The **Weak Spectroscopy Game** to Characterize Behavioral Equivalences

Lisa A. Barthel[1,2], Leonard M. Hübner[1], Caroline Lemke[1,3], Karl P. P. Mattes[1], Lenard Mollenkopf[1], and Benjamin Bisping[1,4]

[1]Technische Universität Berlin, Germany
[2]Technische Universität Ilmenau, Germany
[3]Carl von Ossietzky Universität Oldenburg, Germany
[4]Télécom SudParis, Institut Polytechnique de Paris, France

November 22, 2025

Abstract

We provide an Isabelle/HOL formalization of Bisping and Jansen's weak spectroscopy game [4], which can be used to simultaneously characterize and decide a hierarchy of behavioral equivalences for systems with internal behavior. This is valuable for applications in concurrency theory and formal verification where equivalences and distinctions of the "linear-time–branching-time spectrum" are a recurring topic.

This entry contains a game characterization of most behavioral equivalences from stability-respecting branching bisimilarity to weak trace equivalence. Technically, the results link distinguishing sublanguages of Hennessy–Milner logic to winning attacker budgets in an energy game through an eight-dimensional measurement of syntactic features appearing in formulas.

Overview. This formalization provides theoretical underpinnings of https://equiv.io, a tool to *decide all behavioral equivalences at once*. By phrasing *equivalences as energy games*, one obtains a uniform way to handle a wide range of equivalences in van Glabbeek's *linear-time–branching-time spectrum* [6, 7]. In particular, we treat systems with *silent* τ-steps, which usually arise because of abstraction from internal behavior, for instance, when modeling communication protocols or distributed systems using transition systems.

This formalization follows Bisping and Jansen's *weak spectroscopy game* [4], respectively the proofs from the arXiv version [3].

- Section 1 provides some basics on transition systems with internal behavior.
- Sections 2 and 3 define a version of Hennessy–Milner logic for systems with internal behavior and a syntactic metric to select sublanguages of it through coordinates.
- Sections 4 to 6 prove certain coordinates to correspond to weak trace equivalence, η-bisimilarity, η-similarity and branching bisimilarity as well as their stable variants. (As the relationship is established through modal logics, the results can be understood as Hennessy–Milner theorems [5].)
- Sections 7 to 9 introduce the weak spectroscopy game and prove that winning attacker energies in the game correspond to coordinates of distinguishing formulas according to the syntactic expressiveness metric.

The broader project of *deciding all equivalences at once* is outlined in Bisping's PhD thesis [2]. There, one can also find more gentle introductions to the topic and to the game-theoretic approach in general. The energy game approach is due to [1].

Acknowledgments. Several proofs in this document follow pen-and-paper proofs by David N. Jansen.

Contents

1 Labeled Transition Systems

```
theory Labeled_Transition_Systems
  imports Main
begin
```

1.1 Base LTS

The locale `LTS` represents a labeled transition system consisting of a set of states $\mathcal{P}$, a set of actions Σ, and a transition relation $\mapsto \subseteq \mathcal{P} \times \Sigma \times \mathcal{P}$. 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`.

```
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`.

```
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 α-derivatives for a set of states `P`.

```
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
```

The lifted `step_setp` (`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 by fastforce

end — of locale lts
```

1.2 Labeled Transition Systems with Silent Steps

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

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

The paper [3] introduces a transition $p \xrightarrow{(\alpha)} p'$ if $p \xrightarrow{\alpha} p'$, or if $\alpha = \tau$ and $p = p'$. We define `soft_step` analogously 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)›

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
```

A state `p` can reach `p'` weakly 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''›
  unfolding weak_step_def using silent_reachable_trans[of p p'] by fastforce
```

A sequence of `weak_steps` from one state `p` to another `p'`.

```
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 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 are all possible sequences of weak transitions that can be performed.

```
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
```

τ can be prepended to any weak trace.

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

lemma silent_prepend_weak_traces:
  assumes
    ‹p ↠ p'›
    ‹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'›
    ‹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
```

A state is weakly trace pre-ordered to another other, `weakly_trace_preordered` denoted by `≲WT` if all its traces can also be observed from the second process.

```
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 sreachable_set_eq:
  assumes ‹P ↠S Q›
  shows ‹Q = silent_reachable_set P›
  using 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
```

A state is stable if it cannot make any further internal steps.

```
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 — of locale lts_tau

end
```

1.3 Modal Logics on LTS

We here supply abstract definitions that would work for all modal logics one might define over an LTS. In particular, this contains mechanisms to derive equivalences from sublogics.

```
theory LTS_Semantics
  imports
    Labeled_Transition_Systems
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))
```

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

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}›
```

1.4 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 `q` must also 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 if it 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, then no formula in that set may distinguish them in any direction.

```
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 — of context lts_semantics

end
```

2 Hennessy–Milner Logic for Stability-Respecting Branching Bisimilarity

```
theory HML_SRBB
  imports LTS_Semantics
begin
```

This section describes a variant of Hennessy–Milner logic that characterizes stability-respecting branching bisimilarity (SRBB).

The following mutually-recursive datatype family describes a grammar of `HML_SRBB` formulas.

```
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›
```

The constructors correspond to more conventional notation of HML as follows:

- `hml_srbb` (members usually referred to as φ):
  - `TT` encodes $\top$
  - `Internal` χ encodes $\langle\varepsilon\rangle\chi$
  - `ImmConj I` ψ`s` encodes $\bigwedge_{i\in \mathtt{I}} \psi s(i)$
- `hml_srbb_inner` (usually χ):
  - `Obs` α φ encodes $(\alpha)\varphi$
  - `Conj I` ψ`s` encodes $\bigwedge_{i\in \mathtt{I}} \psi s(i)$
  - `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 $(\alpha)\varphi \wedge \bigwedge_{i\in \mathtt{I}} \psi s(i)$
- `hml_srbb_conjunct` (usually ψ):
  - `Pos` χ encodes $\langle\varepsilon\rangle\chi$
  - `Neg` χ encodes $\neg\langle\varepsilon\rangle\chi$

2.1 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 an HML_{SRBB} formula φ, written as `p` $\models$`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 .
```

2.2 Distinguishing Formulas

```
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

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

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

lemma srbb_dist_conjunct_implies_dist_imm_conjunction:
  assumes
    ‹i∈I›
    ‹hml_srbb_conj.distinguishes (ψs i) p q›
```

```
    ‹∀i∈I. hml_srbb_conjunct_models p (ψs i)›
  shows
    ‹distinguishes (ImmConj I ψs) p q›
  using assms by auto

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

lemma srbb_dist_conjunct_or_branch_implies_dist_branch_conjunction:
  assumes
    ‹∀i ∈ I. hml_srbb_conjunct_models p (ψs i)›
    ‹hml_srbb_inner_models p (Obs α φ)›
    ‹(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
```

2.3 HML_{SRBB} Implication and Equivalence

```
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›

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›
```

2.4 Substitution and Congruence

```
lemma srbb_internal_subst:
  assumes
    ‹χl ⇚χ⇛ χr›
    ‹φ ⇚srbb⇛ (Internal χl)›
  shows
    ‹φ ⇚srbb⇛ (Internal χr)›
  using assms by force

lemma internal_srbb_cong:
  assumes ‹χl ⇚χ⇛ χr›
  shows ‹(Internal χl) ⇚srbb⇛ (Internal χr)›
  using assms by auto


lemma immconj_cong:
  assumes
    ‹ψsl ‘ I = ψsr ‘ I›
    ‹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)+

