Coupled Similarity and Contrasimilarity,
and How to Compute Them

Benjamin Bisping* Luisa Montanari

September 4, 2026

Abstract

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

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

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

Contents

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

2 Preliminaries **4**
- 2.1 Labeled Transition Systems . . . 4
- 2.2 Transition Systems with Silent Steps . . . 6
- 2.3 Finite Transition Systems with Silent Steps . . . 12
- 2.4 Simple Games . . . 12

3 Notions of Equivalence **14**
- 3.1 Strong Simulation and Bisimulation . . . 14
- 3.2 Weak Simulation . . . 15
- 3.3 Weak Bisimulation . . . 17
- 3.4 Trace Inclusion . . . 19
- 3.5 Delay Simulation . . . 19
- 3.6 Coupled Equivalences . . . 19

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

4 Contrasimulation **20**
4.1 Definition of Contrasimulation . . . 20
4.2 Intermediate Relation Mimicking Contrasim . . . 22
4.3 Over-Approximating Contrasimulation by a Single-Step Version . . . 22

5 Coupled Simulation **23**
5.1 Van Glabbeeks's Coupled Simulation . . . 23
5.2 Position between Weak Simulation and Weak Bisimulation . . . 23
5.3 Coupled Simulation and Silent Steps . . . 24
5.4 Closure, Preorder and Symmetry Properties . . . 25
5.5 Coinductive Coupled Simulation Preorder . . . 25
5.6 Coupled Simulation Join . . . 26
5.7 Coupled Delay Simulation . . . 26
5.8 Relationship to Contrasimulation and Weak Simulation . . . 27
5.9 τ-Reachability (and Divergence) . . . 27
5.10 On the Connection to Weak Bisimulation . . . 29
5.11 Reduction Semantics Coupled Simulation . . . 30
5.12 Coupled Simulation as Two Simulations . . . 30
5.13 S-coupled Simulation . . . 31

6 Game for Coupled Similarity with Delay Formulation **34**
6.1 The Coupled Simulation Preorder Game Using Delay Steps . . . 34
6.2 Coupled Simulation Implies Winning Strategy . . . 35
6.3 Winning Strategy Induces Coupled Simulation . . . 36

7 Fixed Point Algorithm for Coupled Similarity **36**
7.1 The Algorithm . . . 36
7.2 Correctness . . . 36

8 The Contrasimulation Preorder Word Game **37**
8.1 Contrasimulation Implies Winning Strategy in Word Game (Completeness) . . . 38
8.2 Winning Strategy Implies Contrasimulation in Word Game (Soundness) . . . 39

9 The Contrasimulation Preorder Set Game **39**
9.1 Contrasimulation Implies Winning Strategy in Set Game (Completeness) . . . 40
9.2 Winning Strategy Implies Contrasimulation in Set Game (Soundness) . . . 41

10 Infinitary Hennessy–Milner Logic **42**
10.1 Satisfaction Relation . . . 43
10.2 Distinguishing Formulas . . . 43
10.3 Weak-NOR Hennessy–Milner Logic . . . 44

11 Weak HML and the Contrasimulation Set Game **44**
11.1 Distinguishing Formulas at Winning Attacker Positions . . . 45
11.2 Attacker Wins on Pairs with Distinguishing Formulas . . . 46

12 Reductions and τ-sinks **47**
12.1 τ-Sink Properties . . . 48
12.2 Contrasimulation Equals Weak Simulation on τ-Sink Systems . . . 48
12.3 Contrasimulation Equals Weak Trace Inclusion on τ-Sink Systems . . . 48
12.4 Weak Simulation Invariant under τ-Sink Extension . . . 49
12.5 Trace Inclusion Invariant under τ-Sink Extension . . . 50

References **50**

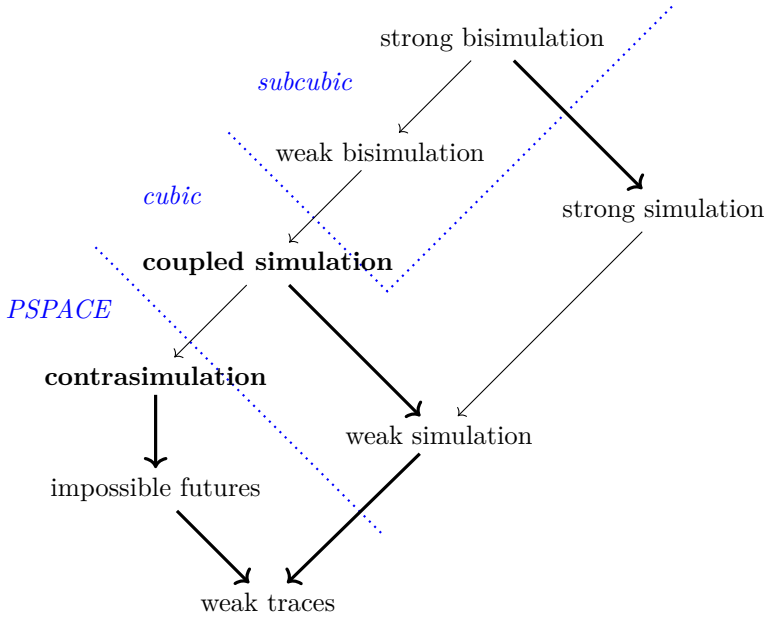

Figure 1: Hierarchy of weak behavioral preorders/equivalences. Arrows denote implication of preordering. Thinner arrows collapse into bi-implication for systems without internal steps. Blue parts indicate a slope of decision problem complexities.

1 Introduction

Coupled similarity and contrasimilarity are among the weakest abstractions of bisimilarity for systems with silent steps presented in van Glabbeek's *linear-time–branching-time spectrum* of behavioral equivalences [8]. In particular, they are *weaker than weak bisimilarity* in that they impose a weaker form of symmetry on the bisimulation link between compared states; coupled similarity implies contrasimilarity. They are weak bisimilarities, however, in the sense that, on systems with no internal behavior, they coincide with strong bisimilarity.

1.1 This Theory

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

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

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

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

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

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

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

1.3 Modal Intuition

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

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

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

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

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

2 Preliminaries

2.1 Labeled Transition Systems

```
theory Transition_Systems
  imports Main
begin

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

begin

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

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

lemma steps_one_step:
```

```
  assumes
    ‹p ⟼a p'›
     ‹A a›
   shows
    ‹p ⟼* A p'› ⟨proof⟩

lemma steps_concat:
  assumes
    ‹p' ⟼* A p''›
     ‹p ⟼* A p'›
   shows
    ‹p ⟼* A p''› ⟨proof⟩

lemma steps_left:
  assumes
    ‹p ≠ p'›
     ‹p ⟼* A p'›
   shows
    ‹∃p'' a . p ⟼a p'' ∧ A a ∧ p'' ⟼* A p'›
   ⟨proof⟩

lemma steps_no_step:
  assumes
    ‹⋀ a p' . p ⟼a p' ⟹ ¬A a›
     ‹p ≠ p''›
     ‹p ⟼* A p''›
   shows
    ‹False›
   ⟨proof⟩

lemma steps_no_step_pos:
  assumes
    ‹⋀ a p' . p ⟼a p' ⟹ ¬A a›
     ‹p ⟼* A p'›
   shows
    ‹p = p'›
   ⟨proof⟩

lemma steps_loop:
  assumes
    ‹⋀ a p' . p ⟼a p' ⟹ p = p'›
     ‹p ≠ p''›
     ‹p ⟼* A p''›
   shows
    ‹False›
   ⟨proof⟩

corollary steps_transp:
  ‹transp (λ p p'. p ⟼* A p')›
   ⟨proof⟩

lemma steps_spec:
  assumes
    ‹p ⟼* A' p'›
     ‹⋀ a . A' a ⟹ A a›
   shows
    ‹p ⟼* A p'› ⟨proof⟩
```

```
interpretation preorder ‹(λ p p'. p ⟼* A p')› ‹λ p p'. p ⟼* A p' ∧ ¬(p' ⟼* A p)›
```
⟨*proof*⟩

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*⟩

`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*⟩

```
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›
```
⟨*proof*⟩

```
end
```

