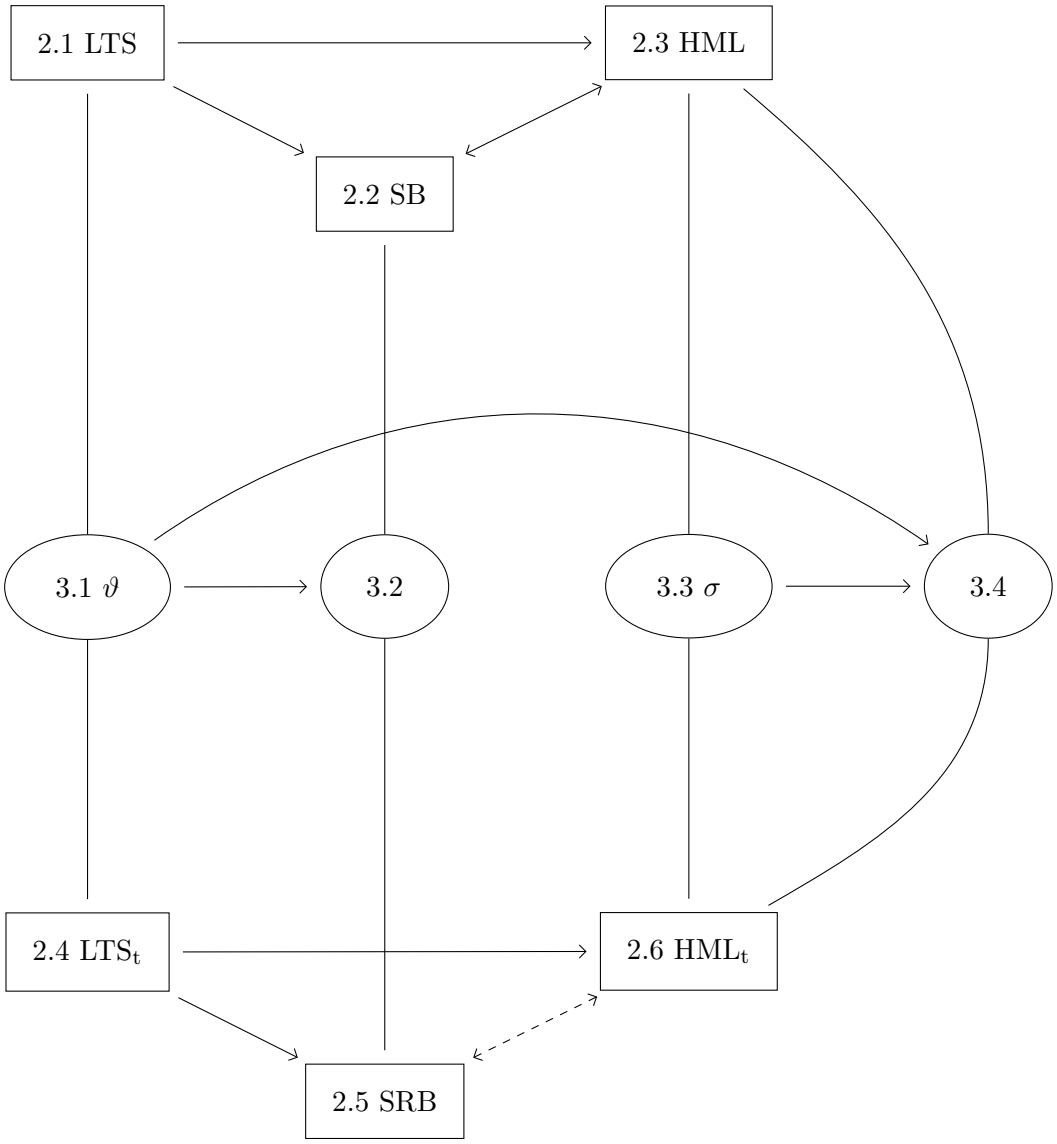
Reducing Reactive Bisimilarity to Strong Bisimilarity

Maximilian Pohlmann

7th June 2021

Contents

Graphical Overview of Main Contents



Chapter 1

Introduction

In this thesis, I show that it is possible to reduce checking strong reactive bisimilarity, as introduced by Rob van Glabbeek in [vG20], to checking ordinary strong bisimilarity. I do this by specifying a mapping that effectively yields a model of the closed system consisting of the original reactive system and its environment. I formalised all concepts discussed in this thesis, and conducted all the proves, in the interactive proof assistant Isabelle.

Reactive systems are systems that continuously interact with their environment (e.g. a user) and whose behaviour is largely dependent on this interaction [HP85]. They can be modelled using labelled transition systems (LTSs) [Kel76]; roughly, an LTS is a labelled directed graph, whose nodes correspond to states of a reactive system and whose edges correspond to transitions between those states.[1]

A user interacting with some system can only perceive it in terms of the interactions it reacts to, i.e. the internal state of the system is hidden from the user. This begets a notion of behavioural/observational equivalence: two non-identical systems can exhibit equivalent behaviour as observed by the user. The simplest such equivalence is known as *strong bisimilarity*.

In classical LTSs, a system cannot react to the absence of interaction, as it would be assumed to simply wait for any interaction. Intuitively, however, a system may be equipped with a clock and perform some activity when it has seen no interaction from the user for a specified time. Such a system would not be describable with classical LTS semantics. Amongst these systems are, e.g., systems implementing mutual exclusion protocols [vG20].

In [vG21], Rob van Glabbeek introduces labelled transition systems with time-outs (LTS_t), which allow for modelling such systems as well. The appertaining equivalence is given in [vG20] as *strong reactive bisimilarity*.

[1] The topics of this thesis are applicable to any such graphs in an abstract way. However, I will continue to use motivations and terminology derived from the interpretation of LTSs as reactive systems.

For the first main result of this thesis, I show that it is possible to reduce checking strong reactive bisimilarity to checking strong bisimilarity. This is in line with reductions of other behavioural equivalences to strong bisimilarity. For example, a strategy used to reduce *weak bisimilarity* to strong bisimilarity is called *saturation* and is described in [AIS11, Section 3.2.5].

The strategy used for reducing reactive bisimilarity to strong bisimilarity is based on the fact that the concept of strong reactive bisimilarity requires an explicit consideration of the environments in which specified systems may exist. Concretely, I specify a mapping from LTS_ts to LTSs, where each state of the mapped LTS corresponds to a state of the original LTS_t in some specific environment.

The reduction of reactive bisimilarity could be of use in the context of automated model checking tools: there are known algorithms for checking equivalences (e.g. see [AIS11]) and tools with efficient implementations thereof;[2] instead of implementing an algorithm for checking strong reactive bisimilarity from scratch, an implementation of the reduction would allow the use of these existing implementations. Moreover, the mapping used for the reduction may aid in the analysis of system specifications utilising LTS_ts, by providing a more explicit view at the system.

Another interesting way to examine the behaviour of an LTS is through the use of modal logics, where formulas describe certain properties and are evaluated on states of an LTS. The modal logic used most commonly in research on reactive systems is known as Hennessy-Milner logic (HML). An extension of HML for evaluation on states of an LTS_t is also given in [vG20]; I will refer to this extension as Hennessy-Milner logic with time-outs (HML_t).

For the second main result of this thesis, I show that it is possible to reduce formula satisfaction of HML_t on LTS_ts to formula satisfaction of HML on LTSs (using another mapping for formulas, along with the mapping from the first reduction).

How This Thesis is Structured / Isabelle

The remainder of this thesis is split into Foundations (chapter 2), where LTSs, bisimilarity, and Hennessy-Milner logic, all without and with time-outs, are discussed and formalised, and The Reductions (chapter 3), where the reduction of bisimilarity and the reduction of formula satisfaction are presented in detail and proved.

All the main topics of this thesis have been formalised, and all proofs conducted, using the interactive proof assistant Isabelle. More information on Isabelle and a short introduction into the most important concepts can be found in appendix A.

[2] e.g. see LTSmin at `github.com/utwente-fmt/ltsmin`

This thesis document itself was generated using the Isabelle document preparation system (see [Wen21a]), which generates LaTeX markup from Isabelle code (and, of course, integrates markup written manually). This allowed me to integrate all the Isabelle code directly into the thesis document. However, almost all proofs are hidden (and replaced simply by ⟨*proof*⟩) and some lemmas excluded. In these cases, an indication of the proof strategy used is given in text. A web version of this thesis, that includes all formalisations, propositions, and proofs, as well as all the text, can be found on GitHub Pages, with one page for each section of this thesis.[3]

All of the sections of chapters 2 and 3 are split into two parts: one containing a prosaic and mathematical description of the topics, and one containing the (documented) formalisation/implementation in Isabelle. I try to clearly distinguish between mathematical structures and their implementation. Although the two are, necessarily, closely related, they are not identical. The former is written in LaTeX math mode in this *italic font*, the latter is Isabelle code in this `monospaced font`.

[3] Not yet.

Chapter 2

Foundations

In this section, the concepts that are relevant for the main part of this thesis will be introduced in text, as well as formalised in Isabelle. The formalisations of sections 2.1 and 2.2 are based on those done by Benjamin Bisping in [Bis18]; the code is available on GitHub.[1] All other formalisations were done as part of this thesis.

2.1 Labelled Transition Systems

A Labelled Transition System (LTS) consists of a set of processes (or states) $Proc$, a set of actions Act, and a relation of transitions $\cdot \xrightarrow{\cdot} \cdot \subseteq Proc \times Act \times Proc$ which directedly connect two processes by an action (the action being the label of the transition) [AILS07]. We call a transition labelled by an action α an *α-transition*.

LTSs can model reactive systems, as discussed in chapter 1. A process of an LTS, then, corresponds to a momentary state of a reactive system. The outgoing transitions of each process correspond to the actions the reactive system could perform in that state (yielding a subsequent process/state), if facilitated by the environment. The choice between the facilitated transitions of a process is non-deterministic.

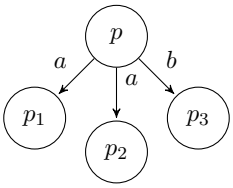
This facilitation by the environment can be thought of as a set of actions that the environment allows in a given moment, or, more intuitively, as a set of simultaneous inputs from the environment to which the system may react. We call the actions not allowed by the environment in a given moment *blocked*.

The environment can also observe which transitions a system performs and change the set of allowed actions in response.

[1] see `coupledsim.bbisping.de/code/`

In classical treatments of LTSs, the actions that the environment allows, blocks, or reacts to, are often considered only implicitly. The reason we put this emphasis on the environment here will become apparent in section 2.4.

Example The process p can perform any of the a-transitions in environments allowing a and the b-transition in environments allowing b. All derivative (subsequent) states cannot perform any transition.



A *hidden action*, denoted by τ, allows for additional semantics: a τ-transition can be performed by a process regardless of the set of actions allowed by the current environment. Depending on the specific semantic context, the performance of a hidden action may also be unobservable (hence the name).

Some Definitions

The α-derivatives of a state are those states that can be reached with one α-transition:

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

An LTS is image-finite iff all derivative sets are finite:

$$\forall p \in Proc, \alpha \in Act.\ Der(p, \alpha) \text{ is finite.}$$

Similarly, we can say an LTS is image-countable iff all derivative sets are countable:

$$\forall p \in Proc, \alpha \in Act.\ Der(p, \alpha) \text{ is countable.}$$

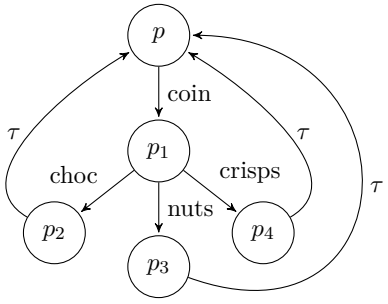
Note on Metavariable usage

States of an LTS range over $p, q, p', q', \ldots$, where p and p' are used for states connected by some transition (i.e. $p \xrightarrow{\alpha} p'$), whereas p and q are used for states possibly related by some equivalence (e.g. $p \leftrightarrow q$), as will be discussed in the next section.

An arbitrary action of an LTS will be referenced by α, whereas an arbitrary *visible* action will be referenced by a.

Practical Example

Before we look at the Isabelle formalisation of LTSs, let us take a detour away from purely theoretical deliberations and consider a real-world machine that may be modelled by an LTS. We can imagine a very simple snack-selling vending machine that accepts only one type of coin and has individual buttons for each of the snacks. When a coin is inserted and a button pressed, the machine dispenses the desired snack and is then ready to accept coins once again. Because the dispensing of the snack itself requires no interaction from the user, we model it as a τ-transition.



Isabelle

In the following Isabelle formalisation of LTSs, the sets of states and actions are denoted by type variables `'s` and `'a`, respectively. A specific LTS on these sets is then determined entirely by its set of transitions, denoted by the predicate `tran`. We can also associate it with a more readable notation (`p ⟼α p'` for $p \xrightarrow{\alpha} p'$).

```
locale lts =
  fixes tran :: ⟨'s ⇒ 'a ⇒ 's ⇒ bool⟩
    ("_ ⟼_ _" [70, 70, 70] 80)
begin
```

The other concepts above can be formalised in a straight-forward manner:

```
abbreviation derivatives :: ⟨'s ⇒ 'a ⇒ 's set⟩
  (⟨Der'(_, _')⟩ [50, 50] 1000)
  where ⟨Der(p, α) ≡ {p'. p ⟼α p'}⟩
```

The following properties concern the entire LTS at hand (given by the locale context) and will be useful when we want to state lemmas that only hold for LTSs that satisfy these properties.

```
definition image_finite :: ⟨bool⟩
```

```
  where ‹image_finite ≡ (∀ p α. finite Der(p, α))›

definition image_countable :: ‹bool›
  where ‹image_countable ≡ (∀ p α. countable Der(p, α))›

end — of locale lts
```

We formalise LTSs with hidden actions as an extension of ordinary LTSs with a fixed action τ.

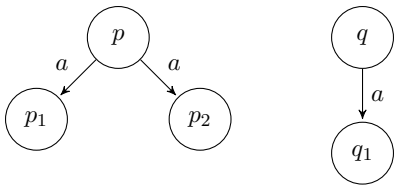
```
locale lts_tau = lts tran
  for tran :: ‹'s ⇒ 'a ⇒ 's ⇒ bool› +
  fixes τ :: ‹'a›
```

2.2 Strong Bisimilarity

As discussed in the previous section, LTSs can describe the behaviour of reactive systems, and this behaviour is observable by the environment (in terms of the transitions performed by the system). This begets a notion of behavioural equivalence, where two processes are said to be behaviourally equivalent if they exhibit the same (observable) behaviour [AILS07].

