Coupled Similarity
and How to Compute It

Benjamin Bisping*

February 15, 2019

Abstract

This theory surveys a range of definitions of *coupled similarity* from the literature, and proves properties relevant for algorithms computing coupled similarity relations.

Coupled similarity is a notion of equivalence for systems with internal actions. It has 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.

We show how the preexisting definitions coincide and that they can be reformulated using *coupled delay simulations*. Our key contribution is to verify a polynomial-time coinductive fixed-point algorithm computing the coupled simulation preorder and to characterize the coupled simulation preorder by a simple game. Our proofs also support the conclusion that computing coupled similarity is at least as complex as computing weak similarity.

Contents

*Technische Universität Berlin, Germany, benjamin.bisping@tu-berlin.de.

1 Introduction

This theory accompanies Benjamin Bisping and Uwe Nestmann's TACAS 2019 paper [2] and Benjamin Bisping's master's thesis "Computing Coupled Similarity" [1], which can be found on https://coupledsim.bbisping.de/bisping_computingCoupledSimilarity_thesis.pdf.

2 Preliminaries

2.1 Some Utilities for Finite Partial Orders

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

```
theory Finite_Partial_Order
  imports Main
begin

context preorder
begin

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

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

```
qed

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

context order — that is: partial order
begin

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

end

end
```

2.2 Labeled Transition Systems

```
theory Transition_Systems
  imports Finite_Partial_Order
begin

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

begin

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

```
abbreviation step ::
  ‹'s ⇒ 'a ⇒ 's ⇒ bool›
   ("_ ⟼_  _" [70, 70, 70] 80)
where
  ‹p ⟼a  q ≡ trans p a q›

inductive steps :: ‹'s ⇒ ('a ⇒ bool) ⇒ 's ⇒ bool›
     ("_ ⟼* _  _" [70, 70, 70] 80)
where
  refl: ‹p ⟼* A p› | step: ‹p ⟼* A q1 ⟹ q1 ⟼a q ⟹ A a ⟹ (p ⟼* A q)›

lemma steps_one_step:
  assumes
    ‹p ⟼a p'›
     ‹A a›
   shows
    ‹p ⟼* A p'› using steps.step[of p A p a p'] steps.refl[of p A] assms .

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

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


lemma steps_no_step:
  assumes
    ‹⋀ a p' . p ⟼a p' ⟹ ¬A a›
     ‹p ≠ p''›
     ‹p ⟼* A p''›
   shows
    ‹False›
   using steps_left[OF assms(2,3)] assms(1) by blast

lemma steps_no_step_pos:
  assumes
    ‹⋀ a p' . p ⟼a p' ⟹ ¬A a›
     ‹p ⟼* A p'›
```

```
   shows
    ⟨p = p'⟩
   using assms steps_no_step by blast

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


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

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

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

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

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

end — end of lts

lemma lts_impl_steps2:
  assumes
    ⟨lts.steps step1 p1 ap p2⟩
```

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

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

end
```

2.3 Transition Systems with Silent Steps

```
theory Weak_Transition_Systems
  imports Transition_Systems
begin

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

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

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

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

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

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

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

lemma weak_step_delay_implies_weak_tau:
  assumes ‹p =▷a p'›
  shows ‹p ⇒^a p'›
  using assms steps.refl[of p' tau] by blast

lemma weak_step_delay_left:
  assumes
    ‹¬ p0 ⟼a p1›
    ‹p0 =▷a p1›
    ‹¬tau a›
  shows
    ‹∃ p0' t. tau t ∧ p0 ⟼t p0' ∧ p0' =▷a p1›
  using assms steps_left by metis

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

lemma step_weak_step_tau:
  assumes ‹p ⟼a p'›
  shows   ‹p ⇒^a p'›
  using step_weak_step[OF assms] steps_one_step[OF assms]
  by blast

lemma step_tau_refl:
  shows   ‹p ⇒^τ p›
  using steps.refl[of p tau]
  by simp

lemma weak_step_tau_weak_step[simp]:
  assumes ‹p ⇒^a p'› ‹¬ tau a›
  shows   ‹p ⇒a p'›
  using assms by auto

lemma weak_steps:
  assumes
    ‹p ⇒a  p'›
    ‹⋀ a . tau a ⟹ A a›
    ‹A a›
  shows
    ‹p ⟼* A  p'›
proof -
  obtain pp pp' where pp:
    ‹p ⟼* tau pp› ‹pp ⟼a  pp'› ‹pp' ⟼* tau p'›
     using assms(1) by blast
  then have cascade:
```

```
    ⟨p ⟼* A pp⟩ ⟨pp ⟼* A  pp'⟩ ⟨pp' ⟼* A p'⟩
     using steps_one_step steps_spec assms(2,3) by auto
  have ⟨p ⟼* A pp'⟩ using steps_concat[OF cascade(2) cascade(1)] .
  show ?thesis using steps_concat[OF cascade(3) `p ⟼* A  pp'`] .
qed

