  by (standard, (simp add: HML_wf_rel.intros)+)

inductive_set HML_direct_subformulas :: <(('a, 's) HML_formula) rel>
  where
    <φ = F i ∧ i ∈ I ⟹ (φ, HML_conj I F) ∈ HML_direct_subformulas>
  | <(φ, HML_neg φ) ∈ HML_direct_subformulas>
  | <(φ, HML_poss α φ) ∈ HML_direct_subformulas>

lemma HML_direct_subformulas_wf: <wf HML_direct_subformulas>
  unfolding wf_def
proof (safe)
  fix P x
  assume <∀x. (∀y. (y, x) ∈ HML_direct_subformulas ⟶ P y) ⟶ P x>
  thus <P x>
  proof induct
    case HML_true
    then show ?case using HML_direct_subformulas.simps by blast
  next
    case (HML_conj I F)
    then show ?case
      by (metis HML_direct_subformulas.cases HML_formula.distinct(7,9)
```

```
          HML_formula.inject(1) range_eqI)
  next
    case (HML_neg φ)
    then show ?case
      by (metis HML_direct_subformulas.simps HML_formula.distinct(11,7) HML_formula.inject(2))
  next
    case (HML_poss a φ)
    then show ?case
      by (metis HML_direct_subformulas.simps HML_formula.distinct(11,9) HML_formula.inject(3))
  qed
qed

definition HML_subformulas where <HML_subformulas ≡ (HML_direct_subformulas)⁺>

lemma HML_subformulas_wf: <wf HML_subformulas>
  using HML_direct_subformulas_wf HML_subformulas_def wf_trancl
  by fastforce

lemma conj_only_depends_on_indexset:
  assumes <∀i∈I. f1 i = f2 i>
  shows <(p ⊨ HML_conj I f1) = (p ⊨ HML_conj I f2)>
  using assms by auto
```

10.2 Distinguishing Formulas

```
definition HML_equivalent :: <'s ⇒ 's ⇒ bool>
  where <HML_equivalent p q
    ≡ (∀ φ::('a, 's) HML_formula. (p ⊨ φ) ⟷ (q ⊨ φ))>

fun distinguishes ::  <('a,'s) HML_formula ⇒ 's ⇒ 's ⇒ bool>
  where
   <distinguishes φ p q = (p ⊨ φ ∧ ¬ q ⊨ φ)>

fun distinguishes_from_set ::  <('a,'s) HML_formula ⇒ 's ⇒ 's set ⇒ bool>
  where
   <distinguishes_from_set φ p Q = (p ⊨ φ ∧ (∀q. q ∈ Q ⟶ ¬ q ⊨ φ))>

lemma distinguishing_formula:
  assumes <¬ HML_equivalent p q>
  shows <∃ φ. p ⊨ φ ∧ ¬ q ⊨ φ>
  using assms  satisfies.simps(3) unfolding HML_equivalent_def
  by blast

lemma HML_equivalent_symm:
  assumes <HML_equivalent p q>
  shows <HML_equivalent q p>
  using HML_equivalent_def assms by presburger
```

10.3 Weak-NOR Hennessy–Milner Logic

```
definition HML_weaknor ::
  <'x set ⇒ ('x ⇒ ('a,'x)HML_formula) ⇒ ('a,'x)HML_formula>
  where <HML_weaknor I F = HML_poss τ (HML_conj I (λf. HML_neg (F f)))>

definition HML_weaknot ::
  <('a,'x)HML_formula ⇒ ('a,'x)HML_formula>
  where <HML_weaknot φ = HML_weaknor {undefined} (λi. φ)>
```

```
inductive_set HML_weak_formulas :: <('a,'x)HML_formula set> where
  Base: <HML_true ∈ HML_weak_formulas> |
  Obs: <φ ∈ HML_weak_formulas ⟹ (⟨τ⟩⟨a⟩φ) ∈ HML_weak_formulas> |
  Conj: <(⋀i. i ∈ I ⟹ F i ∈ HML_weak_formulas) ⟹ HML_weaknor I F ∈ HML_weak_formulas>

lemma weak_backwards_truth:
  assumes
    <φ ∈ HML_weak_formulas>
    <p ⟼* tau  p'>
    <p' ⊨ φ>
  shows
    <p ⊨ φ>
  using assms
proof cases
  case Base
  then show ?thesis by force
next
  case (Obs φ a)
  then show ?thesis
    using assms(2,3) satisfies.simps(4) steps_concat tau_tau by blast
next
  case (Conj I F)
  then show ?thesis
    unfolding HML_weaknor_def tau_def
    using tau_tau assms steps_concat
    by force
qed

lemma tau_a_obs_implies_delay_step:
  assumes <p  ⊨ ⟨τ⟩⟨a⟩φ>
  shows <∃p'. p =▷a p' ∧ p' ⊨ φ>
proof -
  obtain p'' where <p ⇒^τ p'' ∧ p'' ⊨ ⟨a⟩φ> using assms by auto
  thus ?thesis using satisfies.simps(4) steps_concat tau_tau by blast
qed

lemma delay_step_implies_tau_a_obs:
  assumes
    <p =▷a p'>
    <p' ⊨ φ>
  shows <p  ⊨ ⟨τ⟩⟨a⟩φ>
proof -
  obtain p'' where <p ⇒^τ p''> and <p'' ⇒^a p'>
    using assms steps.refl tau_tau by blast
  thus ?thesis
    by (metis assms(1,2) lts_tau.satisfies.simps(4) lts_tau.tau_tau)
qed

end
end
```

11 Weak HML and the Contrasimulation Set Game

```
theory Weak_HML_Contrasimulation
  imports
```

```
    Contrasim_Set_Game
    HM_Logic_Infinitary
begin
```

11.1 Distinguishing Formulas at Winning Attacker Positions