2.2 Transition Systems with Silent Steps

```
theory Weak_Transition_Systems
  imports Transition_Systems
begin

locale lts_tau = lts trans for
  trans :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ⟼_  _› [70, 70, 70] 80) + fixes
  τ :: ‹'a› begin

definition tau :: ‹'a ⇒ bool› where ‹tau a ≡ (a = τ)›

lemma tau_tau[simp]: ‹tau τ› ⟨proof⟩

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'›
  ⟨proof⟩

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'›
  ⟨proof⟩

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›
  ⟨proof⟩

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'›
  ⟨proof⟩

lemma step_tau_refl:
  shows   ‹p ⇒^τ p›
  ⟨proof⟩

lemma weak_step_tau_weak_step[simp]:
  assumes ‹p ⇒^a p'› ‹¬ tau a›
  shows   ‹p ⇒a p'›
  ⟨proof⟩
```

```
lemma weak_steps:
  assumes
    ‹p ⇒a  p'›
    ‹⋀ a . tau a ⟹ A a›
    ‹A a›
  shows
    ‹p ⟼* A  p'›
```
⟨*proof*⟩

```
lemma weak_step_impl_weak_tau:
  assumes
    ‹p ⇒a p'›
  shows
    ‹p ⇒^a p'›
```
⟨*proof*⟩

```
lemma weak_impl_strong_step:
  assumes
    ‹p ⇒a  p''›
  shows
    ‹(∃ a' p' . tau a' ∧ p ⟼a'  p') ∨ (∃ p' . p ⟼a  p')›
```
⟨*proof*⟩

```
lemma weak_step_extend:
  assumes
    ‹p1 ⟼* tau p2›
    ‹p2 ⇒^a p3›
    ‹p3 ⟼* tau p4›
  shows
    ‹p1 ⇒^a p4›
```
⟨*proof*⟩

```
lemma weak_step_tau_tau:
  assumes
    ‹p1 ⟼* tau p2›
    ‹tau a›
  shows
    ‹p1 ⇒^a p2›
```
⟨*proof*⟩

```
lemma weak_single_step[iff]:
  ‹p ⇒$ [a] p' ⟷ p ⇒^a  p'›
```
⟨*proof*⟩

```
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')›
```
⟨*proof*⟩

```
lemma step_tau_concat:
  assumes
    ‹q ⇒^a q'›
    ‹q' ⇒^τ q1›
  shows ‹q ⇒^a q1›
```

⟨*proof*⟩

```
lemma tau_step_concat:
  assumes
    <q ⇒^τ q'>
    <q' ⇒^a q1>
  shows <q ⇒^a q1>
```
⟨*proof*⟩

```
lemma tau_word_concat:
  assumes
    <q ⇒^τ q'>
    <q' ⇒$A q1>
  shows <q ⇒$A q1>
```
⟨*proof*⟩

```
lemma strong_weak_transition_system:
  assumes
    <⋀ p q a. p ⟼ a q ⟹ ¬ tau a>
    <¬ tau a>
  shows
    <p ⇒^a p' = p ⟼ a p'>
```
⟨*proof*⟩

```
lemma rev_seq_split :
  assumes <q ⇒$ (xs @ [x])  q1>
  shows <∃q'. q ⇒$xs q' ∧ q' ⇒^x q1>
```
⟨*proof*⟩

```
lemma rev_seq_concat:
  assumes
    <q ⇒$as q'>
    <q'⇒$A q1>
  shows <q ⇒$(as@A) q1>
```
⟨*proof*⟩

```
lemma rev_seq_step_concat :
  assumes
    <q ⇒$as q'>
    <q'⇒^a q1>
  shows <q ⇒$(as@[a]) q1>
```
⟨*proof*⟩

```
lemma rev_seq_dstep_concat :
  assumes
    <q ⇒$as q'>
    <q'=▷a q1>
  shows <q ⇒$(as@[a]) q1>
```
⟨*proof*⟩

```
lemma word_tau_concat:
  assumes
    <q ⇒$A q'>
    <q' ⇒^τ q1>
  shows <q ⇒$A q1>
```
⟨*proof*⟩

```
lemma list_rev_split :
  assumes ‹A ≠ []›
  shows ‹∃as a. A = as@[a]›
```
⟨*proof*⟩

```
primrec taufree :: ‹'a list ⇒ 'a list›
  where
    ‹taufree [] = []›
  | ‹taufree (a#A) = (if tau a then taufree A else a#(taufree A))›

lemma weak_step_over_tau :
  assumes
    ‹p ⇒$A p'›
  shows ‹p ⇒$(taufree A) p'›
```
⟨*proof*⟩

```
lemma app_tau_taufree_list :
  assumes
    ‹∀a ∈ set A. ¬tau a›
    ‹b = τ›
  shows ‹A = taufree (A@[b])›
```
⟨*proof*⟩

```
lemma word_steps_ignore_tau_addition:
  assumes
    ‹∀a∈set A. a ≠ τ›
    ‹p ⇒$ A p'›
    ‹filter (λa. a ≠ τ) A' = A›
  shows
    ‹p ⇒$ A' p'›
```
⟨*proof*⟩

```
lemma word_steps_ignore_tau_removal:
  assumes
    ‹p ⇒$ A p'›
  shows
    ‹p ⇒$ (filter (λa. a ≠ τ) A) p'›
```
⟨*proof*⟩

```
definition weak_tau_succs :: "'s set ⇒ 's set" where
  ‹weak_tau_succs Q  = {q1. ∃q∈ Q. q ⇒^τ q1}›

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

definition word_reachable_via_delay :: "'a list ⇒ 's ⇒ 's ⇒ 's ⇒ bool" where
  ‹word_reachable_via_delay A p p0 p1 = (∃p00. p ⇒$(butlast A) p00 ∧ p00 =▷(last A) p0 ∧
p0 ⇒^τ p1)›

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

lemma in_dsuccs_implies_word_reachable:
  assumes
    ‹q' ∈ dsuccs_seq_rec (rev A) {q}›
  shows
    ‹q ⇒$A q'›
```

⟨*proof*⟩

```
lemma word_reachable_implies_in_dsuccs :
  assumes
    ‹q ⇒$A q'›
  shows ‹q' ∈ weak_tau_succs (dsuccs_seq_rec (rev A) {q})› ⟨proof⟩

lemma simp_dsuccs_seq_rev:
  assumes
    ‹Q = dsuccs_seq_rec (rev A) {q0}›
  shows
    ‹dsuccs a Q = dsuccs_seq_rec (rev (A@[a])) {q0}›
⟨proof⟩

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'›
  ⟨proof⟩

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›
  ⟨proof⟩

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›
  ⟨proof⟩

lemma ex_not_divergent:
  assumes ‹∀a q. p ⟼a q ⟶ ¬ tau a› ‹divergent_state p›
  shows ‹False› ⟨proof⟩

lemma perpetual_instability_divergence:
  assumes
    ‹∀ p' . p ⟼* tau p' ⟶ ¬ stable_state p'›
  shows
    ‹divergent_state p›
  ⟨proof⟩

corollary non_divergence_implies_eventual_stability:
  assumes
```

```
    ‹¬ divergent_state p›
  shows
    ‹∃ p' . p ⟼* tau p' ∧ stable_state p'›
  ⟨proof⟩

end — context lts_tau
```

2.3 Finite Transition Systems with Silent Steps

```
locale lts_tau_finite = lts_tau trans τ for
  trans :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› and
  τ :: ‹'a› +
assumes
  finite_state_set: ‹finite (top::'s set)›
begin

lemma finite_state_rel: ‹finite (top::('s rel))›
  ⟨proof⟩

end

end
```

2.4 Simple Games

```
theory Simple_Game
imports
  Main
begin
```

Simple games are games where player0 wins all infinite plays.

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

abbreviation player1_position :: ‹'s ⇒ bool›
  where ‹player1_position s ≡ ¬ player0_position s›
```

— Plays (to be precise: play prefixes) are lists. We model them with the most recent move at the beginning. (For our purpose it's enough to consider finite plays.)

