Formalisation: Chain of events—Modular Process Models for the Law

Søren Debois

July 2, 2020

Contents

1 Notation **1**

2 Support and projections **2**
2.1 Support and projection of words and languages 3

3 Refinement **5**

4 Networks **8**
4.1 Actions 9

5 Transition semantic **15**

```
theory Notation
  imports Main
  HOL-Library.LaTeXsugar
  HOL-Library.OptionalSugar
  HOL-Library.Adhoc_Overloading
begin
```

1 Notation

Various objects, notably DCR graphs, have an associated notion of *well-formedness*, e.g., a string is well-formed wrt. some alphabet if every letter in the string is a member of that alphabet. We use the notation `wf` for all well-formedness predicates

```
consts wf :: 't
```

Many objects of interest—DCR markings and relations, labels, strings, languages—can be thought of as being built from underlying objects of some type. E.g., a word or string is built from letters chosen from some alphabet, a DCR marking is built from a set of events. We codify this observation in the notion of the support `support x` of some object `x`. For a particular string, the support of the string will be the set of symbols in the string; for a DCR graph, the set of events mentioned in the graph.

All the supported objects we consider also have a notion of projection π `E x`, where we derive from `x` a new object by taking away all the building blocks not in `E`. Again, for a string, we drop symbols, so π `{1::'b, 2::'b} [1::'c, 2::'c, 3::'c] = [1::'a, 2::'a]`; for DCR graphs, we remove events and any referencing relations.

We will pose some requirements on the interplay of support and projection and see some concrete examples (partial maps, strings, languages) later. For now, we note just the notation. Note that we generally write projection as π `E x` rather than the more cumbersome `projection E x`.

```
consts
  support    :: 't ⇒ 'a set
  projection :: 'a set ⇒ 't ⇒ 't  (π)
```

We use the following types for words and languages.

```
type_synonym 'a word     = 'a list
type_synonym 'a language = 'a word set
```

Note that we cannot use Isabelle's built-in type `string`, which is an alias for `char list`, since we will need to work with strings over arbitrary alphabets. However, the `string` type is nonetheless helpful for examples, so to avoid conflicts, we use the term "word" as opposed to "string".

```
no_notation  (latex)  Cons  (_ ·/ _ [66,65] 65)

end
theory Projection
  imports
    Main
    Notation
    HOL-Library.Finite_Map
begin
```

2 Support and projections

We say that a type `'t` is *supported over* `'a set` when the `support` and the projection π satisfies that (a) the support of projection at `E` is contained in `E`, and (b) the projection at some set `E` is the identity exactly when `E` is *smaller* than the support.

```
locale supported =
  fixes support :: 't ⇒ 'a set
  and   projection :: 'a set ⇒ 't ⇒ 't
```

```
  assumes sound[simp]: support (projection E t) ⊆ E
```
— The support of projection onto E is contained in E.

```
  and     tight[iff]: (support t ⊆ E) ⟷ (projection E t = t)
```
— Projection is non-trivial iff we are projecting onto a proper subset of the support.

```
adhoc_overloading support dom
adhoc_overloading projection λ E f . f |‘ E
```

2.1 Support and projection of words and languages

Both words and languages are supported: for words, the support is the set of element in it; for languages we lift the support of words pointwise.

```
definition support_w :: 'a word ⇒ 'a set
  where support_w w ≡ set w

definition support_L :: 'a language ⇒ 'a set
  where support_L L ≡ ⋃ { support_w w | w . w ∈ L }

adhoc_overloading
  support support_w support_L


definition project_w :: 'a set ⇒ 'a word ⇒ 'a word
  where project_w Y w ≡ List.filter (λx . x ∈ Y) w

definition project_L :: 'a set ⇒ 'a language ⇒ 'a language
  where project_L Y L ≡ (project_w (Y :: 'a set)) ‘ (L :: 'a language)

adhoc_overloading
  projection project_w project_L


lemmas language_simps =
  support_w_def support_L_def project_w_def project_L_def

interpretation supp_word: supported
  support_w :: 'a word ⇒ 'a set
  project_w :: 'a set ⇒ 'a word ⇒ 'a word
  by (unfold_locales, auto simp add: language_simps filter_id_conv subset_code(1))


interpretation supp_language: supported
  support_L :: 'a language ⇒ 'a set
  project_L :: 'a set ⇒ 'a language ⇒ 'a language
  proof (unfold_locales)
    fix E :: 'a set and L :: 'a language
```

```
    show support (π E L) ⊆ E by (auto simp add: language_simps)
  next
    note language_simps[simp]
    fix E :: 'a set and L :: 'a language
    show (support L ⊆ E) = (π E L = L)  proof
      assume support L ⊆ E
      { fix w assume w ∈ L
        then have π E w = w  using support L ⊆ E filter_id_conv by fastforce
}
      thus π E L = L by simp
    next
      assume π E L = L
      thus support L ⊆ E by force
    qed
  qed

context
  notes language_simps[simp]
  fixes L L1 L2 :: 'a language and E :: 'a set
begin
```