```
locale c_game_with_attacker_strategy  =
  c_set_game trans τ
for
  trans :: <'s ⇒ 'a ⇒ 's ⇒ bool> and
  τ :: <'a> +
fixes
  strat :: <('s, 'a) c_set_game_node posstrategy> and
  attacker_winning_region :: <('s, 'a) c_set_game_node set> and
  attacker_order
defines
  <attacker_order ≡ {(g', g). c_set_game_moves g g' ∧
      g ∈ attacker_winning_region ∧ g' ∈ attacker_winning_region ∧
      (player1_position g ⟶ g' = strat g)}^+>
assumes
  finite_win:
    <wf attacker_order> and
  strat_stays_winning:
    <g ∈ attacker_winning_region ⟹ player1_position g ⟹
      strat g ∈ attacker_winning_region ∧ c_set_game_moves g (strat g)> and
  defender_keeps_losing:
    <g ∈ attacker_winning_region ⟹ c_set_game_defender_node g ⟹ c_set_game_moves g g'
    ⟹ g' ∈ attacker_winning_region>
begin
```

This construction of attacker formulas from a game only works if `strat` is a well-founded attacker strategy. (If it's winning and sound, the constructed formula should be distinguishing.)

```
function attack_formula :: <('s, 'a) c_set_game_node ⇒ ('a,'s) HML_formula> where
  <attack_formula (AttackerNode p Q) =
    (if (AttackerNode p Q) ∈ attacker_winning_region
      then attack_formula (strat (AttackerNode p Q))
      else HML_true)>
| <attack_formula (DefenderSimNode a p Q) =
    (if (DefenderSimNode a p Q) ∈ attacker_winning_region
      then ⟨τ⟩⟨a⟩(attack_formula (AttackerNode p (dsuccs a Q)))
      else HML_true)>
| <attack_formula (DefenderSwapNode p Q) =
    (if Q = {} ∨ DefenderSwapNode p Q ∉ attacker_winning_region
      then HML_true
      else (HML_weaknor (weak_tau_succs Q)
        (λq. if q ∈ (weak_tau_succs Q)
              then (attack_formula (AttackerNode q {p}))
              else HML_true )))>
  using c_set_game_defender_node.cases
  by (auto, blast)

termination attack_formula
  using finite_win
proof (standard, safe)
  fix p Q
  assume <AttackerNode p Q ∈ attacker_winning_region>
  thus <(strat (AttackerNode p Q), AttackerNode p Q) ∈ attacker_order>
```

```
    unfolding attacker_order_def
    using strat_stays_winning
    by auto
next
  fix a p Q
  assume attacker_wins: <DefenderSimNode a p Q ∈ attacker_winning_region>
  hence <AttackerNode p (dsuccs a Q) ∈ attacker_winning_region>
    using defender_keeps_losing simulation_answer by force
  with attacker_wins show
    <(AttackerNode p (dsuccs a Q), DefenderSimNode a p Q) ∈ attacker_order>
    unfolding attacker_order_def
    by (simp add: r_into_trancl')
next
  fix p Q q' q
  assume case_assms:
    <q' ∈ weak_tau_succs Q>
    <(AttackerNode q' {p}, DefenderSwapNode p Q) ∉ attacker_order>
    <DefenderSwapNode p Q ∈ attacker_winning_region>
    <q ∈ Q>
  hence <AttackerNode q' {p} ∉ attacker_winning_region>
    unfolding attacker_order_def by auto
  moreover from case_assms have <AttackerNode q' {p} ∈ attacker_winning_region>
    using swap_answer defender_keeps_losing by force
  ultimately show <q ∈ {}> by blast
qed

lemma attacker_defender_switch:
  assumes
    <(AttackerNode p Q) ∈ attacker_winning_region>
  shows
    <(∃a p'. (strat (AttackerNode p Q)) = (DefenderSimNode a p' Q) ∧ p  =▷a p' ∧ ¬tau a)
    ∨(∃p'. (strat (AttackerNode p Q)) = (DefenderSwapNode p' Q) ∧ p ⟼* tau p' )>
  using strat_stays_winning[OF assms] by (cases <strat (AttackerNode p Q)>, auto)

lemma attack_options:
  assumes
    <(AttackerNode p Q) ∈ attacker_winning_region>
  shows
    <(∃a p'. p =▷a p' ∧ ¬tau a ∧ strat (AttackerNode p Q) = (DefenderSimNode a p' Q) ∧
      attack_formula (AttackerNode p Q)
      = ⟨τ⟩⟨a⟩(attack_formula (AttackerNode p' (dsuccs a Q))))
    ∨ (∃p'. p ⟼* tau p' ∧ strat (AttackerNode p Q) = (DefenderSwapNode p' Q) ∧
      attack_formula (AttackerNode p Q) =
      (HML_weaknor (weak_tau_succs Q) (λq.
        if q ∈ (weak_tau_succs Q)
          then (attack_formula (AttackerNode q {p'}))
          else HML_true )))
    ∨ (Q = {} ∧ attack_formula (AttackerNode p Q) = HML_true)>
proof -
  from assms have
    <attack_formula (AttackerNode p Q) = attack_formula (strat (AttackerNode p Q))>
    by simp
  moreover from attacker_defender_switch assms have
    <(∃a p'. (strat (AttackerNode p Q)) = (DefenderSimNode a p' Q) ∧ p  =▷a p' ∧ ¬tau a)
    ∨ (∃p'. (strat (AttackerNode p Q)) = (DefenderSwapNode p' Q) ∧ p ⟼* tau p')>
    by blast
  ultimately have
```