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

2.5 Trivial and Equivalent Formulas

```
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+

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

lemma srbb_obs_τ_is_χTT: ‹Obs τ TT ⇚χ⇛ Conj {} ψs›
  by simp

lemma srbb_obs_is_empty_branch_conj: ‹Obs α φ ⇚χ⇛ BranchConj α φ {} ψs›
  by auto

lemma srbb_TT_is_χTT: ‹TT ⇚srbb⇛ Internal (Conj {} ψs)›
  using lts_tau.refl by force

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:
  fixes Φ p I
  defines
    ‹conjfy q ≡
```

```
      (if distinguishes (Φ q) p q
      then conjunctify_distinctions Φ
      else conjunctify_distinctions_dual Φ) p q›
  assumes
    ‹∀q∈I. distinguishes (Φ q) p q ∨ distinguishes (Φ q) q p›
  shows
    ‹∀q∈I. hml_srbb_conj.distinguishes (conjfy 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 (conjfy q) p q›
  proof cases
    case 1
    then show ?thesis using distinction_conjunctification conjfy_def
      by (smt (verit) singleton_iff)
  next
    case 2
    then show ?thesis using distinction_conjunctification_dual singleton_iff
      unfolding distinguishes_def conjfy_def
      by (smt (verit, ccfv_threshold))
  qed
qed

end — of lts_tau

end
```

3 Expressiveness Prices

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

We intend to work on eight-dimensional vectors in an energy game. The dimensions will encode expressiveness prices to HML_{SRBB} formulas. This price is supposed to capture syntactic features needed to describe a certain property and will later be used to select sublogics of specific expressiveness to characterize behavioural equivalences.

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

```
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›)
```

3.1 Comparing and Subtracting 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)›

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

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)

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
```

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)›
  using assms by (simp_all) (metis add.commute add_diff_assoc_enat le_iff_add)+

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
```

Abbreviations for performing subtraction in the energy games.

```
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)›
```

3.2 Minimum Updates

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)›
```

Abbreviations for identity update.

```
abbreviation ‹id_up ≡ Some Some›
```

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 some lemmas to ease the manipulation of expressions using mininum updates.

```
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 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, ccfv_threshold) 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
```

3.3 Components of Expressiveness Prices

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

The (maximal) modal depth (of observations $\langle\alpha\rangle$, (α)) is increased on each `Obs` and `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›
```

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

```
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›
```

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_conjunct ∘ ψs) ‘ I))› |

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

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`.

```
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›
```

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›
```

The maximal modal depth of positive clauses in conjunctions 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_depth 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›
```

The maximal modal depth of negative clauses in conjunctions 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›
```

The depth of negations 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›
```

3.4 Properties of Price Components

```
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)

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)

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)
```

3.5 Expressiveness Price Function

The `expressiveness_price` function combines the eight component 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)
```

Formalizing HML_{SRBB} by mutually recursive data types leads to expressiveness price functions of these other types 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)›

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)›
```

3.6 Prices of Certain Formulas

```
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›
  by auto

lemma expr_pos:
  assumes ‹expr_pr_inner χ ≤ the (min1_6 e)›
  shows ‹expr_pr_conjunct (Pos χ) ≤ e›
  using assms by auto

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›
  using assms
  by (simp) (metis add_diff_cancel_enat add_mono_thms_linordered_semiring(1)
      enat.simps(3) enat_defs(2) energy.sel idiff_0_right
      le_iff_add le_numeral_extra(4) minus_energy_def option.discI
      option.inject)

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 this(5) 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 conj_upds(3)
    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)
```

Here, we show the prices for some specific formulas.

```
lemma example_φ_cp:
  fixes op a b::‹'a› and left right::‹'s›
  defines ‹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›
    ‹branching_conjunction_depth φ = 0›
    ‹unstable_conjunction_depth  φ = 1›
    ‹stable_conjunction_depth    φ = 0›
    ‹immediate_conjunction_depth φ = 0›
    ‹max_positive_conjunct_depth φ = 1›
    ‹max_negative_conjunct_depth φ = 0›
    ‹negation_depth              φ = 0›
  unfolding φ_def 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

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:
  ‹subtract_fn 1 0 0 0 0 0 0 0 (expr_pr_inner (Obs α φ)) = Some (expressiveness_price φ)›
  by simp

end — pause lts_tau context
```

3.7 Characterizing Equivalence by Energy Coordinates

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

lemma price_hierarchy_entails_modal_hierarchy:
  assumes ‹e1 ≤ e2›
  shows ‹𝒪 e1 ⊆ 𝒪 e2›
  using assms unfolding 𝒪_def by auto

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

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

context lts_tau
begin
```

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 $\mathcal{O}$ `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 $\mathcal{O}$ `e`.

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

lemma price_hierachy_preorder_dual:
  assumes
    ‹e1 ≤ e2›
    ‹p ≼ e2 q›
  shows
    ‹p ≼ e1 q›
  using assms price_hierarchy_entails_modal_hierarchy expr_preord_def by auto
```

3.8 Relational Effects of Prices

Certain properties of prices influence the preorder/equivalence relations that are characterized by price coordinates. (This will be important for some behavioral equivalences that we will prove to be characterized by specific 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) Ψα_def distinction_conjunctification_two_way mem_Collect_eq
      by (smt (verit, best))
  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_distinctions 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
```

4 Weak Traces

```
theory Weak_Traces
  imports Main HML_SRBB Expressiveness_Price
begin
```

The point of this theory is to prove that the coordinate `E ∞ 0 0 0 0 0 0 0` precisely characterizes weak trace preorder and equivalence.

4.1 Weak Traces as Modal Constructs

```
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

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))
```

4.2 Weak Trace Observations through Coordinates

The following three lemmas show that the modal-logical characterization of weak traces 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› and
    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› and
    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.simps(2) 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 trace formula φ is satisfied by a state `p` 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
    ‹is_trace_formula φ ⟹ p ⊨SRBB φ
      ⟹ ∃tr ∈ weak_traces p. wtrace_to_srbb tr ⇐srbb⇒ φ›
    ‹is_trace_formula_inner χ ⟹ hml_srbb_inner_models q χ
      ⟹ ∃tr ∈ weak_traces q. wtrace_to_inner tr ⇐χ⇒ χ›
    True
proof (induction φ and χ and ψ arbitrary: p and q)
  case TT
  thus ?case
    using weak_step_sequence.intros(1) silent_reachable.intros(1) by fastforce
next
  case (Internal χ)
  hence ‹is_trace_formula_inner χ›
    using is_trace_formula.cases by blast
  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'› ‹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 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 (metis wtrace_to_srbb.simps(2))
  qed
  ultimately show ?case by blast
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
  have ‹wtrace_to_srbb [] ⇐srbb⇒ ImmConj I ψs›
    using srbb_TT_is_empty_conj ‹I = {}›
    unfolding wtrace_to_srbb.simps by auto
  thus ‹∃tr∈weak_traces p. wtrace_to_srbb tr ⇐srbb⇒ ImmConj I ψs›
    using ‹[] ∈ weak_traces p› by auto
next
  case (Obs α φ)
  thus ?case
```

```
  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'› ‹q' ⊨SRBB φ› by auto
    hence ‹∃tr' ∈ weak_traces q'. wtrace_to_srbb tr' ⇐srbb⇛ φ›
      using ‹is_trace_formula φ› Obs by auto
    then obtain tr' where ‹tr' ∈ weak_traces q'› ‹wtrace_to_srbb tr' ⇐srbb⇛ φ› by auto
    have ‹(α # tr') ∈ weak_traces q›
      using ‹q ↦ α q'› ‹tr' ∈ weak_traces q'› 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)
  from ‹is_trace_formula_inner (Conj I ψs)› have ‹I = {}›
    by (simp add: is_trace_formula_inner.simps)
  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.
  with empty_trace_allways_weak_trace[of q] show ?case by auto
qed (auto simp add: is_trace_formula_inner.simps)

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)
    then 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 ‹wtrace_to_srbb (a # tail) = (Internal (Obs a (wtrace_to_srbb tail)))›
      by simp
    thus ?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
          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
    thus ?case
      using Cons prepend_τ_weak_trace silent_prepend_weak_traces step_prepend_weak_traces
      by fastforce
  qed
qed

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 expressiveness_to_trace_formula trace_equals_trace_to_formula
      trace_formula_implies_trace
    unfolding weakly_trace_preordered_def
    by (metis (no_types, lifting) eq_equality in_mono)
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 $(\infty, 0, 0, 0, 0, 0, 0, 0)$.