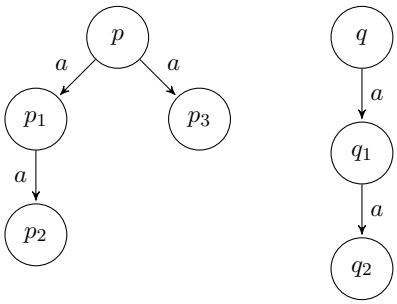
Bisimilarity (or *strong bisimilarity*, to be precise) is the '*finest extensional behavioural equivalence* [...] on processes' [San11a], an extensional property being one that treats the system in question as a black box, i.e. the specific state space of the system remains hidden and performed transitions are only observable in terms of their action-label. This distinguishes bisimilarity from stronger graph equivalences like *graph isomorphism*, where the (intensional) identity of processes (graph nodes) is relevant [San11b].

Example The processes p and q are strongly bisimilar, as both can always perform an a-transition and no transitions afterwards. There is no isomorphism between the left and right subgraphs, as they have a different number of nodes.



It is important to note that not only transitions that are performable, but also those that are not, are relevant.

Example The processes p and q are not strongly bisimilar, as p can perform an a-transition into a subsequent state, where it can perform no further transitions, whereas q can always perform two a-transitions in sequence.



Strong bisimilarity is the *finest* extensional behavioural equivalence, because all actions of an LTS are thought of as observable and blockable. An example of a coarser equivalence is *weak bisimilarity*, which treats the aforementioned hidden action τ as unobservable and unblockable. However, weak bisimilarity is of no further relevance for this thesis and the interested reader is referred to [AILS07, Chapter 3.4].

This notion of strong bisimilarity can be formalised through *strong bisimulation* (SB) relations, introduced originally by David Park in [Par81]. A binary relation $\mathcal{R}$ over the set of processes *Proc* is an SB iff for all $(p, q) \in \mathcal{R}$:

$$\forall p' \in Proc,\ \alpha \in Act.\ p \xrightarrow{\alpha} p' \longrightarrow \exists q' \in Proc.\ q \xrightarrow{\alpha} q' \wedge (p', q') \in \mathcal{R}, \text{ and}$$
$$\forall q' \in Proc,\ \alpha \in Act.\ q \xrightarrow{\alpha} q' \longrightarrow \exists p' \in Proc.\ p \xrightarrow{\alpha} p' \wedge (p', q') \in \mathcal{R}.$$

Isabelle

Strong bisimulations are straightforward to formalise in Isabelle, using the curried function definition approach discussed in appendix A.

```
context lts begin

— strong bisimulation
definition SB :: ⟨('s ⇒ 's ⇒ bool) ⇒ bool⟩
  where ⟨SB R ≡ ∀ p q. R p q ⟶
    (∀ p' α. p ⟼α p' ⟶ (∃ q'. (q ⟼α q') ∧ R p' q')) ∧
    (∀ q' α. q ⟼α q' ⟶ (∃ p'. (p ⟼α p') ∧ R p' q'))⟩
```

Two processes p and q are then strongly bisimilar (written $p \leftrightarrow q$, following [vG20]) iff there is an SB that contains the pair (p, q).

```
definition strongly_bisimilar :: ⟨'s ⇒ 's ⇒ bool⟩
```

```
  (‹_ ↔ _› [70, 70] 70)
  where ‹p ↔ q ≡ ∃ R. SB R ∧ R p q›
```

The following corollaries are immediate consequences of these definitions.

```
corollary strongly_bisimilar_step:
  assumes
    ‹strongly_bisimilar p q›
  shows
    ‹p ⟼a p' ⟹ (∃ q'. (q ⟼a q') ∧ p' ↔ q')›
    ‹q ⟼a q' ⟹ (∃ p'. (p ⟼a p') ∧ p' ↔ q')›
  ⟨proof⟩

corollary strongly_bisimilar_symm:
  assumes ‹p ↔ q›
  shows ‹q ↔ p›
  ⟨proof⟩

end — context lts
```

2.3 Hennessy-Milner Logic

In their seminal paper [HM85], Matthew Hennessy and Robin Milner present a modal-logical characterisation of strong bisimilarity (although they do not call it that), by process properties: 'two processes are equivalent if and only if they enjoy the same set of properties.' These properties are expressed as terms of a modal-logical language, consisting merely of (finite) conjunction, negation, and a family of modal possibility operators. This language is known today as Hennessy-Milner logic (HML), with formulas φ defined by the following grammar (where α ranges over the set of actions Act):

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

The semantics (on LTS processes) is given as follows: all processes satisfy tt, $\varphi_1 \ \wedge \ \varphi_2$ is satisfied if both φ_1 and φ_2 are satisfied, $\neg\varphi$ is satisfied if φ is not satisfied, and $\langle\alpha\rangle\varphi$ is satisfied by a process if it is possible to do an α-transition into a process that satisfies φ.

[HM85] also contains the proof that this modal-logical characterisation of strong bisimilarity coincides with a characterisation that is effectively the same as the one we saw in section 2.2 using strong bisimulations. Although they use different terminology, their result can be summarised as follows: for image-finite LTSs, two processes are strongly bisimilar iff they satisfy the same set of HML formulas. We call this the *modal characterisation* of strong bisimilarity.

Let the *cardinality of conjunction* be the maximally allowed cardinality of sets of formulas conjoined under a conjunction term (for a given variant of HML). For the simple variant above, conjunction has finite cardinality. By allowing for conjunction of arbitrary cardinality (infinitary HML), the modal characterisation of strong bisimilarity can be proved for arbitrary LTSs. This is done in appendix B.

In this section, however, conjunction is constrained to be of countable cardinality, as this turned out to be significantly easier to deal with in the upcoming proofs. The modal characterisation of strong bisimilarity, then, works for LTSs that are image-countable, as we shall see below.

Formulas φ are given by the following grammar, where I ranges over all subsets of the natural numbers $\mathbb{N}$:

$$\varphi ::= \bigwedge_{i\in I} \varphi_i \mid \neg\varphi \mid \langle\alpha\rangle\varphi$$

The semantics of HML formulas on LTSs are as above, with the alteration that a process satisfies $\bigwedge_{i\in I} \varphi_i$ iff it satisfies φ_i for all $i \in I$.

Additional operators can be added as 'syntactic sugar' as follows:

$$\begin{aligned} tt &\equiv \bigwedge_{i\in\emptyset} \varphi_i \\ ff &\equiv \neg tt \\ \bigvee_{i\in I} \varphi_i &\equiv \neg \bigwedge_{i\in I} \neg\varphi_i \end{aligned}$$

Isabelle

Syntax

By definition of countability, all countable sets of formulas can be given by $\{\varphi_i\}_{i\in I} =: \Phi$ for some $I \subseteq \mathbb{N}$ (then $\bigwedge_{i\in I} \varphi_i$ corresponds to $\bigwedge \Phi$). Therefore, the following data type (parameterised by the type of actions `'a`) formalises the definition of HML formulas above (`cset` is the type constructor for countable sets; `acset` and `rcset` are the type morphisms between the types `set` and `cset`; more details below).

I abstained from assigning the constructors a more readable symbolic notation because of the ambiguities and name clashes that would ensue in upcoming sections. The symbolic notations after the constructors below are just code comments.

```
datatype ('a)HML_formula =
  HML_conj  ‹('a)HML_formula cset› — ⋀ Φ
| HML_neg   ‹('a)HML_formula› — ¬φ
| HML_poss  ‹'a› ‹('a)HML_formula› — ⟨α⟩φ
```

The following abbreviations introduce useful constants as syntactic sugar.

abbreviation HML_true :: ⟨('a)HML_formula⟩ — *tt*
  where ⟨HML_true ≡ HML_conj (acset ∅)⟩
abbreviation HML_false :: ⟨('a)HML_formula⟩ — *ff*
  where ⟨HML_false ≡ HML_neg HML_true⟩
abbreviation HML_disj :: ⟨('a)HML_formula cset ⇒ ('a)HML_formula⟩ — $\bigvee \Phi$
  where ⟨HML_disj Φ ≡ HML_neg (HML_conj (cimage HML_neg Φ))⟩

Aside: The Type of Countable Sets

Since sets `set` and countable sets `cset` are different types in Isabelle, they have different membership relation terms. We introduce the following notation for membership of countable sets.

notation cin (⟨_ $\in_c$ _⟩ [100, 100] 100)

The following propositions should clarify how the type constructor `cset` and its morphisms are used. Note how the first proposition requires the assumption `countable X`, whereas the second one does not.

proposition
  fixes X::⟨'x set⟩
  assumes ⟨countable X⟩
  shows ⟨x ∈ X ⟷ x $\in_c$ acset X⟩
  ⟨*proof*⟩
proposition
  fixes X::⟨'x cset⟩
  shows ⟨x $\in_c$ X ⟷ x ∈ rcset X⟩
  ⟨*proof*⟩

Semantics

The semantic satisfaction relation is formalised by the following function. Since the relation is not monotonic (due to negation terms), it cannot be directly defined in Isabelle as an inductive predicate, so we use the `function` command instead. This, then, requires us to prove that the function is well-defined (i.e. the function definition completely and compatibly covers all constructors of our data type) and total (i.e. it terminates). It is easy to see that the former is the case for the function below.

context lts **begin**

function HML_sat :: ⟨'s ⇒ ('a)HML_formula ⇒ bool⟩
  (⟨_ ⊨ _⟩ [50, 50] 50)
  where
    HML_sat_conj: ⟨(p ⊨ HML_conj Φ) = (∀ φ. φ $\in_c$ Φ ⟶ p ⊨ φ)⟩
  | HML_sat_neg: ⟨(p ⊨ HML_neg φ) = (¬ p ⊨ φ)⟩
  | HML_sat_poss: ⟨(p ⊨ HML_poss α φ) = (∃ p'. p ⟼α p' ∧ p' ⊨ φ)⟩
  ⟨*proof*⟩

In order to prove that the function always terminates, we need to show that each sequence of recursive invocations reaches a base case[2] after finitely many steps. We do this by proving that the relation between process-formula pairs given by the recursive definition of the function is (contained within) a well-founded relation. A relation $R \subseteq X \times X$ is called well-founded if each non-empty subset $X' \subseteq X$ has a minimal element m that is not 'R-greater' than any element of X', i.e. $\forall x \in X'.\ (x, m) \notin R$. A property of well-founded relations is that all descending chains $(x_0, x_1, x_2, \dots)$ (with $(x_i, x_{i+1}) \in R$) starting at any element $x_0 \in X$ are finite. This, then, implies that each sequence of recursive invocations terminates after finitely many steps.

```
inductive_set HML_wf_rel :: ‹('s × ('a)HML_formula) rel›
  where
    ‹φ ∈c Φ ⟹ ((p, φ), (p, HML_conj Φ)) ∈ HML_wf_rel›
  | ‹((p, φ), (p, HML_neg φ)) ∈ HML_wf_rel›
  | ‹((p, φ), (p', HML_poss α φ)) ∈ HML_wf_rel›

lemma HML_wf_rel_is_wf: ‹wf HML_wf_rel›
  ⟨proof⟩

termination HML_sat using HML_wf_rel_is_wf
  by (standard, (simp add: HML_wf_rel.intros)+)
```

The semantic clauses for our additional constants are now easily derivable.

```
lemma HML_sat_top:
  shows ‹p ⊨ HML_true›
  ⟨proof⟩
lemma HML_sat_bot:
  shows ‹¬ p ⊨ HML_false›
  ⟨proof⟩
lemma HML_sat_disj:
  shows ‹(p ⊨ HML_disj Φ) = (∃ φ. φ ∈c Φ ∧ p ⊨ φ)›
  ⟨proof⟩
```

Modal Characterisation of Strong Bisimilarity

First, we introduce HML-equivalence as follows.

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

Since formulas are closed under negation, the following lemma holds.

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

[2] For our satisfaction function, the recursive base case is, of course, the empty conjunction, since $\forall \varphi.\ \varphi \in \emptyset \longrightarrow p \vDash \varphi$ is a tautology.

HML-equivalence is clearly symmetrical.

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

We can now formally prove the modal characterisation of strong bisimilarity, i.e.: two processes are HML-equivalent iff they are strongly bisimilar. The proof follows the strategy from [AILS07]. I chose to include these proofs in the thesis document, because they translate quite beautifully, in my opinion, and are not so long as to hamper with the flow of reading.

We show the $\Longrightarrow$-case first, by induction over φ.

```
lemma strong_bisimilarity_implies_HM_equivalence:
  assumes ‹p ↔ q› ‹p ⊨ φ›
  shows ‹q ⊨ φ›
  using assms
proof (induct φ arbitrary: p q)
  case (HML_conj Φ)
  then show ?case
    by (meson HML_sat_conj cin.rep_eq)
next
  case (HML_neg φ)
  then show ?case
    by (meson HML_sat_neg strongly_bisimilar_symm)
next
  case (HML_poss α φ)
  then show ?case
    by (meson HML_sat_poss strongly_bisimilar_step(1))
qed
```

Before we can show the $\Longleftarrow$-case, we need to prove the following lemma: for some binary predicate P, if for every element a of a set A, there exists an element x such that $P(a, x)$ is true, then we can obtain a set X that contains these x (for all $a \in A$) and has the same cardinality as A.