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

lemma weak_impl_strong_step:
  assumes
    ⟨p ⇒a  p''⟩
  shows
    ⟨(∃ a' p' . tau a' ∧ p ⟼a'  p') ∨ (∃ p' . p ⟼a  p')⟩
proof  -
  from assms obtain pq1 pq2 where pq12:
    ⟨p ⟼* tau pq1⟩
     ⟨pq1 ⟼a   pq2⟩
     ⟨pq2 ⟼* tau p''⟩ by blast
  show ?thesis
  proof (cases ⟨p = pq1⟩)
    case True
    then show ?thesis using pq12 by blast
  next
    case False
    then show ?thesis using pq12 steps_left[of p pq1 tau] by blast
  qed
qed

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

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

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

abbreviation weak_enabled :: ⟨'s ⇒ 'a ⇒ bool⟩ where
  ⟨weak_enabled p a ≡
```

```
    ∃ pq1 pq2. p ⟼* tau pq1 ∧ pq1 ⟼a pq2⟩

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

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
```

2.4 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
```

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 ⇒^. Therefore we use ⇒^^ 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, simp del: weak_step_tau2_def, safe)
  fix x1 x2 R a xa
  assume ih: ⟨weak_simulation R⟩ ⟨R x1 x2⟩ ⟨x1 ⟼a  xa⟩
  then obtain q' where ⟨x2 ⇒^^ a  q'⟩ ⟨R xa q'⟩
    unfolding weak_simulation_def weak_step_tau2_def by blast
  thus ⟨∃q'. (xa ⊑ws  q' ∨ greatest_weak_simulation xa q') ∧ x2 ⇒^^ a  q'⟩
    using ih by blast
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) steps.refl by simp
  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 . ¬ step d a p ∧ ¬ step p a d⟩
  shows
    ⟨weak_simulation (λ p q . R p q ∧ q ≠ d)⟩
  unfolding weak_simulation_def
proof safe
  fix p q p' a
  assume
    ⟨R p q⟩
    ⟨q ≠ d⟩
    ⟨p ⟼a  p'⟩
  then obtain q' where ⟨R p' q'⟩ ⟨q ⇒^a  q'⟩
    using assms(1) unfolding weak_simulation_def by blast
  moreover hence ⟨q' ≠ d⟩
    using assms(2) ‘q ≠ d‘ by (metis steps.cases)
  ultimately show ⟨∃q'. (R p' q' ∧ q' ≠ d) ∧ q ⇒^a  q'⟩ by blast
qed

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

lemma weak_sim_trans_constructive:
  fixes R1 R2
  defines
    ⟨R ≡ λ p q . ∃ pq . (R1 p pq ∧ R2 pq q) ∨ (R2 p pq ∧ R1 pq q)⟩
  assumes
    R1_def: ⟨weak_simulation R1⟩ ⟨R1 p pq⟩ and
    R2_def: ⟨weak_simulation R2⟩ ⟨R2 pq q⟩
  shows
    ⟨R p q⟩ ⟨weak_simulation R⟩
proof-
  show ⟨R p q⟩ unfolding R_def using R1_def(2) R2_def(2) by blast
```