```
theorem weak_traces_coordinate: ‹(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
```

5 η-Bisimilarity and η-Similarity

```
theory Eta_Bisimilarity
  imports Expressiveness_Price
begin
```

η-Bisimilarity and η-Similarity are comparably arcane notions of behavioral equivalence. We show that they are characterized by coordinates `E` ∞ ∞ ∞ `0 0` ∞ ∞ ∞ and `E` ∞ ∞ ∞ `0 0` ∞ `0 0` as an illustration of how to connect coordinates and relational characterizations of equivalences.

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

```
context lts_tau
begin
```

We characterize η-bisimilarity through symmetric η-simulations.

```
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
```

5.2 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 fastforce
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 simp
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_distinctions_dual) 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_distinctions_dual) 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 eta_bisim_coordinate: ‹(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)
```

5.3 η-Similarity

```
lemma logic_eta_sim_invariant:
  assumes
    ‹∃R. eta_simulation R ∧  R p0 q0›
    ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
    ‹p0 ⊨SRBB φ›
  shows ‹q0 ⊨SRBB φ›
proof -
  have ‹⋀φ χ ψ.
    (∀p q. (∃R. eta_simulation R ∧  R p q) ⟶ φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)
      ⟶ p ⊨SRBB φ ⟶ q ⊨SRBB φ) ∧
    (∀p q. (∃R. eta_simulation R ∧  R p q) ⟶ χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)
      ⟶ hml_srbb_inner_models p χ ⟶ (∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ)) ∧
    (∀p q. (∃R. eta_simulation R ∧  R p q) ⟶ ψ ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ 0 0)
      ⟶ hml_srbb_conjunct_models p ψ ⟶ hml_srbb_conjunct_models q ψ)›
  proof -
    fix φ χ ψ
    show
      ‹(∀p q. (∃R. eta_simulation R ∧ R p q) ⟶ φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)
        ⟶ p ⊨SRBB φ ⟶ q ⊨SRBB φ) ∧
       (∀p q. (∃R. eta_simulation R ∧ R p q) ⟶ χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)
        ⟶ hml_srbb_inner_models p χ ⟶ (∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ)) ∧
       (∀p q. (∃R. eta_simulation R ∧ R p q) ⟶ ψ ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ 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 R
    assume case_assms:
      ‹eta_simulation R› ‹R p q›  ‹p ⊨SRBB hml_srbb.Internal χ›
      ‹Internal χ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 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 ∞ 0 0)›
      using case_assms(4) unfolding 𝒪_inner_def 𝒪_def by auto
    hence ‹∃q'. q ↠ q' ∧ hml_srbb_inner_models q' χ›
      using Internal case_assms(1,2) p'_spec silence_retains_eta_sim
      by (metis 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 R
    assume case_assms:
      ‹eta_simulation R› ‹R p q›
      ‹Obs α φ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
      ‹hml_srbb_inner_models p (hml_srbb_inner.Obs α φ)›
    hence ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 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'''›  ‹R p' q'''›
      using ‹eta_simulation R› ‹R p q› unfolding eta_simulation_def
      using silent_reachable.refl by blast
    hence ‹q''' ⊨SRBB φ› using ‹p' ⊨SRBB φ› Obs ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
      using case_assms(1) 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 R
    assume case_assms:
      ‹eta_simulation R› ‹R p q›
      ‹Conj I Ψ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
      ‹hml_srbb_inner_models p (Conj I Ψ)›
    hence conj_price: ‹∀i∈I. Ψ i ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ 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 ‹eta_simulation R› ‹R p 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 R
    assume case_assms:
      ‹eta_simulation R› ‹R p q›
      ‹BranchConj α φ I Ψ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
      ‹hml_srbb_inner_models p (BranchConj α φ I Ψ)›
    hence ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 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 ∞ 0 0)›
      unfolding 𝒪_conjunct_def 𝒪_inner_def
      by (simp, metis SUP_bot_conv(1) bot_enat_def bot_eq_sup_iff)
    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'''›
      ‹R p q'› ‹R p' q'''›
      using ‹eta_simulation R› ‹R p q› silent_reachable.refl
      unfolding eta_simulation_def by blast
    hence ‹q''' ⊨SRBB φ›
      using BranchConj ‹p' ⊨SRBB φ› ‹φ ∈ 𝒪 (E ∞ ∞ ∞ 0 0 ∞ 0 0)› case_assms 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 q'_q''_spec conj_price case_assms
        ‹∀i∈I. hml_srbb_conjunct_models p (Ψ i)›
      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 R
      assume case_assms:
        ‹eta_simulation R› ‹R p q›
        ‹Pos χ ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
        ‹hml_srbb_conjunct_models p (Pos χ)›
      hence ‹χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 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 case_assms ‹χ ∈ 𝒪_inner (E ∞ ∞ ∞ 0 0 ∞ 0 0)› silence_retains_eta_sim
        by (smt (verit, ccfv_threshold) 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 R
      assume case_assms:
        ‹eta_simulation R› ‹R p q›
        ‹Neg χ ∈ 𝒪_conjunct (E ∞ ∞ ∞ 0 0 ∞ 0 0)›
        ‹hml_srbb_conjunct_models p (Neg χ)›
      hence False unfolding 𝒪_conjunct_def by auto
      thus ‹hml_srbb_conjunct_models q (Neg χ)› by simp
    qed
  qed
 qed
 thus ?thesis using assms by blast
qed

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 eta_sim_coordinate:
    ‹(p ≼ (E ∞ ∞ ∞ 0 0 ∞ 0 0) q) = (∃R. eta_simulation R ∧ R p q)›
  using modal_eta_sim logic_eta_sim_invariant unfolding expr_preord_def
  by auto

end

end
```

6 Branching Bisimilarity

```
theory Branching_Bisimilarity
  imports Eta_Bisimilarity
begin
```

The whole of the modal logic of `hml_srbb` precisely characterizes stability-respecting branching bisimilarity.

6.1 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›
```

6.2 Properties of Branching Bisimulation Equivalences

```
lemma branching_bisimilarity_branching_sim: ‹branching_simulation (~SRBB)›
  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 (~SRBB)›
  unfolding sr_branching_bisimulated_def stability_respecting_def by blast

lemma sr_branching_bisimulation_silently_retained:
  assumes
    ‹p ~SRBB q›
    ‹p ↠ p'›
```

```
  shows
    ‹∃q'. q ↠ q' ∧ p' ~SRBB q'› using assms(2,1)
  using branching_bisimilarity_branching_sim silence_retains_branching_sim by blast

lemma sr_branching_bisimulation_sim:
  assumes
    ‹p ~SRBB q›
    ‹p ↠ p'› ‹p' ↦a α p''›
  shows
    ‹∃q' q''. q ↠ q' ∧ q' ↦a α q'' ∧ p' ~SRBB q' ∧ p'' ~SRBB 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
    ‹p ~SRBB q›
  shows
    ‹q ~SRBB 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 assms 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
```

6.3 `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_retained)
      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 sr_branching_bisim_coordinate: ‹(p ~SRBB q) = (p ≼ (E ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞) q)›
  using sr_branching_bisim_is_hmlsrbb 𝒪_sup
  unfolding expr_preord_def by auto

end
```

```
end
```

7 Energy Games

```
theory Energy_Games
  imports Main
begin
```

Energy games are the foundation for the weak spectroscopy game. We introduce them through a recursive definition of attacker's winning budgets in energy reachability games.