Alternatively, we can use the following more familiar definition of languages.

```
  lemma alt_project_lang_def[code_abbrev]:
    shows π E L = { π E s | s . s ∈ L }
    by auto

  lemma support_word_lang[simp,elim]:
    w ∈ L ⟹ support w ⊆ support L
    by auto

  lemma support_lang_empty[simp]:
    π E {}  = {} by simp

  lemma support_lang_monotone:
    L1 ⊆ L2 ⟹ support L1 ⊆ support L2
    by auto

  lemma support_word_lang_elim[elim]:
    assumes a ∈ support L
    obtains w where w ∈ L a ∈ support w
    using assms by auto

  lemma project_string_alphabet_weak[iff]:
    ⟦ w ∈ L; support L ⊆ E ⟧ ⟹ π E w = w
    by (meson support_word_lang subset_trans supp_word.supported_axioms
supported_def)
```

end

end

3 Refinement

```
theory Refinement
  imports
    Main
    Projection
begin


definition refines' :: 'a language ⇒ 'a set ⇒ 'a language ⇒ bool
  where
    refines' L1 X L2 ≡ π X L1 ⊆ L2
```

Introduced for DCR in [?, Def. 4.9].

Definition 8

```
definition refines
  where
    refines L1 L2 ≡ refines' L1 (support L2) L2


lemma refines_subset[intro]:
  fixes L1 L2 :: 'a language
  assumes L1 ⊆ L2
  shows refines L1 L2
proof -
  have support L1 ⊆ support L2 using support_lang_monotone assms by metis
  then have
    π (support L2) L1 = L1
    using assms by auto
  then have
    refines' L1 (support L2) L2
    using assms refines'_def by blast
  thus ?thesis
    using refines_def by auto
qed


lemma refines_intersection:
  shows refines (L1 ∩ L2) L1
  by auto

lemma refines_ident[intro,simp]:
  shows refines L L
  by auto

lemma refines_explicit[iff]:
```

```
  fixes P Q :: 'a language
  shows refines P Q ⟷ (project_L (support_L Q) P ⊆ Q)
  by (simp add: refines'_def refines_def)


end
theory Transition_System
  imports Main
begin


locale transition_system =
  fixes move :: 'state ⇒ 'action ⇒ 'state ⇒ bool
begin

    definition enabled :: 'state ⇒ 'action set where
      enabled s ≡ { a .  (∃s' . move s a s') }

  inductive run where
    empty[intro!]: run s0 []
  | move[intro!]:  ⟦ move s1 a s2 ; run s2 r ⟧ ⟹ run s1 ((a,s2) # r)



  inductive_cases run_elim': run s1 ((a, s2) # r)

  lemma run_elim[elim!]:
    assumes run s1 (x # r)
    obtains a s2 where move s1 a s2 run s2 r x = (a, s2)
    using assms run_elim'
    by (metis list.inject list.simps(3) run.simps)


  abbreviation target s r ≡ (if r = [] then s else snd (List.last r))

  lemma run_intro_append_move[intro]:
    assumes run s r move (target s r) a s'
    shows run s (r @ [(a, s')])
    using assms by (induction, auto, smt snd_conv)


  lemma run_intro_append_run[intro]:
    assumes run s r run (target s r) t
    shows run s (r @ t)
    using assms by (induction, auto, metis (full_types) snd_conv)


  lemma run_elim_append[elim]:
    assumes run s (r @ t)
```

```
    shows run s r
    using assms apply (induction r arbitrary: s)
    apply (simp add: run.empty)
    by (metis append.simps(2) append_is_Nil_conv list.simps(1) run.simps)


  inductive_set reachable for s where
    here[intro!]: s ∈ reachable s
  | there[intro!]: ⟦ s1 ∈ reachable s ; move s1 a s2 ⟧ ⟹ s2 ∈ reachable
s

  inductive_cases reachable_elim[elim]: s' ∈ reachable s


  lemma reachable_intro_append[trans]:
    assumes s1 ∈ reachable s0 s2 ∈ reachable s1
    shows s2 ∈ reachable s0
    using assms(2) apply (induction arbitrary: s1)
    using assms reachable.intros by auto

  lemma reachable_intro_rev[intro!]:
    assumes move s1 a s2  s ∈ reachable s2
    shows s ∈ reachable s1
    using assms
    by (meson reachable.intros reachable_intro_append)


  lemma reachable_run[iff]:
    (s ∈ reachable s0) = ((s = s0) ∨ (∃r . run s0 r ∧ target s0 r =
s))
  proof
    fix s assume A: s : reachable s0
    show (s = s0) ∨ (∃r . run s0 r ∧ target s0 r = s)
      using A run_intro_append_move by (induction rule:reachable.induct,
auto)
  next
    fix s assume A: (s = s0) ∨ (∃r . run s0 r ∧ target s0 r = s)
    then consider
        (here)  s = s0
      | (there) r where run s0 r ∧ target s0 r = s by auto
    then show s ∈ reachable s0 proof cases
      case here
      then show ?thesis by auto
    next
      case there
      then show ?thesis proof (induction r arbitrary: s0 s)
        case Nil
        then show ?case by auto
      next
```

```
        case (Cons x r)
        then obtain a s' where x = (a,s') move s0 a s' using run.simps

          by (meson list.inject list.simps(3))
        then have run s' r using Cons.prems by blast
        then have target s' r ∈ reachable s' using Cons.IH by simp
        then have target s0 (x # r) ∈ reachable s0
          by (metis move s0 a s' x = (a, s') last_ConsL last_ConsR reachable.here
reachable_intro_rev snd_conv)
        moreover have s = target s0 (x # r) using Cons by auto
        ultimately show ?case by simp
      qed
    qed
  qed

  lemma reachable_by_run:
    (s ∈ reachable s0) = (∃r . run s0 r ∧ ((s = s0) ∨ target s0 r =
s))
    using reachable_run by blast

  abbreviation trace_of r ≡ map fst r

  definition trace[iff]:
    trace s t ≡ ∃r. run s r ∧ t = trace_of r

  definition lang s = { t . trace s t }

  lemma lang_runs[iff]:
    (lang s) = { trace_of r | r . run s r }
    using lang_def by fastforce

end

end
theory Network
  imports
    Refinement
    HOL-Eisbach.Eisbach
    HOL-Eisbach.Eisbach_Tools
    Transition_System
begin
```