```
    < (∃a p'. (strat (AttackerNode p Q)) = (DefenderSimNode a p' Q) ∧
      (attack_formula (AttackerNode p Q))
       = attack_formula (DefenderSimNode a p' Q) ∧ p  =▷a p' ∧ ¬tau a)
    ∨ (∃p'. (strat (AttackerNode p Q)) = (DefenderSwapNode p' Q) ∧
      (attack_formula (AttackerNode p Q))
       = attack_formula (DefenderSwapNode p' Q) ∧ p ⟼* tau p')>
    by metis
  moreover from assms have <strat (AttackerNode p Q) ∈ attacker_winning_region>
    by (simp add: strat_stays_winning)
  ultimately show ?thesis using assms empty_iff
    by fastforce
qed

lemma distinction_soundness:
  fixes p Q p0 Q0
  defines
    <pQ == AttackerNode p Q>
  defines
    <φ == attack_formula pQ>
  assumes
    <pQ ∈ attacker_winning_region>
  shows
    <p ⊨ φ ∧ (∀q∈Q. ¬ q ⊨ φ)>
  using finite_win assms
proof (induct arbitrary: p Q φ)
  case (less p Q)
  from attack_options[OF this(2)] show ?case
  proof
    assume <∃a p'. p =▷ a  p' ∧ ¬ tau a ∧
      strat (AttackerNode p Q) = DefenderSimNode a p' Q ∧
      attack_formula (AttackerNode p Q) = ⟨τ⟩⟨a⟩attack_formula (AttackerNode p' (dsuccs a Q))>
    then obtain a p' where case_assms:
      <p =▷ a  p' ∧ ¬ tau a>
      <strat (AttackerNode p Q) = DefenderSimNode a p' Q>
      <attack_formula (AttackerNode p Q)
      = ⟨τ⟩⟨a⟩attack_formula (AttackerNode p' (dsuccs a Q))> by blast
    hence moves:
      <c_set_game_moves (AttackerNode p Q) (DefenderSimNode a p' Q)>
      <c_set_game_moves (DefenderSimNode a p' Q) (AttackerNode p' (dsuccs a Q))> by auto
    with case_assms less(2) defender_keeps_losing strat_stays_winning have all_winning:
      <(AttackerNode p' (dsuccs a Q)) ∈ attacker_winning_region>
      <(DefenderSimNode a p' Q) ∈ attacker_winning_region>
        by (metis c_set_game_defender_node.simps(2), force)
    with case_assms moves less(2) have
      <(AttackerNode p' (dsuccs a Q), DefenderSimNode a p' Q) ∈ attacker_order>
      <(DefenderSimNode a p' Q, AttackerNode p Q) ∈ attacker_order>
      unfolding attacker_order_def by (simp add: r_into_trancl')+
    hence <(AttackerNode p' (dsuccs a Q), AttackerNode p Q) ∈ attacker_order>
      unfolding attacker_order_def by auto
    with less.hyps all_winning(1) have
      <p' ⊨ attack_formula (AttackerNode p' (dsuccs a Q)) ∧
      (∀q∈(dsuccs a Q). ¬ q ⊨ attack_formula (AttackerNode p' (dsuccs a Q)))>
      by blast
    with case_assms have
      <p ⊨ ⟨τ⟩⟨a⟩attack_formula (AttackerNode p' (dsuccs a Q))>
      <∀q∈Q. ¬q ⊨ ⟨τ⟩⟨a⟩attack_formula (AttackerNode p' (dsuccs a Q))>
      unfolding dsuccs_def by (auto, blast+)
```

```
  thus ?case unfolding case_assms by blast
next
  assume <(∃p'. p ⟼* tau  p' ∧ strat (AttackerNode p Q) = DefenderSwapNode p' Q ∧
      attack_formula (AttackerNode p Q)
      = HML_weaknor (weak_tau_succs Q) (λq.
        if q ∈ (weak_tau_succs Q)
          then attack_formula (AttackerNode q {p'})
          else HML_true)) ∨
    Q = {} ∧ attack_formula (AttackerNode p Q) = HML_true>
  thus ?case
  proof
    assume <∃p'. p ⟼* tau  p' ∧ strat (AttackerNode p Q) = DefenderSwapNode p' Q ∧
      attack_formula (AttackerNode p Q)
      = HML_weaknor (weak_tau_succs Q) (λq.
        if q ∈ (weak_tau_succs Q)
          then attack_formula (AttackerNode q {p'})
          else HML_true)>
    then obtain p' where case_assms:
      <p ⟼* tau  p'>
      <strat (AttackerNode p Q) = DefenderSwapNode p' Q>
      <attack_formula (AttackerNode p Q)
      = HML_weaknor (weak_tau_succs Q) (λq.
        if q ∈ (weak_tau_succs Q)
        then attack_formula (AttackerNode q {p'})
        else HML_true)>
      by blast
    from case_assms have moves:
      <c_set_game_moves (AttackerNode p Q) (DefenderSwapNode p' Q)>
      <∀q'∈(weak_tau_succs Q).
        c_set_game_moves (DefenderSwapNode p' Q) (AttackerNode q' {p'})>
      by auto
    with case_assms less(2) defender_keeps_losing strat_stays_winning
      have all_winning:
        <(DefenderSwapNode p' Q) ∈ attacker_winning_region>
        <∀q'∈(weak_tau_succs Q). (AttackerNode q' {p'}) ∈ attacker_winning_region>
      by (metis, metis c_set_game_defender_node.simps(1,3))
    with case_assms moves less(2) have
      <∀q'∈ weak_tau_succs Q. (AttackerNode q' {p'}, DefenderSwapNode p' Q) ∈ attacker_order>
      <(DefenderSwapNode p' Q, AttackerNode p Q) ∈ attacker_order>
      unfolding attacker_order_def by (simp add: r_into_trancl')+
    hence <∀q'∈ weak_tau_succs Q. (AttackerNode q' {p'}, AttackerNode p Q) ∈ attacker_order>
      unfolding attacker_order_def by auto
    with less.hyps all_winning have
      <∀q'∈ weak_tau_succs Q.
        q' ⊨ attack_formula (AttackerNode q' {p'}) ∧
        ¬ p' ⊨ attack_formula (AttackerNode q' {p'})>
      by blast
    with case_assms have
      <p' ⊨ HML_conj (weak_tau_succs Q)
        (λq'. HML_neg (attack_formula (AttackerNode q' {p'})))>
      <∀q'∈ weak_tau_succs Q.
        ¬ q' ⊨ HML_conj (weak_tau_succs Q)
        (λqq'. HML_neg (attack_formula (AttackerNode qq' {p'})))>
      by (simp, fastforce)
    with case_assms have
      <p ⊨ HML_weaknor (weak_tau_succs Q)
        (λq. if q ∈ (weak_tau_succs Q)
```

