Measuring expressive power of HML formulas in Isabelle/HOL

Karl Mattes

24th February 2024

Contents

Chapter 1

Introduction

In this thesis, I show the correspondence between various equivalences popular in the reactive systems community and coordinates of a formula price function, as introduced by Bisping in [Bis23]. I formalized the concepts and proofs discussed in this thesis in the interactive proof assistant Isabelle.

Reactive systems are computing systems that continuously interact with their environment, reacting to external stimuli and producing outputs accordingly [HP85]. At a high level of abstraction, these systems can be seen as collections of interacting processes, where each process represents a state or configuration of the system. Labeled Transition Systems (LTS) [Kel76] provide a formal framework for modeling and analyzing the behavior of reactive systems. Roughly, an LTS is a labeled directed graph, whose nodes correspond to processes and whose edges correspond to transitions between those processes or states.

Verification of these systems involves proving statements regarding the behavior of such a system model. Often, verification tasks aim to show that a system's observed behavior aligns with its intended behavior. That requires a criterion of what constitutes similar behavior on LTS, commonly referred to as the *semantics of equality* of processes. Depending on the requirements of a particular user, many different such criterions have been defined. For a subset of processes, namely the class of concrete sequential processes, [vG01] classified many such semantics. *Sequential* means that the processes can only perform one action at a time. *Concrete* processes are processes in which no internal actions occur, meaning that it exclusively captures the system's interactions with its environment. In such LTS, every transition represents an observable event or action between the system and its environment. The classification in [vG01] involved partially ordering many of these semantics by the relation 'makes strictly more identifications on processes than'. The resulting complete lattice is known as the (infinitary) linear-

time–branching-time spectrum [1] [2]. One way to characterize the behavior of LTS is through the use of modal logics. Formulas of a logic can be seen as describing certain properties of states within an LTS. A commonly used modal logic is Hennessy-Milner logic (HML) [HM85]. Equivalence in terms of HML is determined by whether processes satisfy the same set of formulas. The linear-time–branching-time spectrum can be recharted in terms of the subset relation between these modal-logical characterizations.

In the context of this spectrum, demonstrating that a system model's observed behavior aligns with the behavior of a model of the specification can be done by finding the finest notions of behavioral equivalence that equate them. Special bisimulation games and algorithms capable of answering equivalence questions by performing a 'spectroscopy' of the differences between two processes have been developed [BJN22][Bis23]. These approaches rechart the linear-time–branching-time spectrum using an expressiveness function that assigns a *formula price* to every formula. This price is supposed to capture the expressive capabilities of this particular formula. However, to be sure that these characterizations really capture the desired equivalences one has to perform the proofs.

Contributions

This thesis provides a machine-checkable proof that the price bounds of the expressiveness function expr of [Bis23] correspond to the modal-logical characterizations of named equivalences. More precisely, we consider a formula φ to be in an observation language $\mathcal{O}_{\mathcal{X}}$ iff its price is within the given price bound. For every expressiveness price bound e_X, we derive the sublanguage of Hennessy–Miler logic $\mathcal{O}_{\mathcal{X}}$ and show that a formula φ is in $\mathcal{O}_{\mathcal{X}}$ precisely if its price $\texttt{expr}(\varphi)$ is less than or equal to e_X. Then we show that $\mathcal{O}_{\mathcal{X}}$ has exactly the same distinguishing power as the modal-logical characterization of that equivalence. In (ref Foundations (chapter 2)) we discuss and introduce formal definitions of LTSs, Hennessy-Milner logic and the expressiveness function expr, in (ref The Correspondances?! name!) we perform the proofs for the standard notions of equivalence, i.e. the equivalences of (ref Figure 1). Namely for trace-, failures-, failure-trace-, readiness-, ready-trace-, revivals-, possible-futures-, impossible-futures-, simulation-, ready-simulation-, 2-nested-simulation- and bisimulation semantics. All the main concepts and proofs have been formalized and conducted using the interactive proof assistant Isabelle. More information on Isabelle can be found in (appendix?). We tried to present Isabelle implementations directly after

[1] On Infinity?

[2] Linear time describes identification via the order of events, while branching time captures the branching possibilities in system executions.

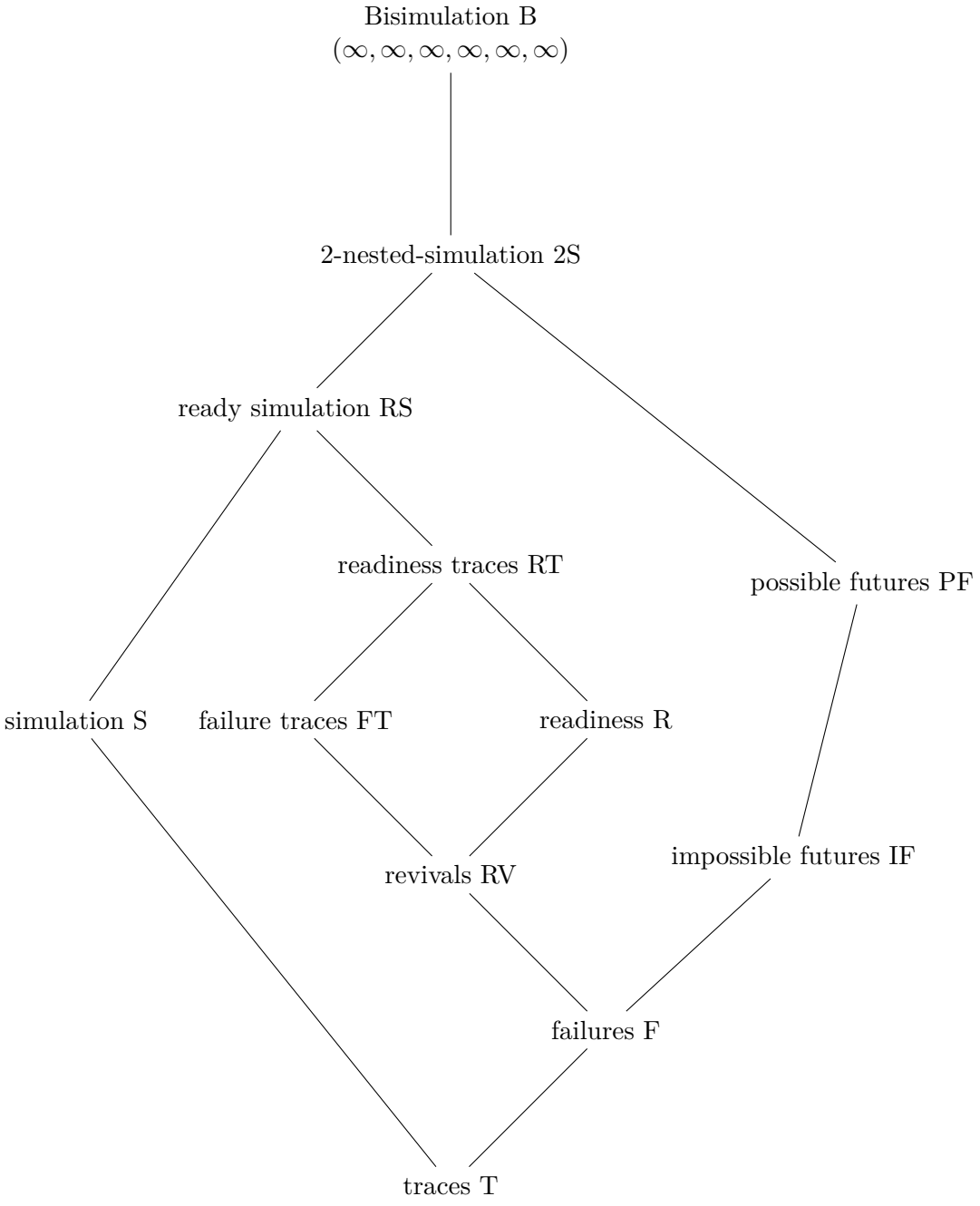


Figure 1.1: TEEEEEEEEEEEEEEEEEEST

their corresponding mathematical definitions. The mathematical definitions are marked as 'definitions' and presented in standard text format. Their corresponding Isabelle implementations are presented right after, distinguished by their `monospaced font` and colored syntax highlighting. However, for readability purposes, a majority of the Isabelle proofs are hidden and replaced by ⟨*proof*⟩ and some lemmas excluded. The whole Isabelle code and a web version of this thesis can be found on Github[3].