Since more than one x might exist for each a such that $P(a, x)$ is true, the set $\{x \mid a \in A \wedge P(a, x)\}$ might have greater cardinality than A. In order to obtain a set X of same cardinality as A, we need to invoke the axiom of choice in our proof.

```
lemma obtaining_set:
  assumes
    ‹∀ a ∈ A. ∃ x. P a x›
    ‹countable A›
  obtains X where
    ‹∀ a ∈ A. ∃ x ∈ X. P a x›
    ‹∀ x ∈ X. ∃ a ∈ A. P a x›
    ‹countable X›
proof
```

```
  — the SOME operator (Hilbert's selection operator ε) invokes the axiom of choice
  define xm where ‹xm ≡ λ a. SOME x. P a x›
  define X where ‹X ≡ {xm a | a. a ∈ A}›

  show ‹∀a∈A. ∃x∈X. P a x›
    using X_def xm_def assms(1) ⟨proof⟩
  show ‹∀x ∈ X. ∃a∈A. P a x›
    using X_def xm_def assms(1) ⟨proof⟩
  show ‹countable X›
    using X_def xm_def assms(2) ⟨proof⟩
qed
```

We can now show, assuming image-countability of the given LTS, that HML-equivalence is a strong bisimulation. The proof utilises classical contradiction: if HML-equivalence were no strong bisimulation, there would be some processes p and q that are HML-equivalent, with $p \xrightarrow{\alpha} p'$ for some p' (i.e. $p' \in \mathrm{Der}(p, \alpha)$), but for all $q' \in \mathrm{Der}(q, \alpha)$, p' and q' are not HML-equivalent. Then, for each $q' \in \mathrm{Der}(q, \alpha)$, there would be a distinguishing formula $\varphi_{q'}$ which p' satisfies but q' does not. Using our `obtaining_set` lemma, we can obtain the set $\Phi = \{\varphi_{q'}\}_{q' \in \mathrm{Der}(q,\alpha)}$ (which is countable, since $\mathrm{Der}(q, \alpha)$ is countable, by the image-countability assumption). Since we allow for conjunction of countable cardinality, $\bigwedge \Phi$ is a valid formula. By construction, p can make an α-transition into a state that satisfies $\bigwedge \Phi$ (i.e. $p \vDash \langle\alpha\rangle \bigwedge \Phi$), whereas q cannot (i.e. $q \nvDash \langle\alpha\rangle \bigwedge \Phi$). This is a contradiction, since, by assumption, p and q are HML-equivalent. Therefore, HML-equivalence must be a strong bisimulation.

```
lemma HML_equivalence_is_SB:
  assumes
    ‹image_countable›
  shows
    ‹SB HML_equivalent›
proof -
  {
    fix p q p' α
    assume ‹HML_equivalent p q› ‹p ⟼α p'›
    assume ‹∀q' ∈ Der(q, α). ¬ HML_equivalent p' q'›

    hence "exists_φq'": ‹∀q' ∈ Der(q, α). ∃φ. p' ⊨ φ ∧ ¬ q' ⊨ φ›
      using distinguishing_formula by blast

    from ‹image_countable› have ‹countable Der(q, α)›
      using image_countable_def by simp

    from obtaining_set[
          where ?A = ‹Der(q, α)›
            and ?P = ‹λ q' φ. p' ⊨ φ ∧ ¬ q' ⊨ φ›,
          OF "exists_φq'" ‹countable Der(q, α)›]
    obtain Φ where *:
```

```
      ⟨∀φ ∈ Φ. ∃q' ∈ Der(q, α). p' ⊨ φ ∧ ¬ q' ⊨ φ⟩
      ⟨∀q' ∈ Der(q, α). ∃φ ∈ Φ. p' ⊨ φ ∧ ¬ q' ⊨ φ⟩
      ⟨countable Φ⟩
      by (this, blast+)

    have ⟨p ⊨ HML_poss α (HML_conj (acset Φ))⟩
      using ⟨p ⟼α p'⟩ *(1,3) HML_sat.simps(1,3)
        acset_inverse mem_Collect_eq cin.rep_eq
      by metis

    moreover have ⟨¬ q ⊨ HML_poss α (HML_conj (acset Φ))⟩
      using *(2,3) cin.rep_eq
      by fastforce

    ultimately have False
      using ⟨HML_equivalent p q⟩
      unfolding HML_equivalent_def
      by meson
  }

  — We showed the case for p ⟼α p', but not q ⟼α q'.
  — Clearly, this case is covered by the symmetry of HML-equivalence.
  from this show ⟨SB HML_equivalent⟩ unfolding SB_def
    using HML_equivalent_symm by blast
qed
```

We can now conclude the modal characterisation of strong bisimilarity.

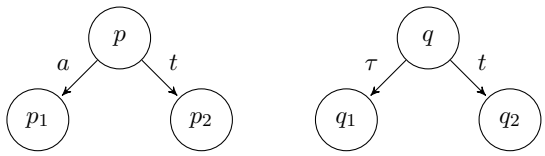
```
theorem modal_characterisation_of_strong_bisimilarity:
  assumes ⟨image_countable⟩
  shows ⟨(p ↔ q)  ⟺  (∀ φ. p ⊨ φ ⟷ q ⊨ φ)⟩
proof
  show ⟨(p ↔ q) ⟹ ∀φ. (p ⊨ φ) = (q ⊨ φ)⟩
    using strong_bisimilarity_implies_HM_equivalence
      strongly_bisimilar_symm
    by blast
next
  show ⟨∀φ. (p ⊨ φ) = (q ⊨ φ) ⟹ (p ↔ q)⟩
    using HML_equivalence_is_SB[OF assms]
      HML_equivalent_def strongly_bisimilar_def
    by blast
qed

end — of context lts
```

2.4 Labelled Transition Systems with Time-Outs

In addition to the hidden action τ, labelled transition systems with time-outs ($\mathrm{LTS_t}$) [vG21] include the *time-out action* t as another special action; t-transitions can only be performed when no other (non–time-out) transition is allowed in a given environment. The rationale is that, in this model, all transitions that are facilitated by or independent of the environment happen instantaneously, and only when no such transition is possible, time elapses and the system is idle. However, since the passage of time is not quantified here, the system does not *have* to take a time-out transition in such a case; instead, the environment can spontaneously change its set of allowed actions (corresponding to a time-out on the part of the environment). Thus, the resolution of an idling period is non-deterministic.

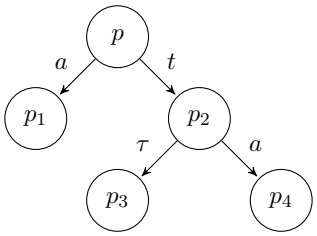
In most works on LTSs, the actions which the environment allows in any given state are usually not modelled explicitly; an (often implicit) requirement for any property of the system is that it should hold for arbitrary (and arbitrarily changing) environments. The introduction of time-outs necessitates an explicit consideration of the environment, as the possibility of a transition does not only depend on whether its action is currently allowed, but potentially on the set of all actions currently allowed by the environment. This is why, in previous sections, I have put unusual emphasis on the actions that are allowed or blocked by the system's environment in a given moment. Henceforth, I will refer to 'environments allowing exactly the actions in X' simply as 'environments X'.

Example The process p can perform an a-transition in environments allowing the action a and a t-transition in environments blocking a. On the other hand, the process q cannot perform a t-transition in any environment, since the τ-transition will always be performed immediately.



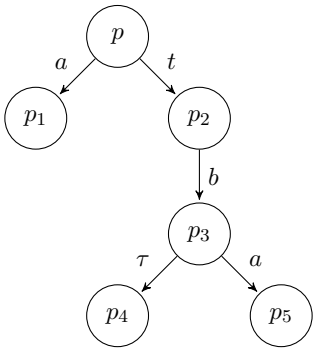
Furthermore, since t-transitions (as well as τ-transitions) are hidden, they cannot trigger a change of the environment, so some states may only ever be entered in certain environments.

Example The process p can perform a t-transition only in environments disallowing a. Therefore, the subsequent state p_2 must be entered in such an environment. The τ-transition is now the only possible transition and will be performed immediately.



On the other hand, transitions with labels other than τ and t require an interaction with the environment, and therefore *are* detectable and can trigger a change in the allowed actions of the environment.

Example Only in environments disallowing a, p can make a t-transition to p_2. However (if b is allowed), the performance of the b-transition into p_3 may trigger a change of the environment, so it is possible that p_3 could perform its a-transition.



These semantics of $\mathrm{LTS_t}$s induce a novel notion of behavioural equivalence, which will be investigated in the next section. For the remainder of this section, we shall formalise $\mathrm{LTS_t}$s, along with some useful concepts.

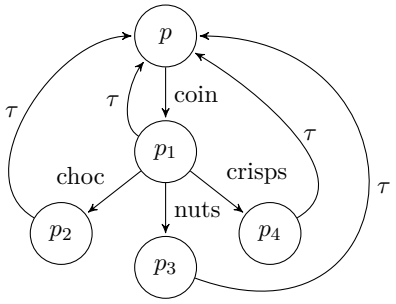
Note on Metavariable usage

If not referenced directly by $\vartheta(p)$ or $\vartheta_X(p)$, arbitrary states of an $\mathrm{LTS_t}$ range over $P, Q, P', Q', \ldots$, where P and P' are used for states connected by some transition (i.e. $P \xrightarrow{\alpha}_{\vartheta} P'$), whereas P and Q are used for states possibly related by some equivalence (e.g. $P \leftrightarrow_r Q$), as will be discussed in the next section.

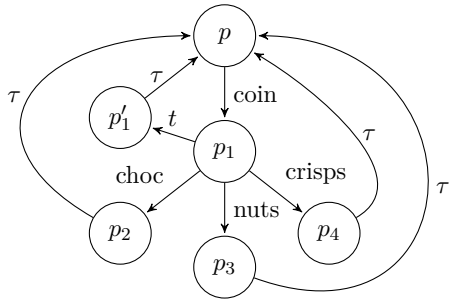
Practical Example

As in section 2.1, we shall consider a real-world machine that may be modelled as an $\mathrm{LTS_t}$. Let us imagine that our simple vending machine ejects

the coin if no snack has been selected after a certain amount of time. We can attempt to model the machine with this extended behaviour as an LTS, where the coin ejection requires no interaction and is therefore also modelled as a τ-transition.



However, this LTS also models a machine that randomly ejects coins right after insertion.[3] In order to distinguish these behaviours, we need LTS_t semantics.



Isabelle

We extend LTSs with hidden actions (`lts_tau`) by the special action `t`. We have to explicitly require (/assume) that $\tau \neq$ `t`; when instantiating the locale `lts_timeout` and specifying a concrete type for the type variable `'a`, this assumption must be (proved to be) satisfied.

```
locale lts_timeout = lts_tau tran τ
  for tran :: "'s ⇒ 'a ⇒ 's ⇒ bool"
    ("_ ⟼_ _" [70, 70, 70] 80)
```

[3] A behaviour that seems to be implemented by the train ticket machines in Berlin.

```
    and τ :: "'a" +
  fixes t :: "'a"
  assumes tau_not_t: ‹τ ≠ t›
begin
```

We define the set of (relevant) visible actions (usually denoted by $A \subseteq Act$) as the set of all actions that are not hidden and that are labels of some transition in the given LTS.

```
definition visible_actions :: ‹'a set›
  where ‹visible_actions
    ≡ {a. (a ≠ τ) ∧ (a ≠ t) ∧ (∃ p p'. p ⟼a p')}›
```

The formalisations in upcoming sections will often involve the type `'a set option`, which has values of the form `None` and `Some X` for some `X :: 'a set`. We will usually use the metavariable `XoN` (for '`X` or `None`'). The following abbreviation will be useful in these situations.

```
abbreviation some_visible_subset :: ‹'a set option ⇒ bool›
  where ‹some_visible_subset XoN
    ≡ ∃ X. XoN = Some X ∧ X ⊆ visible_actions›
```

The initial actions of a process ($\mathcal{I}(p)$ in [vG20]) are the actions for which the process has a transition it can perform immediately (if facilitated by the environment), i.e. it is not a t-transition.

```
definition initial_actions :: ‹'s ⇒ 'a set›
  where ‹initial_actions(p)
    ≡ {α. (α ∈ visible_actions ∨ α = τ) ∧ (∃ p'. p ⟼α p')}›
```

In [vG20], the term $\mathcal{I}(p) \cap (X \cup \{\tau\}) = \emptyset$ is used a lot, which states that there are no immediate transitions the process p can perform (i.e. it is idle) in environments X.

```
abbreviation idle :: ‹'s ⇒ 'a set ⇒ bool›
  where ‹idle p X ≡ initial_actions(p) ∩ (X∪{τ}) = ∅›
```

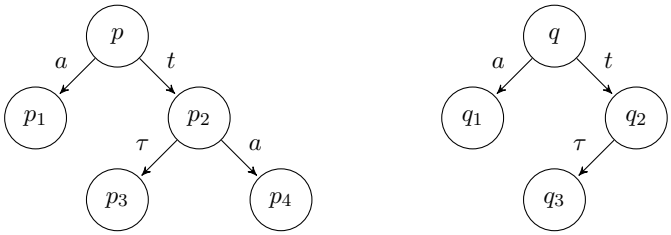
The following corollary is an immediate consequence of this definition.

```
corollary idle_no_derivatives:
  assumes
    ‹idle p X›
    ‹X ⊆ visible_actions›
    ‹α ∈ X∪{τ}›
  shows
    ‹∄ p'. p ⟼α p'›
⟨proof⟩

end — of locale lts_timeout
```

2.5 Reactive Bisimilarity

In the examples of the previous section, we saw that there are LTS_ts with transitions that can never be performed or that can only be performed in certain environments. The behavioural equivalence implied hereby is defined in [vG20] as *strong reactive bisimilarity*.

Example The processes p and q are behaviourally equivalent for LTS_t semantics, i.e. strongly reactive bisimilar.



Strong Reactive Bisimulations

Van Glabbeek introduces several characterisations of this equivalence, beginning with *strong reactive bisimulation* (SRB) relations. These differ from strong bisimulations in that the relations contain not only pairs of processes (p, q), but additionally triples consisting of two processes and a set of actions, (p, X, q). The following definition of SRB relations is quoted, with minor adaptations, from [vG20, Definition 1]:

A *strong reactive bisimulation* is a symmetric relation

$$\mathcal{R} \subseteq (\mathit{Proc} \times \mathcal{P}(A) \times \mathit{Proc}) \cup (\mathit{Proc} \times \mathit{Proc})$$

(meaning that $(p, X, q) \in \mathcal{R} \iff (q, X, p) \in \mathcal{R}$ and $(p, q) \in \mathcal{R} \iff (q, p) \in \mathcal{R}$), such that, for all $(p, q) \in \mathcal{R}$:

1. if $p \xrightarrow{\tau} p'$, then there exists a q' such that $q \xrightarrow{\tau} q'$ and $(p', q') \in \mathcal{R}$,
2. $(p, X, q) \in \mathcal{R}$ for all $X \subseteq A$,

and for all $(p, X, q) \in \mathcal{R}$:

3. if $p \xrightarrow{a} p'$ with $a \in X$, then $\exists q'$ such that $q \xrightarrow{a} q'$ and $(p', q') \in \mathcal{R}$,
4. if $p \xrightarrow{\tau} p'$, then there exists a q' such that $q \xrightarrow{\tau} q'$ and $(p', X, q') \in \mathcal{R}$,
5. if $\mathcal{I}(p) \cap (X \cup \{\tau\}) = \emptyset$, then $(p, q) \in \mathcal{R}$, and

6. if $\mathcal{I}(p) \cap (X \cup \{\tau\}) = \emptyset$ and $p \xrightarrow{t} p'$, then there exists a q' such that $q \xrightarrow{t} q'$ and $(p', X, q') \in \mathcal{R}$.

This definition explicitly formalises the semantics of LTS_ts. We can derive the following intuitions: an environment can either be stable, allowing a specific set of actions, or indeterminate. Indeterminate environments cannot facilitate any transitions, but they can stabilise into arbitrary stable environments. This is expressed by clause 2. Hence, X-bisimilarity is behavioural equivalence in stable environments X, and reactive bisimilarity is behavioural equivalence in indeterminate environments (and thus in arbitrary stable environments).

Since only stable environments can facilitate transitions, there are no clauses involving visible action transitions for $(p, q) \in \mathcal{R}$. However, τ-transitions can be performed regardless of the environment, hence clause 1.

At this point, it is important to discuss what exactly it means for an action to be visible or hidden in this context: as we saw in the last section, the environment cannot react (change its set of allowed actions) when the system performs a τ- or a t-transition, since these are hidden actions. However, since we are talking about a *strong* bisimilarity (as opposed to e.g. weak bisimilarity briefly mentioned in section 2.2), the performance of τ- or t-transitions is still relevant when examining and comparing the behavior of systems.