```
next
  show ⟨weak_simulation R⟩
    unfolding weak_sim_weak_premise R_def
  proof (safe)
    fix p q pq p' a pq1 pq2
    assume
      ⟨R1 p pq⟩
      ⟨R2 pq q⟩
      ⟨¬ tau a⟩
      ⟨p ⟼* tau  pq1⟩
      ⟨pq1 ⟼a  pq2⟩
      ⟨pq2 ⟼* tau  p'⟩
    thus ⟨∃q'. (∃pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q') ∧ q ⇒^a  q'⟩
      using R1_def(1) R2_def(1) unfolding weak_sim_weak_premise by blast
  next
    fix p q pq p' a
    assume
      ⟨R1 p pq⟩
      ⟨R2 pq q⟩
      ⟨p ⟼* tau  p'⟩
      ⟨∄q'.(∃pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q')∧ q ⇒^a  q'⟩
      ⟨tau a⟩
    thus ⟨False⟩
      using R1_def(1) R2_def(1) unfolding weak_sim_weak_premise by blast
  next
    fix p q pq p' a pq1 pq2
    assume
      ⟨R1 p pq⟩
      ⟨R2 pq q⟩
      ⟨p ⟼* tau  p'⟩
      ⟨p ⟼* tau  pq1⟩
      ⟨pq1 ⟼a  pq2⟩
      ⟨pq2 ⟼* tau  p'⟩
    then obtain pq' q' where ⟨R1 p' pq'⟩ ⟨pq ⇒^a  pq'⟩ ⟨R2 pq' q'⟩ ⟨q ⇒^a  q'⟩
      using R1_def(1) R2_def(1) assms(3) unfolding weak_sim_weak_premise by blast
    thus ⟨∃q'. (∃pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q') ∧ q ⇒^a  q'⟩
      by blast
  next
    fix p q pq p' a pq1 pq2
    assume sa:
      ⟨R2 p pq⟩
      ⟨R1 pq q⟩
      ⟨¬ tau a⟩
      ⟨p ⟼* tau  pq1⟩
      ⟨pq1 ⟼a  pq2⟩
      ⟨pq2 ⟼* tau  p'⟩
    then obtain pq' q'  where ⟨R2 p' pq'⟩ ⟨pq ⇒^a  pq'⟩ ⟨R1 pq' q'⟩ ⟨q ⇒^a  q'⟩
      using R2_def(1) R1_def(1) unfolding weak_sim_weak_premise by blast
    thus ⟨∃q'. (∃pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q') ∧ q ⇒^a  q'⟩
      by blast
  next
    fix p q pq p' a
    assume
      ⟨R2 p pq⟩
      ⟨R1 pq q⟩
      ⟨p ⟼* tau  p'⟩
      ⟨∄q'.(∃pq. R1 p' pq ∧ R2 pq q' ∨ R2 p' pq ∧ R1 pq q')∧ q ⇒^a  q'⟩
```

```
      ⟨tau a⟩
    thus ⟨False⟩
      using R1_def(1) R2_def(1) weak_step_tau_tau[OF ‘p ⟼* tau  p’‘ tau_tau]
      unfolding weak_sim_weak_premise by (metis (no_types, lifting))
  next
    fix p q pq p’ a pq1 pq2
    assume sa:
      ⟨R2 p pq⟩
      ⟨R1 pq q⟩
      ⟨p ⟼* tau  p’⟩
      ⟨p ⟼* tau  pq1⟩
      ⟨pq1 ⟼a  pq2⟩
      ⟨pq2 ⟼* tau  p’⟩
    then obtain pq’  where ⟨R2 p’ pq’⟩ ⟨pq ⇒^a  pq’⟩
      using R1_def(1) R2_def(1) weak_step_impl_weak_tau[of p a p’]
      unfolding weak_sim_weak_premise by blast
    moreover then obtain q’ where ⟨R1 pq’ q’⟩ ⟨q ⇒^a  q’⟩
      using R1_def(1) sa(2) unfolding weak_sim_weak_premise by blast
    ultimately show ⟨∃q’. (∃pq. R1 p’ pq ∧ R2 pq q’ ∨ R2 p’ pq ∧ R1 pq q’) ∧ q ⇒^a  q’⟩
      by blast
  qed
qed

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

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
```

3.4 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
```

