A Weak Spectroscopy Game to Characterize Behavioral Equivalences

Lisa Annett Barthel Benjamin Bisping Leonard Moritz Hübner
Caroline Lemke Karl Parvis Philipp Mattes Lenard Mollenkopf

January 3, 2025

Abstract

We provide an Isabelle/HOL formalization of Bisping and Jansen's [1] weak spectroscopy game, which can be used to characterize a range of behavioral equivalences simultaneously, spanning from stability-respecting branching bisimilarity to weak trace equivalence. We relate distinguishing sublanguages of Hennessy-Milner Logic and attacker-winning budgets in an energy game by an eight-dimensional measure of formula expressiveness.

Contents

1 Introduction

Verification and asking wether a model fulfills its specification or if a program can be replaced with one that has the same behaviour are core problems of reactive systems and programming. For this we have to get an idea of what same behaviour for processes actually means and consider different behavioural equivalences. One possibility for this consideration are games where one player winning the game corresponds to the behavioural equivalences of processes. Alternatively, we could use a modal logic, known as Hennessy-Milner Logic (HML), not only to express the specification but also to build formulas that distinguish processes and thereby characterize behavioural equivalences. These techniques for checking whether two processes have the same behaviour may also be combined.

Previously, it was only possible to decide equivalence problems individually, but recently there have been ideas of deciding many of these problems at once. Therefore, Bisping and Jansen [1] included a measure of expressiveness for HML_{SRBB} formulas as eight-dimensional vectors. These vectors are added as costs to the moves of an extended delay bisimulation game such that the following property is obtained: The attacker wins a play with a certain initial energy e if and only if there is a formula that distinguishes the corresponding processes with a price less than or equal to e. Then the initial energy and the price of a formula encode the satisfied behavioural equivalences. It is therefore possible to decide for a whole spectrum of behavioural equivalences at the same time which of them apply [1].

We formalize the eight-dimensional weak spectroscopy game, which "can be used to decide a wide array of behavi[ou]ral equivalences between stability-respecting branching bisimilarity and weak trace equivalence in one go"[1, Abstract]. We then outline the proof of the correspondence between "attacker-winning energy budgets and distinguishing sublanguages of Hennessy-Milner [L]ogic characterized by eight dimensions of formula expressiveness"[1, Abstract]. With our formalization we try to follow [1] as closely as possible in how we formalize the weak spectroscopy game, HML and their correspondence. In doing so, we point out deviations in our formalization from and small corrections of the paper. This report documents the outcome of a project supervised by Benjamin Bisping at the Technical University of Berlin.

First, we formalize labelled transition systems with special handeling of τ-transitions. Afterwards, we describe our formalization of HML and a subset of HML, which we denote HML_{SRBB}. Within these HML sections, we define the semantics of such formulas and based on this prove several implications and equivalences on HML formulas. Additionally, we treat the notion of distinguishing formulas and especially distinguishing conjunctions. In the following sections, we present our formalization of energies as a data type and a price function for formulas. Before we formalize the weak spectroscopy game, we do the same for its basis in the form of energy games and define winning budgets on them. Following these fundamentals, we state our formalization of the theorem 1 of [1], that "relate[s] attacker-winning energy budgets and distinguishing sublanguages of Hennessy-Milner [L]ogic"[1, Abstract]. Based on the proof in [1] we outline a proof for this theorem through three lemmas. The first lemma states that given a distinguishing formula, the attacker is able to win the corresponding weak spectroscopy game. After introducing strategy formulas, we use induction to prove the second lemma, which claimes that if the attacker wins the weak spectroscopy game with an initial energy e, then there exists a (strategy) formula with a price less than or equal to e. Afterwards, the third lemma completes this cycle by stating that if there is a (strategy) formula, then it is a distingushing formula. Finally, we discuss the minor issues we found in the paper and thus present our contributions to [1] and end this report with a conclusion.

2 LTS

```
theory LTS
  imports Main
begin
```

2.1 Labelled Transition Systems

The locale `LTS` represents a labelled transition system consisting of a set of states $\mathcal{P}$, a set of actions Σ, and a transition relation $\mapsto \subseteq \mathcal{P} \times \Sigma \times \mathcal{P}$ (cf. [1, defintion 1]). We formalize the sets of states and actions by the type variables `'s` and `'a`. An LTS is then determined by the transition relation `step`. Due to technical limitations we use the notation `p ↦α p'` which has same meaing as $p \xrightarrow{\alpha} p'$ has in [1].

```
locale LTS =
  fixes step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ↦ _ _› [70,70,70] 80)
begin
```

One may lift `step` to sets of states, written as `P ↦S α Q`. We define `P ↦S α Q` to be true if and only if for all states `q` in `Q` there exists a state `p` in `P` such that `p ↦ α q` and for all `p` in `P` and for all `q`, if `p ↦ α q` then `q` is in `Q`.

```
abbreviation step_setp (‹_ ↦S _ _› [70,70,70] 80) where
  ‹P ↦S α Q ≡ (∀q ∈ Q. ∃p ∈ P. p ↦ α q) ∧ (∀p ∈ P. ∀q. p ↦ α q ⟶ q ∈ Q)›
```

The set of possible α-steps for a set of states `P` are all `q` such that there is a state `p` in `P` with `p ↦ α q`.

```
definition step_set :: ‹'s set ⇒ 'a ⇒ 's set› where
  ‹step_set P α ≡ { q . ∃p ∈ P. p ↦ α q }›
```

The set of possible α-steps for a set of states `P` is an instance of `step` lifted to sets of steps.

```
lemma step_set_is_step_set: ‹P ↦S α (step_set P α)›
  using step_set_def by force
```

For a set of states `P` and an action α there exists exactly one `Q` such that `P ↦S α Q`.

```
lemma exactly_one_step_set: ‹∃!Q. P ↦S α Q›
proof -
  from step_set_is_step_set
  have ‹P ↦S α (step_set P α)›
    and ‹⋀Q. P ↦S α Q ⟹ Q = (step_set P α)›
    by fastforce+
  then show ‹∃!Q. P ↦S α Q›
    by blast
qed
```

The lifted `step` (`P ↦S α Q`) is therefore this set `Q`.

```
lemma step_set_eq:
  assumes ‹P ↦S α Q›
  shows ‹Q = step_set P α›
  using assms step_set_is_step_set exactly_one_step_set by fastforce

end
```

2.2 Labelled Transition Systems with Silent Steps

We formalize labelled transition systems with silent steps as an extension of ordinary labelled transition systems with a fixed silent action τ.

```
locale LTS_Tau =
  LTS step
    for step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ↦ _ _› [70,70,70] 80) +
    fixes τ :: 'a
```

```
begin
```

The paper introduces a transition $p \xrightarrow{(\alpha)} p'$ if $p \xrightarrow{\alpha} p'$, or if $\alpha = \tau$ and $p = p'$ (cf. [1, defintion 2]). We define `soft_step` analagously and provide the notation `p ↦a α p'`.

```
abbreviation soft_step (‹_ ↦a _ _› [70,70,70] 80) where
  ‹p ↦a α q ≡ p ↦α q ∨ (α = τ ∧ p = q)›
```

A state `p` is `silent_reachable`, represented by the symbol ↠, from another state `p'` iff there exists a path of τ-transitions. from `p'` to `p`.

```
inductive silent_reachable :: ‹'s ⇒ 's ⇒ bool›  (infix ‹↠› 80)
  where
    refl: ‹p ↠ p› |
    step: ‹p ↠ p''› if ‹p ↦ τ p'› and ‹p' ↠ p''›
```

If `p'` is silent reachable from `p` and there is a τ-transition from `p'` to `p''` then `p''` is silent reachable from `p`.

```
lemma silent_reachable_append_τ: ‹p ↠ p' ⟹ p' ↦ τ p'' ⟹ p ↠ p''›
proof (induct rule: silent_reachable.induct)
  case (refl p)
  then show ?case using silent_reachable.intros by blast
next
  case (step p p' p'')
  then show ?case using silent_reachable.intros by blast
qed
```

The relation (↠) is transitive.

```
lemma silent_reachable_trans:
  assumes
    ‹p ↠ p'›
    ‹p' ↠ p''›
  shows
    ‹p ↠ p''›
using assms silent_reachable.intros(2)
  by (induct, blast+)
```

The relation `silent_reachable_loopless` is a variation of (↠) that does not use self-loops.

```
inductive silent_reachable_loopless :: ‹'s ⇒ 's ⇒ bool›  (infix ‹↠L› 80)
  where
    ‹p ↠L p› |
    ‹p ↠L p''› if ‹p ↦ τ p'› and ‹p' ↠L p''› and ‹p ≠ p'›
```

If a state `p'` is (↠) from `p` it is also (↠L).

```
lemma silent_reachable_impl_loopless:
  assumes ‹p ↠ p'›
  shows ‹p ↠L p'›
  using assms
proof(induct rule: silent_reachable.induct)
  case (refl p)
  thus ?case by (rule silent_reachable_loopless.intros(1))
next
  case (step p p' p'')
  thus ?case proof(cases ‹p = p'›)
    case True
    thus ?thesis using step.hyps(3) by auto
  next
    case False
    thus ?thesis using step.hyps silent_reachable_loopless.intros(2) by blast
  qed
qed
```

```
lemma tau_chain_reachabilty:
  assumes ‹∀i < length pp - 1.  pp!i ↦ τ pp!(Suc i)›
  shows ‹∀j < length pp. ∀i ≤ j. pp!i ⟶⟫ pp!j›
proof safe
  fix j i
  assume ‹j < length pp› ‹i ≤ j›
  thus ‹pp!i ⟶⟫ pp!j›
  proof (induct j)
    case 0
    then show ?case
      using silent_reachable.refl by blast
  next
    case (Suc j)
    then show ?case
    proof (induct i)
      case 0
      then show ?case using assms silent_reachable_append_τ
        by (metis Suc_lessD Suc_lessE bot_nat_0.extremum diff_Suc_1)
    next
      case (Suc i)
      then show ?case using silent_reachable.refl assms silent_reachable_append_τ
        by (metis Suc_lessD Suc_lessE  diff_Suc_1 le_SucE)
    qed
  qed
qed
```

In the following, we define `weak_step` as a new notion of transition relation between states. A state `p` can reach `p'` by performing an α-transition, possibly proceeded and followed by any number of τ-transitions.

```
definition weak_step (‹_ ⟶⟫↦⟶⟫ _ _› [70, 70, 70] 80) where
  ‹p  ⟶⟫↦⟶⟫ α p' ≡ if α = τ
                  then p ⟶⟫ p'
                  else ∃p1 p2. p ⟶⟫ p1 ∧ p1 ↦ α p2 ∧ p2 ⟶⟫ p'›

lemma silent_prepend_weak_step: ‹p ⟶⟫ p' ⟹ p' ⟶⟫↦⟶⟫ α p'' ⟹ p ⟶⟫↦⟶⟫ α p''›
proof (cases ‹α = τ›)
  case True
  assume ‹p ⟶⟫ p'›
     and ‹p' ⟶⟫↦⟶⟫ α p''›
     and ‹α = τ›
  hence ‹p' ⟶⟫↦⟶⟫ τ p''› by auto
  then have ‹p' ⟶⟫ p''› unfolding weak_step_def by auto
  with ‹p ⟶⟫ p'›
  have ‹p ⟶⟫ p''› using silent_reachable_trans
    by blast
  then have ‹p ⟶⟫↦⟶⟫ τ p''› unfolding weak_step_def by auto
  with ‹α = τ›
  show ‹p ⟶⟫↦⟶⟫ α p''› by auto
next
  case False
  assume ‹p ⟶⟫ p'›
    and ‹p' ⟶⟫↦⟶⟫ α p''›
    and ‹α ≠ τ›
  then have ‹∃p1 p2. p' ⟶⟫ p1 ∧ p1 ↦ α p2 ∧ p2 ⟶⟫ p''›
    using weak_step_def by auto
  then obtain p1 and p2 where ‹p' ⟶⟫ p1› and ‹p1 ↦ α p2› and ‹p2 ⟶⟫ p''› by auto

  from ‹p ⟶⟫ p'› and ‹p' ⟶⟫ p1›
  have ‹p ⟶⟫ p1› by (rule silent_reachable_trans)

  with ‹p1 ↦ α p2› and ‹p2 ⟶⟫ p''› and ‹α ≠ τ›
```

```
  show ‹p ⟶⟫↦⟶⟫ α p''›
    using weak_step_def by auto
qed
```

A sequence of `weak_step`'s from one state `p` to another `p'` is called a `weak_step_sequence` That means that `p'` can be reached from `p` with that sequence of steps.

```
inductive weak_step_sequence :: ‹'s ⇒ 'a list ⇒ 's ⇒ bool› (‹_ ⟶⟫↦⟶⟫$ _ _› [70,70,70]
80) where
  ‹p ⟶⟫↦⟶⟫$ [] p'› if ‹p ⟶⟫ p'› |
  ‹p ⟶⟫↦⟶⟫$ (α#rt) p''› if ‹p ⟶⟫↦⟶⟫ α p'› ‹p' ⟶⟫↦⟶⟫$ rt p''›

lemma weak_step_sequence_trans:
  assumes ‹p ⟶⟫↦⟶⟫$ tr_1 p'› and ‹p' ⟶⟫↦⟶⟫$ tr_2 p''›
  shows ‹p ⟶⟫↦⟶⟫$ (tr_1 @ tr_2) p''›
  using assms weak_step_sequence.intros(2)
proof induct
  case (1 p p')
  then show ?case
    by (metis LTS_Tau.weak_step_sequence.simps append_Nil silent_prepend_weak_step silent_reachable_trans)
next
  case (2 p α p' rt p'')
  then show ?case by fastforce
qed
```

The weak traces of a state or all possible sequences of weak transitions that can be performed. In the context of labelled transition systems, weak traces capture the observable behaviour of a state.

```
abbreviation weak_traces :: ‹'s ⇒ 'a list set›
  where ‹weak_traces p ≡ {tr. ∃p'. p ⟶⟫↦⟶⟫$ tr p'}›
```

The empty trace is in `weak_traces` for all states.

```
lemma empty_trace_allways_weak_trace:
  shows ‹[] ∈ weak_traces p›
  using silent_reachable.intros(1) weak_step_sequence.intros(1) by fastforce
```

Since `weak_step`'s can be proceeded and followed by any number τ-transitions and the empty `weak_step_sequence` also allows τ-transitions, τ can be prepended to a weak trace of a state.

```
lemma prepend_τ_weak_trace:
  assumes ‹tr ∈ weak_traces p›
  shows ‹(τ # tr) ∈ weak_traces p›
  using silent_reachable.intros(1)
    and weak_step_def
    and assms
    and mem_Collect_eq
    and weak_step_sequence.intros(2)
  by fastforce
```

If state `p'` is reachable from state `p` via a sequence of τ-transitions and there exists a weak trace `tr` starting from `p'`, then `tr` is also a weak trace starting from `p`.

```
lemma silent_prepend_weak_traces:
  assumes ‹p ⟶⟫ p'›
      and ‹tr ∈ weak_traces p'›
    shows ‹tr ∈ weak_traces p›
  using assms
proof-
  assume ‹p ⟶⟫ p'›
     and ‹tr ∈ weak_traces p'›
  hence ‹∃p''. p' ⟶⟫↦⟶⟫$ tr p''› by auto
  then obtain p'' where ‹p' ⟶⟫↦⟶⟫$ tr p''› by auto
```

```
  from ‹p' ↠↦↠$ tr p''›
    and ‹p ↠ p'›
  have ‹p ↠↦↠$ tr p''›
    by (metis append_self_conv2 weak_step_sequence.intros(1) weak_step_sequence_trans)

  hence ‹∃p''. p ↠↦↠$ tr p''› by auto
  then show ‹tr ∈ weak_traces p›
    by blast
qed
```

If there is an α-transition from `p` to `p'`, and `p'` has a weak trace `tr`, then the sequence (α `# tr`) is a valid (weak) trace of `p`.

```
lemma step_prepend_weak_traces:
  assumes ‹p ↦ α p'›
      and ‹tr ∈ weak_traces p'›
    shows ‹(α # tr) ∈ weak_traces p›
  using assms
proof -
  from ‹tr ∈ weak_traces p'›
  have ‹∃p''. p' ↠↦↠$ tr p''› by auto
  then obtain p'' where ‹p' ↠↦↠$ tr p''› by auto
  with ‹p ↦ α p'›
  have ‹p ↠↦↠$ (α # tr) p''›
    by (metis LTS_Tau.silent_reachable.intros(1) LTS_Tau.silent_reachable_append_τ LTS_Tau.weak_step_def
LTS_Tau.weak_step_sequence.intros(2))
  then have ‹∃p''. p ↠↦↠$ (α # tr) p''› by auto
  then show ‹(α # tr) ∈ weak_traces p› by auto
qed
```

One of the behavioural pre-orders/equivalences that we talk about is trace pre-order/equivalence. This is the modal characterization for one state is weakly trace pre-ordered to the other, `weakly_trace_preordered` denoted by `≲WT`, and two states are weakly trace equivalent, `weakly_trace_equivalent` denoted `≃WT`.

```
definition weakly_trace_preordered (infix ‹≲WT› 60) where
  ‹p ≲WT q ≡ weak_traces p ⊆ weak_traces q›

definition weakly_trace_equivalent (infix ‹≃WT› 60) where
  ‹p ≃WT q ≡ p ≲WT q ∧ q ≲WT p›
```

Just like `step_setp`, one can lift (`↠`) to sets of states.

```
abbreviation silent_reachable_setp (infix ‹↠S› 80) where
  ‹P ↠S P' ≡ ((∀p' ∈ P'. ∃p ∈ P. p ↠ p') ∧ (∀p ∈ P. ∀p'. p ↠ p' ⟶ p' ∈ P'))›

definition silent_reachable_set :: ‹'s set ⇒ 's set› where
  ‹silent_reachable_set P ≡ { q . ∃p ∈ P. p ↠ q }›

lemma sreachable_set_is_sreachable: ‹P ↠S (silent_reachable_set P)›
  using silent_reachable_set_def by auto

lemma exactly_one_sreachable_set: ‹∃!Q. P ↠S Q›
proof -
  from sreachable_set_is_sreachable
  have ‹P ↠S (silent_reachable_set P)› .

  have ‹⋀Q. P ↠S Q ⟹ Q = (silent_reachable_set P)›
  proof -
    fix Q
    assume ‹P ↠S Q›

    with sreachable_set_is_sreachable
```

```
    have ‹∀q ∈ Q. q ∈ (silent_reachable_set P)›
      by meson

    from ‹P ↠S Q›
     and sreachable_set_is_sreachable
    have ‹∀q ∈ (silent_reachable_set P). q ∈ Q›
      by meson

    from ‹∀q ∈ Q. q ∈ (silent_reachable_set P)›
     and ‹∀q ∈ (silent_reachable_set P). q ∈ Q›
    show ‹Q = (silent_reachable_set P)› by auto
  qed

  with ‹P ↠S (silent_reachable_set P)›
  show ‹∃!Q. P ↠S Q›
    by blast
qed


lemma sreachable_set_eq:
  assumes ‹P ↠S Q›
  shows ‹Q = silent_reachable_set P›
  using exactly_one_sreachable_set sreachable_set_is_sreachable assms by fastforce
```

We likewise lift `soft_step` to sets of states.

```
abbreviation soft_step_setp (‹_ ↦aS _ _› [70,70,70] 80) where
  ‹P ↦aS α Q ≡ (∀q ∈ Q. ∃p ∈ P. p ↦a α q) ∧ (∀p ∈ P. ∀q. p ↦a α q ⟶ q ∈ Q)›

definition soft_step_set :: ‹'s set ⇒ 'a ⇒ 's set› where
  ‹soft_step_set P α ≡ { q . ∃p ∈ P. p ↦a α q }›

lemma soft_step_set_is_soft_step_set:
  ‹P ↦aS α (soft_step_set P α)›
  using soft_step_set_def by auto

lemma exactly_one_soft_step_set:
  ‹∃!Q. P ↦aS α Q›
proof -
  from soft_step_set_is_soft_step_set
  have ‹P ↦aS α (soft_step_set P α)›
    and ‹⋀Q. P ↦aS α Q ⟹ Q = (soft_step_set P α)›
    by fastforce+
  show ‹∃!Q. P ↦aS α Q›
  proof
    from ‹P ↦aS α (soft_step_set P α)›
    show ‹P ↦aS α (soft_step_set P α)› .
  next
    from ‹⋀Q. P ↦aS α Q ⟹ Q = (soft_step_set P α)›
    show ‹⋀Q. P ↦aS α Q ⟹ Q = (soft_step_set P α)› .
  qed
qed

lemma soft_step_set_eq:
  assumes ‹P ↦aS α Q›
  shows ‹Q = soft_step_set P α›
  using exactly_one_soft_step_set soft_step_set_is_soft_step_set assms
  by fastforce

abbreviation ‹stable_state p ≡ ∀p'. ¬(p ↦ τ p')›

lemma stable_state_stable:
```

```
  assumes ‹stable_state p› ‹p ⟶» p'›
  shows ‹p = p'›
  using assms(2,1) by (cases, blast+)

definition stability_respecting :: ‹('s ⇒ 's ⇒ bool) ⇒ bool› where
  ‹stability_respecting R ≡ ∀ p q. R p q ∧ stable_state p ⟶
    (∃q'. q ⟶» q' ∧ R p q' ∧ stable_state q')›

end

end
```

2.3 Modal Logics on LTS

```
theory LTS_Semantics
  imports
    LTS
begin

locale lts_semantics = LTS step
  for step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ↦ _ _› [70,70,70] 80) +
  fixes models :: ‹'s ⇒ 'formula ⇒ bool›
begin

definition entails :: ‹'formula ⇒ 'formula ⇒ bool› where
  entails_def[simp]: ‹entails φl φr ≡ (∀p. (models p φl) ⟶ (models p φr))›

definition logical_eq :: ‹'formula ⇒ 'formula ⇒ bool› where
  logical_eq_def[simp]: ‹logical_eq φl φr ≡ entails φl φr ∧ entails φr φl›
```

Formula implication is a pre-order.

```
lemma entails_preord: ‹reflp (entails)› ‹transp (entails)›
  by (simp add: reflpI transp_def)+

lemma eq_equiv: ‹equivp logical_eq›
  using equivpI reflpI sympI transpI
  unfolding logical_eq_def entails_def
  by (smt (verit, del_insts))
```

The definition given above is equivalent which means formula equivalence is a biimplication on the models predicate.

```
lemma eq_equality[simp]: ‹(logical_eq φl φr) = (∀p. models p φl = models p φr)›
  by force

lemma logical_eqI[intro]:
  assumes
    ‹⋀s. models s φl ⟹ models s φr›
    ‹⋀s. models s φr ⟹ models s φl›
  shows
    ‹logical_eq φl φr›
  using assms by auto

definition distinguishes :: ‹'formula ⇒ 's ⇒ 's ⇒ bool› where
  distinguishes_def[simp]:
  ‹distinguishes φ p q ≡ models p φ ∧ ¬(models q φ)›

definition distinguishes_from :: ‹'formula ⇒ 's ⇒ 's set ⇒ bool› where
  distinguishes_from_def[simp]:
  ‹distinguishes_from φ p Q ≡ models p φ ∧ (∀q ∈ Q. ¬(models q φ))›

lemma distinction_unlifting:
```

```
  assumes
    ‹distinguishes_from φ p Q›
  shows
    ‹∀q∈Q. distinguishes φ p q›
  using assms by simp

lemma no_distinction_fom_self:
  assumes
    ‹distinguishes φ p p›
  shows
    ‹False›
  using assms by simp
```

If φ is equivalent to φ' and φ distinguishes process p from process q, the φ' also distinguishes process p from process q.

```
lemma dist_equal_dist:
  assumes ‹logical_eq φl φr›
      and ‹distinguishes φl p q›
    shows ‹distinguishes φr p q›
  using assms
  by auto

abbreviation model_set :: ‹'formula ⇒ 's set› where
  ‹model_set φ ≡ {p. models p φ}›
```

2.4 Preorders and Equivalences on Processes Derived from Formula Sets

A set of formulas pre-orders two processes p and q if for all formulas in this set the fact that p satisfies a formula means that also q must satisfy this formula.

```
definition preordered :: ‹'formula set ⇒ 's ⇒ 's ⇒ bool› where
  preordered_def[simp]:
  ‹preordered φs p q ≡ ∀φ ∈ φs. models p φ ⟶ models q φ›
```

If a set of formulas pre-orders two processes p and q, then no formula in that set may distinguish p from q.

```
lemma preordered_no_distinction:
  ‹preordered φs p q = (∀φ ∈ φs. ¬(distinguishes φ p q))›
  by simp
```

A formula set derived pre-order is a pre-order.

```
lemma preordered_preord:
  ‹reflp (preordered φs)›
  ‹transp (preordered φs)›
  unfolding reflp_def transp_def by auto
```

A set of formulas equates two processes p and q if this set of formulas pre-orders these two processes in both directions.

```
definition equivalent :: ‹'formula set ⇒ 's ⇒ 's ⇒ bool› where
  equivalent_def[simp]:
  ‹equivalent φs p q ≡ preordered φs p q ∧ preordered φs q p›
```

If a set of formulas equates two processes p and q, then no formula in that set may distinguish p from q nor the other way around.

```
lemma equivalent_no_distinction: ‹equivalent φs p q
     = (∀φ ∈ φs. ¬(distinguishes φ p q) ∧ ¬(distinguishes φ q p))›
  by auto
```

A formula-set-derived equivalence is an equivalence.

```
lemma equivalent_equiv: ‹equivp (equivalent φs)›
proof (rule equivpI)
  show ‹reflp (equivalent φs)›
    by (simp add: reflpI)
  show ‹symp (equivalent φs)›
    unfolding equivalent_no_distinction symp_def
    by auto
  show ‹transp (equivalent φs)›
    unfolding transp_def equivalent_def preordered_def
    by blast
qed

end

end
```

3 Stability-Respecting Branching Bisimilarity (HML_{SRBB})

```
theory HML_SRBB
  imports Main LTS_Semantics
begin
```

This section describes the largest subset of the full HML language in section ?? that we are using for purposes of silent step spectroscopy. It is supposed to characterize the most fine grained behavioural equivalence that we may decide: Stability-Respecting Branching Bisimilarity (SRBB). While there are good reasons to believe that this subset truly characterizes SRBB (c.f.[1]), we do not provide a formal proof. From this sublanguage smaller subsets are derived via the notion of expressiveness prices (5).

The mutually recursive data types `hml_srbb`, `hml_srbb_inner` and `hml_srbb_conjunct` represent the subset of all `hml` formulas, which characterize stability-respecting branching bisimilarity (abbreviated to 'SRBB').

When a parameter is of type `hml_srbb` we typically use φ as a name, for type `hml_srbb_inner` we use χ and for type `hml_srbb_conjunct` we use ψ.
The data constructors are to be interpreted as follows:

- in `hml_srbb`:
  - `TT` encodes $\top$
  - `(Internal` χ`)` encodes $\langle\varepsilon\rangle\chi$
  - `(ImmConj I` ψ`s)` encodes $\bigwedge_{i\in \mathtt{I}} \psi s(i)$
- in `hml_srbb_inner`
  - `(Obs` α φ`)` encodes $(\alpha)\varphi$ (Note the difference to [1])
  - `(Conj I` ψ`s)` encode $\bigwedge_{i\in \mathtt{I}} \psi s(i)$
  - `(StableConj I` ψ`s)` encodes $\neg\langle\tau\rangle\top \wedge \bigwedge_{i\in \mathtt{I}} \psi s(i)$
  - `(BranchConj` α φ `I` ψ`s)` encodes $(\alpha)\varphi \wedge \bigwedge_{i\in \mathtt{I}} \psi s(i)$
- in `hml_srbb_conjunct`
  - `(Pos` χ`)` encodes $\langle\varepsilon\rangle\chi$
  - `(Neg` χ`)` encodes $\neg\langle\varepsilon\rangle\chi$

For justifications regarding the explicit inclusion of `TT` and the encoding of conjunctions via index sets `I` and mapping from indices to conjuncts ψ`s`, reference the `TT` and `Conj` data constructors of the type `hml` in section ??.

```
datatype
  ('act, 'i) hml_srbb =
    TT |
    Internal ‹('act, 'i) hml_srbb_inner› |
    ImmConj ‹'i set› ‹'i ⇒ ('act, 'i) hml_srbb_conjunct›
and
  ('act, 'i) hml_srbb_inner =
    Obs 'act ‹('act, 'i) hml_srbb› |
    Conj ‹'i set› ‹'i ⇒ ('act, 'i) hml_srbb_conjunct› |
    StableConj ‹'i set› ‹'i ⇒ ('act, 'i) hml_srbb_conjunct› |
    BranchConj 'act ‹('act, 'i) hml_srbb›
               ‹'i set› ‹'i ⇒ ('act, 'i) hml_srbb_conjunct›
and
  ('act, 'i) hml_srbb_conjunct =
    Pos ‹('act, 'i) hml_srbb_inner› |
    Neg ‹('act, 'i) hml_srbb_inner›
```

3.1 Semantics of HML_{SRBB} Formulas

This section describes how semantic meaning is assigned to HML_{SRBB} formulas in the context of a LTS. We define what it means for a process p to satisfy a HML_{SRBB} formula φ, written as p ⊨SRBB φ.

```
context LTS_Tau
begin

primrec
      hml_srbb_models :: ‹'s ⇒ ('a, 's) hml_srbb ⇒ bool› (infixl ‹⊨SRBB› 60)
  and hml_srbb_inner_models :: ‹'s ⇒ ('a, 's) hml_srbb_inner ⇒ bool›
  and hml_srbb_conjunct_models :: ‹'s ⇒ ('a, 's) hml_srbb_conjunct ⇒ bool› where
  ‹hml_srbb_models state TT =
    True› |
  ‹hml_srbb_models state (Internal χ) =
    (∃p'. state ↠ p' ∧ (hml_srbb_inner_models p' χ))› |
  ‹hml_srbb_models state (ImmConj I ψs) =
    (∀i∈I. hml_srbb_conjunct_models state (ψs i))› |

  ‹hml_srbb_inner_models state (Obs a φ) =
    ((∃p'. state ↦ a p' ∧ hml_srbb_models p' φ) ∨ a = τ ∧ hml_srbb_models state φ)› |
  ‹hml_srbb_inner_models state (Conj I ψs) =
    (∀i∈I. hml_srbb_conjunct_models state (ψs i))› |
  ‹hml_srbb_inner_models state (StableConj I ψs) =
    ((∄p'. state ↦ τ p') ∧ (∀i∈I. hml_srbb_conjunct_models state (ψs i)))› |
  ‹hml_srbb_inner_models state (BranchConj a φ I ψs) =
    (((∃p'. state ↦ a p' ∧ hml_srbb_models p' φ) ∨ a = τ ∧ hml_srbb_models state φ)
    ∧ (∀i∈I. hml_srbb_conjunct_models state (ψs i)))› |

  ‹hml_srbb_conjunct_models state (Pos χ) =
    (∃p'. state ↠ p' ∧ hml_srbb_inner_models p' χ)› |
  ‹hml_srbb_conjunct_models state (Neg χ) =
    (∄p'. state ↠ p' ∧ hml_srbb_inner_models p' χ)›

sublocale lts_semantics ‹step› ‹hml_srbb_models› .
sublocale hml_srbb_inner: lts_semantics where models = hml_srbb_inner_models .
sublocale hml_srbb_conj: lts_semantics where models = hml_srbb_conjunct_models .
```

3.2 Different Variants of Verum

```
lemma empty_conj_trivial[simp]:
  ‹state ⊨SRBB ImmConj {} ψs›
  ‹hml_srbb_inner_models state (Conj {} ψs)›
  ‹hml_srbb_inner_models state (Obs τ TT)›
  by simp+
```

$\bigwedge\{(\tau)\top\}$ is trivially true.

```
lemma empty_branch_conj_tau:
  ‹hml_srbb_inner_models state (BranchConj τ TT {} ψs)›
  by auto

lemma stable_conj_parts:
  assumes
    ‹hml_srbb_inner_models p (StableConj I Ψ)›
    ‹i ∈ I ›
  shows ‹hml_srbb_conjunct_models p (Ψ i)›
  using assms by auto

lemma branching_conj_parts:
  assumes
    ‹hml_srbb_inner_models p (BranchConj α φ I Ψ)›
```

```
    ‹i ∈ I ›
  shows ‹hml_srbb_conjunct_models p (Ψ i)›
  using assms by auto

lemma branching_conj_obs:
  assumes
    ‹hml_srbb_inner_models p (BranchConj α φ I Ψ)›
  shows ‹hml_srbb_inner_models p (Obs α φ)›
  using assms by auto
```

3.3 Distinguishing Formulas

Now, we take a look at some basic properties of the `distinguishes` predicate:

$\top$ can never distinguish two processes. This is due to the fact that every process satisfies `T`. Therefore, the second part of the definition of `distinguishes` never holds.

```
lemma verum_never_distinguishes:
  ‹¬ distinguishes TT p q›
  by simp
```

If $\bigwedge_{i \in I} \psi s(i)$ distinguishes `p` from `q`, then there must be at least one conjunct in this conjunction that distinguishes `p` from `q`.

```
lemma srbb_dist_imm_conjunction_implies_dist_conjunct:
  assumes ‹distinguishes (ImmConj I ψs) p q›
  shows ‹∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q›
  using assms by auto
```

If there is one conjunct in that distinguishes `p` from `q` and `p` satisfies all other conjuncts in a conjunction then $\bigwedge_{i \in I} \psi s(i)$ (where ψs ranges over the previously mentioned conjunctions) distinguishes `p` from `q`.

```
lemma srbb_dist_conjunct_implies_dist_imm_conjunction:
  assumes ‹i∈I›
      and ‹hml_srbb_conj.distinguishes (ψs i) p q›
      and ‹∀i∈I. hml_srbb_conjunct_models p (ψs i)›
    shows ‹distinguishes (ImmConj I ψs) p q›
  using assms by auto
```

If $\bigwedge_{i \in I} \psi s(i)$ distinguishes `p` from `q`, then there must be at least one conjunct in this conjunction that distinguishes `p` from `q`.

```
lemma srbb_dist_conjunction_implies_dist_conjunct:
  assumes ‹hml_srbb_inner.distinguishes (Conj I ψs) p q›
  shows ‹∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q›
  using assms by auto
```

In the following, we replicate `srbb_dist_conjunct_implies_dist_imm_conjunction` for simple conjunctions in `hml_srbb_inner`.

```
lemma srbb_dist_conjunct_implies_dist_conjunction:
  assumes ‹i∈I›
      and ‹hml_srbb_conj.distinguishes (ψs i) p q›
      and ‹∀i∈I. hml_srbb_conjunct_models p (ψs i)›
  shows ‹hml_srbb_inner.distinguishes (Conj I ψs) p q›
  using assms by auto
```

We also replicate `srbb_dist_imm_conjunction_implies_dist_conjunct` for branching conjunctions $(\alpha)\varphi \wedge \bigwedge_{i \in I} \psi s(i)$. Here, either the branching condition distinguishes `p` from `q` or there must be a distinguishing conjunct.

```
lemma srbb_dist_branch_conjunction_implies_dist_conjunct_or_branch:
  assumes ‹hml_srbb_inner.distinguishes (BranchConj α φ I ψs) p q›
  shows ‹(∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q)
       ∨ (hml_srbb_inner.distinguishes (Obs α φ) p q)›
```

```
  using assms by force
```

In the following, we replicate `srbb_dist_conjunct_implies_dist_imm_conjunction` for branching conjunctions in `hml_srbb_inner`.

```
lemma srbb_dist_conjunct_or_branch_implies_dist_branch_conjunction:
  assumes ‹∀i ∈ I. hml_srbb_conjunct_models p (ψs i)›
      and ‹hml_srbb_inner_models p (Obs α φ)›
      and ‹(i∈I ∧ hml_srbb_conj.distinguishes (ψs i) p q)
         ∨ (hml_srbb_inner.distinguishes (Obs α φ) p q)›
  shows ‹hml_srbb_inner.distinguishes (BranchConj α φ I ψs) p q›
  using assms by force
```

3.4 HML_{SRBB} Implication

```
abbreviation hml_srbb_impl
  :: ‹('a, 's) hml_srbb ⇒ ('a, 's) hml_srbb ⇒ bool›  (infixr ‹⇛› 70)
where
  ‹hml_srbb_impl ≡ entails›

abbreviation
  hml_srbb_impl_inner
  :: ‹('a, 's) hml_srbb_inner ⇒ ('a, 's) hml_srbb_inner ⇒ bool›
  (infix ‹χ⇛› 70)
where
  ‹(χ⇛) ≡ hml_srbb_inner.entails›

abbreviation
  hml_srbb_impl_conjunct
  :: ‹('a, 's) hml_srbb_conjunct ⇒ ('a, 's) hml_srbb_conjunct ⇒ bool›
  (infix ‹ψ⇛› 70)
where
  ‹(ψ⇛) ≡ hml_srbb_conj.entails›
```

3.5 HML_{SRBB} Equivalence

We define HML_{SRBB} formula equivalence to by appealing to HML_{SRBB} implication. A HML_{SRBB} formula is equivalent to another formula if both imply each other.

```
abbreviation
  hml_srbb_eq
  :: ‹('a, 's) hml_srbb ⇒ ('a, 's) hml_srbb ⇒ bool›
  (infix ‹⇚srbb⇛› 70)
where
  ‹(⇚srbb⇛) ≡ logical_eq›

abbreviation
  hml_srbb_eq_inner
  :: ‹('a, 's) hml_srbb_inner ⇒ ('a, 's) hml_srbb_inner ⇒ bool›
  (infix ‹⇚χ⇛› 70)
where
  ‹(⇚χ⇛) ≡ hml_srbb_inner.logical_eq›

abbreviation
  hml_srbb_eq_conjunct
  :: ‹('a, 's) hml_srbb_conjunct ⇒ ('a, 's) hml_srbb_conjunct ⇒ bool›
  (infix ‹⇚ψ⇛› 70)
  where
  ‹(⇚ψ⇛) ≡ hml_srbb_conj.logical_eq›
```

3.6 Substitution

```
lemma srbb_internal_subst:
```

```
  assumes ‹χl ⇐χ⇒ χr›
      and ‹φ ⇐srbb⇒ (Internal χl)›
    shows ‹φ ⇐srbb⇒ (Internal χr)›
  using assms by force
```

3.7 Congruence

This section provides means to derive new equivalences by extending both sides with a given prefix.

Prepending $\langle\varepsilon\rangle\ldots$ preserves equivalence.

```
lemma internal_srbb_cong:
  assumes ‹χl ⇐χ⇒ χr›
  shows ‹(Internal χl) ⇐srbb⇒ (Internal χr)›
  using assms by auto
```

If equivalent conjuncts are included in an otherwise identical conjunction, the equivalence is preserved.

```
lemma immconj_cong:
  assumes ‹ψsl ‘ I = ψsr ‘ I›
      and ‹ψsl s ⇐ψ⇒ ψsr s›
  shows ‹ImmConj (I ∪ {s}) ψsl ⇐srbb⇒ ImmConj (I ∪ {s}) ψsr›
  using assms
  by (auto) (metis (mono_tags, lifting) image_iff)+
```

Prepending $(\alpha)\ldots$ preserves equivalence.

```
lemma obs_srbb_cong:
  assumes ‹φl ⇐srbb⇒ φr›
  shows ‹(Obs α φl) ⇐χ⇒ (Obs α φr)›
  using assms by auto
```

3.8 Known Equivalence Elements

The formula $(\tau)\top$ is equivalent to $\bigwedge\{\}$.

```
lemma srbb_obs_τ_is_χTT: ‹Obs τ TT ⇐χ⇒ Conj {} ψs›
  by simp
```

The formula $(\alpha)\varphi$ is equivalent to $(\alpha)\varphi \wedge \bigwedge\{\}$.

```
lemma srbb_obs_is_empty_branch_conj: ‹Obs α φ ⇐χ⇒ BranchConj α φ {} ψs›
  by auto
```

The formula $\top$ is equivalent to $\langle\varepsilon\rangle\bigwedge\{\}$.

```
lemma srbb_TT_is_χTT: ‹TT ⇐srbb⇒ Internal (Conj {} ψs)›
  using LTS_Tau.refl by force
```

The formula $\top$ is equivalent to $\bigwedge\{\}$.

```
lemma srbb_TT_is_empty_conj: ‹TT ⇐srbb⇒ ImmConj {} ψs›
  by simp
```

Positive conjuncts in stable conjunctions can be replaced by negative ones.

```
lemma srbb_stable_Neg_normalizable:
  assumes
    ‹i ∈ I› ‹Ψ i = Pos χ›
    ‹Ψ’ = Ψ(i:= Neg (StableConj {left} (λ_. Neg χ)))›
  shows
    ‹Internal (StableConj I Ψ) ⇐srbb⇒ Internal (StableConj I Ψ’)›
proof (rule logical_eqI)
  fix p
  assume ‹p ⊨SRBB Internal (StableConj I Ψ)›
```

```
  then obtain p' where p'_spec: ‹p ↠ p'› ‹hml_srbb_inner_models p' (StableConj I Ψ)› by
auto
  hence ‹stable_state p'› by auto
  from p'_spec have ‹∃p''. p' ↠ p'' ∧ hml_srbb_inner_models p'' χ›
    using assms(1,2) by auto
  with ‹stable_state p'› have ‹hml_srbb_inner_models p' χ›
    using stable_state_stable by blast
  hence ‹hml_srbb_conjunct_models p' (Neg (StableConj {left} (λ_. Neg χ)))›
    using ‹stable_state p'› stable_state_stable by (auto, blast)
  hence ‹hml_srbb_inner_models p' (StableConj I Ψ')›
    unfolding assms(3) using p'_spec by auto
  thus ‹p ⊨SRBB hml_srbb.Internal (StableConj I Ψ')›
    using ‹p ↠ p'› by auto
next
  fix p
  assume ‹p ⊨SRBB Internal (StableConj I Ψ')›
  then obtain p' where p'_spec: ‹p ↠ p'› ‹hml_srbb_inner_models p' (StableConj I Ψ')›
by auto
  hence ‹stable_state p'› by auto
  from p'_spec(2) have other_conjuncts: ‹∀j∈I. i ≠ j ⟶ hml_srbb_conjunct_models p' (Ψ
j)›
    using assms stable_conj_parts fun_upd_apply by metis
  from p'_spec(2) have ‹hml_srbb_conjunct_models p' (Ψ' i)›
    using assms(1) stable_conj_parts by blast
  hence ‹hml_srbb_conjunct_models p' (Neg (StableConj {left} (λ_. Neg χ)))›
    unfolding assms(3) by auto
  with ‹stable_state p'› have ‹hml_srbb_inner_models p' χ›
    using stable_state_stable by (auto, metis silent_reachable.simps)
  then have ‹hml_srbb_conjunct_models p' (Pos χ)›
    using LTS_Tau.refl by fastforce
  hence ‹hml_srbb_inner_models p' (StableConj I Ψ)›
    using p'_spec assms other_conjuncts by auto
  thus ‹p ⊨SRBB hml_srbb.Internal (StableConj I Ψ)›
    using p'_spec(1) by auto
qed
```