[3] Link!!!

Chapter 2

Foundations

In this chapter, relevant concepts will be introduced as well as formalised in Isabelle.

- mention sources (Ben / Max Pohlmann?)

2.1 Labeled Transition Systems

As described in **??**, labeled transition systems are formal models used to describe the behavior of reactive systems. A LTS consists of three components: processes, actions, and transitions. Processes represent momentary states or configurations of a system. Actions denote the events or operations that can occur within the system. The outgoing transitions of each process correspond to the actions the system can perform in that state, yielding a subsequent state. A process may have multiple outgoing transitions labeled with the same or different actions. This signifies that the system can choose any of these transitions non-deterministically [1]. The semantic equivalences treated in [vG01] are defined entirely in terms of action relations. We treat processes as being *sequential*, meaning it can perform at most one action at a time, and instantanious. Note that many modeling methods of systems use a special τ-action to represent internal behavior. However, in our definition of LTS, internal behavior is not considered.

[1] Note that "non-determinism" has been used differently in some of the literature (citation needed). In the context of reactive systems, all transitions are directly triggered by external actions or events and represent synchronization with the environment. The next state of the system is then uniquely determined by its current state and the external action. In that sense the behavior of the system is deterministic.

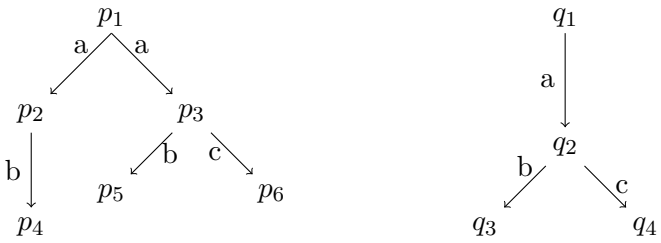
Definition 2.1.1 (Labeled transition Systems)

A *Labeled Transition System* (LTS) is a tuple $\mathcal{S} = (Proc, Act, \rightarrow)$ where $Proc$ is the set of processes, Act is the set of actions and $\cdot \xrightarrow{\cdot} \cdot \subseteq Proc \times Act \times Proc$ is a transition relation. We write $p \xrightarrow{\alpha} p'$ for $(p, \alpha, p') \in \rightarrow$.

Actions and processes are formalized using type variable `'a` and `'s`, respectively. As only actions and states involved in the transition relation are relevant, the set of transitions uniquely defines a specific LTS. We express this relationship using the predicate `tran`. In Isabelle we associate `tran` with a more readable notation, `p` $\mapsto\alpha$ `p'` for $p \xrightarrow{\alpha} p'$.

```
locale lts =
  fixes tran :: <'s ⇒ 'a ⇒ 's ⇒ bool>
    (_ ↦_ _ [70, 70, 70] 80)
begin
```

Example 1 (Taken from (Glabbeeck, counterex. 3)) A simple LTS. Depending on how "close" we look, we might consider the observable behaviors of p_1 and q_2 equivalent or not.



If we compare the states p_1 and q_1 of (ref example 1) we can see many similarities but also differences between their behavior. They can perform the same set of action-sequences, however the p_1 can take a a-transition to p_2 where only a b-transition is possible, while q_1 can only has one a-transition into q_2 where both b and c are possible actions. Abstracting away details of the inner workings of a system leads us to a notion of equivalence that focuses solely on its externally observable behavior, called *trace equivalence*. We can imagine an observer that simply writes down the events of a process as they occur. This observer views two processes as equivalent iff they allow the same sequences of actions. As discussed, p_1 and q_1 are clearly trace-equivalent. Opposite to that we can define a equivalence that also captures internal behavior. *Strong bisimilarity*[2] considers two states equivalent if, for every possible action of one state, there exists a corresponding action of the

[2]Behavioral equivalences are commonly denoted as strong, as opposed to weak, if they do not take internal behavior into account. Since we are only concerned with concrete processes we omit such qualifiers.

other and vice versa. Additionally, the resulting states after taking these actions must also be bisimilar. The states p_1 and q_1 are not bisimilar, since for an a-transition from q_1 to q_2, p_1 can perform an a-transition to p_2 and q_2 and p_2 do not have the same possible actions. Bisimilarity is the finest commonly used *extensional behavioral equivalence*. In extensional equivalences, only observable behavior is taken into account, without considering the identity of the processes. This sets bisimilarity apart from stronger graph equivalences like *graph isomorphism*, where the (intensional) identity of processes is relevant.

We introduce some concepts to better talk about LTS. Note that these Isabelle definitions are only defined in the `context` of LTS.

Definition 2.1.2

The α-*derivatives* of a state refer to the set of states that can be reached with an α-transition:

$$Der(p, \alpha) = \{p' \mid p \xrightarrow{\alpha} p'\}.$$

```
abbreviation derivatives :: <'s ⇒ 'a ⇒ 's set>
  where
<derivatives p α ≡ {p'. p ↦α p'}>
```

The set of *initial actions* of a process p is defined by:

$$I(p) = \{\alpha \in Act \mid \exists p'.p \xrightarrow{\alpha} p'\}$$

```
abbreviation initial_actions:: <'s ⇒ 'a set>
  where
<initial_actions p ≡ {α|α. (∃p'. p ↦α p')}>
```

The step sequence relation $\xrightarrow{\sigma}{}^*$ for $\sigma \in Act^*$ is the reflexive transitive closure of $p \xrightarrow{\alpha} p'$. It is defined recursively by:

$$p \xrightarrow{\varepsilon}{}^* p$$

$$p \xrightarrow{\alpha} p' \text{ with } \alpha \in \text{Act and } p' \xrightarrow{\sigma}{}^* p'' \text{ implies } p' \xrightarrow{\sigma}{}^* p''$$

```
inductive step_sequence :: <'s ⇒ 'a list ⇒ 's ⇒ bool> (<_ ↦$ _ _>[70,70,70]
80) where
<p ↦$ [] p> |
<p ↦$ (a#rt) p''> if <∃p'. p ↦ a p' ∧ p' ↦$ rt p''>
```

p is image-finite if for each $\alpha \in Act$ the set $mathitDer(p, \alpha)$ is finite. An LTS is image-finite if each $p \in Proc$ is image-finite: ”

$$\forall p \in Proc, \alpha \in Act.mathitDer(p, \alpha)$$

is finite.

```
definition image_finite where
‹image_finite ≡ (∀p α. finite (derivatives p α))›
```

nötig?

```
definition image_countable :: ‹bool›
  where ‹image_countable ≡ (∀ p α. countable (derivatives p α))›
```

stimmt definition? definition benötigt nach umstieg auf sets?

```
definition lts_finite where
‹lts_finite ≡ (finite (UNIV :: 's set))›
```

