Measuring expressive power of HML formulas in Isabelle/HOL

Karl Mattes

4th February 2024

Contents

Chapter 1

Introduction

In this thesis, I show the corrspondence between various equivalences popular in the reactive systems community and coordinates of a price function, as introduced by Benjamin Bisping (citation). I formalised the concepts and proofs discussed in this thesis in the interactive proof assistant Isabelle (citation).

The term *reactive system* (citation) describes computing systems that continously interact with their environment. Unlike sequential systems, the behavior or reactive systems is inherently event-driven and concurrent. They can be modeled by labeled directed graphs called *labeled transition systems* (LTSs) (citation), where the nodes of an LTS describe the states of a reactive system and the edges describe transitions between those states.

The semantics of reactive systems can be modeled as equivalences, that determine wether or not two systems behave similarly. In the literature on concurrent systems many different notion of equivalence can be found, the maybe best known being *(strong) bisimilarity.* Rab van Glabbeek‘s *linear-time–branching-spectrum*(citation) ordered some of the most popular in a hierachy of equivalences. -> New Paper characterizes them different...
(HML beschreibung als erstes?!!)

- Reactive Systmes
- modelling (via lts etc)
- Semantics of resysts
- Verification
- different notions of equivalence (because of nondeteminism?) -> van glabbeek
- Different definitions of semantics -> HML/relational/...

–> linear-time–branching-time spectrum understood through properties of HML
–> capture expressiveness capabilities of HML formulas via a function
–> Contribution o Paper: The in (citation) introduced expressiveness function and its coordinates captures the linear time branching time spectrum..

- Isabelle:
- formalization of concepts, proofs
- what is isabelle
- difference between mathematical concepts and their implementation?

Chapter 2

Foundations

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

- mention sources (Ben / Max Pohlmann?)

2.1 Labelled Transition Systems

Definition 1 (Labeled transition Systems)

A Labelled Transition System (LTS) is a tuple $\mathcal{S} = (\mathit{Proc}, \mathit{Act}, \rightarrow)$ where Proc is the set of processes, Act is the set of actions and $\rightarrow \subseteq \mathit{Proc} \times \mathit{Act} \times \mathit{Proc}$ is a transition relation.

In concurrency theory, it is customary that the semantics of Reactive Systems are given in terms of labelled transition systems. The processes represent the states a reactive system can be in. A transition of one state into another, caused by performing an action, can be understood as moving along the corresponding edge in the transition relation.

- Example?

- examples (to reuse later?)???

Some more Definitions

The α-derivatives of a process are the processes that can be reached with one α-transition:

- image finite? nötig?
- image countable? - initial actions - deadlock - relevant actions? - step sequence! -

Isabelle

Zustände: `'s` und Aktionen `'a`, Transitionsrelation ist locale trans. Ein LTS wird dann durch seine Transitionsrelation definiert.

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

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

Transition System is image-finite

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

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

stimmt definition? definition benötigt nach umstieg auf sets?

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

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

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

nötig?

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

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

end
```

```
context lts
begin
```

Introduce these definitions later?

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

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

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

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

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

Trace preorder as inclusion of trace sets

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

Trace equivalence as mutual preorder

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

Trace preorder is transitive

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

Failure Pairs

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

Failure preorder and -equivalence

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

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

Possible future sets

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

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

lemma PF_trans: transp (≃PF)
  ⟨proof⟩
```

isomorphism

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

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

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

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

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

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

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

Bisimilarity is a simulation.