Therewith, we can look more closely at the remaining clauses: in clause 3, given $(p, X, q) \in \mathcal{R}$, for $p \xrightarrow{a} p'$ with $a \in X$, we require for the 'mirroring' state q' that $(p', q') \in \mathcal{R}$, because a is a visible action and the transition can thus trigger a change of the environment;[4] on the other hand, in clause 4, for $p \xrightarrow{\tau} p'$, and in clause 6, for $p \xrightarrow{t} p'$, we require $(p', X, q') \in \mathcal{R}$, because these actions are hidden and cannot trigger a change of the environment.

Lastly, clause 5 formalises the possibility of the environment timing out (i.e. turning into an indeterminate environment) instead of the system.

These intuitions also form the basis for the process mapping which will be presented in section 3.1.

Strong Reactive/X-Bisimilarity

Two processes p and q are *strongly reactive bisimilar* (written $p \leftrightarrow_r q$) iff there is an SRB containing (p, q), and *strongly X-bisimilar* (written $p \leftrightarrow_r^X q$), i.e. 'equivalent' in environments X, when there is an SRB containing (p, X, q).

[4] This is why van Glabbeek talks about *triggered* environments rather than indeterminate ones. I will use both terms interchangeably.

Generalised Strong Reactive Bisimulations

Another characterisation of reactive bisimilarity uses *generalised strong reactive bisimulation* (GSRB) relations, defined over the same set as SRBs, but with different clauses [vG20, Definition 3]. It is proved that both characterisations do, in fact, characterise the same equivalence.

Isabelle

I first formalise both SRB and GSRB relations (as well as strong reactive bisimilarity, defined by the existence of an SRB, as above), and then replicate the proof of their correspondence.

Strong Reactive Bisimulations

SRB relations are defined over the set

$$(Proc \times \mathcal{P}(A) \times Proc) \cup (Proc \times Proc).$$

As can be easily seen, this set it isomorphic to

$$(Proc \times (\mathcal{P}(A) \cup \{\bot\}) \times Proc),$$

which is a subset of

$$(Proc \times (\mathcal{P}(Act) \cup \{\bot\}) \times Proc).$$

This last set can now be easily formalised in terms of a type, where we formalise $\mathcal{P}(Act) \cup \{\bot\}$ as `'a set option`.
The fact that SRBs are defined using the power set of visible actions (A), whereas our type uses all actions (Act / `'a`), is handled by the first line of the definition below. The second line formalises that symmetry is required by definition. All other lines are direct formalisations of the clauses of the original definition.

context `lts_timeout` **begin**

— strong reactive bisimulation [vG20, Definition 1]
definition `SRB :: ‹('s ⇒ 'a set option ⇒ 's ⇒ bool) ⇒ bool›`
  where `‹SRB R ≡`
  `(∀ p X q. R p (Some X) q ⟶ X ⊆ visible_actions) ∧`
  `(∀ p XoN q. R p XoN q ⟶ R q XoN p) ∧`

  `(∀ p q. R p None q ⟶`
    `(∀ p'. p ⟼τ p' ⟶ (∃ q'. (q ⟼τ q') ∧ R p' None q')) ∧`

```
    (∀ X ⊆ visible_actions. (R p (Some X) q))) ∧

  (∀ p X q. R p (Some X) q ⟶
    (∀ p' a. p ⟼a p' ∧ a ∈ X  ⟶  (∃ q'. (q ⟼a q') ∧
      R p' None q')) ∧
    (∀ p'. p ⟼τ p'  ⟶  (∃ q'. (q ⟼τ q') ∧ R p' (Some X) q')) ∧
    (idle p X  ⟶  R p None q) ∧
    (∀ p'. idle p X ∧ (p ⟼t p')  ⟶  (∃ q'. q ⟼t q' ∧
      R p' (Some X) q')))›
```

Strong Reactive/X-Bisimilarity

Van Glabbeek differentiates between strong reactive bisimilarity ($(p, q) \in \mathcal{R}$ for an SRB $\mathcal{R}$) and strong X-bisimilarity ($(p, X, q) \in \mathcal{R}$ for an SRB $\mathcal{R}$).

```
definition strongly_reactive_bisimilar :: ‹'s ⇒ 's ⇒ bool›
  (‹_ ↔r _› [70, 70] 70)
  where ‹p ↔r q ≡ ∃ R. SRB R ∧ R p None q›

definition strongly_X_bisimilar :: ‹'s ⇒ 'a set ⇒ 's ⇒ bool›
  (‹_ ↔r- _› [70, 70, 70] 70)
  where ‹p ↔r^X q ≡ ∃ R. SRB R ∧ R p (Some X) q›
```

For the upcoming proofs, it is useful to combine both reactive and X-bisimilarity into a single one.

```
definition strongly_reactive_or_X_bisimilar
  :: ‹'s ⇒ 'a set option ⇒ 's ⇒ bool›
  where ‹strongly_reactive_or_X_bisimilar p XoN q
    ≡ ∃ R. SRB R ∧ R p XoN q›
```

Obviously, then, these predicates coincide accordingly.

```
corollary ‹p ↔r q ⟺ strongly_reactive_or_X_bisimilar p None q›
  ⟨proof⟩
corollary ‹p ↔r^X q ⟺ strongly_reactive_or_X_bisimilar p (Some X) q›
  ⟨proof⟩
```

Generalised Strong Reactive Bisimulations

Since GSRBs are defined over the same set as SRBs, the same considerations as above hold.

— generalised strong reactive bisimulation [vG20, Definition 3]

```
definition GSRB :: ‹('s ⇒ 'a set option ⇒ 's ⇒ bool) ⇒ bool›
  where ‹GSRB R ≡
    (∀ p X q. R p (Some X) q  ⟶  X ⊆ visible_actions) ∧
    (∀ p XoN q. R p XoN q  ⟶  R q XoN p) ∧
```

```
(∀ p q. R p None q ⟶
  (∀ p' a. p ⟼a p' ∧ a ∈ visible_actions ∪ {τ}  ⟶
    (∃ q'. q ⟼a q' ∧ R p' None q')) ∧
  (∀ X p'. idle p X ∧ X ⊆ visible_actions ∧ p ⟼t p'  ⟶
    (∃ q'. q ⟼t q' ∧ R p' (Some X) q'))) ∧

(∀ p Y q. R p (Some Y) q ⟶
  (∀ p' a. a ∈ visible_actions ∧ p ⟼a p' ∧ (a∈Y ∨ idle p Y) ⟶
    (∃ q'. q ⟼a q' ∧ R p' None q')) ∧
  (∀ p'. p ⟼τ p'  ⟶
    (∃ q'. q ⟼τ q' ∧ R p' (Some Y) q')) ∧
  (∀ p' X. idle p (X∪Y) ∧ X ⊆ visible_actions ∧ p ⟼t p'  ⟶
    (∃ q'. q ⟼t q' ∧ R p' (Some X) q')))›
```

GSRBs characterise strong reactive/X-bisimilarity

[vG20, Proposition 4] reads (notation adapted): '$p \leftrightarrow_r q$ iff there exists a GSRB $\mathcal{R}$ with $(p, q) \in \mathcal{R}$. Likewise, $p \leftrightarrow_r^X q$ iff there exists a GSRB $\mathcal{R}$ with $(p, X, q) \in \mathcal{R}$.' We shall now replicate the proof of this proposition. First, we prove that each SRB is a GSRB (by showing that each SRB satisfies all clauses of the definition of GSRBs).

lemma `SRB_is_GSRB`:
  assumes ‹`SRB R`›
  shows ‹`GSRB R`›
  ⟨*proof*⟩

Then, we show that each GSRB can be extended to yield an SRB. First, we define this extension. Generally, GSRBs can be smaller than SRBs when proving reactive bisimilarity, because they require triples (p, X, q) only after encountering t-transitions, whereas SRBs require these triples for all processes and environments. These triples (and also some process pairs (p, q) related to environment time-outs) are re-added by this extension.

definition `GSRB_extension`
  :: ‹`('s⇒'a set option⇒'s ⇒ bool)⇒('s⇒'a set option⇒'s ⇒ bool)`›
  where ‹`(GSRB_extension R) p XoN q ≡`
    `(R p XoN q)`
    `∨ (some_visible_subset XoN ∧ R p None q)`
    `∨ ((XoN = None ∨ some_visible_subset XoN)`
      `∧ (∃ Y. R p (Some Y) q ∧ idle p Y))`›

Now we show that this extension does, in fact, yield an SRB (again, by showing that all clauses of the definition of SRBs are satisfied).

lemma `GSRB_extension_is_SRB`:
  assumes
    ‹`GSRB R`›

shows
‹SRB (GSRB_extension R)› (**is** ‹SRB ?R_ext›)
⟨*proof*⟩

Finally, we can conclude the following:

lemma GSRB_whenever_SRB:
shows ‹(∃ R. GSRB R ∧ R p XoN q) ⟷ (∃ R. SRB R ∧ R p XoN q)›
⟨*proof*⟩

This, now, directly implies that GSRBs do charactarise strong reactive/X-bisimilarity.

proposition GSRBs_characterise_strong_reactive_bisimilarity:
‹p ↔$_r$ q ⟷ (∃ R. GSRB R ∧ R p None q)›
using GSRB_whenever_SRB strongly_reactive_bisimilar_def **by** blast

proposition GSRBs_characterise_strong_X_bisimilarity:
‹p ↔$_r{}^X$ q ⟷ (∃ R. GSRB R ∧ R p (Some X) q)›
using GSRB_whenever_SRB strongly_X_bisimilar_def **by** blast

end — of `context lts_timeout`

As a little meta-comment, I would like to point out that van Glabbeek's proof spans a total of five lines ('Clearly, […]', 'It is straightforward to check […]'), whereas the Isabelle proof takes up around 250 lines of code. This just goes to show that for things which are clear and straightforward for humans, it might require quite some effort to 'explain' them to a computer.

2.6 Hennessy-Milner Logic with Time-Outs

In [vG20, Section 3], van Glabbeek extends Hennessy-Milner logic by a family of new modal operators $\langle X\rangle\varphi$, for $X \subseteq A$, as well as additional satisfaction relations $\vDash_X$ for each $X \subseteq A$. Intuitively, $p \vDash \langle X\rangle\varphi$ means that p is idle when placed in an environment X *and* p can perform a t-transition into a state that satisfies φ; $p \vDash_X \varphi$ means that p satisfies φ in environments X.

I call this extension *Hennessy-Milner Logic with Time-Outs* ($\mathrm{HML_t}$) and $\langle X\rangle$ for $X \subseteq A$ the *time-out–possibility operators* (to be distinguished from the ordinary possibility operators $\langle\alpha\rangle$ for $\alpha \in Act$).

The precise semantics are given by the following inductive definition of the satisfaction relation [vG20, Section 3] (notation slightly adapted):

$$
\begin{array}{lll}
p \vDash \bigwedge_{i \in I} \varphi_i & \text{if} & \forall i \in I.\ p \vDash \varphi_i \\
p \vDash \neg\varphi & \text{if} & p \nvDash \varphi \\
p \vDash \langle\alpha\rangle\varphi \quad \text{with } \alpha \in A \cup \{\tau\} & \text{if} & \exists p'.\ p \xrightarrow{\alpha} p' \wedge p' \vDash \varphi \\
p \vDash \langle X\rangle\varphi \quad \text{with } X \subseteq A & \text{if} & \mathcal{I}(p) \cap (X \cup \{\tau\}) = \emptyset \wedge \exists p'.\ p \xrightarrow{t} p' \wedge p' \vDash_X \varphi
\end{array}
$$

$$
\begin{array}{lll}
p \vDash_X \bigwedge_{i \in I} \varphi_i & \text{if} & \forall i \in I.\ p \vDash_X \varphi_i \\
p \vDash_X \neg\varphi & \text{if} & p \nvDash_X \varphi \\
p \vDash_X \langle a\rangle\varphi \quad \text{with } a \in A & \text{if} & a \in X \wedge \exists p'.\ p \xrightarrow{a} p' \wedge p' \vDash \varphi \\
p \vDash_X \langle\tau\rangle\varphi & \text{if} & \exists p'.\ p \xrightarrow{\tau} p' \wedge p' \vDash_X \varphi \\
p \vDash_X \varphi & \text{if} & \mathcal{I}(p) \cap (X \cup \{\tau\}) = \emptyset \wedge p \vDash \varphi
\end{array}
$$

The same intuitions regarding triggered and stable environments as for the definition of strong reactive bisimulations in section 2.2 hold. $\vDash$ expresses that a property holds in triggered environments and $\vDash_X$ that a property holds in environments X. The last clause expresses the possibility of stable environments timing out into triggered environments.

Van Glabbeek then also proves that HML_t characterises strong reactive/X-bisimilarity, i.e. that $p \leftrightarrow_r q \iff (\forall\varphi.\ p \vDash \varphi \longleftrightarrow q \vDash \varphi)$ and $p \leftrightarrow_r^X q \iff (\forall\varphi.\ p \vDash_X \varphi \longleftrightarrow q \vDash_X \varphi)$, where φ are formulas of HML_t. A replication of the proof of this characterisation, however, is not part of this thesis.

Isabelle

The following formalisation is analogous to the one in section 2.3.

```
datatype ('a)HMLt_formula =
  HMLt_conj ‹('a)HMLt_formula cset› — ⋀ Φ
| HMLt_neg ‹('a)HMLt_formula› — ¬φ
| HMLt_poss ‹'a› ‹('a)HMLt_formula› — ⟨α⟩φ
| HMLt_time ‹'a set› ‹('a)HMLt_formula› — ⟨X⟩φ
```

In order to formalise the semantics, I combined both the usual satisfaction relation $\vDash$ and the environment satisfaction relations $\vDash_X$ into one predicate, which is formalised by the function `HMLt_sat` below, where `p` $\Vdash$`?[None]` φ corresponds to $p \vDash \varphi$ and `p` $\Vdash$`?[Some X]` φ corresponds to $p \vDash_X \varphi$.

Note that, in Isabelle code, I use the symbol $\Vdash$ for all satisfaction relations in the context of HML_t, whereas I use $\vDash$ for satisfaction relations in the context of ordinary HML. This notational nuance will be important when we examine the relationship between the satisfaction relations of HML_t and HML in the context of the reduction in section 3.4.

The first four clauses of my formalisation, then, are clearly direct translations of the clauses for the satisfaction relation $\vDash$ quoted above. To see that the

next four clauses do, in fact, correspond to the five clauses for $\vDash_X$ above is less straightforward.

First, each of the four clauses requires that `X` is a subset of the visible actions; in the original definition, the satisfaction relations $\vDash_X$ are only defined for those X to begin with.

Next, the clause for `p ⟙?[Some X] (HMLt_poss α φ)` combines the original clauses for $p \vDash_X \langle a \rangle \varphi$ and $p \vDash_X \langle \tau \rangle \varphi$.