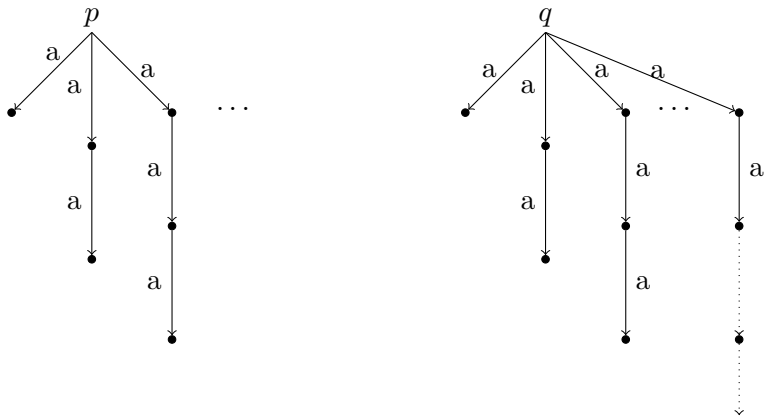
We say that a process is in a *deadlock* if no observation is possible. That is:

$$deadlock(p) = (\forall \alpha. Der(p, \alpha) = \varnothing)$$

```
abbreviation deadlock :: ‹'s ⇒ bool› where
‹deadlock p ≡ (∀α. derivatives p α = {})›

abbreviation relevant_actions :: ‹'a set›
  where
‹relevant_actions ≡ {a. ∃p p'. p ↦a p'}›
```

Example 2 (van Glaabeeck counterex. 1)



Our definition of LTS allows for an unrestricted number of states, all of which can be arbitrarily branching. This means that they have unlimited ways to proceed. Given the possibility of infinity in sequential and branching behavior, we must consider how we identify processes that only differ in their infinite behavior. Take the states p and q of (ref example 2). They have the same (finite) step sequences, however only q has an infinite trace. Do we consider them trace equivalent? We will investigate this further in (Trace Semantics, Simulation?).

```
end
```

2.2 Hennessy–Milner logic

For the purpose of this thesis, we focus on the modal-logical characterizations of equivalences, using Hennessy–Milner logic (HML). First introduced by Matthew Hennessy and Robin Milner (citation), HML is a modal logic for expressing properties of systems described by LTS. Intuitively, HML describes observations on an LTS and two processes are considered equivalent under HML if there exists no observation that distinguishes between them. (citation) defined the modal-logical language as consisting of (finite) conjunctions, negations and a (modal) possibility operator:

$$\varphi ::= tt \mid \varphi_1 \ \wedge \ \varphi_2 \mid \neg\varphi \mid \langle\alpha\rangle\varphi$$

(where α ranges over the set of actions.) The paper also proves that this language characterizes a relation that is effectively the same as bisimilarity. Thsi theorem is called the Hennessy–Milner Theorem and can be expressed as follows: for image-finite LTSs, two processes are bisimilar iff they satisfy the same set of HML formulas. We call this the modal characterisation of bisimilarity. (Infinitary) Hennessy–Milner logic extends the original definition by allowing for conjunction of arbitrary width. This yields the modal characterization of bisimilarity for arbitrary LTS (cite). In (Section Bisimilarity) provide an intuition of the proof along with the Isabelle proof. In the following sections we mean the infinitary version when talking about HML.

Definition 2.2.1 (Hennessy–Milner logic)

Syntax The syntax of Hennessy–Milner logic over a set Σ of actions HML[Σ] is defined by the grammar:

$$\begin{aligned} \varphi ::= \langle a\rangle\varphi & \qquad \text{with } a \in \Sigma \\ \mid \bigwedge_{i\in I} \psi_i & \\ \psi ::= \neg\varphi \mid \varphi. & \end{aligned}$$

Where I denotes an index set.

The data type `('a, 'i)hml` formalizes the definition of HML formulas above. It is parameterized by the type of actions `'a` for Σ and an index type `'i`. We use an index sets of arbitrary type `I :: 'i set` and a mapping `F :: 'i ⇒ ('a, 'i) hml` to formalize conjunctions so that each element of `I` is mapped to a formula[3]

```
datatype ('a, 'i)hml =
```

[3]Note that the formalization via an arbitrary set, i.e. `hml_conj <('a)hml set>` does not yield a valid type, since `set` is not a bounded natural functor.

```
TT |
hml_pos <'a> <('a, 'i)hml> |
hml_conj <'i set> <'i set> <'i ⇒ ('a, 'i) hml>
```

Note that in canonical definitions of HML `TT` is not usually part of the syntax, but is instead synonymous to `⋀{}`. We include `TT` in the definition to enable Isabelle to infer that the type `hml` is not empty.. Corresponding to the mathematical definition, this formalization allows for conjunctions of arbitrary - even of infinite - width.

Semantics The semantics of HML parametrized by Σ (on LTS processes) are given by the relation $\models : (Proc, \text{HML}[\Sigma])$:

$$
\begin{aligned}
p &\models \langle\alpha\rangle\varphi && \text{if there exists } q \text{ such that } q \in Der(p,\alpha) \text{ and } q \models \varphi \\
p &\models \bigwedge_{i\in I} \psi_i && \text{if } p \models \psi_i \text{ for all } i \in I \\
p &\models \neg\varphi && \text{if } p \not\models \varphi
\end{aligned}
$$

```
context lts begin

primrec hml_semantics :: <'s ⇒ ('a, 's)hml ⇒ bool>
(<_ ⊨ _> [50, 50] 50)
where
hml_sem_tt: <(_ ⊨ TT) = True> |
hml_sem_pos: <(p ⊨ (hml_pos α φ)) = (∃ q. (p ↦α q) ∧ q ⊨ φ)> |
hml_sem_conj: <(p ⊨ (hml_conj I J ψs)) = ((∀i ∈ I. p ⊨ (ψs i)) ∧ (∀j
∈ J. ¬(p ⊨ (ψs j))))>
```

A formula that is true for all processes in a LTS can be considered a property that holds universally for the system, akin to a tautology in classical logic.

```
definition HML_true where
HML_true φ ≡ ∀s. s ⊨ φ
⟨proof⟩
```

Two states are HML-equivalent if they satisfy the same formula.

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

HML-equivalence is reflexive, symmetrical and transitive and therefore a valid equivalence.

```
lemma equiv_refl: reflp HML_equivalent
  ⟨proof⟩

lemma equiv_trans: transp HML_equivalent
  ⟨proof⟩
```

```
lemma hml_equiv_sym:
  shows ‹symp HML_equivalent›
  ⟨proof⟩
```

A formula distinguishes one state from another if its true for the first and false for the second.

```
abbreviation distinguishes ::  ‹('a, 's) hml ⇒ 's ⇒ 's ⇒ bool› where
   ‹distinguishes φ p q ≡ p ⊨ φ ∧ ¬ q ⊨ φ›
```

If two states are not HML equivalent then there must be a distinguishing formula.

```
lemma hml_distinctions:
  fixes state:: 's
  assumes ‹¬ HML_equivalent p q›
  shows ‹∃φ. distinguishes φ p q›
⟨proof⟩
```

We can now use HML to capture differences between p_1 and q_1 of (ref Example 1). The formula $\langle a\rangle \bigwedge\{\neg\langle c\rangle\}$ distinguishes p_1 from q_1 and $\langle a\rangle \bigwedge\{\langle c\rangle\}$ distinguishes q_1 from p_1. From the Hennessy–Milner Theorem follows that knowing a distinguishing formula means that p_1 and q_1 are not bisimilar.