3.5 Contrasimulation

```
definition contrasim ::
  ‹('s ⇒ 's ⇒ bool) ⇒ bool›
```

```
where
  ⟨contrasim R ≡ ∀ p q p' A .
    R p q ∧ (p ⇒$ A p') ⟶
    (∃ q'. (q ⇒$ A q')
         ∧ R q' p')⟩

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
    ⟨contrasim 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_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
    ⟨contrasim R⟩
  unfolding contrasim_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

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

lemma contrasim_challenge_strength_step_impl_strong_step:
  assumes
    ⟨contrasim_step R⟩
  shows
    ⟨contrasim_strong_step R⟩
  using assms step_weak_step_tau
  unfolding contrasim_step_def contrasim_strong_step_def by fastforce

lemma contrasim_reflexive:
  shows
    ⟨contrasim (λ p q . p = q)⟩
  unfolding contrasim_def using step_weak_step_tau by blast

lemma contrasim_union:
  assumes
    ⟨contrasim R1⟩
    ⟨contrasim R2⟩
  shows
    ⟨contrasim (λ p q . R1 p q ∨ R2 p q)⟩
  using assms unfolding contrasim_def by (blast)

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

lemma contrasim_implies_coupling:
  assumes
    ⟨contrasim R⟩ — actually also is true with 'weaker' contrasim_step
    ⟨R p q⟩
  shows
    ⟨∃ q'. q ⟼*tau q' ∧ R q' p⟩
proof -
  have ⟨p ⟼* tau p⟩ using steps.refl by blast
  hence ⟨p ⇒^τ p⟩ using tau_tau by blast
  then obtain q' where ⟨q ⇒^τ q'⟩ ⟨R q' p⟩
    using `R p q` `contrasim R` unfolding contrasim_def  by blast
  then moreover have ⟨q ⟼* tau q'⟩ using tau_tau by blast
  ultimately show ?thesis by blast
qed

lemma symm_contrasim_implies_weak_bisim:
  assumes
    ⟨contrasim_strong_step R⟩
    ⟨⋀ p q . R p q ⟹ R q p⟩
  shows
    ⟨weak_bisimulation R⟩
```

```
  unfolding weak_bisimulation_def
proof safe
  fix p q p' a
  assume ⟨R p q⟩ ⟨p ⟼a  p'⟩
  then obtain q' where q'_def: ⟨q ⇒^a q'⟩ ⟨R q' p'⟩
    using assms(1) unfolding contrasim_strong_step_def by blast
  thus ⟨∃q'. R p' q' ∧ q ⇒^a  q'⟩ using assms(2) by blast
next
  fix p q q' a
  assume ⟨R p q⟩ ⟨q ⟼a  q'⟩
  hence ⟨R q p⟩ using assms(2) by blast
  then obtain p' where p'_def: ⟨p ⇒^a p'⟩ ⟨R p' q'⟩
    using ‘q ⟼a  q'‘ assms(1) unfolding contrasim_strong_step_def by blast
  thus ⟨∃p'. R p' q' ∧ p ⇒^a  p'⟩ using assms(2) by blast
qed

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

lemma taufree_contrasim_symm:
  assumes
    ⟨⋀ p1 a p2 . (p1 ⟼a p2 ⟹ ¬ tau a)⟩
    ⟨contrasim R⟩
    ⟨R p q⟩
  shows ⟨R q p⟩
  using assms contrasim_implies_coupling
  by (metis steps.cases)

lemma taufree_contrasim_implies_weak_bisim:
  assumes
    ⟨⋀ p1 a p2 . (p1 ⟼a p2 ⟹ ¬ tau a)⟩
    ⟨contrasim R⟩
  shows
    ⟨weak_bisimulation R⟩
  using assms symm_contrasim_implies_weak_bisim taufree_contrasim_symm
    contrasim_step_weaker_than_seq[OF assms(2)]
    contrasim_challenge_strength_step_impl_strong_step by blast

lemma contrasim_challenge_strength_does_not_imply:
  fixes p1 q1
  defines
    ⟨R ≡ λ p q . p = p1 ∧ q = q1⟩
```

```
  assumes
    ‹p1 ≠ q1›
    ‹trans = (λ p a p' . False)›
  shows
    ‹contrasim_strong_step R› ‹¬contrasim R›
  using taufree_contrasim_symm[of R p1 q1] assms
  unfolding contrasim_strong_step_def by (blast+)

end — context lts_tau
```

3.6 Similarity ignores τ-sinks

```
lemma simulation_tau_sink_1:
  fixes
    step sink τ R
  defines
    ‹step2 ≡ λ p1 a p2 . (p1 ≠ sink ∧ a = τ ∧ p2 = sink) ∨ step p1 a p2›
  assumes
    ‹⋀ a p . ¬ step sink a p›
    ‹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] lts.steps.refl[of step2 q ?tau]  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 τ R
  defines
    ‹step2 ≡ λ p1 a p2 . (p1 ≠ sink ∧ a =  τ ∧ p2 = sink) ∨ step p1 a p2›
  assumes
    ‹⋀ a p . ¬ step sink a p ∧ ¬ step p a sink›
  shows ‹lts_tau.weakly_simulated_by step2 τ p q = lts_tau.weakly_simulated_by step τ p q›
proof (rule)
  have sink_sim_min: ‹lts_tau.weak_simulation step2 τ (λ p q. p = sink)›
    unfolding lts_tau.weak_simulation_def step2_def using assms(2)
    by (meson lts.steps.simps)
  define R where ‹R ≡ lts_tau.weakly_simulated_by step2 τ›
  have weak_sim_R: ‹lts_tau.weak_simulation step2 τ R›
    using lts_tau.weaksim_greatest[of step2 τ] unfolding R_def by blast
  have R_contains_inv_tau_closure:
    ‹R = (λp1 q1. R p1 q1 ∨ lts.steps step2 q1 (lts_tau.tau τ) p1)› unfolding R_def
  proof (rule, rule, rule, simp)
    fix p q
    assume
      ‹(∃R. lts_tau.weak_simulation step2 τ  R ∧ R p q) ∨
       (lts.steps step2 q (lts_tau.tau τ) p)›
    thus ‹∃R. lts_tau.weak_simulation step2 τ R ∧ R p q›
      using weak_sim_R
            lts_tau.weak_sim_tau_step[of step2 ‹τ›]
            lts_tau.weak_sim_union_cl[of step2 ‹τ›] by blast
  qed
  assume Rpq: ‹R p q›
  have ‹⋀ p' q' q . R p' q' ∧ lts.steps step2 q (lts_tau.tau τ) q'  ⟶ R p' q›
    using R_contains_inv_tau_closure lts_tau.weak_sim_trans[of step2 ‹τ› _ _ _] R_def assms(2)
    by meson
  hence closed_R:
      ‹⋀ p' q' q . (p' = sink ∨ R p' q') ∧ lts.steps step2 q (lts_tau.tau τ) q'
         ⟶ (p' = sink ∨ R p' q)›
    using weak_sim_R sink_sim_min lts_tau.weak_sim_union_cl by blast
  have ‹lts_tau.weak_simulation step2 τ (λp q. p = sink ∨ R p q)›
    using weak_sim_R sink_sim_min lts_tau.weak_sim_union_cl[of step2 τ] by blast
  hence ‹lts_tau.weak_simulation step τ (λp q. p = sink ∨ R p q)›
    using  simulation_tau_sink_2[of step sink τ R] assms(2) closed_R
    unfolding step2_def by blast
  thus ‹∃R. lts_tau.weak_simulation step τ R ∧ R p q›
    using Rpq weak_sim_R by blast
next
  show ‹∃R. lts_tau.weak_simulation step τ R ∧ R p q ⟹
        ∃R. lts_tau.weak_simulation step2 τ R ∧ R p q›
  proof clarify
    fix R
    assume
      ‹lts_tau.weak_simulation step τ R›
      ‹R p q›
    hence ‹lts_tau.weak_simulation
      (λp1 a p2. p1 ≠ sink ∧ a = τ ∧ p2 = sink ∨ step p1 a p2) τ (λp q. p = sink ∨ R p q)›
      using simulation_tau_sink_1[of step sink τ R] assms(2) unfolding step2_def by auto
    thus ‹∃R. lts_tau.weak_simulation step2 τ R ∧ R p q›
      using `R p q` unfolding step2_def by blast
```

```
  qed
qed

end
```