```
            then attack_formula (AttackerNode q {p'})
            else HML_true)>
        <∀q∈Q. ¬q ⊨  HML_weaknor (weak_tau_succs Q)
          (λq. if q ∈ (weak_tau_succs Q)
            then attack_formula (AttackerNode q {p'})
            else HML_true)>
        unfolding weak_tau_succs_def HML_weaknor_def
        using conj_only_depends_on_indexset by (auto, force, fastforce)
      thus ?case unfolding case_assms by blast
    next
      assume <Q = {} ∧ attack_formula (AttackerNode p Q) = HML_true>
      thus ?case by auto
    qed
  qed
qed

lemma distinction_in_language:
  fixes p Q
  defines
    <pQ == AttackerNode p Q>
  defines
    <φ == attack_formula pQ>
  assumes
    <pQ ∈ attacker_winning_region>
  shows
    <φ ∈ HML_weak_formulas>
  using assms(2,3) unfolding assms(1)
proof (induct arbitrary: φ rule: attack_formula.induct)
  case (1 p Q)
  then show ?case using strat_stays_winning by auto
next
  case (2 a p Q)
  then show ?case
    by (simp add: HML_weak_formulas.Base HML_weak_formulas.Obs)
next
  case (3 p Q)
  hence <∀q' ∈ weak_tau_succs Q. attack_formula (AttackerNode q' {p}) ∈ HML_weak_formulas>
    using weak_tau_succs_def HML_weak_formulas.Base by fastforce
  then show ?case
    using HML_weak_formulas.Base <DefenderSwapNode p Q ∈ attacker_winning_region>
    by (auto simp add: HML_weak_formulas.Conj)
qed

end
```

11.2 Attacker Wins on Pairs with Distinguishing Formulas

```
locale c_game_with_attacker_formula  =
  c_set_game trans τ
for
  trans :: <'s ⇒ 'a ⇒ 's ⇒ bool> and
  τ :: <'a>
begin

inductive_set attacker_winning_region :: <('s, 'a) c_set_game_node set> where
  Base: <DefenderSwapNode _ {} ∈ attacker_winning_region> |
  Atk: <(c_set_game_moves (AttackerNode p Q) g' ∧ g' ∈ attacker_winning_region)
```

```
    ⟹ (AttackerNode p Q) ∈ attacker_winning_region> |
  Def: <c_set_game_defender_node g ⟹
    (⋀g'. c_set_game_moves g g' ⟹ g' ∈ attacker_winning_region)
    ⟹ g ∈ attacker_winning_region>

lemma attacker_wins_if_defender_set_empty:
  assumes
    <Q = {}>
  shows
    <AttackerNode p Q ∈ attacker_winning_region>
proof -
  have atk_move: <c_set_game_moves (AttackerNode p Q) (DefenderSwapNode p Q)>
    by (simp add: steps.refl)
  have <DefenderSwapNode p Q ∈ attacker_winning_region>
    using assms attacker_winning_region.Base by simp
  thus ?thesis using atk_move attacker_winning_region.Atk by blast
qed

lemma attacker_wr_propagation:
  assumes
    <AttackerNode p' (dsuccs a Q) ∈ attacker_winning_region>
    <p =▷a p'>
    <¬tau a>
  shows
    <AttackerNode p Q ∈ attacker_winning_region>
proof -
  have AtkToSim: <c_set_game_moves (AttackerNode p Q) (DefenderSimNode a p' Q)>
    using assms(2, 3) by simp
  have <∀g. c_set_game_moves
        (DefenderSimNode a p' Q) g
        ⟶ (g = AttackerNode p' (dsuccs a Q))>
    by (simp add: csg_move_defsimnode_to_atknode)
  hence <(DefenderSimNode a p' Q) ∈ attacker_winning_region>
    using assms(1) attacker_winning_region.Def
    by (metis c_set_game_defender_node.simps(2))
  thus ?thesis using AtkToSim attacker_winning_region.Atk by blast
qed

lemma distinction_completeness:
  assumes
    <φ ∈ HML_weak_formulas>
    <distinguishes_from_set φ p Q>
  shows
    <(AttackerNode p Q) ∈ attacker_winning_region>
proof (cases <Q = {}>)
  case True
  then show ?thesis using attacker_wins_if_defender_set_empty by auto
next
  case False
  then show ?thesis using assms
  proof (induct arbitrary: p Q rule: HML_weak_formulas.induct[OF assms(1)])
    case Base: 1
    have <∀q. q ⊨ HML_true> by simp
    hence <False>
      using Base.prems(1, 3) by simp
    then show ?case by auto
  next
```