All positive conjuncts in stable conjunctions can be replaced by negative ones at once.

```
lemma srbb_stable_Neg_normalizable_set:
  assumes
    ‹Ψ' = (λi. case (Ψ i) of
      Pos χ ⇒ Neg (StableConj {left} (λ_. Neg χ)) |
      Neg χ ⇒ Neg χ)›
  shows
    ‹Internal (StableConj I Ψ) ⇐srbb⇒ Internal (StableConj I Ψ')›
proof (rule logical_eqI)
  fix p
  assume ‹p ⊨SRBB Internal (StableConj I Ψ)›
  then obtain p' where p'_spec: ‹p ↠ p'› ‹hml_srbb_inner_models p' (StableConj I Ψ)› by
auto
  hence ‹stable_state p'› by auto
  from p'_spec have
    ‹∀χ i. i∈I ∧ Ψ i = Pos χ ⟶ (∃p''. p' ↠ p'' ∧ hml_srbb_inner_models p'' χ)›
    by fastforce
  with ‹stable_state p'› have ‹∀χ i. i∈I ∧ Ψ i = Pos χ ⟶ hml_srbb_inner_models p' χ›
    using stable_state_stable by blast
  hence pos_rewrite: ‹∀χ i. i∈I ∧ Ψ i = Pos χ ⟶
      hml_srbb_conjunct_models p' (Neg (StableConj {left} (λ_. Neg χ)))›
    using ‹stable_state p'› stable_state_stable by (auto, blast)
  hence ‹hml_srbb_inner_models p' (StableConj I Ψ')›
    unfolding assms using p'_spec
    by (auto, metis (no_types, lifting) hml_srbb_conjunct.exhaust hml_srbb_conjunct.simps(5,6)
```

```
        pos_rewrite)
  thus ‹p ⊨SRBB Internal (StableConj I Ψ')›
    using ‹p ↠ p'› by auto
next
  fix p
  assume ‹p ⊨SRBB Internal (StableConj I Ψ')›
  then obtain p' where p'_spec: ‹p ↠ p'› ‹hml_srbb_inner_models p' (StableConj I Ψ')›
by auto
  hence ‹stable_state p'› by auto
  from p'_spec(2) have other_conjuncts:
      ‹∀χ i. i∈I ∧ Ψ i = Neg χ ⟶ hml_srbb_conjunct_models p' (Ψ i)›
    using assms stable_conj_parts by (metis hml_srbb_conjunct.simps(6))
  from p'_spec(2) have ‹∀χ i. i∈I ∧ Ψ i = Pos χ ⟶ hml_srbb_conjunct_models p' (Ψ' i)›
    using assms(1) stable_conj_parts by blast
  hence ‹∀χ i. i∈I ∧ Ψ i = Pos χ ⟶
      hml_srbb_conjunct_models p' (Neg (StableConj {left} (λ_. Neg χ)))›
    unfolding assms by auto
  with ‹stable_state p'› have ‹∀χ i. i∈I ∧ Ψ i = Pos χ ⟶ hml_srbb_inner_models p' χ›
    using stable_state_stable by (auto, metis silent_reachable.simps)
  then have pos_conjuncts:
      ‹∀χ i. i∈I ∧ Ψ i = Pos χ ⟶hml_srbb_conjunct_models p' (Pos χ)›
    using hml_srbb_conjunct_models.simps(1) silent_reachable.simps by blast
  hence ‹hml_srbb_inner_models p' (StableConj I Ψ)›
    using p'_spec assms other_conjuncts
    by (auto, metis other_conjuncts pos_conjuncts hml_srbb_conjunct.exhaust)
  thus ‹p ⊨SRBB Internal (StableConj I Ψ)›
    using p'_spec(1) by auto
qed

definition conjunctify_distinctions ::
  ‹('s ⇒ ('a, 's) hml_srbb) ⇒ 's ⇒ ('s ⇒ ('a, 's) hml_srbb_conjunct)› where
  ‹conjunctify_distinctions Φ p ≡ λq.
    case (Φ q) of
      TT ⇒ undefined
    | Internal χ ⇒ Pos χ
    | ImmConj I Ψ ⇒ Ψ (SOME i. i∈I ∧ hml_srbb_conj.distinguishes (Ψ i) p q)›

lemma distinction_conjunctification:
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q›
  shows
    ‹∀q∈I. hml_srbb_conj.distinguishes ((conjunctify_distinctions Φ p) q) p q›
  unfolding conjunctify_distinctions_def
proof
  fix q
  assume q_I: ‹q∈I›
  show ‹hml_srbb_conj.distinguishes
          (case Φ q of hml_srbb.Internal x ⇒ hml_srbb_conjunct.Pos x
           | ImmConj I Ψ ⇒ Ψ (SOME i. i ∈ I ∧ hml_srbb_conj.distinguishes (Ψ i) p q))
          p q›
  proof (cases ‹Φ q›)
    case TT
    then show ?thesis using assms q_I by fastforce
  next
    case (Internal χ)
    then show ?thesis using assms q_I by auto
  next
    case (ImmConj J Ψ)
    then have ‹∃i ∈ J. hml_srbb_conj.distinguishes (Ψ i) p q›
      using assms q_I by auto
    then show ?thesis
```

```
      by (metis (mono_tags, lifting) ImmConj hml_srbb.simps(11) someI)
  qed
qed

lemma distinction_combination:
  fixes p q
  defines ‹Qα ≡ {q'. q ↠ q' ∧ (∄φ. distinguishes φ p q')}›
  assumes
    ‹p ↦a α p'›
    ‹∀q'∈ Qα.
      ∀q''. q' ↦a α q'' ⟶ (distinguishes (Φ q'') p' q'')›
  shows
    ‹∀q'∈Qα.
      hml_srbb_inner.distinguishes (Obs α (ImmConj {q''. ∃q'''∈Qα. q''' ↦a α q''}
                                                (conjunctify_distinctions Φ p'))) p q'›
proof -
  have ‹∀q'∈ Qα. ∀q''∈{q''. q' ↦a α q''}.
      hml_srbb_conj.distinguishes ((conjunctify_distinctions Φ p') q'') p' q''›
  proof clarify
    fix q' q''
    assume ‹q' ∈ Qα› ‹q' ↦a α q''›
    thus ‹hml_srbb_conj.distinguishes (conjunctify_distinctions Φ p' q'') p' q''›
      using distinction_conjunctification assms(3)
      by (metis mem_Collect_eq)
  qed
  hence ‹∀q'∈ Qα. ∀q''∈{q''. ∃q1'∈Qα. q1' ↦a α q''}.
      hml_srbb_conj.distinguishes ((conjunctify_distinctions Φ p') q'') p' q''› by blast
  hence ‹∀q'∈ Qα. ∀q''. q' ↦a α q''
    ⟶ distinguishes (ImmConj {q''. ∃q1'∈Qα. q1' ↦a α q''}
                              (conjunctify_distinctions Φ p')) p' q''› by auto
  thus ‹∀q'∈Qα.
      hml_srbb_inner.distinguishes (Obs α (ImmConj {q''. ∃q'''∈Qα. q''' ↦a α q''}
                                                (conjunctify_distinctions Φ p'))) p q'›
    by (auto) (metis assms(2))+
qed

definition conjunctify_distinctions_dual ::
  ‹('s ⇒ ('a, 's) hml_srbb) ⇒ 's ⇒ ('s ⇒ ('a, 's) hml_srbb_conjunct)› where
  ‹conjunctify_distinctions_dual Φ p ≡ λq.
    case (Φ q) of
      TT ⇒ undefined
    | Internal χ ⇒ Neg χ
    | ImmConj I Ψ ⇒
      (case Ψ (SOME i. i∈I ∧ hml_srbb_conj.distinguishes (Ψ i) q p) of
        Pos χ ⇒ Neg χ | Neg χ ⇒ Pos χ)›

lemma dual_conjunct:
  assumes
    ‹hml_srbb_conj.distinguishes ψ p q›
  shows
    ‹hml_srbb_conj.distinguishes (case ψ of
              hml_srbb_conjunct.Pos χ ⇒ hml_srbb_conjunct.Neg χ
              | hml_srbb_conjunct.Neg χ ⇒ hml_srbb_conjunct.Pos χ) q p›
  using assms
  by (cases ψ, auto)

lemma distinction_conjunctification_dual:
  assumes
    ‹∀q∈I. distinguishes (Φ q) q p›
  shows
    ‹∀q∈I. hml_srbb_conj.distinguishes (conjunctify_distinctions_dual Φ p q) p q›
```

```
  unfolding conjunctify_distinctions_dual_def
proof
  fix q
  assume q_I: ‹q∈I›
  show ‹hml_srbb_conj.distinguishes
          (case Φ q of hml_srbb.Internal x ⇒ hml_srbb_conjunct.Neg x
           | ImmConj I Ψ ⇒
               ( case Ψ (SOME i. i ∈ I ∧ hml_srbb_conj.distinguishes (Ψ i) q p) of
                  hml_srbb_conjunct.Pos x ⇒ hml_srbb_conjunct.Neg x
               | hml_srbb_conjunct.Neg x ⇒ hml_srbb_conjunct.Pos x))
          p q›
  proof (cases ‹Φ q›)
    case TT
    then show ?thesis using assms q_I by fastforce
  next
    case (Internal χ)
    then show ?thesis using assms q_I by auto
  next
    case (ImmConj J Ψ)
    then have ‹∃i ∈ J. hml_srbb_conj.distinguishes (Ψ i) q p›
      using assms q_I by auto
    hence ‹hml_srbb_conj.distinguishes (case Ψ
      (SOME i. i ∈ J ∧ hml_srbb_conj.distinguishes (Ψ i) q p) of
              hml_srbb_conjunct.Pos x ⇒ hml_srbb_conjunct.Neg x
              | hml_srbb_conjunct.Neg x ⇒ hml_srbb_conjunct.Pos x) p q›
      by (metis (no_types, lifting) dual_conjunct someI_ex)
    then show ?thesis unfolding ImmConj by auto
  qed
qed

lemma distinction_conjunctification_two_way:
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q ∨ distinguishes (Φ q) q p›
  shows
    ‹∀q∈I. hml_srbb_conj.distinguishes ((if distinguishes (Φ q) p q then conjunctify_distinctions
else conjunctify_distinctions_dual) Φ p q) p q›
proof safe
  fix q
  assume ‹q ∈ I›
  then consider ‹distinguishes (Φ q) p q› | ‹distinguishes (Φ q) q p› using assms by blast
  thus ‹hml_srbb_conj.distinguishes ((if distinguishes (Φ q) p q then conjunctify_distinctions
else conjunctify_distinctions_dual) Φ p q) p q›
  proof cases
    case 1
    then show ?thesis using distinction_conjunctification
      by (smt (verit) singleton_iff)
  next
    case 2
    then show ?thesis using distinction_conjunctification_dual singleton_iff
      unfolding distinguishes_def
      by (smt (verit, ccfv_threshold))
  qed
qed

end

end
```

4 Energy

```
theory Energy
  imports Main "HOL-Library.Extended_Nat"
begin
```

Following the paper [1, p. 5], we define energies as eight-dimensional vectors of natural numbers extended by ∞. But deviate from [1] in also defining an energy `eneg` that represents negative energy. This allows us to express energy updates (cf. [1, p. 8]) as total functions.

```
datatype energy = E (modal_depth: ‹enat›) (br_conj_depth: ‹enat›) (conj_depth: ‹enat›)
(st_conj_depth: ‹enat›)
                    (imm_conj_depth: ‹enat›) (pos_conjuncts: ‹enat›) (neg_conjuncts: ‹enat›)
(neg_depth: ‹enat›)
```

4.1 Ordering Energies

In order to define subtraction on energies, we first lift the orderings $\leq$ and **<** from `enat` to `energy`.

```
instantiation energy :: order begin

definition ‹e1 ≤ e2 ≡
  (case e1 of E a1 b1 c1 d1 e1 f1 g1 h1 ⇒ (
    case e2 of E a2 b2 c2 d2 e2 f2 g2 h2 ⇒
      (a1 ≤ a2 ∧ b1 ≤ b2 ∧ c1 ≤ c2 ∧ d1 ≤ d2 ∧ e1 ≤ e2 ∧ f1 ≤ f2 ∧ g1 ≤ g2 ∧ h1 ≤
h2)
    ))›

definition ‹(x::energy) < y = (x ≤ y ∧ ¬ y ≤ x)›
```

Next, we show that this yields a reflexive transitive antisymmetric order.

```
instance proof
  fix e1 e2 e3 :: energy
  show ‹e1 ≤ e1› unfolding less_eq_energy_def by (simp add: energy.case_eq_if)
  show ‹e1 ≤ e2 ⟹ e2 ≤ e3 ⟹ e1 ≤ e3› unfolding less_eq_energy_def
    by (smt (z3) energy.case_eq_if order_trans)
  show ‹e1 < e2 = (e1 ≤ e2 ∧ ¬ e2 ≤ e1)› using less_energy_def .
  show ‹e1 ≤ e2 ⟹ e2 ≤ e1 ⟹ e1 = e2› unfolding less_eq_energy_def
    by (smt (z3) energy.case_eq_if energy.expand nle_le)
qed


lemma leq_components[simp]:
  shows ‹e1 ≤ e2 ≡ (modal_depth e1 ≤ modal_depth e2 ∧ br_conj_depth e1 ≤ br_conj_depth
e2 ∧ conj_depth e1 ≤ conj_depth e2 ∧
                     st_conj_depth e1 ≤ st_conj_depth e2 ∧ imm_conj_depth e1 ≤ imm_conj_depth
e2 ∧ pos_conjuncts e1 ≤ pos_conjuncts e2 ∧
                     neg_conjuncts e1 ≤ neg_conjuncts e2 ∧ neg_depth e1 ≤ neg_depth e2)›
  unfolding less_eq_energy_def by (simp add: energy.case_eq_if)

lemma energy_leq_cases:
  assumes ‹modal_depth e1 ≤ modal_depth e2› ‹br_conj_depth e1 ≤ br_conj_depth e2› ‹conj_depth
e1 ≤ conj_depth e2›
          ‹st_conj_depth e1 ≤ st_conj_depth e2› ‹imm_conj_depth e1 ≤ imm_conj_depth e2›
‹pos_conjuncts e1 ≤ pos_conjuncts e2›
          ‹neg_conjuncts e1 ≤ neg_conjuncts e2› ‹neg_depth e1 ≤ neg_depth e2›
  shows ‹e1 ≤ e2› using assms unfolding leq_components by blast

end
```

We then use this order to define a predicate that decides if an `e1` may be subtracted from another `e2` without the result being negative. We encode this by `e1` being `somewhere_larger` than `e2`.

```
abbreviation somewhere_larger where ‹somewhere_larger e1 e2 ≡ ¬(e1 ≥ e2)›

lemma somewhere_larger_eq:
  assumes ‹somewhere_larger e1 e2›
  shows ‹modal_depth e1 < modal_depth e2 ∨ br_conj_depth e1 < br_conj_depth e2
         ∨ conj_depth e1 < conj_depth e2 ∨ st_conj_depth e1 < st_conj_depth e2 ∨ imm_conj_depth
e1 < imm_conj_depth e2
         ∨ pos_conjuncts e1 < pos_conjuncts e2 ∨ neg_conjuncts e1 < neg_conjuncts e2 ∨ neg_depth
e1 < neg_depth e2›
  by (smt (z3) assms energy.case_eq_if less_eq_energy_def linorder_le_less_linear)
```

4.2 Subtracting Energies

Using `somewhere_larger` we define subtraction as the `minus` operator on energies.

```
instantiation energy :: minus
begin

definition minus_energy_def[simp]: ‹e1 - e2 ≡ E
  ((modal_depth e1) - (modal_depth e2))
  ((br_conj_depth e1) - (br_conj_depth e2))
  ((conj_depth e1) - (conj_depth e2))
  ((st_conj_depth e1) - (st_conj_depth e2))
  ((imm_conj_depth e1) - (imm_conj_depth e2))
  ((pos_conjuncts e1) - (pos_conjuncts e2))
  ((neg_conjuncts e1) - (neg_conjuncts e2))
  ((neg_depth e1) - (neg_depth e2))›

instance ..

end
```

Afterwards, we prove some lemmas to ease the manipulation of expressions using subtraction on energies.

```
lemma energy_minus[simp]:
  shows ‹E a1 b1 c1 d1 e1 f1 g1 h1 - E a2 b2 c2 d2 e2 f2 g2 h2
         = E (a1 - a2) (b1 - b2) (c1 - c2) (d1 - d2)
             (e1 - e2) (f1 - f2) (g1 - g2) (h1 - h2)›
  unfolding minus_energy_def somewhere_larger_eq by simp

lemma minus_component_leq:
  assumes ‹s ≤ x› ‹x ≤ y›
  shows ‹modal_depth (x - s) ≤ modal_depth (y - s)› ‹br_conj_depth (x - s) ≤ br_conj_depth
(y - s)›
        ‹conj_depth (x - s) ≤ conj_depth (y - s)› ‹st_conj_depth (x - s) ≤ st_conj_depth
(y - s)›
        ‹imm_conj_depth (x - s) ≤ imm_conj_depth (y - s)› ‹pos_conjuncts (x - s) ≤ pos_conjuncts
(y -s)›
        ‹neg_conjuncts (x - s) ≤ neg_conjuncts (y - s)› ‹neg_depth (x - s) ≤ neg_depth
(y - s)›
proof-
  from assms have ‹s ≤ y› by (simp del: leq_components)
  with assms leq_components have
    ‹modal_depth (x - s) ≤ modal_depth (y - s) ∧ br_conj_depth (x - s) ≤ br_conj_depth
(y - s) ∧
    conj_depth (x - s) ≤ conj_depth (y - s) ∧ st_conj_depth (x - s) ≤ st_conj_depth (y
- s) ∧
    imm_conj_depth (x - s) ≤ imm_conj_depth (y - s) ∧ pos_conjuncts (x - s) ≤ pos_conjuncts
(y -s) ∧
    neg_conjuncts (x - s) ≤ neg_conjuncts (y - s) ∧ neg_depth (x - s) ≤ neg_depth (y -
s)›
```

```
    by (smt (verit, del_insts) add_diff_cancel_enat enat_add_left_cancel_le energy.sel
        leD le_iff_add le_less minus_energy_def)+
  thus
    ‹modal_depth (x - s) ≤ modal_depth (y - s)› ‹br_conj_depth (x - s) ≤ br_conj_depth
(y - s)›
    ‹conj_depth (x - s) ≤ conj_depth (y - s)› ‹st_conj_depth (x - s) ≤ st_conj_depth (y
- s)›
    ‹imm_conj_depth (x - s) ≤ imm_conj_depth (y - s)› ‹pos_conjuncts (x - s) ≤ pos_conjuncts
(y -s)›
    ‹neg_conjuncts (x - s) ≤ neg_conjuncts (y - s)› ‹neg_depth (x - s) ≤ neg_depth (y -
s)› by auto
qed

lemma enat_diff_mono:
  assumes ‹(i::enat) ≤ j›
  shows ‹i - k ≤ j - k›
proof (cases i)
  case (enat iN)
  show ?thesis
  proof (cases j)
    case (enat jN)
    then show ?thesis
      using assms enat_ile by (cases k, fastforce+)
  next
    case infinity
    then show ?thesis using assms by auto
  qed
next
  case infinity
  hence ‹j = ∞›
    using assms by auto
  then show ?thesis by auto
qed
```

We further show that the subtraction of energies is decreasing.

```
lemma energy_diff_mono:
  fixes s :: energy
  shows ‹mono_on UNIV (λx. x - s)›
  unfolding mono_on_def
  by (auto simp add: enat_diff_mono)

lemma gets_smaller:
  fixes s :: energy
  shows ‹(λx. x - s) x ≤ x›
  by (auto)
     (metis add.commute add_diff_cancel_enat enat_diff_mono idiff_infinity idiff_infinity_right
le_iff_add not_infinity_eq zero_le)+

lemma mono_subtract:
  assumes ‹x ≤ x'›
  shows ‹(λx. x - (E a b c d e f g h)) x ≤ (λx. x - (E a b c d e f g h)) x'›
  using assms enat_diff_mono by force
```

We also define abbreviations for performing subtraction.

```
abbreviation ‹subtract_fn a b c d e f g h ≡
  (λx. if somewhere_larger x (E a b c d e f g h) then None else Some (x - (E a b c d e f
g h)))›

abbreviation ‹subtract a b c d e f g h ≡ Some (subtract_fn a b c d e f g h)›
```

4.3 Minimum Updates

Next, we define two energy updates that replace the first component with the minimum of two other components.

```
definition ‹min1_6 e ≡ case e of E a b c d e f g h ⇒ Some (E (min a f) b c d e f g h)›
definition ‹min1_7 e ≡ case e of E a b c d e f g h ⇒ Some (E (min a g) b c d e f g h)›
```

lift order to options

```
instantiation option :: (order) order
begin

definition less_eq_option_def[simp]:
  ‹less_eq_option (optA :: 'a option) optB ≡
    case optA of
      (Some a) ⇒
        (case optB of
          (Some b) ⇒ a ≤ b |
          None ⇒ False) |
      None ⇒ True›

definition less_option_def[simp]:
  ‹less_option (optA :: 'a option) optB ≡ (optA ≤ optB ∧ ¬ optB ≤ optA)›

instance proof standard
  fix x y::‹'a option›
  show ‹(x < y) = (x ≤ y ∧ ¬ y ≤ x)› by simp
next
  fix x::‹'a option›
  show ‹x ≤ x›
    by (simp add: option.case_eq_if)
next
  fix x y z::‹'a option›
  assume ‹x ≤ y› ‹y ≤ z›
  thus ‹x ≤ z›
    unfolding less_eq_option_def
    by (metis option.case_eq_if order_trans)
next
  fix x y::‹'a option›
  assume ‹ x ≤ y› ‹y ≤ x›
  thus ‹x = y›
    unfolding less_eq_option_def
    by (smt (z3) inf.absorb_iff2 le_boolD option.case_eq_if option.split_sel order_antisym)
qed

end
```

Again, we prove that these updates only decrease energies.

```
lemma min_1_6_simps[simp]:
  shows ‹modal_depth (the (min1_6 e)) = min (modal_depth e) (pos_conjuncts e)›
        ‹br_conj_depth (the (min1_6 e)) = br_conj_depth e›
        ‹conj_depth (the (min1_6 e)) = conj_depth e›
        ‹st_conj_depth (the (min1_6 e)) = st_conj_depth e›
        ‹imm_conj_depth (the (min1_6 e)) = imm_conj_depth e›
        ‹pos_conjuncts (the (min1_6 e)) = pos_conjuncts e›
        ‹neg_conjuncts (the (min1_6 e)) = neg_conjuncts e›
        ‹neg_depth (the (min1_6 e)) = neg_depth e›
  unfolding min1_6_def by (simp_all add: energy.case_eq_if)

lemma min_1_7_simps[simp]:
  shows ‹modal_depth (the (min1_7 e)) = min (modal_depth e) (neg_conjuncts e)›
        ‹br_conj_depth (the (min1_7 e)) = br_conj_depth e›
```

```
        ‹conj_depth (the (min1_7 e)) = conj_depth e›
        ‹st_conj_depth (the (min1_7 e)) = st_conj_depth e›
        ‹imm_conj_depth (the (min1_7 e)) = imm_conj_depth e›
        ‹pos_conjuncts (the (min1_7 e)) = pos_conjuncts e›
        ‹neg_conjuncts (the (min1_7 e)) = neg_conjuncts e›
        ‹neg_depth (the (min1_7 e)) = neg_depth e›
  unfolding min1_7_def by (simp_all add: energy.case_eq_if)

lemma min_1_6_some:
  shows ‹min1_6 e ≠ None›
  unfolding min1_6_def
  using energy.case_eq_if by blast

lemma min_1_7_some:
  shows ‹min1_7 e ≠ None›
  unfolding min1_7_def
  using energy.case_eq_if by blast

lemma mono_min_1_6:
  shows ‹mono (the ∘ min1_6)›
proof
  fix x y :: energy
  assume ‹x ≤ y›
  thus ‹(the ∘ min1_6) x ≤ (the ∘ min1_6) y› unfolding leq_components
    using min.mono min_1_6_simps min1_6_def by auto
qed

lemma mono_min_1_7:
  shows ‹mono (the ∘ min1_7)›
proof
  fix x y :: energy
  assume ‹x ≤ y›
  thus ‹(the ∘ min1_7) x ≤ (the ∘ min1_7) y› unfolding leq_components
    using min.mono min_1_7_simps min1_7_def by auto
qed

lemma gets_smaller_min_1_6:
  shows ‹the (min1_6 x) ≤ x›
  using min_1_6_simps min_less_iff_conj somewhere_larger_eq by fastforce


lemma gets_smaller_min_1_7:
  shows ‹the (min1_7 x) ≤ x›
  using min_1_7_simps min_less_iff_conj somewhere_larger_eq by fastforce

lemma min_1_7_lower_end:
  assumes ‹(Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7) = None›
  shows ‹neg_depth e = 0›
  using assms
  by (smt (verit) bind.bind_lunit energy.sel ileI1 leq_components min_1_7_some not_gr_zero
one_eSuc zero_le)

lemma min_1_7_subtr_simp:
  shows ‹(Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)
    = (if neg_depth e = 0 then None
        else Some (E (min (modal_depth e) (neg_conjuncts e)) (br_conj_depth e) (conj_depth
e) (st_conj_depth e) (imm_conj_depth e) (pos_conjuncts e) (neg_conjuncts e) (neg_depth e
- 1)))›
  using min_1_7_lower_end
  by (auto simp add: min1_7_def)
```

```
lemma min_1_7_subtr_mono:
  shows ‹mono (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)›
proof
  fix e1 e2 :: energy
  assume ‹e1 ≤ e2›
  thus ‹(λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7) e1
    ≤  (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7) e2›
    unfolding min_1_7_subtr_simp
    by (auto simp add: min.coboundedI1  min.coboundedI2 enat_diff_mono)
qed

lemma min_1_6_subtr_simp:
  shows ‹(Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6)
    = (if br_conj_depth e = 0 ∨ conj_depth e = 0 then None
        else Some (E (min (modal_depth e) (pos_conjuncts e)) (br_conj_depth e - 1) (conj_depth
e - 1) (st_conj_depth e) (imm_conj_depth e) (pos_conjuncts e) (neg_conjuncts e) (neg_depth
e)))›
  by (auto simp add: min1_6_def ileI1 one_eSuc)

instantiation energy :: Sup
begin

definition ‹Sup ee ≡ E (Sup (modal_depth ‘ ee)) (Sup (br_conj_depth ‘ ee )) (Sup (conj_depth
‘ ee)) (Sup (st_conj_depth ‘ ee))
  (Sup (imm_conj_depth ‘ ee)) (Sup (pos_conjuncts ‘ ee)) (Sup (neg_conjuncts ‘ ee)) (Sup
(neg_depth ‘ ee))›

instance ..
end


end
```

5 Expressiveness Price Function

```
theory Expressiveness_Price
  imports HML_SRBB Energy
begin
```

The expressiveness price function assigns a price - an eight-dimensional vector - to a $\mathrm{HML}_{\mathrm{SRBB}}$ formula. This price is supposed to capture the expressiveness power needed to describe a certain property and will later be used to select subsets of specific expressiveness power associated with the behavioural equivalence characterized by that subset of the $\mathrm{HML}_{\mathrm{SRBB}}$ sublanguage.

The expressiveness price function may be defined as a single function:

$$\begin{aligned}
expr(\top) := expr^{\varepsilon}(\top) :=& 0\\
expr(\langle\varepsilon\rangle\chi) :=& expr^{\varepsilon}(\chi)\\
expr(\bigwedge \Psi) :=& \hat{e}_5 + expr^{\varepsilon}(\bigwedge \Psi)\\
expr^{\varepsilon}((\alpha)\varphi) :=& \hat{e}_1 + expr(\varphi)\\
expr^{\varepsilon}(\bigwedge(\{(\alpha)\varphi\} \cup \Psi)) :=& \hat{e}_2 + expr^{\varepsilon}(\bigwedge(\{\langle\varepsilon\rangle(\alpha)\varphi\} \cup \Psi \setminus \{(\alpha)\varphi\}))\\
expr^{\varepsilon}(\bigwedge \Psi) :=& \sup\{expr^{\wedge}(\psi) | \psi \in \Psi\} + \begin{cases} \hat{e}_4 & \text{if} \neg\langle\tau\rangle\top \in \Psi \\ \hat{e}_3 & \text{otherwise} \end{cases}\\
expr^{\wedge}(\neg\langle\tau\rangle\top) :=& 0\\
expr^{\wedge}(\neg\varphi) :=& \sup\{\hat{e}_8 + expr(\varphi), (0,0,0,0,0,0,expr_1(\varphi),0)\}\\
expr^{\wedge}(\varphi) :=& \sup\{expr(\varphi), (0,0,0,0,0,expr_1(\varphi),0,0)\}
\end{aligned}$$

The eight dimensions are intended to measure the following properties of formulas:

1. Modal depth (of observations $\langle\alpha\rangle$, (α)),
2. Depth of branching conjunctions (with one observation clause not starting with $\langle\varepsilon\rangle$),
3. Depth of stable conjunctions (that do enforce stability by a $\neg\langle\tau\rangle\top$-conjunct),
4. Depth of unstable conjunctions (that do not enforce stability by a $\neg\langle\tau\rangle\top$-conjunct),
5. Depth of immediate conjunctions (that are not preceded by $\langle\varepsilon\rangle$),
6. Maximal modal depth of positive clauses in conjunctions,
7. Maximal modal depth of negative clauses in conjunctions,
8. Depth of negations

Instead of defining the expressiveness price function in one go, we define eight functions (one for each dimension) and then use them in combination to build the the result vector.

Note that since all these functions stem from the above singular function, they all look very similar, but differ mostly in where the 1+ is placed.

5.1 Modal Depth

The (maximal) modal depth (of observations $\langle\alpha\rangle$, (α)) is increased on each:

- `Obs`
- `BranchConj`

```
primrec
      modal_depth_srbb :: ‹('act, 'i) hml_srbb ⇒ enat›
  and modal_depth_srbb_inner :: ‹('act, 'i) hml_srbb_inner ⇒ enat›
  and modal_depth_srbb_conjunct :: ‹('act, 'i) hml_srbb_conjunct ⇒ enat› where
 ‹modal_depth_srbb TT = 0› |
 ‹modal_depth_srbb (Internal χ) = modal_depth_srbb_inner χ› |
 ‹modal_depth_srbb (ImmConj I ψs) = Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)› |

 ‹modal_depth_srbb_inner (Obs α φ) = 1 + modal_depth_srbb φ› |
 ‹modal_depth_srbb_inner (Conj I ψs) =
    Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)› |
 ‹modal_depth_srbb_inner (StableConj I ψs) =
    Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)› |
 ‹modal_depth_srbb_inner (BranchConj a φ I ψs) =
    Sup ({1 + modal_depth_srbb φ} ∪ ((modal_depth_srbb_conjunct ∘ ψs) ‘ I))› |

 ‹modal_depth_srbb_conjunct (Pos χ) = modal_depth_srbb_inner χ› |
 ‹modal_depth_srbb_conjunct (Neg χ) = modal_depth_srbb_inner χ›

lemma ‹modal_depth_srbb TT = 0›
  using Sup_enat_def by simp

lemma ‹modal_depth_srbb (Internal (Obs α (Internal (BranchConj β TT {} ψs2)))) = 2›
  using Sup_enat_def by simp

fun observe_n_alphas :: ‹'a ⇒ nat ⇒ ('a, nat) hml_srbb› where
  ‹observe_n_alphas α 0 = TT› |
  ‹observe_n_alphas α (Suc n) = Internal (Obs α (observe_n_alphas α n))›

lemma obs_n_α_depth_n: ‹modal_depth_srbb (observe_n_alphas α n) = n›
proof (induct n)
  case 0
  show ?case unfolding observe_n_alphas.simps(1) and modal_depth_srbb.simps(2)
    using zero_enat_def and Sup_enat_def by force
next
  case (Suc n)
  then show ?case
    using eSuc_enat plus_1_eSuc(1) by auto
qed

lemma sup_nats_in_enats_infinite: ‹(SUP x∈ℕ. enat x) = ∞›
  by (metis Nats_infinite Sup_enat_def enat.inject finite.emptyI finite_imageD inj_on_def)

lemma sucs_of_nats_in_enats_sup_infinite: ‹(SUP x∈ℕ. 1 + enat x) = ∞›
  using sup_nats_in_enats_infinite
  by (metis Sup.SUP_cong eSuc_Sup eSuc_infinity image_image image_is_empty plus_1_eSuc(1))

lemma ‹modal_depth_srbb (ImmConj ℕ (λn. Pos (Obs α (observe_n_alphas α n)))) = ∞›
  unfolding modal_depth_srbb.simps(3)
        and o_def
        and modal_depth_srbb_conjunct.simps(1)
        and modal_depth_srbb_inner.simps(1)
        and obs_n_α_depth_n
  by (metis sucs_of_nats_in_enats_sup_infinite)
```

5.2 Depth of Branching Conjunctions

The depth of branching conjunctions (with one observation clause not starting with $\langle\varepsilon\rangle$) is increased on each:

- `BranchConj` if there are other conjuncts besides the branching conjunct

Note that if the `BranchConj` is empty (has no other conjuncts), then it is treated like a simple `Obs`.

```
primrec
      branching_conjunction_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and branch_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and branch_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹branching_conjunction_depth TT = 0› |
  ‹branching_conjunction_depth (Internal χ) = branch_conj_depth_inner χ› |
  ‹branching_conjunction_depth (ImmConj I ψs) = Sup ((branch_conj_depth_conjunct ∘ ψs) `
I)› |

  ‹branch_conj_depth_inner (Obs _ φ) = branching_conjunction_depth φ› |
  ‹branch_conj_depth_inner (Conj I ψs) = Sup ((branch_conj_depth_conjunct ∘ ψs) ` I)› |
  ‹branch_conj_depth_inner (StableConj I ψs) = Sup ((branch_conj_depth_conjunct ∘ ψs) `
I)› |
  ‹branch_conj_depth_inner (BranchConj _ φ I ψs) =
     1 + Sup ({branching_conjunction_depth φ} ∪ ((branch_conj_depth_conjunct ∘ ψs) ` I))›
|

  ‹branch_conj_depth_conjunct (Pos χ) = branch_conj_depth_inner χ› |
  ‹branch_conj_depth_conjunct (Neg χ) = branch_conj_depth_inner χ›
```

5.3 Depth of Stable Conjunctions

The depth of stable conjunctions (that do enforce stability by a $\neg\langle\tau\rangle\top$-conjunct) is increased on each:

- `StableConj`

Note that if the `StableConj` is empty (has no other conjuncts), it is still counted.

```
primrec
      stable_conjunction_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and st_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and st_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹stable_conjunction_depth TT = 0› |
  ‹stable_conjunction_depth (Internal χ) = st_conj_depth_inner χ› |
  ‹stable_conjunction_depth (ImmConj I ψs) = Sup ((st_conj_depth_conjunct ∘ ψs) ` I)› |

  ‹st_conj_depth_inner (Obs _ φ) = stable_conjunction_depth φ› |
  ‹st_conj_depth_inner (Conj I ψs) = Sup ((st_conj_depth_conjunct ∘ ψs) ` I)› |
  ‹st_conj_depth_inner (StableConj I ψs) = 1 + Sup ((st_conj_depth_conjunct ∘ ψs) ` I)›
|
  ‹st_conj_depth_inner (BranchConj _ φ I ψs) = Sup ({stable_conjunction_depth φ} ∪ ((st_conj_depth_conju
∘ ψs) ` I))› |

  ‹st_conj_depth_conjunct (Pos χ) = st_conj_depth_inner χ› |
  ‹st_conj_depth_conjunct (Neg χ) = st_conj_depth_inner χ›
```

5.4 Depth of Instable Conjunctions

The depth of unstable conjunctions (that do not enforce stability by a $\neg\langle\tau\rangle\top$-conjunct) is increased on each:

- `ImmConj` if there are conjuncts (i.e. $\bigwedge\{\}$ is not counted)
- `Conj` if there are conjuncts, (i.e. the conjunction is not empty)
- `BranchConj` if there are other conjuncts besides the branching conjunct

Note that if the `BranchConj` is empty (has no other conjuncts), then it is treated like a simple `Obs`.

```
primrec
      unstable_conjunction_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and inst_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and inst_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹unstable_conjunction_depth TT = 0› |
  ‹unstable_conjunction_depth (Internal χ) = inst_conj_depth_inner χ› |
  ‹unstable_conjunction_depth (ImmConj I ψs) =
    (if I = {}
     then 0
     else 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ` I))› |

  ‹inst_conj_depth_inner (Obs _ φ) = unstable_conjunction_depth φ› |
  ‹inst_conj_depth_inner (Conj I ψs) =
    (if I = {}
     then 0
     else 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ` I))› |
  ‹inst_conj_depth_inner (StableConj I ψs) = Sup ((inst_conj_depth_conjunct ∘ ψs) ` I)›
|
  ‹inst_conj_depth_inner (BranchConj _ φ I ψs) =
    1 + Sup ({unstable_conjunction_depth φ} ∪ ((inst_conj_depth_conjunct ∘ ψs) ` I))› |

  ‹inst_conj_depth_conjunct (Pos χ) = inst_conj_depth_inner χ› |
  ‹inst_conj_depth_conjunct (Neg χ) = inst_conj_depth_inner χ›
```

5.5 Depth of Immediate Conjunctions

The depth of immediate conjunctions (that are not preceded by $\langle\varepsilon\rangle$) is increased on each:

- `ImmConj` if there are conjuncts (i.e. $\bigwedge\{\}$ is not counted)

```
primrec
      immediate_conjunction_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and imm_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and imm_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹immediate_conjunction_depth TT = 0› |
  ‹immediate_conjunction_depth (Internal χ) = imm_conj_depth_inner χ› |
  ‹immediate_conjunction_depth (ImmConj I ψs) =
    (if I = {}
     then 0
     else 1 + Sup ((imm_conj_depth_conjunct ∘ ψs) ` I))› |

  ‹imm_conj_depth_inner (Obs _ φ) = immediate_conjunction_depth φ› |
  ‹imm_conj_depth_inner (Conj I ψs) = Sup ((imm_conj_depth_conjunct ∘ ψs) ` I)› |
  ‹imm_conj_depth_inner (StableConj I ψs) = Sup ((imm_conj_depth_conjunct ∘ ψs) ` I)› |
  ‹imm_conj_depth_inner (BranchConj _ φ I ψs) = Sup ({immediate_conjunction_depth φ} ∪
((imm_conj_depth_conjunct ∘ ψs) ` I))› |

  ‹imm_conj_depth_conjunct (Pos χ) = imm_conj_depth_inner χ› |
  ‹imm_conj_depth_conjunct (Neg χ) = imm_conj_depth_inner χ›
```

5.6 Maximal Modal Depth of Positive Clauses in Conjunctions

Now, we take a look at the maximal modal depth of positive clauses in conjunctions.
This counter calculates the modal depth for every positive clause in a conjunction (`Pos` χ).

```
primrec
      max_positive_conjunct_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and max_pos_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and max_pos_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹max_positive_conjunct_depth TT = 0› |
  ‹max_positive_conjunct_depth (Internal χ) = max_pos_conj_depth_inner χ› |
```

```
  ‹max_positive_conjunct_depth (ImmConj I ψs) = Sup ((max_pos_conj_depth_conjunct ∘ ψs)
‘ I)› |

  ‹max_pos_conj_depth_inner (Obs _ φ) = max_positive_conjunct_depth φ› |
  ‹max_pos_conj_depth_inner (Conj I ψs) = Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I)›
|
  ‹max_pos_conj_depth_inner (StableConj I ψs) = Sup ((max_pos_conj_depth_conjunct ∘ ψs)
‘ I)› |
  ‹max_pos_conj_depth_inner (BranchConj _ φ I ψs) = Sup ({1 + modal_depth_srbb φ, max_positive_conjunct_
φ} ∪ ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹max_pos_conj_depth_conjunct (Pos χ) = modal_depth_srbb_inner χ› |
  ‹max_pos_conj_depth_conjunct (Neg χ) = max_pos_conj_depth_inner χ›

lemma modal_depth_dominates_pos_conjuncts:
  fixes
    φ::‹('a, 's) hml_srbb› and
    χ::‹('a, 's) hml_srbb_inner› and
    ψ::‹('a, 's) hml_srbb_conjunct›
  shows
    ‹(max_positive_conjunct_depth φ ≤ modal_depth_srbb φ)
    ∧ (max_pos_conj_depth_inner χ ≤ modal_depth_srbb_inner χ)
    ∧ (max_pos_conj_depth_conjunct ψ ≤ modal_depth_srbb_conjunct ψ)›
  using hml_srbb_hml_srbb_inner_hml_srbb_conjunct.induct[of
        ‹λφ::('a, 's) hml_srbb. max_positive_conjunct_depth φ ≤ modal_depth_srbb φ›
        ‹λχ. max_pos_conj_depth_inner χ ≤ modal_depth_srbb_inner χ›
        ‹λψ. max_pos_conj_depth_conjunct ψ ≤ modal_depth_srbb_conjunct ψ›]
  by (auto simp add: SUP_mono' add_increasing sup.coboundedI1 sup.coboundedI2)
```

5.7 Maximal Modal Depth of Negative Clauses in Conjunctions

We take a look at the maximal modal depth of negative clauses in conjunctions.
This counter calculates the modal depth for every negative clause in a conjunction (`Neg` χ).

```
primrec
      max_negative_conjunct_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and max_neg_conj_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and max_neg_conj_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹max_negative_conjunct_depth TT = 0› |
  ‹max_negative_conjunct_depth (Internal χ) = max_neg_conj_depth_inner χ› |
  ‹max_negative_conjunct_depth (ImmConj I ψs) = Sup ((max_neg_conj_depth_conjunct ∘ ψs)
‘ I)› |

  ‹max_neg_conj_depth_inner (Obs _ φ) = max_negative_conjunct_depth φ› |
  ‹max_neg_conj_depth_inner (Conj I ψs) = Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I)›
|
  ‹max_neg_conj_depth_inner (StableConj I ψs) = Sup ((max_neg_conj_depth_conjunct ∘ ψs)
‘ I)› |
  ‹max_neg_conj_depth_inner (BranchConj _ φ I ψs) = Sup ({max_negative_conjunct_depth φ}
∪ ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I))› |

  ‹max_neg_conj_depth_conjunct (Pos χ) = max_neg_conj_depth_inner χ› |
  ‹max_neg_conj_depth_conjunct (Neg χ) = modal_depth_srbb_inner χ›



lemma modal_depth_dominates_neg_conjuncts:
  fixes
    φ::‹('a, 's) hml_srbb› and
    χ::‹('a, 's) hml_srbb_inner› and
    ψ::‹('a, 's) hml_srbb_conjunct›
```

```
  shows
    ‹(max_negative_conjunct_depth φ ≤ modal_depth_srbb φ)
    ∧ (max_neg_conj_depth_inner χ ≤ modal_depth_srbb_inner χ)
    ∧ (max_neg_conj_depth_conjunct ψ ≤ modal_depth_srbb_conjunct ψ)›
  using hml_srbb_hml_srbb_inner_hml_srbb_conjunct.induct[of
        ‹λφ::('a, 's) hml_srbb. max_negative_conjunct_depth φ ≤ modal_depth_srbb φ›
        ‹λχ. max_neg_conj_depth_inner χ ≤ modal_depth_srbb_inner χ›
        ‹λψ. max_neg_conj_depth_conjunct ψ ≤ modal_depth_srbb_conjunct ψ›]
  by (auto simp add: SUP_mono' add_increasing sup.coboundedI1 sup.coboundedI2)
```

5.8 Depth of Negations

The depth of negations (occurrences of `Neg χ` on a path of the syntax tree) is increased on each:

- `Neg χ`

```
primrec
      negation_depth :: ‹('a, 's) hml_srbb ⇒ enat›
  and neg_depth_inner :: ‹('a, 's) hml_srbb_inner ⇒ enat›
  and neg_depth_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ enat› where
  ‹negation_depth TT = 0› |
  ‹negation_depth (Internal χ) = neg_depth_inner χ› |
  ‹negation_depth (ImmConj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ` I)› |

  ‹neg_depth_inner (Obs _ φ) = negation_depth φ› |
  ‹neg_depth_inner (Conj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ` I)› |
  ‹neg_depth_inner (StableConj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ` I)› |
  ‹neg_depth_inner (BranchConj _ φ I ψs) = Sup ({negation_depth φ} ∪ ((neg_depth_conjunct
∘ ψs) ` I))› |

  ‹neg_depth_conjunct (Pos χ) = neg_depth_inner χ› |
  ‹neg_depth_conjunct (Neg χ) = 1 + neg_depth_inner χ›
```

5.9 Expressiveness Price Function

The `expressiveness_price` function combines the eight functions into one.

```
fun expressiveness_price :: ‹('a, 's) hml_srbb ⇒ energy› where
  ‹expressiveness_price φ = E (modal_depth_srbb             φ)
                              (branching_conjunction_depth φ)
                              (unstable_conjunction_depth  φ)
                              (stable_conjunction_depth    φ)
                              (immediate_conjunction_depth φ)
                              (max_positive_conjunct_depth φ)
                              (max_negative_conjunct_depth φ)
                              (negation_depth              φ)›
```

Here, we can see the decomposed price of an immediate conjunction:

```
lemma expressiveness_price_ImmConj_def:
  shows ‹expressiveness_price (ImmConj I ψs) = E
    (Sup ((modal_depth_srbb_conjunct ∘ ψs) ` I))
    (Sup ((branch_conj_depth_conjunct ∘ ψs) ` I))
    (if I = {} then 0 else 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((st_conj_depth_conjunct ∘ ψs) ` I))
    (if I = {} then 0 else 1 + Sup ((imm_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((max_pos_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((max_neg_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((neg_depth_conjunct ∘ ψs) ` I))› by simp

lemma expressiveness_price_ImmConj_non_empty_def:
  assumes ‹I ≠ {}›
```

```
  shows ‹expressiveness_price (ImmConj I ψs) = E
    (Sup ((modal_depth_srbb_conjunct ∘ ψs) ` I))
    (Sup ((branch_conj_depth_conjunct ∘ ψs) ` I))
    (1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((st_conj_depth_conjunct ∘ ψs) ` I))
    (1 + Sup ((imm_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((max_pos_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((max_neg_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((neg_depth_conjunct ∘ ψs) ` I))› using assms  by simp

lemma expressiveness_price_ImmConj_empty_def:
  assumes ‹I = {}›
  shows ‹expressiveness_price (ImmConj I ψs) = E 0 0 0 0 0 0 0 0› using assms
  unfolding expressiveness_price_ImmConj_def by (simp add: bot_enat_def)
```

We can now define a sublanguage of Hennessy-Milner Logic $\mathcal{O}$ by the set of formulas with prices below an energy coordinate.

```
definition 𝒪 :: ‹energy ⇒ (('a, 's) hml_srbb) set› where
  ‹𝒪 energy ≡ {φ . expressiveness_price φ ≤ energy}›

lemma 𝒪_sup: ‹UNIV = 𝒪 (E ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞)› unfolding 𝒪_def by auto
```

Formalizing HML_{SRBB} by mutually recursive data types leads to expressiveness price functions of these other types, namely `expr_pr_inner` and `expr_pr_conjunct`, and corresponding definitions and lemmas.

```
fun expr_pr_inner :: ‹('a, 's) hml_srbb_inner ⇒ energy› where
  ‹expr_pr_inner χ = E (modal_depth_srbb_inner χ)
                 (branch_conj_depth_inner χ)
                 (inst_conj_depth_inner χ)
                 (st_conj_depth_inner χ)
                 (imm_conj_depth_inner χ)
                 (max_pos_conj_depth_inner χ)
                 (max_neg_conj_depth_inner χ)
                 (neg_depth_inner χ)›

definition 𝒪_inner :: ‹energy ⇒ (('a, 's) hml_srbb_inner) set› where
  ‹𝒪_inner energy ≡ {χ . expr_pr_inner χ ≤ energy}›

fun expr_pr_conjunct :: ‹('a, 's) hml_srbb_conjunct ⇒ energy› where
  ‹expr_pr_conjunct ψ = E (modal_depth_srbb_conjunct ψ)
                 (branch_conj_depth_conjunct ψ)
                 (inst_conj_depth_conjunct ψ)
                 (st_conj_depth_conjunct ψ)
                 (imm_conj_depth_conjunct ψ)
                 (max_pos_conj_depth_conjunct ψ)
                 (max_neg_conj_depth_conjunct ψ)
                 (neg_depth_conjunct ψ)›

definition 𝒪_conjunct :: ‹energy ⇒ (('a, 's) hml_srbb_conjunct) set› where
  ‹𝒪_conjunct energy ≡ {χ . expr_pr_conjunct χ ≤ energy}›
```

5.10 Prices of Certain Formulas

We demonstrate the pricing mechanisms for various formulas. These proofs operate under the assumption of an expressiveness price `e` for a given formula χ and proceed to determine the price of a derived formula such as `Pos` χ. The proofs all are of a similar nature. They decompose the expression function(s) into their constituent parts and apply their definitions to the constructed formula ((`Pos` χ)).

```
context LTS_Tau
begin
```

For example, here, we establish that the expressiveness price of `Internal` χ is equal to the expressiveness price of χ.

```
lemma expr_internal_eq:
  shows ‹expressiveness_price (Internal χ) = expr_pr_inner χ›
proof-
  have expr_internal: ‹expressiveness_price (Internal χ) = E (modal_depth_srbb (Internal
χ))
                              (branching_conjunction_depth (Internal χ))
                              (unstable_conjunction_depth  (Internal χ))
                              (stable_conjunction_depth    (Internal χ))
                              (immediate_conjunction_depth (Internal χ))
                              (max_positive_conjunct_depth (Internal χ))
                              (max_negative_conjunct_depth (Internal χ))
                              (negation_depth              (Internal χ))›
            using expressiveness_price.simps by blast
          have ‹modal_depth_srbb (Internal χ) = modal_depth_srbb_inner χ›
            ‹(branching_conjunction_depth (Internal χ)) = branch_conj_depth_inner χ›
            ‹(unstable_conjunction_depth  (Internal χ)) = inst_conj_depth_inner χ›
            ‹(stable_conjunction_depth    (Internal χ)) = st_conj_depth_inner χ›
            ‹(immediate_conjunction_depth (Internal χ)) = imm_conj_depth_inner χ›
            ‹max_positive_conjunct_depth (Internal χ) = max_pos_conj_depth_inner χ›
            ‹max_negative_conjunct_depth (Internal χ) = max_neg_conj_depth_inner χ›
            ‹negation_depth (Internal χ) = neg_depth_inner χ›
            by simp+
          with expr_internal show ?thesis
            by auto
        qed
```