⟨*proof*⟩⟨*proof*⟩
end

2.3 Price Spectra of Behavioral Equivalences

The linear-time–branching-time spectrum can be represented in terms of HML-expressiveness (s.h. section HML). (Deciding all at once)(energy games) show how one can think of the amount of HML-expressiveness used by a formula by its *price*. The equivalences of the spectrum (or their modal-logical characterizations) can then be defined in terms of *price coordinates*, that is equivalence X is characterized by the HML formulas with prices less then or equal to a *X-price bound* e_X. We use the six dimensions from (energy games) to characterize the notions of equivalence we are interested in (In figure xx oder so umschreiben). Intuitively, the dimensions can be described as follows:
1. Formula modal depth of observations: How many modal operations $\langle\alpha\rangle$ may one pass when descending the syntax tree. (Algebraic laws for nondeterminism and concurrency)(Operational and algebraic semantics of concurrent processes)
2. Formula nesting depth of conjunctions: How often may one pass a conjunction?
3. Maximal modal depth of deepest positive clauses in conjunctions

4. Maximal modal depth of other positive clauses in conjunctions
5. Maximal modal depth of negative clauses in conjunctions
6. Formula nesting depth of negations

Definition 2.1 (Formula Prices)

The expressiveness price $\mathsf{expr} : \mathrm{HML}[\Sigma] \to (\mathbb{N} \cup \infty)^6$ of a formula interpreted as 6×1-dimensional vectors is defined recursively by:

$$\mathrm{expr}(\langle a \rangle \varphi) := \begin{pmatrix} 1 + \mathrm{expr}_1(\varphi) \\ \mathrm{expr}_2(\varphi) \\ \mathrm{expr}_3(\varphi) \\ \mathrm{expr}_4(\varphi) \\ \mathrm{expr}_5(\varphi) \\ \mathrm{expr}_6(\varphi) \end{pmatrix}$$

$$\mathrm{expr}(\neg \varphi) := \begin{pmatrix} \mathrm{expr}_1(\varphi) \\ \mathrm{expr}_2(\varphi) \\ \mathrm{expr}_3(\varphi) \\ \mathrm{expr}_4(\varphi) \\ \mathrm{expr}_5(\varphi) \\ 1 + \mathrm{expr}_6(\varphi) \end{pmatrix}$$

$$\mathrm{expr}\left(\bigwedge_{i \in I} \psi_i\right) := \sup(\{ \begin{pmatrix} 0 \\ 1 + \sup_{i \in I} \mathrm{expr}_2(\psi_i) \\ \sup_{i \in \mathrm{Pos}} \mathrm{expr}_1(\psi_i) \\ \sup_{i \in \mathrm{Pos} \setminus \mathcal{R}} \mathrm{expr}_1(\psi_i) \\ \sup_{i \in \mathrm{Neg}} \mathrm{expr}_1(\psi_i) \\ 0 \end{pmatrix} \} \cup \{ \mathsf{expr}(\psi_i) | i \in I \}$$

where:
$Neg := \{i \in I \mid \exists \varphi_i'.\psi_i = \neg \varphi_i'\}$
$Pos := I \setminus \mathrm{Neg}$
$\mathcal{R} := \begin{cases} \varnothing \text{ if } Pos = \varnothing, \\ \{r\} \text{ for some } r \in Pos \text{ where } \mathrm{expr}_1(\psi_r) \text{ maximal for } Pos \end{cases}$
Our Isabelle-definition of HML makes it very easy to derive the sets Pos and Neg, by `Φ ` I` and `Φ ` J` respectively.

Remark: Infinity is included in our definition, due to infinite branching conjunctions. Supremum over infinite set wird zu unendlich.

To better argue about the function we define each dimension as a seperate function.

Vlt als erstes: modaltiefe als beispiel für observation expressiveness von formel, mit isabelle definition, dann pos_r definition, direct_expr definition, einzelne dimensionen, lemma direct_expr = expr...

Formally, the *modal depth* expr_1 of a formula φ is defined recursively by:

$$\begin{aligned}
&\text{if } \varphi = \langle a \rangle \psi \quad \text{with } a \in \Sigma \\
&\quad \text{then } \mathsf{expr}_1(\varphi) = 1 + \mathsf{expr}_1(\psi) \\
&\text{if } \varphi = \bigwedge_{i \in I} \{\psi_1, \psi_2, \ldots\} \\
&\quad \text{then } \mathsf{expr}_1(\varphi) = \sup(\mathsf{expr}_1(\psi_i)) \\
&\text{if } \psi = \neg\varphi \\
&\quad \text{then } \mathsf{expr}_1(\psi) = \mathsf{expr}_1(\varphi)
\end{aligned}$$

```
primrec expr_1 :: ('a, 's)hml ⇒ enat
  where
expr_1_tt: ‹expr_1 TT = 0› |
expr_1_conj: ‹expr_1 (hml_conj I J Φ) = Sup ((expr_1 ∘ Φ) ` I ∪ (expr_1
∘ Φ) ` J)› |
expr_1_pos: ‹expr_1 (hml_pos α φ) =
  1 + (expr_1 φ)›
```

With the help of the modal depth we can derive Pos\R in Isabelle:

```
fun pos_r :: ('a, 's)hml set ⇒ ('a, 's)hml set
  where
pos_r xs = (
let max_val = (Sup (expr_1 ` xs));
  max_elem = (SOME ψ. ψ ∈ xs ∧ expr_1 ψ = max_val);
  xs_new = xs - {max_elem}
in xs_new)
```

Now we can directly define the expressiveness function as `direct_expr`.

```
function direct_expr :: ('a, 's)hml ⇒ enat × enat × enat × enat × enat
× enat where
  direct_expr TT = (0, 1, 0, 0, 0, 0) |
  direct_expr (hml_pos α φ) = (1 + fst (direct_expr φ),
                                fst (snd (direct_expr φ)),
                                fst (snd (snd (direct_expr φ))),
                                fst (snd (snd (snd (direct_expr φ)))),

                                fst (snd (snd (snd (snd (direct_expr φ))))),

                                snd (snd (snd (snd (snd (direct_expr φ))))))
|
  direct_expr (hml_conj I J Φ) = (Sup ((fst ∘ direct_expr ∘ Φ) ` I ∪
(fst ∘ direct_expr ∘ Φ) ` J),
```

```
                                    1 + Sup ((fst ∘ snd ∘ direct_expr
∘ Φ) ` I ∪ (fst ∘ snd ∘ direct_expr ∘ Φ) ` J),
(Sup ((fst ∘ direct_expr ∘ Φ) ` I ∪ (fst ∘ snd ∘ snd ∘ direct_expr ∘ Φ)
` I ∪ (fst ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` J)),
(Sup (((fst ∘ direct_expr) ` (pos_r (Φ ` I))) ∪ (fst ∘ snd ∘ snd ∘ snd
∘ direct_expr ∘ Φ) ` I ∪ (fst ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) `
J)),
(Sup ((fst ∘ snd ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` I ∪ (fst ∘ snd
∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` J ∪ (fst ∘ direct_expr ∘ Φ) `
J)),
(Sup ((snd ∘ snd ∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` I ∪ ((eSuc ∘ snd
∘ snd ∘ snd ∘ snd ∘ direct_expr ∘ Φ) ` J))))
  ⟨proof⟩
```

In order to demonstrate termination of the function, it is necessary to establish that each sequence of recursive function calls reaches a base case. This is accomplished by proving that the relation between process-formula pairs, as defined recursively by the function, is contained within a well-founded relation. A relation $R \subset X \times X$ is considered well-founded if every non-empty subset $X' \subset X$ contains a minimal element m such that $(x, m) \notin R$ for all $x \in X'$. A key property of well-founded relations is that all descending chains $(x_0, x_1, x_2, \ldots)$ (where $(x_i, x_{i+1}) \in R$) originating from any element $x_0 \in X$ are finite. Consequently, this ensures that each sequence of recursive invocations terminates after a finite number of steps.