7.1 Fundamentals

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

An energy game is played by two players on a directed graph labeled by energy updates. These updates represent the costs of choosing a certain move, in our case, for the attacker.

```
locale energy_game =
fixes
  weight_opt :: ‹'gstate ⇒ 'gstate ⇒ 'energy update option› and
  defender :: ‹'gstate ⇒ bool› 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

abbreviation attacker :: ‹'gstate ⇒ bool› where
  ‹attacker p ≡ ¬ defender 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)›
```

7.2 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.

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

lemma attacker_wins_Ga_with_id_step:
  assumes ‹attacker_wins e g'› ‹g ↣wgt Some g'› ‹attacker g›
  shows ‹attacker_wins e g›
  using assms by (metis attacker_wins.simps)
```

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.

```
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.simps 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 — context energy_game

end
```

8 Weak Spectroscopy Game

```
theory Spectroscopy_Game
  imports Energy_Games Energy Labeled_Transition_Systems
begin
```

The weak spectroscopy game is an energy game played over an LTS. The attacker's moves in the weak spectroscopy game depend on the transitions of the processes and the available energy. Intuitively, each move type corresponds to a production in the construction of distinguishing formulas; and each attacker position to a non-terminal in the underlying grammar.

8.1 Game Rules

```
datatype ('s, 'a) spectroscopy_position =
  Attacker_Immediate (attacker_state: <'s>) (defender_states: <'s set>) |
  Attacker_Delayed (attacker_state: <'s>) (defender_states: <'s set>) |
  Attacker_Conjunct (attacker_state: <'s>) (defender_state: <'s>) |
  Attacker_Branch (attacker_state: <'s>) (defender_states: <'s set>) |

  Defender_Conj (attacker_state: <'s>) (defender_states: <'s set>) |
  Defender_Stable_Conj (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>)

context lts_tau
begin
```

The names of moves of the weak spectroscopy game indicate the respective HML constructs they correspond to.

```
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 id_up else None)> |

  procrastination:
    <spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed p' Q')
      = (if (Q' = Q ∧ p ≠ p' ∧ p ↦ τ p') then id_up 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 id_up else None)> |

  conj_answer:
    <spectroscopy_moves (Defender_Conj p Q) (Attacker_Conjunct 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_Conjunct 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 id_up else None)› |

  conj_s_answer:
    ‹spectroscopy_moves (Defender_Stable_Conj p Q) (Attacker_Conjunct 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 id_up
        else None)› |

  br_answer:
    ‹spectroscopy_moves (Defender_Branch p α p' Q Qα) (Attacker_Conjunct 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_Conjunct _ _) = False› |
  ‹spectroscopy_defender (Attacker_Delayed _ _) = False› |
  ‹spectroscopy_defender (Defender_Branch _ _ _ _ _) = True› |
  ‹spectroscopy_defender (Defender_Conj _ _) = True› |
  ‹spectroscopy_defender (Defender_Stable_Conj _ _) = True›
```

8.2 Energy Game Properties

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

```
sublocale weak_spectroscopy_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_Conjunct p q)
    hence ‹∃p' Q'. g'= (Attacker_Delayed p' Q')›
      using monotonicity_assms(1,2)
      by (induct, auto)
    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_Conjunct
      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)›
      using monotonicity_assms(1)
      by (induct, auto)
    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_Conjunct p q)
  hence ‹∃p' Q'. g'= (Attacker_Delayed p' Q')›
    using defender_win_min_assms(2) by (induct, auto)
  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_Conjunct
    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))›
    using defender_win_min_assms(2) by (induct, auto)
  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_Conjunct 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

abbreviation ‹spectro_att_wins ≡ weak_spectroscopy_game.attacker_wins›

end — of lts_tau

end
```

9 Correctness

Energy levels where the defender wins in the spectroscopy game and equivalences coincide in the following sense: 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 attacker's winning budget of `Attacker_Immediate p Q`.

The proof is split into two directions, closely following the structure of [3]. The forward direction is given by the lemma `distinction_implies_winning_budgets` 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`.

9.1 Distinction Implies Winning Budgets

```
theory Distinction_Implies_Winning_Budgets
  imports Spectroscopy_Game Expressiveness_Price
begin

context lts_tau
begin
```

We prove that if a formula distinguishes process `p` from a set of process `Q`, then the price of this formula is in attacker's winning budgets.

```
lemma distinction_implies_winning_budgets_empty_Q:
  assumes
    ‹distinguishes_from φ p {}›
  shows
    ‹spectro_att_wins (expressiveness_price φ) (Attacker_Immediate p {})›
  using assms
proof -
  have ‹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: ‹spectro_att_wins (expressiveness_price φ) (Defender_Conj p {})›
    by (simp add: weak_spectroscopy_game.attacker_wins.Defense)
  from late_inst_conj[of p ‹{}› p ‹{}›] have next_move0:
    ‹spectroscopy_moves (Attacker_Delayed p {}) (Defender_Conj p {}) = id_up› by force
  from delay[of p ‹{}› p ‹{}›] have next_move1:
    ‹spectroscopy_moves (Attacker_Immediate p {}) (Attacker_Delayed p {}) = id_up› by force
  moreover have ‹weak_spectroscopy_game.attacker (Attacker_Immediate p {})› by simp
  ultimately show ?thesis
    using weak_spectroscopy_game.attacker_wins.Attack next_move0 next_move1
    by (metis conj_win option.distinct(1) option.sel spectroscopy_defender.simps(4))
qed