If the price of a formula χ is not greater than the minimum update `min1_6` apllied to some energy e, then `Pos` χ is not greater than `e`.

```
lemma expr_pos:
  assumes ‹expr_pr_inner χ ≤ the (min1_6 e)›
  shows ‹expr_pr_conjunct (Pos χ) ≤ e›
proof-
  have expr_internal: ‹expr_pr_conjunct (Pos χ) = E (modal_depth_srbb_conjunct (Pos χ))
                              (branch_conj_depth_conjunct (Pos χ))
                              (inst_conj_depth_conjunct   (Pos χ))
                              (st_conj_depth_conjunct     (Pos χ))
                              (imm_conj_depth_conjunct (Pos χ))
                              (max_pos_conj_depth_conjunct (Pos χ))
                              (max_neg_conj_depth_conjunct (Pos χ))
                              (neg_depth_conjunct            (Pos χ))›
            using expr_pr_conjunct.simps by blast
  have pos_upd: ‹(modal_depth_srbb_conjunct (Pos χ)) = modal_depth_srbb_inner χ›
                ‹(branch_conj_depth_conjunct (Pos χ)) = branch_conj_depth_inner χ›
                ‹(inst_conj_depth_conjunct   (Pos χ)) = inst_conj_depth_inner χ›
                ‹(st_conj_depth_conjunct     (Pos χ)) = st_conj_depth_inner χ›
                ‹(imm_conj_depth_conjunct (Pos χ)) = imm_conj_depth_inner χ›
                ‹(max_pos_conj_depth_conjunct (Pos χ)) = modal_depth_srbb_inner χ›
                ‹(max_neg_conj_depth_conjunct (Pos χ)) = max_neg_conj_depth_inner χ›
                ‹(neg_depth_conjunct            (Pos χ)) = neg_depth_inner χ›
    by simp+
  obtain e1 e2 e3 e4 e5 e6 e7 e8 where ‹e = E e1 e2 e3 e4 e5 e6 e7 e8›
    by (metis energy.exhaust_sel)
  hence ‹min1_6 e = Some (E (min e1 e6) e2 e3 e4 e5 e6 e7 e8)›
    by (simp add: min1_6_def)
  hence ‹modal_depth_srbb_inner χ ≤ (min e1 e6)›
    using assms leq_components by fastforce
  hence ‹modal_depth_srbb_inner χ ≤ e6›
    using min.boundedE by blast
```

```
  thus ‹expr_pr_conjunct (Pos χ) ≤ e›
    using expr_internal pos_upd ‹e = E e1 e2 e3 e4 e5 e6 e7 e8› assms leq_components by
auto
qed

lemma expr_neg:
  assumes
    ‹expr_pr_inner χ ≤ e'›
    ‹(Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7) = Some e'›
  shows ‹expr_pr_conjunct (Neg χ) ≤ e›
proof-
  have expr_neg: ‹expr_pr_conjunct (Neg χ) =
    E (modal_depth_srbb_conjunct (Neg χ))
      (branch_conj_depth_conjunct (Neg χ))
      (inst_conj_depth_conjunct (Neg χ))
      (st_conj_depth_conjunct (Neg χ))
      (imm_conj_depth_conjunct (Neg χ))
      (max_pos_conj_depth_conjunct (Neg χ))
      (max_neg_conj_depth_conjunct (Neg χ))
      (neg_depth_conjunct (Neg χ))›
    using expr_pr_conjunct.simps by blast
  have neg_ups:
    ‹modal_depth_srbb_conjunct (Neg χ) = modal_depth_srbb_inner χ›
    ‹(branch_conj_depth_conjunct (Neg χ)) = branch_conj_depth_inner χ›
    ‹inst_conj_depth_conjunct (Neg χ) = inst_conj_depth_inner χ›
    ‹st_conj_depth_conjunct (Neg χ) = st_conj_depth_inner χ›
    ‹imm_conj_depth_conjunct (Neg χ) = imm_conj_depth_inner χ›
    ‹max_pos_conj_depth_conjunct (Neg χ) = max_pos_conj_depth_inner χ›
    ‹max_neg_conj_depth_conjunct (Neg χ) = modal_depth_srbb_inner χ›
    ‹neg_depth_conjunct (Neg χ) = 1 + neg_depth_inner χ›
    by simp+
  obtain e1 e2 e3 e4 e5 e6 e7 e8 where e_def: ‹e = E e1 e2 e3 e4 e5 e6 e7 e8›
    by (metis energy.exhaust_sel)
  hence is_some: ‹(subtract_fn 0 0 0 0 0 0 0 1 e = Some (E e1 e2 e3 e4 e5 e6 e7 (e8-1)))›
    using assms bind_eq_None_conv by fastforce
  hence ‹modal_depth_srbb_inner χ ≤ (min e1 e7)›
    using assms expr_pr_inner.simps leq_components min_1_7_subtr_simp e_def
    by (metis energy.sel(1) energy.sel(7) option.discI option.inject)
  moreover have ‹neg_depth_inner χ ≤ (e8-1)›
    using e_def is_some energy_minus leq_components min_1_7_simps assms
    by (smt (verit, ccfv_threshold) bind.bind_lunit energy.sel(8) expr_pr_inner.simps option.sel)
  moreover hence ‹neg_depth_conjunct (Neg χ) ≤ e8›
    using ‹neg_depth_conjunct (Neg χ) = 1 + neg_depth_inner χ›
    by (metis is_some add_diff_assoc_enat add_diff_cancel_enat e_def enat.simps(3)
        enat_defs(2) enat_diff_mono energy.sel(8) leq_components linorder_not_less
        option.distinct(1) order_le_less)
  ultimately show ‹expr_pr_conjunct (Neg χ) ≤ e›
    using expr_neg e_def is_some assms neg_ups assms leq_components min_1_7_subtr_simp
    by (metis energy.sel expr_pr_inner.simps min.bounded_iff option.distinct(1) option.inject)
qed

lemma expr_obs:
  assumes
    ‹expressiveness_price φ ≤ e'›
    ‹subtract_fn 1 0 0 0 0 0 0 0 e = Some e'›
  shows ‹expr_pr_inner (Obs α φ) ≤ e›
proof-
  have expr_pr_obs:
    ‹expr_pr_inner (Obs α φ) =
      (E (modal_depth_srbb_inner (Obs α φ))
      (branch_conj_depth_inner (Obs α φ))
```

```
      (inst_conj_depth_inner (Obs α φ))
      (st_conj_depth_inner (Obs α φ))
      (imm_conj_depth_inner (Obs α φ))
      (max_pos_conj_depth_inner (Obs α φ))
      (max_neg_conj_depth_inner (Obs α φ))
      (neg_depth_inner (Obs α φ)))›
    using expr_pr_inner.simps by blast
  have obs_upds:
    ‹modal_depth_srbb_inner (Obs α φ) = 1 + modal_depth_srbb φ›
    ‹branch_conj_depth_inner (Obs α φ) = branching_conjunction_depth φ›
    ‹inst_conj_depth_inner (Obs α φ) = unstable_conjunction_depth φ›
    ‹st_conj_depth_inner (Obs α φ) = stable_conjunction_depth φ›
    ‹imm_conj_depth_inner (Obs α φ) = immediate_conjunction_depth φ›
    ‹max_pos_conj_depth_inner (Obs α φ) = max_positive_conjunct_depth φ›
    ‹max_neg_conj_depth_inner (Obs α φ) = max_negative_conjunct_depth φ›
    ‹neg_depth_inner (Obs α φ) = negation_depth φ›
    by simp_all
  obtain e1 e2 e3 e4 e5 e6 e7 e8 where e_def: ‹e = E e1 e2 e3 e4 e5 e6 e7 e8›
    by (metis energy.exhaust_sel)
  then have is_some: ‹(subtract_fn 1 0 0 0 0 0 0 0 e = Some (E (e1-1) e2 e3 e4 e5 e6 e7
e8))›
    using energy_minus idiff_0_right assms
    by (metis option.discI)
  hence ‹modal_depth_srbb φ ≤ (e1 - 1)›
    using assms
    by (auto simp add: e_def)
  hence ‹modal_depth_srbb_inner (Obs α φ) ≤ e1›
    using obs_upds is_some
    unfolding leq_components e_def
    by (metis add_diff_assoc_enat add_diff_cancel_enat antisym enat.simps(3) enat_defs(2)
        enat_diff_mono energy.sel(1) linorder_linear option.distinct(1))
  then show ?thesis
    using is_some assms
    unfolding  e_def leq_components
    by auto
qed

lemma expr_st_conj:
  assumes
    ‹subtract_fn  0 0 0 1 0 0 0 0 e = Some e'›
    ‹I ≠ {}›
    ‹∀q ∈ I. expr_pr_conjunct (ψs q) ≤ e'›
  shows
    ‹expr_pr_inner (StableConj I ψs) ≤ e›
proof -
  have st_conj_upds:
    ‹modal_depth_srbb_inner (StableConj I ψs) = Sup ((modal_depth_srbb_conjunct ∘ ψs) `
I)›
    ‹branch_conj_depth_inner (StableConj I ψs) = Sup ((branch_conj_depth_conjunct ∘ ψs)
` I)›
    ‹inst_conj_depth_inner (StableConj I ψs) = Sup ((inst_conj_depth_conjunct ∘ ψs) ` I)›
    ‹st_conj_depth_inner (StableConj I ψs) = 1 + Sup ((st_conj_depth_conjunct ∘ ψs) ` I)›
    ‹imm_conj_depth_inner (StableConj I ψs) = Sup ((imm_conj_depth_conjunct ∘ ψs) ` I)›
    ‹max_pos_conj_depth_inner (StableConj I ψs) = Sup ((max_pos_conj_depth_conjunct ∘ ψs)
` I)›
    ‹max_neg_conj_depth_inner (StableConj I ψs) = Sup ((max_neg_conj_depth_conjunct ∘ ψs)
` I)›
    ‹neg_depth_inner (StableConj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ` I)›
    by force+
  obtain e1 e2 e3 e4 e5 e6 e7 e8 where e_def: ‹e = E e1 e2 e3 e4 e5 e6 e7 e8›
    using energy.exhaust_sel by blast
```

```
  hence is_some: ‹subtract_fn  0 0 0 1 0 0 0 0 e = Some (E e1 e2 e3 (e4-1) e5 e6 e7 e8)›
    using assms minus_energy_def
    by (smt (verit, del_insts) energy_minus idiff_0_right option.distinct(1))
  hence
    ‹∀i ∈ I. modal_depth_srbb_conjunct (ψs i) ≤ e1›
    ‹∀i ∈ I. branch_conj_depth_conjunct (ψs i) ≤ e2›
    ‹∀i ∈ I. inst_conj_depth_conjunct (ψs i) ≤ e3›
    ‹∀i ∈ I. st_conj_depth_conjunct (ψs i) ≤ (e4 - 1)›
    ‹∀i ∈ I. imm_conj_depth_conjunct (ψs i) ≤ e5›
    ‹∀i ∈ I. max_pos_conj_depth_conjunct (ψs i) ≤ e6›
    ‹∀i ∈ I. max_neg_conj_depth_conjunct (ψs i) ≤ e7›
    ‹∀i ∈ I. neg_depth_conjunct (ψs i) ≤ e8›
    using assms unfolding leq_components by auto
  hence sups:
    ‹Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I) ≤ e1›
    ‹Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e2›
    ‹Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e3›
    ‹Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I) ≤ (e4 - 1)›
    ‹Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e5›
    ‹Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e6›
    ‹Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e7›
    ‹Sup ((neg_depth_conjunct ∘ ψs) ‘ I) ≤ e8›
    by (simp add: Sup_le_iff)+
  hence ‹st_conj_depth_inner (StableConj I ψs) ≤ e4›
    using e_def is_some minus_energy_def leq_components st_conj_upds(4)
    by (metis add_diff_cancel_enat add_left_mono enat.simps(3) enat_defs(2) energy.sel(4)
le_iff_add option.distinct(1))
  then show ?thesis
    using st_conj_upds sups
    by (simp add: e_def)
qed

lemma expr_imm_conj:
  assumes
    ‹subtract_fn  0 0 0 0 1 0 0 0 e = Some e'›
    ‹I ≠ {}›
    ‹expr_pr_inner (Conj I ψs) ≤ e'›
  shows ‹expressiveness_price (ImmConj I ψs) ≤ e›
proof-
  have conj_upds:
    ‹modal_depth_srbb_inner (Conj I ψs) = Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)›
    ‹branch_conj_depth_inner (Conj I ψs) = Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹inst_conj_depth_inner (Conj I ψs) = 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹st_conj_depth_inner (Conj I ψs) = Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹imm_conj_depth_inner (Conj I ψs) = Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹max_pos_conj_depth_inner (Conj I ψs) = Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹max_neg_conj_depth_inner (Conj I ψs) = Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹neg_depth_inner (Conj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ‘ I)›
    using assms
    by force+
  have imm_conj_upds:
    ‹modal_depth_srbb (ImmConj I ψs) = Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)›
    ‹branching_conjunction_depth (ImmConj I ψs) = Sup ((branch_conj_depth_conjunct ∘ ψs)
‘ I)›
    ‹unstable_conjunction_depth (ImmConj I ψs) = 1 + Sup ((inst_conj_depth_conjunct ∘ ψs)
‘ I)›
    ‹stable_conjunction_depth (ImmConj I ψs) = Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹immediate_conjunction_depth (ImmConj I ψs) = 1 + Sup ((imm_conj_depth_conjunct ∘ ψs)
‘ I)›
    ‹max_positive_conjunct_depth (ImmConj I ψs) = Sup ((max_pos_conj_depth_conjunct ∘ ψs)
‘ I)›
```

```
    ‹max_negative_conjunct_depth (ImmConj I ψs) = Sup ((max_neg_conj_depth_conjunct ∘ ψs)
‘ I)›
    ‹negation_depth (ImmConj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ‘ I)›
    using assms
    by force+
  obtain e1 e2 e3 e4 e5 e6 e7 e8 where e_def: ‹e = E e1 e2 e3 e4 e5 e6 e7 e8›
    using assms by (metis energy.exhaust_sel)
  hence is_some: ‹(e - (E 0 0 0 0 1 0 0 0)) = (E e1 e2 e3 e4 (e5-1) e6 e7 e8)›
    using minus_energy_def
    by simp
  hence ‹e5>0› using assms(1) e_def leq_components by auto
  have
    ‹E (modal_depth_srbb_inner (Conj I ψs))
       (branch_conj_depth_inner (Conj I ψs))
       (inst_conj_depth_inner (Conj I ψs))
       (st_conj_depth_inner (Conj I ψs))
       (imm_conj_depth_inner (Conj I ψs))
       (max_pos_conj_depth_inner (Conj I ψs))
       (max_neg_conj_depth_inner (Conj I ψs))
       (neg_depth_inner (Conj I ψs)) ≤ (E e1 e2 e3 e4 (e5-1) e6 e7 e8)›
    using is_some assms
    by (metis expr_pr_inner.simps option.discI option.inject)
  hence
    ‹(modal_depth_srbb_inner (Conj I ψs))≤ e1›
    ‹(branch_conj_depth_inner (Conj I ψs)) ≤ e2›
    ‹(inst_conj_depth_inner (Conj I ψs)) ≤ e3›
    ‹(st_conj_depth_inner (Conj I ψs))≤ e4›
    ‹(imm_conj_depth_inner (Conj I ψs))≤ (e5-1)›
    ‹(max_pos_conj_depth_inner (Conj I ψs)) ≤ e6›
    ‹(max_neg_conj_depth_inner (Conj I ψs)) ≤ e7›
    ‹(neg_depth_inner (Conj I ψs))≤ e8›
    by auto
  hence E:
    ‹Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I) ≤ e1›
    ‹Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e2›
    ‹1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e3›
    ‹Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e4›
    ‹Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I) ≤ (e5-1)›
    ‹Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e6›
    ‹Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e7›
    ‹Sup ((neg_depth_conjunct ∘ ψs) ‘ I) ≤ e8›
    using conj_upds by force+
  from ‹Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I) ≤ (e5-1)› have ‹(1 + Sup ((imm_conj_depth_conjunct
∘ ψs) ‘ I)) ≤ e5›
    using assms(1) ‹e5>0› is_some e_def add.right_neutral add_diff_cancel_enat enat_add_left_cancel_le
ileI1 le_iff_add plus_1_eSuc(1)
    by metis
  thus ‹expressiveness_price (ImmConj I ψs) ≤ e› using imm_conj_upds E
    by (metis e_def  energy.sel expressiveness_price.elims leD somewhere_larger_eq)

qed

lemma expr_conj:
  assumes
    ‹subtract_fn 0 0 1 0 0 0 0 0 e = Some e'›
    ‹I ≠ {}›
    ‹∀q ∈ I. expr_pr_conjunct (ψs q) ≤ e'›
  shows ‹expr_pr_inner (Conj I ψs) ≤ e›
proof-
  have conj_upds:
    ‹modal_depth_srbb_inner (Conj I ψs) = Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I)›
```

```
    ‹branch_conj_depth_inner (Conj I ψs) = Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹inst_conj_depth_inner (Conj I ψs) = 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹st_conj_depth_inner (Conj I ψs) = Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹imm_conj_depth_inner (Conj I ψs) = Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹max_pos_conj_depth_inner (Conj I ψs) = Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹max_neg_conj_depth_inner (Conj I ψs) = Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I)›
    ‹neg_depth_inner (Conj I ψs) = Sup ((neg_depth_conjunct ∘ ψs) ‘ I)›
    using assms by force+
  obtain e1 e2 e3 e4 e5 e6 e7 e8 where e_def: ‹e = E e1 e2 e3 e4 e5 e6 e7 e8›
    using energy.exhaust_sel by metis
  hence is_some: ‹e - (E 0 0 1 0 0 0 0 0) = E e1 e2 (e3-1) e4 e5 e6 e7 e8›
    using minus_energy_def by simp
  hence ‹e3>0› using assms(1) e_def leq_components by auto
  hence
    ‹∀i ∈ I. modal_depth_srbb_conjunct (ψs i) ≤ e1›
    ‹∀i ∈ I. branch_conj_depth_conjunct (ψs i) ≤ e2›
    ‹∀i ∈ I. inst_conj_depth_conjunct (ψs i) ≤ (e3-1)›
    ‹∀i ∈ I. st_conj_depth_conjunct (ψs i) ≤ e4›
    ‹∀i ∈ I. imm_conj_depth_conjunct (ψs i) ≤ e5›
    ‹∀i ∈ I. max_pos_conj_depth_conjunct (ψs i) ≤ e6›
    ‹∀i ∈ I. max_neg_conj_depth_conjunct (ψs i) ≤ e7›
    ‹∀i ∈ I. neg_depth_conjunct (ψs i) ≤ e8›
    using assms is_some energy.sel leq_components
    by (metis expr_pr_conjunct.elims option.distinct(1) option.inject)+
  hence sups:
    ‹Sup ((modal_depth_srbb_conjunct ∘ ψs) ‘ I) ≤ e1›
    ‹Sup ((branch_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e2›
    ‹Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I) ≤ (e3-1)›
    ‹Sup ((st_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e4›
    ‹Sup ((imm_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e5›
    ‹Sup ((max_pos_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e6›
    ‹Sup ((max_neg_conj_depth_conjunct ∘ ψs) ‘ I) ≤ e7›
    ‹Sup ((neg_depth_conjunct ∘ ψs) ‘ I) ≤ e8›
    by (simp add: Sup_le_iff)+
  hence ‹inst_conj_depth_inner (Conj I ψs) ≤ e3›
    using ‹e3>0› is_some e_def
    unfolding
      ‹inst_conj_depth_inner (Conj I ψs) = 1 + Sup ((inst_conj_depth_conjunct ∘ ψs) ‘ I)›
    by (metis add.right_neutral add_diff_cancel_enat enat_add_left_cancel_le ileI1 le_iff_add
plus_1_eSuc(1))
  then show ?thesis
    using conj_upds sups
    by (simp add: e_def)
qed

lemma expr_br_conj:
  assumes
    ‹subtract_fn 0 1 1 0 0 0 0 0 e = Some e’›
    ‹min1_6 e’ = Some e’’›
    ‹subtract_fn 1 0 0 0 0 0 0 0 e’’ = Some e’’’›
    ‹expressiveness_price φ ≤ e’’’›
    ‹∀q ∈ Q. expr_pr_conjunct (Φ q) ≤ e’›
    ‹1 + modal_depth_srbb φ ≤ pos_conjuncts e›
  shows ‹expr_pr_inner (BranchConj α φ Q Φ) ≤ e›
proof-
  obtain e1 e2 e3 e4 e5 e6 e7 e8 where e_def: ‹e = E e1 e2 e3 e4 e5 e6 e7 e8›
    by (smt (z3) energy.exhaust)
  hence e’’’_def: ‹e’’’ = (E ((min e1 e6)-1) (e2-1) (e3-1) e4 e5 e6 e7 e8)›
    using minus_energy_def
    by (smt (z3) assms energy.sel idiff_0_right min_1_6_simps option.distinct(1) option.sel)
  hence min_vals: ‹the (min1_6 (e - E 0 1 1 0 0 0 0 0)) - (E 1 0 0 0 0 0 0 0) = (E ((min
```

```
e1 e6)-1) (e2-1) (e3-1) e4 e5 e6 e7 e8)›
    using assms
    by (metis not_Some_eq option.sel)
  hence ‹0 < e1› ‹0 < e2› ‹0 < e3› ‹0 < e6›
    using assms energy.sel min_1_6_simps
    unfolding e_def minus_energy_def leq_components
    by (metis (no_types, lifting) gr_zeroI idiff_0_right min_enat_simps(3) not_one_le_zero
option.distinct(1) option.sel, auto)
  have e_comp: ‹e - (E 0 1 1 0 0 0 0 0) = E e1 (e2-1) (e3-1) e4 e5 e6 e7 e8› using e_def
    by simp
  have conj:
    ‹E (modal_depth_srbb             φ)
       (branching_conjunction_depth φ)
       (unstable_conjunction_depth  φ)
       (stable_conjunction_depth    φ)
       (immediate_conjunction_depth φ)
       (max_positive_conjunct_depth φ)
       (max_negative_conjunct_depth φ)
       (negation_depth              φ)
          ≤ ((E ((min e1 e6)-1) (e2-1) (e3-1) e4 e5 e6 e7 e8))›
    using assms e'''_def by force
  hence conj_single:
    ‹modal_depth_srbb φ                ≤ ((min e1 e6)-1)›
    ‹branching_conjunction_depth  φ   ≤ e2 -1›
    ‹(unstable_conjunction_depth  φ)  ≤ e3-1›
    ‹(stable_conjunction_depth    φ)  ≤ e4›
    ‹(immediate_conjunction_depth φ)  ≤ e5›
    ‹(max_positive_conjunct_depth φ)  ≤ e6›
    ‹(max_negative_conjunct_depth φ)  ≤ e7›
    ‹(negation_depth              φ)  ≤ e8›
    using leq_components by auto
  have ‹0 < (min e1 e6)› using ‹0 < e1› ‹0 < e6›
    using min_less_iff_conj by blast
  hence ‹1 + modal_depth_srbb φ ≤ (min e1 e6)›
    using conj_single add.commute add_diff_assoc_enat add_diff_cancel_enat add_right_mono
conj_single(2) i1_ne_infinity ileI1 one_eSuc
    by (metis (no_types, lifting))
  hence ‹1 + modal_depth_srbb φ ≤ e1› ‹1 + modal_depth_srbb φ ≤ e6›
    using min.bounded_iff by blast+
  from conj have ‹1 + branching_conjunction_depth φ ≤ e2›
    by (metis ‹0 < e2› add.commute add_diff_assoc_enat add_diff_cancel_enat add_right_mono
conj_single(2) i1_ne_infinity ileI1 one_eSuc)
  from conj_single have ‹1 + unstable_conjunction_depth φ ≤ e3›
    using ‹0 < e3› add.commute add_diff_assoc_enat add_diff_cancel_enat add_right_mono conj_single(2)
i1_ne_infinity ileI1 one_eSuc
    by (metis (no_types, lifting))
  have branch: ‹∀q∈Q.
    E (modal_depth_srbb_conjunct (Φ q))
      (branch_conj_depth_conjunct  (Φ q))
      (inst_conj_depth_conjunct  (Φ q))
      (st_conj_depth_conjunct  (Φ q))
      (imm_conj_depth_conjunct  (Φ q))
      (max_pos_conj_depth_conjunct  (Φ q))
      (max_neg_conj_depth_conjunct  (Φ q))
      (neg_depth_conjunct  (Φ q))
    ≤ (E e1 (e2-1) (e3-1) e4 e5 e6 e7 e8)›
    using assms e_def e_comp
    by (metis expr_pr_conjunct.simps option.distinct(1) option.sel)
  hence branch_single:
    ‹∀q∈Q. (modal_depth_srbb_conjunct (Φ q)) ≤ e1›
    ‹∀q∈Q. (branch_conj_depth_conjunct  (Φ q)) ≤ (e2-1)›
```

```
    ‹∀q∈Q. (inst_conj_depth_conjunct  (Φ q)) ≤ (e3-1)›
    ‹∀q∈Q. (st_conj_depth_conjunct  (Φ q)) ≤ e4›
    ‹∀q∈Q. (imm_conj_depth_conjunct  (Φ q)) ≤ e5›
    ‹∀q∈Q. (max_pos_conj_depth_conjunct  (Φ q)) ≤ e6›
    ‹∀q∈Q. (max_neg_conj_depth_conjunct  (Φ q)) ≤ e7›
    ‹∀q∈Q. (neg_depth_conjunct  (Φ q)) ≤ e8›
    by auto
  hence ‹∀q∈Q. (1 + branch_conj_depth_conjunct  (Φ q)) ≤ e2›
    by (metis ‹0 < e2› add.commute add_diff_assoc_enat add_diff_cancel_enat add_right_mono
i1_ne_infinity ileI1 one_eSuc)
  from branch_single have ‹∀q∈Q. (1 + inst_conj_depth_conjunct  (Φ q)) ≤ e3›
    using ‹0 < e3›
    by (metis add.commute add_diff_assoc_enat add_diff_cancel_enat add_right_mono i1_ne_infinity
ileI1 one_eSuc)
  have
    ‹expr_pr_inner (BranchConj α φ Q Φ)
    = E (modal_depth_srbb_inner (BranchConj α φ Q Φ))
      (branch_conj_depth_inner (BranchConj α φ Q Φ))
      (inst_conj_depth_inner (BranchConj α φ Q Φ))
      (st_conj_depth_inner (BranchConj α φ Q Φ))
      (imm_conj_depth_inner (BranchConj α φ Q Φ))
      (max_pos_conj_depth_inner (BranchConj α φ Q Φ))
      (max_neg_conj_depth_inner (BranchConj α φ Q Φ))
      (neg_depth_inner (BranchConj α φ Q Φ))› by simp
  hence expr:
    ‹expr_pr_inner (BranchConj α φ Q Φ)
    = E (Sup ({1 + modal_depth_srbb φ} ∪ ((modal_depth_srbb_conjunct ∘ Φ) ‘ Q)))
      (1 + Sup ({branching_conjunction_depth φ} ∪ ((branch_conj_depth_conjunct ∘ Φ) ‘ Q)))
      (1 + Sup ({unstable_conjunction_depth φ} ∪ ((inst_conj_depth_conjunct ∘ Φ) ‘ Q)))
      (Sup ({stable_conjunction_depth φ} ∪ ((st_conj_depth_conjunct ∘ Φ) ‘ Q)))
      (Sup ({immediate_conjunction_depth φ} ∪ ((imm_conj_depth_conjunct ∘ Φ) ‘ Q)))
      (Sup ({1 + modal_depth_srbb φ, max_positive_conjunct_depth φ} ∪ ((max_pos_conj_depth_conjunct
∘ Φ) ‘ Q)))
      (Sup ({max_negative_conjunct_depth φ} ∪ ((max_neg_conj_depth_conjunct ∘ Φ) ‘ Q)))
      (Sup ({negation_depth φ} ∪ ((neg_depth_conjunct ∘ Φ) ‘ Q)))› by auto
  from branch_single ‹1 + modal_depth_srbb φ ≤ e1›
    have ‹∀x ∈ ({1 + modal_depth_srbb φ} ∪ ((modal_depth_srbb_conjunct ∘ Φ) ‘ Q)). x ≤
e1›
    by fastforce
  hence e1_le: ‹(Sup ({1 + modal_depth_srbb φ} ∪ ((modal_depth_srbb_conjunct ∘ Φ) ‘ Q)))
≤ e1›
    using Sup_least by blast
  have ‹∀x ∈ {branching_conjunction_depth φ} ∪ ((branch_conj_depth_conjunct ∘ Φ) ‘ Q).
x ≤ e2 -1›
    using branch_single conj_single comp_apply image_iff insertE by auto
  hence e2_le: ‹1 + Sup ({branching_conjunction_depth φ} ∪ ((branch_conj_depth_conjunct
∘ Φ) ‘ Q)) ≤ e2›
    using Sup_least
    by (metis Un_insert_left ‹0 < e2› add.commute eSuc_minus_1 enat_add_left_cancel_le ileI1
le_iff_add one_eSuc plus_1_eSuc(2) sup_bot_left)
  have ‹∀x ∈ ({unstable_conjunction_depth φ} ∪ ((inst_conj_depth_conjunct ∘ Φ) ‘ Q)). x
≤ e3-1›
    using conj_single branch_single
    using comp_apply image_iff insertE by auto
  hence e3_le: ‹1 + Sup ({unstable_conjunction_depth φ} ∪ ((inst_conj_depth_conjunct ∘ Φ)
‘ Q)) ≤ e3›
    using Un_insert_left ‹0<e3›  add.commute eSuc_minus_1 enat_add_left_cancel_le ileI1
le_iff_add one_eSuc plus_1_eSuc(2) sup_bot_left
    by (metis Sup_least)
  have fa:
    ‹∀x ∈ ({stable_conjunction_depth φ} ∪ ((st_conj_depth_conjunct ∘ Φ) ‘ Q)). x ≤ e4›
```

```
    ‹∀x ∈ ({immediate_conjunction_depth φ} ∪ ((imm_conj_depth_conjunct ∘ Φ) ` Q)). x ≤
e5›
    ‹∀x ∈ ({1 + modal_depth_srbb φ, max_positive_conjunct_depth φ} ∪ ((max_pos_conj_depth_conjunct
∘ Φ) ` Q)). x ≤ e6›
    ‹∀x ∈ ({max_negative_conjunct_depth φ} ∪ ((max_neg_conj_depth_conjunct ∘ Φ) ` Q)).
x ≤ e7›
    ‹∀x ∈ ({negation_depth φ} ∪ ((neg_depth_conjunct ∘ Φ) ` Q)). x ≤ e8›
      using conj_single branch_single ‹1 + modal_depth_srbb φ ≤ e6› by auto
  hence
    ‹(Sup ({stable_conjunction_depth φ} ∪ ((st_conj_depth_conjunct ∘ Φ) ` Q))) ≤ e4›
    ‹(Sup ({immediate_conjunction_depth φ} ∪ ((imm_conj_depth_conjunct ∘ Φ) ` Q))) ≤ e5›
    ‹(Sup ({1 + modal_depth_srbb φ, max_positive_conjunct_depth φ} ∪ ((max_pos_conj_depth_conjunct
∘ Φ) ` Q))) ≤ e6›
    ‹(Sup ({max_negative_conjunct_depth φ} ∪ ((max_neg_conj_depth_conjunct ∘ Φ) ` Q))) ≤
e7›
    ‹(Sup ({negation_depth φ} ∪ ((neg_depth_conjunct ∘ Φ) ` Q))) ≤ e8›
    using Sup_least
    by metis+
  thus ‹expr_pr_inner (BranchConj α φ Q Φ) ≤ e›
    using expr e3_le e2_le e1_le e_def energy.sel leq_components by presburger
qed

lemma expressiveness_price_ImmConj_geq_parts:
  assumes ‹i ∈ I›
  shows ‹expressiveness_price (ImmConj I ψs) - E 0 0 1 0 1 0 0 0 ≥ expr_pr_conjunct (ψs
i)›
proof-
  from assms have ‹I ≠ {}› by blast
  from expressiveness_price_ImmConj_non_empty_def[OF ‹I ≠ {}›]
  have ‹expressiveness_price (ImmConj I ψs) ≥ E 0 0 1 0 1 0 0 0›
    using energy_leq_cases by force
  hence
  ‹expressiveness_price (ImmConj I ψs) - E 0 0 1 0 1 0 0 0 = E
    (Sup ((modal_depth_srbb_conjunct ∘ ψs) ` I))
    (Sup ((branch_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((inst_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((st_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((imm_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((max_pos_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((max_neg_conj_depth_conjunct ∘ ψs) ` I))
    (Sup ((neg_depth_conjunct ∘ ψs) ` I))›
    unfolding expressiveness_price_ImmConj_non_empty_def[OF ‹I ≠ {}›]
    by simp
  also have ‹... ≥ expr_pr_conjunct (ψs i)›
    using assms ‹I ≠ {}› SUP_upper unfolding leq_components by fastforce
  finally show ?thesis .
qed

lemma expressiveness_price_ImmConj_geq_parts':
  assumes ‹i ∈ I›
  shows ‹(expressiveness_price (ImmConj I ψs) - E 0 0 0 0 1 0 0 0) - E 0 0 1 0 0 0 0 0 ≥
expr_pr_conjunct (ψs i)›
  using expressiveness_price_ImmConj_geq_parts[OF assms]
    less_eq_energy_def minus_energy_def
  by (smt (z3) energy.sel idiff_0_right)

end
```

Here, we show the prices for some specific formulas.

```
locale Inhabited_LTS = LTS step
  for step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ↦ _ _› [70,70,70] 80) +
```

```
  fixes left :: 's
    and right :: 's
  assumes left_right_distinct: ‹(left::'s) ≠ (right::'s)›

begin

lemma example_φ_cp:
  fixes op::‹'a› and a::‹'a› and b::‹'a›
  defines φ: ‹φ ≡
    (Internal
      (Obs op
        (Internal
          (Conj {left, right}
                (λi. (if i = left
                      then (Pos (Obs a TT))
                      else if i = right
                           then (Pos (Obs b TT))
                           else undefined))))))›
  shows
      ‹modal_depth_srbb             φ = 2›
  and ‹branching_conjunction_depth φ = 0›
  and ‹unstable_conjunction_depth  φ = 1›
  and ‹stable_conjunction_depth    φ = 0›
  and ‹immediate_conjunction_depth φ = 0›
  and ‹max_positive_conjunct_depth φ = 1›
  and ‹max_negative_conjunct_depth φ = 0›
  and ‹negation_depth              φ = 0›
  unfolding φ
  by simp+

lemma ‹expressiveness_price (Internal
      (Obs op
        (Internal
          (Conj {left, right}
                (λi. (if i = left
                      then (Pos (Obs a TT))
                      else if i = right
                           then (Pos (Obs b TT))
                           else undefined)))))) = E 2 0 1 0 0 1 0 0›
  by simp

end

context LTS_Tau
begin

lemma ‹expressiveness_price TT = E 0 0 0 0 0 0 0 0›
  by simp

lemma ‹expressiveness_price (ImmConj {} ψs) = E 0 0 0 0 0 0 0 0›
  by (simp add: Sup_enat_def)

lemma ‹expressiveness_price (Internal (Conj {} ψs)) = E 0 0 0 0 0 0 0 0›
  by (simp add: Sup_enat_def)

lemma ‹expressiveness_price (Internal (BranchConj α TT {} ψs)) = E 1 1 1 0 0 1 0 0›
  by (simp add: Sup_enat_def)

lemma expr_obs_phi:
  shows ‹subtract_fn 1 0 0 0 0 0 0 0 (expr_pr_inner (Obs α φ)) = Some (expressiveness_price
φ)›
```

```
  by simp
```

5.11 Characterizing Equivalence by Energy Coordinates

A state p pre-orders another state q with respect to some energy e if and only if p HML pre-orders q with respect to the HML sublanguage $\mathcal{O}$ derived from e.

```
definition expr_preord :: ‹'s ⇒ energy ⇒ 's ⇒ bool› (‹_ ≼ _ _› 60) where
  ‹(p ≼ e q) ≡ preordered (𝒪 e) p q›
```

Conversely, p and q are equivalent with respect to e if and only if they are equivalent with respect to that HML sublanguage $\mathcal{O}$.

```
definition expr_equiv :: ‹'s ⇒ energy ⇒ 's ⇒ bool› (‹_ ∼ _ _› 60) where
  ‹(p ∼ e q) ≡ equivalent (𝒪 e) p q›
```

5.12 Relational Effects of Prices

```
lemma distinction_combination_eta:
  fixes p q
  defines ‹Qα ≡ {q'. q ↠ q' ∧  (∄φ. φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0) ∧ distinguishes φ
p q')}›
  assumes
    ‹p ↦a α p'›
    ‹∀q'∈ Qα.
      ∀q'' q'''. q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ distinguishes (Φ q''') p' q'''›
  shows
    ‹∀q'∈ Qα. hml_srbb_inner.distinguishes (Obs α (Internal (Conj
      {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''} (conjunctify_distinctions Φ p'))))
p q'›
proof -
  have ‹∀q'∈ Qα. ∀q'''∈{q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}.
      hml_srbb_conj.distinguishes ((conjunctify_distinctions Φ p') q''') p' q'''›
  proof clarify
    fix q' q'' q'''
    assume ‹q' ∈ Qα› ‹q' ↦a α q''› ‹q'' ↠ q'''›
    thus ‹hml_srbb_conj.distinguishes (conjunctify_distinctions Φ p' q''') p' q'''›
      using assms(3)  distinction_conjunctification by blast
  qed
  hence ‹∀q'∈ Qα. ∀q''. q' ↦a α q''
    ⟶ distinguishes (Internal (Conj {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
(conjunctify_distinctions Φ p')))  p' q''›
    using silent_reachable.refl unfolding Qα_def by fastforce
  thus ‹∀q'∈ Qα.
     hml_srbb_inner.distinguishes (Obs α (Internal (Conj
       {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''} (conjunctify_distinctions Φ
p')))) p q'›
    using assms(2) by (auto) (metis silent_reachable.refl)+
qed

lemma distinction_conjunctification_two_way_price:
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q ∨ distinguishes (Φ q) q p›
    ‹∀q∈I. Φ q ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
  shows
    ‹∀q∈I.
      (if distinguishes (Φ q) p q then conjunctify_distinctions else conjunctify_distinctions_dual)
Φ p q
      ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
proof
  fix q
  assume ‹q ∈ I›
```

```
  show ‹(if distinguishes (Φ q) p q then conjunctify_distinctions else conjunctify_distinctions_dual)
Φ p q ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
  proof (cases ‹Φ q›)
    case TT
    then show ?thesis
      using assms ‹q ∈ I›
      by fastforce
  next
    case (Internal χ)
    then show ?thesis
      using assms ‹q ∈ I›
      unfolding conjunctify_distinctions_def conjunctify_distinctions_dual_def 𝒪_def 𝒪_conjunct_def
      by fastforce
  next
    case (ImmConj J Ψ)
    hence ‹J = {}›
      using assms ‹q ∈ I› unfolding 𝒪_def
      by (simp, metis iadd_is_0 immediate_conjunction_depth.simps(3) zero_one_enat_neq(1))
    then show ?thesis
      using assms ‹q ∈ I› ImmConj by fastforce
  qed
qed

lemma distinction_combination_eta_two_way:
  fixes p q p' Φ
  defines
    ‹Qα ≡ {q'. q ↠ q' ∧  (∄φ. φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ∧ (distinguishes φ p q'
∨ distinguishes φ q' p))}› and
    ‹Ψα ≡ λq'''. (if distinguishes (Φ q''') p' q''' then conjunctify_distinctions else
conjunctify_distinctions_dual) Φ p' q'''›
  assumes
    ‹p ↦a α p'›
    ‹∀q'∈ Qα.
      ∀q'' q'''. q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ distinguishes (Φ q''') p' q''' ∨ distinguishes
(Φ q''') q''' p'›
  shows
    ‹∀q'∈ Qα. hml_srbb_inner.distinguishes (Obs α (Internal (Conj
      {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
      Ψα))) p q'›
proof -
  have ‹∀q'∈ Qα. ∀q'''∈{q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}.
      hml_srbb_conj.distinguishes (Ψα q''') p' q'''›
  proof clarify
    fix q' q'' q'''
    assume ‹q' ∈ Qα› ‹q' ↦a α q''› ‹q'' ↠ q'''›
    thus ‹hml_srbb_conj.distinguishes
        (Ψα q''') p' q''' ›
      using assms(4) distinction_conjunctification_two_way Ψα_def by blast
  qed
  hence ‹∀q'∈ Qα. ∀q'''∈{q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}.
    hml_srbb_inner.distinguishes (Conj {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
      Ψα)  p' q'''›
    using srbb_dist_conjunct_implies_dist_conjunction
    unfolding lts_semantics.distinguishes_def
    by (metis (no_types, lifting))
  hence ‹∀q'∈ Qα. ∀q'''. (∃q''. q' ↦a α q'' ∧ q'' ↠ q''') ⟶
    hml_srbb_inner.distinguishes (Conj {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
Ψα)  p' q'''›
    by blast
  hence ‹∀q'∈ Qα. ∀q''. q' ↦a α q'' ⟶
    distinguishes (Internal  (Conj {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''} Ψα))
```

```
p' q''›
    by (meson distinguishes_def hml_srbb_inner.distinguishes_def hml_srbb_models.simps(2)
silent_reachable.refl)
  thus ‹∀q'∈ Qα.
     hml_srbb_inner.distinguishes (Obs α (Internal (Conj
        {q'''. ∃q'∈ Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''} Ψα))) p q'›
    using assms(3)
    by auto (metis silent_reachable.refl)+
qed

lemma distinction_conjunctification_price:
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q›
    ‹∀q∈I. Φ q ∈ 𝒪 pr›
    ‹modal_depth pr ≤ pos_conjuncts pr›
  shows
    ‹∀q∈I. ((conjunctify_distinctions Φ p) q) ∈ 𝒪_conjunct pr›
proof
  fix q
  assume ‹q ∈ I›
  show ‹conjunctify_distinctions Φ p q ∈ 𝒪_conjunct pr›
  proof (cases ‹Φ q›)
    case TT
    then show ?thesis
      using assms ‹q ∈ I›
      by fastforce
  next
    case (Internal χ)
    then show ?thesis
      using assms ‹q ∈ I›
      unfolding conjunctify_distinctions_def 𝒪_def 𝒪_conjunct_def
      by fastforce
  next
    case (ImmConj J Ψ)
    hence ‹∃i. i∈J ∧ hml_srbb_conj.distinguishes (Ψ i) p q›
      using ‹q ∈ I› assms(1) by fastforce
    moreover have ‹conjunctify_distinctions Φ p q  = Ψ (SOME i. i∈J ∧ hml_srbb_conj.distinguishes
(Ψ i) p q)›
      unfolding ImmConj conjunctify_distinctions_def by simp
    ultimately have Ψ_i: ‹∃i∈J. hml_srbb_conj.distinguishes (Ψ i) p q ∧ conjunctify_distinctions
Φ p q = Ψ i›
      by (metis (no_types, lifting) some_eq_ex)
    hence ‹conjunctify_distinctions Φ p q ∈ Ψ`J›
      unfolding image_iff by blast
    hence ‹expr_pr_conjunct (conjunctify_distinctions Φ p q) ≤ expressiveness_price (ImmConj
J Ψ)›
      by (smt (verit, best) Ψ_i dual_order.trans expressiveness_price_ImmConj_geq_parts
gets_smaller)
    then show ?thesis
      using assms ‹q ∈ I› ImmConj
      unfolding 𝒪_def 𝒪_conjunct_def
      by auto
  qed
qed

lemma modal_stability_respecting:
  ‹stability_respecting (preordered (𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8)))›
  unfolding stability_respecting_def
proof safe
  fix p q
  assume p_stability:
```

```
    ‹preordered (𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8)) p q›
    ‹stable_state p›
  have ‹¬(∀q'. q ↠ q' ⟶ ¬ preordered (𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8)) p q' ∨ ¬ stable_state
q')›
  proof safe
    assume ‹∀q'. q ↠ q' ⟶ ¬ preordered (𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8)) p q' ∨ ¬ stable_state
q'›
    hence  ‹∀q'. q ↠ q' ⟶ stable_state q' ⟶ (∃φ ∈ 𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8).
distinguishes φ p q')› by auto
    then obtain Φ where Φ_def:
      ‹∀q'∈(silent_reachable_set {q}). stable_state q'
      ⟶ distinguishes (Φ q') p q' ∧ Φ q' ∈ 𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8)›
      using singleton_iff sreachable_set_is_sreachable by metis
    hence distinctions:
      ‹∀q'∈(silent_reachable_set {q} ∩ {q'. stable_state q'}). distinguishes (Φ q') p q'›
      ‹∀q'∈(silent_reachable_set {q} ∩ {q'. stable_state q'}). Φ q' ∈ 𝒪 (E e1 e2 e3 ∞
e5 ∞ e7 e8)› by blast+
    from distinction_conjunctification_price[OF this] have
      ‹∀q'∈(silent_reachable_set {q} ∩ {q'. stable_state q'}). conjunctify_distinctions
Φ p q' ∈ 𝒪_conjunct (E e1 e2 e3 ∞ e5 ∞ e7 e8)›
      by fastforce
    hence conj_price: ‹StableConj (silent_reachable_set {q} ∩ {q'. stable_state q'}) (conjunctify_distinc
Φ p)
       ∈ 𝒪_inner (E e1 e2 e3 ∞ e5 ∞ e7 e8)›
      unfolding 𝒪_inner_def 𝒪_conjunct_def using SUP_le_iff by fastforce
    from Φ_def have
      ‹∀q'∈(silent_reachable_set {q}). stable_state q' ⟶
         hml_srbb_conj.distinguishes (conjunctify_distinctions Φ p q') p q'›
      using singleton_iff distinction_conjunctification by metis
    hence ‹hml_srbb_inner.distinguishes_from
        (StableConj (silent_reachable_set {q} ∩ {q'. stable_state q'}) (conjunctify_distinctions
Φ p))
        p (silent_reachable_set {q})›
      using p_stability(2) by fastforce
    hence
      ‹distinguishes
        (Internal (StableConj (silent_reachable_set {q} ∩ {q'. stable_state q'})
                (conjunctify_distinctions Φ p)))
         p q›
      unfolding silent_reachable_set_def
      using silent_reachable.refl by auto
    moreover have
      ‹Internal (StableConj (silent_reachable_set {q} ∩ {q'. stable_state q'}) (conjunctify_distinctions
Φ p))
         ∈ 𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8)›
      using conj_price unfolding 𝒪_def 𝒪_inner_def by simp
    ultimately show False
      using p_stability(1) preordered_no_distinction by blast
  qed
  thus ‹∃q'. q ↠ q' ∧ preordered (𝒪 (E e1 e2 e3 ∞ e5 ∞ e7 e8)) p q' ∧ stable_state q'›
    by blast
qed

end

end
```

6 Weak Traces

```
theory Weak_Traces
```

```
  imports Main HML_SRBB Expressiveness_Price
begin
```

The inductive `is_trace_formula` represents the modal-logical characterization of weak traces HML_{WT}. In particular:

- $\top \in \mathrm{HML}_{WT}$ encoded by `is_trace_formula TT`, `is_trace_formula ImmConj I ψs` if `I = {}` and `is_trace_formula Conj I ψs` if `I = {}`..
- $\langle\varepsilon\rangle\chi \in \mathrm{HML}_{WT}$ if $\varphi \in \mathrm{HML}_{WT}$ encoded by `is_trace_formula Internal χ` if `is_trace_formula χ`.
- $(\alpha)\varphi \in \mathrm{HML}_{WT}$ if $\varphi \in \mathrm{HML}_{WT}$ encoded by `is_trace_formula Obs α φ` if `is_trace_formula φ`.
- $\bigwedge$`{`$(\alpha)\varphi$`}` $\cup$ `Ψ` $\in \mathrm{HML}_{WT}$ if $\varphi \in \mathrm{HML}_{WT}$ and `Ψ = {}` encoded by `is_trace_formula BranchConj α φ I ψs` if `is_trace_formula φ` and `I = {}`.

```
inductive
      is_trace_formula :: ‹('act, 'i) hml_srbb ⇒ bool›
  and is_trace_formula_inner :: ‹('act, 'i) hml_srbb_inner ⇒ bool› where
  ‹is_trace_formula TT› |
  ‹is_trace_formula (Internal χ)› if ‹is_trace_formula_inner χ› |
  ‹is_trace_formula (ImmConj I ψs)› if ‹I = {}› |

  ‹is_trace_formula_inner (Obs α φ)› if ‹is_trace_formula φ› |
  ‹is_trace_formula_inner (Conj I ψs)› if ‹I = {}›
```

We define a function that translates a (weak) trace `tr` to a formula φ such that a state `p` models φ, `p` $\models \varphi$ if and only if `tr` is a (weak) trace of `p`.

```
fun wtrace_to_srbb :: ‹'act list ⇒ ('act, 'i) hml_srbb›
  and wtrace_to_inner :: ‹'act list ⇒ ('act, 'i) hml_srbb_inner›
  and wtrace_to_conjunct :: ‹'act list ⇒ ('act, 'i) hml_srbb_conjunct› where
  ‹wtrace_to_srbb [] = TT› |
  ‹wtrace_to_srbb tr = (Internal (wtrace_to_inner tr))› |

  ‹wtrace_to_inner [] = (Conj {} (λ_. undefined))› | — Should never happen
  ‹wtrace_to_inner (α # tr) = (Obs α (wtrace_to_srbb tr))› |

  ‹wtrace_to_conjunct tr = Pos (wtrace_to_inner tr)› — Should never happen
```

`wtrace_to_srbb trace` is in our modal-logical characterization of weak traces.

```
lemma trace_to_srbb_is_trace_formula:
  ‹is_trace_formula (wtrace_to_srbb trace)›
  by (induct trace,
      auto simp add: is_trace_formula.simps is_trace_formula_is_trace_formula_inner.intros(1,4))
```