4 Networks

This section formalises Network as introduced in [**?**]. Networks are mechanisms to compose DCR graphs. The behaviour of a network is defined in terms of the transitions of its constituent graphs at the level of actions. Behaviour is composed by synchronising on *actions*, somewhat like in CSP.

4.1 Actions

```
datatype 'lab action =
    lim: Lim 'lab
  | unl: Unl 'lab
```

There are two kinds of action. A *limited action* `action.Lim l` indicates that a network is willing to allow the underlying action `l`, but will not produce it independently. An *unlimited action* `Unlim l` indicates that a network will produce that action independently.

Both kinds of action has an underlying label. In the paper, the operator to retrieve the underlying label is called @termγ; here, it is convenient to use the alphanumeric name @term`label`.

```
abbreviation label a ≡ (case a of Lim l ⇒ l | Unl l ⇒ l)
```

For parallel composition of networks, we define an operator which combines two actions `l1`, `l2` with the same label. The resulting action `l` is limited if both `l1` ad `l2` are, unlimited otherwise.

```
definition
   join l1 l2 l3 ≡
      label l1 = label l2 ∧
      label l2 = label l3 ∧
      lim l3 = (lim l1 ∧ lim l2)
```

```
join l1.0 l2.0 l3.0 ≡ label l1.0 = label l2.0 ∧ label l2.0 = label l3.0
∧ action.lim l3.0 = (action.lim l1.0 ∧ action.lim l2.0)
```

We demonstrate that join means what it should.

```
lemma join_characterisation:
  join x y z =
     (∃e .
       (x = Lim e ∧ y = Lim e ∧ z = Lim e) ∨
       (x = Lim e ∧ y = Unl e ∧ z = Unl e) ∨
       (x = Unl e ∧ y = Lim e ∧ z = Unl e) ∨
       (x = Unl e ∧ y = Unl e ∧ z = Unl e))
  by (join_cases x y z) locale Process =
  fixes labels :: 'proc ⇒ 'lab set
  and   excluded :: 'proc ⇒ 'lab set
  and   step :: 'proc ⇒ 'lab ⇒ 'proc ⇒ bool

  assumes
      step_lab:       step P l Q ⟹ l ∈ labels P
  and step_lab_pres:  step P1 l P2 ⟹ labels P1 = labels P2
  and step_det:       step P l Q1 ⟹ step P l Q2 ⟹ Q1 = Q2
```

Definition 5

begin

Technically, we have restricted ourselves to a single underlying process notation; however, note that if the sets of processes are disjoint, we can always combine two distinct notations into one simply by forming the union of their @termstep relations.

Figure 1

```
datatype ('l,'p) network =
      Proc 'p
    | Link 'l 'l set ('l,'p) network
    | Network ('l,'p) network ('l,'p) network
    | Zero
```

Syntactically, a network is a collection of processes, possibly linked with the `Link l ls N` construct.

Figure 2

```
fun alph where
    alph (Proc P) = labels P
  | alph (Link x xs N) = alph N - {x} ∪ xs
  | alph (Network N1 N2) = alph N1 ∪ alph N2
  | alph Zero = {}
```

```
abbreviation
  actions (N :: ('lab,'proc) network) ≡
     { Lim x | x .  x ∈ alph N } ∪ { Unl x | x . x ∈ alph N }
```