Lastly and most importantly, the last clause of the original definition, stating that $p \vDash_X \varphi$ if p is idle in environments X and $p \vDash \varphi$, is added disjunctively to the cases `p ⟙?[Some X] (HMLt_poss α φ)` and `p ⟙?[Some X] (HMLt_time Y φ)`; the latter case is not part of the original definition and can only be true by virtue of the last clause of the original definition, wherefore this is the only way for the last clause of the function definition below to be true.

I will show below that this is sufficient to assure that my satisfaction function satisfies the last clause of the original definition, i.e. that it does not need to be added disjunctively to the cases `p ⟙?[Some X] (HMLt_conj Φ)` and `p ⟙?[Some X] (HMLt_neg φ)`.

context `lts_timeout` **begin**

function `HMLt_sat :: ‹'s ⇒ 'a set option ⇒ ('a)HMLt_formula ⇒ bool›`

```
  (‹_ ⟙?[_] _› [50, 50, 50] 50)
  where
    ‹(p ⟙?[None] (HMLt_conj Φ)) =
      (∀ φ. φ ∈c Φ ⟶ p ⟙?[None] φ)›
  | ‹(p ⟙?[None] (HMLt_neg φ)) =
      (¬ p ⟙?[None] φ)›
  | ‹(p ⟙?[None] (HMLt_poss α φ)) =
      ((α ∈ visible_actions ∪ {τ}) ∧
        (∃ p'. p ⟼α p' ∧ p' ⟙?[None] φ))›
  | ‹(p ⟙?[None] (HMLt_time X φ)) =
      ((X ⊆ visible_actions) ∧ (idle p X) ∧
        (∃ p'. p ⟼t p' ∧ p' ⟙?[Some X] φ))›

  | ‹(p ⟙?[Some X] (HMLt_conj Φ)) = (X ⊆ visible_actions ∧
      (∀ φ. φ ∈c Φ ⟶ p ⟙?[Some X] φ))›
  | ‹(p ⟙?[Some X] (HMLt_neg φ)) = (X ⊆ visible_actions ∧
      (¬ p ⟙?[Some X] φ))›
  | ‹(p ⟙?[Some X] (HMLt_poss α φ)) = (X ⊆ visible_actions ∧
      (((α ∈ X) ∧ (∃ p'. p ⟼α p' ∧ p' ⟙?[None] φ)) ∨
        ((α = τ) ∧ (∃ p'. p ⟼τ p' ∧ p' ⟙?[Some X] φ)) ∨
        ((idle p X) ∧ (p ⟙?[None] (HMLt_poss α φ)))))›
  | ‹(p ⟙?[Some X] (HMLt_time Y φ)) = (X ⊆ visible_actions ∧
      ((idle p X) ∧ (p ⟙?[None] (HMLt_time Y φ))))›
  ⟨proof⟩
```

The well-founded relation used for the termination proof of the satisfaction function is considerably more difficult due to the last line of the definition containing the same formula on both sides of the implication (as opposed to the other lines of the definition, where the premises only contain subformulas of the formula in the conclusion). This required me to define two relations, prove their well-foundedness separately, and then prove that their union is well-founded using the theorem ⟦`wf R; wf S; R O S ⊆ R`⟧ ⟹ `wf (R ∪ S)` (where `O` is relation composition). Further details have been excluded from the thesis document.

termination `HMLt_sat` ⟨*proof*⟩

We can now introduce the more readable notation (more closely corresponding to the notation in [vG20]) through abbreviations.

abbreviation `HMLt_sat_triggered :: ⟨'s⇒('a)HMLt_formula ⇒ bool⟩`
  `("_ ⊨ _" [50, 50] 50)`
  where ⟨`p ⊨ φ ≡ p ⊨?[None] φ`⟩
abbreviation `HMLt_sat_stable :: ⟨'s⇒'a set⇒('a)HMLt_formula ⇒ bool⟩`
  `("_ ⊨[_] _" [70, 70, 70] 80)`
  where ⟨`p ⊨[X] φ ≡ p ⊨?[Some X] φ`⟩

Lastly, we show (by induction over φ) that the function `HMLt_sat` does indeed satisfy the last clause of the original definition.

proposition
  assumes
    ⟨`X ⊆ visible_actions`⟩
    ⟨`idle p X`⟩
    ⟨`p ⊨ φ`⟩
  shows
    ⟨`p ⊨[X] φ`⟩
  ⟨*proof*⟩

As the last clause of van Glabbeek definition is the main difference to the function definition of `HMLt_sat`, this proposition gives confidence that the original definition and the function definition correspond.

end — of `context lts_timeout`

Chapter 3

The Reductions

In [vG20], van Glabbeek presents various characterisations of reactive bisimilarity, three of which have been presented in previous sections. Another one introduces *environment operators* θ_X (for $X \subseteq A$), which 'place a process in [a *stable*] environment that allows exactly the actions in X to occur' [vG20, section 4]. The precise semantics are given by structural operational rules, e.g.: $p \xrightarrow{\tau} p' \implies \theta_X(p) \xrightarrow{\tau} \theta_X(p')$. However, for the characterisation of reactive bisimilarity, the definition of another kind of relations, namely *time-out bisimulations*, is required.

This inspired me to come up with a mapping (from LTS_ts to LTSs) that explicitly models the entire behaviour of the environment (including triggered environments that may stabilise into arbitrary stable environments) and its interaction with the reactive system. By doing this, the resulting LTS will not be a model of a reactive system, but of the closed system consisting of the original underlying system and its environment, modelling all possible combined states and the transitions between those.

Since the entire semantics of LTS_ts will be incorporated in the mapping, the observable behaviour of the closed system will be equivalent for underlying reactive systems that are strongly reactive bisimilar. In other words: two processes of an LTS_t are strongly reactive bisimilar iff their corresponding processes in the mapped LTS are strongly bisimilar. This will be the main result of section 3.2.

A similar thing can be done for formulas of HML_t. I will present a mapping from HML_t formulas to HML formulas such that a mapped formula holds in a process of a mapped LTS iff the original formula holds in the corresponding process of the original LTS_t. This will be the result of section 3.4.

3.1 A Mapping for Transition Systems

Let $\mathbb{T} = (Proc, Act, \to)$ be an $\mathrm{LTS_t}$. Let $A = Act \setminus \{\tau, t\}$.

In reference to van Glabbeek's θ_X-operators, I introduce a family of operators ϑ_X with similar but not identical semantics. Additionally, I introduce the operator ϑ that places a process in an indeterminate environment.

Furthermore, I introduce a family of special actions ε_X for $X \subseteq A$ that represent a triggered environment stabilising into an environment X, as well as a special action t_ε that represents a time-out of the environment.

We assume that $t_\varepsilon \notin Act$ and $\forall X \subseteq A.\ \varepsilon_X \notin Act$.

Then we define $\mathbb{T}_\vartheta = (Proc_\vartheta, Act_\vartheta, \to_\vartheta)$ with

$$\begin{aligned} Proc_\vartheta &= \{\vartheta(p) \mid p \in Proc\} \cup \{\vartheta_X(p) \mid p \in Proc \wedge X \subseteq A\}, \\ Act_\vartheta &= Act \cup \{t_\varepsilon\} \cup \{\varepsilon_X \mid X \subseteq A\}, \end{aligned}$$

and $\to_\vartheta$ defined by the following rules:

$$(1)\ \frac{}{\vartheta(p) \xrightarrow{\varepsilon_X}_\vartheta \vartheta_X(p)}\ X \subseteq A \qquad (2)\ \frac{p \xrightarrow{\tau} p'}{\vartheta(p) \xrightarrow{\tau}_\vartheta \vartheta(p')}$$

$$(3)\ \frac{p \overset{\alpha}{\not\to} \text{ for all } \alpha \in X \cup \{\tau\}}{\vartheta_X(p) \xrightarrow{t_\varepsilon}_\vartheta \vartheta(p)}$$

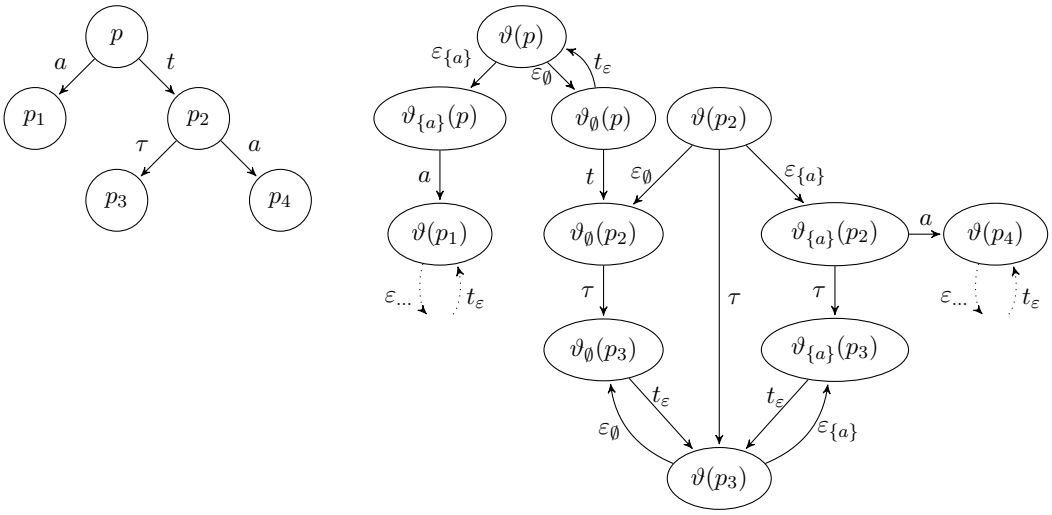
$$(4)\ \frac{p \xrightarrow{a} p'}{\vartheta_X(p) \xrightarrow{a}_\vartheta \vartheta(p')}\ a \in X \qquad (5)\ \frac{p \xrightarrow{\tau} p'}{\vartheta_X(p) \xrightarrow{\tau}_\vartheta \vartheta_X(p')}$$

$$(6)\ \frac{p \overset{\alpha}{\not\to} \text{ for all } \alpha \in X \cup \{\tau\} \quad p \xrightarrow{t} p'}{\vartheta_X(p) \xrightarrow{t}_\vartheta \vartheta_X(p')}$$

These rules are motivated by the intuitions developed in sections 2.4 and 2.5:

1. triggered environments can stabilise into arbitrary stable environments X for $X \subseteq A$,
2. τ-transitions can be performed regardless of the environment,
3. if the underlying system is idle, the environment may time-out and turn into an indeterminate/triggered environment,
4. facilitated visible transitions can be performed and can trigger a change in the environment,
5. τ-transitions cannot be observed by the environment and hence cannot trigger a change,
6. if the underlying system is idle and has a t-transition, the transition may be performed and is not observable by the environment.

Example The LTS_t on the left (with $Act = \{a, \tau, t\}$) gets mapped to the LTS on the right. States that have no incoming or outgoing transitions other than ε_X or t_ε are omitted. Note how $\vartheta(T)$ is not reachable from $\vartheta(P)$.



Isabelle

Formalising $Proc_\vartheta$ and Act_ϑ

For the formalisation, we specify another type of LTS based on `lts_timeout`, where the aforementioned special actions and operators are considered; we call it `lts_timeout_mappable`. Since $Proc \cap Proc_\vartheta = \emptyset$, we introduce a new type variable `'ss` for $Proc_\vartheta$, but use `'a` for both Act and Act_ϑ. We formalise the family of special actions ε_X as a mapping ε`[_] :: 'a set ⇒ 'a`, and the environment operators ϑ/ϑ_X as a single mapping ϑ`?[_](_) :: 'a set option ⇒ 's ⇒ 'ss`.

As for `lts_timeout` in section 2.4, we require that no two special actions are equal, formalised by the first set of assumptions `distinctness_special_actions`.

As an operator, the term $\vartheta_X(p)$ simply refers to the specific state p in an environment X; when understood as a mapping, we have to be more careful, since ϑ`?[Some X](p)` is now a state itself. Specifically, we have to assume that the mapping ϑ`?[_](_)` is injective (when restricted to domains where `X ⊆ visible_actions`). Otherwise, we might have that ϑ`?[None](p) = ϑ?[None](q)` for `p ≠ q`, which is obviously problematic. Hence, the (restricted) injectivity of ϑ`?[_](_)` is formalised as the set of assumptions `injectivity_theta`. (We do not require injectivity for subsets of the domain with `¬ X ⊆ visible_actions`, since ϑ_X is not defined for those X).

The same is required for the mapping ε`[_]`, as formalised in the last clause of the set of assumptions `distinctness_special_actions` (the (restricted) injectivity of ε`[_]` is part of the requirement that all special actions must be distinct). Again, we only require injectivity for the mapping restricted to the domain `visible_actions`. If we would require that ε`[_] :: 'a set ⇒ 'a` were injective over its entire domain `'a set`, we would run into problems, since such a function cannot exist by Cantor's theorem.

For now, we assume that such mappings exist, in order to proof the general case for any such mappings. In appendix C, I give examples for these mappings and show that, with these, any `lts_timeout` can be interpreted as an `lts_timeout_mappable`.

Lastly, we formalise our requirements $t_\varepsilon \notin Act$ and $\forall X \subseteq A.\ \varepsilon_X \notin Act$ as the last set of assumptions `no_epsilon_in_tran`. Technically, these assumptions only state that the ε-actions do not label any transition of $\mathbb{T}$. However, we can assume that $Act = \{\alpha \mid \exists p, p'.\ p \xrightarrow{\alpha} p'\}$, since actions that do not label any transitions are not relevant to the behaviour of an LTS.

```
locale lts_timeout_mappable = lts_timeout tran τ t
  for tran :: "'s ⇒ 'a ⇒ 's ⇒ bool"
      ("_ ⟼_ _" [70, 70, 70] 80)
    and τ :: 'a
    and t :: 'a +
  fixes t_ε :: 'a
    and stabilise :: ‹'a set ⇒ 'a›
      (‹ε[_]›)
    and in_env :: ‹'a set option ⇒ 's ⇒ 'ss›
      (‹ϑ?[_]'(_')›)
  assumes
    distinctness_special_actions:
    ‹τ ≠ t› ‹τ ≠ t_ε› ‹t ≠ t_ε›
    ‹ε[X] ≠ τ› ‹ε[X] ≠ t› ‹ε[X] ≠ t_ε›
    ‹X ⊆ visible_actions ⟹ ε[X] = ε[Y] ⟹ X = Y›
    and

    injectivity_theta:
    ‹ϑ?[None](p) ≠ ϑ?[Some X](q)›
    ‹(ϑ?[None](p) = ϑ?[None](q)) ⟶ p = q›
    ‹X ⊆ visible_actions ⟹
      (ϑ?[Some X](p) = ϑ?[Some Y](q)) ⟶ X = Y ∧ p = q›
    and

    no_epsilon_in_tran:
    ‹¬ p ⟼ε[X] q›
    ‹¬ p ⟼t_ε q›
begin
```

We can now define abbreviations with notations that correspond more closely to our operators defined above.

abbreviation triggered_env :: ‹'s ⇒ 'ss›
  (‹ϑ'(_')›)
  where ‹ϑ(p) ≡ ϑ?[None](p)›
abbreviation stable_env :: ‹'a set ⇒ 's ⇒ 'ss›
  (‹ϑ[_]'(_')›)
  where ‹ϑ[X](p) ≡ ϑ?[Some X](p)›

With four special actions (three plus a family), the following abbreviation will be convenient later on.

abbreviation no_special_action :: ‹'a ⇒ bool›
  where ‹no_special_action α
    ≡ α ≠ τ ∧ α ≠ t ∧ α ≠ t_ε ∧ (∀ X. α ≠ ε[X])›

Formalising $\rightarrow_\vartheta$

We now formalise the transition relation of our mapping, given above by the structural operational rules. We use the notation _ ⟼ϑ _ _ instead of the more obvious _ ⟼ϑ _ _ simply because of better readability.