lemma distinction_implies_winning_budgets:
  assumes
    ‹distinguishes_from φ p Q›
  shows
    ‹spectro_att_wins (expressiveness_price φ) (Attacker_Immediate p Q)›
proof -
  have
    ‹ (∀Q p. Q ≠ {} ⟶ distinguishes_from φ p Q
            ⟶ spectro_att_wins (expressiveness_price φ)(Attacker_Immediate p Q))
    ∧
      ((∀p Q. Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from χ p Q ⟶ Q ↠S Q
          ⟶ spectro_att_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
          ⟶ spectro_att_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')
        ⟶ spectro_att_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)
        ⟶ spectro_att_wins (expr_pr_inner χ) (Defender_Branch p α p' (Q - Q_α) Q_α)))
  ∧
    (∀p q. hml_srbb_conj.distinguishes ψ p q
          ⟶ spectro_att_wins (expr_pr_conjunct ψ) (Attacker_Conjunct p q))›
  for φ χ ψ
proof -
  fix φ χ ψ
  show ‹(∀Q p. Q ≠ {} ⟶ distinguishes_from φ p Q
           ⟶ spectro_att_wins (expressiveness_price φ) (Attacker_Immediate p Q))
    ∧
      ((∀p Q. Q ≠ {} ⟶ hml_srbb_inner.distinguishes_from χ p Q ⟶ Q ↠S Q
          ⟶ spectro_att_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
          ⟶ spectro_att_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')
          ⟶ spectro_att_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)
          ⟶ spectro_att_wins (expr_pr_inner χ) (Defender_Branch p α p' (Q - Q_α) Q_α)))
    ∧
      (∀p q. hml_srbb_conj.distinguishes ψ p q
            ⟶ spectro_att_wins (expr_pr_conjunct ψ) (Attacker_Conjunct 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 ≠ {}› ‹distinguishes_from TT p Q›
      hence ‹∃q. q ∈ Q›
        by blast
      then obtain q where ‹q ∈ Q› by auto
      hence ‹distinguishes TT p q›
        using ‹distinguishes_from TT p Q› distinguishes_from_def by auto
      with verum_never_distinguishes
      show ‹spectro_att_wins (expressiveness_price TT) (Attacker_Immediate p Q)›
        by blast
    qed
  next
    case (Internal χ)
    show ?case
    proof (clarify)
      fix Q p
      assume ‹Q ≠ {}› ‹distinguishes_from (Internal χ) p Q›
      then have
        ‹∃p'. p ↠ p' ∧ hml_srbb_inner_models p' χ›
        ‹∀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'› ‹hml_srbb_inner_models p' χ› by auto
    from this(1) have ‹p ↠L p'› using silent_reachable_impl_loopless by blast
    have ‹Qτ ≠ {}›
      using silent_reachable.intros(1) sreachable_set_is_sreachable Qτ_def ‹Q ≠ {}›
      by fastforce
    from ‹hml_srbb_inner_models p' χ› ‹∀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 ‹spectro_att_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 ‹spectro_att_wins (expressiveness_price (Internal χ))
        (Attacker_Delayed p' Qτ)› by simp
    hence ‹spectro_att_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 ‹spectro_att_wins (expressiveness_price (Internal χ))
                              (Attacker_Delayed p' Qτ)›
        by simp
      moreover have ‹spectroscopy_moves (Attacker_Delayed p Qτ) (Attacker_Delayed p' Qτ)
        = id_up› using spectroscopy_moves.simps(2) ‹p ≠ p'› ‹p ↦τ p'› by auto
      moreover have ‹weak_spectroscopy_game.attacker (Attacker_Delayed p Qτ)› by simp
      ultimately show ?case
        using weak_spectroscopy_game.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) = id_up›
      using spectroscopy_moves.simps(1) by simp
    with ‹spectro_att_wins (expressiveness_price (Internal χ)) (Attacker_Delayed p Qτ)›
    show ‹spectro_att_wins (expressiveness_price (Internal χ)) (Attacker_Immediate p Q)›
      using weak_spectroscopy_game.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.
```

```
      spectro_att_wins (expr_pr_conjunct (ψs i)) (Attacker_Conjunct p q)›
    using ImmConj by blast
  hence a_clause_wina:
    ‹∀q∈Q. ∃i∈I.
      spectro_att_wins (expressiveness_price (ImmConj I ψs) - E 0 0 1 0 1 0 0 0)
                       (Attacker_Conjunct p q)›
    using expressiveness_price_ImmConj_geq_parts
      weak_spectroscopy_game.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_Conjunct (attacker_state N) (defender_state N)› for N
    by (metis spectroscopy_moves.simps(29,30,33,34,58,62)
        spectroscopy_position.exhaust_sel)
  hence move_kind: ‹spectroscopy_moves (Defender_Conj p Q) N ≠ None
      ⟹ ∃q∈Q. N = Attacker_Conjunct p q› for N
    using conj_answer by metis
  hence update: ‹⋀g'. spectroscopy_moves (Defender_Conj p Q) g' ≠ None ⟹
    weak_spectroscopy_game.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)
     ∧  spectro_att_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 weak_spectroscopy_game.attacker_wins.Defense[OF def_conj] update move_wina have
    def_conj_wina:
      ‹spectro_att_wins (expressiveness_price (ImmConj I ψs) - E 0 0 0 0 1 0 0 0)
                        (Defender_Conj p Q)›
    by (metis (lifting))
  have imm_to_conj:
      ‹spectroscopy_moves (Attacker_Immediate p Q) (Defender_Conj p Q) ≠ None›
    by (simp add: ‹Q ≠ {}›)
  have imm_to_conj_wgt:
    ‹weak_spectroscopy_game.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 weak_spectroscopy_game.Attack[OF _ imm_to_conj imm_to_conj_wgt] def_conj_wina
  show
    ‹spectro_att_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 ⟶ spectro_att_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'
          ∧ spectro_att_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 ‹spectro_att_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 ‹spectro_att_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: ‹spectro_att_wins (expressiveness_price φ) (Attacker_Immediate p' Q')› by blast
    have attacker: ‹weak_spectroscopy_game.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 subtract 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') =
          subtract 1 0 0 0 0 0 0 0› using p'_Q' by auto
    with expr_obs_phi[of α φ] show
      ‹spectro_att_wins (expr_pr_inner (hml_srbb_inner.Obs α φ)) (Attacker_Delayed p Q)›
      using weak_spectroscopy_game.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
  ⟶ spectro_att_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
      ∧ spectro_att_wins (expr_pr_conjunct (ψs i)) (Attacker_Conjunct 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
      ∧ spectro_att_wins (expr_pr_conjunct ψ) (Attacker_Conjunct 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
      ∧ spectro_att_wins (expr_pr_conjunct (ψqs q)) (Attacker_Conjunct 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 weak_spectroscopy_game.win_a_upwards_closure
    have clause_win: ‹∀q∈Q. spectro_att_wins e' (Attacker_Conjunct 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_Conjunct 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_Conjunct 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_Conjunct 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'
          ∧ spectro_att_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 ‹spectro_att_wins e (Defender_Conj p Q)›
    unfolding e_def using weak_spectroscopy_game.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 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
    ‹spectro_att_wins (expr_pr_inner (hml_srbb_inner.Conj I ψs)) (Defender_Conj p Q)›
    using weak_spectroscopy_game.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  ⟶ spectro_att_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
    ‹spectro_att_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) = id_up›
    by auto
  ultimately show
    ‹spectro_att_wins (expr_pr_inner (hml_srbb_inner.Conj I ψs))
                      (Attacker_Delayed p Q)›
    by (metis weak_spectroscopy_game.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')
       ⟶ spectro_att_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
       ∧ spectro_att_wins (expr_pr_conjunct (ψs i)) (Attacker_Conjunct 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
        ∧ spectro_att_wins (expr_pr_conjunct ψ) (Attacker_Conjunct 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
        ∧ spectro_att_wins (expr_pr_conjunct (ψqs q)) (Attacker_Conjunct 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 weak_spectroscopy_game.win_a_upwards_closure
      have clause_win: ‹∀q∈Q. spectro_att_wins e' (Attacker_Conjunct 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_Conjunct 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_Conjunct 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_Conjunct 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'
            ∧ spectro_att_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 ‹spectro_att_wins e (Defender_Stable_Conj p Q)›
      unfolding e_def
      by (auto simp add: weak_spectroscopy_game.attacker_wins.Defense)
    moreover have ‹e ≤ expr_pr_inner (StableConj I ψs)›
      using ‹e' ≤ eu'› eu'_characterization 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
        ‹spectro_att_wins (expr_pr_inner (StableConj I ψs)) (Defender_Stable_Conj p Q)›
      using weak_spectroscopy_game.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
    ⟶ spectro_att_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') = id_up›
    unfolding Q'_def by auto
  show ‹spectro_att_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
        ∧ spectro_att_wins (e - E 0 0 0 1 0 0 0 0) (Defender_Conj p {})
        ⟶ spectro_att_wins e (Defender_Stable_Conj p Q')›
      using weak_spectroscopy_game.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. spectro_att_wins e (Defender_Conj p {})›
      using weak_spectroscopy_game.attacker_wins.Defense 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)
      ⟶ spectro_att_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 weak_spectroscopy_game.attacker_wins_Ga_with_id_step option.discI
          option.sel spectroscopy_defender.simps(4))
  next
    case False
    with move show ?thesis
      using main_case Q'_spec weak_spectroscopy_game.attacker_wins_Ga_with_id_step
      unfolding Q'_def
      by (metis (mono_tags, lifting) mem_Collect_eq option.distinct(1) option.sel
          spectroscopy_defender.simps(4))
  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)
```