These proofs were inspired by the Isabelle formalizations presented in [WEP+16].

```
inductive_set HML_wf_rel :: (('a, 's)hml) rel where
φ = Φ i ∧ i ∈ (I ∪ J) ⟹ (φ, (hml_conj I J Φ)) ∈ HML_wf_rel |
(φ, (hml_pos α φ)) ∈ HML_wf_rel

lemma HML_wf_rel_is_wf: ‹wf HML_wf_rel›
  ⟨proof⟩

lemma pos_r_subs: pos_r (Φ ` I) ⊆ (Φ ` I)
  ⟨proof⟩

termination
  ⟨proof⟩
```

The other functions are also defined recursively:

Formula nesting depth of conjunctions expr_2:

$$\begin{aligned}
&\text{if } \varphi = \langle a \rangle \psi \quad \text{with } a \in \Sigma \\
&\quad \text{then } \mathsf{expr}_2(\varphi) = \mathsf{expr}_2(\psi) \\
&\text{if } \varphi = \bigwedge_{i \in I} \{\psi_i\} \\
&\quad \text{then } \mathsf{expr}_2(\varphi) = 1 + \sup(\mathsf{expr}_2(\psi_i)) \\
&\text{if } \psi = \neg\varphi \\
&\quad \text{then } \mathsf{expr}_2(\psi) = \mathsf{expr}_2(\varphi)
\end{aligned}$$

```
primrec expr_2 :: ('a, 's)hml ⇒ enat
  where
expr_2_tt: ‹expr_2 TT = 1› |
expr_2_conj: ‹expr_2 (hml_conj I J Φ) = 1 + Sup ((expr_2 ∘ Φ) ` I ∪ (expr_2
∘ Φ) ` J)› |
expr_2_pos: ‹expr_2 (hml_pos α φ) = expr_2 φ›
```

Maximal modal depth of the deepest positive branch expr_3:

$$\begin{aligned}
&\text{if } \varphi = \langle a \rangle \psi \quad \text{with } a \in \Sigma \\
&\quad \text{then } \mathsf{md}(\varphi) = \mathsf{md}(\psi) \\
&\text{if } \varphi = \bigwedge_{i \in I} \{\psi_i\} \\
&\quad \text{then } \mathsf{md}(\varphi) = \sup(\{\mathsf{expr}_1(\psi_i) | i \in \mathrm{Pos}\} \cup \{\mathsf{expr}_3(\psi_i) | i \in I\}) \\
&\text{if } \psi = \neg\varphi \\
&\quad \text{then } \mathsf{expr}_3(\psi) = \mathsf{expr}_3(\varphi)
\end{aligned}$$

```
primrec expr_3 :: ('a, 's) hml ⇒ enat
  where
expr_3_tt: ‹expr_3 TT = 0› |
 expr_3_pos: ‹expr_3 (hml_pos α φ) = expr_3 φ› |
expr_3_conj: ‹expr_3 (hml_conj I J Φ) = (Sup ((expr_1 ∘ Φ) ` I ∪ (expr_3
∘ Φ) ` I ∪ (expr_3 ∘ Φ) ` J))›
```

Maximal modal depth of other positive clauses in conjunctions expr_4:

if $\varphi = \langle a \rangle \psi \quad$ with $a \in \Sigma$
then $\mathsf{expr}_4(\varphi) = \mathsf{expr}_4(\psi)$
if $\varphi = \bigwedge_{i \in I} \{\ \psi_i\}$
then $\mathrm{md}(\varphi) = \sup(\{\mathsf{expr}_1(\psi_i) | i \in \mathrm{Pos} \backslash \mathcal{R}\} \cup \{\mathsf{expr}_4(\psi_i) | i \in I\})$
if $\psi = \neg\varphi$
then $\mathsf{expr}_4(\psi) = \mathsf{expr}_4(\varphi)$

```
primrec expr_4 :: ('a, 's)hml ⇒ enat
  where
expr_4_tt: expr_4 TT = 0 |
expr_4_pos: expr_4 (hml_pos a φ) = expr_4 φ |
expr_4_conj: expr_4 (hml_conj I J Φ) = Sup ((expr_1 ` (pos_r (Φ ` I)))
∪ (expr_4 ∘ Φ) ` I ∪ (expr_4 ∘ Φ) ` J)
```

Maximal modal depth of negative clauses in conjunctions expr_5:

if $\varphi = \langle a \rangle \psi \quad$ with $a \in \Sigma$
then $\mathsf{expr}_5(\varphi) = \mathsf{expr}_5(\psi)$
if $\varphi = \bigwedge_{i \in I} \{\psi_i\}$
then $\mathsf{expr}_5(\varphi) = \sup(\{\mathsf{expr}_1(\psi_i) | i \in \mathrm{Neg}\} \cup \{\mathsf{expr}_5(\psi_i) | i \in I\})$
if $\psi = \neg\varphi$
then $\mathsf{expr}_5(\psi) = \mathsf{expr}_5(\varphi)$

```
primrec expr_5 :: ('a, 's)hml ⇒ enat
  where
expr_5_tt: ‹expr_5 TT = 0› |
expr_5_pos:‹expr_5 (hml_pos α φ) = expr_5 φ›|
expr_5_conj: ‹expr_5 (hml_conj I J Φ) =
(Sup ((expr_5 ∘ Φ) ` I ∪ (expr_5 ∘ Φ) ` J ∪ (expr_1 ∘ Φ) ` J))›
```

Formula nesting depth of negations expr_6:

$$\begin{aligned}
&\text{if } \varphi = \langle a \rangle \psi \quad \text{with } a \in \Sigma \\
&\quad \text{then } \mathsf{expr}_6(\varphi) = \mathsf{expr}_6(\psi) \\
&\text{if } \varphi = \bigwedge_{i \in I} \{\psi_i\} \\
&\quad \text{then } \mathsf{expr}_6(\varphi) = \sup(\{\mathsf{expr}_6(\psi_i) | i \in I\}) \\
&\text{if } \psi = \neg\varphi \\
&\quad \text{then } \mathsf{expr}_6(\psi) = 1 + \mathsf{expr}_6(\varphi)
\end{aligned}$$

```
primrec expr_6 :: ('a, 's)hml ⇒ enat
  where
expr_6_tt: ‹expr_6 TT = 0› |
expr_6_pos: ‹expr_6 (hml_pos α φ) = expr_6 φ›|
expr_6_conj: ‹expr_6 (hml_conj I J Φ) =
(Sup ((expr_6 ∘ Φ) ` I ∪ ((eSuc ∘ expr_6 ∘ Φ) ` J)))›
```

That leaves us with a definition expr of the expressiveness function that is easier to use.

```
fun expr :: ('a, 's)hml ⇒ enat × enat × enat ×  enat × enat × enat

  where
‹expr φ = (expr_1 φ, expr_2 φ, expr_3 φ, expr_4 φ, expr_5 φ, expr_6 φ)›
```

We show that `direct_expr` and `expr` are the same:

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

```
lemma
  shows expr φ = direct_expr φ
```

⟨*proof*⟩

```
context lts
begin
```

Introduce these definitions later?