```
type_synonym ('s2) play = ‹'s2 list›
type_synonym ('s2) strategy = ‹'s2 play ⇒ 's2›
type_synonym ('s2) posstrategy = ‹'s2 ⇒ 's2›

definition strategy_from_positional :: ‹'s posstrategy ⇒ 's strategy› where
  ‹strategy_from_positional pf = (λ play. pf (hd play))›

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

definition play_continuation :: ‹'s play ⇒ 's play ⇒ bool›
  where ‹play_continuation p1 p2 ≡ (drop (length p2 - length p1) p2) = p1›
```

— Plays for a given player 0 strategy

```
inductive_set plays_for_0strategy :: ‹'s strategy ⇒ 's ⇒ 's play set›
  for f initial where
  init: ‹[initial] ∈ plays_for_0strategy f initial› |
  p0move:
    ‹n0#play ∈ plays_for_0strategy f initial ⟹ player0_position n0 ⟹ n0 ⟼♡ f (n0#play)
    ⟹ (f (n0#play))#n0#play ∈ plays_for_0strategy f initial› |
  p1move:
    ‹n1#play ∈ plays_for_0strategy f initial ⟹ player1_position n1 ⟹ n1 ⟼♡ n1'
    ⟹ n1'#n1#play ∈ plays_for_0strategy f initial›

lemma strategy0_step:
  assumes
    ‹n0 # n1 # rest ∈ plays_for_0strategy f initial›
    ‹player0_position n1›
  shows
    ‹f (n1 # rest) = n0›
  ⟨proof⟩
inductive_set plays_for_1strategy :: ‹'s strategy ⇒ 's ⇒ 's play set›
  for f initial where
  init: ‹[initial] ∈ plays_for_1strategy f initial› |
  p0move:
    ‹n0#play ∈ plays_for_1strategy f initial ⟹ player0_position n0 ⟹ n0 ⟼♡ n0'
    ⟹ n0'#n0#play ∈ plays_for_1strategy f initial› |
  p1move:
    ‹n1#play ∈ plays_for_1strategy f initial ⟹ player1_position n1 ⟹ n1 ⟼♡ f (n1#play)
    ⟹ (f (n1#play))#n1#play ∈ plays_for_1strategy f initial›

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

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

```
definition sound_0strategy :: ‹'s strategy ⇒ 's ⇒ bool› where
  ‹sound_0strategy f initial ≡
    ∀ n0 play .
      n0#play ∈ plays_for_0strategy f initial ∧
      player0_position n0 ⟶ n0 ⟼♡ f (n0#play)›

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

lemma strategy0_plays_subset:
  assumes ‹play ∈ plays_for_0strategy f initial›
  shows ‹play ∈ plays initial›
  ⟨proof⟩
lemma strategy1_plays_subset:
  assumes ‹play ∈ plays_for_1strategy f initial›
  shows ‹play ∈ plays initial›
  ⟨proof⟩

lemma no_empty_plays:
  assumes ‹[] ∈ plays initial›
  shows ‹False›
  ⟨proof⟩
```

Player1 wins a play if the play has reached a deadlock where it's player0's turn

```
definition player1_wins_immediately :: <'s play ⇒ bool> where
  <player1_wins_immediately play ≡ player0_position (hd play) ∧ (∄ p' . (hd play) ⟼♡ p')>

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

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

lemma stuck_player0_win:
  assumes <player1_position initial> <(∄ p' . initial ⟼♡ p')>
  shows <player0_wins initial>
⟨proof⟩

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

end
end
```

3 Notions of Equivalence

3.1 Strong Simulation and Bisimulation

```
theory Strong_Relations
  imports Transition_Systems
begin

context lts
begin

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

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

lemma bisim_ruleformat:
  assumes <bisimulation R>
    and <R p q>
  shows
    <p ⟼a p' ⟹ (∃ q'. R p' q' ∧ (q ⟼a q'))>
    <q ⟼a q' ⟹ (∃ p'. R p' q' ∧ (p ⟼a p'))>
  ⟨proof⟩
```

```
end — context lts

end
```

3.2 Weak Simulation

```
theory Weak_Relations
imports
  Weak_Transition_Systems
  Strong_Relations
begin

context lts_tau
begin

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

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

```
definition weak_step_tau2 :: <'s ⇒ 'a ⇒ 's ⇒ bool>
    (<_ ⇒^^ _  _> [70, 70, 70] 80)
where [simp]:
  <(p ⇒^^ a q) ≡ p ⇒^a q>

coinductive greatest_weak_simulation ::
  <'s ⇒ 's ⇒ bool>
where
  <(∀ p' a. p ⟼a p' ⟶ (∃ q'. greatest_weak_simulation p' q' ∧ (q ⇒^^ a q')))
  ⟹ greatest_weak_simulation p q>

lemma weak_sim_ruleformat:
assumes <weak_simulation R>
  and <R p q>
shows
  <p ⟼a p' ⟹ ¬tau a ⟹ (∃ q'. R p' q' ∧ (q ⇒a q'))>
  <p ⟼a p' ⟹  tau a ⟹ (∃ q'. R p' q' ∧ (q ⟼* tau q'))>
  ⟨proof⟩

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)>
  ⟨proof⟩


lemma gws_is_weak_simulation:
  shows <weak_simulation greatest_weak_simulation>
  ⟨proof⟩
```

```
lemma weakly_sim_by_implies_gws:
  assumes ‹p ⊑ws q›
  shows ‹greatest_weak_simulation p q›
  ⟨proof⟩

lemma gws_eq_weakly_sim_by:
  shows ‹p ⊑ws q = greatest_weak_simulation p q›
  ⟨proof⟩

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'›
  ⟨proof⟩

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⟩

lemma weak_sim_enabled_subs:
  assumes
    ‹p ⊑ws q›
    ‹weak_enabled p a›
    ‹¬ tau a›
  shows ‹weak_enabled q a›
⟨proof⟩

lemma weak_sim_union_cl:
  assumes
    ‹weak_simulation RA›
    ‹weak_simulation RB›
  shows
    ‹weak_simulation (λ p q. RA p q ∨ RB p q)›
  ⟨proof⟩

lemma weak_sim_remove_dead_state:
  assumes
    ‹weak_simulation R›
    ‹⋀ a p . ¬ d ⟼a p ∧ ¬ p ⟼a d›
  shows
    ‹weak_simulation (λ p q . R p q ∧ q ≠ d)›
  ⟨proof⟩

lemma weak_sim_tau_step:
  ‹weak_simulation (λ p1 q1 . q1 ⟼* tau p1)›
  ⟨proof⟩

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*⟩

```
lemma weak_sim_trans:
  assumes
    ‹p ⊑ws pq›
    ‹pq ⊑ws q›
  shows
    ‹p ⊑ws q›
```
⟨*proof*⟩

```
lemma weak_sim_word_impl:
  fixes
    p q p' A
  assumes
    ‹weak_simulation R› ‹R p q› ‹p ⇒$ A  p'›
  shows
    ‹∃q'. R p' q' ∧ q ⇒$ A  q'›
```
⟨*proof*⟩

```
lemma weak_sim_word_impl_contra:
  assumes
    ‹∀ p q . R p q ⟶
      (∀ p' A. p ⇒$A p' ⟶ (∃ q'. R p' q' ∧ q ⇒$A q'))›
  shows
    ‹weak_simulation R›
```
⟨*proof*⟩

```
lemma weak_sim_word:
  ‹weak_simulation R =
    (∀ p q . R p q ⟶
      (∀ p' A. p ⇒$A p' ⟶ (∃ q'. R p' q' ∧ q ⇒$A q')))›
```
⟨*proof*⟩

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'))›
```
⟨*proof*⟩

```
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›
⟨proof⟩

lemma weak_bisim_invert:
  assumes
    ‹weak_bisimulation R›
  shows
    ‹weak_bisimulation (λ p q. R q p)›
⟨proof⟩

lemma bisim_weak_bisim:
  assumes ‹bisimulation R›
  shows ‹weak_bisimulation R›
  ⟨proof⟩

lemma weak_bisim_weak_sim:
  shows ‹weak_bisimulation R = (weak_simulation R ∧ weak_simulation (λ p q . R q p))›
⟨proof⟩

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'›
    ⟨proof⟩

lemma weak_bisim_union:
  assumes
    ‹weak_bisimulation R1›
    ‹weak_bisimulation R2›
  shows
    ‹weak_bisimulation (λ p q . R1 p q ∨ R2 p q)›
  ⟨proof⟩

lemma weak_bisim_taufree_strong:
  assumes
    ‹weak_bisimulation R›
    ‹⋀ p q a. p ⟼ a q ⟹ ¬ tau a›
  shows
    ‹bisimulation R›
  ⟨proof⟩
```

3.4 Trace Inclusion

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

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

lemma weak_trace_inlcusion_greatest:
  shows ‹trace_inclusion (λ p q . p ⊑T q)›
  ⟨proof⟩
```