The following three lemmas show that the modal-logical characterization of HML_{WT} corresponds to the sublanguage of $\mathrm{HML}_{\mathrm{SRBB}}$, obtain by the energy coordinates `(∞, 0, 0, 0, 0, 0, 0, 0)`.

```
lemma trace_formula_to_expressiveness:
  fixes φ :: ‹('act, 'i) hml_srbb›
  fixes χ :: ‹('act, 'i) hml_srbb_inner›
  shows  ‹(is_trace_formula φ        ⟶ (φ ∈ 𝒪        (E ∞ 0 0 0 0 0 0 0)))
        ∧ (is_trace_formula_inner χ ⟶ (χ ∈ 𝒪_inner (E ∞ 0 0 0 0 0 0 0)))›
  by (rule is_trace_formula_is_trace_formula_inner.induct) (simp add: Sup_enat_def 𝒪_def
𝒪_inner_def)+

lemma expressiveness_to_trace_formula:
  fixes φ :: ‹('act, 'i) hml_srbb›
  fixes χ :: ‹('act, 'i) hml_srbb_inner›
```

```
  shows ‹(φ ∈ 𝒪 (E ∞ 0 0 0 0 0 0 0) ⟶ is_trace_formula φ)
       ∧ (χ ∈ 𝒪_inner (E ∞ 0 0 0 0 0 0 0) ⟶ is_trace_formula_inner χ)
       ∧ True›
proof (induct rule: hml_srbb_hml_srbb_inner_hml_srbb_conjunct.induct)
  case TT
  then show ?case
    using is_trace_formula_is_trace_formula_inner.intros(1) by blast
next
  case (Internal x)
  then show ?case
    by (simp add: 𝒪_inner_def 𝒪_def is_trace_formula_is_trace_formula_inner.intros(2))
next
  case (ImmConj x1 x2)
  then show ?case
    using 𝒪_def is_trace_formula_is_trace_formula_inner.intros(3)
    by(auto simp add: 𝒪_def)
next
  case (Obs x1 x2)
  then show ?case by (simp add: 𝒪_def 𝒪_inner_def is_trace_formula_is_trace_formula_inner.intros(4))
next
  case (Conj I ψs)
  show ?case
  proof (rule impI)
    assume ‹Conj I ψs ∈ 𝒪_inner (E ∞ 0 0 0 0 0 0 0)›
    hence ‹I = {}›
      unfolding 𝒪_inner_def
      by (metis bot.extremum_uniqueI bot_enat_def energy.sel(3) expr_pr_inner.simps inst_conj_depth_inner
le_iff_add leq_components mem_Collect_eq not_one_le_zero)
    then show ‹is_trace_formula_inner (Conj I ψs)›
      by (simp add: is_trace_formula_is_trace_formula_inner.intros(5))
  qed
next
  case (StableConj I ψs)
  show ?case
  proof (rule impI)
    assume ‹StableConj I ψs ∈ 𝒪_inner (E ∞ 0 0 0 0 0 0 0)›
    have ‹StableConj I ψs ∉ 𝒪_inner (E ∞ 0 0 0 0 0 0 0)›
      by (simp add: 𝒪_inner_def)
    with ‹StableConj I ψs ∈ 𝒪_inner (E ∞ 0 0 0 0 0 0 0)›
    show ‹is_trace_formula_inner (StableConj I ψs)› by contradiction
  qed
next
  case (BranchConj α φ I ψs)
  have ‹expr_pr_inner (BranchConj α φ I ψs) ≥ E 0 1 1 0 0 0 0 0›
    by simp
  hence ‹BranchConj α φ I ψs ∉ 𝒪_inner (E ∞ 0 0 0 0 0 0 0)›
    unfolding 𝒪_inner_def by simp
  thus ?case by blast
next
  case (Pos x)
  then show ?case by auto
next
  case (Neg x)
  then show ?case by auto
qed

lemma modal_depth_only_is_trace_form:
  ‹(is_trace_formula φ) = (φ ∈ 𝒪 (E ∞ 0 0 0 0 0 0 0))›
  using expressiveness_to_trace_formula trace_formula_to_expressiveness by blast

context LTS_Tau
```

```
begin
```

If a formula φ is in HML_{WT} and a state `p` models φ, then there exists a weak trace `tr` of `p` such that `wtrace_to_srbb tr` is equivalent to φ.

```
lemma trace_formula_implies_trace:
  fixes ψ ::‹('a, 's) hml_srbb_conjunct›
  shows
       trace_case: ‹is_trace_formula φ ⟹ p ⊨SRBB φ ⟹ (∃tr ∈ weak_traces p. wtrace_to_srbb
tr ⇚srbb⇛ φ)›
    and conj_case: ‹is_trace_formula_inner χ ⟹ hml_srbb_inner_models q χ ⟹ (∃tr ∈ weak_traces
q. wtrace_to_inner tr ⇚χ⇛ χ)›
    and           True
proof (induction φ and χ and ψ arbitrary: p and q)
  case TT
  then have ‹[] ∈ weak_traces p›
    using weak_step_sequence.intros(1) silent_reachable.intros(1) by fastforce
  moreover have ‹wtrace_to_srbb [] ⇚srbb⇛ TT›
    unfolding wtrace_to_srbb.simps
    by (simp add: equivp_reflp)
  ultimately show ?case by auto
next
  case (Internal χ)

  from ‹is_trace_formula (Internal χ)›
  have ‹is_trace_formula_inner χ›
    using is_trace_formula.cases by auto

  from ‹p ⊨SRBB Internal χ›
  have ‹∃p'. p ↠ p' ∧ hml_srbb_inner_models p' χ›
    unfolding hml_srbb_models.simps.
  then obtain p' where ‹p ↠ p'› and ‹hml_srbb_inner_models p' χ› by auto
  hence ‹hml_srbb_inner_models p' χ› by auto
  with ‹is_trace_formula_inner χ›
  have ‹∃tr∈weak_traces p'. wtrace_to_inner tr ⇚χ⇛ χ›
    using Internal.IH by blast
  then obtain tr where tr_spec:
    ‹tr ∈ weak_traces p'› ‹wtrace_to_inner tr ⇚χ⇛ χ› by auto
  with ‹p ↠ p'› have ‹tr ∈ weak_traces p›
    using silent_prepend_weak_traces by auto

  moreover
  have ‹wtrace_to_srbb tr ⇚srbb⇛ Internal χ›
  proof (cases tr)
    case Nil
    thus ?thesis
      using srbb_TT_is_χTT tr_spec by auto
  next
    case (Cons a tr)
    thus ?thesis
      using tr_spec internal_srbb_cong by auto
  qed

  ultimately show ?case by auto
next
  case (ImmConj I ψs)

  from ‹is_trace_formula (ImmConj I ψs)›
  have ‹I = {}›
    by (simp add: is_trace_formula.simps)

  have ‹[] ∈ weak_traces p›
```

```
    using silent_reachable.intros(1) weak_step_sequence.intros(1) by auto

  from srbb_TT_is_empty_conj
   and ‹I = {}›
  have ‹wtrace_to_srbb [] ⇐srbb⇒ ImmConj I ψs›
    unfolding wtrace_to_srbb.simps by auto

  from ‹[] ∈ weak_traces p›
   and ‹wtrace_to_srbb [] ⇐srbb⇒ ImmConj I ψs›
  show ‹∃tr∈weak_traces p. wtrace_to_srbb tr ⇐srbb⇒ ImmConj I ψs› by auto
next
  case (Obs α φ)
  assume IH: ‹⋀p1. is_trace_formula φ ⟹ p1 ⊨SRBB φ ⟹ ∃tr∈weak_traces p1. wtrace_to_srbb
tr ⇐srbb⇒ φ›
     and ‹is_trace_formula_inner (Obs α φ)›
     and ‹hml_srbb_inner_models q (Obs α φ)›
  then show ‹∃tr ∈ weak_traces q. wtrace_to_inner tr ⇐χ⇒ Obs α φ›
  proof (cases ‹α = τ›)
    case True

    with ‹hml_srbb_inner_models q (Obs α φ)› have ‹q ⊨SRBB φ›
      using Obs.prems(1) silent_reachable.step empty_conj_trivial(1)
      by (metis (no_types, lifting) hml_srbb_inner.distinct(1) hml_srbb_inner.inject(1)
          hml_srbb_inner_models.simps(1) hml_srbb_models.simps(1,2) is_trace_formula.cases
          is_trace_formula_inner.cases)

    moreover have ‹is_trace_formula φ›
      using ‹is_trace_formula_inner (Obs α φ)› is_trace_formula_inner.cases by auto

    ultimately show ‹∃tr ∈ weak_traces q. wtrace_to_inner tr ⇐χ⇒ Obs α φ›
      using Obs.IH
      by (metis ‹α = τ› obs_srbb_cong prepend_τ_weak_trace wtrace_to_inner.simps(2))
  next
    case False

    from ‹is_trace_formula_inner (Obs α φ)›
    have ‹is_trace_formula φ›
      by (simp add: is_trace_formula_inner.simps)

    from ‹hml_srbb_inner_models q (Obs α φ)› and ‹α ≠ τ›
    have ‹∃q'. q ↦ α q' ∧ q' ⊨SRBB φ› by simp
    then obtain q' where ‹q ↦ α q'› and ‹q' ⊨SRBB φ› by auto

    from ‹is_trace_formula φ›
     and ‹q' ⊨SRBB φ›
     and IH
    have ‹∃tr' ∈ weak_traces q'. wtrace_to_srbb tr' ⇐srbb⇒ φ› by auto
    then obtain tr' where ‹tr' ∈ weak_traces q'› and ‹wtrace_to_srbb tr' ⇐srbb⇒ φ› by
auto

    from ‹q ↦ α q'›
     and ‹tr' ∈ weak_traces q'›
    have ‹(α # tr') ∈ weak_traces q›
      using step_prepend_weak_traces by auto

    from ‹wtrace_to_srbb tr' ⇐srbb⇒ φ›
    have ‹Obs α (wtrace_to_srbb tr') ⇐χ⇒ Obs α φ›
      using obs_srbb_cong by auto
    then have ‹wtrace_to_inner (α # tr') ⇐χ⇒ Obs α φ›
      unfolding wtrace_to_inner.simps.
```

```
    with ‹(α # tr') ∈ weak_traces q›
    show ‹∃tr ∈ weak_traces q. wtrace_to_inner tr ⇐χ⇒ Obs α φ› by blast
  qed
next
  case (Conj I ψs)
  assume ‹is_trace_formula_inner (Conj I ψs)›
     and ‹hml_srbb_inner_models q (Conj I ψs)›

  from ‹is_trace_formula_inner (Conj I ψs)›
  have ‹I = {}›
    by (simp add: is_trace_formula_inner.simps)

  have ‹[] ∈ weak_traces q› by (rule empty_trace_allways_weak_trace)

  have ‹(Conj {} (λ_. undefined)) ⇐χ⇒ (Conj {} ψs)›
    using srbb_obs_τ_is_χTT by simp
  then have ‹(Conj {} (λ_. undefined)) ⇐χ⇒ (Conj I ψs)›
    using ‹I = {}› by auto
  then have ‹wtrace_to_inner [] ⇐χ⇒ Conj I ψs›
    unfolding wtrace_to_inner.simps.

  from ‹[] ∈ weak_traces q›
   and ‹wtrace_to_inner [] ⇐χ⇒ Conj I ψs›
  show ?case by auto
next
  case (StableConj I ψs)
  have ‹¬is_trace_formula_inner (StableConj I ψs)›
    by (simp add: is_trace_formula_inner.simps)
  with ‹is_trace_formula_inner (StableConj I ψs)›
  show ?case by contradiction
next
  case (BranchConj α φ I ψs)
  assume IH: ‹⋀p1. is_trace_formula φ ⟹ p1 ⊨SRBB φ ⟹ ∃tr∈weak_traces p1. wtrace_to_srbb
tr ⇐srbb⇒ φ›
  from ‹is_trace_formula_inner (BranchConj α φ I ψs)›
  have ‹is_trace_formula φ ∧ I = {}›
    by (simp add: is_trace_formula_inner.simps)
  hence ‹is_trace_formula φ› and ‹I = {}› by auto
  from ‹hml_srbb_inner_models q (BranchConj α φ I ψs)›
   and ‹I = {}›
  have ‹hml_srbb_inner_models q (Obs α φ)›
    using srbb_obs_is_empty_branch_conj
    by auto

  have ‹∃tr ∈ weak_traces q. wtrace_to_inner tr ⇐χ⇒ Obs α φ›
  proof (cases ‹α = τ›)
    assume ‹α = τ›

    from ‹hml_srbb_inner_models q (Obs α φ)›
    show ‹∃tr ∈ weak_traces q. wtrace_to_inner tr ⇐χ⇒ Obs α φ›
      using BranchConj.prems(1) is_trace_formula_inner.simps by fastforce
  next
    assume ‹α ≠ τ›

    from ‹hml_srbb_inner_models q (Obs α φ)›
     and ‹α ≠ τ›
    have ‹∃q'. q ↦ α q' ∧ q' ⊨SRBB φ› by auto
    then obtain q' where ‹q ↦ α q'› and ‹q' ⊨SRBB φ› by auto

    from ‹is_trace_formula φ›
     and ‹q' ⊨SRBB φ›
```

```
      and IH
    have ‹∃tr' ∈ weak_traces q'. wtrace_to_srbb tr' ⇐srbb⇒ φ› by auto
    then obtain tr' where ‹tr' ∈ weak_traces q'› and ‹wtrace_to_srbb tr' ⇐srbb⇒ φ› by
auto

    from ‹q ↦ α q'›
      and ‹tr' ∈ weak_traces q'›
    have ‹(α # tr') ∈ weak_traces q›
      using step_prepend_weak_traces by auto

    from ‹wtrace_to_srbb tr' ⇐srbb⇒ φ›
    have ‹Obs α (wtrace_to_srbb tr') ⇐χ⇒ Obs α φ›
      using obs_srbb_cong by auto
    then have ‹wtrace_to_inner (α # tr') ⇐χ⇒ Obs α φ›
      unfolding wtrace_to_inner.simps.

    with ‹(α # tr') ∈ weak_traces q›
    show ‹∃tr ∈ weak_traces q. wtrace_to_inner tr ⇐χ⇒ Obs α φ› by blast
  qed
  then obtain tr where ‹tr ∈ weak_traces q› and ‹wtrace_to_inner tr ⇐χ⇒ Obs α φ› by auto

  from ‹wtrace_to_inner tr ⇐χ⇒ Obs α φ›
    and ‹I = {}›
  have ‹wtrace_to_inner tr ⇐χ⇒ (BranchConj α φ I ψs)›
    using srbb_obs_is_empty_branch_conj by simp
  with ‹tr ∈ weak_traces q›
  show ?case by blast
next
  case (Pos χ)
  then show ?case by auto
next
  case (Neg χ)
  then show ?case by auto
qed
```

t is a weak trace of a state p if and only if p models the formula obtained from wtrace_to_srbb t.

```
lemma trace_equals_trace_to_formula:
  ‹t ∈ weak_traces p = (p ⊨SRBB (wtrace_to_srbb t))›
proof
  assume ‹t ∈ weak_traces p›
  show ‹p ⊨SRBB (wtrace_to_srbb t)›
    using ‹t ∈ weak_traces p›
  proof(induction t arbitrary: p)
    case Nil
    then show ?case
      by simp
  next
    case (Cons a tail)
    from Cons obtain p'' p' where ‹p ↠↦↠ a p''› ‹p'' ↠↦↠$ tail p'› using weak_step_sequence.simps
      by (smt (verit, best) list.discI list.inject mem_Collect_eq)
    with Cons(1) have IS: ‹p'' ⊨SRBB wtrace_to_srbb tail›
      by blast
    from Cons have goal_eq: ‹wtrace_to_srbb (a # tail) = (Internal (Obs a (wtrace_to_srbb
tail)))›
      by simp
    show ?case
      by (smt (verit) Cons.IH IS LTS_Tau.hml_srbb_inner_models.simps(1)
          LTS_Tau.silent_reachable_trans ‹p ↠↦↠ a p''› empty_trace_allways_weak_trace
goal_eq
          hml_srbb_models.simps(2) weak_step_def wtrace_to_srbb.elims)
```

```
  qed
next
  assume ‹p ⊨SRBB wtrace_to_srbb t›
  then show ‹t ∈ weak_traces p›
  proof(induction t arbitrary: p)
    case Nil
    then show ?case
      using weak_step_sequence.intros(1) silent_reachable.intros(1) by auto
  next
    case (Cons a tail)
    hence ‹p ⊨SRBB (Internal (Obs a (wtrace_to_srbb tail)))›
      by simp
    show ?case
      using Cons.IH ‹p ⊨SRBB hml_srbb.Internal (hml_srbb_inner.Obs a (wtrace_to_srbb tail))›
prepend_τ_weak_trace silent_prepend_weak_traces step_prepend_weak_traces by fastforce
  qed
qed
```

If a state p weakly trace-pre-orders another state q, φ is in our modal-logical characterization HML_{WT}, and p models φ then q models φ.

```
lemma aux:
  fixes φ :: ‹('a, 's) hml_srbb›
  fixes χ :: ‹('a, 's) hml_srbb_inner›
  fixes ψ :: ‹('a, 's) hml_srbb_conjunct›
  shows ‹p ≲WT q ⟹ is_trace_formula φ ⟹ p ⊨SRBB φ ⟹ q ⊨SRBB φ›
proof -
  assume φ_trace: ‹is_trace_formula φ› and p_sat_srbb: ‹p ⊨SRBB φ› and assms: ‹p ≲WT
q›
  show ‹q ⊨SRBB φ›
  proof-
    from assms have p_trace_implies_q_trace: ‹∀tr p'. (p ↠↦↠$ tr p') ⟶ (∃q'. q ↠↦↠$
tr q')›
      unfolding weakly_trace_preordered_def by auto
    from p_sat_srbb trace_formula_implies_trace obtain tr p' where
      ‹(p ↠↦↠$ tr p')› ‹wtrace_to_srbb tr ⇐srbb⇒ φ›
      using φ_trace by blast
    with p_trace_implies_q_trace obtain q' where ‹q ↠↦↠$ tr q'›
      by blast
    with trace_equals_trace_to_formula show ?thesis
      using ‹wtrace_to_srbb tr ⇐srbb⇒ φ› by auto
  qed
qed
```

These are the main lemmas of this theory. They establish that the colloquial, relational notion of of weak trace pre-order/equivalence has the same distinctive power as the one derived from the coordinate (∞, 0, 0, 0, 0, 0, 0, 0).

A state p weakly trace-pre-orders a state q iff and only if it also pre-orders q with respect to the coordinate (∞, 0, 0, 0, 0, 0, 0, 0).

```
lemma expr_preorder_characterizes_relational_preorder_traces:
  ‹(p ≲WT q) = (p ≼ (E ∞ 0 0 0 0 0 0 0) q)›
  unfolding expr_preord_def preordered_def
proof
  assume ‹p ≲WT q›
  thus ‹∀φ∈𝒪 (E ∞ 0 0 0 0 0 0 0). p ⊨SRBB φ ⟶ q ⊨SRBB φ›
    using aux expressiveness_to_trace_formula weakly_trace_preordered_def
    by blast+
next
  assume φ_eneg: ‹∀φ∈𝒪 (E ∞ 0 0 0 0 0 0 0). p ⊨SRBB φ ⟶ q ⊨SRBB φ›
  thus ‹p ≲WT q›
```

```
    unfolding weakly_trace_preordered_def
    using trace_equals_trace_to_formula trace_formula_to_expressiveness trace_to_srbb_is_trace_formula
    by fastforce
qed
```

Two states p and q are weakly trace equivalent if and only if they they are equivalent with respect to the coordinate (∞, 0, 0, 0, 0, 0, 0, 0).

```
lemma ‹(p ≃WT q) = (p ∼ (E ∞ 0 0 0 0 0 0 0) q)›
  using expr_preorder_characterizes_relational_preorder_traces
  unfolding weakly_trace_equivalent_def expr_equiv_def 𝒪_def expr_preord_def
  by simp

end

end
```

7 η-Bisimilarity

```
theory Eta_Bisimilarity
  imports Expressiveness_Price
begin
```

7.1 Definition and Properties of η-(Bi-)Similarity

```
context LTS_Tau
begin
```

— Following Def 2.1 in Divide and congruence

```
definition eta_simulation :: ‹('s ⇒ 's ⇒ bool) ⇒ bool› where
  ‹eta_simulation R ≡ ∀p α p' q. R p q ⟶ p ↦ α p' ⟶
    ((α = τ ∧ R p' q) ∨ (∃q' q'' q'''. q ↠ q' ∧ q' ↦ α q'' ∧ q'' ↠ q'''  ∧ R p q' ∧
R p' q''')))›

definition eta_bisimulated :: ‹'s ⇒ 's ⇒ bool› (infix ‹~η› 40) where
  ‹p ~η q ≡ ∃R. eta_simulation R ∧ symp R ∧ R p q›

lemma eta_bisim_sim:
  shows ‹eta_simulation (~η)›
  unfolding eta_bisimulated_def eta_simulation_def by blast

lemma eta_bisim_sym:
  assumes ‹p ~η q›
  shows ‹q ~η p›
  using assms unfolding eta_bisimulated_def
  by (meson sympD)

lemma silence_retains_eta_sim:
assumes
  ‹eta_simulation R›
  ‹R p q›
  ‹p ↠ p'›
shows ‹∃q'. R p' q' ∧ q ↠ q'›
  using assms(3,2)
proof (induct arbitrary: q)
  case (refl p)
  then show ?case
    using silent_reachable.refl by blast
next
  case (step p p' p'')
  then obtain q' where ‹R p' q'› ‹q ↠ q'›
```

```
    using ‹eta_simulation R› silent_reachable.refl silent_reachable_append_τ silent_reachable_trans
    unfolding eta_simulation_def by blast
  then obtain q'' where ‹R p'' q''› ‹q' ↠ q''› using step by blast
  then show ?case
    using ‹q ↠ q'› silent_reachable_trans by blast
qed

lemma eta_bisimulated_silently_retained:
  assumes
    ‹p ~η q›
    ‹p ↠ p'›
  shows
    ‹∃q'. q ↠ q' ∧ p' ~η q'› using assms(2,1)
  using silence_retains_eta_sim unfolding eta_bisimulated_def by blast
```

7.2 Logical Characterization of η-Bisimilarity through Expressiveness Price

```
lemma logic_eta_bisim_invariant:
  assumes
    ‹p0 ~η q0›
    ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
    ‹p0 ⊨SRBB φ›
  shows ‹q0 ⊨SRBB φ›
proof -
  have ‹⋀φ χ ψ.
    (∀p q. p ~η q ⟶ φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ⟶ p ⊨SRBB φ ⟶ q ⊨SRBB φ) ∧
    (∀p q. p ~η q ⟶ χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ⟶ hml_srbb_inner_models p
χ ⟶ (∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ)) ∧
    (∀p q. p ~η q ⟶ ψ ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ⟶ hml_srbb_conjunct_models
p ψ ⟶ hml_srbb_conjunct_models q ψ)›
  proof-
    fix φ χ ψ
    show
      ‹(∀p q. p ~η q ⟶ φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ⟶ p ⊨SRBB φ ⟶ q ⊨SRBB φ)
∧
      (∀p q. p ~η q ⟶ χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ⟶ hml_srbb_inner_models
p χ ⟶ (∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ)) ∧
      (∀p q. p ~η q ⟶ ψ ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ⟶ hml_srbb_conjunct_models
p ψ ⟶ hml_srbb_conjunct_models q ψ)›
    proof (induct rule: hml_srbb_hml_srbb_inner_hml_srbb_conjunct.induct)
      case TT
      then show ?case by simp
    next
      case (Internal χ)
      show ?case
      proof safe
        fix p q
        assume case_assms:
          ‹p ~η q› ‹p ⊨SRBB hml_srbb.Internal χ› ‹Internal χ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞
∞)›
        then obtain p' where p'_spec: ‹p ↠ p'› ‹hml_srbb_inner_models p' χ› by auto
        have ‹χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
          using case_assms(3) unfolding 𝒪_inner_def 𝒪_def by auto
        hence ‹∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ›
          using Internal case_assms(1) p'_spec eta_bisimulated_silently_retained
          by (meson  silent_reachable_trans)
        thus ‹q ⊨SRBB hml_srbb.Internal χ› by auto
      qed
    next
      case (ImmConj I Ψ)
```

```
      then show ?case unfolding 𝒪_inner_def 𝒪_def by auto
    next
      case (Obs α φ)
      then show ?case
      proof (safe)
        fix p q
        assume case_assms:
          ‹p ~η q›
          ‹Obs α φ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
          ‹hml_srbb_inner_models p (hml_srbb_inner.Obs α φ)›
        hence ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)› unfolding 𝒪_inner_def 𝒪_def by auto
        hence no_imm_conj: ‹∄I Ψ. φ = ImmConj I Ψ ∧ I ≠ {}› unfolding 𝒪_def by force
        have back_step: ‹∀p0 p1. p1 ⊨SRBB φ ⟶ p0 ↠ p1 ⟶ p0 ⊨SRBB φ›
        proof (cases φ)
          case TT
          then show ?thesis by auto
        next
          case (Internal _)
          then show ?thesis
            using silent_reachable_trans by auto
        next
          case (ImmConj _ _)
          then show ?thesis using no_imm_conj by auto
        qed
        from case_assms obtain p’ where ‹p ↦a α p’› ‹p’ ⊨SRBB φ› by auto
        then obtain q’ q’’ q’’’ where ‹q ↠ q’› ‹q’ ↦a α q’’› ‹q’’ ↠ q’’’›  ‹p’ ~η
q’’’›
          using ‹p ~η q› eta_bisim_sim unfolding eta_simulation_def
          using silent_reachable.refl by blast
        hence ‹q’’’ ⊨SRBB φ› using ‹p’ ⊨SRBB φ› Obs ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
by blast
        hence ‹hml_srbb_inner_models q’ (hml_srbb_inner.Obs α φ)›
          using ‹q’ ↦a α q’’› ‹q’’ ↠ q’’’› back_step by auto
        thus ‹∃q’. q ↠ q’ ∧ hml_srbb_inner_models q’ (hml_srbb_inner.Obs α φ)›
          using ‹q ↠ q’› by blast
      qed
    next
      case (Conj I Ψ)
      show ?case
      proof safe
        fix p q
        assume case_assms:
          ‹p ~η q›
          ‹Conj I Ψ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
          ‹hml_srbb_inner_models p (Conj I Ψ)›
        hence conj_price: ‹∀i∈I. Ψ i ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
          unfolding 𝒪_conjunct_def 𝒪_inner_def
          by (simp, metis SUP_bot_conv(1) le_zero_eq sup_bot_left sup_ge1)
        from case_assms have ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)› by auto
        hence ‹∀i∈I. hml_srbb_conjunct_models q (Ψ i)›
          using Conj ‹p ~η q› conj_price by blast
        hence ‹hml_srbb_inner_models q (hml_srbb_inner.Conj I Ψ)› by simp
        thus ‹∃q’. q ↠ q’ ∧ hml_srbb_inner_models q’ (hml_srbb_inner.Conj I Ψ)›
          using silent_reachable.refl by blast
      qed
    next
      case (StableConj I Ψ)
      thus ?case unfolding 𝒪_inner_def 𝒪_def by auto
    next
      case (BranchConj α φ I Ψ)
      show ?case
```

```
    proof safe
      fix p q
      assume case_assms:
        ‹p ~η q›
        ‹BranchConj α φ I Ψ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
        ‹hml_srbb_inner_models p (BranchConj α φ I Ψ)›
      hence ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)› unfolding 𝒪_inner_def 𝒪_def
        by (simp, metis le_zero_eq sup_ge1)
      hence no_imm_conj: ‹∄I Ψ. φ = ImmConj I Ψ ∧ I ≠ {}› unfolding 𝒪_def by force
      have back_step: ‹∀p0 p1. p1 ⊨SRBB φ ⟶ p0 ↠ p1 ⟶ p0 ⊨SRBB φ›
      proof (cases φ)
        case TT
        then show ?thesis by auto
      next
        case (Internal _)
        then show ?thesis
          using silent_reachable_trans by auto
      next
        case (ImmConj _ _)
        then show ?thesis using no_imm_conj by auto
      qed
      from case_assms have conj_price: ‹∀i∈I. Ψ i ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞
∞)›
        unfolding 𝒪_conjunct_def 𝒪_inner_def
        by (simp, metis SUP_bot_conv(1) le_zero_eq sup_bot_left sup_ge1)
      from case_assms have ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)›
            ‹hml_srbb_inner_models p (Obs α φ)›
        using branching_conj_parts branching_conj_obs by blast+
      then obtain p’ where ‹p ↦a α p’› ‹p’ ⊨SRBB φ› by auto
      then obtain q’ q’’ q’’’ where q’_q’’_spec:
        ‹q ↠ q’› ‹q’ ↦a α q’’› ‹q’’ ↠ q’’’›
        ‹p ~η q’› ‹p’ ~η q’’’›
        using eta_bisim_sim ‹p ~η q› silent_reachable.refl
        unfolding eta_simulation_def by blast
      hence ‹q’’’ ⊨SRBB φ›
        using BranchConj.hyps ‹p’ ⊨SRBB φ› ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)› by auto
      hence ‹q’’ ⊨SRBB φ› using back_step q’_q’’_spec by blast
      hence ‹hml_srbb_inner_models q’ (Obs α φ)› using q’_q’’_spec by auto
      moreover have ‹∀i∈I. hml_srbb_conjunct_models q’ (Ψ i)›
        using BranchConj.hyps ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)› q’_q’’_spec conj_price
        by blast
      ultimately show ‹∃q’. q ↠ q’ ∧ hml_srbb_inner_models q’ (BranchConj α φ I Ψ)›
        using ‹q ↠ q’› by auto
    qed
  next
    case (Pos χ)
    show ?case
    proof safe
      fix p q
      assume case_assms:
        ‹p ~η q›
        ‹Pos χ ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
        ‹hml_srbb_conjunct_models p (Pos χ)›
      hence ‹χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
        unfolding 𝒪_inner_def 𝒪_conjunct_def by simp
      from case_assms obtain p’ where ‹p ↠ p’› ‹hml_srbb_inner_models p’ χ› by auto
      then obtain q’ where ‹q ↠ q’› ‹hml_srbb_inner_models q’ χ›
        using Pos ‹p ~η q› ‹χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
        by (meson eta_bisimulated_silently_retained silent_reachable_trans)
      thus ‹hml_srbb_conjunct_models q (Pos χ)› by auto
    qed
```

```
    next
      case (Neg χ)
      show ?case
      proof safe
        fix p q
        assume case_assms:
          ‹p ~η q›
          ‹Neg χ ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
          ‹hml_srbb_conjunct_models p (Neg χ)›
        hence ‹χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
          unfolding 𝒪_inner_def 𝒪_conjunct_def by simp
        from case_assms have ‹∀p'. p ↠ p' ⟶ ¬hml_srbb_inner_models p' χ› by simp
        moreover have
          ‹(∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ) ⟶ (∃p'. p ↠ p' ∧ hml_srbb_inner_models
p' χ)›
          using Neg eta_bisim_sym[OF ‹p ~η q›] eta_bisimulated_silently_retained
            silent_reachable_trans ‹χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)› by blast
        ultimately have ‹∀q'. q ↠ q' ⟶ ¬hml_srbb_inner_models q' χ› by blast
        thus ‹hml_srbb_conjunct_models q (Neg χ)› by simp
      qed
    qed
  qed
  thus ?thesis using assms by blast
qed

lemma modal_eta_sim_eq: ‹eta_simulation (equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)))›
proof -
  have ‹∄p α p' q. (equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞))) p q ∧ p ↦ α p' ∧
      (α ≠ τ ∨ ¬(equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞))) p' q) ∧
      (∀q' q'' q'''. q ↠ q' ⟶ q' ↦ α q'' ⟶ q'' ↠ q''' ⟶
    ¬ equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)) p q' ∨ ¬ equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞
∞ ∞)) p' q''')›
  proof clarify
    fix p α p' q
    define Qα where ‹Qα ≡ {q'. q ↠ q' ∧ (∄φ. φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ∧ (distinguishes
φ p q' ∨ distinguishes φ q' p))}›
    assume contradiction:
      ‹equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)) p q› ‹p ↦ α p'›
      ‹∀q' q'' q'''. q ↠ q' ⟶ q' ↦ α q'' ⟶ q'' ↠ q''' ⟶
      ¬ equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)) p q' ∨ ¬ equivalent (𝒪 (E ∞ ∞ ∞ 0 0
∞ ∞ ∞)) p' q'''›
      ‹α ≠ τ ∨ ¬ equivalent (𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)) p' q›
    hence distinctions: ‹∀q'. q ↠ q' ⟶
      (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes φ q' p) ∨
      (∀q'' q'''. q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes
φ p' q''' ∨ distinguishes φ q''' p'))›
      unfolding equivalent_no_distinction
      by (metis silent_reachable.cases silent_reachable.refl)
    hence ‹∀q'' q''' . ∀q'∈Qα.
        q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes
φ p' q''' ∨ distinguishes φ q''' p')›
      unfolding Qα_def using silent_reachable.refl by fastforce
    hence ‹∀q'' q'''. q'' ↠ q''' ⟶ (∃q'. q ↠ q' ∧ (∄φ. φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞
∞) ∧ (distinguishes φ p q' ∨ distinguishes φ q' p)) ∧ q' ↦a α q'')
        ⟶ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p' q''' ∨ distinguishes φ
q''' p')›
      unfolding Qα_def by blast
    hence ‹∀q'''. (∃q' q''. q ↠ q' ∧ (∄φ. φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ∧ (distinguishes
φ p q' ∨ distinguishes φ q' p)) ∧ q' ↦a α q'' ∧  q'' ↠ q''')
        ⟶ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p' q''' ∨ distinguishes φ
q''' p')›
```

```
      by blast
    then obtain Φα where Φα_def:
      ‹∀q'''. (∃q' q''. q ↠ q' ∧ (∄φ. φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ∧ (distinguishes
φ p q' ∨ distinguishes φ q' p)) ∧ q' ↦a α q'' ∧ q'' ↠ q''')
        ⟶ (Φα q''') ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) ∧ (distinguishes (Φα q''') p' q'''
∨ distinguishes (Φα q''') q''' p')› by metis
    hence distinctions_α: ‹∀q'∈Qα. ∀q'' q'''.
        q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ distinguishes (Φα q''') p' q''' ∨ distinguishes
(Φα q''') q''' p'›
      unfolding Qα_def by blast
    from distinctions obtain Φη where
      ‹∀q'. q'∈{q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨
distinguishes φ q' p)}
        ⟶ (distinguishes (Φη q') p q' ∨ distinguishes (Φη q') q' p) ∧ (Φη q') ∈ 𝒪 (E
∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
      unfolding mem_Collect_eq by moura
    hence
      ‹∀q'∈{q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)}.
        (distinguishes (Φη q') p q' ∨ distinguishes (Φη q') q' p)›
      ‹∀q'∈{q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)}.
         (Φη q') ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
      by blast+
    from distinction_conjunctification_two_way[OF this(1)] distinction_conjunctification_two_way_price[OF
this]
      have ‹∀q'∈{q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q'
∨ distinguishes φ q' p)}.
        hml_srbb_conj.distinguishes ((if distinguishes (Φη q') p q' then conjunctify_distinctions
else conjunctify_distinctions_dual) Φη p q') p q' ∧
         (if distinguishes (Φη q') p q' then conjunctify_distinctions else conjunctify_distinctions_dual)
Φη p q' ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
        by blast
    then obtain Ψη where distinctions_η:
      ‹∀q'∈{q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)}.
        hml_srbb_conj.distinguishes (Ψη q') p q' ∧ Ψη q' ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞
∞ ∞)›
      by auto
    have ‹p ↦a α p'› using ‹p ↦ α p'› by auto
    from distinction_combination_eta_two_way[OF this, of q Φα] distinctions_α have obs_dist:
      ‹∀q'∈Qα.
        hml_srbb_inner.distinguishes (Obs α (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a
α q'' ∧ q'' ↠ q'''}
        (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions else conjunctify_distin
Φα p'
                   q''')))) p q'›
      unfolding Qα_def by fastforce
    have ‹Qα ≠ {}›
      using Qα_def contradiction(1) silent_reachable.refl by fastforce
    hence conjunct_prices: ‹∀q'''∈{q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}.
          ((if distinguishes (Φα q''') p' q''' then conjunctify_distinctions else conjunctify_distinction
Φα p' q''') ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
      using distinction_conjunctification_two_way_price[of ‹{q'''. ∃q'∈Qα. ∃q''. q' ↦a
α q'' ∧ q'' ↠ q'''}›]
      using Qα_def Φα_def by auto
    have ‹(Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
          (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions else
conjunctify_distinctions_dual) Φα p'
                   q''')) ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
    proof (cases ‹{q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''} = {}›)
```

```
      case True
      then show ?thesis
        unfolding 𝒪_inner_def 𝒪_conjunct_def
        by (auto simp add: True bot_enat_def)
    next
      case False
      then show ?thesis
        using conjunct_prices
        unfolding 𝒪_inner_def 𝒪_conjunct_def by force
    qed
    hence obs_price: ‹(Obs α (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠
q'''}
         (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions else
conjunctify_distinctions_dual) Φα p'
                   q''')))) ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
      using distinction_conjunctification_price distinctions_α unfolding 𝒪_inner_def 𝒪_def
by simp
    from obs_dist distinctions_η have
      ‹hml_srbb_inner_models p (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions
else conjunctify_distinctions_dual) Φα p'
                   q''')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)} Ψη)›
      using ‹Qα ≠ {}› silent_reachable.refl
      unfolding hml_srbb_conj.distinguishes_def hml_srbb_inner.distinguishes_def
      by (smt (verit) Qα_def empty_Collect_eq hml_srbb_inner_models.simps(1,4) mem_Collect_eq)
    moreover have ‹∀q'. q ↠ q' ⟶ ¬ hml_srbb_inner_models q'
        (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions
else conjunctify_distinctions_dual) Φα p'
                   q''')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)} Ψη)›
    proof safe
      fix q'
      assume contradiction: ‹q ↠ q'›
        ‹hml_srbb_inner_models q' (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions
else conjunctify_distinctions_dual) Φα p'
                   q''')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)} Ψη)›
      thus ‹False›
        using obs_dist distinctions_η  branching_conj_obs branching_conj_parts
        unfolding distinguishes_def hml_srbb_conj.distinguishes_def hml_srbb_inner.distinguishes_def
Qα_def
        by blast
    qed
    moreover have branch_price: ‹(BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions
else conjunctify_distinctions_dual) Φα p'
                   q''')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)} Ψη)
      ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
      using distinctions_η obs_price
```

```
      unfolding Qα_def 𝒪_inner_def 𝒪_def 𝒪_conjunct_def Φα_def
      by (simp, metis (mono_tags, lifting) SUP_bot_conv(2) bot_enat_def sup_bot_left)
    ultimately have ‹distinguishes (Internal (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions
else conjunctify_distinctions_dual) Φα p'
                   q''')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)} Ψη)) p q›
      unfolding distinguishes_def Qα_def
      using silent_reachable.refl hml_srbb_models.simps(2) by blast
    moreover have ‹(Internal (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (λq'''. (if distinguishes (Φα q''') p' q''' then conjunctify_distinctions
else conjunctify_distinctions_dual) Φα p'
                   q''')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞). distinguishes φ p q' ∨ distinguishes
φ q' p)} Ψη))
      ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞)›
      using branch_price
      unfolding Qα_def 𝒪_def 𝒪_conjunct_def
      by (metis (no_types, lifting) 𝒪_inner_def expr_internal_eq mem_Collect_eq)
    ultimately show False using contradiction(1) equivalent_no_distinction by blast
  qed
  thus ?thesis
    unfolding eta_simulation_def by blast
qed

theorem ‹(p ~η q) = (p ∼ (E ∞ ∞ ∞ 0 0 ∞ ∞ ∞) q)›
  using modal_eta_sim_eq logic_eta_bisim_invariant sympD equivalent_no_distinction
  unfolding eta_bisimulated_def expr_equiv_def distinguishes_def
  by (smt (verit, best) equivalent_equiv equivpE)
```

— This proof essentially is a simpler version of the proof for the equivalence

```
lemma modal_eta_sim: ‹eta_simulation (preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)))›
proof -
  have ‹∄p α p' q. (preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0))) p q ∧ p ↦ α p' ∧
      (α ≠ τ ∨ ¬(preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0))) p' q) ∧
      (∀q' q'' q'''. q ↠ q' ⟶ q' ↦ α q'' ⟶ q'' ↠ q''' ⟶
    ¬ preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)) p q' ∨ ¬ preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞
0 0)) p' q''')›
  proof clarify
    have less_obs: ‹modal_depth (E ∞ ∞ ∞ 0 0 ∞ 0 0) ≤ pos_conjuncts (E ∞ ∞ ∞ 0 0
∞ 0 0)› by simp
    fix p α p' q
    define Qα where ‹Qα ≡ {q'. q ↠ q' ∧ (∄φ. φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0) ∧ distinguishes
φ p q')}›
    assume contradiction:
      ‹preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)) p q› ‹p ↦ α p'›
      ‹∀q' q'' q'''. q ↠ q' ⟶ q' ↦ α q'' ⟶ q'' ↠ q''' ⟶
      ¬ preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)) p q' ∨ ¬ preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞
0 0)) p' q'''›
      ‹α ≠ τ ∨ ¬ preordered (𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)) p' q›
    hence distinctions: ‹∀q'. q ↠ q' ⟶
      (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q') ∨
      (∀q'' q'''. q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes
φ p' q'''))›
      unfolding preordered_no_distinction
      by (metis silent_reachable.cases silent_reachable.refl)
    hence ‹∀q'' q''' . ∀q'∈Qα.
        q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes
```

```
φ p' q''')›
      unfolding Qα_def using silent_reachable.refl by fastforce
    hence ‹∀q'' q'''. q'' ↠ q''' ⟶ (∃q'. q ↠ q' ∧ (∄φ. φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)
∧ distinguishes φ p q') ∧ q' ↦a α q'')
        ⟶ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p' q''')›
      unfolding Qα_def by blast
    hence ‹∀q'''. (∃q' q''. q ↠ q' ∧ (∄φ. φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0) ∧ distinguishes
φ p q') ∧ q' ↦a α q'' ∧  q'' ↠ q''')
        ⟶ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p' q''')›
      by blast
    then obtain Φα where Φα_def:
      ‹∀q'''. (∃q' q''. q ↠ q' ∧ (∄φ. φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0) ∧ distinguishes φ
p q') ∧ q' ↦a α q'' ∧ q'' ↠ q''')
        ⟶ (Φα q''') ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0) ∧ distinguishes (Φα q''') p' q'''› by
metis
    hence distinctions_α: ‹∀q'∈Qα. ∀q'' q'''.
        q' ↦a α q'' ⟶ q'' ↠ q''' ⟶ distinguishes (Φα q''') p' q'''›
      unfolding Qα_def by blast
    from distinctions obtain Φη where
      ‹∀q'. q'∈{q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q')}
        ⟶ distinguishes (Φη q') p q' ∧ (Φη q') ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)› unfolding
mem_Collect_eq by moura
    then obtain Ψη where distinctions_η:
      ‹∀q'∈{q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q')}.
        hml_srbb_conj.distinguishes (Ψη q') p q' ∧ (Ψη q') ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0
∞ 0 0)›
      using less_obs  distinction_conjunctification distinction_conjunctification_price
      by (smt (verit, del_insts))
    have ‹p ↦a α p'› using ‹p ↦ α p'› by auto
    from distinction_combination_eta[OF this] distinctions_α have obs_dist:
      ‹∀q'∈Qα.
        hml_srbb_inner.distinguishes (Obs α (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a
α q'' ∧ q'' ↠ q'''}
                                                    (conjunctify_distinctions Φα p'))))
p q'›
      unfolding Qα_def by blast
    have ‹Qα ≠ {}›
      using Qα_def contradiction(1) silent_reachable.refl by fastforce
    hence conjunct_prices: ‹∀q'''∈{q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}.
         (conjunctify_distinctions Φα p' q''') ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
      using distinction_conjunctification_price[of ‹{q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧
q'' ↠ q'''}›]
      using Qα_def Φα_def by auto
    have ‹(Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
         (conjunctify_distinctions Φα p')) ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
    proof (cases ‹{q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''} = {}›)
      case True
      then show ?thesis
        unfolding 𝒪_inner_def 𝒪_conjunct_def
        by (auto simp add: True bot_enat_def)
    next
      case False
      then show ?thesis
        using conjunct_prices
        unfolding 𝒪_inner_def 𝒪_conjunct_def by force
    qed
    hence obs_price: ‹(Obs α (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠
q'''}
        (conjunctify_distinctions Φα p')))) ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
      using distinction_conjunctification_price distinctions_α unfolding 𝒪_inner_def 𝒪_def
by simp
```

```
    from obs_dist distinctions_η have
      ‹hml_srbb_inner_models p (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (conjunctify_distinctions Φα p')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q')} Ψη)›
      using contradiction(1) silent_reachable.refl
      unfolding Qα_def by force
    moreover have ‹∀q'. q ↠ q' ⟶ ¬ hml_srbb_inner_models q'
        (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (conjunctify_distinctions Φα p')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q')} Ψη)›
    proof safe
      fix q'
      assume contradiction: ‹q ↠ q'›
        ‹hml_srbb_inner_models q' (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (conjunctify_distinctions Φα p')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q')} Ψη)›
      thus ‹False›
        using obs_dist distinctions_η
        unfolding distinguishes_def hml_srbb_conj.distinguishes_def hml_srbb_inner.distinguishes_def
Qα_def
        by (auto) blast+
    qed
    moreover have branch_price: ‹(BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (conjunctify_distinctions Φα p')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q')} Ψη)
      ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
      using distinctions_η obs_price
      unfolding Qα_def 𝒪_inner_def 𝒪_def 𝒪_conjunct_def Φα_def
      by (simp, metis (mono_tags, lifting) SUP_bot_conv(2) bot_enat_def sup_bot_left)
    ultimately have ‹distinguishes (Internal (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (conjunctify_distinctions Φα p')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q')} Ψη)) p
q›
      unfolding distinguishes_def Qα_def
      using silent_reachable.refl hml_srbb_models.simps(2) by blast
    moreover have ‹(Internal (BranchConj α
          (Internal (Conj {q'''. ∃q'∈Qα. ∃q''. q' ↦a α q'' ∧ q'' ↠ q'''}
                   (conjunctify_distinctions Φα p')))
          {q'. q ↠ q' ∧ (∃φ∈𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0). distinguishes φ p q')} Ψη))
      ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
      using branch_price
      unfolding Qα_def 𝒪_def 𝒪_conjunct_def
      by (metis (no_types, lifting) 𝒪_inner_def expr_internal_eq mem_Collect_eq)
    ultimately show False using contradiction(1) preordered_no_distinction by blast
  qed
  thus ?thesis
    unfolding eta_simulation_def by blast
qed

theorem ‹(p ≼ (E ∞ ∞ ∞ 0 0 ∞ 0 0) q) ⟹ (∃R. eta_simulation R ∧ R p q)›
  using modal_eta_sim unfolding expr_preord_def
  by auto

end

end
```

8 Branching Bisimilarity

```
theory Branching_Bisimilarity
  imports Eta_Bisimilarity
begin
```

8.1 Definitions of (Stability-Respecting) Branching Bisimilarity

```
context LTS_Tau
begin

definition branching_simulation :: ‹('s ⇒ 's ⇒ bool) ⇒ bool› where
  ‹branching_simulation R ≡ ∀p α p' q. R p q ⟶ p ↦ α p' ⟶
    ((α = τ ∧ R p' q) ∨ (∃q' q''. q ↠ q' ∧ q' ↦ α q'' ∧ R p q' ∧ R p' q''))›

lemma branching_simulation_intro:
  assumes
    ‹⋀p α p' q. R p q ⟹ p ↦ α p' ⟹
      ((α = τ ∧ R p' q) ∨ (∃q' q''. q ↠ q' ∧ q' ↦ α q'' ∧ R p q' ∧ R p' q''))›
  shows
    ‹branching_simulation R›
  using assms unfolding branching_simulation_def by simp

definition branching_simulated :: ‹'s ⇒ 's ⇒ bool› where
  ‹branching_simulated p q ≡ ∃R. branching_simulation R ∧ R p q›

definition branching_bisimulated :: ‹'s ⇒ 's ⇒ bool› where
  ‹branching_bisimulated p q ≡ ∃R. branching_simulation R ∧ symp R ∧ R p q›

definition sr_branching_bisimulated :: ‹'s ⇒ 's ⇒ bool› (infix ‹~SRBB› 40) where
  ‹p ~SRBB q ≡ ∃R. branching_simulation R ∧ symp R ∧ stability_respecting R ∧ R p q›
```