```
    case Obs: (2 φ a)
    then obtain p' where p'_def: <p =▷a p' ∧ p' ⊨ φ >
      using tau_a_obs_implies_delay_step[of p a φ] by auto
    have <∀q. q ∈ Q ⟶ ¬ q ⊨ ⟨τ⟩⟨a⟩φ> using Obs by auto
    hence <∀q. q ∈ Q ⟶ (∀q'.  ¬q  =▷a  q' ∨ ¬q' ⊨ φ)>
      using delay_step_implies_tau_a_obs by blast
    hence <∀q'. q' ∈ dsuccs a Q ⟶ ¬ q' ⊨ φ>
      unfolding dsuccs_def by blast
    hence phi_distinguishing: <distinguishes_from_set φ p' (dsuccs a Q)>
      using p'_def by simp
    thus ?case
    proof (cases <dsuccs a Q = {}>)
      case dsuccs_empty: True
      then show ?thesis
      proof (cases <tau a>)
        case True
        hence <{q1. ∃q∈ Q. q ⟼* tau q1} = {}> using dsuccs_def dsuccs_empty by auto
        hence <Q = {}> using steps.refl by blast
        then show ?thesis using attacker_wins_if_defender_set_empty by auto
      next
        case False
        hence <AttackerNode p' (dsuccs a Q) ∈ attacker_winning_region>
          using attacker_wins_if_defender_set_empty dsuccs_empty by auto
        thus ?thesis using attacker_wr_propagation False p'_def by blast
      qed
    next
      case False
      hence wr_pred_atk_node:
        <AttackerNode p' (dsuccs a Q) ∈ attacker_winning_region>
        using Obs.hyps phi_distinguishing
        by auto
      thus ?thesis
      proof(cases <tau a>)
        case True
        hence <∀p. (p ⊨ ⟨τ⟩⟨a⟩φ) = (p  ⊨ φ)>
          using delay_step_implies_tau_a_obs p'_def satisfies.simps(4) tau_tau
            Obs.hyps(1) weak_backwards_truth
          by (meson lts.refl)
        hence <distinguishes_from_set φ p Q> using Obs.prems by auto
        thus ?thesis using Obs.hyps Obs.prems by blast
      next
        case False
        then show ?thesis
          using wr_pred_atk_node attacker_wr_propagation p'_def
          by blast
      qed
    qed
  next
    case Conj: (3 I F)
    then obtain p' where <p ⇒^τ p'> and p_sat:  <p' ⊨ HML_conj I (λf. HML_neg (F f))>
      unfolding HML_weaknor_def by auto
    have <⋀q . q ∈ Q  ⟹ ¬q  ⊨  HML_poss τ (HML_conj I (λf. HML_neg (F f)))>
      by (metis Conj.prems(3) HML_weaknor_def distinguishes_from_set.elims(2))
    hence <⋀q q'. q ∈ Q ⟹ ¬q ⇒^τ q' ∨ ¬q'  ⊨ HML_conj I (λf. HML_neg (F f))>
      using satisfies.simps(4) tau_tau by blast
    hence <⋀q'. ¬q' ∈ (weak_tau_succs Q) ∨ ¬q'  ⊨ HML_conj I (λf. HML_neg (F f))>
      using weak_tau_succs_def by auto
```

```
    hence Ex: <⋀q'.  q' ∈ (weak_tau_succs Q) ⟹ (∃i. i ∈ I ∧ q'  ⊨  (F i))>
      by auto
    have atk_move: <c_set_game_moves (AttackerNode p Q) (DefenderSwapNode p' Q)>
      using <p ⇒^τ p'> by auto
    have Ex_i:
      <∀q1 P1. c_set_game_moves (DefenderSwapNode p' Q) (AttackerNode q1 P1) ⟶
        (∃i. i ∈ I ∧ q1  ⊨  (F i)) ∧ P1 = {p'}>
      using Ex by auto
    hence <∀q1 P1.
      c_set_game_moves (DefenderSwapNode p' Q) (AttackerNode q1 P1)
      ⟶ (∃i. i ∈ I ∧ q1  ⊨  (F i) ∧ (∀p'. p' ∈ P1 ⟶ ¬ p' ⊨ (F i)))>
      using p_sat by auto
    hence  <∀q1 P1.
      c_set_game_moves (DefenderSwapNode p' Q) (AttackerNode q1 P1)
      ⟶ (∃i. i ∈ I ∧ distinguishes_from_set (F i) q1 P1)>
      unfolding distinguishes_from_set.simps using p_sat by blast
    hence all_atk_succs_in_wr:
      <∀q1 P1. c_set_game_moves (DefenderSwapNode p' Q) (AttackerNode q1 P1)
        ⟶ (AttackerNode q1 P1 ∈ attacker_winning_region)>
      using Conj.hyps Ex_i by blast
    hence <∀g. c_set_game_moves (DefenderSwapNode p' Q) g
        ⟶ (∃ q1 P1. g = (AttackerNode q1 P1))>
      using csg_move_defswapnode_to_atknode by blast
    hence <∀g. c_set_game_moves (DefenderSwapNode p' Q) g
        ⟶ g ∈ attacker_winning_region>
      using all_atk_succs_in_wr by auto
    hence <DefenderSwapNode p' Q ∈ attacker_winning_region>
      using attacker_winning_region.Def
      by (meson c_set_game_defender_node.simps(3))
    then show ?case using atk_move attacker_winning_region.Atk by blast
  qed
qed

end
end
```

12 Reductions and τ-sinks

Checking trace inclusion can be reduced to contrasimulation checking, as can weak simulation checking to coupled simulation checking. The trick is to add a τ-sink to the transition system, that is, a state that is reachable by τ-steps from every other state, and cannot be left. An illustration of such an extension is given in Figure 2. Intuitively, the extension means that the model is allowed to just stop progressing at any point.

We here prove that, on systems with a τ-sink, weak similarity equals coupled similarity and weak trace inclusion equals contrasimilarity. We also prove that adding a τ-sink to a system does not change weak similarity nor weak trace inclusion relationships within the system. As adding the τ-sink only has negligible effect on the system sizes, these facts establish the reducibility relationships.