3.5 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›
  ⟨proof⟩
```

3.6 Coupled Equivalences

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

lemma coupling_tau_max_symm:
  assumes
    ‹R p q ⟶ (∃ q'. q ⟼*tau q' ∧ R q' p)›
    ‹tau_max q›
    ‹R p q›
  shows
    ‹R q p›
  ⟨proof⟩

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›
  ⟨proof⟩

end — context lts_tau
end
```

4 Contrasimulation

```
theory Contrasimulation
imports
  Weak_Relations
begin

context lts_tau
begin
```

4.1 Definition of Contrasimulation

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

lemma contrasim_simpler_def:
  shows ‹contrasimulation R =
    (∀ p q p' A . R p q ∧ (p ⇒$ A p') ⟶ (∃ q'. (q ⇒$ A q') ∧ R q' p'))›
⟨proof⟩

abbreviation contrasimulated_by :: ‹'s ⇒ 's ⇒ bool› (‹_ ⊑c  _› [60, 60] 65)
  where ‹contrasimulated_by p q ≡ ∃ R . contrasimulation R ∧ R p q›

lemma contrasim_preorder_is_contrasim:
  shows ‹contrasimulation (λ p q . p ⊑c q)›
  ⟨proof⟩

lemma contrasim_preorder_is_greatest:
  assumes ‹contrasimulation R›
  shows ‹⋀ p q. R p q ⟹ p ⊑c q›
  ⟨proof⟩

lemma contrasim_tau_step:
  ‹contrasimulation (λ p1 q1 . q1 ⟼* tau p1)›
  ⟨proof⟩

lemma contrasim_trans_constructive:
  fixes R1 R2
  defines
    ‹R ≡ λ p q . ∃ pq . (R1 p pq ∧ R2 pq q) ∨ (R2 p pq ∧ R1 pq q)›
  assumes
    R1_def: ‹contrasimulation R1› ‹R1 p pq› and
    R2_def: ‹contrasimulation R2› ‹R2 pq q›
  shows
    ‹R p q› ‹contrasimulation R›
  ⟨proof⟩

lemma contrasim_trans:
  assumes
    ‹p ⊑c pq›
    ‹pq ⊑c q›
  shows
    ‹p ⊑c q›
  ⟨proof⟩
```

```
lemma contrasim_refl:
  shows
    ‹p ⊑c p›
  ⟨proof⟩

lemma contrasimilarity_equiv:
  defines ‹contrasimilarity ≡ λ p q. p ⊑c q ∧ q ⊑c p›
  shows ‹equivp contrasimilarity›
⟨proof⟩

lemma contrasim_implies_trace_incl:
  assumes ‹contrasimulation R›
  shows ‹trace_inclusion R›
⟨proof⟩

lemma contrasim_coupled:
  assumes
    ‹contrasimulation R›
    ‹R p q›
  shows
    ‹∃ q'. q ⟼* tau q' ∧ R q' p›
  ⟨proof⟩

lemma contrasim_taufree_symm:
  assumes
    ‹contrasimulation R›
    ‹R p q›
    ‹stable_state q›
  shows
    ‹R q p›
  ⟨proof⟩

lemma symm_contrasim_is_weak_bisim:
  assumes
    ‹contrasimulation R›
    ‹⋀ p q. R p q ⟹ R q p›
  shows
    ‹weak_bisimulation R›
  ⟨proof⟩

lemma contrasim_weakest_bisim:
  assumes
    ‹contrasimulation R›
    ‹⋀ p q a. p ⟼ a q ⟹ ¬ tau a›
  shows
    ‹bisimulation R›
  ⟨proof⟩

lemma symm_weak_sim_is_contrasim:
  assumes
    ‹weak_simulation R›
    ‹⋀ p q. R p q ⟹ R q p›
  shows
    ‹contrasimulation R›
  ⟨proof⟩
```

4.2 Intermediate Relation Mimicking Contrasim

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

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

lemma R_is_in_mimicking_of_R :
  assumes ‹R p Q›
  shows ‹mimicking R p Q›
  ⟨proof⟩

lemma mimicking_of_C_guarantees_tau_succ:
  assumes
    ‹contrasimulation C›
    ‹mimicking (set_lifted C) p Q›
    ‹p ⇒^τ p'›
  shows ‹∃q'. q' ∈ (weak_tau_succs Q) ∧ mimicking (set_lifted C) q' {p'}›
⟨proof⟩

lemma mimicking_of_C_guarantees_action_succ:
 assumes
    ‹contrasimulation C›
    ‹mimicking (set_lifted C) p Q›
    ‹p =▷a p'›
    ‹a ≠ τ›
  shows ‹mimicking (set_lifted C) p' (dsuccs a Q)›
⟨proof⟩
```

4.3 Over-Approximating Contrasimulation by a Single-Step Version

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

lemma contrasim_step_weaker_than_seq:
  assumes
    ‹contrasimulation R›
  shows
    ‹contrasim_step R›
  ⟨proof⟩

lemma contrasim_step_seq_coincide_for_sims:
  assumes
    ‹contrasim_step R›
    ‹weak_simulation R›
  shows
    ‹contrasimulation R›
  ⟨proof⟩
```

```
end
end
```

5 Coupled Simulation

```
theory Coupled_Simulation
  imports Contrasimulation
begin

context lts_tau
begin
```

5.1 Van Glabbeeks's Coupled Simulation

We mainly use van Glabbeek's coupled simulation from his 2017 CSP paper [7]. Later on, we will compare it to other definitions of coupled (delay/weak) simulations.

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

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

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

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

5.2 Position between Weak Simulation and Weak Bisimulation

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

```
lemma coupled_simulation_weak_simulation:
  ‹coupled_simulation R =
    (weak_simulation R ∧ (∀ p q . R p q ⟶ (∃ q'. q ⟼*tau q' ∧ R q' p)))›
  ⟨proof⟩

corollary coupled_simulation_implies_weak_simulation:
  assumes ‹coupled_simulation R›
  shows ‹weak_simulation R›
  ⟨proof⟩

corollary coupledsim_enabled_subs:
  assumes
    ‹p ⊑cs q›
    ‹weak_enabled p a›
    ‹¬ tau a›
  shows ‹weak_enabled q a›
  ⟨proof⟩
```

```
lemma coupled_simulation_implies_coupling:
  assumes
    ‹coupled_simulation R›
    ‹R p q›
  shows
    ‹∃ q'. q ⟼*tau q' ∧ R q' p›
  ⟨proof⟩

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›
⟨proof⟩
```

5.3 Coupled Simulation and Silent Steps

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

```
lemma coupledsim_step_gla17:
  ‹coupled_simulation (λ p1 q1 . q1 ⟼* tau p1)›
  ⟨proof⟩

corollary coupledsim_step:
  assumes
    ‹p ⟼* tau  q›
  shows
    ‹q ⊑cs p›
  ⟨proof⟩
```

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›
  ⟨proof⟩

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'›
  ⟨proof⟩

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))›
  ⟨proof⟩
```

5.4 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)›
  ⟨proof⟩

lemma coupledsim_refl:
  ‹p ⊑cs p›
  ⟨proof⟩

lemma coupledsim_trans:
  assumes
    ‹p ⊑cs pq›
    ‹pq ⊑cs q›
  shows
    ‹p ⊑cs q›
⟨proof⟩

interpretation preorder ‹λ p q. p ⊑cs q› ‹λ p q. p ⊑cs q ∧ ¬(q ⊑cs p)›
  ⟨proof⟩

lemma coupled_similarity_equivalence:
  ‹equivp (λ p q. p ≡cs q)›
⟨proof⟩

lemma coupledsim_tau_max_eq:
  assumes
    ‹p ⊑cs q›
    ‹tau_max q›
  shows  ‹p ≡cs q›
  ⟨proof⟩

corollary coupledsim_stable_eq:
  assumes
    ‹p ⊑cs q›
    ‹stable_state q›
  shows  ‹p ≡cs q›
  ⟨proof⟩
```

5.5 Coinductive Coupled Simulation Preorder

$\sqsubseteq_{cs}$ can also be characterized coinductively. $\sqsubseteq_{cs}$ 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›
  ⟨proof⟩
```

```
lemma gcs_is_coupled_simulation:
  shows ‹coupled_simulation greatest_coupled_simulation›
⟨proof⟩

lemma coupled_similarity_implies_gcs:
  assumes ‹p ⊑cs q›
  shows ‹greatest_coupled_simulation p q›
⟨proof⟩

lemma gcs_eq_coupled_sim_by:
  shows ‹p ⊑cs q = greatest_coupled_simulation p q›
⟨proof⟩

lemma coupled_sim_by_is_coupled_sim:
  shows
    ‹coupled_simulation (λ p q . p ⊑cs q)›
⟨proof⟩

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))›
⟨proof⟩
```