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

abbreviation all_traces :: 'a list set where
all_traces ≡{tr. ∃p p'. p ↦$ tr p'}

inductive paths:: ‹'s ⇒ 's list ⇒ 's ⇒ bool› where
‹paths p [] p› |
‹paths p (a#as) p''› if ∃α. p ↦ α a ∧ (paths a as p'')

lemma path_implies_seq:
  assumes A1: ∃xs. paths p xs p'
```

```
  shows ∃ys. p ↦$ ys p'
```
⟨*proof*⟩

```
lemma seq_implies_path:
  assumes A1: ∃ys. p ↦$ ys p'
  shows ∃xs. paths p xs p'
```
⟨*proof*⟩

Trace preorder as inclusion of trace sets

```
definition trace_preordered (infix ‹≲T› 60)where
‹trace_preordered p q ≡ traces p ⊆ traces q›
```

Trace equivalence as mutual preorder

```
abbreviation trace_equivalent (infix ‹≃T› 60) where
‹p ≃T q ≡ p ≲T q ∧ q ≲T p›
```

Trace preorder is transitive

```
lemma trace_preorder_transitive:
  shows ‹transp (≲T)›
```
⟨*proof*⟩

```
lemma empty_trace_trivial:
  fixes p
  shows ‹[] ∈ traces p›
```
⟨*proof*⟩

```
lemma ‹equivp (≃T)›
```
⟨*proof*⟩

Failure Pairs

```
abbreviation failure_pairs :: ‹'s ⇒ ('a list × 'a set) set›
  where
‹failure_pairs p ≡ {(xs, F)|xs F. ∃p'. p ↦$ xs p' ∧ (initial_actions
p' ∩ F = {})}›
```

Failure preorder and -equivalence

```
definition failure_preordered (infix ‹≲F› 60) where
‹p ≲F q ≡ failure_pairs p ⊆ failure_pairs q›

abbreviation failure_equivalent (infix ‹≃F› 60) where
‹ p ≃F q ≡ p ≲F q ∧ q ≲F p›
```

Possible future sets

```
abbreviation possible_future_pairs :: ‹'s ⇒ ('a list × 'a list set) set›
  where
‹possible_future_pairs p ≡ {(xs, X)|xs X. ∃p'. p ↦$ xs p' ∧ traces p'
= X}›
```

```
definition possible_futures_preordered (infix ‹≲PF› 60) where
‹p ≲PF q ≡ (possible_future_pairs p ⊆ possible_future_pairs q)›

definition possible_futures_equivalent (infix ‹≃PF› 60) where
‹p ≃PF q ≡ (possible_future_pairs p = possible_future_pairs q)›

lemma PF_trans: transp (≃PF)
  ⟨proof⟩

lemma pf_implies_trace_preord:
  assumes ‹p ≲PF q›
  shows ‹p ≲T q›
  ⟨proof⟩
```

isomorphism

```
definition isomorphism :: ‹('s ⇒ 's) ⇒ bool› where
‹isomorphism f ≡ bij f ∧ (∀p a p'. p ↦ a p' ⟷ f p ↦ a (f p'))›

definition is_isomorphic :: ‹'s ⇒ 's ⇒ bool› (infix ‹≃ISO› 60) where
‹p ≃ISO q ≡ ∃f. isomorphism f ∧ (f p) = q›
```

Two states are simulation preordered if they can be related by a simulation relation. (Implied by isometry.)

```
definition simulation
  where ‹simulation R ≡
    ∀p q a p'. p ↦ a p' ∧ R p q ⟶ (∃q'. q ↦ a q' ∧ R p' q')›

definition simulated_by (infix ‹≲S› 60)
  where ‹p ≲S q ≡ ∃R. R p q ∧ simulation R›
```

Simulation preorder implies trace preorder

```
lemma sim_implies_trace_preord:
  assumes ‹p ≲S q›
  shows ‹p ≲T q›
  ⟨proof⟩
```

Two states are bisimilar if they can be related by a symmetric simulation.

```
definition bisimilar (infix ‹≃B› 80) where
  ‹p ≃B q ≡ ∃R. simulation R ∧ symp R ∧ R p q›
```

Bisimilarity is a simulation.

```
lemma bisim_sim:
  shows ‹simulation (≃B)›
  ⟨proof⟩


end
end
```

```
inductive TT_like :: ('a, 'i) hml ⇒ bool
  where
TT_like TT |
TT_like (hml_conj I J Φ) if (Φ `I) = {} (Φ ` J) = {}

inductive nested_empty_pos_conj :: ('a, 'i) hml ⇒ bool
  where
nested_empty_pos_conj TT |
nested_empty_pos_conj (hml_conj I J Φ)
if ∀x ∈ (Φ `I). nested_empty_pos_conj x (Φ ` J) = {}

inductive nested_empty_conj :: ('a, 'i) hml ⇒ bool
  where
nested_empty_conj TT |
nested_empty_conj (hml_conj I J Φ)
if ∀x ∈ (Φ `I). nested_empty_conj x ∀x ∈ (Φ `J). nested_empty_pos_conj
x

inductive stacked_pos_conj_pos :: ('a, 'i) hml ⇒ bool
  where
stacked_pos_conj_pos TT |
stacked_pos_conj_pos (hml_pos _ ψ) if nested_empty_pos_conj ψ |
stacked_pos_conj_pos (hml_conj I J Φ)
if ((∃φ ∈ (Φ ` I). ((stacked_pos_conj_pos φ) ∧
                    (∀ψ ∈ (Φ ` I). ψ ≠ φ ⟶ nested_empty_pos_conj
ψ))) ∨
   (∀ψ ∈ (Φ ` I). nested_empty_pos_conj ψ))
(Φ ` J) = {}

inductive stacked_pos_conj :: ('a, 'i) hml ⇒ bool
  where
stacked_pos_conj TT |
stacked_pos_conj (hml_pos _ ψ) if nested_empty_pos_conj ψ |
stacked_pos_conj (hml_conj I J Φ)
if ∀φ ∈ (Φ ` I). ((stacked_pos_conj φ) ∨ nested_empty_conj φ)
(∀ψ ∈ (Φ ` J). nested_empty_conj ψ)

inductive stacked_pos_conj_J_empty :: ('a, 'i) hml ⇒ bool
  where
stacked_pos_conj_J_empty TT |
stacked_pos_conj_J_empty (hml_pos _ ψ) if stacked_pos_conj_J_empty ψ
|
stacked_pos_conj_J_empty (hml_conj I J Φ)
if ∀φ ∈ (Φ ` I). (stacked_pos_conj_J_empty φ) Φ ` J = {}

inductive single_pos_pos :: ('a, 'i) hml ⇒ bool
  where
single_pos_pos TT |
```

```
single_pos_pos (hml_pos _ ψ) if nested_empty_pos_conj ψ |
single_pos_pos (hml_conj I J Φ) if
(∀φ ∈ (Φ `I). (single_pos_pos φ))
(Φ ` J) = {}

inductive single_pos :: ('a, 'i) hml ⇒ bool
  where
single_pos TT |
single_pos (hml_pos _ ψ) if nested_empty_conj ψ |
single_pos (hml_conj I J Φ)
if ∀φ ∈ (Φ ` I). (single_pos φ)
∀φ ∈ (Φ ` J). single_pos_pos φ

context lts begin

lemma index_sets_conj_disjunct:
  assumes I ∩ J ≠ {}
  shows ∀s. ¬ (s ⊨ (hml_conj I J Φ))
⟨proof⟩

lemma HML_true_TT_like:
  assumes TT_like φ
  shows HML_true φ
  ⟨proof⟩

lemma HML_true_nested_empty_pos_conj:
  assumes nested_empty_pos_conj φ
  shows HML_true φ
  ⟨proof⟩

end



inductive HML_trace :: ('a, 's)hml ⇒ bool
  where
trace_tt : HML_trace TT |
trace_conj: HML_trace (hml_conj {} {} ψs)|
trace_pos: HML_trace (hml_pos α φ) if HML_trace φ

definition HML_trace_formulas where
HML_trace_formulas ≡ {φ. HML_trace φ}
```

translation of a trace to a formula