```
    ⟶ spectro_att_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
      ∧ spectro_att_wins (expr_pr_conjunct (ψs i)) (Attacker_Conjunct 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
      ∧ spectro_att_wins (expr_pr_conjunct ψ) (Attacker_Conjunct 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
        ∧ spectro_att_wins (expr_pr_conjunct (ψqs q)) (Attacker_Conjunct 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›
    using e'0_def SUP_upper energy.sel energy.simps(3) energy_leq_cases image_iff
    by (smt (z3))
  with ψqs_spec weak_spectroscopy_game.win_a_upwards_closure have
    conj_wins: ‹∀q∈(Q ∩ hml_srbb_inner.model_set (Obs α φ)).
      spectro_att_wins e'0 (Attacker_Conjunct 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:
  ‹spectro_att_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:
  ‹spectro_att_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 weak_spectroscopy_game.attacker_wins.Attack have obs_later_win:
    ‹spectro_att_wins (expr_pr_inner (Obs α φ)) (Attacker_Branch p’ Q’)›
  by force
hence e’_win: ‹spectro_att_wins e’ (Attacker_Branch p’ Q’)›
  unfolding e’_def using weak_spectroscopy_game.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_conjuncts 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: ‹spectro_att_wins (the (min1_6 e')) (Attacker_Branch p' Q')›
 using obs_later_win weak_spectroscopy_game.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_Conjunct 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_Conjunct 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'
        ∧ spectro_att_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_Conjunct p q)
      = (subtract 0 1 1 0 0 0 0 0)
      ⟶ spectro_att_wins e'0 (Attacker_Conjunct 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' ∧ spectro_att_wins e' g')›
      using  ‹eu'0 ≤ e'› e'_comp ‹e'0 ≤ eu'0› weak_spectroscopy_game.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 ‹spectro_att_wins (expr_pr_inner (BranchConj α φ I ψs))
                                      (Defender_Branch p α p' (Q - Q_α) Q_α)›
      using weak_spectroscopy_game.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
       ⟶ spectro_att_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 ‹spectro_att_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) = id_up›
      using p'_spec Q_α_def by auto
    ultimately show
        ‹spectro_att_wins (expr_pr_inner (BranchConj α φ I ψs)) (Attacker_Delayed p Q)›
      using weak_spectroscopy_game.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 ‹spectro_att_wins (expr_pr_inner χ)
                          (Attacker_Delayed p' (silent_reachable_set {q}))›
    using distinction Pos by blast
  from p'_spec(1) this have
      ‹spectro_att_wins (expr_pr_inner χ) (Attacker_Delayed p (silent_reachable_set {q}))›
    by (induct, auto,
        metis weak_spectroscopy_game.attacker_wins_Ga_with_id_step procrastination
          option.distinct(1) option.sel spectroscopy_defender.simps(4))
  moreover have ‹spectroscopy_moves (Attacker_Conjunct 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 ‹spectro_att_wins (expr_pr_conjunct (Pos χ)) (Attacker_Conjunct p q)›
    using weak_spectroscopy_game.attacker_wins.simps
      weak_spectroscopy_game.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 ‹spectro_att_wins (expr_pr_inner χ)
                            (Attacker_Delayed q' (silent_reachable_set {p}))›
      using distinction Neg by blast
    from q'_spec(1) this have
        ‹spectro_att_wins (expr_pr_inner χ) (Attacker_Delayed q (silent_reachable_set {p}))›
      by (induct, auto,
          metis weak_spectroscopy_game.attacker_wins_Ga_with_id_step procrastination
            option.distinct(1) option.sel spectroscopy_defender.simps(4))
    moreover have ‹spectroscopy_moves (Attacker_Conjunct 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 ‹spectro_att_wins (expr_pr_conjunct (Neg χ)) (Attacker_Conjunct p q)›
        using weak_spectroscopy_game.attacker_wins.Attack
          weak_spectroscopy_game.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
```

9.2 Strategy Formulas

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

Strategy formulas express attacker strategies in HML. They bridge between HML formulas, the spectroscopy game and winning budgets. We show that, if some energy `e` suffices for the attacker to win, there exists a strategy formula with expressiveness price $\leq$ `e`. We also prove that this formula actually distinguishes the processes of the attacker position.

```
context lts_tau
begin

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') = id_up
      ∧ spectro_att_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) = id_up
        ∧ spectro_att_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)
        ∧ spectro_att_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)
          ∧ spectro_att_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)
       = id_up ∧ (spectro_att_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_Conjunct p q)
        = (subtract 0 0 1 0 0 0 0 0)
          ∧ (spectro_att_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Conjunct p q))
          ∧ strategy_formula_conjunct (Attacker_Conjunct 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_Conjunct p q)
        = (subtract 0 0 1 0 0 0 0 0)
          ∧ (spectro_att_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Conjunct p q))
          ∧ strategy_formula_conjunct (Attacker_Conjunct p q) (e - E 0 0 1 0 0 0 0 0) (Φ q)›
|
  pos:  ‹strategy_formula_conjunct (Attacker_Conjunct p q) e (Pos χ)›
    if ‹(∃Q'. spectroscopy_moves (Attacker_Conjunct p q) (Attacker_Delayed p Q')
      = Some min1_6 ∧ spectro_att_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_Conjunct p q) e (Neg χ)›
    if ‹∃P'. (spectroscopy_moves (Attacker_Conjunct p q) (Attacker_Delayed q P')
      = Some (λe. Option.bind ((subtract_fn 0 0 0 0 0 0 0 1) e) min1_7)
        ∧ spectro_att_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')
      = id_up ∧ spectro_att_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_Conjunct p q)
      = (subtract 0 0 0 1 0 0 0 0)
        ∧ spectro_att_wins (e - (E 0 0 0 1 0 0 0 0)) (Attacker_Conjunct p q)
        ∧ strategy_formula_conjunct (Attacker_Conjunct 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) = id_up
          ∧ spectro_att_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
            ∧ (spectro_att_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_Conjunct p q)
```

```
        = (subtract 0 1 1 0 0 0 0 0)
        ∧ spectro_att_wins (e - (E 0 1 1 0 0 0 0 0)) (Attacker_Conjunct p q)
        ∧ strategy_formula_conjunct (Attacker_Conjunct p q) (e - E 0 1 1 0 0 0 0 0) (Φ q)›