5.6 Coupled Simulation Join

The following lemmas reproduce Proposition 3 from [7] 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⟩

lemma coupledsim_choice_2:
  assumes
    ‹pqc ⊑cs q›
    ‹⋀ pq a . pqc ⟼a pq ⟷ (a = τ ∧ (pq = p ∨ pq = q))›
  shows
    ‹p ⊑cs q›
⟨proof⟩

lemma coupledsim_choice_join:
  assumes
    ‹⋀ pq a . pqc ⟼a pq ⟷ (a = τ ∧ (pq = p ∨ pq = q))›
  shows
    ‹p ⊑cs q ⟷ pqc ≡cs q›
  ⟨proof⟩
```

5.7 Coupled Delay Simulation

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

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

```
lemma coupledsim_tau_challenge_trivial:
  assumes
    ‹p ⊑cs q›
    ‹p ⟼* tau p'›
  shows
    ‹p' ⊑cs q›
  ⟨proof⟩

lemma coupled_similarity_s_delay_simulation:
  ‹delay_simulation (λ p q. p ⊑cs q)›
  ⟨proof⟩

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)›
  ⟨proof⟩
```

5.8 Relationship to Contrasimulation and Weak Simulation

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

```
lemma weak_sim_and_contrasim_implies_coupled_sim:
  assumes
    ‹contrasimulation R›
    ‹weak_simulation R›
  shows
    ‹coupled_simulation R›
  ⟨proof⟩

lemma coupledsim_implies_contrasim:
  assumes
    ‹coupled_simulation R›
  shows
    ‹contrasimulation R›
⟨proof⟩

lemma coupled_simulation_iff_weak_sim_and_contrasim:
  shows ‹coupled_simulation R ⟷ contrasimulation R ∧ weak_simulation R›
  ⟨proof⟩
```

5.9 τ-Reachability (and Divergence)

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

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

```
lemma taufree_coupledsim_symm:
```

```
  assumes
    ‹⋀ p1 a p2 . (p1 ⟼a p2 ⟹ ¬ tau a)›
    ‹coupled_simulation R›
    ‹R p q›
  shows ‹R q p›
```
⟨*proof*⟩

```
lemma taufree_coupledsim_weak_bisim:
  assumes
    ‹⋀ p1 a p2 . (p1 ⟼a p2 ⟹ ¬ tau a)›
    ‹coupled_simulation R›
  shows ‹weak_bisimulation R›
```
⟨*proof*⟩

```
lemma coupledsim_stable_state_symm:
  assumes
    ‹coupled_simulation R›
    ‹R p q›
    ‹stable_state q›
  shows
    ‹R q p›
```
⟨*proof*⟩

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*⟩

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'›
```
⟨*proof*⟩

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*⟩

5.10 On the Connection to Weak Bisimulation

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

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

```
lemma coupledsim_eventual_symmetry:
  assumes
    contracted_cycles: ‹⋀ r1 r2 . r1 ⟼* tau r2 ∧ r2 ⟼* tau r1 ⟹ r1 = r2› and
    finite_taus: ‹⋀ r. finite {r'. r ⟼* tau r'}› and
    cs: ‹p ⊑cs q› and
    step: ‹p ⇒^a p'› and
    tau_max_p': ‹tau_max p'›
  shows
    ‹∃ q'. tau_max q' ∧ q ⇒^a q' ∧ p' ≡cs q'›
⟨proof⟩
```

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⟩

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⟩

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⟩
```

5.11 Reduction Semantics Coupled Simulation

The tradition to describe coupled simulation as special delay/weak simulation is quite common for coupled simulations on reduction semantics as in [11, 6], of which [11] can also be found in the AFP [12]. The notions coincide (for systems just with τ-transitions).

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

lemma weak_bisim_implies_coupled_sim_gp15:
  assumes
    wbisim: ‹weak_bisimulation R› and
    symmetry: ‹⋀ p q . R p q ⟹ R q p›
  shows ‹coupled_simulation_gp15 R›
⟨proof⟩

lemma coupledsim_gla17_implies_gp15:
  assumes
    ‹coupled_simulation R›
  shows
    ‹coupled_simulation_gp15 R›
  ⟨proof⟩

lemma coupledsim_gp15_implies_gla17_on_tau_systems:
  assumes
    ‹coupled_simulation_gp15 R›
    ‹⋀ a . tau a›
  shows
    ‹coupled_simulation R›
  ⟨proof⟩
```

5.12 Coupled Simulation as Two Simulations

Historically, coupled similarity has been defined in terms of *two* weak simulations coupled in some way [14, 10]. We reproduce these (more well-known) formulations and show that they are equivalent to the coupled (delay) simulations we are using.

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

lemma weak_bisim_implies_coupled_sim_san12:
  assumes ‹weak_bisimulation R›
  shows ‹coupled_simulation_san12 R R›
  ⟨proof⟩

lemma coupledsim_gla17_resembles_san12:
  shows
    ‹coupled_simulation R1 =
    coupled_simulation_san12 R1 (λ p q . R1 q p)›
```

⟨*proof*⟩

```
lemma coupledsim_san12_impl_gla17:
  assumes
    ‹coupled_simulation_san12 R1 R2›
  shows
    ‹coupled_simulation (λ p q. R1 p q ∨ R2 q p)›
```
⟨*proof*⟩

5.13 S-coupled Simulation

Originally coupled simulation was introduced as two weak simulations coupled at the stable states. We give the definitions from [9, 10] and a proof connecting this notion to "our" coupled similarity in the absence of divergences following [14].

```
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›
```
⟨*proof*⟩

```
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)›
```
⟨*proof*⟩

```
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›
```

⟨*proof*⟩

```
corollary weak_bisim_implies_s_coupled_sim:
  assumes ‹weak_bisimulation R›
  shows ‹s_coupled_simulation_san12 R R›
  ⟨proof⟩

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)›
  ⟨proof⟩

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)›
  ⟨proof⟩

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)›
  ⟨proof⟩

lemma s_coupledsim_stable_eq:
  assumes
    ‹p ⊑scs q›
    ‹stable_state p›
  shows  ‹p ≡scs q›
⟨proof⟩

lemma s_coupledsim_symm:
  assumes
    ‹p ≡scs q›
  shows
    ‹q ≡scs p›
  ⟨proof⟩

lemma s_coupledsim_eq_parts:
  assumes
    ‹p ≡scs q›
  shows
    ‹p ⊑scs q›
    ‹q ⊑scs p›
  ⟨proof⟩
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›
  ⟨proof⟩
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›
```
⟨*proof*⟩

While this proof follows [14], 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*⟩

```
end
```
— context `lts_tau`

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

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

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

lemma no_root_coupled_sim:
  fixes R1 R2
  assumes
    coupled:
      ‹coupled_simulation_san12 R1 R2› and
    root:
      ‹R1 a0 b0› ‹R2 a0 b0›
  shows
    False
```
⟨*proof*⟩

```
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›
  ⟨proof⟩

end — ex_lts// example lts

end
```