4 Coupled Similarity

```
theory Coupled_Simulation
  imports Weak_Relations
begin

context lts_tau
begin
```

4.1 Van Glabbeeks's Coupled Simulation

We mainly use van Glabbeek's coupled simulation from his 2017 CSP paper [4]. 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.

4.2 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
```

4.3 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
```

4.4 Closure, Preorder and Symmetry Properties

The coupled simulation preorder $\sqsubseteq$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
```

4.5 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, simp del: weak_step_tau2_def, safe)
  fix x1 x2 R a xa
  assume ih: ⟨coupled_simulation R⟩ ⟨R x1 x2⟩ ⟨x1 ⟼a  xa⟩
  then obtain q' where ⟨x2 ⇒^^ a  q'⟩ ⟨R xa q'⟩
    unfolding coupled_simulation_def weak_step_tau2_def by blast
  thus ⟨∃q'. x2 ⇒^^ a  q' ∧ (xa ⊑cs  q' ∨ greatest_coupled_simulation xa q')⟩
    using ih by blast
next
  fix x1 x2 R
  assume ih: ⟨coupled_simulation R⟩ ⟨R x1 x2⟩
  then obtain q' where ⟨x2 ⟼* tau  q'⟩ ⟨R q' x1⟩
    unfolding coupled_simulation_def by blast
  thus ⟨∃q'. x2 ⟼* tau  q' ∧ (q' ⊑cs  x1 ∨ greatest_coupled_simulation q' x1)⟩
    using ih by blast
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)
```

4.6 Coupled Simulation Join

The following lemmas reproduce Proposition 3 from [4] 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, auto)
    fix x1 x2 x xa
    assume ih:
      ‹R x1 x2›
      ‹x1 ⟼x  xa›
    hence ‹x1 = q ∧ x2 = pqc ∨ x1 = pqc ∧ x2 = q ∨ x1 = p ∧ x2 = q› using R_def by auto
    thus ‹∃q'. (tau x ⟶ x2 ⟼* tau  q') ∧
      (¬ tau x ⟶ (∃pq1. x2 ⟼* tau  pq1 ∧
        (∃pq2. pq1 ⟼x  pq2 ∧ pq2 ⟼* tau  q'))) ∧
      (R xa q' ∨ greatest_coupled_simulation xa q')›
    proof safe
      assume ‹x1 = q› ‹x2 = pqc›
      thus ‹∃q'.
        (tau x ⟶ pqc ⟼* tau  q') ∧
        (¬tau x ⟶ (∃pq1. pqc ⟼* tau  pq1 ∧
          (∃pq2. pq1 ⟼x  pq2 ∧ pq2 ⟼* tau  q'))) ∧
        (R xa q' ∨ greatest_coupled_simulation xa q')›
        using ih ‘q ⊑cs pqc‘
          coupled_simulation_implies_weak_simulation[OF gcs_is_coupled_simulation]
        unfolding gcs_eq_coupled_sim_by
        by (metis (full_types) weak_sim_ruleformat)
    next
      assume ‹x1 = pqc› ‹x2 = q›
      hence ‹x = τ ∧ (xa = q ∨ xa = p)› using ih(2) assms(2) by blast
      thus ‹∃q'.
```