lemma winning_budget_implies_strategy_formula:
  assumes
    ‹spectro_att_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_Conjunct 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: weak_spectroscopy_game.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 (weak_spectroscopy_game.weight g g' e)) φ
          ∧ expr_pr_inner φ ≤ weak_spectroscopy_game.updated g g' e) ∧
      (∃p Q. g' = Attacker_Delayed p Q) ⟶
        (∃φ. strategy_formula_inner g' (the (weak_spectroscopy_game.weight g g' e)) φ
          ∧ expr_pr_inner φ ≤ weak_spectroscopy_game.updated g g' e)›
      using weak_spectroscopy_game.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 ‹(spectro_att_wins e (Attacker_Delayed p Q'))›
        using ‹g' = Attacker_Delayed p' Q'› ‹p' = p› Attacker_Immediate
          weak_spectroscopy_game.win_a_upwards_closure e_comp
        by simp
      ultimately have ‹(∃χ.
        strategy_formula_inner g'
          (the (weak_spectroscopy_game.weight (Attacker_Immediate p Q) g' e)) χ ∧
        expr_pr_inner χ ≤ weak_spectroscopy_game.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› e_comp g'_att_del by auto
  hence ‹∃Q'. spectroscopy_moves (Attacker_Immediate p Q) (Attacker_Delayed p Q') = id_up
    ∧ (spectro_att_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
          ‹spectro_att_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 M':
    ‹weak_spectroscopy_game.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 M'': ‹(spectro_att_wins (e - (E 0 0 0 0 1 0 0 0)) (Defender_Conj p Q'))›
    using g'_def_conj Qp' Attacker_Immediate weak_spectroscopy_game.win_a_upwards_closure
    by force
  with g'_def_conj have IH_case: ‹∃χ.
    strategy_formula_inner g'
      (weak_spectroscopy_game.updated (Attacker_Immediate p Q) g' e) χ
    ∧ expr_pr_inner χ ≤ weak_spectroscopy_game.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 ‹spectro_att_wins (e - (E 0 0 0 0 1 0 0 0)) (Defender_Conj p Q')› IH_case
      Qp' M' 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
      g'_def_conj Attacker_Immediate
    unfolding Qp'
    by (smt (verit) spectroscopy_moves.simps(64,70) spectroscopy_position.distinct(17)
        spectroscopy_position.inject(5) strategy_formula_inner.cases)
  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_strategy_formula_inner_strategy_formula_conjunct.imm_conj
      Qp' Attacker_Immediate unfolding g'_def_conj
    by (smt (verit) lts_tau.spectroscopy_moves.simps(70) hml_srbb_inner.inject(2)
        spectroscopy_position.distinct(17,37) strategy_formula_inner.cases)
  hence SI: ‹strategy_formula (Attacker_Immediate p Q) e (ImmConj Q ψ)›
    using delay early_conj Qp'
    by (metis (no_types, lifting) M'' 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 (induct g', auto) (metis option.discI)+
  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' (weak_spectroscopy_game.updated (Attacker_Branch p Q) g' e) φ ∧
     expressiveness_price φ ≤ weak_spectroscopy_game.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_Conjunct 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_Conjunct by presburger
    hence post_win:
      ‹(the (spectroscopy_moves (Attacker_Conjunct p q) g') e) = min1_6 e›
       ‹(spectro_att_wins (the (min1_6 e)) (Attacker_Delayed p Q'))›
      using ‹{q} ↠S Q'› Attacker_Conjunct weak_spectroscopy_game.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_Conjunct Attack True post_win unfolding g'_att_del
      by (smt (verit) option.sel spectroscopy_position.simps(51))
    hence
      ‹spectroscopy_moves (Attacker_Conjunct p q) (Attacker_Delayed p Q') = Some min1_6›
      ‹spectro_att_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_Conjunct 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_Conjunct unfolding g'_att_del
        by presburger+
      hence move: ‹spectroscopy_moves (Attacker_Conjunct 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:
        ‹spectro_att_wins (the (min1_7 (e - E 0 0 0 0 0 0 0 1))) (Attacker_Delayed p' Q')›
```

```
      using Attacker_Conjunct 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_Conjunct move unfolding g'_att_del
      by (smt (verit, del_insts) bind.bind_lunit bind_eq_None_conv option.discI
          option.sel spectroscopy_position.simps(51))
    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_Conjunct 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_Conjunct 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 (cases g', auto)
  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: ‹(spectro_att_wins e (Attacker_Delayed p' Q'))›
      using g'_att_del Qp' Attacker_Delayed weak_spectroscopy_game.attacker_wins.Defense e_comp
      by (metis option.sel)
    have ‹(weak_spectroscopy_game.updated (Attacker_Delayed p Q) g' e) = e›
      using g'_att_del Attacker_Delayed e_comp by fastforce
    then obtain χ where χ_spec:
        ‹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) = id_up
       ∧  spectro_att_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 procrastination by blast
    moreover have ‹expr_pr_inner χ ≤ e›
      using χ_spec 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:
    ‹weak_spectroscopy_game.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 ‹spectro_att_wins (e - (E 1 0 0 0 0 0 0 0)) (Attacker_Immediate p' Q')›
    using weak_spectroscopy_game.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
      ‹spectro_att_wins e (Defender_Conj p' Q')›
      ‹weak_spectroscopy_game.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') = id_up
    ∧ spectro_att_wins e (Defender_Conj p' Q')
    ∧ strategy_formula_inner (Defender_Conj p' Q') e χ›
    by (simp add: ‹Q = Q'› ‹spectro_att_wins e (Defender_Conj p' Q')› ‹p = p'›)
  then show ?thesis
    using χ_prop ‹Q = Q'› ‹spectro_att_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 ‹spectro_att_wins e (Defender_Stable_Conj p' Q')›
        ‹weak_spectroscopy_game.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') = id_up
      ∧ spectro_att_wins e (Defender_Stable_Conj p' Q')
      ∧ strategy_formula_inner (Defender_Stable_Conj p' Q') e χ›
      using Attack χ_prop ‹spectro_att_wins e (Defender_Stable_Conj p' Q')› late_stbl_conj
        pQ' 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 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:
        ‹spectro_att_wins e (Defender_Branch p' α p'' Q' Qα)›
        ‹weak_spectroscopy_game.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) = id_up
       ∧ spectro_att_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
 qed force+
next
 case (Defense g e)
 thus ?case
 proof (induct g)
  case (Defender_Branch p α p' Q Qa)
  hence conjs:
    ‹∀q∈ Q. spectroscopy_moves (Defender_Branch p α p' Q Qa) (Attacker_Conjunct p q)
      = subtract 0 1 1 0 0 0 0 0›
    by simp
  obtain e' where e'_spec:
    ‹∀q∈Q. weak_spectroscopy_game.weight (Defender_Branch p α p' Q Qa)
        (Attacker_Conjunct p q) e = Some e'
      ∧ spectro_att_wins e' (Attacker_Conjunct p q)
      ∧ (∃ψ. strategy_formula_conjunct (Attacker_Conjunct 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_Conjunct 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(53))
then obtain φ where
  ‹strategy_formula (Attacker_Immediate p' (soft_step_set Qa α))
    (weak_spectroscopy_game.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
    ≤ weak_spectroscopy_game.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(53))
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)›
    ‹spectro_att_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›
  ‹spectro_att_wins (weak_spectroscopy_game.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 weak_spectroscopy_game.attacker_wins.cases
  by (metis spectroscopy_defender.simps(2) option.sel)
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 (cases g'', auto) (metis option.discI)+
then obtain e''' where win_immediate:
  ‹Some e''' = subtract_fn 1 0 0 0 0 0 0 0 e''›
  ‹spectro_att_wins e''' (Attacker_Immediate p' (soft_step_set Qa α))›
```