6 Game for Coupled Similarity with Delay Formulation

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

6.1 The Coupled Simulation Preorder Game Using Delay Steps

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

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

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

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

```
  τ :: <'a>
begin
```

6.2 Coupled Simulation Implies Winning Strategy

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

lemma defender_preceded_by_attacker:
  assumes
    <n0 # play ∈ plays (AttackerNode p0 q0)>
    <cs_game_defender_node n0>
  shows
    <∃ p q . hd play = AttackerNode p q ∧ cs_game_moves (AttackerNode p q) n0>
    <play ≠ []>
⟨proof⟩

lemma defender_only_challenged_by_visible_actions:
  assumes
    <(DefenderStepNode a p q) # play ∈ plays (AttackerNode p0 q0)>
  shows
    <¬tau a>
  ⟨proof⟩

lemma strategy_from_coupleddsim_retains_coupledsim:
  assumes
    <R p0 q0>
    <coupled_delay_simulation R>
    <initial = AttackerNode p0 q0>
    <play ∈ plays_for_0strategy (strategy_from_coupleddsim R) initial>
  shows
    <hd play = AttackerNode p q ⟹ R p q>
    <length play > 1 ⟹ hd (tl play) = AttackerNode p q ⟹ R p q>
  ⟨proof⟩

lemma strategy_from_coupleddsim_sound:
  assumes
    <R p0 q0>
    <coupled_delay_simulation R>
    <initial = AttackerNode p0 q0>
  shows
    <sound_0strategy (strategy_from_coupleddsim R) initial>
  ⟨proof⟩

lemma coupleddsim_implies_winning_strategy:
  assumes
    <R p q>
    <coupled_delay_simulation R>
    <initial = AttackerNode p q>
  shows
    <player0_winning_strategy (strategy_from_coupleddsim R) initial>
  ⟨proof⟩
```

6.3 Winning Strategy Induces Coupled Simulation

```
lemma winning_strategy_implies_coupleddsim:
  assumes
    ‹player0_winning_strategy f initial›
    ‹sound_0strategy f initial›
  defines
    ‹R == λ p q . (∃ play ∈ plays_for_0strategy f initial. hd play = AttackerNode p q)›
  shows
    ‹coupled_delay_simulation R›
  ⟨proof⟩

theorem winning_strategy_iff_coupledsim:
  assumes
    ‹initial = AttackerNode p q›
  shows
    ‹(∃ f . player0_winning_strategy f initial ∧ sound_0strategy f initial)
    = p ⊑cs q›
⟨proof⟩

end
end
```

7 Fixed Point Algorithm for Coupled Similarity

7.1 The Algorithm

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

context lts_tau
begin

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

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

7.2 Correctness

```
lemma mono_fp_step:
  ‹mono fp_step›
⟨proof⟩

lemma fp_fp_step:
  assumes
```

```
    ‹R = fp_step R›
  shows
    ‹coupled_delay_simulation (λ p q. (p, q) ∈ R)›
  ⟨proof⟩

lemma gfp_fp_step_subset_gcs:
  shows ‹(gfp fp_step) ⊆ { (p,q) . greatest_coupled_simulation p q }›
  ⟨proof⟩

lemma fp_fp_step_gcs:
  assumes
    ‹R = { (p,q) . greatest_coupled_simulation p q }›
  shows
    ‹fp_step R = R›
  ⟨proof⟩

lemma gfp_fp_step_gcs: ‹gfp fp_step = { (p,q) . greatest_coupled_simulation p q }›
  ⟨proof⟩

end

context lts_tau_finite
begin
lemma gfp_fp_step_while:
  shows
    ‹gfp fp_step = fp_compute_cs›
  ⟨proof⟩

theorem coupled_sim_fp_step_while:
  shows ‹fp_compute_cs = { (p,q) . greatest_coupled_simulation p q }›
  ⟨proof⟩

end

end
```

8 The Contrasimulation Preorder Word Game

```
theory Contrasim_Word_Game
imports
  Simple_Game
  Contrasimulation
begin

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

fun (in lts_tau) c_word_game_moves ::
  ‹('s, 'a) c_word_game_node ⇒ ('s, 'a) c_word_game_node ⇒ bool› where

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

  simulation_answer:
```

```
    ‹c_word_game_moves (DefenderNode A p1 q0) (AttackerNode q1 p10) =
      (q0 ⇒$A q1 ∧ p1 = p10)› |

  c_word_game_moves_no_step:
    ‹c_word_game_moves _ _ = False›

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

8.1 Contrasimulation Implies Winning Strategy in Word Game (Completeness)

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

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

lemma cwg_atknodes_precede_defnodes_in_plays:
  assumes
    ‹c_word_game_defender_node n0›
    ‹n0 = DefenderNode A p' q›
    ‹(n0#play) ∈ plays initial›
    ‹initial = AttackerNode p0 q0›
  shows ‹∃p. (hd play) = AttackerNode p q ∧ c_word_game_moves (hd play) n0›
⟨proof⟩

lemma cwg_second_play_elem_in_play_set :
  assumes
    ‹(n0#play) ∈ plays initial›
    ‹initial = AttackerNode p0 q0›
    ‹n0 = DefenderNode A p q›
  shows ‹hd play ∈ set (n0 # play)›
⟨proof⟩

lemma cwg_contrasim_contains_all_strat_consistent_atknodes:
  assumes
    ‹contrasimulation R›
    ‹R p0 q0›
    ‹initial = AttackerNode p0 q0›
    ‹play ∈ plays_for_0strategy (strategy_from_contrasim R) initial›
shows ‹((AttackerNode p q) ∈ set play) ⟹ R p q›
  ⟨proof⟩

lemma contrasim_word_game_complete:
  assumes
    ‹contrasimulation R›
    ‹R p q›
```

```
    ‹initial = AttackerNode p q›
  shows ‹player0_winning_strategy (strategy_from_contrasim R) initial›
    ⟨proof⟩
```

8.2 Winning Strategy Implies Contrasimulation in Word Game (Soundness)

```
lemma cwg_strategy_from_contrasim_sound:
  assumes
    ‹R p0 q0›
    ‹contrasimulation R›
    ‹initial = AttackerNode p0 q0›
  shows
    ‹sound_0strategy (strategy_from_contrasim R) initial›
  ⟨proof⟩

lemma contrasim_word_game_sound:
  assumes
    ‹player0_winning_strategy f initial›
    ‹sound_0strategy f initial›
  defines
    ‹R == λ p q . (∃ play ∈ plays_for_0strategy f initial. hd play = AttackerNode p q)›
  shows
    ‹contrasimulation R›  ⟨proof⟩

theorem winning_strategy_in_c_word_game_iff_contrasim:
  assumes
    ‹initial = AttackerNode p q›
  shows
    ‹(∃ f . player0_winning_strategy f initial ∧ sound_0strategy f initial)
    = (∃ C. contrasimulation C ∧ C p q)›
⟨proof⟩

end
end
```

9 The Contrasimulation Preorder Set Game

```
theory Contrasim_Set_Game
imports
  Simple_Game
  Contrasimulation
begin

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

fun (in lts_tau) c_set_game_moves ::
  ‹('s, 'a) c_set_game_node ⇒ ('s, 'a) c_set_game_node ⇒ bool› where

  simulation_challenge:
   ‹c_set_game_moves (AttackerNode p Q) (DefenderSimNode a p1 Q0) =
     (p =▷a p1 ∧ Q = Q0 ∧ ¬ tau a)› |
```

```
  simulation_answer:
    ‹c_set_game_moves (DefenderSimNode a p1 Q) (AttackerNode p10 Q1) =
      (p1 = p10 ∧ Q1 = dsuccs a Q)› |

  swap_challenge:
    ‹c_set_game_moves (AttackerNode p Q) (DefenderSwapNode p1 Q0) =
     (p ⇒^τ p1 ∧ Q = Q0)› |

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

  c_set_game_moves_no_step:
    ‹c_set_game_moves _ _ = False›

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

9.1 Contrasimulation Implies Winning Strategy in Set Game (Completeness)

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

fun strategy_from_mimicking_of_C ::
    ‹('s ⇒ ('s set) ⇒ bool) ⇒ ('s, 'a) c_set_game_node strategy›
  where

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

    ‹strategy_from_mimicking_of_C R ((DefenderSimNode a p1 Q)#play) =
      (AttackerNode p1 (SOME Q1 . Q1 = dsuccs a Q ∧ R p1 Q1))› |

    ‹strategy_from_mimicking_of_C _ _ = undefined›

lemma csg_atknodes_precede_defnodes_in_plays:
  assumes
    ‹c_set_game_defender_node n0›
    ‹(n0#play) ∈ plays (AttackerNode p0 Q0)›
  shows ‹∃p Q. (hd play) = AttackerNode p Q ∧ c_set_game_moves (hd play) n0›
⟨proof⟩

lemma csg_second_play_elem_in_play_set:
  assumes
    ‹(n0#play) ∈ plays (AttackerNode p0 Q0)›
    ‹c_set_game_defender_node n0›
  shows
    ‹hd play ∈ set (n0 # play)›
⟨proof⟩
```