8.2 Properties of Branching Bisimulation Equivalences

```
lemma branching_bisimilarity_branching_sim: ‹branching_simulation sr_branching_bisimulated›
  unfolding sr_branching_bisimulated_def branching_simulation_def by blast

lemma branching_sim_eta_sim:
  assumes ‹branching_simulation R›
  shows ‹eta_simulation R›
  using assms silent_reachable.refl unfolding branching_simulation_def eta_simulation_def
by blast

lemma silence_retains_branching_sim:
assumes
  ‹branching_simulation R›
  ‹R p q›
  ‹p ↠ p'›
shows ‹∃q'. R p' q' ∧ q ↠ q'›
  using assms silence_retains_eta_sim branching_sim_eta_sim by blast

lemma branching_bisimilarity_stability: ‹stability_respecting sr_branching_bisimulated›
  unfolding sr_branching_bisimulated_def stability_respecting_def by blast

lemma sr_branching_bisimulation_silently_retained:
  assumes
    ‹sr_branching_bisimulated p q›
    ‹p ↠ p'›
  shows
    ‹∃q'. q ↠ q' ∧ sr_branching_bisimulated p' q'› using assms(2,1)
  using branching_bisimilarity_branching_sim silence_retains_branching_sim by blast
```

```
lemma sr_branching_bisimulation_sim:
  assumes
    ‹sr_branching_bisimulated p q›
    ‹p ↠ p'› ‹p' ↦a α p''›
  shows
    ‹∃q' q''. q ↠ q' ∧ q' ↦a α q'' ∧ sr_branching_bisimulated p' q' ∧ sr_branching_bisimulated
p'' q''›
proof -
  obtain q' where ‹q ↠ q'› ‹sr_branching_bisimulated p' q'›
    using assms sr_branching_bisimulation_silently_retained by blast
  thus ?thesis
    using assms(3) branching_bisimilarity_branching_sim silent_reachable_trans
    unfolding branching_simulation_def
    by blast
qed

lemma sr_branching_bisimulated_sym:
  assumes
    ‹sr_branching_bisimulated p q›
  shows
    ‹sr_branching_bisimulated q p›
  using assms unfolding sr_branching_bisimulated_def by (meson sympD)

lemma sr_branching_bisimulated_symp:
  shows ‹symp (~SRBB)›
  using sr_branching_bisimulated_sym
  using sympI by blast

lemma sr_branching_bisimulated_reflp:
  shows ‹reflp (~SRBB)›
    unfolding sr_branching_bisimulated_def stability_respecting_def reflp_def
    using silence_retains_branching_sim silent_reachable.refl
    by (smt (verit) DEADID.rel_symp branching_simulation_intro)

lemma establish_sr_branching_bisim:
  assumes
    ‹∀α p'. p ↦ α p' ⟶
    ((α = τ ∧ p' ~SRBB q) ∨ (∃q' q''. q ↠ q' ∧ q' ↦ α q'' ∧ p ~SRBB q' ∧ p' ~SRBB q''))›
    ‹∀α q'. q ↦ α q' ⟶
    ((α = τ ∧ p ~SRBB q') ∨ (∃p' p''. p ↠ p' ∧ p' ↦ α p'' ∧ p' ~SRBB q ∧ p'' ~SRBB q'))›
    ‹stable_state p ⟶ (∃q'. q ↠ q' ∧ p ~SRBB q' ∧ stable_state q')›
    ‹stable_state q ⟶ (∃p'. p ↠ p' ∧ p' ~SRBB q ∧ stable_state p')›
  shows ‹p ~SRBB q›
proof -
  define R where ‹R ≡ λpp qq. pp ~SRBB qq ∨ (pp = p ∧ qq = q) ∨ (pp = q ∧ qq = p)›
  hence
    R_cases: ‹⋀pp qq. R pp qq ⟹ pp ~SRBB qq ∨ (pp = p ∧ qq = q) ∨ (pp = q ∧ qq = p)›
and
    bisim_extension: ‹∀pp qq. pp ~SRBB qq ⟶ R pp qq› by blast+
  have ‹symp R›
    unfolding symp_def R_def sr_branching_bisimulated_def
    by blast
  moreover have ‹stability_respecting R›
    unfolding stability_respecting_def
  proof safe
    fix pp qq
    assume ‹R pp qq› ‹stable_state pp›
    then consider ‹pp ~SRBB qq› | ‹pp = p ∧ qq = q› | ‹pp = q ∧ qq = p›
      using R_cases by blast
    thus ‹∃q'. qq ↠ q' ∧ R pp q' ∧ stable_state q'›
```

```
    proof cases
      case 1
      then show ?thesis
        using branching_bisimilarity_stability ‹stable_state pp› bisim_extension
        unfolding stability_respecting_def
        by blast
    next
      case 2
      then show ?thesis
        using assms(3) ‹stable_state pp› unfolding R_def by blast
    next
      case 3
      then show ?thesis
        using assms(4) ‹stable_state pp› ‹symp R› unfolding R_def
        by (meson sr_branching_bisimulated_sym)
    qed
  qed
  moreover have ‹branching_simulation R› unfolding branching_simulation_def
  proof clarify
    fix pp α p' qq
    assume bc: ‹R pp qq› ‹pp ↦ α p'› ‹∄q' q''. qq ↠ q' ∧ q' ↦ α q'' ∧ R pp q' ∧ R
p' q''›
    then consider ‹pp ~SRBB qq› | ‹pp = p ∧ qq = q› | ‹pp = q ∧ qq = p›
      using R_cases by blast
    thus ‹α = τ ∧ R p' qq›
    proof cases
      case 1
      then show ?thesis
        by (smt (verit, del_insts) bc bisim_extension
            branching_bisimilarity_branching_sim branching_simulation_def)
    next
      case 2
      then show ?thesis
        using bc assms(1) bisim_extension by blast
    next
      case 3
      then show ?thesis
        using bc assms(2) bisim_extension sr_branching_bisimulated_sym by metis
    qed
  qed
  moreover have ‹R p q› unfolding R_def by blast
  ultimately show ?thesis
    unfolding sr_branching_bisimulated_def by blast
qed

lemma sr_branching_bisimulation_stuttering:
  assumes
    ‹pp ≠ []›
    ‹∀i < length pp - 1.  pp!i ↦ τ pp!(Suc i)›
    ‹hd pp ~SRBB last pp›
    ‹i < length pp›
  shows
    ‹hd pp ~SRBB pp!i›
proof -
  have chain_reachable: ‹∀j < length pp. ∀i ≤ j. pp!i ↠ pp!j›
    using tau_chain_reachabilty assms(2) .
  hence chain_hd_last:
    ‹∀i < length pp. hd pp ↠ pp!i›
    ‹∀i < length pp.  pp!i ↠ last pp›
    by (auto simp add: assms(1) hd_conv_nth last_conv_nth)
  define R where ‹R ≡ λp q. (p = hd pp ∧ (∃i < length pp. pp!i = q)) ∨ ((q = hd pp ∧ (∃i
```

```
< length pp. pp!i = p))) ∨ p ~SRBB q›
  have later_hd_sim: ‹⋀i p' α. i < length pp ⟹ pp!i ↦ α p'
    ⟹ (hd pp) ↠ (pp!i) ∧ (pp!i) ↦ α p' ∧ R (pp!i) (pp!i) ∧ R p' p'›
    using chain_hd_last sr_branching_bisimulated_reflp
    unfolding R_def
    by (simp add: reflp_def)
  have hd_later_sim: ‹⋀i p' α. i < length pp - 1 ⟹ (hd pp) ↦ α p'
    ⟹ (∃q' q''. (pp!i) ↠ q' ∧ q' ↦ α q'' ∧ R (hd pp) q' ∧ R p' q'')›
  proof -
    fix i p' α
    assume case_assm: ‹i < length pp - 1› ‹(hd pp) ↦ α p'›
    hence ‹(α = τ ∧ p' ~SRBB (last pp)) ∨ (∃q' q''. (last pp) ↠ q' ∧ q' ↦ α q'' ∧ (hd
pp) ~SRBB q' ∧ p' ~SRBB q'')›
      using ‹hd pp ~SRBB last pp› branching_bisimilarity_branching_sim branching_simulation_def
      by auto
    thus ‹(∃q' q''. (pp!i) ↠ q' ∧ q' ↦ α q'' ∧ R (hd pp) q' ∧ R p' q'')›
    proof
      assume tau_null_step: ‹α = τ ∧ p' ~SRBB last pp›
      have ‹pp ! i ↠ (pp!(length pp - 2))›
        using case_assm(1) chain_reachable by force
      moreover have ‹pp!(length pp - 2) ↦ α (last pp)›
        using assms(1,2) case_assm(1) last_conv_nth tau_null_step
        by (metis Nat.lessE Suc_1 Suc_diff_Suc less_Suc_eq zero_less_Suc zero_less_diff)
      moreover have ‹R (hd pp) (pp!(length pp - 2)) ∧ R p' (last pp)›
        unfolding R_def
        by (metis assms(1) diff_less length_greater_0_conv less_2_cases_iff tau_null_step)
      ultimately show ‹∃q' q''. pp ! i ↠ q' ∧ q' ↦ α q'' ∧ R (hd pp) q' ∧ R p' q''› by
blast
    next
      assume ‹∃q' q''. last pp ↠ q' ∧ q' ↦ α q'' ∧ hd pp ~SRBB q' ∧ p' ~SRBB q''›
      hence ‹∃q' q''. last pp ↠ q' ∧ q' ↦ α q'' ∧ R (hd pp) q' ∧ R p' q''›
        unfolding R_def by blast
      moreover have ‹i < length pp› using case_assm by auto
      ultimately show ‹∃q' q''. pp ! i ↠ q' ∧ q' ↦ α q'' ∧ R (hd pp) q' ∧ R p' q''›
        using chain_hd_last silent_reachable_trans by blast
    qed
  qed
  have ‹branching_simulation R›
  proof (rule branching_simulation_intro)
    fix p α p' q
    assume challenge: ‹R p q› ‹p ↦ α p'›
    from this(1) consider
      ‹(p = hd pp ∧ (∃i < length pp. pp!i = q))› |
      ‹(q = hd pp ∧ (∃i < length pp. pp!i = p))› |
      ‹ p ~SRBB q› unfolding R_def by blast
    thus ‹α = τ ∧ R p' q ∨ (∃q' q''. q ↠ q' ∧ q' ↦ α q'' ∧ R p q' ∧ R p' q'')›
    proof cases
      case 1
      then obtain i where i_spec: ‹i < length pp› ‹pp ! i = q› by blast
      from 1 have ‹p = hd pp› ..
      show ?thesis
      proof (cases ‹i = length pp - 1›)
        case True
        then have ‹q = last pp› using i_spec assms(1)
          by (simp add: last_conv_nth)
        then show ?thesis using challenge(2) assms(3) branching_bisimilarity_branching_sim
          unfolding R_def branching_simulation_def ‹p = hd pp›
          by metis
      next
        case False
        hence ‹i < length pp - 1› using i_spec by auto
```

```
      then show ?thesis using ‹p = hd pp› i_spec hd_later_sim challenge(2) by blast
    qed
  next
    case 2
    then show ?thesis
      using later_hd_sim challenge(2) by blast
  next
    case 3
    then show ?thesis
      using challenge(2) branching_bisimilarity_branching_sim
      unfolding branching_simulation_def R_def by metis
  qed
 qed
 moreover have ‹symp R›
   using sr_branching_bisimulated_sym
   unfolding R_def sr_branching_bisimulated_def
   by (smt (verit, best) sympI)
 moreover have ‹stability_respecting R›
   using assms(3) stable_state_stable sr_branching_bisimulated_sym
     branching_bisimilarity_stability
   unfolding R_def stability_respecting_def
   by (metis chain_hd_last)
 moreover have ‹⋀i. i < length pp ⟹ R (hd pp) (pp!i)› unfolding R_def by auto
 ultimately show ?thesis
   using assms(4) sr_branching_bisimulated_def by blast
qed

lemma sr_branching_bisimulation_stabilizes:
  assumes
    ‹sr_branching_bisimulated p q›
    ‹stable_state p›
  shows
    ‹∃q'. q ↠ q' ∧ sr_branching_bisimulated p q' ∧ stable_state q'›
proof -
  from assms obtain R where
    R_spec: ‹branching_simulation R› ‹symp R› ‹stability_respecting R› ‹R p q›
    unfolding sr_branching_bisimulated_def by blast
  then obtain q' where ‹q ↠ q'› ‹stable_state q'›
    using assms(2) unfolding stability_respecting_def by blast
  moreover have ‹sr_branching_bisimulated p q'›
    using sr_branching_bisimulation_stuttering
     assms(1) calculation(1) sr_branching_bisimulated_def sympD
    by (metis assms(2) sr_branching_bisimulation_silently_retained stable_state_stable)
  ultimately show ?thesis by blast
qed

lemma sr_branching_bisim_stronger:
  assumes
    ‹sr_branching_bisimulated p q›
  shows
    ‹branching_bisimulated p q›
  using assms unfolding sr_branching_bisimulated_def branching_bisimulated_def by auto
```

8.3 HML_SRBB as Modal Characterization of Stability-Respecting Branching Bisimilarity

```
lemma modal_sym: ‹symp (preordered UNIV)›
proof-
  have ‹∄ p q. preordered UNIV p q ∧ ¬preordered UNIV q p›
  proof safe
    fix p q
```

```
    assume contradiction:
      ‹preordered UNIV p q›
      ‹¬preordered UNIV q p›
    then obtain φ where φ_distinguishes: ‹distinguishes φ q p› by auto
    thus False
    proof (cases φ)
      case TT
      then show ?thesis using φ_distinguishes by auto
    next
      case (Internal χ)
      hence ‹distinguishes (ImmConj {undefined} (λi. Neg χ)) p q›
        using φ_distinguishes by simp
      then show ?thesis using contradiction preordered_no_distinction by blast
    next
      case (ImmConj I Ψ)
      then obtain i where i_def: ‹i ∈ I› ‹hml_srbb_conj.distinguishes (Ψ i) q p›
        using φ_distinguishes srbb_dist_imm_conjunction_implies_dist_conjunct by auto
      then show ?thesis
      proof (cases ‹Ψ i›)
        case (Pos χ)
        hence ‹distinguishes (ImmConj {undefined} (λi. Neg χ)) p q› using i_def by simp
        thus ?thesis using contradiction preordered_no_distinction by blast
      next
        case (Neg χ)
        hence ‹distinguishes (Internal χ) p q› using i_def by simp
        thus ?thesis using contradiction preordered_no_distinction by blast
      qed
    qed
  qed
  thus ?thesis unfolding symp_def by blast
qed

lemma modal_branching_sim: ‹branching_simulation (preordered UNIV)›
proof -
  have ‹∄p α p' q. (preordered UNIV) p q ∧ p ↦ α p' ∧
      (α ≠ τ ∨ ¬(preordered UNIV) p' q) ∧
      (∀q' q''. q ↠ q' ⟶ q' ↦ α q'' ⟶ ¬ preordered UNIV p q' ∨ ¬ preordered UNIV
p' q'')›
  proof clarify
    fix p α p' q
    define Qα where ‹Qα ≡ {q'. q ↠ q' ∧ (∄φ. distinguishes φ p q')}›
    assume contradiction:
      ‹preordered UNIV p q› ‹p ↦ α p'›
      ‹∀q' q''. q ↠ q' ⟶ q' ↦ α q'' ⟶ ¬ preordered UNIV p q' ∨ ¬ preordered UNIV
p' q''›
      ‹α ≠ τ ∨ ¬ preordered UNIV p' q›
    hence distinctions: ‹∀q'. q ↠ q' ⟶
      (∃φ. distinguishes φ p q') ∨
      (∀q''. q' ↦a α q'' ⟶ (∃φ. distinguishes φ p' q''))›
      using preordered_no_distinction
      by (metis equivpI equivp_def lts_semantics.preordered_preord modal_sym)
    hence ‹∀q''. ∀q'∈Qα.
        q' ↦a α q'' ⟶ (∃φ. distinguishes φ p' q'')›
      unfolding Qα_def by auto
    hence ‹∀q''. (∃q'. q ↠ q' ∧ (∄φ. distinguishes φ p q') ∧ q' ↦a α q'')
        ⟶ (∃φ. distinguishes φ p' q'')›
      unfolding Qα_def by blast
    then obtain Φα where
      ‹∀q''. (∃q'. q ↠ q' ∧ (∄φ. distinguishes φ p q') ∧ q' ↦a α q'')
        ⟶ distinguishes (Φα q'') p' q''› by metis
    hence distinctions_α: ‹∀q'∈Qα. ∀q''.
```

```
        q' ↦a α q'' ⟶ distinguishes (Φα q'') p' q''›
      unfolding Qα_def by blast
    from distinctions obtain Φη where
      ‹∀q'. q'∈{q'. q ↠ q' ∧ (∃φ. distinguishes φ p q')}
        ⟶ distinguishes (Φη q') p q'› unfolding mem_Collect_eq by moura
    with distinction_conjunctification obtain Ψη where distinctions_η:
      ‹∀q'∈{q'. q ↠ q' ∧ (∃φ. distinguishes φ p q')}. hml_srbb_conj.distinguishes (Ψη
q') p q'›
      by blast
    have ‹p ↦a α p'› using ‹p ↦ α p'› by auto
    from distinction_combination[OF this] distinctions_α have obs_dist:
      ‹∀q'∈Qα.
        hml_srbb_inner.distinguishes (Obs α (ImmConj {q''. ∃q'''∈Qα. q''' ↦a α q''}
                                             (conjunctify_distinctions Φα p')))
p q'›
      unfolding Qα_def by blast
    with distinctions_η have
      ‹hml_srbb_inner_models p (BranchConj α
          (ImmConj {q''. ∃q'''∈Qα. q''' ↦a α q''}
                   (conjunctify_distinctions Φα p'))
          {q'. q ↠ q' ∧ (∃φ. distinguishes φ p q')} Ψη)›
      using contradiction(1) silent_reachable.refl
      unfolding Qα_def distinguishes_def hml_srbb_conj.distinguishes_def hml_srbb_inner.distinguishes_def
preordered_def
      by simp force
    moreover have ‹∀q'. q ↠ q' ⟶ ¬ hml_srbb_inner_models q'
        (BranchConj α (ImmConj {q''. ∃q'''∈Qα. q''' ↦a α q''} (conjunctify_distinctions
Φα p')) {q'. q ↠ q' ∧ (∃φ. distinguishes φ p q')} Ψη)›
    proof safe
      fix q'
      assume contradiction: ‹q ↠ q'›
        ‹hml_srbb_inner_models q' (BranchConj α (ImmConj {q''. ∃q'''∈Qα. q''' ↦a α q''}
(conjunctify_distinctions Φα p')) {q'. q ↠ q' ∧ (∃φ. distinguishes φ p q')} Ψη)›
      thus ‹False›
        using obs_dist distinctions_η
        unfolding distinguishes_def hml_srbb_conj.distinguishes_def hml_srbb_inner.distinguishes_def
Qα_def
        by (auto) blast+
    qed
    ultimately have ‹distinguishes (Internal (BranchConj α (ImmConj {q''. ∃q'''∈Qα. q'''
↦a α q''} (conjunctify_distinctions Φα p')) {q'. q ↠ q' ∧ (∃φ. distinguishes φ p q')}
Ψη)) p q›
      unfolding distinguishes_def Qα_def
      using silent_reachable.refl by (auto) blast+
    thus False using contradiction(1) preordered_no_distinction by blast
  qed
  thus ?thesis
    unfolding branching_simulation_def by blast
qed

lemma logic_sr_branching_bisim_invariant:
  assumes
    ‹sr_branching_bisimulated p0 q0›
    ‹p0 ⊨SRBB φ›
  shows ‹q0 ⊨SRBB φ›
proof-
  have ‹⋀φ χ ψ.
    (∀p q. sr_branching_bisimulated p q ⟶ p ⊨SRBB φ ⟶ q ⊨SRBB φ) ∧
    (∀p q. sr_branching_bisimulated p q ⟶ hml_srbb_inner_models p χ ⟶ (∃q'. q ↠ q'
∧ hml_srbb_inner_models q' χ)) ∧
    (∀p q. sr_branching_bisimulated p q ⟶ hml_srbb_conjunct_models p ψ ⟶ hml_srbb_conjunct_models
```

```
q ψ)›
  proof-
    fix φ χ ψ
    show
      ‹(∀p q. sr_branching_bisimulated p q ⟶ p ⊨SRBB φ ⟶ q ⊨SRBB φ) ∧
      (∀p q. sr_branching_bisimulated p q ⟶ hml_srbb_inner_models p χ ⟶ (∃q'. q ↠
q' ∧ hml_srbb_inner_models q' χ)) ∧
      (∀p q. sr_branching_bisimulated p q ⟶ hml_srbb_conjunct_models p ψ ⟶ hml_srbb_conjunct_models
q ψ)›
    proof (induct rule: hml_srbb_hml_srbb_inner_hml_srbb_conjunct.induct)
      case TT
      then show ?case by simp
    next
      case (Internal χ)
      show ?case
      proof safe
        fix p q
        assume ‹sr_branching_bisimulated p q› ‹p ⊨SRBB hml_srbb.Internal χ›
        then obtain p' where ‹p ↠ p'› ‹hml_srbb_inner_models p' χ› by auto
        hence ‹∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ› using Internal ‹hml_srbb_inner_models
p' χ›
          by (meson LTS_Tau.silent_reachable_trans ‹p ~SRBB q› sr_branching_bisimulation_silently_retaine
        thus ‹q ⊨SRBB hml_srbb.Internal χ› by auto
      qed
    next
      case (ImmConj I Ψ)
      then show ?case by auto
    next
      case (Obs α φ)
      then show ?case
      proof (safe)
        fix p q
        assume
          ‹sr_branching_bisimulated p q›
          ‹hml_srbb_inner_models p (hml_srbb_inner.Obs α φ)›
        then obtain p' where ‹p ↦a α p'› ‹p' ⊨SRBB φ› by auto
        then obtain q' q'' where ‹q ↠ q'› ‹q' ↦a α q''› ‹sr_branching_bisimulated p'
q''›
          using sr_branching_bisimulation_sim[OF ‹sr_branching_bisimulated p q›] silent_reachable.refl
          by blast
        hence ‹q'' ⊨SRBB φ› using ‹p' ⊨SRBB φ› Obs by blast
        hence ‹hml_srbb_inner_models q' (hml_srbb_inner.Obs α φ)›
          using ‹q' ↦a α q''› by auto
        thus ‹∃q'. q ↠ q' ∧ hml_srbb_inner_models q' (hml_srbb_inner.Obs α φ)›
          using ‹q ↠ q'› by blast
      qed
    next
      case (Conj I Ψ)
      show ?case
      proof safe
        fix p q
        assume
          ‹sr_branching_bisimulated p q›
          ‹hml_srbb_inner_models p (hml_srbb_inner.Conj I Ψ)›
        hence ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)› by auto
        hence ‹∀i∈I. hml_srbb_conjunct_models q (Ψ i)›
          using Conj ‹sr_branching_bisimulated p q› by blast
        hence ‹hml_srbb_inner_models q (hml_srbb_inner.Conj I Ψ)› by simp
        thus ‹∃q'. q ↠ q' ∧ hml_srbb_inner_models q' (hml_srbb_inner.Conj I Ψ)›
          using silent_reachable.refl by blast
      qed
```

```
    next
      case (StableConj I Ψ) show ?case
      proof safe
        fix p q
        assume
          ‹sr_branching_bisimulated p q›
          ‹hml_srbb_inner_models p (StableConj I Ψ)›
        hence ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)›
          using stable_conj_parts by blast
        from ‹hml_srbb_inner_models p (StableConj I Ψ)› have ‹stable_state p› by auto
        then obtain q' where ‹q ↠ q'› ‹stable_state q'› ‹sr_branching_bisimulated p q'›
          using ‹sr_branching_bisimulated p q› sr_branching_bisimulation_stabilizes by blast
        hence ‹∀i∈I. hml_srbb_conjunct_models q' (Ψ i)›
          using ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)› StableConj by blast
        hence ‹hml_srbb_inner_models q' (StableConj I Ψ)› using ‹stable_state q'› by simp
        thus ‹∃q'. q ↠ q' ∧ hml_srbb_inner_models q' (StableConj I Ψ)›
          using ‹q ↠ q'› by blast
      qed
    next
      case (BranchConj α φ I Ψ)
      show ?case
      proof safe
        fix p q
        assume
          ‹sr_branching_bisimulated p q›
          ‹hml_srbb_inner_models p (BranchConj α φ I Ψ)›
        hence ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)›
              ‹hml_srbb_inner_models p (Obs α φ)›
          using branching_conj_parts branching_conj_obs by blast+
        then obtain p' where ‹p ↦a α p'› ‹p' ⊨SRBB φ› by auto
        then obtain q' q'' where q'_q''_spec:
          ‹q ↠ q'› ‹q' ↦a α q''›
          ‹sr_branching_bisimulated p q'› ‹sr_branching_bisimulated p' q''›
          using sr_branching_bisimulation_sim[OF ‹sr_branching_bisimulated p q›]
            silent_reachable.refl[of p]
          by blast
        hence ‹q'' ⊨SRBB φ› using BranchConj.hyps ‹p' ⊨SRBB φ› by auto
        hence ‹hml_srbb_inner_models q' (Obs α φ)› using q'_q''_spec by auto
        moreover have ‹∀i∈I. hml_srbb_conjunct_models q' (Ψ i)›
          using BranchConj.hyps ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)› q'_q''_spec by
blast
        ultimately show ‹∃q'. q ↠ q' ∧ hml_srbb_inner_models q' (BranchConj α φ I Ψ)›
          using ‹q ↠ q'› by auto
      qed
    next
      case (Pos χ)
      show ?case
      proof safe
        fix p q
        assume
          ‹sr_branching_bisimulated p q›
          ‹hml_srbb_conjunct_models p (Pos χ)›
        then obtain p' where ‹p ↠ p'› ‹hml_srbb_inner_models p' χ› by auto
        then obtain q' where ‹q ↠ q'› ‹hml_srbb_inner_models q' χ›
          using Pos ‹p ~SRBB q› sr_branching_bisimulation_silently_retained
          by (meson  silent_reachable_trans)
        thus ‹hml_srbb_conjunct_models q (Pos χ)› by auto
      qed
    next
      case (Neg χ)
      show ?case
```

```
      proof safe
        fix p q
        assume
          ‹sr_branching_bisimulated p q›
          ‹hml_srbb_conjunct_models p (Neg χ)›
        hence ‹∀p'. p ↠ p' ⟶ ¬hml_srbb_inner_models p' χ› by simp
        moreover have
          ‹(∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ) ⟶ (∃p'. p ↠ p' ∧ hml_srbb_inner_models
p' χ)›
          using Neg sr_branching_bisimulated_sym[OF ‹sr_branching_bisimulated p q›]
            sr_branching_bisimulation_silently_retained
          by (meson silent_reachable_trans)
        ultimately have ‹∀q'. q ↠ q' ⟶ ¬hml_srbb_inner_models q' χ› by blast
        thus ‹hml_srbb_conjunct_models q (Neg χ)› by simp
      qed
    qed
  qed
  thus ?thesis using assms by blast
qed

lemma sr_branching_bisim_is_hmlsrbb: ‹sr_branching_bisimulated p q = preordered UNIV p q›
  using modal_stability_respecting modal_sym modal_branching_sim logic_sr_branching_bisim_invariant
    𝒪_sup preordered_def
  unfolding sr_branching_bisimulated_def by metis

lemma sr_branching_bisimulated_transitive:
  assumes
    ‹p ~SRBB q›
    ‹q ~SRBB r›
  shows
    ‹p ~SRBB r›
  using assms unfolding sr_branching_bisim_is_hmlsrbb by simp

lemma sr_branching_bisimulated_equivalence: ‹equivp (~SRBB)›
proof (rule equivpI)
  show ‹symp (~SRBB)› using sr_branching_bisimulated_symp .
  show ‹reflp (~SRBB)› using sr_branching_bisimulated_reflp .
  show ‹transp (~SRBB)›
    unfolding transp_def using sr_branching_bisimulated_transitive by blast
qed

lemma sr_branching_bisimulation_stuttering_all:
  assumes
    ‹pp ≠ []›
    ‹∀i < length pp - 1.  pp!i ↦ τ pp!(Suc i)›
    ‹hd pp ~SRBB last pp›
    ‹i ≤ j› ‹j < length pp›
  shows
    ‹pp!i ~SRBB pp!j›
  using assms equivp_def sr_branching_bisimulated_equivalence equivp_def order_le_less_trans
    sr_branching_bisimulation_stuttering
  by metis

theorem ‹(p ~SRBB q) = (p ≼ (E ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞) q)›
  using sr_branching_bisim_is_hmlsrbb 𝒪_sup
  unfolding expr_preord_def by auto

end

end
```

9 Energy Games

```
theory Energy_Games
  imports Main Misc
begin
```

In this theory, we introduce energy games and give basic definitions such as winning budgets. Energy games are the foundation for the later introduced weak spectroscopy game, which is an energy game itself, characterizing equivalence problems.

9.1 Fundamentals

We use an abstract concept of energies and only later consider eight-dimensional energy games. Through our later given definition of energies as a data type, we obtain certain properties that we enforce for all energy games. We therefore assume that an energy game has a partial order on energies such that all updates are monotonic and have sink where the defender wins.

```
type_synonym 'energy update = ‹'energy ⇒ 'energy option›
```

An energy game is played by two players on a directed graph labelled by energy updates. These updates represent the costs of choosing a certain move. Since we will only consider cases in which the attacker's moves may actually have non-zero costs, only they can run out of energy. This is the case when the energy level reaches the `defender_win_level`. In contrast to other definitions of games, we do not fix a starting position.

```
locale energy_game =
fixes
  weight_opt :: ‹'gstate ⇒ 'gstate ⇒ 'energy update option› and
  defender :: ‹'gstate ⇒ bool› (‹Gd›) and
  ord:: ‹'energy ⇒ 'energy ⇒ bool›
assumes
  antisim: ‹⋀e e'. (ord e e') ⟹ (ord e' e) ⟹ e = e'› and
  monotonicity: ‹⋀g g' e e' eu eu'.
    weight_opt g g' ≠ None ⟹ the (weight_opt g g') e = Some eu ⟹ the (weight_opt g g')
e' = Some eu'
    ⟹ ord e e' ⟹ ord eu eu'› and
  defender_win_min: ‹⋀g g' e e'. ord e e' ⟹ weight_opt g g' ≠ None ⟹ the (weight_opt
g g') e' = None ⟹ the (weight_opt g g') e = None›
begin
```

In the following, we introduce some abbreviations for attacker positions and moves.

```
abbreviation attacker :: ‹'gstate ⇒ bool› (‹Ga›) where ‹Ga p ≡ ¬ Gd p›

abbreviation moves :: ‹'gstate ⇒ 'gstate ⇒ bool› (infix ‹↣› 70) where ‹g1 ↣ g2 ≡ weight_opt
g1 g2 ≠ None›

abbreviation weighted_move :: ‹'gstate ⇒ 'energy update ⇒ 'gstate ⇒  bool› (‹_ ↣wgt
_ _› [60,60,60] 70) where
  ‹weighted_move g1 u g2 ≡ g1 ↣ g2 ∧ (the (weight_opt g1 g2) = u)›

abbreviation ‹weight g1 g2 ≡ the (weight_opt g1 g2)›

abbreviation ‹updated g g' e ≡ the (weight g g' e)›
```

9.1.1 Winning Budgets

The attacker wins a game if and only if they manage to force the defender to get stuck before running out of energy. The needed amount of energy is described by winning budgets: `e` is in the winning budget of `g` if and only if there exists a winning strategy for the attacker when starting in `g` with energy `e`. In more detail, this yields the following definition:

- If g is an attacker position and e is not the defender_win_level then e is in the winning budget of g if and only if there exists a position g' the attacker can move to. In other words, if the updated energy level is in the winning budget of g'. (This corresponds to the second case of the following definition.)

- If g is a defender position and e is not the defender_win_level then e is in the winning budget of g if and only if for all successors g' the accordingly updated energy is in the winning budget of g'. In other words, if the attacker will win from every successor the defender can move to.

```
inductive attacker_wins:: ‹'energy ⇒ 'gstate ⇒ bool› where
  Attack: ‹attacker_wins e g› if
    ‹Ga g› ‹g ↣ g'› ‹weight g g' e = Some e'› ‹attacker_wins e' g'› |
  Defense: ‹attacker_wins e g› if
    ‹Gd g› ‹∀g'. (g ↣ g') ⟶ (∃e'. weight g g' e = Some e' ∧ attacker_wins e' g')›
```

If from a certain starting position g a game is won by the attacker with some energy e (i.e. e is in the winning budget of g), then the game is also won by the attacker with more energy. This is proven using the inductive definition of winning budgets and the given properties of the partial order ord.

```
lemma win_a_upwards_closure:
  assumes
    ‹attacker_wins e g›
    ‹ord e e'›
  shows
    ‹attacker_wins e' g›
using assms proof (induct arbitrary: e' rule: attacker_wins.induct)
  case (Attack g g' e eu e')
  with defender_win_min obtain eu' where ‹weight g g' e' = Some eu'› by fastforce
  then show ?case
    using Attack monotonicity attacker_wins_Ga by blast
next
  case (Defense g e)
  with defender_win_min have ‹∀g'. g ↣ g' ⟶ (∃eu'. weight g g' e' = Some eu')› by fastforce
  then show ?case
    using Defense attacker_wins.Defense monotonicity by meson
qed

end

end
```

9.2 Instantiation of an Energy Game

```
theory Example_Instantiation
  imports Energy_Games "HOL-Library.Extended_Nat"
begin
```

In this theory, we create an instantiation of a two-dimensional energy game to test our definitions.

We first define energies in a similar manner to our definition of energies with two dimensions. We define component-wise subtraction.

```
datatype energy = E (one: ‹enat›) (two: ‹enat›)

abbreviation ‹direct_minus e1 e2 ≡ E ((one e1) - (one e2)) ((two e1) - (two e2))›

instantiation energy :: order
begin
```

```
fun less_eq_energy:: ‹energy ⇒ energy ⇒ bool› where
  ‹less_eq_energy (E ea1 ea2) (E eb1 eb2) = (ea1 ≤ eb1 ∧ ea2 ≤ eb2)›

fun less_energy:: ‹energy ⇒ energy ⇒ bool› where
  ‹less_energy eA eB = (eA ≤ eB ∧ ¬ eB ≤ eA)›

instance proof standard
  fix eA eB :: energy
  show ‹(eA < eB) = (eA ≤ eB ∧ ¬ eB ≤ eA)› by auto
next
  fix e :: energy
  show ‹e ≤ e›
    using less_eq_energy.elims(3) by fastforce
next
  fix eA eB eC:: energy
  assume ‹eA ≤ eB› ‹eB ≤ eC›
  thus ‹eA ≤ eC›
    by (smt (verit, del_insts) energy.inject less_eq_energy.elims order.trans)
next
  fix eA eB :: energy
  assume ‹eA ≤ eB› ‹eB ≤ eA›
  thus ‹eA = eB›
    using less_eq_energy.elims(2) by fastforce
qed
end

fun order_opt:: ‹energy option ⇒ energy option ⇒ bool› where
  ‹order_opt (Some eA) (Some eB) = (eA ≤ eB)› |
  ‹order_opt None _ = True› |
  ‹order_opt (Some eA) None = False›

definition minus_energy_def[simp]: ‹minus_energy e1 e2 ≡ if (¬e2 ≤ e1) then None
                                                else Some (direct_minus e1 e2)›
lemma energy_minus[simp]:
  assumes ‹E c d ≤ E a b›
  shows ‹minus_energy (E a b) (E c d) = Some (E (a - c) (b - d))› using assms by auto

definition min_update_def[simp]: ‹min_update e1 ≡ Some (E (min (one e1) (two e1)) (two e1))›
```

In preparation for our instantiation, we define our states, the updates for our energy levels and which states are defender positions.

```
datatype state = a | b1 | b2 | c | d1 | d2 | e

fun weight_opt :: ‹state ⇒ state ⇒ energy update option› where
  ‹weight_opt a b1 = Some (λx. minus_energy x (E 1 0))› |
  ‹weight_opt a b2 = Some (λx. minus_energy x (E 0 1))› |
  ‹weight_opt a _  = None›  |
  ‹weight_opt b1 c = Some Some› |
  ‹weight_opt b1 _  = None›  |
  ‹weight_opt b2 c = Some min_update› |
  ‹weight_opt b2 _  = None›  |
  ‹weight_opt c d1 = Some (λx. minus_energy x (E 0 1))› |
  ‹weight_opt c d2 = Some (λx. minus_energy x (E 1 0))› |
  ‹weight_opt c _  = None›  |
  ‹weight_opt d1 e = Some Some› |
  ‹weight_opt d1 _  = None›  |
  ‹weight_opt d2 e = Some Some› |
  ‹weight_opt d2 _  = None› |
  ‹weight_opt e _  = None›
```

```
find_theorems weight_opt

fun defender :: ‹state ⇒ bool› where
  ‹defender b1 = True› |
  ‹defender b2 = True› |
  ‹defender c = True› |
  ‹defender e = True› |
  ‹defender _ = False›
```

Now, we can state our energy game example.

```
interpretation Game: energy_game ‹weight_opt› ‹defender› ‹(≤)›
proof
  fix g g' and e e' eu eu' :: energy
  show ‹e ≤ e' ⟹ e' ≤ e ⟹ e = e'› by auto

  assume case_assms: ‹e ≤ e'›
   ‹the (weight_opt g g') e = Some eu› ‹the (weight_opt g g') e' = Some eu'›
   ‹weight_opt g g' ≠ None›
  hence Y: ‹weight_opt g g' = Some Some ∨ weight_opt g g' = Some min_update ∨ weight_opt
g g' = Some (λx. minus_energy x (E 1 0)) ∨ weight_opt g g' = Some (λx. minus_energy x (E
0 1))›
    using weight_opt.simps by (smt (verit, del_insts) defender.cases)
  then consider (id) ‹weight_opt g g' = Some Some› | (min) ‹weight_opt g g' = Some min_update›
|(10) ‹ weight_opt g g' = Some (λx. minus_energy x (E 1 0))› | (01) ‹ weight_opt g g' =
Some (λx. minus_energy x (E 0 1))› by auto


  then show ‹eu ≤ eu'›
  proof (cases)
    case id
    then show ?thesis
      using case_assms by auto
  next
    case min
    hence ‹min_update e = Some eu› ‹min_update e' = Some eu'› using case_assms by auto
    then show ?thesis
      using case_assms(1) by (cases e, cases e', auto simp add: min_le_iff_disj)
  next
    case 10
    hence ‹minus_energy e (E 1 0) = Some eu› ‹minus_energy e' (E 1 0) = Some eu'› using
case_assms by auto
    then show ?thesis using case_assms(1)
      by (cases e, cases e', auto,
         metis add.commute add_diff_assoc_enat energy.sel idiff_0_right le_iff_add less_eq_energy.simps
option.distinct(1) option.inject)
  next
    case 01
    hence ‹minus_energy e (E 0 1) = Some eu› ‹minus_energy e' (E 0 1) = Some eu'› using
case_assms by auto
    then show ?thesis  using case_assms(1)
      by (cases e, cases e', auto,
         metis add.commute add_diff_assoc_enat energy.sel idiff_0_right le_iff_add less_eq_energy.simps
option.distinct(1) option.inject)
  qed
next
  fix g g' e e'
  assume ‹e ≤ e'› ‹weight_opt g g' ≠ None› ‹the (weight_opt g g') e' = None›
  thus ‹the (weight_opt g g') e = None›
    by (induct g) (induct g', auto simp add: order.trans)+
qed
```

```
notation Game.moves (infix ‹↣› 70)

lemma moves:
  shows ‹a ↣ b1› ‹a ↣ b2›
        ‹b1 ↣ c› ‹b2 ↣ c›
        ‹c ↣ d1› ‹c ↣ d2›
        ‹d1 ↣ e› ‹d2 ↣ e›
        ‹¬(c ↣ e)› ‹¬(e ↣ d1)›
  by simp+
```

Our definition of winning budgets.

```
lemma wina_of_e:
  shows ‹Game.attacker_wins (E 9 8) e›
  by (simp add: Game.attacker_wins.Defense)

lemma wina_of_e_exist:
  shows ‹∃e1. Game.attacker_wins e1 e›
  using wina_of_e by blast

lemma attacker_wins_at_e:
  shows ‹∀e'. Game.attacker_wins e' e›
  by (simp add: Game.attacker_wins.Defense)

lemma wina_of_d1:
  shows ‹Game.attacker_wins (E 9 8) d1›
proof -
  have A1: ‹¬(defender d1)› by simp
  have A2: ‹d1 ↣ e› by simp
  have A3: ‹Game.attacker_wins (E 9 8) e› by (rule wina_of_e)
  have Aid: ‹Game.weight d1 e = Some› by simp
  hence ‹(Game.weight d1 e (E 9 8)) = Some (E 9 8)› by simp
  hence ‹(Game.attacker_wins (the ((Game.weight d1 e (E 9 8)))) e)› using A3 by simp
  from this A3 have A4: ‹¬(defender d1) ∧ (∃g'. ((d1 ↣ g') ∧ (Game.attacker_wins (the
((Game.weight d1 g' (E 9 8)))) g')))›
    by (meson A1 A2 Game.attacker_wins.Defense defender.simps(4) weight_opt.simps(38))
  thus ‹Game.attacker_wins (E 9 8) d1› using Game.attacker_wins.Attack A2 Aid wina_of_e
by presburger
qed

lemma wina_of_d2:
  shows ‹Game.attacker_wins (E 8 9) d2›
proof -
  have A1: ‹¬(defender d2)› by simp
  have A2: ‹d2 ↣ e› by simp
  have A3: ‹Game.attacker_wins (E 8 9) e› by (simp add: attacker_wins_at_e)
  have Aid: ‹Game.weight d2 e = Some› by simp
  hence ‹(Game.weight d2 e (E 8 9)) = Some (E 8 9)› by simp
  hence ‹(Game.attacker_wins (the ((Game.weight d2 e (E 8 9)))) e)› using A3 by simp
  from this A3 have A4: ‹¬(defender d2) ∧ (∃g'. ((d2 ↣ g') ∧ (Game.attacker_wins (the
((Game.weight d2 g' (E 8 9)))) g')))›
    by (meson A1 A2 Game.attacker_wins.Defense defender.simps(4) weight_opt.simps(38))
  thus ‹Game.attacker_wins (E 8 9) d2› using Game.attacker_wins.Attack A2 A3 Aid wina_of_e
by presburger
qed

lemma wina_of_c:
  shows ‹Game.attacker_wins (E 9 9) c›
proof -
  have A1: ‹defender c› by auto
  have A2: ‹∀g'. (c ↣ g') ⟶ (g' = d1 ∨ g' = d2)›
    by (metis moves(9) state.exhaust weight_opt.simps(24,25,26,27))
```

```
  have A3: ‹Game.attacker_wins (E 9 8) d1› using wina_of_d1 by blast
  have A4: ‹Game.attacker_wins (E 8 9) d2› using wina_of_d2 by blast

  have ‹¬(E 9 9) ≤ (E 0 1)› by simp
  hence ‹minus_energy (E 9 9) (E 0 1) = Some (E ((one (E 9 9)) - (one (E 0 1))) ((two (E
9 9)) - (two (E 0 1))))›
    by simp
  hence A5: ‹minus_energy (E 9 9) (E 0 1) = Some (E 9 8)›
    using numeral_eq_enat one_enat_def
    by (auto, metis diff_Suc_1 eval_nat_numeral(3) idiff_enat_enat)

  have ‹(Game.weight c d1) (E 9 9) = minus_energy (E 9 9) (E 0 1)› using weight_opt.simps(5)
by simp
  hence A56: ‹(Game.weight c d1) (E 9 9) = Some (E 9 8)› using A5 by simp
  hence A6: ‹Game.attacker_wins (the ((Game.weight c d1) (E 9 9))) d1› using A3 by simp

  have ‹¬(E 9 9) ≤ (E 1 0)› by simp
  hence ‹minus_energy (E 9 9) (E 1 0) = Some (E ((one (E 9 9)) - (one (E 1 0))) ((two (E
9 9)) - (two (E 1 0))))›
    by simp
  hence A7: ‹minus_energy (E 9 9) (E 1 0) = Some (E 8 9)›
    using numeral_eq_enat one_enat_def
    by (simp, metis add_diff_cancel_right' idiff_enat_enat inc.simps(2) numeral_inc)

  have ‹(Game.weight c d2) (E 9 9) = minus_energy (E 9 9) (E 1 0)›
    using weight_opt.simps(6)by simp
  moreover hence ‹(Game.weight c d2) (E 9 9) = Some (E 8 9)›
    using A7 by simp
  moreover hence ‹Game.attacker_wins (the ((Game.weight c d2) (E 9 9))) d2›
    using A4 by simp
  ultimately show ‹Game.attacker_wins (E 9 9) c›
    using A7 Game.attacker_wins.Defense A2 A1 A6  wina_of_d1 wina_of_d2 A56 by blast
qed

lemma not_wina_of_c:
  shows ‹¬Game.attacker_wins (E 0 0) c›
proof -
  have ‹E 0 0 ≤ E 0 1› by simp
  hence ‹minus_energy (E 0 0) (E 0 1) = None› by auto
  hence no_win_a: ‹(Game.weight c d1) (E 0 0) = None› by simp
  have ‹(E 0 0) ≤ (E 1 0)› by simp
  hence ‹minus_energy (E 0 0) (E 1 0) = None› by auto
  hence no_win_b: ‹(Game.weight c d2)(E 0 0) = None› by simp
  have ‹∀g'. (c ↣ g') ⟶ (g' = d1 ∨ g' = d2)›
    by (metis defender.cases moves(9) weight_opt.simps(24,25,26,27))
  thus ‹¬Game.attacker_wins (E 0 0) c›
    using no_win_a no_win_b Game.attacker_wins.intros Game.attacker_wins.cases
    by (metis moves(5) option.distinct(1))
qed

end
```

10 Weak Spectroscopy Game

```
theory Spectroscopy_Game
  imports Energy_Games Energy LTS
begin
```

In this theory, we define the weak spectroscopy game as a locale. This game is an energy game constructed by adding stable and branching conjunctions to a delay bisimulation game that depends on a LTS. We play the weak spectroscopy game to compare the behaviour of processes and analyze which behavioural equivalences apply. The moves of a weak spectroscopy game depend on the transitions of the processes and the available energy. So in other words: If the defender wins the weak spectroscopy game starting with a certain energy, the corresponding behavioural equivalence applies.
Since we added adding stable and branching conjunctions to a delay bisimulation game, we differentiate the positions accordingly.

```
datatype ('s, 'a) spectroscopy_position =
        Attacker_Immediate (attacker_state: ‹'s›) (defender_states: ‹'s set›) |
        Attacker_Branch (attacker_state: ‹'s›) (defender_states: ‹'s set›) |
        Attacker_Clause (attacker_state: ‹'s›) (defender_state: ‹'s›) |
        Attacker_Delayed (attacker_state: ‹'s›) (defender_states: ‹'s set›) |

        Defender_Branch (attacker_state: ‹'s›) (attack_action: ‹'a›)
                        (attacker_state_succ: ‹'s›) (defender_states: ‹'s set›)
                        (defender_branch_states: ‹'s set›) |
        Defender_Conj (attacker_state: ‹'s›) (defender_states: ‹'s set›) |
        Defender_Stable_Conj (attacker_state: ‹'s›) (defender_states: ‹'s set›)

context LTS_Tau begin
```