```
        (tau x ⟶ q ⟼⟶* tau  q') ∧
        (¬ tau x ⟶ (∃pq1. q ⟼⟶* tau  pq1 ∧
          (∃pq2. pq1 ⟼⟶x  pq2 ∧ pq2 ⟼⟶* tau  q'))) ∧
        (R xa q' ∨ greatest_coupled_simulation xa q')⟩
        unfolding gcs_eq_coupled_sim_by[symmetric]
        using steps.refl[of q tau] ‘p ⊑cs q‘ tau_tau
        by auto
    next
      assume ⟨x1 = p⟩ ⟨x2 = q⟩
      thus ⟨∃q'.
         (tau x ⟶ q ⟼⟶* tau  q') ∧
         (¬ tau x ⟶ (∃pq1. q ⟼⟶* tau  pq1 ∧
            (∃pq2. pq1 ⟼⟶x  pq2 ∧ pq2 ⟼⟶* tau  q'))) ∧
         (R xa q' ∨ greatest_coupled_simulation xa q')⟩
        using ih ‘p ⊑cs q‘
          coupled_simulation_implies_weak_simulation[OF gcs_is_coupled_simulation]
        unfolding gcs_eq_coupled_sim_by
        by (metis (full_types) weak_sim_ruleformat)
    qed
  next
    fix x1 x2
    assume
      ⟨R x1 x2⟩
    hence ⟨x1 = q ∧ x2 = pqc ∨ x1 = pqc ∧ x2 = q ∨ x1 = p ∧ x2 = q⟩ using R_def by auto
    thus ⟨∃q'. x2 ⟼⟶* tau  q' ∧ (R q' x1 ∨ greatest_coupled_simulation q' x1)⟩
    proof (auto)
      show ⟨∃q'. pqc ⟼⟶* tau  q' ∧ (R q' q ∨ greatest_coupled_simulation q' q)⟩
        using ⟨R pqc q⟩ steps.simps by blast
    next
      show ⟨∃q'. q ⟼⟶* tau  q' ∧ (R q' pqc ∨ greatest_coupled_simulation q' pqc)⟩
        using ⟨R1 q pqc⟩ ⟨coupled_simulation R1⟩ coupled_similarity_implies_gcs steps.refl
        by blast
    next
      show ⟨∃q'. q ⟼⟶* tau  q' ∧ (R q' p ∨ greatest_coupled_simulation q' p)⟩
        using assms(1) coupled_simulation_weak_simulation
          lts_tau.coupled_similarity_implies_gcs by fastforce
    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
```

4.7 Coupled Delay Simulation

$\sqsubseteq_{cs}$ 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
```

4.8 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
    ⟨contrasim R⟩
    ⟨weak_simulation R⟩
  shows
    ⟨coupled_simulation R⟩
  unfolding coupled_simulation_weak_simulation
  using contrasim_implies_coupling assms by blast

lemma coupledsim_implies_contrasim:
  assumes
    ⟨coupled_simulation R⟩
  shows
    ⟨contrasim 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 ⟨contrasim 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 ⟷ contrasim R ∧ weak_simulation R›
  using weak_sim_and_contrasim_implies_coupled_sim
    coupledsim_implies_contrasim coupled_simulation_weak_simulation by blast
```

If there is a sink every state can reach via tau steps, then weak simulation implies (and thus coincides with) coupled simulation.

```
lemma tau_sink_sim_coupledsim:
  assumes
    ‹⋀ p . (p ⟼* tau sink)›
    ‹⋀ p . R sink p›
    ‹weak_simulation R›
  shows
    ‹coupled_simulation R›
  unfolding coupled_simulation_def
proof safe
  show ‹ ⋀p q p' a. R p q ⟹ p ⟼a  p' ⟹ ∃q'. R p' q' ∧ q ⇒^a  q'›
    using assms(3) unfolding weak_simulation_def by blast
next
  fix p q
  assume ‹R p q›
  hence ‹q ⟼* tau sink ∧ R sink p›
    using assms(1,2) by blast
  thus ‹∃q'. q ⟼* tau  q' ∧ R q' p› by blast
qed
```

4.9 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) taufree_contrasim_symm coupledsim_implies_contrasim[OF assms(2)]
  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 taufree_contrasim_implies_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) contrasim_implies_coupling coupledsim_implies_contrasim by fastforce
  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
```

4.10 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
```

4.11 Reduction Semantics Coupled Simulation

The tradition to described coupled simulation as special delay/weak simulation is quite common for coupled simulations on reduction semantics as in [8, 3], of which [8] can also be found in the AFP [7]. The notions coincide (in 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
```

4.12 Coupled Simulation as Two Simulations

Historically, coupled similarity has been defined in terms of *two* weak simulations coupled in some way [9, 6]. We reproduce these (more well-known) formulations and show that they are equivalent to the coupled (delay) simulations we are using.