```
fun trace_to_formula :: 'a list ⇒ ('a, 's)hml
  where
trace_to_formula [] = TT |
trace_to_formula (a#xs) = hml_pos a (trace_to_formula xs)
```

```
inductive HML_failure :: ('a, 's)hml ⇒ bool
  where
failure_tt: HML_failure TT |
failure_pos: HML_failure (hml_pos α φ) if HML_failure φ |
failure_conj: HML_failure (hml_conj I J ψs)
if (∀i ∈ I. TT_like (ψs i)) ∧ (∀j ∈ J. (TT_like (ψs j)) ∨ (∃α χ. ((ψs
j) = hml_pos α χ ∧ (TT_like χ))))

inductive HML_simulation :: ('a, 's)hml ⇒ bool
  where
sim_tt: HML_simulation TT |
sim_pos: HML_simulation (hml_pos α φ) if HML_simulation φ|
sim_conj: HML_simulation (hml_conj I J ψs)
if (∀x ∈ (ψs ` I). HML_simulation x) ∧ (ψs ` J = {})

definition HML_simulation_formulas where
HML_simulation_formulas ≡ {φ. HML_simulation φ}

inductive HML_readiness :: ('a, 's)hml ⇒ bool
  where
read_tt: HML_readiness TT |
read_pos: HML_readiness (hml_pos α φ) if HML_readiness φ|
read_conj: HML_readiness (hml_conj I J Φ)
if (∀x ∈ (Φ ` (I ∪ J)). TT_like x ∨ (∃α χ. x = hml_pos α χ ∧ TT_like
χ))

inductive HML_impossible_futures ::  ('a, 's)hml ⇒ bool
  where
  if_tt: HML_impossible_futures TT |
  if_pos: HML_impossible_futures (hml_pos α φ) if HML_impossible_futures
φ |
if_conj: HML_impossible_futures (hml_conj I J Φ)
if ∀x ∈ (Φ ` I). TT_like x ∀x ∈ (Φ ` J). (HML_trace x)

inductive HML_possible_futures :: ('a, 's)hml ⇒ bool
  where
pf_tt: HML_possible_futures TT |
pf_pos: HML_possible_futures (hml_pos α φ) if HML_possible_futures φ
|
pf_conj: HML_possible_futures (hml_conj I J Φ)
if ∀x ∈ (Φ ` (I∪ J)). (HML_trace x)

definition HML_possible_futures_formulas where
HML_possible_futures_formulas ≡ {φ. HML_possible_futures φ}

inductive HML_failure_trace :: ('a, 's)hml ⇒ bool
  where
f_trace_tt: HML_failure_trace TT |
f_trace_pos: HML_failure_trace (hml_pos α φ) if HML_failure_trace φ|
```

```
f_trace_conj: HML_failure_trace (hml_conj I J Φ)
if ((∃ψ ∈ (Φ ` I). (HML_failure_trace ψ) ∧ (∀y ∈ (Φ ` I). ψ ≠ y ⟶
nested_empty_conj y)) ∨
(∀y ∈ (Φ ` I). nested_empty_conj y)) ∧
(∀y ∈ (Φ ` J). stacked_pos_conj_pos y)


inductive HML_ready_trace :: ('a, 's)hml ⇒ bool
  where
r_trace_tt: HML_ready_trace TT |
r_trace_pos: HML_ready_trace (hml_pos α φ) if HML_ready_trace φ|
r_trace_conj: HML_ready_trace (hml_conj I J Φ)
if (∃x ∈ (Φ ` I). HML_ready_trace x ∧ (∀y ∈ (Φ ` I). x ≠ y ⟶ single_pos
y))
∨ (∀y ∈ (Φ ` I).single_pos y)
(∀y ∈ (Φ ` J). single_pos_pos y)

inductive HML_ready_sim :: ('a, 's) hml ⇒ bool
  where
HML_ready_sim TT |
HML_ready_sim (hml_pos α φ) if HML_ready_sim φ |
HML_ready_sim (hml_conj I J Φ) if
(∀x ∈ (Φ ` I). HML_ready_sim x) ∧ (∀y ∈ (Φ ` J). single_pos_pos y)

inductive HML_2_nested_sim :: ('a, 's) hml ⇒ bool
  where
HML_2_nested_sim TT |
HML_2_nested_sim (hml_pos α φ) if HML_2_nested_sim φ |
HML_2_nested_sim (hml_conj I J Φ)
if (∀x ∈ (Φ ` I). HML_2_nested_sim x) ∧ (∀y ∈ (Φ ` J). HML_simulation
y)

inductive HML_revivals :: ('a, 's) hml ⇒ bool
  where
revivals_tt: HML_revivals TT |
revivals_pos: HML_revivals (hml_pos α φ) if HML_revivals φ |
revivals_conj: HML_revivals (hml_conj I J Φ) if (∃x ∈ (Φ ` I). (∃α χ.
(x = hml_pos α χ) ∧ TT_like χ) ∧ (∀y ∈ (Φ ` I). x ≠ y ⟶ TT_like y))
∨ (∀y ∈ (Φ ` I).TT_like y)
(∀x ∈ (Φ ` J). TT_like x ∨ (∃α χ. (x = hml_pos α χ) ∧ TT_like χ))

end
theory HML_definitions
imports HML_list
begin

inductive hml_trace :: ('a, 's)hml ⇒ bool where
hml_trace TT |
hml_trace (hml_pos α φ) if hml_trace φ
```

```
inductive hml_failure :: ('a, 's)hml ⇒ bool
  where
failure_tt: hml_failure TT |
failure_pos: hml_failure (hml_pos α φ) if hml_failure φ |
failure_conj: hml_failure (hml_conj I J ψs)
if I = {} (∀j ∈ J. (∃α. ((ψs j) = hml_pos α TT)) ∨ ψs j = TT)

inductive hml_readiness :: ('a, 's)hml ⇒ bool
  where
read_tt: hml_readiness TT |
read_pos: hml_readiness (hml_pos α φ) if hml_readiness φ|
read_conj: hml_readiness (hml_conj I J Φ)
if ∀x ∈ (Φ ` (I ∪ J)). (∃α. x = (hml_pos α TT::('a, 's)hml)) ∨ x = TT

inductive hml_impossible_futures ::  ('a, 's)hml ⇒ bool
  where
  if_tt: hml_impossible_futures TT |
  if_pos: hml_impossible_futures (hml_pos α φ) if hml_impossible_futures
φ |
if_conj: hml_impossible_futures (hml_conj I J Φ)
if I = {} ∀x ∈ (Φ ` J). (hml_trace x)