It should be easy to see that the clauses below correspond directly to the rules above. Like in previous sections, we have to take extra care to handle the requirement X ⊆ visible_actions.

We use the **inductive** command, because this allows us to define separate individual (as opposed to the **definition** command). Technically speaking, this inductive definition only has base cases, though, since no premise involves ⟼ϑ.

inductive tran_theta :: ‹'ss ⇒ 'a ⇒ 'ss ⇒ bool›
  (‹_ ⟼ϑ _ _› [70, 70, 70] 70)
  where
    env_stabilise: ‹X ⊆ visible_actions ⟹
      ϑ(p) ⟼ϑ ε[X] ϑ[X](p)›
  | triggered_tau:
      ‹p ⟼τ q ⟹ ϑ(p) ⟼ϑ τ ϑ(q)›
  | env_timeout: ‹X ⊆ visible_actions ⟹
      idle p X ⟹ ϑ[X](p) ⟼ϑ t_ε ϑ(p)›
  | tran_visible: ‹X ⊆ visible_actions ⟹
      a ∈ X ⟹ p ⟼a q ⟹ ϑ[X](p) ⟼ϑ a ϑ(q)›
  | stable_tau: ‹X ⊆ visible_actions ⟹
      p ⟼τ q ⟹ ϑ[X](p) ⟼ϑ τ ϑ[X](q)›
  | sys_timeout: ‹X ⊆ visible_actions ⟹
      idle p X ⟹ p ⟼t q ⟹ ϑ[X](p) ⟼ϑ t ϑ[X](q)›

Generation Lemmas

For the remainder of this section, we will derive a set of generation lemmas, i.e. lemmas that allow us to reason backwards: if we know P ⟼ϑ α P' and

some other information about P and/or α, we can deduce some information about the other variables as well as the transitions of the original LTS.

lemma generation_triggered_transitions:
assumes ‹ϑ(p) $\longmapsto^{\vartheta}\alpha$ P'›
shows ‹(∃X. α = ε[X] ∧ P' = ϑ[X](p) ∧ X ⊆ visible_actions)
∨ (α = τ ∧ (∃p'. p $\longmapsto\tau$ p'))›
⟨*proof*⟩

lemma generation_stable_transitions:
assumes ‹ϑ[X](p) $\longmapsto^{\vartheta}\alpha$ P'›
shows ‹α = t_ε ∨ (∃ p'. p $\longmapsto\alpha$ p' ∧ (α ∈ X ∨ α = τ ∨ α = t))›
⟨*proof*⟩

lemma generation_env_stabilise:
assumes ‹P $\longmapsto^{\vartheta}\varepsilon$[X] P'›
shows ‹∃ p. P = ϑ(p) ∧ P' = ϑ[X](p)›
⟨*proof*⟩

lemma generation_triggered_tau:
assumes ‹ϑ(p) $\longmapsto^{\vartheta}\tau$ P'›
shows ‹∃ p'. P' = ϑ(p') ∧ p $\longmapsto\tau$ p'›
⟨*proof*⟩

lemma generation_env_timeout:
assumes ‹ϑ[X](p) $\longmapsto^{\vartheta}$t_ε P'›
shows ‹P' = ϑ(p) ∧ idle p X›
⟨*proof*⟩

lemma generation_tran_visible:
assumes ‹ϑ[X](p) $\longmapsto^{\vartheta}$a P'› ‹a ∈ visible_actions›
shows ‹a ∈ X ∧ (∃ p'. P' = ϑ(p') ∧ p $\longmapsto$a p')›
⟨*proof*⟩

lemma generation_stable_tau:
assumes ‹ϑ[X](p) $\longmapsto^{\vartheta}\tau$ P'›
shows ‹∃ p'. P' = ϑ[X](p') ∧ p $\longmapsto\tau$ p'›
⟨*proof*⟩

lemma generation_sys_timeout:
assumes ‹ϑ[X](p) $\longmapsto^{\vartheta}$t P'›
shows ‹∃ p'. P' = ϑ[X](p') ∧ idle p X ∧ p $\longmapsto$t p'›
⟨*proof*⟩

end — of locale lts_timeout_mappable

3.2 Reduction of Bisimilarity

The main result of this section will be that two processes p and q of an $\mathrm{LTS_t}$ $\mathbb{T}$ are strongly reactive bisimilar (strongly X-bisimilar) iff the corresponding processes $\vartheta(p)$ and $\vartheta(q)$ ($\vartheta_X(p)$ and $\vartheta_X(q)$) of $\mathbb{T}_\vartheta$ are strongly bisimilar.

We show the $\Longrightarrow$-direction first. For an SRB $\mathcal{R}$, let

$$\mathcal{S} = \{(\vartheta(p), \vartheta(q)) \mid (p, q) \in R\} \cup \{(\vartheta_X(p), \vartheta_X(q)) \mid (p, X, q) \in R\}.$$

We can prove that $\mathcal{S}$ is an SB, by showing that the mapping satiesfies all clauses of the definition of SBs, using the fact that $\mathcal{R}$ is an SRB as well as the rules and generation lemmas for $\rightarrow_\vartheta$. Hence, the existence of an SRB $\mathcal{R}$ with $(p, q) \in R$ implies the existence of an SB $\mathcal{S}$ with $(\vartheta(p), \vartheta(q)) \in \mathcal{S}$ (and similarly for ϑ_X), so strong reactive/X-bisimilarity in $\mathbb{T}$ implies strong bisimilarity in $\mathbb{T}_\vartheta$.

Next, we show the $\Longleftarrow$-direction. Let

$$\mathcal{R} = \{(p, q) \mid \vartheta(p) \leftrightarrow \vartheta(q)\} \cup \{(p, X, q) \mid \vartheta_X(p) \leftrightarrow \vartheta_X(q)\}.$$

We can prove that $\mathcal{R}$ is an SRB, again, by showing that all clauses of the definition are satisfied. Hence, strong bisimilarity of $\vartheta(p)$ and $\vartheta(q)$ implies the existence of an SRB $\mathcal{R}$ with $(p, q) \in \mathcal{R}$ (and similarly for ϑ_X), so strong bisimilarity in $\mathbb{T}_\vartheta$ implies strong reactive/X-bisimilarity in $\mathbb{T}$.

Thus, we have that strong reactive/X-bisimilarity in $\mathbb{T}$ corresponds to strong bisimilarity in $\mathbb{T}_\vartheta$.

Isabelle

We begin by *interpreting* our transition mapping `tran_theta` as an `lts` and call it `lts_theta`. Therefore, we are handling two separate LTSs: the $\mathrm{LTS_t}$ $\mathbb{T}$ given by the local `context lts_timeout_mappable`, and the LTS $\mathbb{T}_\vartheta$ given by the `interpretation lts_theta`. When referring to definitions involving the transition relation of `lts_theta`, we have to prefix them, e.g. `lts_theta.SB` for the definition of strong bisimulations using $\longmapsto^\vartheta$ instead of $\longmapsto$.

By default, `interpretations` do not import special notation, so we reassign strong bisimilarity notation $\leftrightarrow$ to `lts_theta`, since we do not care about strong bisimilarity in $\mathbb{T}$.

context `lts_timeout_mappable` **begin**

interpretation `lts_theta: lts tran_theta` ⟨*proof*⟩
no_notation `local.strongly_bisimilar` (‹_ ↔ _› [70, 70] 70)
notation `lts_theta.strongly_bisimilar` (‹_ ↔ _› [70, 70] 70)

We can now formalise the proof as described above.

If ... ($\Longrightarrow$)

definition SRB_mapping — $\mathcal{S}$
  :: ‹('s⇒'a set option⇒'s ⇒ bool) ⇒ ('ss⇒'ss ⇒ bool)›
  where ‹SRB_mapping R P Q ≡
    (∃ p q. P = ϑ(p) ∧ Q = ϑ(q) ∧ R p None q) ∨
    (∃ p q X. P = ϑ[X](p) ∧ Q = ϑ[X](q) ∧ R p (Some X) q)›

lemma SRB_mapping_is_SB:
  assumes ‹SRB R›
  shows ‹lts_theta.SB (SRB_mapping R)›
    (is ‹lts_theta.SB ?S›**)**
⟨*proof*⟩

lemma srby_implies_sby:
  assumes ‹p $\leftrightarrow_r$ q›
  shows ‹ϑ(p) ↔ ϑ(q)›
  ⟨*proof*⟩

lemma sxby_implies_sby:
  assumes ‹p $\leftrightarrow_r{}^X$ q›
  shows ‹ϑ[X](p) ↔ ϑ[X](q)›
  ⟨*proof*⟩

... and only if ($\Longleftarrow$)

definition strong_bisimilarity_mapping — $\mathcal{R}$
  :: ‹'s⇒'a set option⇒'s ⇒ bool›
  where ‹(strong_bisimilarity_mapping) p XoN q
    ≡ (XoN = None ∧ (ϑ(p)) ↔ (ϑ(q))) ∨
    (∃ X. XoN = Some X ∧ X ⊆ visible_actions ∧ (ϑ[X](p)) ↔ (ϑ[X](q)))›

lemma strong_bisimilarity_mapping_is_SRB:
  shows ‹SRB strong_bisimilarity_mapping›
    (is ‹SRB ?R›**)**
  ⟨*proof*⟩

lemma sby_implies_srby:
  assumes ‹ϑ(p) ↔ ϑ(q)›
  shows ‹p $\leftrightarrow_r$ q›
  ⟨*proof*⟩

lemma sby_implies_sxby:
  assumes ‹ϑ[X](p) ↔ ϑ[X](q)› ‹X ⊆ visible_actions›
  shows ‹p $\leftrightarrow_r{}^X$ q› ⟨*proof*⟩

We need to include the assumption X ⊆ visible_actions, since for ¬ X ⊆ visible_actions, ϑ[X](p) and ϑ[X](q) might be identical (since we do not require injectivity for that subset of the domain), so ϑ[X](p) ↔ ϑ[X](q) would be true, whereas p $\leftrightarrow_r{}^X$ q would be false (since X ⊆ visible_actions

is part of the definition of SRBs).

Iff ($\Longleftrightarrow$)

theorem `strongly_reactive_bisimilar_iff_theta_strongly_bisimilar:`
shows ‹`p` $\leftrightarrow_r$ `q` $\Longleftrightarrow$ ϑ`(p)` $\leftrightarrow$ ϑ`(q)`›
using `sby_implies_srby srby_implies_sby` **by** `fast`

theorem `strongly_X_bisimilar_iff_theta_X_strongly_bisimilar:`
assumes ‹`X` $\subseteq$ `visible_actions`›
shows ‹`p` $\leftrightarrow_r^X$ `q` $\Longleftrightarrow$ ϑ`[X](p)` $\leftrightarrow$ ϑ`[X](q)`›
using `sxby_implies_sby sby_implies_sxby assms` **by** `fast`

end — of `context lts_timeout_mappable`

3.3 A Mapping for Formulas

I will now introduce a mapping $\sigma(\cdot)$ that maps formulas of $\mathrm{HML_t}$ to formulas of HML, in the context of the process mapping from section 3.1.

Again, we have $\mathbb{T} = (Proc, Act, \rightarrow)$ and $\mathbb{T}_\vartheta = (Proc_\vartheta, Act_\vartheta, \rightarrow_\vartheta)$ as defined in section 3.1, with with $A = Act \setminus \{\tau, t\}$, and we assume that $t_\varepsilon \notin Act$ and $\forall X \subseteq A.\ \varepsilon_X \notin Act$.

Let σ : ($\mathrm{HML_t}$ formulas) $\longrightarrow$ (HML formulas) be recursively defined by

$$
\begin{aligned}
\sigma(\textstyle\bigwedge_{i\in I}\varphi_i) &= \textstyle\bigwedge_{i\in I}\sigma(\varphi_i) && \\
\sigma(\neg\varphi) &= \neg\,\sigma(\varphi) && \\
\sigma(\langle\tau\rangle\varphi) &= \langle\tau\rangle\,\sigma(\varphi) && \\
\sigma(\langle\alpha\rangle\varphi) &= \langle\alpha\rangle\,\sigma(\varphi) \vee && \\
&\quad\ \langle\varepsilon_A\rangle\langle\alpha\rangle\,\sigma(\varphi) \vee && \\
&\quad\ \langle t_\varepsilon\rangle\langle\varepsilon_A\rangle\langle\alpha\rangle\,\sigma(\varphi) && \text{if } \alpha \in A \\
\sigma(\langle\alpha\rangle\varphi) &= \mathit{ff} && \text{if } \alpha \notin A \cup \{\tau\} \\
\sigma(\langle X\rangle\varphi) &= \langle\varepsilon_X\rangle\langle t\rangle\,\sigma(\varphi) \vee && \\
&\quad\ \langle t_\varepsilon\rangle\langle\varepsilon_X\rangle\langle t\rangle\,\sigma(\varphi) && \text{if } X \subseteq A \\
\sigma(\langle X\rangle\varphi) &= \mathit{ff} && \text{if } X \not\subseteq A
\end{aligned}
$$

This mapping simply expresses the time-out semantics given by the satisfaction relations of $\mathrm{HML_t}$ in terms of ordinary HML evaluated on our mapped LTS $\mathbb{T}_\vartheta$. The disjunctive clauses compensate for the additional environment transitions (ε-actions) that are not present in $\mathbb{T}$.

Isabelle

The implementation of the mapping in Isabelle is rather straightforward, although some details might not be obvious:

`cimage (λ φ. σ(φ))` Φ is the image of the countable set Φ under the function $\lambda\ \varphi.\ \sigma(\varphi)$, so it corresponds to $\{\sigma(\varphi) \mid \varphi \in \Phi\}$ for countable Φ.