— From [9]

```
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
```

4.13 S-coupled Simulation

Originally coupled simulation was introduced as two weak simulations coupled at the stable states. We give the definitions from [5, 6] and a proof connecting this notion to "our" coupled similarity in the absence of divergences following [9].

— From [5]

```
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 [9], 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 [9], 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 [9], 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⟩ 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
```

5 Fixed Point Algorithm for Coupled Similarity

5.1 The Algorithm

```
theory CS_Fixpoint_Algo_Delay
imports
  Coupled_Simulation
  "~~/src/HOL/Library/While_Combinator"
  "~~/src/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⟩
```

5.2 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

thm   prod.simps(2)

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
.
  moreover from asm have  ‹∀ p' a. p ⟼a p' ∧ ¬tau a ⟶
      (∃ q'. (greatest_coupled_simulation p' q') ∧ (q =▷a 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 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
```

6 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⟩ and
  initial :: 's
begin

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

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

```
type_synonym ('s2) play = ⟨'s2 list⟩
type_synonym ('s2) strategy = ⟨'s2 play ⇒ 's2⟩

inductive_set plays :: ⟨'s play set⟩ where
  ⟨[initial] ∈ plays⟩ |
  ⟨p#play ∈ plays ⟹ p ⟼♡ p' ⟹ p'#p#play ∈ plays⟩
```

— plays for a given player 0 strategy

```
inductive_set plays_for_strategy :: ⟨'s strategy ⇒ 's play set⟩ for f where
  init: ⟨[initial] ∈ plays_for_strategy f⟩ |
  p0move: ⟨n0#play ∈ plays_for_strategy f ⟹ player0_position n0 ⟹ n0 ⟼♡ f (n0#play)
    ⟹ (f (n0#play))#n0#play ∈ plays_for_strategy f⟩ |
  p1move: ⟨n1#play ∈ plays_for_strategy f ⟹ player1_position n1 ⟹ n1 ⟼♡ n1'
    ⟹ n1'#n1#play ∈ plays_for_strategy f⟩

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

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

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

```
definition sound_strategy :: ⟨'s strategy ⇒ bool⟩ where
  ⟨sound_strategy f ≡
    ∀ n0 play . n0#play ∈ plays_for_strategy f ∧ player0_position n0 ⟶ n0 ⟼♡ f (n0#play)⟩

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

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

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

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

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

end

end
```

7 Game for Coupled Similarity with Delay Formulation

```
theory CS_Game_Delay
imports
  Coupled_Simulation
  Simple_Game
begin
```

7.1 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 initial
for
  trans :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩ and
```

```
  τ :: ⟨'a⟩ and
  initial :: ⟨('s, 'a) cs_game_node⟩
begin
```

7.2 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⟩
    ⟨cs_game_defender_node n0⟩
    ⟨initial = AttackerNode p0 q0⟩
  shows ⟨∃ p q . hd play = AttackerNode p q ∧ cs_game_moves (AttackerNode p q) n0⟩ ⟨play ≠
[]⟩
proof -
  have n0_not_init: ⟨n0 ≠ initial⟩ using assms(2,3) 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
    ⟨initial = AttackerNode p0 q0⟩
    ⟨(DefenderStepNode a p q) # play ∈ plays⟩
  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_strategy (strategy_from_coupleddsim R)⟩
  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_strategy.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› using strategy_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_strategy (strategy_from_coupleddsim R)›
  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[OF 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 unfolding player1_position_def 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_strategy (strategy_from_coupleddsim R)⟩
  unfolding sound_strategy_def
proof clarify
  fix n0 play
  assume subassms:
    ⟨n0 # play ∈ plays_for_strategy(strategy_from_coupleddsim R)⟩
    ⟨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[OF _ `cs_game_defender_node n0` assms(3)]
      strategy_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)⟩
  unfolding player0_winning_strategy_def
proof (clarify)
  fix play
  assume subassms:
    ⟨play ∈ plays_for_strategy (strategy_from_coupleddsim R)⟩
    ⟨player1_wins play⟩
  show ⟨False⟩ using subassms
  proof (induct rule: simple_game.plays_for_strategy.induct[OF subassms(1)])
    case 1
    then show ?case unfolding player1_wins_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 (strategy_from_coupleddsim R (n0 # play) # n0 # play)⟩
      unfolding player1_wins_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⟩
      unfolding player1_position_def 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
      by (metis cs_game_defender_node.elims(2) 3(6) list.sel(1) player1_wins_def)
    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 (n1' # n1 # play)‘ show False unfolding player1_wins_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 (n1' # n1 # play)‘ show False unfolding player1_wins_def n1'_def
        by (metis list.sel(1))
    qed
  qed
qed
```