Figure 3

```
inductive nt :: ('lab,'proc) network ⇒ 'lab action ⇒ ('lab,'proc) network
⇒ bool (_ −_→ _)
  where
    Excl:  ⟦ x ∈ labels P ; x ∈ excluded P ⟧ ⟹
                nt (Proc P) (Lim x) (Proc P)
  | Link1: ⟦ nt N (Unl x) N' ; l ∈ xs ⟧ ⟹
                nt (Link x xs N) (Lim l) (Link x xs N')
  | Link2: ⟦ nt N l N' ; label l ∉ {x} ∪ xs ⟧ ⟹
                nt (Link x xs N) l (Link x xs N')
  | Step:  ⟦ step P1 x P2; x ∉ excluded P1 ⟧ ⟹ nt (Proc P1) (Unl x)
(Proc P2)
  | Sync:  ⟦ nt N1 l1 N1' ; nt N2 l2 N2' ; join l1 l2 l ⟧ ⟹
                nt (Network N1 N2) l (Network N1' N2')
  | Pass1: ⟦ label l ∉ alph N1 ; nt N2 l N2' ⟧ ⟹
                nt (Network N1 N2) l (Network N1 N2')
  | Pass2: ⟦ nt N1 l N1' ; label l ∉ alph N2 ⟧ ⟹
                nt (Network N1 N2) l (Network N1' N2)
```

$$\frac{\texttt{x} \in \texttt{labels P} \wedge \texttt{x} \in \texttt{excluded P}}{\texttt{Proc P} \; -\texttt{action.Lim x}\rightarrow \; \texttt{Proc P}}$$

$$\frac{\texttt{N} \; -\texttt{Unl x}\rightarrow \; \texttt{N'} \wedge \texttt{l} \in \texttt{xs}}{\texttt{Link x xs N} \; -\texttt{action.Lim l}\rightarrow \; \texttt{Link x xs N'}}$$

$$\frac{\texttt{N} \; -\texttt{l}\rightarrow \; \texttt{N'} \wedge \texttt{label l} \notin \{\texttt{x}\} \cup \texttt{xs}}{\texttt{Link x xs N} \; -\texttt{l}\rightarrow \; \texttt{Link x xs N'}}$$