We also define the moves of the weak spectroscopy game. Their names indicate the respective HML formulas they correspond to. This correspondence will be shown in section 11.2.

```
fun spectroscopy_moves :: ‹('s, 'a) spectroscopy_position ⇒ ('s, 'a) spectroscopy_position
⇒ energy update option› where
  delay:
    ‹spectroscopy_moves (Attacker_Immediate p Q) (Attacker_Delayed p' Q')
     = (if p' = p ∧ Q ↠S Q' then Some Some else None)› |

  procrastination:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed p' Q')
      = (if (Q' = Q ∧ p ≠ p' ∧ p ↦ τ p') then Some Some else None)› |

  observation:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q')
      = (if (∃a. p ↦a a p' ∧ Q ↦aS a Q') then (subtract 1 0 0 0 0 0 0 0)
         else None)› |

  f_or_early_conj:
    ‹spectroscopy_moves (Attacker_Immediate p Q) (Defender_Conj p' Q')
      =(if (Q≠{} ∧ Q = Q' ∧ p = p') then (subtract 0 0 0 0 1 0 0 0)
        else None)› |

  late_inst_conj:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Conj p' Q')
      = (if p = p' ∧ Q = Q' then Some Some else None)› |

  conj_answer:
    ‹spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause p' q)
      = (if p = p' ∧ q ∈ Q then (subtract 0 0 1 0 0 0 0 0) else None)› |

  pos_neg_clause:
```

```
    ‹spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed p' Q')
      = (if (p = p') then
          (if {q} ↠S Q' then Some min1_6 else None)
         else (if ({p} ↠S Q'∧ q=p')
               then Some (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7) else
None))› |

  late_stbl_conj:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Stable_Conj p' Q')
      = (if (p = p' ∧ Q' = { q ∈ Q. (∄q'. q ↦τ q')} ∧ (∄p''. p ↦τ p''))
          then Some Some else None)› |

  conj_s_answer:
    ‹spectroscopy_moves (Defender_Stable_Conj p Q) (Attacker_Clause p' q)
      = (if p = p' ∧ q ∈ Q then (subtract 0 0 0 1 0 0 0 0)
         else None)› |

  empty_stbl_conj_answer:
    ‹spectroscopy_moves (Defender_Stable_Conj p Q) (Defender_Conj p' Q')
      = (if Q = {} ∧ Q = Q' ∧ p = p' then (subtract 0 0 0 1 0 0 0 0)
         else None)› |

  br_conj:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Branch p' α p'' Q' Qα)
      = (if (p = p' ∧ Q' = Q - Qα ∧ p ↦a α p'' ∧ Qα ⊆ Q) then Some Some
         else None)› |

  br_answer:
    ‹spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Clause p'' q)
      = (if (p = p'' ∧ q ∈ Q) then (subtract 0 1 1 0 0 0 0 0) else None)› |

  br_obsv:
    ‹spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Branch p'' Q')
      = (if (p' = p'' ∧ Qα ↦aS α Q')
         then Some (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6) else None)›
|

  br_acct:
    ‹spectroscopy_moves (Attacker_Branch p Q) (Attacker_Immediate p' Q')
      = (if p = p' ∧ Q = Q' then subtract 1 0 0 0 0 0 0 0 else None)› |

  others: ‹spectroscopy_moves _ _ = None›

fun spectroscopy_defender where
  ‹spectroscopy_defender (Attacker_Immediate _ _) = False› |
  ‹spectroscopy_defender (Attacker_Branch _ _) = False› |
  ‹spectroscopy_defender (Attacker_Clause _ _) = False› |
  ‹spectroscopy_defender (Attacker_Delayed _ _) = False› |
  ‹spectroscopy_defender (Defender_Branch _ _ _ _ _) = True› |
  ‹spectroscopy_defender (Defender_Conj _ _) = True› |
  ‹spectroscopy_defender (Defender_Stable_Conj _ _) = True›

interpretation Game: energy_game ‹spectroscopy_moves› ‹spectroscopy_defender› ‹(≤)›
proof
  fix e e' ::energy
  show ‹e ≤ e' ⟹ e' ≤ e ⟹ e = e'› unfolding less_eq_energy_def
    by (smt (z3) energy.case_eq_if energy.expand nle_le)
next
  fix g g' e e' eu eu'
  assume monotonicity_assms:
    ‹spectroscopy_moves g g' ≠ None›
```

```
    ‹the (spectroscopy_moves g g') e = Some eu›
    ‹the (spectroscopy_moves g g') e' = Some eu'›
    ‹e ≤ e'›
  show ‹eu ≤ eu'›
  proof (cases g)
    case (Attacker_Immediate p Q)
    with monotonicity_assms
    show ?thesis
      by (cases g', simp_all, (smt (z3) option.distinct(1) option.sel minus_component_leq)+)
  next
    case (Attacker_Branch p Q)
    with monotonicity_assms
    show ?thesis
      by (cases g', simp_all, (smt (z3) option.distinct(1) option.sel minus_component_leq)+)
  next
    case (Attacker_Clause p q)
    hence ‹∃p' Q'. g'= (Attacker_Delayed p' Q')›
      using monotonicity_assms(1,2)
      by (metis spectroscopy_defender.cases spectroscopy_moves.simps(22,23,26,46,62,67))
    hence ‹spectroscopy_moves g g' = Some min1_6 ∨ spectroscopy_moves g g' = Some (λe. Option.bind
((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)›
      using monotonicity_assms(1,2) Attacker_Clause
      by (smt (verit, ccfv_threshold) spectroscopy_moves.simps(7))
    thus ?thesis
    proof safe
      assume ‹spectroscopy_moves g g' = Some min1_6›
      thus ?thesis
        using monotonicity_assms min.mono
        unfolding leq_components
        by (metis min_1_6_simps option.sel)
    next
      assume ‹spectroscopy_moves g g' = Some (λe. Option.bind (if ¬ E 0 0 0 0 0 0 0 1 ≤
e then None else Some (e - E 0 0 0 0 0 0 0 1)) min1_7)›
      thus ?thesis
        unfolding min_1_7_subtr_simp
        using monotonicity_assms
        by (smt (z3) enat_diff_mono energy.sel leq_components min.mono option.distinct(1)
option.sel)
    qed
  next
    case (Attacker_Delayed p Q)
    hence ‹(∃p' Q'. g'=(Attacker_Delayed p' Q')) ∨
      (∃p' Q'. g'=(Attacker_Immediate p' Q')) ∨
      (∃p' Q'. g'=(Defender_Conj p' Q')) ∨
      (∃p' Q'. g'=(Defender_Stable_Conj p' Q')) ∨
      (∃p' p'' Q' α Qα . g'= (Defender_Branch p' α p'' Q' Qα))›
      by (metis monotonicity_assms(1) spectroscopy_defender.cases spectroscopy_moves.simps(27,59))
    thus ?thesis
    proof (safe)
      fix p' Q'
      assume ‹g' = Attacker_Delayed p' Q'›
      thus ‹eu ≤ eu'›
        using Attacker_Delayed monotonicity_assms local.procrastination
        by (metis option.sel)
    next
      fix p' Q'
      assume ‹g' = Attacker_Immediate p' Q'›
      hence ‹spectroscopy_moves g g' = (subtract 1 0 0 0 0 0 0 0)›
        using Attacker_Delayed monotonicity_assms local.observation
        by (clarify, presburger)
      thus ‹eu ≤ eu'›
```

```
        by (smt (verit, best) mono_subtract monotonicity_assms option.distinct(1) option.sel)
    next
      fix p' Q'
      assume ‹g' = Defender_Conj p' Q'›
      thus ‹eu ≤ eu'›
        using Attacker_Delayed monotonicity_assms local.late_inst_conj
        by (metis option.sel)
    next
      fix p' Q'
      assume ‹g' = Defender_Stable_Conj p' Q'›
      thus ‹eu ≤ eu'›
        using Attacker_Delayed monotonicity_assms local.late_stbl_conj
        by (metis (no_types, lifting) option.sel)
    next
      fix p' p'' Q' α Qα
      assume ‹g' = Defender_Branch p' α p'' Q' Qα›
      thus ‹eu ≤ eu'›
        using Attacker_Delayed monotonicity_assms local.br_conj
        by (metis (no_types, lifting) option.sel)
    qed
  next
    case (Defender_Branch p a p' Q' Qa)
    with monotonicity_assms show ?thesis
      by (cases g', auto simp del: leq_components, unfold min_1_6_subtr_simp)
        (smt (z3) enat_diff_mono mono_subtract option.discI energy.sel leq_components min.mono
option.distinct(1) option.inject)+
  next
    case (Defender_Conj p Q)
    with monotonicity_assms show ?thesis
      by (cases g', simp_all del: leq_components)
        (smt (verit, ccfv_SIG) mono_subtract option.discI option.sel)
  next
    case (Defender_Stable_Conj x71 x72)
    with monotonicity_assms show ?thesis
      by (cases g', simp_all del: leq_components)
       (smt (verit, ccfv_SIG) mono_subtract option.discI option.sel)+
  qed
next
  fix g g' e e'
  assume defender_win_min_assms:
    ‹e ≤ e'›
    ‹spectroscopy_moves g g' ≠ None›
    ‹the (spectroscopy_moves g g') e' = None›
  thus
    ‹the (spectroscopy_moves g g') e = None›
  proof (cases g)
    case (Attacker_Immediate p Q)
    with defender_win_min_assms show ?thesis
      by (cases g', auto simp del: leq_components)
        (smt (verit, best) option.distinct(1) option.inject order.trans)+
  next
    case (Attacker_Branch p Q)
    with defender_win_min_assms show ?thesis
      by (cases g', auto)
        (smt (verit, best) option.distinct(1) option.inject order.trans)+
  next
    case (Attacker_Clause p q)
    hence ‹∃p' Q'. g'= (Attacker_Delayed p' Q')›
      using defender_win_min_assms(2)
      by (metis spectroscopy_defender.cases spectroscopy_moves.simps(21,52,58,62,67,72))
    hence ‹spectroscopy_moves g g' = Some min1_6 ∨ spectroscopy_moves g g' = Some (λe. Option.bind
```

```
((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)›
      using defender_win_min_assms(2) Attacker_Clause
      by (smt (verit, ccfv_threshold) spectroscopy_moves.simps(7))
    thus ?thesis
    proof safe
      assume ‹spectroscopy_moves g g' = Some min1_6›
      thus ‹the (spectroscopy_moves g g') e = None›
        using defender_win_min_assms min_1_6_some by fastforce
    next
      assume ‹spectroscopy_moves g g' = Some (λe. Option.bind (if ¬ E 0 0 0 0 0 0 0 1 ≤
e then None else Some (e - E 0 0 0 0 0 0 0 1)) min1_7)›
      thus ‹the (spectroscopy_moves g g') e = None›
        using defender_win_min_assms(1,3) bind.bind_lunit dual_order.trans min_1_7_some
        by (smt (verit, best) option.sel)
    qed
  next
    case (Attacker_Delayed p Q)
    hence ‹(∃p' Q'. g'=(Attacker_Delayed p' Q')) ∨
      (∃p' Q'. g'=(Attacker_Immediate p' Q')) ∨
      (∃p' Q'. g'=(Defender_Conj p' Q')) ∨
      (∃p' Q'. g'=(Defender_Stable_Conj p' Q')) ∨
      (∃p' p'' Q' α Qα . g'= (Defender_Branch p' α p'' Q' Qα))›
      by (metis defender_win_min_assms(2) spectroscopy_defender.cases spectroscopy_moves.simps(27,59))
    thus ?thesis
    proof (safe)
      fix p' Q'
      assume ‹g' = Attacker_Delayed p' Q'›
      hence False
        using Attacker_Delayed defender_win_min_assms(2,3) local.procrastination
        by (metis option.distinct(1) option.sel)
      thus ‹the (spectroscopy_moves g (Attacker_Delayed p' Q')) e = None› ..
    next
      fix p' Q'
      assume ‹g' = Attacker_Immediate p' Q'›
      moreover hence ‹spectroscopy_moves g g' = (subtract 1 0 0 0 0 0 0 0)›
        using Attacker_Delayed defender_win_min_assms(2,3) local.observation
        by (clarify, presburger)
      moreover hence ‹¬E 1 0 0 0 0 0 0 0 ≤ e'›
        using  defender_win_min_assms by force
      ultimately show  ‹the (spectroscopy_moves g (Attacker_Immediate p' Q')) e = None›
        using defender_win_min_assms(1) by force
    next
      fix p' Q'
      assume ‹g' = Defender_Conj p' Q'›
      hence False
        using Attacker_Delayed defender_win_min_assms(2,3) local.late_inst_conj
        by (metis option.distinct(1) option.sel)
      thus ‹the (spectroscopy_moves g (Defender_Conj p' Q')) e = None› ..
    next
      fix p' Q'
      assume ‹g' = Defender_Stable_Conj p' Q'›
      hence False
        using Attacker_Delayed defender_win_min_assms(2,3) local.late_stbl_conj
        by (metis (no_types, lifting) option.distinct(1) option.sel)
      thus ‹the (spectroscopy_moves g (Defender_Stable_Conj p' Q')) e = None› ..
    next
      fix p' p'' Q' α Qα
      assume ‹g' = Defender_Branch p' α p'' Q' Qα›
      hence False
        using Attacker_Delayed defender_win_min_assms(2,3) local.br_conj
        by (metis (no_types, lifting) option.distinct(1) option.sel)
```

```
      thus ‹the (spectroscopy_moves g (Defender_Branch p' α p'' Q' Qα)) e = None› ..
    qed
  next
    case (Defender_Branch p a p' Q' Qa)
    hence ‹(∃q'∈Q'. g' = Attacker_Clause p q')
      ∨ (∃Qa'. Qa ↦aS a Qa' ∧ g' = Attacker_Branch p' Qa')›
      using defender_win_min_assms by (cases g', auto) (metis not_None_eq)+
    hence ‹(spectroscopy_moves g g') = (subtract 0 1 1 0 0 0 0 0) ∨
      (spectroscopy_moves g g') = Some (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e)
min1_6)›
      using Defender_Branch option.collapse[OF defender_win_min_assms(2)]
      by (cases g', auto)
    thus ?thesis
      using defender_win_min_assms min_1_6_some
      by (smt (verit, best) bind.bind_lunit option.distinct(1) dual_order.trans option.sel)
  next
    case (Defender_Conj p Q)
    with defender_win_min_assms show ?thesis
      by (cases g', auto)
        (smt (verit, best) option.distinct(1) option.inject order.trans)+
  next
    case (Defender_Stable_Conj x71 x72)
    with defender_win_min_assms show ?thesis
      by (cases g', simp_all del: leq_components)
         (smt (verit) dual_order.trans option.discI option.sel)+
  qed
qed

end
```

Now, we are able to define the weak spectroscopy game on an arbitrary (but inhabited) LTS.

```
locale weak_spectroscopy_game =
  LTS_Tau step τ
  + energy_game ‹spectroscopy_moves› ‹spectroscopy_defender› ‹less_eq›
  for step :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› (‹_ ↦_ _› [70, 70, 70] 80) and
      τ :: 'a

end
```

11 Correctness

As in the main theorem of [1], we state in what sense winning energy levels and equivalences coincide as the theorem `spectroscopy_game_correctness`: There exists a formula φ distinguishing a process `p` from a set of processes `Q` with expressiveness price of at most `e` if and only if `e` is in the winning budget of `Attacker_Immediate p Q`.

The proof is split into three lemmas. The forward direction is given by the lemma `distinction_implies_winning_budg` combined with the upwards closure of winning budgets. To show the other direction one can construct a (strategy) formula with an appropriate price using the constructive proof of `winning_budget_implies_strategy_formula`. From lemma `strategy_formulas_distinguish` we then know that this formula actually distinguishes `p` from `Q`.

11.1 Distinction Implies Winning Budgets

```
theory Distinction_Implies_Winning_Budgets
  imports Spectroscopy_Game Expressiveness_Price
begin

context weak_spectroscopy_game
begin
```

In this section, we prove that if a formula distinguishes a process p from a set of process Q, then the price of this formula is in the attackers-winning budget. This is the same statement as that of lemma 1 in the paper [1, p. 20]. We likewise also prove it in the same manner.
First, we show that the statement holds if Q = {}. This is the case, as the attacker can move, at no cost, from the starting position, Attacker_Immediate p {}, to the defender position Defender_Conj p {}. In this position the defender is then unable to make any further moves. Hence, the attacker wins the game with any budget.

```
lemma distinction_implies_winning_budgets_empty_Q:
  assumes ‹distinguishes_from φ p {}›
  shows ‹attacker_wins (expressiveness_price φ) (Attacker_Immediate p {})›
proof-
  have is_last_move: ‹spectroscopy_moves (Defender_Conj p {}) p' = None› for p'
    by(rule spectroscopy_moves.elims, auto)
  moreover have ‹spectroscopy_defender (Defender_Conj p {})› by simp
  ultimately have conj_win: ‹attacker_wins (expressiveness_price φ) (Defender_Conj p {})›
    by (simp add: attacker_wins.Defense)

  from late_inst_conj[of p ‹{}› p ‹{}›] have next_move0:
    ‹spectroscopy_moves (Attacker_Delayed p {}) (Defender_Conj p {}) = Some Some› by force

  from delay[of p ‹{}› p ‹{}›] have next_move1:
    ‹spectroscopy_moves (Attacker_Immediate p {}) (Attacker_Delayed p {}) = Some Some› by
force

  moreover have ‹attacker (Attacker_Immediate p {})› by simp
  ultimately show ?thesis using attacker_wins.Attack[of ‹Attacker_Immediate p {}› _ ‹expressiveness_price
φ›]
    using next_move0 next_move1
    by (metis conj_win attacker_wins.Attack option.distinct(1) option.sel spectroscopy_defender.simps(4))
qed
```

Next, we show the statement for the case that Q ≠ {}. Following the proof of [1, p. 20], we do this by induction on a more complex property.

```
lemma distinction_implies_winning_budgets:
  assumes ‹distinguishes_from φ p Q›
  shows ‹attacker_wins (expressiveness_price φ) (Attacker_Immediate p Q)›
proof-
  have ‹⋀φ χ ψ.
        (∀Q p. Q ≠ {} ⟶ distinguishes_from φ p Q
               ⟶ attacker_wins (expressiveness_price φ) (Attacker_Immediate p Q))
      ∧
        ((∀p Q. Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from χ p Q ⟶ Q ↠S Q
            ⟶ attacker_wins (expr_pr_inner χ) (Attacker_Delayed p Q))
        ∧ (∀Ψ_I Ψ p Q. χ = Conj Ψ_I Ψ ⟶
            Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from χ p Q
            ⟶ attacker_wins (expr_pr_inner χ) (Defender_Conj p Q))
        ∧ (∀Ψ_I Ψ p Q. χ = StableConj Ψ_I Ψ ⟶
            Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from χ p Q ⟶ (∀q ∈ Q. ∄q'. q ↦ τ
q')
            ⟶ attacker_wins (expr_pr_inner χ) (Defender_Stable_Conj p Q))
        ∧ (∀Ψ_I Ψ α φ p Q p' Q_α. χ = BranchConj α φ Ψ_I Ψ ⟶
            hml_srbb_inner.distinguishes_from χ p Q ⟶ p ↦a α p' ⟶ p' ⊨SRBB φ ⟶
             Q_α = Q - hml_srbb_inner.model_set (Obs α φ)
            ⟶ attacker_wins (expr_pr_inner χ) (Defender_Branch p α p' (Q - Q_α) Q_α)))
      ∧
        (∀p q. hml_srbb_conj.distinguishes ψ p q
               ⟶ attacker_wins (expr_pr_conjunct ψ) (Attacker_Clause p q))›
  proof -
    fix φ χ ψ
    show ‹(∀Q p. Q ≠ {} ⟶ distinguishes_from φ p Q
```

```
                ⟶ attacker_wins (expressiveness_price φ) (Attacker_Immediate p Q))
        ∧
          ((∀p Q. Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from χ p Q ⟶ Q ↠S Q
              ⟶ attacker_wins (expr_pr_inner χ) (Attacker_Delayed p Q))
          ∧ (∀Ψ_I Ψ p Q. χ = Conj Ψ_I Ψ ⟶
              Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from χ p Q
              ⟶ attacker_wins (expr_pr_inner χ) (Defender_Conj p Q))
          ∧ (∀Ψ_I Ψ p Q. χ = StableConj Ψ_I Ψ ⟶
              Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from χ p Q ⟶ (∀q ∈ Q. ∄q'. q ↦ τ
q')
              ⟶ attacker_wins (expr_pr_inner χ) (Defender_Stable_Conj p Q))
          ∧ (∀Ψ_I Ψ α φ p Q p' Q_α. χ = BranchConj α φ Ψ_I Ψ ⟶
              hml_srbb_inner.distinguishes_from χ p Q ⟶ p ↦a α p' ⟶ p' ⊨SRBB φ ⟶
               Q_α = Q - hml_srbb_inner.model_set (Obs α φ)
              ⟶ attacker_wins (expr_pr_inner χ) (Defender_Branch p α p' (Q - Q_α) Q_α)))
        ∧
          (∀p q. hml_srbb_conj.distinguishes ψ p q
                ⟶ attacker_wins (expr_pr_conjunct ψ) (Attacker_Clause p q))›
    proof (induct rule: hml_srbb_hml_srbb_inner_hml_srbb_conjunct.induct[of _ _ _ φ χ ψ])
      case TT
      then show ?case
      proof (clarify)
        fix Q p
        assume ‹Q ≠ {}›
          and ‹distinguishes_from TT p Q›
        hence ‹∃q. q ∈ Q›
          by blast
        then obtain q where ‹q ∈ Q› by auto

        from ‹distinguishes_from TT p Q›
         and ‹q ∈ Q›
        have ‹distinguishes TT p q›
          using distinguishes_from_def by auto

        with verum_never_distinguishes
        show ‹attacker_wins (expressiveness_price TT) (Attacker_Immediate p Q)›
          by blast
      qed
    next
      case (Internal χ)
      show ?case
      proof (clarify)
        fix Q p
        assume ‹Q ≠ {}›
           and ‹distinguishes_from (Internal χ) p Q›
        then have ‹∃p'. p ↠ p' ∧ hml_srbb_inner_models p' χ›
              and ‹∀q ∈ Q. (∄q'. q ↠ q' ∧ hml_srbb_inner_models q' χ)›
          by auto
        hence ‹∀q ∈ Q. (∀q'. q ↠ q' ⟶ ¬(hml_srbb_inner_models q' χ))› by auto
        then have ‹∀q ∈ Q. (∀q'∈Q'. q ↠ q' ⟶ ¬(hml_srbb_inner_models q' χ))›
          for Q' by blast
        then have ‹Q ↠S Q' ⟶ (∀q' ∈ Q'. ¬(hml_srbb_inner_models q' χ))›
          for Q' using ‹Q ≠ {}› by blast

        define Qτ where ‹Qτ ≡ silent_reachable_set Q›
        with ‹⋀Q'. Q ↠S Q' ⟶ (∀q' ∈ Q'. ¬(hml_srbb_inner_models q' χ))›
        have ‹∀q' ∈ Qτ. ¬(hml_srbb_inner_models q' χ)›
          using sreachable_set_is_sreachable by presburger
        have ‹Qτ ↠S Qτ› unfolding Qτ_def
          by (metis silent_reachable_trans sreachable_set_is_sreachable
              silent_reachable.intros(1))
```

```
        from ‹∃p'. p ↠ p' ∧ (hml_srbb_inner_models p' χ)›
        obtain p' where ‹p ↠ p'› and ‹hml_srbb_inner_models p' χ› by auto
        from this(1) have ‹p ↠L p'› by(rule silent_reachable_impl_loopless)

        have ‹Qτ ≠ {}›
          using silent_reachable.intros(1) sreachable_set_is_sreachable Qτ_def ‹Q ≠ {}›
          by fastforce

        from ‹hml_srbb_inner_models p' χ›
         and ‹∀q' ∈ Qτ. ¬(hml_srbb_inner_models q' χ)›
        have ‹hml_srbb_inner.distinguishes_from χ p' Qτ› by simp

        with ‹Qτ ↠S Qτ› ‹Qτ ≠ {}› Internal
        have ‹attacker_wins (expr_pr_inner χ) (Attacker_Delayed p' Qτ)›
          by blast

        moreover have ‹expr_pr_inner χ = expressiveness_price (Internal χ)› by simp
        ultimately have ‹attacker_wins (expressiveness_price (Internal χ))
            (Attacker_Delayed p' Qτ)› by simp

        hence ‹attacker_wins (expressiveness_price (Internal χ)) (Attacker_Delayed p Qτ)›
        proof(induct rule: silent_reachable_loopless.induct[of ‹p› ‹p'›, OF ‹p ↠L p'›])
          case (1 p)
          thus ?case by simp
        next
          case (2 p p' p'')
          hence ‹attacker_wins (expressiveness_price (Internal χ)) (Attacker_Delayed p'
Qτ)›

            by simp
          moreover have ‹spectroscopy_moves (Attacker_Delayed p Qτ) (Attacker_Delayed p'
Qτ)

            = Some Some› using spectroscopy_moves.simps(2) ‹p ≠ p'› ‹p ↦τ p'› by auto
          moreover have ‹attacker (Attacker_Delayed p Qτ)› by simp
          ultimately show ?case using attacker_wins_Ga_with_id_step by auto
        qed
        have  ‹Q ↠S Qτ›
          using Qτ_def sreachable_set_is_sreachable by simp
        hence ‹spectroscopy_moves (Attacker_Immediate p Q) (Attacker_Delayed p Qτ) = Some
Some›
          using spectroscopy_moves.simps(1) by simp
        with ‹attacker_wins (expressiveness_price (Internal χ)) (Attacker_Delayed p Qτ)›
        show ‹attacker_wins (expressiveness_price (Internal χ)) (Attacker_Immediate p Q)›
          using attacker_wins_Ga_with_id_step
          by (metis option.discI option.sel spectroscopy_defender.simps(1))
        qed
    next
      case (ImmConj I ψs)
      show ?case
      proof (clarify)
        fix Q p
        assume ‹Q ≠ {}› and ‹distinguishes_from (ImmConj I ψs) p Q›
        from this(2) have ‹∀q∈Q. p ⊨SRBB ImmConj I ψs ∧ ¬ q ⊨SRBB ImmConj I ψs›
          unfolding distinguishes_from_def distinguishes_def by blast
        hence ‹∀q∈Q. ∃i∈I. hml_srbb_conjunct_models p (ψs i) ∧ ¬hml_srbb_conjunct_models
q (ψs i)›
          by simp
        hence ‹∀q∈Q. ∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q›
          using hml_srbb_conj.distinguishes_def by simp
        hence ‹∀q∈Q. ∃i∈I. ((ψs i) ∈ range ψs) ∧ hml_srbb_conj.distinguishes (ψs i) p
q› by blast
```

```
        hence ‹∀q∈Q. ∃i∈I. attacker_wins (expr_pr_conjunct (ψs i)) (Attacker_Clause p q)›
using ImmConj by blast
        hence a_clause_wina: ‹∀q∈Q. ∃i∈I. attacker_wins (expressiveness_price (ImmConj
I ψs) - E 0 0 1 0 1 0 0 0) (Attacker_Clause p q)›
          using expressiveness_price_ImmConj_geq_parts win_a_upwards_closure by fast
        from this ‹Q ≠ {}› have ‹I ≠ {}› by blast
        hence subtracts:
          ‹subtract_fn 0 0 1 0 1 0 0 0 (expressiveness_price (ImmConj I ψs)) = Some (expressiveness_price
(ImmConj I ψs) - E 0 0 1 0 1 0 0 0)›
          ‹subtract_fn 0 0 1 0 0 0 0 0 (expressiveness_price (ImmConj I ψs) - E 0 0 0 0
1 0 0 0) = Some (expressiveness_price (ImmConj I ψs) - E 0 0 1 0 1 0 0 0)›
          by (simp add: ‹I ≠ {}›)+
        have def_conj: ‹spectroscopy_defender (Defender_Conj p Q)› by simp
        have ‹spectroscopy_moves (Defender_Conj p Q) N ≠ None
            ⟹ N = Attacker_Clause (attacker_state N) (defender_state N)› for N
          by (metis spectroscopy_moves.simps(34,35,36,38,64,74) spectroscopy_position.exhaust_sel)
        hence move_kind: ‹spectroscopy_moves (Defender_Conj p Q) N ≠ None ⟹ ∃q∈Q. N =
Attacker_Clause p q› for N
          using conj_answer by metis
        hence update: ‹⋀g'. spectroscopy_moves (Defender_Conj p Q) g' ≠ None ⟹
          weight (Defender_Conj p Q) g' = subtract_fn 0 0 1 0 0 0 0 0›
          by fastforce
        hence move_wina: ‹⋀g'. spectroscopy_moves (Defender_Conj p Q) g' ≠ None ⟹
          (subtract_fn 0 0 1 0 0 0 0 0) (expressiveness_price (ImmConj I ψs) - E 0 0 0 0
1 0 0 0) = Some (expressiveness_price (ImmConj I ψs) - E 0 0 1 0 1 0 0 0) ∧
          attacker_wins (expressiveness_price (ImmConj I ψs) - E 0 0 1 0 1 0 0 0) g'›
          using move_kind a_clause_wina subtracts by blast
        from attacker_wins_Gd[OF def_conj] update move_wina have def_conj_wina:
          ‹attacker_wins (expressiveness_price (ImmConj I ψs) - E 0 0 0 0 1 0 0 0) (Defender_Conj
p Q)›
          by blast
        have imm_to_conj: ‹spectroscopy_moves (Attacker_Immediate p Q) (Defender_Conj p
Q) ≠ None›
          by (simp add: ‹Q ≠ {}›)
        have imm_to_conj_wgt: ‹weight (Attacker_Immediate p Q) (Defender_Conj p Q) (expressiveness_price
(ImmConj I ψs))
          = Some (expressiveness_price (ImmConj I ψs) - E 0 0 0 0 1 0 0 0)›
          using ‹Q ≠ {}› leq_components subtracts(1) by force
        from Attack[OF _ imm_to_conj imm_to_conj_wgt] def_conj_wina
        show ‹attacker_wins (expressiveness_price (ImmConj I ψs)) (Attacker_Immediate p
Q)›
          by simp
      qed
    next
      case (Obs α φ)
      have ‹∀p Q. Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from (hml_srbb_inner.Obs α φ)
p Q ⟶ Q ↠S Q
                ⟶ attacker_wins (expr_pr_inner (hml_srbb_inner.Obs α φ)) (Attacker_Delayed
p Q)›
      proof(clarify)
        fix p Q
        assume ‹Q ≠ {}› ‹hml_srbb_inner.distinguishes_from (hml_srbb_inner.Obs α φ) p Q›
‹ ∀p∈Q. ∀q. p ↠ q ⟶ q ∈ Q›
        have ‹∃p' Q'. p ↦a α p' ∧ Q ↦aS α Q' ∧ attacker_wins (expressiveness_price φ)
(Attacker_Immediate p' Q')›
        proof(cases ‹α = τ›)
          case True
          with ‹hml_srbb_inner.distinguishes_from (hml_srbb_inner.Obs α φ) p Q›
          have dist_unfold:  ‹((∃p'. p ↦τ p' ∧ p' ⊨SRBB φ) ∨ p ⊨SRBB φ)› by simp
          then obtain p' where ‹p' ⊨SRBB φ› ‹p ↦a α p'›
            unfolding True by blast
```

```
          from ‹hml_srbb_inner.distinguishes_from (hml_srbb_inner.Obs α φ) p Q› have
            ‹∀q∈Q. (¬ q ⊨SRBB φ) ∧ (∄q'. q ↦τ q' ∧ q' ⊨SRBB φ)›
            using True by auto
          hence ‹∀q∈Q. ¬q ⊨SRBB φ›
            using ‹∀p∈Q. ∀q. p ↠ q ⟶ q ∈ Q› by fastforce

          hence ‹distinguishes_from φ p' Q›
            using ‹p' ⊨SRBB φ› by auto

          with Obs have ‹attacker_wins (expressiveness_price φ) (Attacker_Immediate p' Q)›
            using ‹Q ≠ {}› by blast
          moreover have ‹Q ↦aS α Q›
            unfolding True
            using ‹∀p∈Q. ∀q. p ↠ q ⟶ q ∈ Q› silent_reachable_append_τ silent_reachable.intros(1)
by blast
          ultimately show ?thesis using ‹p ↦a α p'› by blast
        next
          case False
          with ‹hml_srbb_inner.distinguishes_from (hml_srbb_inner.Obs α φ) p Q›
          obtain p'' where ‹(p ↦α p'') ∧ (p'' ⊨SRBB φ)› by auto

          let ?Q' = ‹step_set Q α›
          from ‹hml_srbb_inner.distinguishes_from (hml_srbb_inner.Obs α φ) p Q›
          have ‹∀q∈?Q'. ¬ q ⊨SRBB φ›
            using ‹Q ≠ {}› and step_set_is_step_set
            by force
          from ‹∀q∈step_set Q α. ¬ q ⊨SRBB φ› ‹p ↦α p'' ∧ p'' ⊨SRBB φ›
          have ‹distinguishes_from φ p'' ?Q'› by simp
          hence ‹attacker_wins (expressiveness_price φ) (Attacker_Immediate p'' ?Q')›
            by (metis Obs distinction_implies_winning_budgets_empty_Q)
          moreover have ‹p ↦α p''› using ‹p ↦α p'' ∧ p'' ⊨SRBB φ› by simp
          moreover have ‹Q ↦aS α ?Q'› by (simp add: False LTS.step_set_is_step_set)
          ultimately show ?thesis by blast
        qed
        then obtain p' Q' where p'_Q': ‹p ↦a α p'› ‹Q ↦aS α Q'› and
          wina: ‹attacker_wins (expressiveness_price φ) (Attacker_Immediate p' Q')› by blast
        have attacker: ‹attacker (Attacker_Delayed p Q)› by simp
        have ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q') =
              (if (∃a. p ↦a a p' ∧ Q ↦aS a Q') then Some (subtract_fn 1 0 0 0 0 0 0 0)
else None)›
          for p Q p' Q' by simp
        from this[of p Q p' Q'] have
          ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q') =
               Some (subtract_fn 1 0 0 0 0 0 0 0)› using p'_Q' by auto
        with expr_obs_phi[of α φ] show
          ‹attacker_wins (expr_pr_inner (hml_srbb_inner.Obs α φ)) (Attacker_Delayed p Q)›
          using Attack[OF attacker _ _ wina]
          by (smt (verit, best) option.sel option.simps(3))
      qed
      then show ?case by fastforce
    next
      case (Conj I ψs)
      have main_case: ‹∀Ψ_I Ψ p Q. hml_srbb_inner.Conj I ψs = hml_srbb_inner.Conj Ψ_I
Ψ ⟶
            Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from (hml_srbb_inner.Conj I ψs) p Q
            ⟶ attacker_wins (expr_pr_inner (hml_srbb_inner.Conj I ψs)) (Defender_Conj
p Q)›
      proof clarify
        fix p Q
        assume case_assms:
```

```
    ‹Q ≠ {}›
    ‹hml_srbb_inner.distinguishes_from (hml_srbb_inner.Conj I ψs) p Q›
  hence distinctions: ‹∀q∈Q. ∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q›
    by auto
  hence inductive_wins: ‹∀q∈Q. ∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q
      ∧ attacker_wins (expr_pr_conjunct (ψs i)) (Attacker_Clause p q)›
    using Conj by blast
  define ψqs where
    ‹ψqs ≡ λq. (SOME ψ. ∃i∈I. ψ = ψs i ∧  hml_srbb_conj.distinguishes ψ p q
      ∧ attacker_wins (expr_pr_conjunct ψ) (Attacker_Clause p q))›
  with inductive_wins someI have ψqs_spec:
    ‹∀q∈Q. ∃i∈I. ψqs q = ψs i ∧ hml_srbb_conj.distinguishes (ψqs q ) p q
      ∧ attacker_wins (expr_pr_conjunct (ψqs q)) (Attacker_Clause p q)›
    by (smt (verit))
  have conjuncts_present: ‹∀q∈Q. expr_pr_conjunct (ψqs q) ∈ expr_pr_conjunct ‘ (ψqs
‘ Q)›
    using ‹Q ≠ {}› by blast
  define e' where ‹e' = E
    (Sup (modal_depth   ‘ (expr_pr_conjunct ‘ (ψqs ‘ Q))))
    (Sup (br_conj_depth   ‘ (expr_pr_conjunct ‘ (ψqs ‘ Q))))
    (Sup (conj_depth ‘ (expr_pr_conjunct ‘ (ψqs ‘ Q))))
    (Sup (st_conj_depth  ‘ (expr_pr_conjunct ‘ (ψqs ‘ Q))))
    (Sup (imm_conj_depth  ‘ (expr_pr_conjunct ‘ (ψqs ‘ Q))))
    (Sup (pos_conjuncts   ‘ (expr_pr_conjunct ‘ (ψqs ‘ Q))))
    (Sup (neg_conjuncts ‘ (expr_pr_conjunct ‘ (ψqs ‘ Q))))
    (Sup (neg_depth ‘ (expr_pr_conjunct ‘ (ψqs ‘ Q))))›
  from conjuncts_present have ‹∀q∈Q. (expr_pr_conjunct (ψqs q)) ≤ e'›
    unfolding e'_def
    by (metis SUP_upper energy.sel leq_components)
  with ψqs_spec win_a_upwards_closure
    have clause_win: ‹∀q∈Q. attacker_wins e' (Attacker_Clause p q)› by blast
  define eu' where ‹eu' = E
    (Sup (modal_depth   ‘ (expr_pr_conjunct ‘ (ψs ‘ I))))
    (Sup (br_conj_depth   ‘ (expr_pr_conjunct ‘ (ψs ‘ I))))
    (Sup (conj_depth ‘ (expr_pr_conjunct ‘ (ψs ‘ I))))
    (Sup (st_conj_depth  ‘ (expr_pr_conjunct ‘ (ψs ‘ I))))
    (Sup (imm_conj_depth  ‘ (expr_pr_conjunct ‘ (ψs ‘ I))))
    (Sup (pos_conjuncts   ‘ (expr_pr_conjunct ‘ (ψs ‘ I))))
    (Sup (neg_conjuncts ‘ (expr_pr_conjunct ‘ (ψs ‘ I))))
    (Sup (neg_depth ‘ (expr_pr_conjunct ‘ (ψs ‘ I))))›
  have subset_form: ‹ψqs ‘ Q ⊆ ψs ‘ I›
    using ψqs_spec by fastforce
  hence ‹e' ≤ eu'› unfolding e'_def eu'_def leq_components
    by (simp add: Sup_subset_mono image_mono)
  define e where ‹e = E
    (modal_depth e')
    (br_conj_depth e')
    (1 + conj_depth e')
    (st_conj_depth e')
    (imm_conj_depth e')
    (pos_conjuncts e')
    (neg_conjuncts e')
    (neg_depth e')›
  have ‹e' = e - (E 0 0 1 0 0 0 0 0)› unfolding e_def e'_def by simp
  hence ‹Some e' = (subtract_fn 0 0 1 0 0 0 0 0) e›
    by (auto, smt (verit) add_increasing2 e_def energy.sel energy_leq_cases i0_lb le_numeral_extra(4
  have expr_lower: ‹(E 0 0 1 0 0 0 0 0) ≤ expr_pr_inner (Conj I ψs)›
    using case_assms(1) subset_form by auto
  have eu'_comp: ‹eu' = (expr_pr_inner (Conj I ψs)) - (E 0 0 1 0 0 0 0 0)›
    unfolding eu'_def
    by (auto simp add: bot_enat_def image_image)
```

```
        with expr_lower have eu'_characterization: ‹Some eu' = (subtract_fn 0 0 1 0 0 0
0 0) (expr_pr_inner (Conj I ψs))›
          by presburger
        have ‹∀g'. spectroscopy_moves (Defender_Conj p Q) g' ≠ None
        ⟶ (∃q∈Q. (Attacker_Clause p q) = g') ∧ spectroscopy_moves (Defender_Conj p Q)
g' = Some (subtract_fn 0 0 1 0 0 0 0 0)›
        proof clarify
          fix g' upd
          assume upd_def: ‹spectroscopy_moves (Defender_Conj p Q) g' = Some upd›
          hence ‹⋀px q. g' = Attacker_Clause px q ⟹ p = px ∧ q ∈ Q ∧ upd = (subtract_fn
0 0 1 0 0 0 0 0)›
            by (metis (mono_tags, lifting) local.conj_answer option.sel option.simps(3))
          with upd_def show ‹(∃q∈Q. Attacker_Clause p q = g') ∧ spectroscopy_moves (Defender_Conj
p Q) g' = Some (subtract_fn 0 0 1 0 0 0 0 0)›
            by (cases g', auto)
        qed
        hence ‹∀g'. spectroscopy_moves (Defender_Conj p Q) g' ≠ None
          ⟶ (∃e'. (the (spectroscopy_moves (Defender_Conj p Q) g')) e = Some e' ∧ attacker_wins
e' g')›
          unfolding e_def
          using clause_win ‹Some e' = (subtract_fn 0 0 1 0 0 0 0 0) e› e_def by force
        hence ‹attacker_wins e (Defender_Conj p Q)›
          unfolding e_def using attacker_wins.Defense
          by auto
        moreover have ‹e ≤ expr_pr_inner (Conj I ψs)›
          using ‹e' ≤ eu'› eu'_characterization ‹Some e' = (subtract_fn 0 0 1 0 0 0 0 0)
e› expr_lower case_assms(1) subset_form
          unfolding e_def
          by (smt (verit, ccfv_threshold) eu'_comp add_diff_cancel_enat
              add_mono_thms_linordered_semiring(1) enat.simps(3) enat_defs(2) energy.sel
              expr_pr_inner.simps idiff_0_right inst_conj_depth_inner.simps(2) le_numeral_extra(4)
              leq_components minus_energy_def not_one_le_zero)
        ultimately show ‹attacker_wins (expr_pr_inner (hml_srbb_inner.Conj I ψs)) (Defender_Conj
p Q)›
          using win_a_upwards_closure by blast
      qed
      moreover have
        ‹∀p Q. Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from (hml_srbb_inner.Conj I ψs)
p Q ⟶ Q ↠S Q
            ⟶ attacker_wins (expr_pr_inner (hml_srbb_inner.Conj I ψs)) (Attacker_Delayed
p Q)›
      proof clarify
        fix p Q
        assume
          ‹Q ≠ {}›
          ‹hml_srbb_inner.distinguishes_from (hml_srbb_inner.Conj I ψs) p Q›
        hence ‹attacker_wins (expr_pr_inner (hml_srbb_inner.Conj I ψs)) (Defender_Conj p
Q)›
          using main_case by blast
        moreover have ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Conj p Q) = Some
Some›
          by auto
        ultimately show ‹attacker_wins (expr_pr_inner (hml_srbb_inner.Conj I ψs)) (Attacker_Delayed
p Q)›
          by (metis attacker_wins_Ga_with_id_step option.discI option.sel spectroscopy_defender.simps(4))
      qed
      ultimately show ?case by fastforce
    next
      case (StableConj I ψs)
      — The following proof is virtually the same as for Conj I ψs
      have main_case: ‹(∀Ψ_I Ψ p Q. StableConj I ψs = StableConj Ψ_I Ψ ⟶
```

```
              Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from (StableConj I ψs) p Q ⟶ (∀q∈Q.
∄q'. q ↦τ q')
              ⟶ attacker_wins (expr_pr_inner (StableConj I ψs)) (Defender_Stable_Conj p
Q))›
      proof clarify
        fix p Q
        assume case_assms:
          ‹Q ≠ {}›
          ‹hml_srbb_inner.distinguishes_from (StableConj I ψs) p Q›
          ‹∀q∈Q. ∄q'. q ↦τ q'›
        hence distinctions: ‹∀q∈Q. ∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q›
          by (metis hml_srbb_conj.distinguishes_def hml_srbb_inner.distinguishes_from_def
hml_srbb_inner_models.simps(3))
        hence inductive_wins: ‹∀q∈Q. ∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q
            ∧ attacker_wins (expr_pr_conjunct (ψs i)) (Attacker_Clause p q)›
          using StableConj by blast
        define ψqs where
          ‹ψqs ≡ λq. (SOME ψ. ∃i∈I. ψ = ψs i ∧  hml_srbb_conj.distinguishes ψ p q
            ∧ attacker_wins (expr_pr_conjunct ψ) (Attacker_Clause p q))›
        with inductive_wins someI have ψqs_spec:
          ‹∀q∈Q. ∃i∈I. ψqs q = ψs i ∧ hml_srbb_conj.distinguishes (ψqs q ) p q
            ∧ attacker_wins (expr_pr_conjunct (ψqs q)) (Attacker_Clause p q)›
          by (smt (verit))
        have conjuncts_present: ‹∀q∈Q. expr_pr_conjunct (ψqs q) ∈ expr_pr_conjunct ` (ψqs
` Q)›
          using ‹Q ≠ {}› by blast
        define e' where ‹e' = E
          (Sup (modal_depth   ` (expr_pr_conjunct ` (ψqs ` Q))))
          (Sup (br_conj_depth   ` (expr_pr_conjunct ` (ψqs ` Q))))
          (Sup (conj_depth ` (expr_pr_conjunct ` (ψqs ` Q))))
          (Sup (st_conj_depth  ` (expr_pr_conjunct ` (ψqs ` Q))))
          (Sup (imm_conj_depth  ` (expr_pr_conjunct ` (ψqs ` Q))))
          (Sup (pos_conjuncts   ` (expr_pr_conjunct ` (ψqs ` Q))))
          (Sup (neg_conjuncts ` (expr_pr_conjunct ` (ψqs ` Q))))
          (Sup (neg_depth ` (expr_pr_conjunct ` (ψqs ` Q))))›
        from conjuncts_present have ‹∀q∈Q. (expr_pr_conjunct (ψqs q)) ≤ e'› unfolding e'_def
          by (smt (verit, best) SUP_upper energy.sel energy.simps(3) energy_leq_cases image_iff)
        with ψqs_spec win_a_upwards_closure
          have clause_win: ‹∀q∈Q. attacker_wins e' (Attacker_Clause p q)› by blast
        define eu' where ‹eu' = E
          (Sup (modal_depth   ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (br_conj_depth   ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (conj_depth ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (st_conj_depth  ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (imm_conj_depth  ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (pos_conjuncts   ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (neg_conjuncts ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (neg_depth ` (expr_pr_conjunct ` (ψs ` I))))›
        have subset_form: ‹ψqs ` Q ⊆ ψs ` I›
          using ψqs_spec by fastforce
        hence ‹e' ≤ eu'› unfolding e'_def eu'_def
          by (simp add: Sup_subset_mono image_mono)
        define e where ‹e = E
          (modal_depth e')
          (br_conj_depth e')
          (conj_depth e')
          (1 + st_conj_depth e')
          (imm_conj_depth e')
          (pos_conjuncts e')
          (neg_conjuncts e')
          (neg_depth e')›
```

```
        have ‹e' = e - (E 0 0 0 1 0 0 0 0)› unfolding e_def e'_def by auto
        hence ‹Some e' = (subtract_fn 0 0 0 1 0 0 0 0) e›
          by (metis e_def energy.sel energy_leq_cases i0_lb le_iff_add)
        have expr_lower: ‹(E 0 0 0 1 0 0 0 0) ≤ expr_pr_inner (StableConj I ψs)›
          using case_assms(1) subset_form by force
        have eu'_comp: ‹eu' = (expr_pr_inner (StableConj I ψs)) - (E 0 0 0 1 0 0 0 0)›
          unfolding eu'_def using energy.sel
          by (auto simp add: bot_enat_def, (metis (no_types, lifting) SUP_cong image_image)+)
        with expr_lower have eu'_characterization: ‹Some eu' = (subtract_fn 0 0 0 1 0 0
0 0) (expr_pr_inner (StableConj I ψs))›
          by presburger
        have ‹∀g'. spectroscopy_moves (Defender_Stable_Conj p Q) g' ≠ None
        ⟶ (∃q∈Q. (Attacker_Clause p q) = g') ∧ spectroscopy_moves (Defender_Stable_Conj
p Q) g' = (subtract 0 0 0 1 0 0 0 0)›
        proof clarify
          fix g' upd
          assume upd_def: ‹spectroscopy_moves (Defender_Stable_Conj p Q) g' = Some upd›
          hence ‹⋀px q. g' = Attacker_Clause px q ⟹ p = px ∧ q ∈ Q ∧ upd = (subtract_fn
0 0 0 1 0 0 0 0)›
            by (metis (no_types, lifting) local.conj_s_answer option.discI option.inject)
          with upd_def case_assms(1) show
            ‹(∃q∈Q. Attacker_Clause p q = g') ∧ spectroscopy_moves (Defender_Stable_Conj
p Q) g' = (subtract 0 0 0 1 0 0 0 0)›
            by (cases g', auto)
        qed
        hence ‹∀g'. spectroscopy_moves (Defender_Stable_Conj p Q) g' ≠ None
          ⟶ (∃e'. (the (spectroscopy_moves (Defender_Stable_Conj p Q) g')) e = Some e'
∧ attacker_wins e' g')›
          unfolding e_def
          using clause_win ‹Some e' = (subtract_fn 0 0 0 1 0 0 0 0) e› e_def by force
        hence ‹attacker_wins e (Defender_Stable_Conj p Q)›
          unfolding e_def
          by (auto simp add: attacker_wins.Defense)
        moreover have ‹e ≤ expr_pr_inner (StableConj I ψs)›
          using ‹e' ≤ eu'› eu'_characterization ‹Some e' = (subtract_fn 0 0 0 1 0 0 0 0)
e› expr_lower case_assms(1) subset_form
          unfolding e_def eu'_comp minus_energy_def leq_components
          by (metis add_diff_assoc_enat add_diff_cancel_enat add_left_mono enat.simps(3)
enat_defs(2) energy.sel idiff_0_right)
       ultimately show ‹attacker_wins (expr_pr_inner (StableConj I ψs)) (Defender_Stable_Conj
p Q)›
          using win_a_upwards_closure by blast
      qed
      moreover have
        ‹(∀p Q. Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from (StableConj I ψs) p Q ⟶
Q ↠S Q
          ⟶ attacker_wins (expr_pr_inner (StableConj I ψs)) (Attacker_Delayed p Q))›
      proof clarify
        — This is where things are more complicated than in the Conj-case. (We have to differentiate
situations where the stability requirement finishes the distinction.)
        fix p Q
        assume case_assms:
          ‹Q ≠ {}›
          ‹hml_srbb_inner.distinguishes_from (StableConj I ψs) p Q›
          ‹∀q'∈Q. ∃q∈Q. q ↠ q'›
          ‹∀q∈Q. ∀q'. q ↠ q' ⟶ q' ∈ Q›
        define Q' where ‹Q' = { q ∈ Q. (∄q'. q ↦τ q')}›
        with case_assms(2) have Q'_spec: ‹hml_srbb_inner.distinguishes_from (StableConj
I ψs) p Q'› ‹∄p''. p ↦τ p''›
          unfolding hml_srbb_inner.distinguishes_from_def by auto
        hence move: ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Stable_Conj p Q')
```

```
= Some Some›
          unfolding Q'_def by auto
        show ‹attacker_wins (expr_pr_inner (StableConj I ψs)) (Attacker_Delayed p Q)›
        proof (cases ‹Q' = {}›)
          case True
          hence
            ‹spectroscopy_moves (Defender_Stable_Conj p Q') (Defender_Conj p {})
            = (subtract 0 0 0 1 0 0 0 0)› by auto
          moreover have
            ‹∀g'. spectroscopy_moves (Defender_Stable_Conj p Q') g' ≠ None ⟶ g' = (Defender_Conj
p {})›
          proof clarify
            fix g' u
            assume
              ‹spectroscopy_moves (Defender_Stable_Conj p Q') g' = Some u›
            with True show ‹g' = Defender_Conj p {}›
              by (induct g', auto, metis option.discI, metis empty_iff option.discI)
          qed
          ultimately have win_transfer:
            ‹∀e. E 0 0 0 1 0 0 0 0 ≤ e ∧ attacker_wins (e - E 0 0 0 1 0 0 0 0) (Defender_Conj
p {}) ⟶ attacker_wins e (Defender_Stable_Conj p Q')›
            using attacker_wins.Defense
            by (smt (verit, ccfv_SIG)  option.sel spectroscopy_defender.simps(7))
          have ‹∀g'. spectroscopy_moves (Defender_Conj p {}) g' = None›
          proof
            fix g'
            show ‹spectroscopy_moves (Defender_Conj p {}) g' = None› by (induct g', auto)
          qed
          hence ‹∀e. attacker_wins e (Defender_Conj p {})› using attacker_wins_Gd by fastforce
          moreover have ‹∀e. (subtract_fn 0 0 0 1 0 0 0 0) e ≠ None ⟶ e ≥ (E 0 0 0 1
0 0 0 0)›
            using minus_energy_def by presburger
          ultimately have ‹∀e. e ≥ (E 0 0 0 1 0 0 0 0) ⟶ attacker_wins e (Defender_Stable_Conj
p Q')›
            using win_transfer by presburger
          moreover have ‹expr_pr_inner (StableConj I ψs) ≥ (E 0 0 0 1 0 0 0 0)›
            by auto
          ultimately show ?thesis
            by (metis move attacker_wins_Ga_with_id_step option.discI option.sel spectroscopy_defender.si
        next
          case False
          with move show ?thesis using main_case Q'_spec attacker_wins_Ga_with_id_step unfolding
Q'_def
            by (metis (mono_tags, lifting) mem_Collect_eq option.distinct(1) option.sel spectroscopy_defe
        qed
      qed
      ultimately show ?case by blast
    next
      case (BranchConj α φ I ψs)
      have main_case:
        ‹∀p Q p' Q_α.
             hml_srbb_inner.distinguishes_from (BranchConj α φ I ψs) p Q ⟶ p ↦a α p'
⟶ p' ⊨SRBB φ ⟶
             Q_α = Q - hml_srbb_inner.model_set (Obs α φ)
             ⟶ attacker_wins (expr_pr_inner (BranchConj α φ I ψs)) (Defender_Branch p
α p' (Q - Q_α) Q_α)›
      proof ((rule allI)+, (rule impI)+)
        fix p Q p' Q_α
        assume case_assms:
          ‹hml_srbb_inner.distinguishes_from (BranchConj α φ I ψs) p Q›
          ‹p ↦a α p'›
```

```
          ‹p' ⊨SRBB φ›
          ‹Q_α = Q - hml_srbb_inner.model_set (Obs α φ)›
        from case_assms(1) have distinctions:
          ‹∀q∈(Q ∩ hml_srbb_inner.model_set (Obs α φ)).
            ∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q›
          using srbb_dist_branch_conjunction_implies_dist_conjunct_or_branch
            hml_srbb_inner.distinction_unlifting unfolding hml_srbb_inner.distinguishes_def
          by (metis Int_Collect)
        hence inductive_wins: ‹∀q∈(Q ∩ hml_srbb_inner.model_set (Obs α φ)).
          ∃i∈I. hml_srbb_conj.distinguishes (ψs i) p q
            ∧ attacker_wins (expr_pr_conjunct (ψs i)) (Attacker_Clause p q)›
          using BranchConj by blast
        define ψqs where
          ‹ψqs ≡ λq. (SOME ψ. ∃i∈I. ψ = ψs i ∧  hml_srbb_conj.distinguishes ψ p q
            ∧ attacker_wins (expr_pr_conjunct ψ) (Attacker_Clause p q))›
        with inductive_wins someI have ψqs_spec:
          ‹∀q∈(Q ∩ hml_srbb_inner.model_set (Obs α φ)).
            ∃i∈I. ψqs q = ψs i ∧ hml_srbb_conj.distinguishes (ψqs q ) p q
              ∧ attacker_wins (expr_pr_conjunct (ψqs q)) (Attacker_Clause p q)›
          by (smt (verit))
        have conjuncts_present:
          ‹∀q∈(Q ∩ hml_srbb_inner.model_set (Obs α φ)). expr_pr_conjunct (ψqs q)
            ∈ expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set (Obs α φ)))›
          by blast
        define e'0 where ‹e'0 = E
          (Sup (modal_depth   ` (expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set
(Obs α φ))))))
          (Sup (br_conj_depth   ` (expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set
(Obs α φ))))))
          (Sup (conj_depth ` (expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set (Obs
α φ))))))
          (Sup (st_conj_depth  ` (expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set
(Obs α φ))))))
          (Sup (imm_conj_depth  ` (expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set
(Obs α φ))))))
          (Sup (pos_conjuncts   ` (expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set
(Obs α φ))))))
          (Sup (neg_conjuncts ` (expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set
(Obs α φ))))))
          (Sup (neg_depth ` (expr_pr_conjunct ` (ψqs ` (Q ∩ hml_srbb_inner.model_set (Obs
α φ))))))›
        from conjuncts_present have branch_answer_bound:
            ‹∀q∈(Q ∩ hml_srbb_inner.model_set (Obs α φ)). (expr_pr_conjunct (ψqs q)) ≤
e'0›
          unfolding e'0_def using SUP_upper energy.sel energy.simps(3) energy_leq_cases image_iff
          by (smt (z3))
        with ψqs_spec win_a_upwards_closure have
          conj_wins: ‹∀q∈(Q ∩ hml_srbb_inner.model_set (Obs α φ)). attacker_wins e'0 (Attacker_Clause
p q)› by blast
        define eu'0 where ‹eu'0 = E
          (Sup (modal_depth   ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (br_conj_depth   ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (conj_depth ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (st_conj_depth  ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (imm_conj_depth  ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (pos_conjuncts   ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (neg_conjuncts ` (expr_pr_conjunct ` (ψs ` I))))
          (Sup (neg_depth ` (expr_pr_conjunct ` (ψs ` I))))›
        have subset_form: ‹ψqs ` (Q ∩ hml_srbb_inner.model_set (Obs α φ)) ⊆ ψs ` I›
          using ψqs_spec by fastforce
        hence ‹e'0 ≤ eu'0› unfolding e'0_def eu'0_def
```

```
      by (metis (mono_tags, lifting) Sup_subset_mono energy.sel energy_leq_cases image_mono)
    have no_q_way: ‹∀q∈Q_α. ∄q'. q ↦ α q' ∧ hml_srbb_models q' φ›
      using case_assms(4)
      by fastforce
    define Q' where ‹Q' ≡ (soft_step_set Q_α α)›
    hence ‹distinguishes_from φ p' Q'›
      using case_assms(2,3) no_q_way soft_step_set_is_soft_step_set mem_Collect_eq
      unfolding case_assms(4)
      by fastforce
    with BranchConj have win_a_branch:
      ‹attacker_wins (expressiveness_price φ) (Attacker_Immediate p' Q')›
      using distinction_implies_winning_budgets_empty_Q by (cases ‹Q' = {}›) auto
    have ‹expr_pr_inner (Obs α φ) ≥ (E 1 0 0 0 0 0 0 0)› by auto
    hence ‹(subtract_fn 1 0 0 0 0 0 0 0) (expr_pr_inner (Obs α φ)) = Some (expressiveness_price
φ)›
      using expr_obs_phi by auto
    with win_a_branch have win_a_step:
      ‹attacker_wins (the ((subtract_fn 1 0 0 0 0 0 0 0) (expr_pr_inner (Obs α φ))))
(Attacker_Immediate p' Q')› by auto
    define e' where ‹e' = E
      (Sup (modal_depth  ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs ` I)))))
      (Sup (br_conj_depth  ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs `
I)))))
      (Sup (conj_depth ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs ` I)))))
      (Sup (st_conj_depth  ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs `
I)))))
      (Sup (imm_conj_depth  ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs `
I)))))
      (Sup ({1 + modal_depth_srbb φ}
        ∪ (pos_conjuncts  ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs
` I))))))
      (Sup (neg_conjuncts ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs ` I)))))
      (Sup (neg_depth ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs ` I)))))›
    have ‹eu'0 ≤ e'› unfolding e'_def eu'0_def
      by (auto, meson sup.cobounded2 sup.coboundedI2)
    have ‹spectroscopy_moves (Attacker_Branch p' Q') (Attacker_Immediate p' Q') = Some
(subtract_fn 1 0 0 0 0 0 0 0)› by simp
    with win_a_step attacker_wins_Ga have obs_later_win: ‹attacker_wins (expr_pr_inner
(Obs α φ)) (Attacker_Branch p' Q')›
      by force
    hence e'_win: ‹attacker_wins e' (Attacker_Branch p' Q')›
      unfolding e'_def using win_a_upwards_closure
      by auto
    have depths: ‹1 + modal_depth_srbb φ = modal_depth (expr_pr_inner (Obs α φ))› by
simp
    have six_e': ‹pos_conjuncts e' = Sup ({1 + modal_depth_srbb φ} ∪ (pos_conjuncts
` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs ` I)))))›
      using energy.sel(6) unfolding e'_def by blast
    hence six_e'_simp: ‹pos_conjuncts e' = Sup ({1 + modal_depth_srbb φ} ∪ (pos_conjuncts
` (expr_pr_conjunct ` (ψs ` I))))›
      by (auto simp add: modal_depth_dominates_pos_conjuncts add_increasing  sup.absorb2
sup.coboundedI1)
    hence ‹pos_conjuncts e' ≤ modal_depth e'›
      unfolding e'_def
      by (auto, smt (verit) SUP_mono energy.sel(1) energy.sel(6) image_iff modal_depth_dominates_pos_c
sup.coboundedI2)
    hence ‹modal_depth (the (min1_6 e')) = (pos_conjuncts e')›
      by simp
    with six_e' have min_e'_def: ‹min1_6 e' = Some (E
      (Sup ({1 + modal_depth_srbb φ} ∪ pos_conjuncts ` (expr_pr_conjunct ` (ψs ` I))))
      (Sup (br_conj_depth  ` ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ` (ψs `
```

```
I)))))
          (Sup (conj_depth ‘ ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ‘ (ψs ‘ I)))))
          (Sup (st_conj_depth  ‘ ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ‘ (ψs ‘
I)))))
          (Sup (imm_conj_depth  ‘ ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ‘ (ψs ‘
I)))))
          (Sup ({1 + modal_depth_srbb φ} ∪ (pos_conjuncts ‘ ({expr_pr_inner (Obs α φ)} ∪
(expr_pr_conjunct ‘ (ψs ‘ I))))))
          (Sup (neg_conjuncts ‘ ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ‘ (ψs ‘ I)))))
          (Sup (neg_depth ‘ ({expr_pr_inner (Obs α φ)} ∪ (expr_pr_conjunct ‘ (ψs ‘ I))))))›
          using e'_def min1_6_def six_e'_simp
          by (smt (z3) energy.case_eq_if energy.sel min_1_6_simps(1))
        hence ‹expr_pr_inner (Obs α φ) ≤ the (min1_6 e')›
          by force
        hence obs_win: ‹attacker_wins (the (min1_6 e')) (Attacker_Branch p' Q')›
          using obs_later_win win_a_upwards_closure by blast
        define e where ‹e = E
          (modal_depth e')
          (1 + br_conj_depth e')
          (1 + conj_depth e')
          (st_conj_depth e')
          (imm_conj_depth e')
          (pos_conjuncts e')
          (neg_conjuncts e')
          (neg_depth e')›
        have ‹e' = e - (E 0 1 1 0 0 0 0 0)› unfolding e_def e'_def by auto
        hence e'_comp: ‹Some e' = (subtract_fn 0 1 1 0 0 0 0 0) e›
          by (metis e_def energy.sel energy_leq_cases i0_lb le_iff_add)
        have expr_lower: ‹(E 0 1 1 0 0 0 0 0) ≤ expr_pr_inner (BranchConj α φ I ψs)›
          using case_assms subset_form by auto
        have e'_minus: ‹e' = expr_pr_inner (BranchConj α φ I ψs) - E 0 1 1 0 0 0 0 0›
          unfolding e'_def using energy.sel
          by (auto simp add: bot_enat_def sup.left_commute,
             (metis (no_types, lifting) SUP_cong image_image)+)
        with expr_lower have e'_characterization:
            ‹Some e' = (subtract_fn 0 1 1 0 0 0 0 0) (expr_pr_inner (BranchConj α φ I ψs))›
          by presburger
        have moves: ‹∀g'. spectroscopy_moves (Defender_Branch p α p' (Q - Q_α) Q_α) g'
≠ None
        ⟶ (((Attacker_Branch p' Q' = g')
            ∧ (spectroscopy_moves (Defender_Branch p α p' (Q - Q_α) Q_α) g' = Some (λe.
Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6)))
          ∨ ((∃q∈(Q - Q_α). Attacker_Clause p q = g'
            ∧ spectroscopy_moves (Defender_Branch p α p' (Q - Q_α) Q_α) g' = (subtract 0
1 1 0 0 0 0 0))))›
        proof clarify
          fix g' u
          assume no_subtr_move:
            ‹spectroscopy_moves (Defender_Branch p α p' (Q - Q_α) Q_α) g' = Some u›
            ‹¬ (∃q∈Q - Q_α. Attacker_Clause p q = g' ∧ spectroscopy_moves (Defender_Branch
p α p' (Q - Q_α) Q_α) g' = subtract 0 1 1 0 0 0 0 0)›
          hence ‹g' = Attacker_Branch p' Q'›
            unfolding Q'_def using soft_step_set_is_soft_step_set no_subtr_move local.br_answer
            by (cases g', auto, (metis (no_types, lifting)  option.discI)+)
          moreover have ‹Attacker_Branch p' Q' = g' ⟶ spectroscopy_moves (Defender_Branch
p α p' (Q - Q_α) Q_α) g' =  Some (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6)›
            unfolding Q'_def using soft_step_set_is_soft_step_set by auto
          ultimately show ‹Attacker_Branch p' Q' = g' ∧ spectroscopy_moves (Defender_Branch
p α p' (Q - Q_α) Q_α) g' =  Some (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6)›
            by blast
        qed
```

```
        have obs_e: ‹∃e'. (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6) e =
Some e' ∧ attacker_wins e' (Attacker_Branch p' Q')›
          using obs_win e'_comp min_e'_def
          by (smt (verit, best) bind.bind_lunit min_1_6_some option.collapse)
        have ‹∀q∈(Q - Q_α). spectroscopy_moves (Defender_Branch p α p' (Q - Q_α) Q_α) (Attacker_Clause
p q) = (subtract 0 1 1 0 0 0 0 0)
          ⟶ attacker_wins e'0 (Attacker_Clause p q)›
          using conj_wins ‹eu'0 ≤ e'› case_assms(4) by blast
        with obs_e moves have move_wins: ‹∀g'. spectroscopy_moves (Defender_Branch p α p'
(Q - Q_α) Q_α) g' ≠ None
          ⟶ (∃e'. (the (spectroscopy_moves (Defender_Branch p α p' (Q - Q_α) Q_α) g'))
e = Some e' ∧ attacker_wins e' g')›
          using  ‹eu'0 ≤ e'› e'_comp ‹e'0 ≤ eu'0› win_a_upwards_closure
         by (smt (verit, ccfv_SIG) option.sel)
        moreover have ‹expr_pr_inner (BranchConj α φ I ψs) = e›
          using e'_characterization e'_minus unfolding e_def by force
        ultimately show ‹attacker_wins (expr_pr_inner (BranchConj α φ I ψs)) (Defender_Branch
p α p' (Q - Q_α) Q_α)›
        using attacker_wins.Defense spectroscopy_defender.simps(5)
          by metis
      qed
      moreover have
        ‹∀p Q. Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from (BranchConj α φ I ψs) p Q
            ⟶ attacker_wins (expr_pr_inner (BranchConj α φ I ψs)) (Attacker_Delayed p
Q)›
      proof clarify
        fix p Q
        assume case_assms:
          ‹hml_srbb_inner.distinguishes_from (BranchConj α φ I ψs) p Q›
        from case_assms(1) obtain p' where p'_spec: ‹p ↦a α p'› ‹p' ⊨SRBB φ›
          unfolding hml_srbb_inner.distinguishes_from_def
              and distinguishes_def by auto
        define Q_α where ‹Q_α = Q - hml_srbb_inner.model_set (Obs α φ)›
        have ‹attacker_wins (expr_pr_inner (BranchConj α φ I ψs)) (Defender_Branch p α
p' (Q - Q_α) Q_α)›
          using main_case case_assms(1) p'_spec Q_α_def by blast
        moreover have ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Branch p α p'
(Q - Q_α) Q_α) = Some Some›
          using p'_spec Q_α_def by auto
        ultimately show ‹attacker_wins (expr_pr_inner (BranchConj α φ I ψs)) (Attacker_Delayed
p Q)›
          using attacker_wins_Ga_with_id_step by auto
      qed
      ultimately show ?case by blast
    next
      case (Pos χ)
      show ?case
      proof clarify
        fix p q
        assume case_assms: ‹hml_srbb_conj.distinguishes (Pos χ) p q›
        then obtain p' where p'_spec: ‹p ↠ p'› ‹p' ∈ hml_srbb_inner.model_set χ›
          unfolding hml_srbb_conj.distinguishes_def by auto
        moreover have q_reach: ‹silent_reachable_set {q} ∩ hml_srbb_inner.model_set χ =
{}›
          using case_assms sreachable_set_is_sreachable
          unfolding hml_srbb_conj.distinguishes_def by force
        ultimately have distinction: ‹hml_srbb_inner.distinguishes_from χ p' (silent_reachable_set
{q})›
          unfolding hml_srbb_inner.distinguishes_from_def by auto
        have q_reach_nonempty:
            ‹silent_reachable_set {q} ≠ {}›
```

```
          ‹silent_reachable_set {q} ⟶S silent_reachable_set {q} ›
          unfolding silent_reachable_set_def
          using silent_reachable.intros(1) silent_reachable_trans by auto
        hence ‹attacker_wins (expr_pr_inner χ) (Attacker_Delayed p' (silent_reachable_set
{q}))›
          using distinction Pos by blast
        from p'_spec(1) this have ‹attacker_wins (expr_pr_inner χ) (Attacker_Delayed p (silent_reachable_
{q}))›
          by (induct, auto,
              metis attacker_wins_Ga_with_id_step local.procrastination option.distinct(1)
option.sel spectroscopy_defender.simps(4))
        moreover have ‹spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed p (silent_reachable_set
{q})) = Some min1_6›
          using q_reach_nonempty sreachable_set_is_sreachable by fastforce
        moreover have ‹the (min1_6 (expr_pr_conjunct (Pos χ))) ≥ expr_pr_inner χ›
          unfolding min1_6_def by (auto simp add: energy_leq_cases modal_depth_dominates_pos_conjuncts)
        ultimately show ‹attacker_wins (expr_pr_conjunct (Pos χ)) (Attacker_Clause p q)›
          using attacker_wins_Ga win_a_upwards_closure spectroscopy_defender.simps(3)
          by (metis (no_types, lifting) min_1_6_some option.discI option.exhaust_sel option.sel)
      qed
    next
      case (Neg χ)
      show ?case
      proof clarify
        fix p q
        assume case_assms: ‹hml_srbb_conj.distinguishes (Neg χ) p q›
        then obtain q' where q'_spec: ‹q ↠ q'› ‹q' ∈ hml_srbb_inner.model_set χ›
          unfolding hml_srbb_conj.distinguishes_def by auto
        moreover have p_reach: ‹silent_reachable_set {p} ∩ hml_srbb_inner.model_set χ =
{}›
          using case_assms sreachable_set_is_sreachable
          unfolding hml_srbb_conj.distinguishes_def by force
        ultimately have distinction: ‹hml_srbb_inner.distinguishes_from χ q' (silent_reachable_set
{p})›
          unfolding hml_srbb_inner.distinguishes_from_def by auto
        have ‹p ≠ q› using case_assms unfolding hml_srbb_conj.distinguishes_def by auto
        have p_reach_nonempty:
            ‹silent_reachable_set {p} ≠ {}›
            ‹silent_reachable_set {p} ⟶S silent_reachable_set {p}›
          unfolding silent_reachable_set_def
          using silent_reachable.intros(1) silent_reachable_trans by auto
        hence ‹attacker_wins (expr_pr_inner χ) (Attacker_Delayed q' (silent_reachable_set
{p}))›
          using distinction Neg by blast
        from q'_spec(1) this have ‹attacker_wins (expr_pr_inner χ) (Attacker_Delayed q (silent_reachable_
{p}))›
          by (induct, auto,
              metis attacker_wins_Ga_with_id_step local.procrastination option.distinct(1)
option.sel spectroscopy_defender.simps(4))
        moreover have ‹spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed q (silent_reachable_set
{p}))
            = Some (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)›
          using p_reach_nonempty sreachable_set_is_sreachable ‹p ≠ q› by fastforce
        moreover have ‹the (min1_7 (expr_pr_conjunct (Neg χ) - E 0 0 0 0 0 0 0 1)) ≥ (expr_pr_inner
χ)›
          using min1_7_def energy_leq_cases
          by (simp add: modal_depth_dominates_neg_conjuncts)
        moreover from this have ‹∃e'. Some e' = ((λe. Option.bind ((subtract_fn 0 0 0 0
0 0 0 1) e) min1_7) (expr_pr_conjunct (Neg χ))) ∧ e' ≥ (expr_pr_inner χ)›
          unfolding min_1_7_subtr_simp by auto
        ultimately show ‹attacker_wins (expr_pr_conjunct (Neg χ)) (Attacker_Clause p q)›
```

```
        using attacker_wins.Attack win_a_upwards_closure spectroscopy_defender.simps(3)
        by (metis (no_types, lifting) option.discI option.sel)
      qed
    qed
  qed
  thus ?thesis
    by (metis assms distinction_implies_winning_budgets_empty_Q)
qed

end

end
```