```
lemma csg_only_defnodes_move_to_atknodes:
  assumes
    ‹c_set_game_moves n0 n1›
    ‹n1 = AttackerNode p Q›
  shows
    ‹(∃Qpred a. n0 = (DefenderSimNode a p Qpred)) ∨
    (∃q Ppred. n0 = (DefenderSwapNode q Ppred) ∧ Q = {q})›
⟨proof⟩

lemma c_set_game_strategy_retains_mimicking:
  assumes
    ‹contrasimulation C›
    ‹C p0 q0›
    ‹play ∈ plays_for_0strategy
      (strategy_from_mimicking_of_C (mimicking (set_lifted C))) (AttackerNode p0 {q0})›
  shows
    ‹n = AttackerNode p Q ⟹ n ∈ set play ⟹ mimicking (set_lifted C) p Q ›
⟨proof⟩

lemma contrasim_set_game_complete:
  assumes
    ‹contrasimulation C›
    ‹C p0 q0›
  shows
    ‹player0_winning_strategy (strategy_from_mimicking_of_C
      (mimicking (set_lifted C))) (AttackerNode p0 {q0})›
  ⟨proof⟩

lemma csg_strategy_from_mimicking_of_C_sound:
  assumes
    ‹contrasimulation C›
    ‹C p0 q0›
  shows
    ‹sound_0strategy
      (strategy_from_mimicking_of_C (mimicking (set_lifted C)))
      (AttackerNode p0 {q0})›
  ⟨proof⟩
```

9.2 Winning Strategy Implies Contrasimulation in Set Game (Soundness)

```
lemma csg_move_defsimnode_to_atknode:
  assumes
    ‹c_set_game_moves (DefenderSimNode a p Q) n0›
  shows
    ‹n0 = AttackerNode p (dsuccs a Q)›
⟨proof⟩

lemma csg_move_defswapnode_to_atknode:
  assumes
    ‹c_set_game_moves (DefenderSwapNode p' Q) n0›
  shows
    ‹∃q'. n0 = AttackerNode q' {p'} ∧ q' ∈ weak_tau_succs Q›
⟨proof⟩

lemma csg_defsimnode_never_stuck:
```

```
  assumes ‹n0 = DefenderSimNode a p Q›
  shows ‹∃Q'. c_set_game_moves n0 (AttackerNode p Q')›
```
⟨*proof*⟩

```
lemma csg_defender_can_simulate_prefix:
  assumes
    ‹A ≠ []›
    ‹p ⇒$A p1›
    ‹∀a∈set A. a ≠ τ›
    ‹sound_0strategy f (AttackerNode p00 {q00})›
    ‹play ∈ plays_for_0strategy f (AttackerNode p00 {q00})›
    ‹hd play = AttackerNode p {q}›
  shows
    ‹∃play p0.
      ((DefenderSimNode (last A) p0 (dsuccs_seq_rec (rev (butlast A)) {q}))#play)
        ∈ plays_for_0strategy f (AttackerNode p00 {q00})
      ∧ word_reachable_via_delay A p p0 p1›
```
⟨*proof*⟩

```
lemma contrasim_set_game_sound:
  assumes
    ‹player0_winning_strategy f (AttackerNode p00 {q00})›
    ‹sound_0strategy f (AttackerNode p00 {q00})›
  defines
    ‹C == λ p q . (∃ play ∈ plays_for_0strategy f (AttackerNode p00 {q00}) .
      hd play = AttackerNode p {q} ∧ (hd play = (AttackerNode p00 {q00})
      ∨ (∃P. hd (tl play) = DefenderSwapNode q P)))›
  shows
    ‹contrasimulation C›
```
⟨*proof*⟩

```
theorem winning_strategy_in_c_set_game_iff_contrasim:
  shows
    ‹(∃ f . player0_winning_strategy f (AttackerNode p0 {q0})
      ∧ sound_0strategy f (AttackerNode p0 {q0}))
    = p0 ⊑c q0›
```
⟨*proof*⟩

```
end
end
```

10 Infinitary Hennessy–Milner Logic

```
theory HM_Logic_Infinitary
  imports
    Weak_Relations
begin

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

— The HML formulation is derived from that by Max Pohlmann [13].

10.1 Satisfaction Relation

```
context lts_tau
begin

function satisfies :: ‹'s ⇒ ('a, 's) HML_formula ⇒ bool›
  (‹_ ⊨ _› [50, 50] 50)
  where
    ‹(p ⊨ HML_true) = True›
  | ‹(p ⊨ HML_conj I F) = (∀ i ∈ I. p ⊨ (F i))›
  | ‹(p ⊨ HML_neg φ) = (¬ p ⊨ φ)›
  | ‹(p ⊨ HML_poss α φ) =
      (∃ p'. ((tau α ∧ p ⟼* tau p') ∨ (¬ tau α ∧ p ⟼α p')) ∧ p' ⊨ φ)›
  ⟨proof⟩

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

lemma HML_wf_rel_is_wf: ‹wf HML_wf_rel›
  ⟨proof⟩

termination satisfies ⟨proof⟩

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

lemma HML_direct_subformulas_wf: ‹wf HML_direct_subformulas›
  ⟨proof⟩

definition HML_subformulas where ‹HML_subformulas ≡ (HML_direct_subformulas)⁺›

lemma HML_subformulas_wf: ‹wf HML_subformulas›
  ⟨proof⟩

lemma conj_only_depends_on_indexset:
  assumes ‹∀i∈I. f1 i = f2 i›
  shows ‹(p ⊨ HML_conj I f1) = (p ⊨ HML_conj I f2)›
  ⟨proof⟩
```

10.2 Distinguishing Formulas

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

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

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

```
lemma distinguishing_formula:
  assumes ‹¬ HML_equivalent p q›
  shows ‹∃ φ. p ⊨ φ ∧ ¬ q ⊨ φ›
  ⟨proof⟩

lemma HML_equivalent_symm:
  assumes ‹HML_equivalent p q›
  shows ‹HML_equivalent q p›
  ⟨proof⟩
```

10.3 Weak-NOR Hennessy–Milner Logic

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

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

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

lemma weak_backwards_truth:
  assumes
    ‹φ ∈ HML_weak_formulas›
    ‹p ⟼* tau  p'›
    ‹p' ⊨ φ›
  shows
    ‹p ⊨ φ›
  ⟨proof⟩

lemma tau_a_obs_implies_delay_step:
  assumes ‹p  ⊨ ⟨τ⟩⟨a⟩φ›
  shows ‹∃p'. p =▷a p' ∧ p' ⊨ φ›
⟨proof⟩

lemma delay_step_implies_tau_a_obs:
  assumes
    ‹p =▷a p'›
    ‹p' ⊨ φ›
  shows ‹p  ⊨ ⟨τ⟩⟨a⟩φ›
⟨proof⟩

end
end
```

11 Weak HML and the Contrasimulation Set Game

```
theory Weak_HML_Contrasimulation
  imports
    Contrasim_Set_Game
    HM_Logic_Infinitary
begin
```

11.1 Distinguishing Formulas at Winning Attacker Positions

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

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

```
function attack_formula :: ‹('s, 'a) c_set_game_node ⇒ ('a,'s) HML_formula› where
  ‹attack_formula (AttackerNode p Q) =
    (if (AttackerNode p Q) ∈ attacker_winning_region
      then attack_formula (strat (AttackerNode p Q))
      else HML_true)›
| ‹attack_formula (DefenderSimNode a p Q) =
    (if (DefenderSimNode a p Q) ∈ attacker_winning_region
      then ⟨τ⟩⟨a⟩(attack_formula (AttackerNode p (dsuccs a Q)))
      else HML_true)›
| ‹attack_formula (DefenderSwapNode p Q) =
    (if Q = {} ∨ DefenderSwapNode p Q ∉ attacker_winning_region
      then HML_true
      else (HML_weaknor (weak_tau_succs Q)
        (λq. if q ∈ (weak_tau_succs Q)
              then (attack_formula (AttackerNode q {p}))
              else HML_true )))›
  ⟨proof⟩

termination attack_formula
  ⟨proof⟩

lemma attacker_defender_switch:
  assumes
    ‹(AttackerNode p Q) ∈ attacker_winning_region›
  shows
    ‹(∃a p'. (strat (AttackerNode p Q)) = (DefenderSimNode a p' Q) ∧ p  =▷a p' ∧ ¬tau a)
    ∨(∃p'. (strat (AttackerNode p Q)) = (DefenderSwapNode p' Q) ∧ p ⟼* tau p' )›
  ⟨proof⟩