`no_special_action` α corresponds to $\alpha \in A$ with our assumption about there being no ε-actions in Act. Similarly, α `= t ∨ α = t_`ε `∨ α = ε[X]` corresponds to $\alpha \notin A \cup \{\tau\}$.

context `lts_timeout_mappable` **begin**

```
function HMt_mapping :: ‹('a)HMLt_formula ⇒ ('a)HML_formula›
  (‹σ'(_')›)
  where
    ‹σ(HMLt_conj Φ) = HML_conj (cimage (λ φ. σ(φ)) Φ)›
  | ‹σ(HMLt_neg φ) = HML_neg σ(φ)›
  | ‹α = τ ⟹
      σ(HMLt_poss α φ) = HML_poss α σ(φ)›
  | ‹no_special_action α ⟹
      σ(HMLt_poss α φ) = HML_disj (acset {
        HML_poss α σ(φ),
        HML_poss ε[visible_actions] (HML_poss α σ(φ)),
        HML_poss t_ε (HML_poss ε[visible_actions] (HML_poss α σ(φ)))
      })›
  | ‹α = t ∨ α = t_ε ∨ α = ε[X] ⟹
      σ(HMLt_poss α φ) = HML_false›
  | ‹X ⊆ visible_actions ⟹
      σ(HMLt_time X φ) = HML_disj (acset {
        HML_poss ε[X] (HML_poss t σ(φ)),
        HML_poss t_ε (HML_poss ε[X] (HML_poss t σ(φ)))
      })›
  | ‹¬ X ⊆ visible_actions ⟹
      σ(HMLt_time X φ) = HML_false›
  ⟨proof⟩
```

Once again, we show that our function terminates using a well-founded relation.

```
inductive_set sigma_wf_rel :: ‹(('a)HMLt_formula) rel›
  where
    ‹φ ∈c Φ ⟹ (φ, HMLt_conj Φ) ∈ sigma_wf_rel›
  | ‹(φ, HMLt_neg φ) ∈ sigma_wf_rel›
  | ‹(φ, HMLt_poss α φ) ∈ sigma_wf_rel›
  | ‹(φ, HMLt_time X φ) ∈ sigma_wf_rel›

lemma sigma_wf_rel_is_wf: ‹wf sigma_wf_rel›
  ⟨proof⟩
```

termination `HMt_mapping` ⟨*proof*⟩

end — of `context lts_timeout_mappable`

3.4 Reduction of Formula Satisfaction

We will show that, for a process p of an LTS_t and a formula φ of HML_t, we have $p \vDash \varphi \iff \vartheta(p) \vDash \sigma(\varphi)$ and $p \vDash_X \varphi \iff \vartheta_X(p) \vDash \sigma(\varphi)$. The proof is rather straightforward: we simply use induction over HML_t formulas and show that, for each case, the semantics given by van Glabbeek's satisfaction relations and those given by the mappings σ and ϑ/ϑ_X coincide. Due to the relative complexity of the mapping and the satisfaction relations, the proof is quite tedious, however.

Isabelle

Similar to the formalisations of section 3.2, we begin by interpreting our transition mapping `tran_theta` as an `lts` and reassigning notation appropriately (we only care about HML formula satisfaction for $\mathbb{T}_\vartheta$, not $\mathbb{T}$).

context `lts_timeout_mappable` **begin**

interpretation `lts_theta: lts tran_theta` ⟨*proof*⟩
no_notation `local.HML_sat ("_ ⊨ _" [50, 50] 50)`
notation `lts_theta.HML_sat ("_ ⊨ _" [50, 50] 50)`

We show `p ⊨?[XoN] φ ⟷ ϑ?[XoN](p) ⊨ σ(φ)` by induction over φ. By using those terms for formula satisfaction and process mappings that handle both triggered and stable environments, we can handle both situations simultaneously, which is required due to the interdependence of $\vDash$ and $\vDash_X$. However, this requires us to consider four cases (each combination of $\models?[\text{XoN}_1]$ and $\vartheta?[\text{XoN}_2]$ for $\text{XoN}_1, \text{XoN}_2 \in$ `{None, Some X}`[1]) per inductive case for φ. Together with the many disjunctive clauses in the mapping, a large number of cases needs to be considered, leading to a proof spanning roughly 350 lines of Isabelle code.

lemma `HMLt_sat_iff_HML_sat:`
  assumes ‹`XoN = None ∨ (XoN = (Some X) ∧ X ⊆ visible_actions)`›
  shows ‹`p ⊨?[XoN] φ ⟷ ϑ?[XoN](p) ⊨ σ(φ)`›
  ⟨*proof*⟩

[1] Once again, we do not consider cases where ¬ `X ⊆ visible_actions`.

The theorems using the nicer notation are now immediate consequences of this lemma.

theorem HMLt_HMLt_sat_triggered_iff_triggered_env_HML_sat:
  shows ‹p $\models \varphi \iff \vartheta$(p) $\models \sigma(\varphi)$›
  using HMLt_sat_iff_HML_sat **by** blast
theorem HMLt_HMLt_sat_stable_iff_stable_env_HML_sat:
  assumes ‹X $\subseteq$ visible_actions›
  shows ‹p $\models$[X] $\varphi \iff \vartheta$[X](p) $\models \sigma(\varphi)$›
  using HMLt_sat_iff_HML_sat assms **by** blast

end — of context lts_timeout_mappable

Chapter 4

Discussion

We have shown that checking strong reactive/X-bisimilarity (in an $\mathrm{LTS_t}$) is reducable to checking strong bisimilarity. This result may be useful in the context of automated tools for checking equivalences on LTSs. Since, the mapping creates a state for every subset of the visible actions A, for each original state, plus another triggered state (i.e. $|Proc_\vartheta| = |Proc| \cdot (1 + 2^{|A|})$), checking reactive bisimilarity by using the mapping would be exponentially harder (in the worst case) than simply checking ordinary bisimilarity. However, at least for SRBs, the size of the relations also grows exponentially with the number of visible actions (due to the clause $(p, q) \in \mathcal{R} \Longrightarrow (p, X, q) \in \mathcal{R}$ for $X \subseteq A$), so a a naïve implementation using SRBs would not necessarily be more efficient. Below, I propose an optimisation that significantly reduces the number of states for a large number of LTSs.

As I mentioned in section 3.1, the mapped LTS represents the closed system consisting of the original reactive system and its environment. Hence, the reduction does in no way challenge the semantic value offered by $\mathrm{LTS_t}$s, e.g. for protocol specifications. Rather, when shown as a graph, the mapping offers a useful view at systems specified as $\mathrm{LTS_t}$s that explicilty shows them in all possible 'situations'. In a mapped LTS, for example, it is easy to find states that are unreachable, or reachable only in certain environments, whereas the reachability of states in an $\mathrm{LTS_t}$ may not be directly obvious, as we saw in the example on page 37. Admittedly, the mapping gets very crowded even for small $\mathrm{LTS_t}$s; on a local level or merely as a thinking aid, however, the 'explicitness' of the mapping may be useful.

The formula mapping is probably less useful in that regard, due to the large number of disjunctions. It might, however, also be useful in the context of automated tools.

Possible Optimisations

Let $\mathcal{I}^*(p)$ be the set of visible actions that can be encountered as initial actions after arbitrary sequences of τ- and t-transitions starting at p.
More concretely, for $n \in \mathbb{N}$, let

$$\begin{aligned} p_0 \xrightarrow{X}{}^* p_n &\equiv \exists p_1, \ldots, p_{n-1}.\ \forall i \in [0, n-1].\ \exists \alpha \in X.\ p_i \xrightarrow{\alpha} p_{i+1}; \\ \mathrm{reach}(p, X) &\equiv \{p' \mid p \xrightarrow{X}{}^* p'\}; \\ \mathcal{I}^*(p) &\equiv \bigcup_{p' \in \mathrm{reach}(p, \{\tau, t\})} \mathcal{I}(p'). \end{aligned}$$

Then[1] we can modify first rule of the process mapping (from section 3.1) by changing the side condition from $X \subseteq A$ to $X \subseteq \mathcal{I}^*(p)$, yielding:

$$(1)\ \frac{}{\vartheta(p) \xrightarrow{\varepsilon_X}_\vartheta \vartheta_X(p)}\ X \subseteq \mathcal{I}^*(p).$$

This way, we only include environment stabilisations that are relevant for the current process: all transitions other than τ and t will always trigger a change in the environment; hence, after having stabilised, the actions in $\mathcal{I}^*(p)$ are the only ones the process p could ever perform before triggering the environment.
In the worst case, the number of mapping-states is still exponential in the size of the alphabet, i.e. $|Proc_\vartheta| = \mathcal{O}(|Proc| \cdot 2^{|Act|})$. For a large number of LTS$_\mathrm{t}$s, however, this alteration would reduce the number of mapping-states significantly.
Unfortunately, I did not manage to prove the reduction with this altered mapping, but I am convinced that it is possible.

Necessity of Special Actions

Environment Time-Outs t_ε It should be possible to replace the action t_ε by the normal time-out action t in the mapping. Since, in the present version, all t_ε-transitions end in a $\vartheta(p)$-state, where always at least an $\varepsilon_\emptyset$-transition can be performed, whereas all t-transitions end in a $\vartheta_X(p)$-state, where no ε_Y-transition can ever be performed, the distinction between environment time-outs and system time-outs should be possible without distinguishing the actions t and t_ε.

Environment Stabilisations ε_X In the first version of the mapping, I used a single stabilisation action s_ε, but got stuck trying to prove that

[1] Note that $p \in \mathrm{reach}(p, X)$ for all p and X.

$\vartheta(p) \leftrightarrow \vartheta(q)$ implies $\forall X \subseteq A.\ \vartheta_X(p) \leftrightarrow \vartheta_X(q)$, which lead me to define the family of stabilisation actions instead. Sadly, I did not manage to find a counterexample where the reduction using this simpler mapping does not work.

However, in the context of $\mathrm{HML}_{\mathrm{t}}$, there would be no obvious way to define the formula mapping for $\sigma(\langle X \rangle \varphi)$; in the present version, the mapping relies on being able to use ε_X in this case (see section 3.3). Hence, I have come believe that the ε_X-actions are indeed required in their present form.

Furthermore, although including the environment information both in the states $\vartheta_X(p)$ as well as the stabilisation actions ε_X may seem redundant, it might be required. As we discussed in section 2.2, the intensional identity of the state is not 'knowable for bisimilarity'; rather, only the observable transitions are relevant. Hence, it is plausible that the information about allowed actions is required to be included in the transition labels themselves, in order for the reduction to work.

Bibliography

[AILS07] Luca Aceto, Anna Ingolfsdottir, Kim Guldstrand Larsen, and Jirí Srba. *Reactive systems: modelling, specification and verification*. Cambridge University Press, 2007.

[AIS11] Luca Aceto, Anna Ingolfsdottir, and Jirí Srba. The algorithmics of bisimilarity. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, Cambridge Tracts in Theoretical Computer Science, pages 100—172. Cambridge University Press, 2011. doi:10.1017/CBO9780511792588.004.

[Bis18] Benjamin Bisping. Computing coupled similarity. Master's thesis, Technische Universität Berlin, 2018.

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

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1985.

[Kel76] Robert M Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[Nip21] Tobias Nipkow. Programming and proving in Isabelle/HOL, February 2021. Accessed on 07.06.2021. URL: https://isabelle.in.tum.de/doc/prog-prove.pdf.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical computer science*, pages 167–183. Springer, 1981.

[San11a] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[San11b] Davide Sangiorgi. Origins of bisimulation and coinduction. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in*

Bisimulation and Coinduction, Cambridge Tracts in Theoretical Computer Science, pages 1—37. Cambridge University Press, 2011. doi:10.1017/CBO9780511792588.002.

[vG20] Rob van Glabbeek. Reactive bisimulation semantics for a process algebra with time-outs. *arXiv preprint arXiv:2008.11499*, 2020.

[vG21] Rob van Glabbeek. Failure trace semantics for a process algebra with time-outs. *Logical Methods in Computer Science*, 17, 2021.

[Wen21a] Makarius Wenzel. The Isabelle system manual, February 2021. Accessed on 28.05.2021. URL: https://isabelle.in.tum.de/doc/system.pdf.

[Wen21b] Makarius Wenzel. The Isabelle/Isar reference manual, February 2021. Accessed on 28.05.2021. URL: https://isabelle.in.tum.de/doc/isar-ref.pdf.

Appendix A

Isabelle

Isabelle is an interactive proof assistant and Isabelle/HOL is an implementation of *higher-order logic* in Isabelle. With it, one can interactively prove propositions about theories that are formalised in terms of higher-order logic. Many theories have been formalised (and many theorems proven) in Isabelle/HOL and are publicly available.[1]

In this appendix, I will give a short introduction into the most important concepts of Isabelle. For an extensive tutorial, see [Nip21]. A complete documentation can be found in [Wen21b].

All formalisations are based on Isabelle/HOL, allowing us to use e.g. quantifiers $\forall$ and $\exists$.

Simple Definitions

The command `definition` defines a term by establishing an equality, denoted by $\equiv$. This term can be a function or a constant (i.e. 0-ary function). Predicates are functions to Boolean values.

Definitions are annotated by their type. As an example, we define the predicate `even`, which maps an integer to a Boolean value.

```
definition even :: ‹int ⇒ bool›
  where ‹even n ≡ ∃ m::int . n = 2 * m›
```

Functions can be defined in uncurried form (e.g. `(int × int) ⇒ bool`) or in curried form (e.g. `int ⇒ int ⇒ bool`). As a very trivial example, we can define equality predicates for integers. Compared to the curried version, the uncurried version does not allow for easy pattern matching. This is why, in this thesis, I will usually specify functions in curried form.

```
definition equal_uncurried :: ‹(int × int) ⇒ bool›
  where ‹equal_uncurried pair ≡ ∃ n m. pair = (n, m) ∧ n = m›
```

[1] see Isabelle's Archive of Formal Proofs at isa-afp.org

```
definition equal_curried :: ‹int ⇒ int ⇒ bool›
  where ‹equal_curried n m ≡ n = m›
```

We can also use type variables (prefixed with an apostrophe, e.g. `'a`) instead of concrete types to get more abstract terms.

```
definition equal_abstract :: ‹'a ⇒ 'a ⇒ bool›
  where ‹equal_abstract a b ≡ a = b›
```

For a less trivial example, we define a predicate `symmetric` that determines whether a given relation is symmetric. An arbitrary homogeneous relation in curried form has the type `'a ⇒ 'a ⇒ bool`.

```
definition symmetric :: ‹('a ⇒ 'a ⇒ bool) ⇒ bool›
  where ‹symmetric R ≡ ∀ a b. R a b ⟶ R b a›
```

We can also assign notation to a term during the definition, where `_` is a placeholder (and the numbers behind the notation specification represent priorities for parsing, which may be ignored by the reader).

```
definition approx :: ‹int ⇒ int ⇒ bool›
  (‹_ ≈ _› [50, 50] 50)
  where ‹n ≈ m ≡ n=m-1 ∨ n=m ∨ n=m+1›
```

Abbreviations are used the same way as definitions, except that, in order to use the equality established by definitions in proofs, we need to explicitly refer to the definition, whereas abbreviations are always expanded by the proof system. An example a little further down below should clarify the distinction.

```
abbreviation reflexive :: ‹('a ⇒ 'a ⇒ bool) ⇒ bool›
  where ‹reflexive R ≡ ∀ a. R a a›
```

Proving Propositions

Propositions are terms of Boolean type. Propositions can be given using any of the commands `proposition`, `lemma`, `theorem`, `corollary`, and require a proof.

