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

Søren Debois

July 2, 2020

Contents


```
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
  ⟨proof⟩

interpretation supp_language: supported
  support_L :: 'a language ⇒ 'a set
  project_L :: 'a set ⇒ 'a language ⇒ 'a language
  ⟨proof⟩

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 }
    ⟨proof⟩

  lemma support_word_lang[simp,elim]:
    w ∈ L ⟹ support w ⊆ support L
    ⟨proof⟩

  lemma support_lang_empty[simp]:
    π E {}  = {} ⟨proof⟩

  lemma support_lang_monotone:
    L1 ⊆ L2 ⟹ support L1 ⊆ support L2
    ⟨proof⟩

  lemma support_word_lang_elim[elim]:
    assumes a ∈ support L
    obtains w where w ∈ L a ∈ support w
    ⟨proof⟩

  lemma project_string_alphabet_weak[iff]:
    ⟦ w ∈ L; support L ⊆ E ⟧ ⟹ π E w = w
    ⟨proof⟩


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 refines
Definition 8  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⟩


lemma refines_intersection:
  shows refines (L1 ∩ L2) L1
  ⟨proof⟩

lemma refines_ident[intro,simp]:
  shows refines L L
  ⟨proof⟩

lemma refines_explicit[iff]:
  fixes P Q :: 'a language
  shows refines P Q ⟷ (project_L (support_L Q) P ⊆ Q)
  ⟨proof⟩


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)
  ⟨proof⟩


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')])
  ⟨proof⟩


lemma run_intro_append_run[intro]:
  assumes run s r run (target s r) t
  shows run s (r @ t)
  ⟨proof⟩


lemma run_elim_append[elim]:
  assumes run s (r @ t)
  shows run s r
  ⟨proof⟩


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
  ⟨proof⟩

lemma reachable_intro_rev[intro!]:
  assumes move s1 a s2  s ∈ reachable s2
  shows s ∈ reachable s1
  ⟨proof⟩


lemma reachable_run[iff]:
  (s ∈ reachable s0) = ((s = s0) ∨ (∃r . run s0 r ∧ target s0 r =
s))
⟨proof⟩

lemma reachable_by_run:
```

```
    (s ∈ reachable s0) = (∃r . run s0 r ∧ ((s = s0) ∨ target s0 r =
s))
    ⟨proof⟩

  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 }
    ⟨proof⟩

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))
  ⟨proof⟩ locale Process =
Definition 5
  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
```

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 @term`step` 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)
```

```
         x ∈ labels P ∧ x ∈ excluded P
         ─────────────────────────────
         Proc P −action.Lim x→ Proc P

              N −Unl x→ N' ∧ l ∈ xs
      ──────────────────────────────────────
      Link x xs N −action.Lim l→ Link x xs N'

          N −l→ N' ∧ label l ∉ {x} ∪ xs
          ─────────────────────────────
           Link x xs N −l→ Link x xs N'

       step P1.0 x P2.0 ∧ x ∉ excluded P1.0
       ────────────────────────────────────
          Proc P1.0 −Unl x→ 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
  ⟨*proof*⟩

lemma nt_action_not_in_alph[elim]:
  assumes label l ∉ alph N1
  shows ¬ nt N1 l N2
  ⟨*proof*⟩

lemma nt_alph_preserved[elim]:
  assumes nt N1 l N2
  shows (e ∈ alph N1) = (e ∈ alph N2)
⟨*proof*⟩

lemma nt_proc_proc[elim]:
  assumes nt (Proc P) a N
  shows ∃! P' . N = Proc P'
  ⟨*proof*⟩

lemma weak_preimage_excluded:
  assumes nt (Proc P) (Lim x) Q
  shows x ∈ excluded P **and** Q = Proc P
⟨*proof*⟩

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*⟩ **lemma** nt_action_deterministic[elim]:
Lemma 11 **assumes** nt N x1 N1 nt N x2 N2 label x1 = label x2

shows N1 = N2
⟨*proof*⟩

lemma
assumes nt (Link l ls N1) x N2
shows unl x ⟹ label x ∉ ls
⟨*proof*⟩

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 ⟨*proof*⟩

lemma nt_enabled_may_execute:
assumes nt_enabled e N
shows ∃! N' . nt N e N'
⟨*proof*⟩

lemma nt_execute_function[iff]:
assumes nt_enabled e N
shows (nt_execute e N = N') = nt N e N'
⟨*proof*⟩

lemma nt_to_execute[iff]:
assumes nt N e N'
shows nt_execute e N = N'
⟨*proof*⟩

interpretation nt: transition_system
nt ⟨*proof*⟩

lemma nt_enabled a N = (a : nt.enabled N)
⟨*proof*⟩ **definition** unlimited :: 'lab set ⇒ ('lab, 'proc) network ⇒ bool
Definition 9 **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
  ⟨*proof*⟩

lemma nt_action_in_actions[elim]:
  assumes nt N1 l N2
  shows l ∈ actions N1
⟨*proof*⟩

lemma action_iff_actions:
  shows (label l ∈ alph N) = (l ∈ actions N)
  ⟨*proof*⟩

lemma nt_trace_in_actions[intro]:
  fixes N :: ('lab,'proc) network
  assumes nt.run N r
  shows set (nt.trace_of r) ⊆ actions N
⟨*proof*⟩

lemma nt_preserves_actions:
  assumes nt N1 l N2
  shows actions N1 = actions N2 ⟨*proof*⟩

lemma nt_actions:
  support (nt.lang N) ⊆ actions N
⟨*proof*⟩

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

Definition 7
```
lang (N :: ('lab,'proc) network) ≡
  { map label t | t . nt.trace N t ∧ list_all unl t }
```

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)
```
⟨*proof*⟩ **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*⟩

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*⟩ **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)
```

⟨*proof*⟩

lemma proc_reachable:
```
assumes N ∈ nt.reachable (Proc P)
shows ∃P'. N = Proc P'
```
⟨*proof*⟩ **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*⟩

end

end