7.3 Winning Strategy Induces Coupled Simulation

```
lemma winning_strategy_implies_coupleddsim:
  assumes
    ⟨player0_winning_strategy f⟩
    ⟨sound_strategy f⟩
  defines
    ⟨R == λ p q . (∃ play ∈ plays_for_strategy f . 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_strategy f . hd play = AttackerNode p q)⟩
    using challenge(1) assms by blast
  then obtain play where
      play_spec: ⟨AttackerNode p q # play ∈ plays_for_strategy f⟩
    by (metis list.sel(1) simple_game.plays.cases strategy_plays_subset)
  hence interplay: ⟨(AttackerNode p' q) # AttackerNode p q # play ∈ plays_for_strategy f⟩
    using game_move by (simp add: player1_position_def simple_game.plays_for_strategy.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_strategy f . hd play = AttackerNode p q)›
    using challenge(1) assms by blast
  then obtain play where
      play_spec: ‹AttackerNode p q # play ∈ plays_for_strategy f›
    by (metis list.sel(1) simple_game.plays.cases strategy_plays_subset)
  hence interplay: ‹(DefenderStepNode a p' q) # AttackerNode p q # play ∈ plays_for_strategy
f›
    using game_move by (simp add: player1_position_def simple_game.plays_for_strategy.p1move)
  hence ‹¬ player1_wins ((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_strategy_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_strategy f›
    using interplay n1_def by (simp add: simple_game.plays_for_strategy.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_strategy f . hd play = AttackerNode p q)›
    using challenge assms by blast
  then obtain play where
      play_spec: ‹AttackerNode p q # play ∈ plays_for_strategy f›
    by (metis list.sel(1) simple_game.plays.cases strategy_plays_subset)
  hence interplay: ‹(DefenderCouplingNode p q) # AttackerNode p q # play
      ∈ plays_for_strategy f›
    using game_move by (simp add: player1_position_def simple_game.plays_for_strategy.p1move)
  hence ‹¬ player1_wins ((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_strategy_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_strategy f›
    using interplay n1_def by (simp add: simple_game.plays_for_strategy.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 ∧ sound_strategy f)
    = p ⊑cs q›
proof (rule)
  assume
    ‹(∃f. player0_winning_strategy f ∧ sound_strategy f)›
  then obtain f where
    ‹coupled_delay_simulation (λp q. ∃play∈plays_for_strategy f. hd play = AttackerNode p
q)›
    using winning_strategy_implies_coupleddsim by blast
  moreover have ‹(λp q. ∃play∈plays_for_strategy f. hd play = AttackerNode p q) p q›
    using assms plays_for_strategy.init[of f] by (meson list.sel(1))
  ultimately show ‹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 ∧ sound_strategy f)›
    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
```

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., Nestmann, U.: Computing coupled similarity. In: 25th TACAS 2019: Prague, Czech Republic (Part of ETAPS 2019). Lecture Notes in Computer Science, Springer (2019), forthcoming

[3] Fournet, C., Gonthier, G.: A hierarchy of equivalences for asynchronous calculi. The Journal of Logic and Algebraic Programming **63**(1), 131–173 (2005)

[4] van Glabbeek, R.J.: A branching time model of CSP. CoRR (2017), http://arxiv.org/abs/1702.07844

[5] Parrow, J., Sjödin, P.: Multiway synchronization verified with coupled simulation. In: Cleaveland, W. (ed.) CONCUR '92: Third International Conference on Concurrency Theory Stony Brook, NY, USA, August 24–27, 1992 Proceedings. pp. 518–533. Springer Berlin Heidelberg (1992). https://doi.org/10.1007/BFb0084813

[6] Parrow, J., Sjödin, P.: The complete axiomatization of Cs-congruence. In: Enjalbert, P., Mayr, E.W., Wagner, K.W. (eds.) STACS 94: 11th Annual Symposium on Theoretical Aspects of Computer Science Caen, France, February 24–26, 1994 Proceedings. pp. 555–568. Springer Berlin Heidelberg (1994). https://doi.org/10.1007/3-540-57785-8_171

[7] Peters, K., van Glabbeek, R.: Analysing and comparing encodability criteria for process calculi. Archive of Formal Proofs (Aug 2015), http://isa-afp.org/entries/Encodability_Process_Calculi.html, Formal proof development

[8] Peters, K., van Glabbeek, R.J.: Analysing and comparing encodability criteria. In: Proceedings of the Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS. pp. 46–60 (2015). https://doi.org/10.4204/EPTCS.190.4

[9] Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York, NY, USA (2012)