$$\frac{\texttt{step P1.0 x P2.0} \wedge \texttt{x} \notin \texttt{excluded P1.0}}{\texttt{Proc P1.0} \; -\texttt{Unl x}\rightarrow \; \texttt{Proc P2.0}}$$

```
N1.0 −l1.0→ N1' ∧ N2.0 −l2.0→ N2' ∧ join l1.0 l2.0 l
─────────────────────────────────────────────────────
        Network N1.0 N2.0 −l→ Network N1' N2'

        label l ∉ alph N1.0 ∧ N2.0 −l→ N2'
       ─────────────────────────────────────
       Network N1.0 N2.0 −l→ Network N1.0 N2'

        N1.0 −l→ N1' ∧ label l ∉ alph N2.0
       ─────────────────────────────────────
       Network N1.0 N2.0 −l→ Network N1' N2.0
```

```
inductive_cases nt_network
  [elim, consumes 1, case_names Sync Pass1 Pass2]:
  nt (Network N1 N2) t N'

inductive_cases nt_proc[elim]:
  nt (Proc P) t N'

inductive_cases nt_link[elim]:
  nt (Link x xs N) t N'


method rule_inversion uses nt =
  (   (cases rule: nt_network[ case_names Sync Pass1 Pass2])
    | (cases rule: nt_proc[consumes 1, case_names Excl Step])
    | (cases rule: nt_link[consumes 1, case_names Link1 Link2])
  )



lemma nt_action_in_alph[elim]:
  assumes t: nt N1 l N2
  shows label l ∈ alph N1
  using assms proof (induction N1 arbitrary: N2 l)

  case (Proc P)
  then show ?case
    apply (cases rule: nt_proc)
    using step_lab by auto
next
  case (Link x xs N)
  show ?case using Link(2) proof rule_inversion
    case (Link1 l N')
    then show ?thesis by simp
  next
    case (Link2 N')
    then show ?thesis using Link.IH by auto
  qed

next
  case (Network N11 N12)
  show ?case using Network(3) proof (rule_inversion)
      case (Sync l1 N1' l2 N2')
```

```
      then show ?thesis
        using Network.IH  join_def by (metis Un_iff alph.simps(3))
    next
      case (Pass1 N2')
      then show ?thesis
        using Network.IH(2) by auto
    next
      case (Pass2 N1')
      then show ?thesis
        using Network.IH(1) by auto
    qed

next
  case Zero
  then show ?case using nt.simps by blast

qed


lemma nt_action_not_in_alph[elim]:
  assumes label l ∉ alph N1
  shows ¬ nt N1 l N2
  using assms nt_action_in_alph by blast


lemma nt_alph_preserved[elim]:
  assumes nt N1 l N2
  shows (e ∈ alph N1) = (e ∈ alph N2)
using assms proof (induction N1 arbitrary: l N2)
  case (Proc x1 x2)
  then show ?case using step_lab_pres by auto
next
  case (Link x xs N)
  then show ?case by auto
next
  case (Network N11 N12)
  then show ?case by auto
next
  case Zero
  then show ?case using nt.simps by blast
qed



lemma nt_proc_proc[elim]:
  assumes nt (Proc P) a N
  shows ∃! P' . N = Proc P'
  using assms nt.cases step_det by auto
```

```
lemma weak_preimage_excluded:
  assumes nt (Proc P) (Lim x) Q
  shows x ∈ excluded P and Q = Proc P
using nt_proc assms by blast+


lemma nt_proc_action_deterministic[elim]:
  assumes nt (Proc P) l1 N1 nt (Proc P) l2 N2 label l1 = label l2
  shows   l1 = l2 ∧ N1 = N2
  proof -

    from assms show ?thesis proof (cases l1)

    case (Lim x)
    then have *: x ∈ excluded P using assms by blast
    have l2 = Lim x using assms(2) proof (rule_inversion)
      case (Excl x')
      then have x = x' using assms by (simp add: Lim)
      then show ?thesis by (simp add: local.Excl(1))
    next
      case (Step x' Q)
      then have x = x' using assms by (simp add: Lim)
      then show ?thesis using * Step by simp
    qed

    thus ?thesis using Lim assms by blast
  next
    case (Unl x)
    then have *: x ∉ excluded P using assms by blast
    have l1 = l2 using assms proof (rule_inversion)
      case (Excl x')
      then show ?thesis using * Unl assms by auto
    next
      case (Step x' P2)
      then show ?thesis using Unl assms by auto
    qed
    thus ?thesis using Unl assms
      by (metis action.sel(2) nt_proc step_det weak_preimage_excluded(2))
  qed
qed lemma nt_action_deterministic[elim]:
Lemma 11
  assumes nt N x1 N1 nt N x2 N2 label x1 = label x2
  shows N1 = N2
using assms proof (induction N arbitrary: x1 x2 N1 N2)
  case (Proc T M)
    then show ?case
      using Proc.prems(1) Proc.prems(2) Proc.prems(3) nt_proc_action_deterministic
by simp
```

```
  next case (Link l ls N)
    show ?case using nt (Link l ls N) x1 N1 proof (rule_inversion)
      case (Link1 l' N')
      then show ?thesis
        using Link.prems nt_action_in_alph
        using Link.IH Link.prems(2) Link.prems(3) by auto
    next
      case (Link2 N')
      then show ?thesis
        using Link.IH Link.prems by auto
    qed

  next case (Network M1 M2)
    show N1 = N2  using nt (Network M1 M2) x1 N1 proof (rule_inversion)

      case (Sync x11 M11 x12 M12)
      then have
        label x11 = label x1 label x12 = label x1
        by (simp_all add: join_def)

      then have
        eq: label x11 = label x2 label x12 = label x2
        by (simp_all add: Network.prems(3))

      then have N2 = Network M11 M12
          proof (cases rule: nt_network[OF nt (Network M1 M2) x2 N2])
            case (1 x1 N1' x2 N2')
            then show ?thesis using eq Network Sync 1
              using join_def by smt
          next
            case (2 N2')
            then show ?thesis
              using Network.prems(3) label x11 = label x1 local.Sync(2)
nt_action_in_alph by fastforce
          next
            case (3 N1')
            then show ?thesis
              using Network.prems(3) label x12 = label x1 local.Sync(3)
nt_action_in_alph by fastforce
          qed

        thus ?thesis
          by (simp add: local.Sync(1))

    next
      case p1: (Pass1 N2')
      show ?thesis
        using nt (Network M1 M2) x2 N2 proof (cases rule: nt_network)
        case (Sync x1 N1' x2 N2')
```

```
        then show ?thesis
          using Network.prems join_def nt_action_not_in_alph p1(2) by
smt
      next
        case (Pass1 N2')
        then show ?thesis using p1
          using Network.IH(2) Network.prems by blast
        next
        case (Pass2 N1')
        then show ?thesis using p1
          using Network.prems(3) nt_action_not_in_alph by auto
        qed

    next
      case p2: (Pass2 N1')
      show ?thesis
         using nt (Network M1 M2) x2 N2 proof (cases rule: nt_network)
         case (Sync x1 N1' x2 N2')
         then show ?thesis
           using Network.prems join_def nt_action_not_in_alph p2(3) by
metis
      next
         case (Pass1 N2')
         then show ?thesis using p2
           using Network.prems(3) nt_action_in_alph by auto
      next
         case (Pass2 N1')
         then show ?thesis using p2
           using Network.IH(1) Network.prems by blast
      qed
    qed

  next case Zero
    then show N1 = N2
      using nt.simps by blast
  qed


lemma
  assumes nt (Link l ls N1) x N2
  shows unl x ⟹ label x ∉ ls
  using assms by (rule_inversion, auto)
```

5 Transition semantic

```
definition nt_enabled
  where
    nt_enabled e N ≡ ∃N'. nt N e N'
```

```
definition nt_execute
  where
    nt_execute e N ≡ THE N' . nt N e N'

lemma nt_function:
  assumes nt N l N1 nt N l N2
  shows N1 = N2 using assms nt_action_deterministic  nt_enabled_def[iff]
by blast

lemma nt_enabled_may_execute:
  assumes nt_enabled e N
  shows ∃! N' . nt N e N'
  using  assms nt_enabled_def nt_function  nt_enabled_def[iff] by auto


lemma nt_execute_function[iff]:
  assumes nt_enabled e N
  shows (nt_execute e N = N') = nt N e N'
  using  nt_enabled_def[iff] assms nt_enabled_may_execute nt_execute_def
theI_unique
  by metis

lemma nt_to_execute[iff]:
  assumes nt N e N'
  shows nt_execute e N = N'
  using assms nt_enabled_def nt_execute_function by blast


interpretation nt: transition_system
  nt .



lemma nt_enabled a N = (a : nt.enabled N)
  by (simp add: nt_enabled_def transition_system.enabled_def) definition
```