```
theory Tau_Sinks
imports
  Coupled_Simulation
begin
```

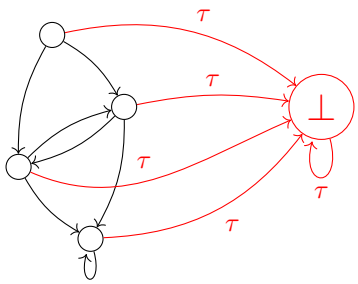

Figure 2: Example of a τ-sink extension with the original transition system in black and the extension in red.

12.1 τ-Sink Properties

```
context lts_tau
begin

definition tau_sink ::
  <'s ⇒ bool>
where
  <tau_sink p ≡
    (∀a p'. p ⟼a p' ⟶ a = τ ∧ p = p') ∧
    (∀p0. p0 ⟼τ p)>
```

The tau sink is a supremum for the weak transition relation.

```
lemma tau_sink_maximal:
  assumes <tau_sink sink>
  shows
    <tau_max sink>
    <(p ⟼* tau sink)>
  using assms steps_loop step_weak_step_tau tau_tau unfolding tau_sink_def by metis+

lemma sink_has_no_word_transitions:
  assumes
    <tau_sink sink>
    <A ≠ []>
    <∀ a ∈ set(A). a ≠ τ>
  shows <∄s'. sink ⇒$A s'>
proof -
  obtain a where <∃B. A = a#B> using assms(2) list.exhaust_sel by auto
  hence <∄s' . sink ⇒^a s'>
    by (metis assms(1,3) list.set_intros(1) lts_tau.tau_def steps_loop tau_sink_def)
  thus ?thesis using <∃B. A = a#B> by fastforce
qed
```

12.2 Contrasimulation Equals Weak Simulation on τ-Sink Systems

```
lemma sink_coupled_simulates_all_states:
  assumes
    <⋀ p . (p ⟼* tau sink)>
  shows
    <sink ⊑cs p>
  by (simp add: assms coupledsim_refl coupledsim_step)

theorem coupledsim_weaksim_equiv_on_sink_expansion:
```

```
  assumes
    <⋀ p . (p ⟼* tau sink)>
  shows
    <p ⊑ws q ⟷ p ⊑cs q>
  using assms
  using coupled_simulation_weak_simulation weak_sim_tau_step weaksim_greatest by auto
```

12.3 Contrasimulation Equals Weak Trace Inclusion on τ-Sink Systems

```
lemma sink_contrasimulates_all_states:
fixes A :: " 'a list"
  assumes
    <tau_sink sink>
    <⋀ p . (p ⟼* tau sink)>
  shows
    <∀ p. sink ⊑c p>
proof (cases A)
  case Nil
  hence empty_word: <sink ⇒$A sink> by (simp add: steps.refl)
  have <∀p. p ⇒$A sink> using assms(2) Nil by auto
  have <sink ⊑c sink> using contrasim_tau_step empty_word Nil by auto
  show ?thesis using assms(2) contrasim_tau_step by auto
next
  case Cons
  hence <∄s'. (∀ a ∈ set(A). a ≠ τ) ∧ sink ⇒$A s'>
    using assms(1) sink_has_no_word_transitions by fastforce
  show ?thesis using assms(2) contrasim_tau_step by auto
qed

lemma sink_trace_includes_all_states:
  assumes
    <⋀ p . (p ⟼* tau sink)>
  shows
    <sink ⊑T p>
  by (metis assms contrasim_tau_step lts_tau.contrasim_implies_trace_incl)

lemma trace_incl_with_sink_is_contrasim:
  assumes
    <⋀ p . (p ⟼* tau sink)>
    <⋀ p . R sink p>
    <trace_inclusion R>
  shows
    <contrasimulation R>
 unfolding contrasimulation_def
proof clarify
  fix p q p' A
  assume <R p q> <p ⇒$A  p'> <∀ a ∈ set(A). a ≠ τ>
  hence <∃q'. q ⇒$A  q'>
    using assms(3) unfolding trace_inclusion_def by blast
  hence <q ⇒$A  sink>
    using assms(1) tau_tau word_tau_concat by blast
  thus <∃q'. q ⇒$ A  q' ∧ R q' p'>
    using assms(2) by auto
qed

theorem contrasim_trace_incl_equiv_on_sink_expansion_R:
  assumes
```

```
    <⋀ p . (p ⟼* tau sink)>
    <⋀ p . R sink p>
  shows
    <contrasimulation R = trace_inclusion R>
proof
  assume <contrasimulation R>
  thus <trace_inclusion R> by (simp add: contrasim_implies_trace_incl)
next
  assume <trace_inclusion R>
  thus <contrasimulation R>  by (meson assms lts_tau.trace_incl_with_sink_is_contrasim)
qed

theorem contrasim_trace_incl_equiv_on_sink_expansion:
  assumes
    <⋀ p . (p ⟼* tau sink)>
  shows
    <p ⊑T q ⟷ p ⊑c q>
  using assms weak_trace_inlcusion_greatest
    contrasim_tau_step contrasim_trace_incl_equiv_on_sink_expansion_R contrasim_implies_trace_incl
  by (metis (no_types, lifting))