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


end
end
theory HML_list
  imports
 Main
```

```
 Transition_Systems

begin
datatype ('a, 'i)hml =
TT |
hml_pos ‹'a› ‹('a, 'i)hml› |
hml_conj 'i set 'i set 'i ⇒ ('a, 'i) hml

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

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

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




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

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

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

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

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


primrec flatten :: ('a, 'i) hml ⇒ ('a, 'i) hml where
flatten TT = TT |
flatten (hml_pos α ψ) = (hml_pos α (flatten ψ)) |
flatten (hml_conj I J Φ) = (hml_conj I J Φ)
```

```
inductive flattened :: ('a, 'i) hml ⇒ bool
  where
flattened TT |
flattened (hml_pos _ ψ) if flattened ψ|
flattened (hml_conj I J Φ)
if ∀x ∈ (Φ ` I). (∃α ψ. x = (hml_pos α ψ) ∧ flattened ψ)
∀y ∈ (Φ ` J). (∃α ψ. y = (hml_pos α ψ) ∧ flattened ψ) ∨
(∃I' J' Φ'. y = (hml_conj I' J' Φ') ∧ (∀x ∈ Φ' ` I'. (∃α ψ. x = (hml_pos
α ψ) ∧ flattened ψ))
                                    ∧ (Φ' ` J' = {}))
```

```
context lts begin

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

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

lemma
  assumes TT_like φ
  shows p ⊨ φ
  ⟨proof⟩

lemma nested_empty_pos_conj_TT:
  assumes nested_empty_pos_conj φ
  shows p ⊨ φ
  ⟨proof⟩
```

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

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

An HML formula `φl` implies another (`φr`) if the fact that some process `p` satisfies `φl` implies that `p` must also satisfy `φr`, no matter the process `p`.

```
definition hml_impl :: ('a, 's) hml ⇒ ('a, 's) hml ⇒ bool (infix ⇛
60)  where
  φl ⇛ φr ≡ (∀p. (p ⊨ φl) ⟶ (p ⊨ φr))

lemma hml_impl_iffI: φl ⇛ φr = (∀p. (p ⊨ φl) ⟶ (p ⊨ φr))
  ⟨proof⟩
```

Equivalence

A HML formula `φl` is said to be equivalent to some other HML formula `φr` (written `φl` ⇚⇛ `φr`) iff process `p` satisfies `φl` iff it also satisfies `φr`, no matter the process `p`.

We have chosen to define this equivalence by appealing to HML formula implication (c.f. pre-order).

```
definition hml_formula_eq :: ('a, 's) hml ⇒ ('a, 's) hml ⇒ bool (infix
⇐⇒ 60)  where
  φl ⇐⇒ φr ≡ φl ⇛ φr ∧ φr ⇛ φl
```

⇐⇒ is truly an equivalence relation.

```
lemma hml_eq_equiv: equivp (⇐⇒)
  ⟨proof⟩

lemma equiv_der:
  assumes HML_equivalent p q ∃p'. p ↦α p'
  shows ∃p' q'. (HML_equivalent p' q') ∧ q ↦α q'
  ⟨proof⟩


lemma equiv_trans: transp HML_equivalent
  ⟨proof⟩
```

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

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

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

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

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

end



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

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

translation of a trace to a formula

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

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

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

definition HML_simulation_formulas where
HML_simulation_formulas ≡ {φ. HML_simulation φ}

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

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

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

definition HML_possible_futures_formulas where
HML_possible_futures_formulas ≡ {φ. HML_possible_futures φ}
```

```
inductive HML_failure_trace :: ('a, 's)hml ⇒ bool
  where
f_trace_tt: HML_failure_trace TT |
f_trace_pos: HML_failure_trace (hml_pos α φ) if HML_failure_trace φ|
f_trace_conj: HML_failure_trace (hml_conj I J Φ)
if ((∃ψ ∈ (Φ ` I). (HML_failure_trace ψ) ∧ (∀y ∈ (Φ ` I). ψ ≠ y ⟶
nested_empty_conj y)) ∨
(∀y ∈ (Φ ` I). nested_empty_conj y)) ∧
(∀y ∈ (Φ ` J). stacked_pos_conj_pos y)


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

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

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

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

end
theory formula_prices_list
  imports
    Main
```

```
    HML_list
    HOL-Library.Extended_Nat
begin

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


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


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

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

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

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

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


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

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

end
```