Definition 9

```
unlimited :: 'lab set ⇒ ('lab, 'proc) network ⇒ bool where
  unlimited X N0 ≡
    ∀N' a . label a ∈ X ∧ N' ∈ nt.reachable N0 ∧ a ∈ nt.enabled N'
            ⟶ unl a


lemma limitation_preservation[intro]:
  assumes unlimited X N1 nt N1 l N2
  shows unlimited X N2
  by (metis assms unlimited_def nt.reachable_intro_rev)


lemma nt_action_in_actions[elim]:
```

```
  assumes nt N1 l N2
  shows l ∈ actions N1
proof -
  have label l ∈ alph N1 using assms by auto
  thus ?thesis by (cases l, auto)
qed

lemma action_iff_actions:
  shows (label l ∈ alph N) = (l ∈ actions N)
  using alph.simps by (cases l, simp_all)

lemma nt_trace_in_actions[intro]:
  fixes N :: ('lab,'proc) network
  assumes nt.run N r
  shows set (nt.trace_of r) ⊆ actions N
using assms proof (induction r arbitrary: N rule: list.induct)
  case Nil
  then show ?case by simp
next
  case (Cons x r)
  then obtain N' a where N':
    nt N a N' nt.run N' r x = (a, N') by auto

  then have
    set (nt.trace_of r) ⊆ actions N
    using Cons.IH nt_alph_preserved by blast

  moreover with N' have
    a ∈ actions N
    using nt_action_in_actions by blast

  ultimately show ?case
    using x = (a, N') by auto
qed

lemma nt_preserves_actions:
  assumes nt N1 l N2
  shows actions N1 = actions N2 using assms
  using nt_alph_preserved by auto

lemma nt_actions:
  support (nt.lang N) ⊆ actions N
proof (rule subsetI)
  fix l assume
    l ∈ support (nt.lang N)

  then have
    l ∈ support { nt.trace_of r | r . nt.run N r } by simp
```

```
  then obtain t r where
    l ∈ set t t ∈ { nt.trace_of r | t . nt.run N r }
    using support_w_def support_L_def by (smt Union_iff mem_Collect_eq)

  moreover then have
    nt.run N r by simp

  ultimately show
    l ∈ actions N using nt_trace_in_actions by blast
qed
```

```
fun select where
    select f [] = []
  | select f (x # xs) =
      (case f x of
         Some x ⇒ x # select f xs
       | None ⇒ select f xs)

abbreviation proj1 where
    proj1 X x ≡
      (case x of
          (a, Network N1 N2) ⇒
            (if a ∈ X then Some (a, N1) else None)
        | _ ⇒ None)

abbreviation
  Γ X t ≡ (select (proj1 X) t)

abbreviation
  is_network N ≡
    (case N of (Network _ _) ⇒ True | _ ⇒ False)
```

We do not define "trace" independently, going instead directly for the notion of language.

```
definition
  lang (N :: ('lab,'proc) network) ≡
    { map label t | t . nt.trace N t ∧ list_all unl t }
```