end
```

12.4 Weak Simulation Invariant under τ-Sink Extension

```
lemma simulation_tau_sink_1:
  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>
    <lts_tau.weak_simulation step τ R>
  shows
    <lts_tau.weak_simulation step2 τ (λ p q. p = sink ∨ R p q)>
proof -
  let ?tau = <(lts_tau.tau τ)>
  let ?tauEx = <τ>
  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 τ (λ p q. p = sink ∨ R p q)>
proof -
  let ?tau = <(lts_tau.tau τ)>
  let ?tauEx = <τ>
  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 <∃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'')> ..
next
  fix p q p' a
  assume <R p q> <step p a p'>
  hence not_sink: <p ≠ sink> <p' ≠ sink>
    using assms(2)[of a p] assms(2)[of a p'] by auto
  have <step2 p a p'> using `step p a p'` unfolding step2_def by blast
  then obtain q' where q'_def:
    <p' = sink ∨ R p' q'>
    <?tau a ⟶ ?steps step2 q ?tau q'>
    <¬ ?tau a ⟶ (∃pq1 pq2. ?steps step2 q ?tau pq1 ∧ step2 pq1 a pq2
      ∧ ?steps step2 pq2 ?tau q')>
    using assms(3) `R p q` unfolding lts_tau.weak_simulation_def by blast
  hence outer_goal_a: <R p' q'> using not_sink by blast
  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'))>
  proof (cases <q' = sink>)
    assume <q' = sink>
    then obtain q'' where q''_def:
      <?tau a ⟶ (?steps step q ?tau q'' ∧ ?steps step2 q'' ?tau q')>
      <¬ ?tau a ⟶ (∃pq1. ?steps step2 q ?tau pq1 ∧ step pq1 a q''
        ∧ ?steps step2 q'' ?tau q')>
      using q'_def(2,3) assms(1) step2_def lts_tau.step_tau_refl[of τ]
        lts_tau.tau_tau[of τ] by metis
    hence <q'' = sink ⟶ q = sink>
      using assms(2) unfolding step2_def by (metis lts.steps.cases)
    have <?steps step2 q'' ?tau q'> using q''_def by auto
    hence <p' = sink ∨ R p' q''> using  q'_def(1) assms(4)[of p' q' q''] by blast
    moreover have <¬ ?tau a ⟶ (∃pq1 pq2. ?steps step q ?tau pq1 ∧ step pq1 a pq2
      ∧ ?steps step pq2 ?tau q'')>
    proof
      assume <¬ ?tau a>
      hence <q ≠ sink> using q'_def by (metis assms(2) lts.steps_left step2_def)
      hence <q'' ≠ sink> using `q'' = sink ⟶ q = sink` by simp
      obtain pq1 where pq1_def:
          <?steps step2 q ?tau pq1> <step pq1 a q''> <?steps step2 q'' ?tau q'>
        using q''_def(2) `¬ ?tau a` by blast
      hence <pq1 ≠ sink> using `q'' ≠ sink` assms(2) by blast
      hence <?steps step q ?tau pq1> using `q ≠ sink` `?steps 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 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>
  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 using lts_tau.weak_sim_tau_step by fastforce
  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
```

12.5 Trace Inclusion Invariant under τ-Sink Extension

```
lemma trace_inclusion_sink_invariant:
  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>
  shows
    <lts_tau.weakly_trace_included_by step2 τ p q
    = lts_tau.weakly_trace_included_by step τ p q>
proof -
  let ?tau = <(lts_tau.tau τ)>
  let ?weak_step = <lts_tau.weak_step_tau step τ>
  let ?weak_step2 = <lts_tau.weak_step_tau step2 τ>
  let ?weak_seq = <lts_tau.weak_step_seq step τ>
  let ?weak_seq2 = <lts_tau.weak_step_seq step2 τ>
  {
    fix A
    have <∀p p'. (∀ a ∈ set(A). a ≠ τ)
    ∧ ?weak_seq2 p A p'
    ⟶ (∃p''. ?weak_seq p A p''
    ∧ ?weak_step2 p'' τ p')>
    proof(clarify, induct A rule: rev_induct)
      case Nil
      hence <?weak_step p τ p>
        using lts_tau.step_tau_refl by fastforce
      thus ?case
        by (metis Nil.prems(2) lts_tau.tau_tau lts_tau.weak_step_seq.simps(1))
    next
      case (snoc a A)
      hence not_in_set: <∀a∈set A. a ≠ τ> by force
      then obtain p01 where
        <?weak_seq2 p A p01> and
        p01_def2: <?weak_step2 p01 a p'>
        using snoc by (metis lts_tau.rev_seq_split)
      then obtain p02 where p02_def:
        <?weak_seq p A p02>
        <?weak_step2 p02 τ p01>
        using snoc.hyps[of p p01] snoc.prems(1) not_in_set by auto
      hence <?weak_step2 p02 a p'>
        using p01_def2 lts_tau.step_tau_concat lts_tau.tau_tau
        by (smt (verit, del_insts))
      then obtain p03 p04 where
        tau1: <?weak_step2 p02 τ p03> and
        a_step2: <step2 p03 a p04> and
        tau2: <?weak_step2 p04 τ p'>
        using snoc.prems(1) lts_tau.tau_def
        by (metis last_in_set snoc_eq_iff_butlast)
```

```
    hence <p04 ≠ sink> using assms snoc.prems(1) by force
    hence a_step: <step p03 a p04> using a_step2 assms by auto
    have notsinkp03: <p03 ≠ sink> using a_step2 assms snoc.prems(1) by force
    have <lts.steps step2 p02 ?tau p03> using tau1 by (simp add: lts_tau.tau_tau)
    hence <lts.steps step p02 ?tau p03> using notsinkp03
    proof (induct rule: lts.steps.induct[OF ‘lts.steps step2 p02 ?tau  p03‘])
      case (1 p A)
      thus ?case by (simp add: lts.refl)
    next
      case (2 p A q1 a q)
      hence <q1 ≠ sink> using assms(2) step2_def by blast
      thus ?case using 2 lts.step step2_def by metis
    qed
    hence <?weak_step p02 τ p03> by (simp add: lts_tau.tau_tau)
    hence <?weak_step p02 a p04> using a_step
      by (metis lts.step lts_tau.step_tau_refl lts_tau.tau_tau)
    hence <?weak_seq p (A@[a]) p04>
      using lts_tau.rev_seq_step_concat p02_def(1) by fastforce
    thus ?case using tau2 by auto
  qed
}
hence step2_to_step: <∀A p p'. (∀ a ∈ set(A). a ≠ τ)
  ∧ ?weak_seq2 p A p'
  ⟶ (∃p''. ?weak_seq p A p'')>
  by fastforce