```
    using g''_spec win_branch option.distinct(1) option.sel spectroscopy_defender.elims(1)
      spectroscopy_defender.simps(2)
      weak_spectroscopy_game.attacker_wins.cases[OF win_branch(2)]
    by (smt (verit, del_insts) local.br_acct spectroscopy_moves.simps(23,53,57,61,66,72))
  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 e'_def obs Defender_Branch(2) win_branch(1) win_immediate(1)
    by (smt (verit, best) bind_eq_None_conv 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'. weak_spectroscopy_game.weight (Defender_Conj p Q) g' e = Some e'
          ∧ spectro_att_wins e' g')
        ∧ (∃q. g' = (Attacker_Conjunct p q))›
    using local.conj_answer
      lts_tau.spectroscopy_defender.elims spectroscopy_moves.simps(30,33,34,47,58,62)
    by (smt (verit, best))
  show ?case
  proof (cases ‹Q = {}›)
    case True
    then obtain Φ where ‹∀q ∈ Q.
      spectroscopy_moves (Defender_Conj p Q) (Attacker_Conjunct p q)
         = subtract 0 0 1 0 0 0 0 0
      ∧ spectro_att_wins (e - (E 0 0 1 0 0 0 0 0)) (Attacker_Conjunct p q)
      ∧ strategy_formula_conjunct (Attacker_Conjunct 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 ‹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_Conjunct p q)
        = subtract 0 0 1 0 0 0 0 0
    ∧ spectro_att_wins e' (Attacker_Conjunct 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_Conjunct p q)
        = subtract 0 0 1 0 0 0 0 0
    ∧ Some e' = subtract_fn 0 0 1 0 0 0 0 0 e
    ∧ spectro_att_wins e' (Attacker_Conjunct p q)
    ∧ (∃φ. strategy_formula_conjunct (Attacker_Conjunct 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_Conjunct p q)
        = subtract 0 0 1 0 0 0 0 0›
      ‹spectro_att_wins e' (Attacker_Conjunct p q)›
      using fa_q by blast
    hence ‹weak_spectroscopy_game.weight (Defender_Conj p Q) (Attacker_Conjunct p q) e
        = Some e'›
      by simp
    then have ‹∃ψ. strategy_formula_conjunct (Attacker_Conjunct 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_Conjunct p q)
          = subtract 0 0 1 0 0 0 0 0
      ∧ Some e' = subtract_fn 0 0 1 0 0 0 0 0 e
      ∧ spectro_att_wins e' (Attacker_Conjunct p q)
      ∧ (∃φ. strategy_formula_conjunct (Attacker_Conjunct p q) e' φ
          ∧ expr_pr_conjunct φ ≤ e')›
      using e'_spec by blast
  qed
  hence ‹∃Φ. ∀q ∈ Q.
    spectro_att_wins (e - E 0 0 1 0 0 0 0 0) (Attacker_Conjunct p q)
    ∧ (strategy_formula_conjunct (Attacker_Conjunct 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. spectro_att_wins (e - E 0 0 1 0 0 0 0 0) (Attacker_Conjunct p q)
        ∧ (strategy_formula_conjunct (Attacker_Conjunct 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'. weak_spectroscopy_game.weight (Defender_Stable_Conj p Q) g' e = Some e'
      ∧ spectro_att_wins e' g')
      ∧ ((∃p' q. g' = Attacker_Conjunct p' q) ∨ (∃p' Q'. g' = Defender_Conj p' Q'))›
    by (metis (no_types, opaque_lifting)
          spectroscopy_defender.elims(2,3) spectroscopy_moves.simps(35,36,37,38,74))
  show ?case
  proof(cases ‹Q = {}›)
    case True
    then obtain e' where e'_spec:
      ‹weak_spectroscopy_game.weight (Defender_Stable_Conj p Q) (Defender_Conj p Q) e
        = Some e'›
      ‹e' = e - (E 0 0 0 1 0 0 0 0)›
      ‹spectro_att_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. weak_spectroscopy_game.weight (Defender_Stable_Conj p Q)
          (Attacker_Conjunct p q) e = Some e'
        ∧ spectro_att_wins e' (Attacker_Conjunct 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_Conjunct 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_Conjunct 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_Conjunct 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 force+
qed

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_Conjunct 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)
next
  case (procrastination p Q e χ)
  from this obtain p' where IH:
    ‹spectroscopy_moves (Attacker_Delayed p Q) (Attacker_Delayed p' Q) = id_up ∧
     spectro_att_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(2))
  hence ‹Q ↠S Q ⟶ distinguishes_from (hml_srbb.Internal χ) p Q› using D
    by (smt (verit) 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 ∧
     spectro_att_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))))›
    by (simp, meson D ‹Q ↦aS α Q'› distinguishes_from_def)
  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_Conjunct p q) (Attacker_Delayed q P')
      = Some (λe. Option.bind (subtract_fn 0 0 0 0 0 0 0 1 e) min1_7)›
    ‹spectro_att_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(2) 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') = id_up›
     ‹spectro_att_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 — stable cases
        thus ‹False› using X ‹hml_srbb_inner.distinguishes_from χ p Q'›
          by simp
      next
        case False — unstable cases
        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_defender.simps(4,7)
              spectroscopy_position.distinct(37,41) spectroscopy_position.inject(6))
        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) = id_up›
    ‹spectro_att_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 ∧
     spectro_att_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 ‹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
```

9.3 Correctness Theorem

```
theory Silent_Step_Spectroscopy
  imports
    Distinction_Implies_Winning_Budgets
    Strategy_Formulas
begin
```

We now only combine the results of `Distinction_Implies_Winning_Budgets` and `Strategy_Formulas` to obtain the main characterization theorem of the weak spectroscopy game characterizing a whole spectrum of weak equivalences.

```
context lts_tau
begin

theorem spectroscopy_game_correctness:
  fixes e p Q
  shows ‹(∃φ. distinguishes_from φ p Q ∧ expressiveness_price φ ≤ e)
       ⟷ spectro_att_wins e (Attacker_Immediate p Q)›
proof
  assume ‹∃φ. distinguishes_from φ p Q ∧ expressiveness_price φ ≤ e›
  then obtain φ where φ_spec:
    ‹distinguishes_from φ p Q› ‹expressiveness_price φ ≤ e›
    by blast
  from distinction_implies_winning_budgets φ_spec(1) have
    ‹spectro_att_wins (expressiveness_price φ) (Attacker_Immediate p Q)› .
  thus ‹spectro_att_wins e (Attacker_Immediate p Q)›
    using weak_spectroscopy_game.win_a_upwards_closure φ_spec(2) by simp
next
  assume ‹spectro_att_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›
```

```
    by blast
  thus ‹∃φ. distinguishes_from φ p Q ∧ expressiveness_price φ ≤ e›
    using strategy_formulas_distinguish by fastforce
qed
```

An implicit result of the correctness theorem is that attacker wins on bigger `Q` imply wins on smaller ones.

```
proposition attacker_subet_wins:
  assumes
    ‹spectro_att_wins e (Attacker_Immediate p Q)›
    ‹Q' ⊆ Q›
  shows
    ‹spectro_att_wins e (Attacker_Immediate p Q')›
  using assms spectroscopy_game_correctness
  unfolding distinguishes_from_def subset_iff
  by meson

end

end
```

References

[1] Benjamin Bisping. Process equivalence problems as energy games. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 85–106, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-37706-8_5.

[2] Benjamin Bisping. *Generalized Equivalence Checking of Concurrent Programs*. PhD thesis, Technische Universität Berlin, 2025. URL: https://generalized-equivalence-checking.equiv.io/.

[3] Benjamin Bisping and David N. Jansen. Linear-time–branching-time spectroscopy accounting for silent steps, 2023. arXiv:2305.17671, doi:10.48550/arXiv.2305.17671.

[4] Benjamin Bisping and David N. Jansen. One energy game for the spectrum between branching bisimilarity and weak trace semantics. In Georgiana Caltais and Cinzia Di Giusto, editors, Proceedings Combined 31st International Workshop on *Expressiveness in Concurrency* and 21st Workshop on *Structural Operational Semantics,* Calgary, Canada, 9th September 2024, volume 412, pages 71–88. Open Publishing Association, 2024. doi:10.4204/EPTCS.412.6.

[5] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85, pages 299–309. Springer, Berlin, 1980. doi:10.1007/3-540-10003-2_79.

[6] Rob J. van Glabbeek. The linear time–branching time spectrum: extended abstract. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR'90*, volume 458, pages 278–297. Springer, Berlin, 1990. doi:10.1007/BFb0039066.

[7] Rob J. van Glabbeek. The linear time–branching time spectrum II: The semantics of sequential systems with silent moves; extended abstract. In Eike Best, editor, *CONCUR'93*, volume 715, pages 66–81. Springer, Berlin, 1993. doi:10.1007/3-540-57208-2_6.