Definition 7

```
lemma independent_run:
  fixes N1 :: ('lab,'proc) network
  assumes unlimited X N1 alph N2 ∩ alph N1 ⊆ X
  assumes nt.run (Network N1 N2) t
  shows   nt.run N1 (Γ (actions N1) t) ∧
           nt.trace_of (Γ (actions N1) t) = π (actions N1) (nt.trace_of
t)
using assms proof (induction t arbitrary: N1 N2)
```

```
  case Nil
  then show ?case
    using nt.empty supp_word.tight
    by (simp add: language_simps(3))
next
  case (Cons x t)
  then obtain N l  where t:
    nt (Network N1 N2) l N and [simp]: x = (l,N) and tr: nt.run N t
    by auto

  show ?case using t proof (cases rule: nt_network)
    case (Sync l1 M1 l2 M2)

    then have
      label l1 ∈ alph N1
      using nt_action_in_alph by simp

    then have
      l1 ∈ actions N1 using action_iff_actions by simp

    from Sync have r:
      nt.run (Network M1 M2) t using Cons.IH
      using tr by blast

    have alph N1 = alph M1 alph N2 = alph M2
      using nt_alph_preserved Sync assms by auto

    moreover then have
      alph M1 ∩ alph M2 ⊆ X using Cons by auto

    moreover ultimately have alphs:
      unlimited X M1 alph M1 ∩ alph M2 ⊆ X
      using Cons Sync by auto

    then have
      nt.run M1 (Γ (actions M1) t) and **:
      nt.trace_of (Γ (actions M1) t) = π (actions M1) (nt.trace_of t)
      using Cons.IH r by auto

    then have r0:
      nt.run N1 ((l1, M1) # (Γ (actions M1) t))
      using nt.run.intros Sync by blast

    have
      actions M1 = actions N1
        using nt N1 l1 M1 nt_preserves_actions by simp

    moreover have
      l1 = l
```

```
    proof -
      have
        label l1 ∈ alph M2 using Sync
        using alph N2 = alph M2 join_def nt_action_not_in_alph
        by metis
      moreover have
        label l1 ∈ alph M1
        using alph N1 = alph M1 label l1 ∈ alph N1 by blast
      ultimately have
        label l1 ∈ X
        using alphs by blast
      with unlimited X N1 Sync(2) have
          unl l1
          using unlimited_def nt.enabled_def assms by auto
        thus
          l1 = l
          using join l1 l2 l by join
      qed

    ultimately have
      nt.run N1 ((l, M1) # (Γ (actions N1) t))
      using r0 by auto

    moreover with l1 = l have ***:
      proj1 (actions N1) x = Some (l, M1)
      using Sync nt_action_in_actions by simp

    ultimately have nt.run N1 (Γ (actions N1) (x # t))
      by auto

    moreover have
      nt.trace_of (Γ (actions N1) (x # t)) = π (actions N1) (nt.trace_of
(x # t))
    proof -
        have
          nt.trace_of (Γ (actions N1) (x # t)) =
           nt.trace_of ((l, M1) # (Γ (actions N1) t))
          using *** by auto
        also have
          ... = l # nt.trace_of (Γ (actions N1) t) by simp
        also have
          ... = l # nt.trace_of (Γ (actions M1) t) using actions M1 =
actions N1 by auto
        also have
          ... = l # π (actions M1) (nt.trace_of t) using ** by auto
        also have
          ... = l # π (actions N1) (nt.trace_of t) using actions M1 =
actions N1 by simp
        also have
```

```
          ... = π (actions N1) (nt.trace_of (x # t))  using x = (l, N)
l1 ∈ actions N1 l1 = l
          by (simp add: language_simps)
        finally show ?thesis .
      qed

    ultimately show ?thesis by simp
 next
    case (Pass1 M2)

    moreover then have
      label l ∉ alph N1 by blast

    moreover have eq:
      Γ (actions N1) (x # t) = Γ (actions N1) t
      using Pass1 by auto

    moreover have
      alph M2 ∩ alph N1 ⊆ X
      using Cons.prems(2) calculation(3) nt_alph_preserved by auto

    ultimately have *:
      nt.run N1 (Γ (actions N1) (x # t)) ∧
       nt.trace_of (Γ (actions N1) t) = π (actions N1) (nt.trace_of t)

      using Cons.IH[of N1 M2] Cons tr by simp

    then have
      l ∉ actions N1 using action_iff_actions label l ∉ alph N1 by simp


    { have
        nt.trace_of (Γ (actions N1) (x # t)) =
         nt.trace_of (Γ (actions N1) t) using eq by simp
      also have
        ... = π (actions N1) (nt.trace_of t) using * by simp
      also have
        ... = π (actions N1) (l # nt.trace_of t) using l ∉ actions N1

        by (simp add: language_simps)
      also have
        ... = π (actions N1) (nt.trace_of (x # t))
        using l ∉ actions N1 x = (l, N) by simp
      finally have
        nt.trace_of (Γ (actions N1) (x # t)) =
         π (actions N1) (nt.trace_of (x # t)) .
    }
    with * show
      nt.run N1 (Γ (actions N1) (x # t)) ∧
```

```
       nt.trace_of (Γ (actions N1) (x # t)) = π (actions N1) (nt.trace_of
(x # t))
      by auto

 next
    case (Pass2 M1)

    then have
      l ∈ (actions N1)
      using nt_action_in_actions by simp

    then have eq:
      Γ (actions N1) (x # t) = (l,M1) # Γ (actions N1) t
      using Pass2 tr by auto

    have unlimited X M1
      using Cons.prems local.Pass2 by blast

    then have
      alph N2 ∩ alph M1 ⊆ X
      using Cons.prems(2) local.Pass2(2) nt_alph_preserved by auto

    moreover have
      nt.run (Network M1 N2) t
      using local.Pass2(1) tr by blast

    ultimately have *:
      nt.run M1 (Γ (actions M1) t) ∧
       nt.trace_of (Γ (actions M1) t) = π (actions M1) (nt.trace_of t)

      using Cons.IH [of M1 N2] unlimited X M1 by blast

    then have **: nt.run N1 ((l,M1) # (Γ (actions M1) t))
      using nt N1 l M1  nt_enabled_def nt_to_execute by blast

    moreover have actions M1 = actions N1
      using nt N1 l M1 nt_alph_preserved by simp

    moreover have
      nt.trace_of (Γ (actions N1) (x # t)) =
      π (actions N1) (nt.trace_of (x # t)) proof -
      have
        nt.trace_of (Γ (actions N1) (x # t)) =
         l # nt.trace_of (Γ (actions N1) t) using l ∈ actions N1
        using eq by auto
      also have
        ... = l # nt.trace_of (Γ (actions M1) t)
        using actions M1 = actions N1 by auto
      also have
```

```
      ... = l # π (actions N1) (nt.trace_of t)
      using * by (simp add: actions M1 = actions N1)
    also have
      ... = π (actions N1) (nt.trace_of (x # t))
      using l ∈ actions N1 by (auto simp add: language_simps)
    finally show ?thesis .
  qed

  ultimately show ?thesis
    using eq ** by force
 qed
qed lemma independent_trace:
```
Lemma 12
```
 assumes unlimited X N1 alph N2 ∩ alph N1 ⊆ X
 assumes nt.trace (Network N1 N2) t
 shows   nt.trace N1 (π (actions N1) t)
proof -
 obtain r where
   nt.run (Network N1 N2) r and t = nt.trace_of r
   using assms nt.trace by auto
 then have
   nt.run N1 (Γ (actions N1) r) ∧
    nt.trace_of (Γ (actions N1) r) = π (actions N1) t
   using independent_run assms by auto
 then show
   nt.trace N1 (π (actions N1) t) by auto
qed


lemma independent_string:
 assumes unlimited X N1 alph N2 ∩ alph N1 ⊆ X
 and     s ∈ lang (Network N1 N2)
 shows   π (alph N1) s ∈ lang N1
proof -
 note language_simps[simp]

 obtain t where *:
   nt.trace (Network N1 N2) t list_all unl t s = map label t
   using assms lang_def by auto

 let ?t = π (actions N1) t

 from * have
   nt.trace N1 ?t
   using independent_trace assms by simp

 moreover have list_all unl ?t
   using list_all unl t by ( induction t, auto)

 ultimately have
```

```
    map label ?t ∈ lang N1
    using lang_def by auto

  have
    map label (π (actions N1) t) = π (alph N1) (map label t)
    proof (induction t)
      case Nil
      then show ?case by simp
    next
      case (Cons a t)
      then show ?case proof (cases a ∈ actions N1)
        case True
        then show ?thesis
          using Cons.IH by auto
      next
        case False
        then have
          π (alph N1) (map label (a # t)) =
           π (alph N1) (label a # map label t)
          by simp
        also have
          ... =  π (alph N1) (map label t)
          proof -
            have label a ∉ alph N1
              using False action_iff_actions by auto
            thus ?thesis by auto
          qed
        also have
          ... = map label (π (actions N1) t)
          using Cons.IH by auto
        also have
          ... = map label (π (actions N1) (a # t))
          using False by auto
        finally show ?thesis ..
      qed
    qed

  with * show π (alph N1) s ∈ lang N1
    using lang_def map label ?t ∈ lang N1 by auto
qed theorem refinement:
```