Since Isabelle is an *interactive* proof assistant, proofs are usually meant to be spelled out in code so as to be readable by humans, and the validity of individual steps is verified by certain automated proof methods (e.g. `simp`, `arith`, `auto`, `fast`, `blast`, …).

As an example, we will show that the relation `approx` is `symmetric`.

Since `symmetric` was defined using the command `definition`, we need to explicitly unfold it, where `symmetric_def` is the fact (about the equality) introduced by the definition.

The method specified after the command `proof` adjusts the proof goal in some way. Ideally, the proof steps should be clear to the reader even without

seeing what exactly the automated methods are doing. I have explained each of the steps using comments below.

```
proposition ‹symmetric approx›
  unfolding symmetric_def
proof (clarify)
  — We want to show that for any n and m with n ≈ m, we have m ≈ n.
  fix n m
  assume ‹n ≈ m›
  — Using the definition of approx, we now this about n and m.
  hence ‹n=m-1 ∨ n=m ∨ n=m+1› unfolding approx_def .
  thus ‹m ≈ n›
  — With disjunction elimination, we examine each case in a sub-proof.
  proof (elim disjE)
    assume ‹n = m - 1›
    hence ‹m = n + 1› by arith
    thus ‹m ≈ n› unfolding approx_def by blast
  next
    assume ‹n = m›
    thus ‹m ≈ n› using approx_def by blast
  next
    assume ‹n = m + 1›
    hence ‹m = n - 1› by arith
    thus ‹m ≈ n› using approx_def by blast
  qed
qed
```

This proof was probably more detailed than was necessary. By unfolding the other definition as well, this proposition can be proven directly with the proof method `arith`.

```
proposition ‹symmetric approx›
  unfolding symmetric_def approx_def by arith
```

To see the difference between definitions and abbreviations, note that the following proposition is provable without unfolding `reflexive_def` (since `reflexive` is an abbreviation, there is no such fact in this context).

```
proposition ‹reflexive approx›
  unfolding approx_def by auto
```

In practice, of course, one has to strike a balance between transparency/comprehensibility and conciseness of proofs.

Inductive Definitions

Inductively defined predicates can be given using premise-conclusion pairs and multiple clauses.

```
inductive even_ind :: ‹int ⇒ bool›
```

```
  where
    ⟨even_ind 0⟩
  | ⟨even_ind n ⟹ even_ind (n+2)⟩
```

Function Definitions

The command `function` also establishes equalities, but usually in more complex ways, so that, sometimes, it is not obvious whether a function is well-defined. Hence, the command `function` introduces proof obligations to show that the function is, in fact, well-defined. (These proofs are usually easily solved by the automated proof methods.)

The function is then assumed to be partial. The command `termination` introduces proof obligations to show that the function always terminates (and is thus total). For the example below, this proof, again, is trivial.

```
function factorial :: ⟨nat ⇒ nat⟩
  where
    ⟨n = 0 ⟹ factorial n = 1⟩
  | ⟨n > 0 ⟹ factorial n = n * factorial (n-1)⟩
  by auto

termination factorial using "termination" by force
```

After proving well-definedness and totality, we have access to facts about the function that can be used in proofs, e.g. induction principles. More details can be found in section 2.3, when we define our first non-trivial `function`.

Data Types

With the command `datatype`, new types can be defined, possibly in dependence on existing types, by defining a set of (object) constructor functions. For example, we can (re-)define the type of natural numbers.

```
datatype natural_number =
  Zero — 0-ary base constructor
| Suc ⟨natural_number⟩ — unary recursive/inductive constructor
```

We can define type constructors, i.e. types depending on other types (to be distinguished from the object constructors above) by parameterising the type with type variables.

```
datatype ('a)list =
  Empty
| Cons ⟨'a⟩ ⟨('a)list⟩
— binary constructor taking an element of type 'a and another ('a)list
```

Locales

Locales define a context consisting of type variables, object variables, and assumptions. These can be accessed in the entire context. Locales can also be instantiated by specifying concrete types (or type variables from another context) for the type variables, and extended to form new locales. We can reenter the context of a locale later on, using the command `context`.

Section 2.1 provides a good example for how locales are used in Isabelle to formalise linear transition systems.

Appendix B

Infinitary Hennessy-Milner Logic

We will show that a modal characterisation of strong bisimilarity is possible without any assumptions about the cardinality of derivative sets `Der(p, α)`, using infinitary HML (with conjunction of arbitrary cardinality). Instead of formalising formulas under a conjunction as a countable set,[1] we use an index set of arbitrary type `I :: 'x set` and a mapping `F :: 'x ⇒ ('a, 'x)HML_formula` so that each element of `I` is mapped to a formula. This closely resembles the semantics of $\bigwedge_{i \in I} \varphi_i$. Since the mapping is total, we cannot use the empty conjunction as a base for the type. Instead of using partial mappings `F :: 'x ⇒ ('a, 'x)HML_formula option`, I included a constructor `HML_true` and implicitly assume that `F` maps to `HML_true` for all objects of type `'x` that are not elements of `I`.

```
datatype ('a, 'x)HML_formula =
  HML_true
| HML_conj ‹'x set› ‹'x ⇒ ('a, 'x)HML_formula›
| HML_neg ‹('a, 'x)HML_formula›
| HML_poss ‹'a› ‹('a, 'x)HML_formula›
```

Satisfaction Relation

Data types cannot be used with arbitrary parameter types `'x` in concrete contexts; so when using our data type in the `context lts`, we use the type of processes `'s` as the type for conjunction index sets. Since this suffices to proof the modal characterisation, we can conclude that it suffices for the cardinality of conjunction to be equal to the cardinality of the set of processes *Proc*. As we can deduce from the part of the proof where formula

[1] Note that it is not possible to define a dataype with a constructor depending on a set of the type itself, i.e. `HML_conj ‹('a)HML_formula set›` would not yield a valid data type. Dunno y tho.

conjunction is used, a weaker requirement would be to allow for conjunction of cardinality equal to $\max_{\substack{p \in Proc \\ \alpha \in Act}} |\mathrm{Der}(p, \alpha)|$.

The remainder of this section follows the same structure as section 2.3. The explanations from there mostly apply here as well.

context `lts` **begin**

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

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

lemma `HML_wf_rel_is_wf: ‹wf HML_wf_rel›`
  ⟨*proof*⟩

termination `satisfies` **using** `HML_wf_rel_is_wf`
  by `(standard, (simp add: HML_wf_rel.intros)+)`

Modal Characterisation of Strong Bisimilarity

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

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

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

lemma `strong_bisimilarity_implies_HML_equivalent:`
  assumes `‹p ↔ q› ‹p ⊨ φ›`
  shows `‹q ⊨ φ›`
  using `assms`

```
proof (induct φ arbitrary: p q)
  case HML_true
  then show ?case
    by force
next
  case (HML_conj X F)
  then show ?case
    by force
next
  case (HML_neg φ)
  then show ?case
    using satisfies.simps(3) strongly_bisimilar_symm by blast
next
  case (HML_poss α φ)
  then show ?case
    by (meson satisfies.simps(4) strongly_bisimilar_step(1))
qed


lemma HML_equivalence_is_SB:
  shows ⟨SB HML_equivalent⟩
proof -
  {
    fix p q p' α
    assume ⟨HML_equivalent p q⟩ ⟨p ⟼α p'⟩
    assume ⟨∀ q' ∈ Der(q, α). ¬ HML_equivalent p' q'⟩

    hence "exists_φq'": ⟨∀q' ∈ Der(q, α). ∃φ. p' ⊨ φ ∧ ¬ q' ⊨ φ⟩
      using distinguishing_formula by blast

    let ?I = ⟨Der(q, α)⟩
    let ?F = ⟨(λ q'. SOME φ. p' ⊨ φ ∧ ¬ q' ⊨ φ)⟩
    let ?φ = ⟨HML_conj ?I ?F⟩

    from "exists_φq'" have ⟨p' ⊨ ?φ⟩
      by (smt (z3) satisfies.simps(2) someI_ex)
    hence ⟨p ⊨ HML_poss α ?φ⟩ using ⟨p ⟼α p'⟩
      by auto

    from "exists_φq'" have ⟨∀q' ∈ Der(q, α). ¬ q' ⊨ ?φ⟩
      by (smt (z3) satisfies.simps(2) someI_ex)
    hence ⟨¬ q ⊨ HML_poss α ?φ⟩
      by simp

    from ⟨p ⊨ HML_poss α ?φ⟩ ⟨¬ q ⊨ HML_poss α ?φ⟩ have False
      using ⟨HML_equivalent p q⟩
      unfolding HML_equivalent_def by blast
  }
```

```
  thus ‹SB HML_equivalent› unfolding SB_def
    using HML_equivalent_symm by blast
qed


theorem modal_characterisation_of_strong_bisimilarity:
  shows ‹p ↔ q  ⟺  (∀ φ. p ⊨ φ ⟷ q ⊨ φ)›
proof
  show ‹p ↔ q ⟹ ∀φ. (p ⊨ φ) = (q ⊨ φ)›
    using strong_bisimilarity_implies_HML_equivalent
      strongly_bisimilar_symm
    by blast
next
  show ‹∀φ. (p ⊨ φ) = (q ⊨ φ) ⟹ p ↔ q›
    using HML_equivalence_is_SB HML_equivalent_def
      strongly_bisimilar_def
    by blast
qed

end — of context lts
```

Appendix C

Example Instantiation

To complete the proofs from chapter 3, I will show that mappings `stabilise` (ε`(_)`) and `in_env` (ϑ`?[_](_)`), which existence we assumed up to now, do, in fact, exist. I will define example mappings and show that, together with these, arbitrary `lts_timeout` can be interpreted as `lts_timeout_mappable`, thereby showing that my reductions are valid for arbitrary LTS_ts.

First, we define the data types for $Proc_\vartheta$ and Act_ϑ in dependence to arbitrary types `'s` and `'a` for *Proc* and *Act*, respectively:

```
datatype ('s, 'a)Proc_ϑ = triggered 's | stable ⟨'a set⟩ 's | DumpState
datatype ('a)Act_ϑ = act 'a | t_ε | ε ⟨'a set⟩ | DumpAction
```

Since $Act \neq Act_\vartheta$, we have to define a new predicate `tran_mappable`.

```
context lts_timeout begin

inductive tran_mappable
  :: ⟨'s ⇒ ('a)Act_ϑ ⇒ 's ⇒ bool⟩
  where ⟨tran p α p' ⟹ tran_mappable p (act α) p'⟩
```

We can now specify mappings `stabilise` and `in_env`.

```
function stabilise :: ⟨('a)Act_ϑ set ⇒ ('a)Act_ϑ⟩
  where
    ⟨∀ α∈X. (∃ α'. α = act α') ⟹ stabilise X = ε {α' . act α' ∈ X}⟩
  | ⟨∃ α∈X. (∄ α'. α = act α') ⟹ stabilise X = DumpAction⟩
  ⟨proof⟩
termination ⟨proof⟩

function in_env :: ⟨('a)Act_ϑ set option ⇒ 's ⇒ ('s, 'a)Proc_ϑ⟩
  where
    ⟨in_env None p = triggered p⟩
  | ⟨∀ α∈X. (∃ α'. α = act α') ⟹
      in_env (Some X) p = stable {α' . act α' ∈ X} p⟩
  | ⟨∃ α∈X. (∄ α'. α = act α') ⟹
      in_env (Some X) p = DumpState⟩
```

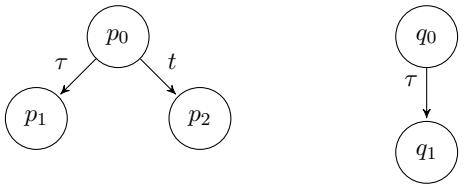
⟨*proof*⟩
termination ⟨*proof*⟩

We show that together with these mappings, any `lts_timeout` (since that is the context we are in) is an `lts_timeout_mappable`.

lemma `is_mappable: ‹lts_timeout_mappable`
`tran_mappable (act τ) (act t) t_ε stabilise in_env›`
⟨*proof*⟩

end — of `context lts_timeout`

A Tiny Example LTS_t



datatype `Proc = p0|p1|p2|q0|q1`
datatype `Act = τ|t`
inductive `Tran :: ‹Proc ⇒ Act ⇒ Proc ⇒ bool›`
where
`‹Tran p0 τ p1›`
`| ‹Tran p0 t p2›`
`| ‹Tran q0 τ q1›`

We interpret the `Tran` predicate as an `lts_timeout`, and then together with our mappings as an `lts_timeout_mappable`.

interpretation `tiny_lts:`
`lts_timeout Tran τ t`
by `(simp add: lts_timeout.intro)`
interpretation `tiny_lts_mappable:`
`lts_timeout_mappable tiny_lts.tran_mappable ‹act τ› ‹act t› ‹t_ε›`
`tiny_lts.stabilise tiny_lts.in_env`
using `tiny_lts.is_mappable .`
— (notation assignments omitted from thesis document)

We can now prove a few lemmas about our example LTS_t that we would need for any bisimilarity proofs. I abstained from actually including a proof, but the lemmas should suffice to convince you that it would be possible.

lemma `‹tiny_lts_mappable.visible_actions = ∅›`
proof -
have `‹tiny_lts.visible_actions = ∅›`
using `tiny_lts.visible_actions_def`

```
    Act.exhaust Collect_cong empty_def
  by auto
moreover have ‹tiny_lts_mappable.visible_actions
    = image (λ α. act α) tiny_lts.visible_actions›
  using Act.exhaust
    tiny_lts.tran_mappable.simps
    tiny_lts.visible_actions_def
    tiny_lts_mappable.visible_actions_def
  by auto
ultimately show ?thesis by force
qed

lemma ‹∄ P'. ϑ[∅](p0) ⟼ϑ(act t) P'›
proof safe
  fix P'
  assume ‹ϑ[∅](p0) ⟼ϑ(act t) P'›
  hence ‹idle p0 ∅›
    using tiny_lts_mappable.generation_sys_timeout by blast

  have ‹p0 ⟼(act τ) p1› using Tran.intros(1)
    by (simp add: tiny_lts.tran_mappable.intros)

  with ‹idle p0 ∅› show False
    unfolding tiny_lts_mappable.initial_actions_def by blast
qed
```

Zusammenfassung in Deutscher Sprache

Etikettierte Übergangssysteme

Eidesstattliche Versicherung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändigsowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Ort, Datum Unterschrift