11.2 Strategy Formulas

```
theory Strategy_Formulas
    imports Spectroscopy_Game Expressiveness_Price
begin
```

In this section, we introduce strategy formulas as a tool of proving the corresponding lemma, `spectroscopy_game_correctness`, in section 11.3. We first define strategy formulas, creating a bridge between HML formulas, the spectroscopy game and winning budgets. We then show that for some energy `e` in a winning budget there exists a strategy formula with expressiveness price ≤ `e`. Afterwards, we prove that this formula actually distinguishes the corresponding processes.

```
context weak_spectroscopy_game
begin
```

We define strategy formulas inductively. For example for $\langle\alpha\rangle\varphi$ to be a strategy formula for some attacker delayed position `g` with energy `e` the following must hold: φ is a strategy formula at the from `g` through an observation move reached attacker (immediate) position with the energy `e` updated according to the move. Then the function `strategy_formula_inner g e` $\langle\alpha\rangle\varphi$ returns true. Similarly, every derivation rule for strategy formulas corresponds to possible moves in the spectroscopy game. To account for the three different data types a HML_{SRBB} formula can have in our formalization, we define three functions at the same time:

```
inductive
strategy_formula :: ‹('s, 'a) spectroscopy_position ⇒ energy ⇒ ('a, 's)hml_srbb ⇒ bool›
and strategy_formula_inner
  :: ‹('s, 'a) spectroscopy_position ⇒ energy ⇒ ('a, 's)hml_srbb_inner ⇒ bool›
and strategy_formula_conjunct
  :: ‹('s, 'a) spectroscopy_position ⇒ energy ⇒ ('a, 's)hml_srbb_conjunct ⇒ bool›
where
  delay:
    ‹strategy_formula (Attacker_Immediate p Q) e (Internal χ)›
      if ‹((∃Q'. (spectroscopy_moves (Attacker_Immediate p Q) (Attacker_Delayed p Q')
        = (Some Some)) ∧ (attacker_wins e (Attacker_Delayed p Q'))
          ∧ strategy_formula_inner (Attacker_Delayed p Q') e χ))› |

  procrastination:
    ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
      if ‹(∃p'. spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed p' Q)
         = (Some Some) ∧ attacker_wins e (Attacker_Delayed p' Q)
          ∧ strategy_formula_inner (Attacker_Delayed p' Q) e χ)› |

  observation:
    ‹strategy_formula_inner (Attacker_Delayed p Q) e (Obs α φ)›
      if ‹∃p' Q'. spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q')
         = (subtract 1 0 0 0 0 0 0 0)
          ∧ attacker_wins (e - (E 1 0 0 0 0 0 0 0)) (Attacker_Immediate p' Q')
```

```
        ∧ strategy_formula (Attacker_Immediate p' Q') (e - (E 1 0 0 0 0 0 0 0)) φ
        ∧ p ↦aα p' ∧ Q ↦aS α Q'› |

  early_conj:
    ‹strategy_formula (Attacker_Immediate p Q) e φ›
      if ‹∃p'. spectroscopy_moves (Attacker_Immediate p Q) (Defender_Conj p' Q')
              = (subtract 0 0 0 0 1 0 0 0)
                ∧ attacker_wins (e - (E 0 0 0 0 1 0 0 0)) (Defender_Conj p' Q')
                ∧ strategy_formula (Defender_Conj p' Q') (e - (E 0 0 0 0 1 0 0 0)) φ›
|

  late_conj:
    ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
      if ‹(spectroscopy_moves (Attacker_Delayed p Q) (Defender_Conj p Q)
         = (Some Some) ∧ (attacker_wins e (Defender_Conj p Q))
          ∧ strategy_formula_inner (Defender_Conj p Q) e χ)› |

  conj:
  ‹strategy_formula_inner (Defender_Conj p Q) e (Conj Q Φ)›
      if ‹∀q ∈ Q. spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause p q)
          = (subtract 0 0 1 0 0 0 0 0)
            ∧ (attacker_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Clause p q))
            ∧ strategy_formula_conjunct (Attacker_Clause p q) (e - (E 0 0 1 0 0 0 0 0)) (Φ
q)› |

  imm_conj:
  ‹strategy_formula (Defender_Conj p Q) e (ImmConj Q Φ)›
      if ‹∀q ∈ Q. spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause p q)
          = (subtract 0 0 1 0 0 0 0 0)
            ∧ (attacker_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Clause p q))
            ∧ strategy_formula_conjunct (Attacker_Clause p q) (e - (E 0 0 1 0 0 0 0 0)) (Φ
q)› |

  pos:
  ‹strategy_formula_conjunct (Attacker_Clause p q) e (Pos χ)›
    if ‹(∃Q'. spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed p Q')
      = Some min1_6 ∧ attacker_wins (the (min1_6 e)) (Attacker_Delayed p Q')
        ∧ strategy_formula_inner (Attacker_Delayed p Q') (the (min1_6 e)) χ)› |

  neg:
  ‹strategy_formula_conjunct (Attacker_Clause p q) e (Neg χ)›
    if ‹∃P'. (spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed q P')
      = Some (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)
        ∧ attacker_wins (the (min1_7 (e - (E 0 0 0 0 0 0 0 1)))) (Attacker_Delayed q P'))
        ∧ strategy_formula_inner (Attacker_Delayed q P') (the (min1_7 (e - (E 0 0 0 0 0 0
0 1)))) χ› |

  stable:
  ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
    if ‹(∃Q'. spectroscopy_moves (Attacker_Delayed p Q) (Defender_Stable_Conj p Q')
      = (Some Some) ∧ attacker_wins e (Defender_Stable_Conj p Q')
        ∧ strategy_formula_inner (Defender_Stable_Conj p Q') e χ)› |

  stable_conj:
    ‹strategy_formula_inner (Defender_Stable_Conj p Q) e (StableConj Q Φ)›
      if ‹∀q ∈ Q. spectroscopy_moves (Defender_Stable_Conj p Q) (Attacker_Clause p q)
        = (subtract 0 0 0 1 0 0 0 0)
          ∧ attacker_wins (e - (E 0 0 0 1 0 0 0 0)) (Attacker_Clause p q)
          ∧ strategy_formula_conjunct (Attacker_Clause p q) (e - (E 0 0 0 1 0 0 0 0)) (Φ
q)› |
```

```
  branch:
  ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
    if ‹∃p' Q' α Qα. spectroscopy_moves (Attacker_Delayed p Q) (Defender_Branch p α p'
Q' Qα)
      = (Some Some) ∧ attacker_wins e (Defender_Branch p α p' Q' Qα)
        ∧ strategy_formula_inner (Defender_Branch p α p' Q' Qα) e χ› |

  branch_conj:
  ‹strategy_formula_inner (Defender_Branch p α p' Q Qα) e (BranchConj α φ Q Φ)›
    if ‹∃Q'. spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Branch p' Q')
          = Some (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6)
            ∧ spectroscopy_moves (Attacker_Branch p' Q') (Attacker_Immediate p' Q')
            = subtract 1 0 0 0 0 0 0 0
            ∧ (attacker_wins (the (min1_6 (e - E 0 1 1 0 0 0 0 0)) - (E 1 0 0 0 0 0 0 0))
                  (Attacker_Immediate p' Q'))
            ∧ strategy_formula (Attacker_Immediate p' Q') (the (min1_6 (e - E 0 1 1 0 0 0
0 0)) - (E 1 0 0 0 0 0 0 0)) φ›
        ‹∀q ∈ Q. spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Clause p q)
          = (subtract 0 1 1 0 0 0 0 0)
          ∧ attacker_wins (e - (E 0 1 1 0 0 0 0 0)) (Attacker_Clause p q)
          ∧ strategy_formula_conjunct (Attacker_Clause p q) (e - (E 0 1 1 0 0 0 0 0)) (Φ
q)›
```

To prove `spectroscopy_game_correctness` we need the following implication: If `e` is in the winning budget of `Attacker_Immediate p Q`, there is a strategy formula φ for `Attacker_Immediate p Q` with energy `e` with expressiveness price $\leq$ `e`.

We prove a more detailed result for all possible game positions `g` by induction over the structure of winning budgets (Cases $1-3$:

1. We first show that the statement holds if `g` has no outgoing edges. This can only be the case for defender positions.
2. If `g` is an attacker position, by `e` being in the winning budget of `g`, we know there exists a successor of `g` that the attacker can move to. If by induction the property holds true for that successor, we show that it then holds for `g` as well.
3. If a defender position `g` has outgoing edges and the statement holds true for all successors, we show that the statement holds true for `g` as well.

```
lemma winning_budget_implies_strategy_formula:
  fixes g e
  assumes ‹attacker_wins e g›
  shows
    ‹case g of
        Attacker_Immediate p Q ⇒ ∃φ. strategy_formula g e φ ∧ expressiveness_price φ ≤
e
      | Attacker_Delayed p Q ⇒ ∃χ. strategy_formula_inner g e χ ∧ expr_pr_inner χ ≤ e
      | Attacker_Clause p q ⇒ ∃ψ. strategy_formula_conjunct g e ψ ∧ expr_pr_conjunct ψ
≤ e
      | Defender_Conj p Q ⇒ ∃χ. strategy_formula_inner g e χ ∧ expr_pr_inner χ ≤ e
      | Defender_Stable_Conj p Q ⇒ ∃χ. strategy_formula_inner g e χ  ∧ expr_pr_inner χ
≤ e
      | Defender_Branch p α p' Q Qa ⇒ ∃χ. strategy_formula_inner g e χ ∧ expr_pr_inner
χ ≤ e
      | Attacker_Branch p Q ⇒
            ∃φ. strategy_formula (Attacker_Immediate p Q) (e - E 1 0 0 0 0 0 0 0) φ
              ∧ expressiveness_price φ ≤ e - E 1 0 0 0 0 0 0 0›
  using assms
proof(induction rule: attacker_wins.induct)
  case (Attack g g' e e')
```

```
  then show ?case
  proof (induct g)
    case (Attacker_Immediate p Q)
    hence move: ‹
      (∃p Q. g' = Defender_Conj p Q) ⟶
        (∃φ. strategy_formula_inner g' (the (weight g g' e)) φ ∧ expr_pr_inner φ ≤ updated
g g' e) ∧
      (∃p Q. g' = Attacker_Delayed p Q) ⟶
        (∃φ. strategy_formula_inner g' (the (weight g g' e)) φ ∧ expr_pr_inner φ ≤ updated
g g' e)›
      using attacker_wins.cases
      by simp
    from move Attacker_Immediate have move_cases: ‹(∃p' Q'. g' = (Attacker_Delayed p' Q'))
∨ (∃ p' Q'. g' = (Defender_Conj p' Q'))›
      using spectroscopy_moves.simps
      by (smt (verit, del_insts) spectroscopy_defender.elims(2,3))
    show ?case using move_cases
    proof(rule disjE)
      assume ‹∃p' Q'. g' = Attacker_Delayed p' Q'›
      then obtain p' Q' where g'_att_del: ‹g' = Attacker_Delayed p' Q'› by blast
      have e_comp: ‹(the (spectroscopy_moves (Attacker_Immediate p Q) (Attacker_Delayed
p' Q')) e) = (Some e)›
        by (smt (verit, ccfv_threshold) Spectroscopy_Game.LTS_Tau.delay g'_att_del Attacker_Immediate
move option.exhaust_sel option.inject)
      have ‹p' = p›
        by (metis g'_att_del Attacker_Immediate(2) spectroscopy_moves.simps(1))
      moreover have ‹(attacker_wins e (Attacker_Delayed p Q'))›
        using ‹g' = Attacker_Delayed p' Q'› ‹p' = p› Attacker_Immediate win_a_upwards_closure
e_comp
        by simp
      ultimately have ‹(∃χ. strategy_formula_inner g' (the (weight (Attacker_Immediate p
Q) g' e)) χ ∧
        expr_pr_inner χ ≤ updated (Attacker_Immediate p Q) g' e)›
        using g'_att_del Attacker_Immediate by fastforce
      then obtain χ where ‹(strategy_formula_inner (Attacker_Delayed p Q') e χ ∧ expr_pr_inner
χ ≤ e)›
        using ‹p' = p› ‹weight (Attacker_Immediate p Q) (Attacker_Delayed p' Q') e = Some
e› g'_att_del by auto
      hence ‹((∃Q'. (spectroscopy_moves (Attacker_Immediate p Q) (Attacker_Delayed p Q')
      = (Some Some)) ∧ (attacker_wins e (Attacker_Delayed p Q'))
        ∧ strategy_formula_inner (Attacker_Delayed p Q') e χ))›
        using g'_att_del
        by (smt (verit) Spectroscopy_Game.LTS_Tau.delay ‹attacker_wins e (Attacker_Delayed
p Q')› Attacker_Immediate)
      hence ‹strategy_formula (Attacker_Immediate p Q) e (Internal χ)›
        using strategy_formula_strategy_formula_inner_strategy_formula_conjunct.delay by
blast
      moreover have ‹expressiveness_price (Internal χ) ≤ e›
        using ‹(strategy_formula_inner (Attacker_Delayed p Q') e χ ∧ expr_pr_inner χ ≤
e)›
        by auto
      ultimately show ?case by auto
    next
      assume ‹∃p' Q'. g' = Defender_Conj p' Q'›
      then obtain p' Q' where g'_def_conj: ‹g' = Defender_Conj p' Q'› by blast
      hence M: ‹spectroscopy_moves (Attacker_Immediate p Q) (Defender_Conj p' Q') = (subtract
0 0 0 0 1 0 0 0)›
        using local.f_or_early_conj Attacker_Immediate by presburger
      hence Qp': ‹Q≠{}› ‹Q = Q'› ‹p = p'›
        using Attack.hyps(2) Attacker_Immediate g'_def_conj local.f_or_early_conj by metis+
      from M have ‹updated (Attacker_Immediate p Q) (Defender_Conj p' Q') e
```

```
                = e - (E 0 0 0 0 1 0 0 0)›
        using Attack.hyps(3) g'_def_conj Attacker_Immediate
        by (smt (verit) option.distinct(1) option.sel)
      hence ‹(attacker_wins (e - (E 0 0 0 0 1 0 0 0)) (Defender_Conj p Q'))›
         using g'_def_conj Qp' Attacker_Immediate win_a_upwards_closure by force
      with g'_def_conj have IH_case: ‹∃χ. strategy_formula_inner g' (updated (Attacker_Immediate
p Q) g' e) χ ∧
        expr_pr_inner χ ≤ updated (Attacker_Immediate p Q) g' e›
        using Attacker_Immediate by auto
      hence ‹(∃χ. strategy_formula_inner (Defender_Conj p Q) (e - (E 0 0 0 0 1 0 0 0)) χ
∧ expr_pr_inner χ ≤ (e - (E 0 0 0 0 1 0 0 0)))›
        using ‹attacker_wins (e - (E 0 0 0 0 1 0 0 0)) (Defender_Conj p Q')› IH_case Qp'
          ‹the (weight (Attacker_Immediate p Q) (Defender_Conj p' Q') e) = e - E 0 0 0 0
1 0 0 0› g'_def_conj by auto
      then obtain χ where S: ‹(strategy_formula_inner (Defender_Conj p Q) (e - (E 0 0 0
0 1 0 0 0)) χ ∧ expr_pr_inner χ ≤ (e - (E 0 0 0 0 1 0 0 0)))›
        by blast
      hence ‹∃ψ. χ = Conj Q ψ›
        using strategy_formula_strategy_formula_inner_strategy_formula_conjunct.conj Qp'
g'_def_conj Attacker_Immediate unfolding Qp'
        by (smt (verit) spectroscopy_moves.simps(60,70) spectroscopy_position.distinct(33)
spectroscopy_position.inject(6) strategy_formula_inner.simps)
      then obtain ψ where ‹χ = Conj Q ψ› by auto
      hence ‹strategy_formula (Defender_Conj p Q) (e - (E 0 0 0 0 1 0 0 0)) (ImmConj Q ψ)›
        using S strategy_formula_strategy_formula_inner_strategy_formula_conjunct.conj strategy_formula_st
        by (smt (verit) Qp' g'_def_conj hml_srbb_inner.inject(2) Attacker_Immediate spectroscopy_defender
spectroscopy_moves.simps(60) spectroscopy_moves.simps(70) strategy_formula_inner.cases)
      hence SI: ‹strategy_formula (Attacker_Immediate p Q) e (ImmConj Q ψ)›
         using strategy_formula_strategy_formula_inner_strategy_formula_conjunct.delay early_conj
Qp'
         by (metis (no_types, lifting) ‹attacker_wins (e - E 0 0 0 0 1 0 0 0) (Defender_Conj
p Q')› local.f_or_early_conj)
      have ‹expr_pr_inner (Conj Q ψ) ≤ (e - (E 0 0 0 0 1 0 0 0))› using S ‹χ = Conj Q ψ›
by simp
      hence ‹expressiveness_price (ImmConj Q ψ) ≤ e› using expr_imm_conj Qp'
        by (smt (verit) M g'_def_conj Attacker_Immediate option.sel option.simps(3))
      thus ?thesis using SI by auto
    qed
  next
    case (Attacker_Branch p Q)
    hence g'_def: ‹g' = Attacker_Immediate p Q› using br_acct
      by (metis (no_types, lifting) spectroscopy_defender.elims(2,3) spectroscopy_moves.simps(17,51,57,61
    hence move: ‹spectroscopy_moves (Attacker_Branch p Q) g' = subtract 1 0 0 0 0 0 0 0›
by simp
    then obtain φ where
      ‹strategy_formula g' (updated (Attacker_Branch p Q) g' e) φ ∧
       expressiveness_price φ ≤ updated (Attacker_Branch p Q) g' e›
      using Attacker_Branch g'_def by auto
    hence ‹(strategy_formula (Attacker_Immediate p Q) (e - E 1 0 0 0 0 0 0 0) φ)
        ∧ expressiveness_price φ ≤ e - E 1 0 0 0 0 0 0 0›
      using move Attacker_Branch unfolding g'_def
      by (smt (verit, del_insts) option.distinct(1) option.sel)
    then show ?case by auto
  next
    case (Attacker_Clause p q)
    hence ‹(∃p' Q'. g' = (Attacker_Delayed p' Q'))›
      using Attack.hyps spectroscopy_moves.simps
      by (smt (verit, del_insts) spectroscopy_defender.elims(1))
    then obtain p' Q' where
      g'_att_del: ‹g' = Attacker_Delayed p' Q'› by blast
    show ?case
```

```
    proof(cases ‹p = p'›)
      case True
      hence ‹{q} ↠S Q'›
        using g'_att_del local.pos_neg_clause Attacker_Clause by presburger
      hence post_win:
        ‹(the (spectroscopy_moves (Attacker_Clause p q) g') e) = min1_6 e›
         ‹(attacker_wins (the (min1_6 e)) (Attacker_Delayed p Q'))›
        using ‹{q} ↠S Q'› Attacker_Clause win_a_upwards_closure unfolding True g'_att_del
        by auto
      then obtain χ where χ_spec:
        ‹strategy_formula_inner (Attacker_Delayed p Q') (the (min1_6 e)) χ›
        ‹expr_pr_inner χ ≤ the (min1_6 e)›
        using Attacker_Clause Attack True post_win unfolding g'_att_del
        by (smt (verit) option.sel spectroscopy_position.simps(53))
      hence
        ‹spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed p Q') = Some min1_6›
        ‹attacker_wins (the (min1_6 e)) (Attacker_Delayed p Q')›
        ‹strategy_formula_inner (Attacker_Delayed p Q') (the (min1_6 e)) χ›
        using ‹{q} ↠S Q'› local.pos_neg_clause post_win by auto
      hence ‹strategy_formula_conjunct (Attacker_Clause p q) e (Pos χ)›
        using strategy_formula_strategy_formula_inner_strategy_formula_conjunct.delay pos
        by blast
      thus ?thesis
        using χ_spec expr_pos by fastforce
      next
        case False
        hence Qp': ‹{p} ↠S Q'› ‹p' = q›
          using  local.pos_neg_clause Attacker_Clause unfolding g'_att_del
          by presburger+
        hence move: ‹spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed p' Q')
          = Some (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)›
          using False by auto
        hence win: ‹attacker_wins (the (min1_7 (e - E 0 0 0 0 0 0 0 1))) (Attacker_Delayed
p' Q')›
          using Attacker_Clause unfolding g'_att_del
          by (smt (verit) bind.bind_lunit bind.bind_lzero option.distinct(1) option.sel)
        hence ‹(∃φ. strategy_formula_inner (Attacker_Delayed p' Q') (the (min1_7 (e - E
0 0 0 0 0 0 0 1))) φ
          ∧ expr_pr_inner φ ≤ the (min1_7 (e - E 0 0 0 0 0 0 0 1)))›
          using Attack Attacker_Clause move unfolding g'_att_del
          by (smt (verit, del_insts) bind.bind_lunit bind_eq_None_conv option.discI option.sel
spectroscopy_position.simps(53))
        then obtain χ where χ_spec:
            ‹strategy_formula_inner (Attacker_Delayed p' Q') (the (min1_7 (e - E 0 0 0 0
0 0 0 1))) χ›
            ‹expr_pr_inner χ ≤ the (min1_7 (e - E 0 0 0 0 0 0 0 1))›
          by blast
        hence ‹strategy_formula_conjunct (Attacker_Clause p q) e (Neg χ)›
          using strategy_formula_strategy_formula_inner_strategy_formula_conjunct.delay
            neg Qp' win move by blast
        thus ?thesis
          using χ_spec Attacker_Clause expr_neg move
          unfolding g'_att_del
          by (smt (verit, best) bind.bind_lunit bind_eq_None_conv option.distinct(1) option.sel
spectroscopy_position.simps(52))
      qed
  next
    case (Attacker_Delayed p Q)
    then consider
      (Att_Del) ‹(∃p Q. g' = Attacker_Delayed p Q)› | (Att_Imm) ‹(∃p' Q'. g' = (Attacker_Immediate
p' Q'))› |
```

```
      (Def_Conj) ‹(∃p Q. g' = (Defender_Conj p Q))› | (Def_St_Conj) ‹(∃p Q. g' = (Defender_Stable_Conj
p Q))› |
      (Def_Branch) ‹(∃p' α p'' Q' Qα. g' = (Defender_Branch p' α p'' Q' Qα))›
      by (smt (verit, ccfv_threshold) spectroscopy_defender.elims(1) spectroscopy_moves.simps(27,28))
    then show ?case
    proof (cases)
      case Att_Del
      then obtain p' Q' where
        g'_att_del: ‹g' = Attacker_Delayed p' Q'› by blast
      have Qp': ‹Q' = Q› ‹p ≠ p'› ‹p ↦ τ p'›
        using Attacker_Delayed g'_att_del Spectroscopy_Game.LTS_Tau.procrastination
        by metis+
      hence e_comp: ‹(the (spectroscopy_moves (Attacker_Delayed p Q) g') e) = Some e›
        using g'_att_del
        by simp
      hence att_win: ‹(attacker_wins e (Attacker_Delayed p' Q'))›
        using g'_att_del Qp' Attacker_Delayed attacker_wins.Defense e_comp
        by (metis option.sel)
      have ‹(updated (Attacker_Delayed p Q) g' e) = e›
        using g'_att_del Attacker_Delayed e_comp by fastforce
      then obtain χ where ‹(strategy_formula_inner (Attacker_Delayed p' Q') e χ ∧ expr_pr_inner
χ ≤ e)›
        using Attacker_Delayed g'_att_del by auto
      hence ‹∃p'. spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed p' Q) = (Some
Some)
         ∧  attacker_wins e (Attacker_Delayed p' Q)
         ∧ strategy_formula_inner (Attacker_Delayed p' Q) e χ›
        using e_comp g'_att_del Qp' local.procrastination Attack.hyps att_win
          Spectroscopy_Game.LTS_Tau.procrastination
        by metis
      hence ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
        using strategy_formula_strategy_formula_inner_strategy_formula_conjunct.procrastination
by blast
      moreover have ‹expr_pr_inner χ ≤ e›
        using ‹strategy_formula_inner (Attacker_Delayed p' Q') e χ ∧ expr_pr_inner χ ≤
e› by blast
      ultimately show ?thesis by auto
    next
      case Att_Imm
      then obtain p' Q' where
        g'_att_imm: ‹g' = Attacker_Immediate p' Q'› by blast
      hence move: ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q')
≠ None›
        using Attacker_Delayed by blast
      hence ‹∃a. p ↦a a p' ∧ Q ↦aS a Q'› unfolding spectroscopy_moves.simps(3) by presburger
      then obtain α where α_prop: ‹p ↦a α p'› ‹Q ↦aS α Q'› by blast
      moreover then have weight:
        ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q') = subtract
1 0 0 0 0 0 0 0›
        by (simp, metis)
      moreover then have update: ‹updated (Attacker_Delayed p Q) g' e = e - (E 1 0 0 0 0
0 0 0)›
        using g'_att_imm Attacker_Delayed
        by (smt (verit, del_insts) option.distinct(1) option.sel)
      moreover then obtain χ where χ_prop:
        ‹strategy_formula (Attacker_Immediate p' Q') (e - E 1 0 0 0 0 0 0 0) χ›
        ‹expressiveness_price χ ≤ e - E 1 0 0 0 0 0 0 0›
        using Attacker_Delayed g'_att_imm
        by auto
      moreover have ‹attacker_wins (e - (E 1 0 0 0 0 0 0 0)) (Attacker_Immediate p' Q')›
        using attacker_wins.Attack Attack.hyps(4) Attacker_Delayed.prems(3) calculation(4)
```

```
g'_att_imm
        by force
      ultimately have ‹strategy_formula_inner (Attacker_Delayed p Q) e (Obs α χ)›
        using local.observation[of p Q e χ α] by blast
      moreover have ‹expr_pr_inner (Obs α χ) ≤ e›
        using expr_obs χ_prop Attacker_Delayed g'_att_imm weight update
        by (smt (verit) option.sel)
      ultimately show ?thesis by auto
    next
      case Def_Conj
      then obtain p' Q' where
        g'_def_conj: ‹g' = Defender_Conj p' Q'› by blast
      hence  ‹p = p'› ‹Q = Q'›
        using local.late_inst_conj Attacker_Delayed by presburger+
      hence ‹the (spectroscopy_moves (Attacker_Delayed p Q) (Defender_Conj p' Q')) e = Some
e›
        by fastforce
      hence ‹attacker_wins e (Defender_Conj p' Q')›  ‹updated g g' e = e›
        using Attacker_Delayed Attack unfolding g'_def_conj by simp+
      then obtain χ where
        χ_prop: ‹(strategy_formula_inner (Defender_Conj p' Q') e χ ∧ expr_pr_inner χ ≤
e)›
        using Attack g'_def_conj by auto
      hence
        ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Conj p' Q') = Some Some
        ∧ attacker_wins e (Defender_Conj p' Q')
        ∧ strategy_formula_inner (Defender_Conj p' Q') e χ›
        by (simp add: ‹Q = Q'› ‹attacker_wins e (Defender_Conj p' Q')› ‹p = p'›)
      then show ?thesis
        using χ_prop ‹Q = Q'› ‹attacker_wins e (Defender_Conj p' Q')› ‹p = p'› late_conj
        by fastforce
    next
      case Def_St_Conj
      then obtain p' Q' where g'_def: ‹g' = Defender_Stable_Conj p' Q'› by blast
      hence pQ': ‹p = p'› ‹Q' = { q ∈ Q. (∄q'. q ↦τ q')}› ‹∄p''. p ↦τ p''›
        using local.late_stbl_conj Attacker_Delayed
        by meson+
      hence ‹(the (spectroscopy_moves (Attacker_Delayed p Q) (Defender_Stable_Conj p' Q'))
e) = Some e›
        by auto
      hence ‹attacker_wins e (Defender_Stable_Conj p' Q')› ‹updated g g' e = e›
        using Attacker_Delayed Attack unfolding g'_def by force+
      then obtain χ where χ_prop:
        ‹strategy_formula_inner (Defender_Stable_Conj p' Q') e χ› ‹expr_pr_inner χ ≤ e›
        using Attack g'_def by auto
      have ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Stable_Conj p' Q') = Some
Some
        ∧ attacker_wins e (Defender_Stable_Conj p' Q')
        ∧ strategy_formula_inner (Defender_Stable_Conj p' Q') e χ›
        using Attack χ_prop ‹attacker_wins e (Defender_Stable_Conj p' Q')› local.late_stbl_conj
pQ'
        unfolding g'_def
        by force
      thus ?thesis using local.stable[of p Q e χ] pQ' χ_prop by fastforce
    next
      case Def_Branch
      then obtain p' α p'' Q' Qα where
        g'_def_br: ‹g' = Defender_Branch p' α p'' Q' Qα› by blast
      hence pQ': ‹p = p'› ‹Q' = Q - Qα› ‹p ↦a α p''› ‹Qα ⊆ Q›
        using local.br_conj Attacker_Delayed by metis+
      hence ‹the (spectroscopy_moves (Attacker_Delayed p Q) (Defender_Branch p' α p'' Q'
```

```
Qα)) e = Some e›
        by auto
      hence post: ‹attacker_wins e (Defender_Branch p' α p'' Q' Qα)› ‹updated g g' e =
e›
        using Attack option.inject Attacker_Delayed unfolding g'_def_br by auto
      then obtain χ where χ_prop:
        ‹strategy_formula_inner (Defender_Branch p' α p'' Q' Qα) e χ› ‹expr_pr_inner χ
≤ e›
        using g'_def_br Attack Attacker_Delayed
        by auto
      hence ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Branch p α p'' Q' Qα) =
Some Some
         ∧ attacker_wins e (Defender_Branch p α p'' Q' Qα)
         ∧ strategy_formula_inner (Defender_Branch p α p'' Q' Qα) e χ›
        using g'_def_br local.branch Attack post pQ' by simp
      hence ‹strategy_formula_inner (Attacker_Delayed p Q) e χ›
        using Attack Attacker_Delayed local.br_conj branch
        unfolding g'_def_br by fastforce
      thus ?thesis using χ_prop
        by fastforce
    qed
  next
    case (Defender_Branch)
    thus ?case by force
  next
    case (Defender_Conj)
    thus ?case by force
  next
    case (Defender_Stable_Conj)
    thus ?case by force
  qed
next
  case (Defense g e)
  thus ?case
  proof (induct g)
    case (Attacker_Immediate)
    thus ?case by force
  next
    case (Attacker_Branch)
    thus ?case by force
  next
    case (Attacker_Clause)
    thus ?case by force
  next
    case (Attacker_Delayed)
    thus ?case by force
  next
    case (Defender_Branch p α p' Q Qa)
    hence conjs:
      ‹∀q∈ Q. spectroscopy_moves (Defender_Branch p α p' Q Qa) (Attacker_Clause p q) = (subtract
0 1 1 0 0 0 0 0)›
      by simp
    obtain e' where e'_spec:
      ‹∀q∈Q. weight (Defender_Branch p α p' Q Qa) (Attacker_Clause p q) e = Some e'
        ∧ attacker_wins e' (Attacker_Clause p q)
        ∧ (∃ψ. strategy_formula_conjunct (Attacker_Clause p q) e' ψ ∧ expr_pr_conjunct
ψ ≤ e')›
      using conjs Defender_Branch option.distinct(1) option.sel
      by (smt (z3) spectroscopy_position.simps(52))
    hence e'_def: ‹Q ≠ {} ⟹ e' = e - E 0 1 1 0 0 0 0 0› using conjs
        by (smt (verit) all_not_in_conv option.distinct(1) option.sel)
```

```
    then obtain Φ where Φ_spec:
      ‹∀q ∈ Q. strategy_formula_conjunct (Attacker_Clause p q) e' (Φ q) ∧ expr_pr_conjunct
(Φ q) ≤ e'›
      using e'_spec by metis

    have obs: ‹spectroscopy_moves (Defender_Branch p α p' Q Qa) (Attacker_Branch p' (soft_step_set
Qa α)) =
      Some (λe. Option.bind ((subtract_fn 0 1 1 0 0 0 0 0) e) min1_6)›
      by (simp add: soft_step_set_is_soft_step_set)
    have ‹∀p Q. (Attacker_Branch p' (soft_step_set Qa α)) = (Attacker_Branch p Q) ⟶ p
= p' ∧ Q = soft_step_set Qa α› by blast
    with option.discI[OF obs] obtain e'' where
      ‹∃φ. strategy_formula (Attacker_Immediate p' (soft_step_set Qa α)) (e'' - E 1 0 0
0 0 0 0 0) φ
        ∧ expressiveness_price φ ≤ e'' - E 1 0 0 0 0 0 0 0›
      using Defense.IH option.distinct(1) option.sel
      by (smt (verit, best) Defender_Branch.prems(2) spectroscopy_position.simps(51))
    then obtain φ where
      ‹strategy_formula (Attacker_Immediate p' (soft_step_set Qa α))
        (updated (Defender_Branch p α p' Q Qa) (Attacker_Branch p' (soft_step_set Qa α))
e - E 1 0 0 0 0 0 0 0) φ›
      ‹expressiveness_price φ ≤ updated (Defender_Branch p α p' Q Qa) (Attacker_Branch
p' (soft_step_set Qa α)) e - E 1 0 0 0 0 0 0 0›
      using Defender_Branch.prems(2) option.discI[OF obs]
      by (smt (verit, best) option.sel spectroscopy_position.simps(51))
    hence obs_strat:
      ‹strategy_formula (Attacker_Immediate p' (soft_step_set Qa α)) (the (min1_6 (e - E
0 1 1 0 0 0 0 0)) - (E 1 0 0 0 0 0 0 0)) φ›
      ‹expressiveness_price φ ≤ (the (min1_6 (e - E 0 1 1 0 0 0 0 0)) - (E 1 0 0 0 0 0 0
0))›
      by (smt (verit, best) Defender_Branch.prems(2) bind.bind_lunit bind.bind_lzero obs
option.distinct(1) option.sel)+
    have  ‹spectroscopy_moves (Attacker_Branch p' (soft_step_set Qa α)) (Attacker_Immediate
p' (soft_step_set Qa α))
          = (subtract 1 0 0 0 0 0 0 0)› by simp
    obtain e'' where win_branch:
        ‹Some e'' = min1_6 (e - E 0 1 1 0 0 0 0 0)›
        ‹attacker_wins e'' (Attacker_Branch p' (soft_step_set Qa α))›
      using Defender_Branch
      by (smt (verit, ccfv_threshold) bind.bind_lunit bind_eq_None_conv obs option.discI
option.sel)
    then obtain g'' where g''_spec:
      ‹spectroscopy_moves (Attacker_Branch p' (soft_step_set Qa α)) g'' ≠ None›
      ‹attacker_wins (updated (Attacker_Branch p' (soft_step_set Qa α)) g'' (the (min1_6
(e - E 0 1 1 0 0 0 0 0)))) g''›
      using attacker_wins_GaE
      by (metis option.sel spectroscopy_defender.simps(2))
    hence move_immediate:
      ‹g'' = (Attacker_Immediate p' (soft_step_set Qa α))
      ∧ spectroscopy_moves (Attacker_Branch p' (soft_step_set Qa α)) (Attacker_Immediate
p' (soft_step_set Qa α)) = subtract 1 0 0 0 0 0 0 0›
      using br_acct
      by (metis (no_types, lifting) spectroscopy_defender.elims(2,3) spectroscopy_moves.simps(17,51,57,61
    then obtain e''' where win_immediate:
      ‹Some e''' = subtract_fn 1 0 0 0 0 0 0 0 e''›
      ‹attacker_wins e''' (Attacker_Immediate p' (soft_step_set Qa α))›
      using g''_spec win_branch attacker_wins.simps local.br_acct
      by (smt (verit) option.distinct(1) option.sel spectroscopy_defender.elims(1) spectroscopy_moves.simp
    hence strat: ‹strategy_formula_inner (Defender_Branch p α p' Q Qa) e (BranchConj α φ
Q Φ)›
      using branch_conj obs move_immediate obs_strat Φ_spec conjs e'_def e'_spec
```

```
      by (smt (verit, best) option.distinct(1) option.sel win_branch(1))

    have ‹E 1 0 0 0 0 0 0 0 ≤ e''› using win_branch g''_spec
      by (metis option.distinct(1) win_immediate(1))
    hence above_one: ‹0 < min (modal_depth e) (pos_conjuncts e)›
      using win_immediate win_branch
      by (metis energy.sel(1) energy.sel(6) gr_zeroI idiff_0_right leq_components
            min_1_6_simps(1) minus_energy_def not_one_le_zero option.sel)
    have ‹∀q ∈ Q. expr_pr_conjunct (Φ q) ≤ (e - (E 0 1 1 0 0 0 0 0))›
      using Φ_spec e'_def by blast
    moreover have ‹expressiveness_price φ ≤ the (min1_6 (e - E 0 1 1 0 0 0 0 0)) - E 1 0
0 0 0 0 0 0›
      using obs_strat(2) by blast
    moreover hence ‹modal_depth_srbb φ ≤ min (modal_depth e) (pos_conjuncts e) - 1›
      by simp
    hence ‹1 + modal_depth_srbb φ ≤ min (modal_depth e) (pos_conjuncts e)›
      by (metis above_one add.right_neutral add_diff_cancel_enat add_mono_thms_linordered_semiring(1)
enat.simps(3) enat_defs(2) ileI1 le_iff_add plus_1_eSuc(1))
    moreover hence ‹1 + modal_depth_srbb φ ≤ pos_conjuncts e› by simp
    ultimately have ‹expr_pr_inner (BranchConj α φ Q Φ) ≤ e›
      using expr_br_conj[of e e' e'' e''' φ Q Φ α] e'_def obs Defender_Branch(2) win_branch(1)
win_immediate(1)
      by (smt (verit, best) bind_eq_None_conv expr_br_conj option.distinct(1) option.sel
option.simps(3))
    then show ?case using strat by force
  next
    case (Defender_Conj p Q)
    hence moves:
      ‹∀g'. spectroscopy_moves (Defender_Conj p Q) g' ≠ None ⟶ (∃e'. weight (Defender_Conj
p Q) g' e = Some e' ∧ attacker_wins e' g')
        ∧ (∃q. g' = (Attacker_Clause p q))›
      using local.conj_answer
      by (metis (no_types, lifting) spectroscopy_defender.elims(2,3) spectroscopy_moves.simps(34,35,36,37
    show ?case
    proof (cases ‹Q = {}›)
      case True
      then obtain Φ where ‹∀q ∈ Q.
          spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause p q)  = (subtract 0 0 1
0 0 0 0 0)
          ∧ (attacker_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Clause p q))
          ∧ strategy_formula_conjunct (Attacker_Clause p q) (e - (E 0 0 1 0 0 0 0 0)) (Φ
q)›
        by (auto simp add: emptyE)
      hence Strat: ‹strategy_formula_inner (Defender_Conj p Q) e (Conj {} Φ)›
        using ‹Q = {}› conj by blast
      hence
        ‹modal_depth_srbb_inner (Conj Q Φ) = Sup ((modal_depth_srbb_conjunct ∘ Φ) ` Q)›
        ‹branch_conj_depth_inner (Conj Q Φ) = Sup ((branch_conj_depth_conjunct ∘ Φ) ` Q)›
        ‹inst_conj_depth_inner (Conj Q Φ) = 0›
        ‹st_conj_depth_inner (Conj Q Φ) = Sup ((st_conj_depth_conjunct ∘ Φ) ` Q)›
        ‹imm_conj_depth_inner (Conj Q Φ) = Sup ((imm_conj_depth_conjunct ∘ Φ) ` Q)›
        ‹max_pos_conj_depth_inner (Conj Q Φ) = Sup ((max_pos_conj_depth_conjunct ∘ Φ) `
Q)›
        ‹max_neg_conj_depth_inner (Conj Q Φ) = Sup ((max_neg_conj_depth_conjunct ∘ Φ) `
Q)›
        ‹neg_depth_inner (Conj Q Φ) = Sup ((neg_depth_conjunct ∘ Φ) ` Q)›
        using modal_depth_srbb_inner.simps(3) branch_conj_depth_inner.simps st_conj_depth_inner.simps
          inst_conj_depth_inner.simps imm_conj_depth_inner.simps max_pos_conj_depth_inner.simps
          max_neg_conj_depth_inner.simps neg_depth_inner.simps ‹Q = {}›
          by auto+
      hence
```

```
        ‹modal_depth_srbb_inner (Conj Q Φ) = 0›
        ‹branch_conj_depth_inner (Conj Q Φ) = 0›
        ‹inst_conj_depth_inner (Conj Q Φ) = 0›
        ‹st_conj_depth_inner (Conj Q Φ) = 0›
        ‹imm_conj_depth_inner (Conj Q Φ) = 0›
        ‹max_pos_conj_depth_inner (Conj Q Φ) = 0›
        ‹max_neg_conj_depth_inner (Conj Q Φ) = 0›
        ‹neg_depth_inner (Conj Q Φ) = 0›
        using ‹Q = {}› by (simp add: bot_enat_def)+
      hence ‹expr_pr_inner (Conj Q Φ) = (E 0 0 0 0 0 0 0 0)›
        using ‹Q = {}› by force
      hence price: ‹expr_pr_inner (Conj Q Φ) ≤ e›
        by auto
      with Strat price True show ?thesis by auto
    next
      case False
      hence fa_q: ‹∀q ∈ Q. ∃e'.
        Some e' = subtract_fn 0 0 1 0 0 0 0 0 e
        ∧ spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause p q) = (subtract 0 0 1
0 0 0 0 0)
        ∧ attacker_wins e' (Attacker_Clause p q)›
        using moves local.conj_answer option.distinct(1)
        by (smt (z3) option.sel)
      have q_ex_e': ‹∀q ∈ Q.  ∃e'.
           spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause p q) = subtract 0 0 1
0 0 0 0 0
        ∧ Some e' = subtract_fn 0 0 1 0 0 0 0 0 e
        ∧ attacker_wins e' (Attacker_Clause p q)
        ∧ (∃φ. strategy_formula_conjunct (Attacker_Clause p q) e' φ ∧ expr_pr_conjunct
φ ≤ e')›
      proof safe
        fix q
        assume ‹q ∈ Q›
        then obtain e' where e'_spec:
          ‹Some e' = subtract_fn 0 0 1 0 0 0 0 0 e›
          ‹spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause p q) = (subtract 0 0
1 0 0 0 0 0)›
          ‹attacker_wins e' (Attacker_Clause p q)›
          using fa_q by blast
        hence ‹weight (Defender_Conj p Q) (Attacker_Clause p q) e = Some e'›
          by simp
        then have ‹∃ψ. strategy_formula_conjunct (Attacker_Clause p q) e' ψ ∧ expr_pr_conjunct
ψ ≤ e'›
          using Defender_Conj e'_spec
          by (smt (verit, best) option.distinct(1) option.sel spectroscopy_position.simps(52))
        thus ‹∃e'. spectroscopy_moves (Defender_Conj p Q) (Attacker_Clause p q) = (subtract
0 0 1 0 0 0 0 0) ∧
              Some e' = subtract_fn 0 0 1 0 0 0 0 0 e ∧
              attacker_wins e' (Attacker_Clause p q) ∧ (∃φ. strategy_formula_conjunct (Attacker_Clause
p q) e' φ ∧ expr_pr_conjunct φ ≤ e')›
          using e'_spec by blast
      qed
      hence ‹∃Φ. ∀q ∈ Q.
        attacker_wins (e - E 0 0 1 0 0 0 0 0) (Attacker_Clause p q)
        ∧ (strategy_formula_conjunct (Attacker_Clause p q) (e - E 0 0 1 0 0 0 0 0) (Φ q)
        ∧ expr_pr_conjunct (Φ q) ≤ (e - E 0 0 1 0 0 0 0 0))›
        by (metis (no_types, opaque_lifting) option.distinct(1) option.inject)
      then obtain Φ where IH:
          ‹∀q ∈ Q. attacker_wins (e - E 0 0 1 0 0 0 0 0) (Attacker_Clause p q)
            ∧ (strategy_formula_conjunct (Attacker_Clause p q) (e - E 0 0 1 0 0 0 0 0) (Φ
q)
```

```
        ∧ expr_pr_conjunct (Φ q) ≤ (e - E 0 0 1 0 0 0 0 0))› by auto
    hence ‹strategy_formula_inner (Defender_Conj p Q) e (Conj Q Φ)›
      by (simp add: conj)
    moreover have ‹expr_pr_inner (Conj Q Φ) ≤ e›
      using IH expr_conj ‹Q ≠ {}› q_ex_e'
      by (metis (no_types, lifting) equals0I option.distinct(1))
    ultimately show ?thesis by auto
  qed
next
  case (Defender_Stable_Conj p Q)
  hence cases:
    ‹∀g'. spectroscopy_moves (Defender_Stable_Conj p Q) g' ≠ None ⟶
     (∃e'. weight (Defender_Stable_Conj p Q) g' e = Some e' ∧ attacker_wins e' g')
     ∧ ((∃p' q. g' = (Attacker_Clause p' q)) ∨ (∃p' Q'. g' = (Defender_Conj p' Q')))›
    by (metis (no_types, opaque_lifting)
        spectroscopy_defender.elims(2,3) spectroscopy_moves.simps(40,42,43,44,55))
  show ?case
  proof(cases ‹Q = {}›)
    case True
    then obtain e' where e'_spec:
      ‹weight (Defender_Stable_Conj p Q) (Defender_Conj p Q) e = Some e'›
      ‹e' = e - (E 0 0 0 1 0 0 0 0)›
      ‹attacker_wins e' (Defender_Conj p Q)›
      using cases local.empty_stbl_conj_answer
      by (smt (verit, best) option.discI option.sel)
    then obtain Φ where Φ_prop: ‹strategy_formula_inner (Defender_Conj p Q) e' (Conj Q
Φ)›
      using conj True by blast
    hence strategy: ‹strategy_formula_inner (Defender_Stable_Conj p Q) e (StableConj Q
Φ)›
      by (simp add: True stable_conj)
    have ‹E 0 0 0 1 0 0 0 0 ≤ e› using e'_spec
      using option.sel True by fastforce
    moreover have ‹expr_pr_inner (StableConj Q Φ) = E 0 0 0 1 0 0 0 0›
      using True by (simp add: bot_enat_def)
    ultimately have ‹expr_pr_inner (StableConj Q Φ) ≤ e› by simp
    with strategy show ?thesis by auto
  next
    case False
    then obtain e' where e'_spec:
      ‹e' = e - (E 0 0 0 1 0 0 0 0)›
      ‹∀q ∈ Q. weight (Defender_Stable_Conj p Q) (Attacker_Clause p q) e = Some e'
        ∧ attacker_wins e' (Attacker_Clause p q)›
      using cases local.conj_s_answer
      by (smt (verit, del_insts) option.distinct(1) option.sel)
    hence IH: ‹∀q ∈ Q. ∃ψ.
      strategy_formula_conjunct (Attacker_Clause p q) e' ψ ∧
      expr_pr_conjunct ψ ≤ e'›
      using Defender_Stable_Conj local.conj_s_answer
      by (smt (verit, best) option.distinct(1) option.inject spectroscopy_position.simps(52))
    hence ‹∃Φ. ∀q ∈ Q.
      strategy_formula_conjunct (Attacker_Clause p q) e' (Φ q) ∧
      expr_pr_conjunct (Φ q) ≤ e'›
      by meson
    then obtain Φ where Φ_prop: ‹∀q ∈ Q.
      strategy_formula_conjunct (Attacker_Clause p q) e' (Φ q)
      ∧ expr_pr_conjunct (Φ q) ≤ e'›
      by blast
    have ‹E 0 0 0 1 0 0 0 0 ≤ e›
      using e'_spec False by fastforce
    hence ‹expr_pr_inner (StableConj Q Φ) ≤ e›
```

```
      using expr_st_conj e'_spec Φ_prop False by metis
    moreover have ‹strategy_formula_inner (Defender_Stable_Conj p Q) e (StableConj Q Φ)›
      using Φ_prop e'_spec stable_conj
      unfolding e'_spec by fastforce
    ultimately show ?thesis by auto
  qed
 qed
qed
```

To prove `spectroscopy_game_correctness` we need the following implication: If φ is a strategy formula for `Attacker_Immediate p Q` with energy `e`, then φ distinguishes `p` from `Q`.

We prove a more detailed result for all possible game positions `g` by induction. Note that the case of `g` being an attacker branching position is not explicitly needed as part of the induction hypothesis but is proven as a part of case `branch_conj`. The induction relies on the inductive structure of strategy formulas.
Since our formalization differentiates immediate conjunctions and conjunctions, two `Defender_Conj` cases are necessary. Specifically, the strategy construction rule `early_conj` uses immediate conjunctions, while `late_conj` uses conjunctions.

```
lemma strategy_formulas_distinguish:
  shows ‹(strategy_formula g e φ ⟶
        (case g of
        Attacker_Immediate p Q ⇒  distinguishes_from φ p Q
      | Defender_Conj p Q ⇒ distinguishes_from φ p Q
      | _ ⇒ True))
      ∧
      (strategy_formula_inner g e χ ⟶
        (case g of
       Attacker_Delayed p Q ⇒ (Q ↠S Q) ⟶ distinguishes_from (Internal χ) p Q
      | Defender_Conj p Q ⇒ hml_srbb_inner.distinguishes_from χ p Q
      | Defender_Stable_Conj p Q ⇒ (∀q. ¬ p ↦ τ q)
          ⟶ hml_srbb_inner.distinguishes_from χ p Q
      | Defender_Branch p α p' Q Qa ⇒(p ↦a α p')
          ⟶ hml_srbb_inner.distinguishes_from χ p (Q∪Qa)
      | _ ⇒ True))
      ∧
      (strategy_formula_conjunct g e ψ ⟶
        (case g of
        Attacker_Clause p q ⇒ hml_srbb_conj.distinguishes ψ p q
      | _ ⇒ True))›
proof(induction rule: strategy_formula_strategy_formula_inner_strategy_formula_conjunct.induct)
  case (delay p Q e χ)
  then show ?case
    by (smt (verit) distinguishes_from_def option.discI silent_reachable.intros(1) silent_reachable_trans
spectroscopy_moves.simps(1) spectroscopy_position.simps(50) spectroscopy_position.simps(53))
next
  case (procrastination p Q e χ)
  from this obtain p' where IH: ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed
p' Q) = Some Some ∧
       attacker_wins e (Attacker_Delayed p' Q) ∧
       strategy_formula_inner (Attacker_Delayed p' Q) e χ ∧
       (case Attacker_Delayed p' Q of Attacker_Delayed p Q ⇒ Q ↠S Q ⟶ distinguishes_from
(hml_srbb.Internal χ) p Q
        | Defender_Branch p α p' Q Qa ⇒ p ↦α p' ∧ Qa ≠ {} ⟶ hml_srbb_inner.distinguishes_from
χ p (Q ∪ Qa)
        | Defender_Conj p Q ⇒ hml_srbb_inner.distinguishes_from χ p Q
        | Defender_Stable_Conj p Q ⇒ (∀q. ¬ p ↦τ q) ⟶ hml_srbb_inner.distinguishes_from
χ p Q | _ ⇒ True)› by fastforce
  hence D: ‹Q ↠S Q ⟶ distinguishes_from (hml_srbb.Internal χ) p' Q›
    using spectroscopy_position.simps(53) by fastforce
```

```
  from IH have ‹p ↠p'›
    by (metis option.discI silent_reachable.intros(1) silent_reachable_append_τ spectroscopy_moves.simps(
  hence ‹Q ↠S Q ⟶ distinguishes_from (hml_srbb.Internal χ) p Q› using D
    by (smt (verit) LTS_Tau.silent_reachable_trans distinguishes_from_def hml_srbb_models.simps(2))
  then show ?case by simp
next
  case (observation p Q e φ α)
  then obtain p' Q' where IH: ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate
p' Q') = subtract 1 0 0 0 0 0 0 0 ∧
     attacker_wins (e - E 1 0 0 0 0 0 0 0) (Attacker_Immediate p' Q') ∧
     (strategy_formula (Attacker_Immediate p' Q') (e - E 1 0 0 0 0 0 0 0) φ ∧
      (case Attacker_Immediate p' Q' of Attacker_Immediate p Q ⇒ distinguishes_from φ p
Q
       | Defender_Conj p Q ⇒ distinguishes_from φ p Q | _ ⇒ True)) ∧
     p ↦a α p' ∧ Q ↦aS α Q'› by auto
  hence D: ‹distinguishes_from φ p' Q'› by auto
  hence ‹p' ⊨SRBB φ› by auto

  have observation: ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q')
      = (if (∃a. p ↦a a p' ∧ Q ↦aS a Q') then (subtract 1 0 0 0 0 0 0 0) else None)›
    for p p' Q Q' by simp
  from IH have ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Immediate p' Q')
    = subtract 1 0 0 0 0 0 0 0› by simp
  also have ‹... ≠ None› by blast
  finally have ‹(∃a. p ↦a a p' ∧ Q ↦aS a Q')› unfolding observation by metis

  from IH have ‹p ↦a α p'› and ‹Q ↦aS α Q'›  by auto
  hence P: ‹p ⊨SRBB (Internal (Obs α φ))› using ‹p' ⊨SRBB φ›
    using silent_reachable.intros(1) by auto
  have ‹Q ↠S Q ⟶ (∀q∈Q. ¬(q ⊨SRBB (Internal (Obs α φ))))›
  proof (rule+)
    fix q
    assume
      ‹Q ↠S Q›
      ‹q ∈ Q›
      ‹q ⊨SRBB Internal (Obs α φ)›
    hence ‹∃q'.  q ↠ q' ∧ hml_srbb_inner_models q' (Obs α φ)› by simp
    then obtain q' where X: ‹q ↠ q' ∧ hml_srbb_inner_models q' (Obs α φ)› by auto
    hence ‹hml_srbb_inner_models q' (Obs α φ)› by simp

    from X have ‹q'∈Q› using ‹Q ↠S Q› ‹q ∈ Q› by blast

    hence ‹∃q''∈Q'. q' ↦a α q'' ∧ q'' ⊨SRBB φ›
      using ‹Q ↦aS α Q'› ‹hml_srbb_inner_models q' (Obs α φ)› by auto
    then obtain q'' where ‹q''∈Q'∧ q' ↦a α q'' ∧ q'' ⊨SRBB φ› by auto
    thus ‹False› using D by auto
  qed
  hence ‹Q ↠S Q ⟶ distinguishes_from (hml_srbb.Internal (hml_srbb_inner.Obs α φ)) p
Q›
    using P by fastforce
  then show ?case by simp
next
  case (early_conj Q p Q' e φ)
  then show ?case
    by (simp, metis not_None_eq)
next
  case (late_conj p Q e χ)
  then show ?case
    using silent_reachable.intros(1)
    by auto
next
```

```
  case (conj Q p e Φ)
  then show ?case by auto
next
  case (imm_conj Q p e Φ)
  then show ?case by auto
next
  case (pos p q e χ)
  then show ?case using silent_reachable.refl
    by (simp) (metis option.discI silent_reachable_trans)
next
  case (neg p q e χ)
  then obtain P' where IH:
    ‹spectroscopy_moves (Attacker_Clause p q) (Attacker_Delayed q P') =  Some (λe. Option.bind
(subtract_fn 0 0 0 0 0 0 0 1 e) min1_7)›
    ‹attacker_wins (the (min1_7 (e - E 0 0 0 0 0 0 0 1))) (Attacker_Delayed q P') ∧
      strategy_formula_inner (Attacker_Delayed q P') (the (min1_7 (e - E 0 0 0 0 0 0 0 1)))
χ ∧
      (case Attacker_Delayed q P' of Attacker_Delayed p Q ⇒ Q ↠S Q ⟶ distinguishes_from
(hml_srbb.Internal χ) p Q
         | Defender_Branch p α p' Q Qa ⇒ p ↦α p' ∧ Qa ≠ {} ⟶ hml_srbb_inner.distinguishes_from
χ p (Q ∪ Qa)
         | Defender_Conj p Q ⇒ hml_srbb_inner.distinguishes_from χ p Q
         | Defender_Stable_Conj p Q ⇒ (∀q. ¬ p ↦τ q) ⟶ hml_srbb_inner.distinguishes_from
χ p Q | _ ⇒ True)› by fastforce
  hence D: ‹P' ↠S P' ⟶ distinguishes_from (hml_srbb.Internal χ) q P'› by simp
  have ‹{p} ↠S P'› using IH(1) spectroscopy_moves.simps
    by (metis (no_types, lifting) not_Some_eq)
  have ‹P' ↠S P' ⟶ p ∈ P'› using ‹{p} ↠S P'›  by (simp add: silent_reachable.intros(1))
  hence ‹hml_srbb_conj.distinguishes (hml_srbb_conjunct.Neg χ) p q› using D ‹{p} ↠S P'›
    unfolding hml_srbb_conj.distinguishes_def distinguishes_from_def
    by (smt (verit) LTS_Tau.silent_reachable_trans hml_srbb_conjunct_models.simps(2) hml_srbb_models.simps
silent_reachable.refl)
  then show ?case by simp
next
  case (stable p Q e χ)
  then obtain Q' where IH: ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Stable_Conj
p Q') = Some Some›
      ‹attacker_wins e (Defender_Stable_Conj p Q') ∧
      strategy_formula_inner (Defender_Stable_Conj p Q') e χ ∧
      (case Defender_Stable_Conj p Q' of Attacker_Delayed p Q ⇒ Q ↠S Q ⟶ distinguishes_from
(hml_srbb.Internal χ) p Q
         | Defender_Branch p α p' Q Qa ⇒ p ↦α p' ∧ Qa ≠ {} ⟶ hml_srbb_inner.distinguishes_from
χ p (Q ∪ Qa)
         | Defender_Conj p Q ⇒ hml_srbb_inner.distinguishes_from χ p Q
         | Defender_Stable_Conj p Q ⇒ (∀q. ¬ p ↦τ q) ⟶ hml_srbb_inner.distinguishes_from
χ p Q | _ ⇒ True)› by auto
  hence ‹(∄p''. p ↦τ p'')›
    by (metis local.late_stbl_conj option.distinct(1))

  from IH have ‹(∀q. ¬ p ↦τ q) ⟶ hml_srbb_inner.distinguishes_from χ p Q'› by simp
  hence ‹hml_srbb_inner.distinguishes_from χ p Q'› using ‹∄p''. p ↦τ p''› by auto
  hence ‹hml_srbb_inner_models p χ› by simp
  hence ‹p ⊨SRBB (hml_srbb.Internal χ)›
    using LTS_Tau.refl by force
  have ‹Q ↠S Q ⟶ distinguishes_from (hml_srbb.Internal χ) p Q›
  proof
    assume ‹Q ↠S Q›
    have ‹(∀q ∈ Q. ¬(q ⊨SRBB (hml_srbb.Internal χ)))›
    proof (clarify)
      fix q
      assume ‹q ∈ Q› ‹(q ⊨SRBB (hml_srbb.Internal χ))›
```

```
      hence ‹∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ› by simp
      then obtain q' where X: ‹q ↠ q' ∧ hml_srbb_inner_models q' χ› by auto
      hence ‹q' ∈ Q› using ‹Q ↠S Q› ‹q ∈ Q› by blast
      then show ‹False›
      proof (cases ‹q' ∈ Q'›)
        case True
        thus ‹False› using X ‹hml_srbb_inner.distinguishes_from χ p Q'›
          by simp
      next
        case False
        from IH have ‹strategy_formula_inner (Defender_Stable_Conj p Q') e χ› by simp
        hence ‹∃Φ. χ=(StableConj Q' Φ)› using strategy_formula_inner.simps
          by (smt (verit) spectroscopy_position.distinct(35) spectroscopy_position.distinct(39)
spectroscopy_position.distinct(41) spectroscopy_position.inject(7))
        then obtain Φ where P: ‹χ=(StableConj Q' Φ)› by auto
        from IH(1) have ‹Q' = { q ∈ Q. (∄q'. q ↦τ q')}›
          by (metis (full_types) local.late_stbl_conj option.distinct(1))
        hence ‹∃q''. q' ↦τ q''› using False ‹q' ∈ Q› by simp
        from X have ‹hml_srbb_inner_models q' (StableConj Q' Φ)› using P by auto
        then show ?thesis using ‹∃q''. q' ↦τ q''› by simp
      qed
    qed
    thus ‹distinguishes_from (hml_srbb.Internal χ) p Q›
      using ‹p ⊨SRBB (hml_srbb.Internal χ)› by simp
  qed
  then show ?case by simp
next
  case (stable_conj Q p e Φ)
  hence IH: ‹∀q∈ Q. hml_srbb_conj.distinguishes (Φ q) p q› by simp
  hence Q: ‹∀q ∈ Q. hml_srbb_conjunct_models p (Φ q)› by simp
  hence ‹(∀q. ¬ p ↦τ q) ⟶ hml_srbb_inner.distinguishes_from (StableConj Q Φ) p Q›
    using IH by auto
  then show ?case by simp
next
  case (branch p Q e χ)
  then obtain p' Q' α Qα where IH:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Defender_Branch p α p' Q' Qα) = Some Some›
    ‹attacker_wins e (Defender_Branch p α p' Q' Qα) ∧
     strategy_formula_inner (Defender_Branch p α p' Q' Qα) e χ ∧
     (case Defender_Branch p α p' Q' Qα of Attacker_Delayed p Q ⇒ Q ↠S Q ⟶ distinguishes_from
(Internal χ) p Q
      | Defender_Branch p α p' Q Qa ⇒ p ↦a α p' ⟶ hml_srbb_inner.distinguishes_from
χ p (Q ∪ Qa)
      | Defender_Conj p Q ⇒ hml_srbb_inner.distinguishes_from χ p Q
      | Defender_Stable_Conj p Q ⇒ (∀q. ¬ p ↦τ q) ⟶ hml_srbb_inner.distinguishes_from
χ p Q | _ ⇒ True)› by blast
  from IH(1) have ‹p ↦a α p'›
    by (metis local.br_conj option.distinct(1))
  from IH have ‹p ↦a α p' ⟶ hml_srbb_inner.distinguishes_from χ p (Q' ∪ Qα)› by simp
  hence D: ‹hml_srbb_inner.distinguishes_from χ p (Q' ∪ Qα)› using ‹p ↦a α p'› by auto
  from IH have ‹Q' = Q - Qα ∧ p ↦a α p' ∧ Qα ⊆ Q›
    by (metis (no_types, lifting) br_conj option.discI)
  hence ‹Q=(Q' ∪ Qα)› by auto
  then show ?case
    using D silent_reachable.refl by auto
next
  case (branch_conj p α p' Q1 Qα e ψ Φ)
  hence A1: ‹∀q∈Q1. hml_srbb_conjunct_models p (Φ q)› by simp
  from branch_conj obtain Q' where IH:
    ‹spectroscopy_moves (Defender_Branch p α p' Q1 Qα) (Attacker_Branch p' Q')
      = Some (λe. Option.bind (subtract_fn 0 1 1 0 0 0 0 0 e) min1_6)›
```

```
    ‹spectroscopy_moves (Attacker_Branch p' Q') (Attacker_Immediate p' Q') = subtract 1
0 0 0 0 0 0 0 ∧
     attacker_wins (the (min1_6 (e - E 0 1 1 0 0 0 0 0)) - E 1 0 0 0 0 0 0 0) (Attacker_Immediate
p' Q') ∧
     strategy_formula (Attacker_Immediate p' Q') (the (min1_6 (e - E 0 1 1 0 0 0 0 0)) -
E 1 0 0 0 0 0 0 0) ψ ∧
     (case Attacker_Immediate p' Q' of Attacker_Immediate p Q ⇒ distinguishes_from ψ p Q
          | Defender_Conj p Q ⇒ distinguishes_from ψ p Q | _ ⇒ True)› by auto
  hence ‹distinguishes_from ψ p' Q'› by simp
  hence X: ‹p' ⊨SRBB ψ› by simp
  have Y: ‹∀q ∈ Q'. ¬(q ⊨SRBB ψ)› using ‹distinguishes_from ψ p' Q'› by simp

  have ‹(p ↦a α p') ⟶ hml_srbb_inner.distinguishes_from (BranchConj α ψ Q1 Φ) p (Q1
∪ Qα)›
  proof
    assume ‹p ↦a α p'›
    hence ‹p ↦a α p'› by simp
    with IH(1) have ‹Qα ↦aS α Q'›
      by (simp, metis option.discI)
    hence A2: ‹hml_srbb_inner_models p (Obs α ψ)› using X ‹p ↦a α p'›  by auto
    have A3: ‹∀q ∈ (Q1 ∪ Qα). hml_srbb_inner.distinguishes (BranchConj α ψ Q1 Φ) p q›
    proof (safe)
      fix q
      assume ‹q ∈ Q1›
      hence  ‹hml_srbb_conj.distinguishes (Φ q) p q› using branch_conj(2) by simp
      thus ‹hml_srbb_inner.distinguishes (BranchConj α ψ Q1 Φ) p q›
        using A1 A2 srbb_dist_conjunct_or_branch_implies_dist_branch_conjunction ‹q ∈ Q1›
by blast
    next
      fix q
      assume ‹q ∈ Qα›
      hence ‹¬(hml_srbb_inner_models q (Obs α ψ))›
        using Y ‹Qα ↦aS α Q'› by auto
      hence ‹hml_srbb_inner.distinguishes (Obs α ψ) p q›
        using A2 by auto
      thus ‹hml_srbb_inner.distinguishes (BranchConj α ψ Q1 Φ) p q›
        using A1 A2 srbb_dist_conjunct_or_branch_implies_dist_branch_conjunction by blast
    qed
    have A4: ‹hml_srbb_inner_models p (BranchConj α ψ Q1 Φ)›
      using A3 A2 by fastforce
    with A3 show ‹hml_srbb_inner.distinguishes_from (BranchConj α ψ Q1 Φ) p (Q1 ∪ Qα)›
      by simp
  qed
  then show ?case by simp
qed

end

end
```

11.3 Correctness Theorem

```
theory Silent_Step_Spectroscopy
  imports
    Distinction_Implies_Winning_Budgets
    Strategy_Formulas
begin

context weak_spectroscopy_game
begin
```

```
theorem spectroscopy_game_correctness:
  fixes e p Q
  shows ‹(∃φ. distinguishes_from φ p Q ∧ expressiveness_price φ ≤ e)
       = (attacker_wins e (Attacker_Immediate p Q))›
proof
  assume ‹∃φ. distinguishes_from φ p Q ∧ expressiveness_price φ ≤ e›
  then obtain φ where
    ‹distinguishes_from φ p Q› and le: ‹expressiveness_price φ ≤ e›
    unfolding 𝒪_def by blast
  from distinction_implies_winning_budgets this(1)
    have budget: ‹attacker_wins (expressiveness_price φ) (Attacker_Immediate p Q)› .
  thus ‹attacker_wins e (Attacker_Immediate p Q)› using win_a_upwards_closure le by simp
next
  assume ‹attacker_wins e (Attacker_Immediate p Q)›
  with winning_budget_implies_strategy_formula have
    ‹∃φ. strategy_formula (Attacker_Immediate p Q) e φ ∧ expressiveness_price φ ≤ e›
    by force
  hence ‹∃φ. strategy_formula (Attacker_Immediate p Q) e φ ∧ expressiveness_price φ ≤
e›
    unfolding 𝒪_def by blast
  thus ‹∃φ. distinguishes_from φ p Q ∧ expressiveness_price φ ≤ e›
    using strategy_formulas_distinguish by fastforce
qed

end

end
```

12 Conclusion

We were able to formalize the majority of the paper, including the weak spectroscopy game as introduced by Bisping and Jansen in [1], and to prove one direction of the theorem stating correctness, namely 'if the attacker wins the weak spectroscopy game, given an energy e, then there exists a formula $\varphi \in \text{HML}_{\text{SRBB}}$ with price $\text{expr}(\varphi) \leq e$'(c.f. [1, lemma 2, 3]). For the other direction, we provide a comprehensive proof skeleton, including proofs for individual induction cases.

Due to the nature of Isabelle, the formalization differs from [1]. The gravest change is to the definition of HML_{SRBB}. We have implemented this definition using three mutually recursive data types. As a result, we had two definitions for a conjunction $\bigwedge \psi$, `ImmConj` and `Conj`, each with a different type. The other difference to the HML_{SRBB} definition of [1] concerns the observation of actions. We argue that both definitions have the same distinguishing power. These changes led to necessary adaptations of our definition of the weak spectroscopy game and thereby affected the following definitions and proofs. An overview of these and other deviations can be found in appendix **??**.

A major change compared to [1] is the addition of new game move $(p, \emptyset)_d^s \overset{\hat{e}_4}{\rightarrowtail} (p, \emptyset)_d$ from `Defender_Stable_Conj` to `Defender_Conj` if $Q = \emptyset$. Without this move, the attacker could use an empty stability conjunction `StableConj` without having the proper budget. We formalized a weak spectroscopy game closely related to [1] that can decide (almost) all behavioural equivalences between stability-respecting branching bisimilarity and weak trace equivalence at once. Provided our definition of energies as eight-dimensional vectors corresponds to these equivalences, we implemented a (mostly) machine-checkable proof for the correctness of this spectroscopy game.

To further increase confidence in the results of [1], additional proofs are necessary. Firstly, the proof for 'given an energy e, if there exists a formula $\varphi \in \text{HML}_{\text{SRBB}}$ with price $\text{expr}(\varphi) \leq e$, then the attacker wins the weak spectroscopy game' is senseful (c.f. [1, lemma 1]). Secondly, [1] uses coordinates of energies to define equivalences. One can show that the HML sublanguages obtained from these coordinates correspond to the desired equivalences. Since our formalization of the model relation `hml_models` is only defined on the parameterization of HML by the state type `'s`, one could also show that this formalization sufficiently captures the expressiveness power of HML on labelled transition systems. Finally, [1, proposition 1] claims that their slightly different modal characterization of HML_{SRBB} corresponds to the modal characterization of [2]. The proof for proposition 1 in [1] could be turned into a machine-checkable proof.

References

[1] B. Bisping and D. N. Jansen. Linear-time–branching-time spectroscopy accounting for silent steps, 2023.

[2] W. Fokkink, R. van Glabbeek, and B. Luttik. Divide and congruence iii: From decomposition of modal formulas to preservation of stability and divergence. *Information and Computation*, 268:104435, 2019.