Theorem 13

```
  fixes T M
  assumes unlimited X P
  assumes alph N ∩ alph P ⊆ X
  shows   refines' (lang (Network P N)) (alph P) (lang P)

  using refines_def refines'_def
proof -
  have π (alph P) (lang (Network P N)) ⊆ lang P proof
    fix t'
```

```
    assume t' ∈ π (alph P) (lang (Network P N))

    then obtain t where *:
      nt.trace (Network P N) t list_all unl t t' = π (alph P) (map label
t)
      using lang_def project_L_def mem_Collect_eq by (auto simp add: language_simps)

    then have
      map label t ∈ lang (Network P N) using lang_def by auto

    then have
      π (alph P) (map label t) ∈ lang P
      using independent_string assms by blast

    thus t' ∈ lang P using * by simp
  qed

  thus ?thesis
    using refines'_def by blast
qed


lemma proc_reachable:
  assumes N ∈ nt.reachable (Proc P)
  shows ∃P'. N = Proc P'
  using assms by (induction, simp, fastforce) lemma unlimited_proc:
```

Lemma 10

```
  fixes r
  assumes ⋀ N . N ∈ nt.reachable (Proc P) ⟹
             (⋀Q . N = Proc Q ⟹ excluded Q ∩ X = {})
  shows unlimited X (Proc P)
proof -
  have ∀N' a.
        label a ∈ X ∧ N' ∈ nt.reachable (Proc P) ∧ a ∈ nt.enabled N'
          ⟶ unl a
  proof -
    {
    fix N a
    assume
          label a ∈ X
      and re: N ∈ nt.reachable (Proc P)
      and a ∈ nt.enabled N

    then obtain P' N' where
      nt: nt (Proc P') a N' and eq: N = Proc P'
      using a ∈ nt.enabled N nt.enabled_def
      by (smt mem_Collect_eq proc_reachable)

    then have *: label a ∉ excluded P'
      using label a ∈ X assms re by blast
```

```
    have unl a
      using nt (Proc P') a N' by (rule_inversion; cases a; insert *; auto)

    }
    thus ?thesis by blast
  qed
  thus ?thesis using unlimited_def by simp
qed

end

end
```