lemma attack_options:
```

```
  assumes
    ‹(AttackerNode p Q) ∈ attacker_winning_region›
  shows
    ‹(∃a p'. p =▷a p' ∧ ¬tau a ∧ strat (AttackerNode p Q) = (DefenderSimNode a p' Q) ∧
      attack_formula (AttackerNode p Q)
      = ⟨τ⟩⟨a⟩(attack_formula (AttackerNode p' (dsuccs a Q))))
    ∨ (∃p'. p ⟼* tau p' ∧ strat (AttackerNode p Q) = (DefenderSwapNode p' Q) ∧
      attack_formula (AttackerNode p Q) =
      (HML_weaknor (weak_tau_succs Q) (λq.
        if q ∈ (weak_tau_succs Q)
          then (attack_formula (AttackerNode q {p'}))
          else HML_true )))
    ∨ (Q = {} ∧ attack_formula (AttackerNode p Q) = HML_true)›
⟨proof⟩

lemma distinction_soundness:
  fixes p Q p0 Q0
  defines
    ‹pQ == AttackerNode p Q›
  defines
    ‹φ == attack_formula pQ›
  assumes
    ‹pQ ∈ attacker_winning_region›
  shows
    ‹p ⊨ φ ∧ (∀q∈Q. ¬ q ⊨ φ)›
  ⟨proof⟩

lemma distinction_in_language:
  fixes p Q
  defines
    ‹pQ == AttackerNode p Q›
  defines
    ‹φ == attack_formula pQ›
  assumes
    ‹pQ ∈ attacker_winning_region›
  shows
    ‹φ ∈ HML_weak_formulas›
  ⟨proof⟩

end
```

11.2 Attacker Wins on Pairs with Distinguishing Formulas

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

inductive_set attacker_winning_region :: ‹('s, 'a) c_set_game_node set› where
  Base: ‹DefenderSwapNode _ {} ∈ attacker_winning_region› |
  Atk: ‹(c_set_game_moves (AttackerNode p Q) g' ∧ g' ∈ attacker_winning_region)
    ⟹ (AttackerNode p Q) ∈ attacker_winning_region› |
  Def: ‹c_set_game_defender_node g ⟹
    (⋀g'. c_set_game_moves g g' ⟹ g' ∈ attacker_winning_region)
    ⟹ g ∈ attacker_winning_region›
```

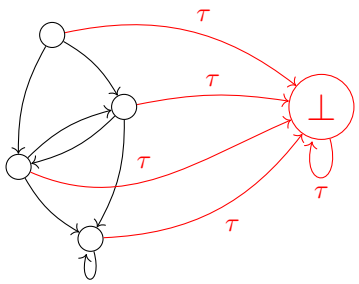

Figure 2: Example of a τ-sink extension with the original transition system in black and the extension in red.

```
lemma attacker_wins_if_defender_set_empty:
  assumes
    ‹Q = {}›
  shows
    ‹AttackerNode p Q ∈ attacker_winning_region›
⟨proof⟩

lemma attacker_wr_propagation:
  assumes
    ‹AttackerNode p' (dsuccs a Q) ∈ attacker_winning_region›
    ‹p =▷a p'›
    ‹¬tau a›
  shows
    ‹AttackerNode p Q ∈ attacker_winning_region›
⟨proof⟩

lemma distinction_completeness:
  assumes
    ‹φ ∈ HML_weak_formulas›
    ‹distinguishes_from_set φ p Q›
  shows
    ‹(AttackerNode p Q) ∈ attacker_winning_region›
⟨proof⟩

end
end
```

12 Reductions and τ-sinks

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

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

```
theory Tau_Sinks
imports
  Coupled_Simulation
begin
```

12.1 τ-Sink Properties

```
context lts_tau
begin

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

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

```
lemma tau_sink_maximal:
  assumes <tau_sink sink>
  shows
    <tau_max sink>
    <(p ⟼* tau sink)>
  ⟨proof⟩

lemma sink_has_no_word_transitions:
  assumes
    <tau_sink sink>
    <A ≠ []>
    <∀ a ∈ set(A). a ≠ τ>
  shows <∄s'. sink ⇒$A s'>
⟨proof⟩
```

12.2 Contrasimulation Equals Weak Simulation on τ-Sink Systems

```
lemma sink_coupled_simulates_all_states:
  assumes
    <⋀ p . (p ⟼* tau sink)>
  shows
    <sink ⊑cs p>
  ⟨proof⟩

theorem coupledsim_weaksim_equiv_on_sink_expansion:
  assumes
    <⋀ p . (p ⟼* tau sink)>
  shows
    <p ⊑ws q ⟷ p ⊑cs q>
  ⟨proof⟩
```

12.3 Contrasimulation Equals Weak Trace Inclusion on τ-Sink Systems

```
lemma sink_contrasimulates_all_states:
fixes A :: " 'a list"
  assumes
    <tau_sink sink>
    <⋀ p . (p ⟼* tau sink)>
  shows
```

```
    ‹∀ p. sink ⊑c p›
⟨proof⟩

lemma sink_trace_includes_all_states:
  assumes
    ‹⋀ p . (p ⟼* tau sink)›
  shows
    ‹sink ⊑T p›
  ⟨proof⟩

lemma trace_incl_with_sink_is_contrasim:
  assumes
    ‹⋀ p . (p ⟼* tau sink)›
    ‹⋀ p . R sink p›
    ‹trace_inclusion R›
  shows
    ‹contrasimulation R›
 ⟨proof⟩

theorem contrasim_trace_incl_equiv_on_sink_expansion_R:
  assumes
    ‹⋀ p . (p ⟼* tau sink)›
    ‹⋀ p . R sink p›
  shows
    ‹contrasimulation R = trace_inclusion R›
⟨proof⟩

theorem contrasim_trace_incl_equiv_on_sink_expansion:
  assumes
    ‹⋀ p . (p ⟼* tau sink)›
  shows
    ‹p ⊑T q ⟷ p ⊑c q›
  ⟨proof⟩

end
```

12.4 Weak Simulation Invariant under τ-Sink Extension

```
lemma simulation_tau_sink_1:
  fixes
    step sink R τ
  defines
    ‹step2 ≡ λ p1 a p2 . (p1 ≠ sink ∧ a = τ ∧ p2 = sink) ∨ step p1 a p2›
  assumes
    ‹⋀ a p . ¬ step sink a p›
    ‹lts_tau.weak_simulation step τ R›
  shows
    ‹lts_tau.weak_simulation step2 τ (λ p q. p = sink ∨ R p q)›
⟨proof⟩

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⟩

lemma simulation_sink_invariant:
  fixes
    step sink R τ
  defines
    ‹step2 ≡ λ p1 a p2 . (p1 ≠ sink ∧ a =  τ ∧ p2 = sink) ∨ step p1 a p2›
  assumes
    ‹⋀ a p . ¬ step sink a p ∧ ¬ step p a sink›
  shows ‹lts_tau.weakly_simulated_by step2 τ p q = lts_tau.weakly_simulated_by step τ p q›
⟨proof⟩
```

12.5 Trace Inclusion Invariant under τ-Sink Extension

```
lemma trace_inclusion_sink_invariant:
  fixes
    step sink R τ
  defines
    ‹step2 ≡ λ p1 a p2 . (p1 ≠ sink ∧ a =  τ ∧ p2 = sink) ∨ step p1 a p2›
  assumes
    ‹⋀ a p . ¬ step sink a p ∧ ¬ step p a sink›
  shows
    ‹lts_tau.weakly_trace_included_by step2 τ p q
    = lts_tau.weakly_trace_included_by step τ p q›
⟨proof⟩

end
```

References

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

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

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

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

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

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

[7] van Glabbeek, R.: A branching time model of CSP. In: Gibson-Robinson, T., Hopcroft, P., Lazić, R. (eds.) Concurrency, Security, and Puzzles: Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday, pp. 272–293. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-51046-0_14

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

[9] 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

[10] 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

[11] 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

[12] Peters, K., van Glabbeek, R.J.: 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

[13] Pohlmann, M.: Reducing Reactive to Strong Bisimilarity. Bachelor's thesis, Technische Universität Berlin (2021), https://maxpohlmann.github.io/Reducing-Reactive-to-Strong-Bisimilarity/

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