inductive hml_possible_futures :: ('a, 's)hml ⇒ bool
  where
pf_tt: hml_possible_futures TT |
pf_pos: hml_possible_futures (hml_pos α φ) if hml_possible_futures φ
|
pf_conj: hml_possible_futures (hml_conj I J Φ)
if ∀x ∈ (Φ ` (I∪ J)). (hml_trace x)

definition hml_possible_futures_formulas where
hml_possible_futures_formulas ≡ {φ. hml_possible_futures φ}

inductive hml_failure_trace :: ('a, 's)hml ⇒ bool where
hml_failure_trace TT |
hml_failure_trace (hml_pos α φ) if hml_failure_trace φ |
hml_failure_trace (hml_conj I J Φ)
  if (Φ ` I) = {} ∨ (∃i ∈ Φ ` I. Φ ` I = {i} ∧ hml_failure_trace i)
     ∀j ∈ Φ ` J. ∃α. j = (hml_pos α TT) ∨ j = TT

inductive hml_ready_trace :: ('a, 's)hml ⇒ bool
  where
r_trace_tt: hml_ready_trace TT |
r_trace_pos: hml_ready_trace (hml_pos α φ) if hml_ready_trace φ|
r_trace_conj: hml_ready_trace (hml_conj I J Φ)
if (∃x ∈ (Φ ` I). hml_ready_trace x ∧ (∀y ∈ (Φ ` I). x ≠ y ⟶ (∃α.
y = (hml_pos α TT))))
∨ (∀y ∈ (Φ ` I).(∃α. y = (hml_pos α TT)))
```

```
(∀y ∈ (Φ ` J). (∃α. y = (hml_pos α TT)))

inductive hml_ready_sim :: ('a, 's) hml ⇒ bool
  where
hml_ready_sim TT |
hml_ready_sim (hml_pos α φ) if hml_ready_sim φ |
hml_ready_sim (hml_conj I J Φ) if
(∀x ∈ (Φ ` I). hml_ready_sim x) ∧ (∀y ∈ (Φ ` J). (∃α. y = (hml_pos
α TT)))

inductive hml_2_nested_sim :: ('a, 's) hml ⇒ bool
  where
hml_2_nested_sim TT |
hml_2_nested_sim (hml_pos α φ) if hml_2_nested_sim φ |
hml_2_nested_sim (hml_conj I J Φ)
if (∀x ∈ (Φ ` I). hml_2_nested_sim x) ∧ (∀y ∈ (Φ ` J). HML_simulation
y)

context lts begin

lemma alt_trace_def_implies_trace_def:
  fixes φ :: ('a, 's) hml
  assumes hml_trace φ
  shows ∃ψ. HML_trace ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma trace_def_implies_alt_trace_def:
  fixes φ :: ('a, 's) hml
  assumes HML_trace φ
  shows ∃ψ. hml_trace ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma trace_definitions_equivalent:
  ∀φ. (HML_trace φ ⟶ (∃ψ. hml_trace ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ∀φ. (hml_trace φ ⟶ (∃ψ. HML_trace ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ⟨proof⟩

lemma alt_failure_def_implies_failure_def:
  fixes φ :: ('a, 's) hml
  assumes hml_failure φ
  shows ∃ψ. HML_failure ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma failure_def_implies_alt_failure_def:
  fixes φ :: ('a, 's) hml
  assumes HML_failure φ
  shows ∃ψ. hml_failure ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩
```

```
lemma failure_definitions_equivalent:
  ∀φ. (HML_failure φ ⟶ (∃ψ. hml_failure ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ∀φ. (hml_failure φ ⟶ (∃ψ. HML_failure ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ⟨proof⟩

lemma alt_readiness_def_implies_readiness_def:
  fixes φ :: ('a, 's) hml
  assumes hml_readiness φ
  shows ∃ψ. HML_readiness ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma readiness_def_implies_alt_readiness_def:
  fixes φ :: ('a, 's) hml
  assumes HML_readiness φ
  shows ∃ψ. hml_readiness ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma readiness_definitions_equivalent:
  ∀φ. (HML_readiness φ ⟶ (∃ψ. hml_readiness ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ∀φ. (hml_readiness φ ⟶ (∃ψ. HML_readiness ψ ∧ (s ⊨ ψ ⟷ s ⊨ φ)))
  ⟨proof⟩

lemma alt_impossible_futures_def_implies_impossible_futures_def:
  fixes φ :: ('a, 's) hml
  assumes hml_impossible_futures φ
  shows ∃ψ. HML_impossible_futures ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma impossible_futures_def_implies_alt_impossible_futures_def:
  fixes φ :: ('a, 's) hml
  assumes HML_impossible_futures φ
  shows ∃ψ. hml_impossible_futures ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma alt_failure_trace_def_implies_failure_trace_def:
  fixes φ :: ('a, 's) hml
  assumes hml_failure_trace φ
  shows ∃ψ. HML_failure_trace ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

lemma stacked_pos_rewriting:
  assumes stacked_pos_conj_pos φ ¬HML_true φ
  shows ∃α. (∀s. (s ⊨ φ) ⟷ (s ⊨ (hml_pos α TT)))
  ⟨proof⟩

lemma nested_empty_conj_TT_or_FF:
  assumes nested_empty_conj φ
  shows (∀s. (s ⊨ φ)) ∨ (∀s. ¬(s ⊨ φ))
  ⟨proof⟩
```

```
lemma failure_trace_def_implies_alt_failure_trace_def:
  fixes φ :: ('a, 's) hml
  assumes HML_failure_trace φ
  shows ∃ψ. hml_failure_trace ψ ∧ (∀s. (s ⊨ φ) ⟷ (s ⊨ ψ))
  ⟨proof⟩

end
end
theory HML_equivalences
imports Main
HML_list HML_definitions
begin

context lts begin

definition HML_trace_equivalent where
HML_trace_equivalent p q ≡ (∀ φ. φ ∈ HML_trace_formulas ⟶ (p ⊨ φ)
⟷ (q ⊨ φ))

definition HML_simulation_equivalent :: <'s ⇒ 's ⇒ bool> where
  HML_simulation_equivalent p q ≡
(∀φ. φ ∈ HML_simulation_formulas ⟶ (p ⊨ φ ⟷ q ⊨ φ))

definition HML_possible_futures_equivalent where
HML_possible_futures_equivalent p q ≡ (∀ φ. φ ∈ HML_possible_futures_formulas
⟶ (p ⊨ φ) ⟷ (q ⊨ φ))

definition hml_possible_futures_equivalent where
hml_possible_futures_equivalent p q ≡ (∀ φ. φ ∈ hml_possible_futures_formulas
⟶ (p ⊨ φ) ⟷ (q ⊨ φ))

end
end
```

Bibliography

[Bis23] Benjamin Bisping. Process equivalence problems as energy games, 2023. arXiv:2303.08904.

[BJN22] Benjamin Bisping, David N. Jansen, and Uwe Nestmann. Deciding all behavioral equivalences at once: A game for linear-time–branching-time spectroscopy, 2022. arXiv:2109.15295.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137––161, 1985. doi:10.1145/2455.2460.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and models of concurrent systems*, volume 13, pages 477–498. Springer Berlin Heidelberg, 1985. doi:10.1007/978-3-642-82453-1_17.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976. URL: https://doi.org/10.1145/360248.360251, doi:10.1145/360248.360251.

[vG01] R.J. van Glabbeek. Chapter 1 - the linear time - branching time spectrum i. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier Science, Amsterdam, 2001. doi:https://doi.org/10.1016/B978-044482830-9/50019-9.