have step_to_step2:  <∀A p p'. (∀ a ∈ set(A). a ≠ τ)
  ∧ ?weak_seq p A p'
  ⟶ ?weak_seq2 p A p'>
proof
  fix A
  show <∀p p'. (∀ a ∈ set(A). a ≠ τ)
  ∧ ?weak_seq p A p'
  ⟶ ?weak_seq2 p A p'>
  proof(clarify, induct A rule: list.induct)
    case Nil
    assume <?weak_seq p [] p'>
    hence tau_step: <?weak_step p τ p'>
      by (simp add: lts_tau.weak_step_seq.simps(1) lts_tau.tau_tau)
    hence <?weak_step2 p τ p'>
      using lts_impl_steps step2_def lts_tau.tau_tau  by force
    thus ?case by (simp add: lts_tau.weak_step_seq.simps(1) lts_tau.tau_tau)
  next
    case (Cons x xs)
    then obtain p1 where p1_def: <?weak_step p x p1>
    <?weak_seq p1 xs p'>
      by (metis (mono_tags, lifting) lts_tau.weak_step_seq.simps(2))
    hence IH: <?weak_seq2 p1 xs p'> using Cons by auto
    then obtain p01 p02 where <?weak_step p τ p01> and
      strong: <step p01 x p02> and
      p02_weak: <?weak_step p02 τ p1>
      using Cons.prems(1) p1_def lts_tau.tau_def by (metis list.set_intros(1))
    hence tau1: <?weak_step2 p τ p01>
      using lts_impl_steps step2_def
      by (smt (verit, best))
    have <?weak_step2 p02 τ p1>
      using p02_weak lts_impl_steps step2_def by (smt (verit, best))
```

```
      hence <?weak_step2 p x p1>
        using tau1 strong step2_def Cons.prems(1) lts_tau.tau_def
        by (metis list.set_intros(1))
      thus <?weak_seq2 p (x#xs) p'>
        using IH lts_tau.weak_step_seq_def[of step2 τ] by auto
    qed
  qed
  show ?thesis
  proof (rule)
    assume <∃R. lts_tau.trace_inclusion step2 τ R ∧ R p q>
    then obtain R where weak_sim_R: <lts_tau.trace_inclusion step2 τ R>
      and Rpq: <R p q>
      by blast
    have <lts_tau.trace_inclusion step τ R>
      unfolding lts_tau.trace_inclusion_def
    proof clarify
      fix p q p' A
      assume subassms:
        <∀a∈set A. a ≠ τ>
        <R p q>
        <?weak_seq p A p'>
      hence <(∀a∈set A. a ≠ τ) ∧
       ?weak_seq2 p A p' ⟶
       (∃q'. ?weak_seq2 q A q')>
        using weak_sim_R
        unfolding lts_tau.trace_inclusion_def by simp
      hence <(∀a∈set A. a ≠ τ) ∧
       ?weak_seq p A p' ⟶
       (∃q'. ?weak_seq q A q')>
        using step2_to_step step_to_step2
        by auto
      thus "∃q'. ?weak_seq q A q'"
        by (simp add: subassms)
    qed
    thus <∃R. lts_tau.trace_inclusion step τ R ∧ R p q>
      using Rpq by auto
  next
    assume <∃R. lts_tau.trace_inclusion step τ R ∧ R p q>
    then obtain R where weak_sim_R: <lts_tau.trace_inclusion step τ R>
      and Rpq: <R p q>
      by blast
    have <lts_tau.trace_inclusion step2 τ R>
      unfolding lts_tau.trace_inclusion_def
    proof clarify
      fix p q p' A
      assume subassms:
        <∀a∈set A. a ≠ τ>
        <R p q>
        <?weak_seq2 p A p'>
      thus <∃q'. ?weak_seq2 q A q'>
        using step2_to_step step_to_step2
        by (metis (full_types) lts_tau.trace_inclusion_def weak_sim_R)
    qed
    thus <∃R. lts_tau.trace_inclusion step2 τ R ∧ R p q> using Rpq by auto
  qed
qed
```

```
end
```

References

[1] Bisping, B.: Computing Coupled Similarity. Master's thesis, Technische Universität Berlin (2018), https://coupledsim.bbisping.de/bisping_computingCoupledSimilarity_thesis.pdf

[2] Bisping, B., Jansen, D.N.: Linear-time–branching-time spectroscopy accounting for silent steps (2023). https://doi.org/s10.48550/arXiv.2305.17671

[3] Bisping, B., Montanari, L.: A game characterization for contrasimilarity. In: Dardha, O., Castiglioni, V. (eds.) Proceedings Combined 28th International Workshop on Expressiveness in Concurrency and 18th Workshop on Structural Operational Semantics, Electronic Proceedings in Theoretical Computer Science, vol. 339, pp. 27–42. Open Publishing Association (2021). https://doi.org/10.4204/EPTCS.339.5

[4] Bisping, B., Nestmann, U.: Computing coupled similarity. In: Proceedings of TACAS. pp. 244–261. LNCS, Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_14

[5] Bisping, B., Nestmann, U., Peters, K.: Coupled similarity: the first 32 years. Acta Informatica **57**(3–5), 439–463 (2020). https://doi.org/10.1007/s00236-019-00356-4

[6] van Glabbeek, R.J.: The linear time–branching time spectrum II. In: International Conference on Concurrency Theory. pp. 66–81. Springer (1993)