Documentation of RDA Library

A. Fontaine, A. Zemmari
Université de Bordeaux - LaBRI
351 cours de la Libération, 33405 Talence, France
{allyx.fontaine,akka.zemmari}@labri.fr

March 31, 2014

Contents

Chapter 1

RDA: Randomised Distributed Algorithms

1.1 Description

This is our work on the formal specification and analysis of randomised distributed algorithms. The library RDA is composed of the following parts :

- prelude: tools about ssreflect and Alea
- graph: tools to reason about graphs
- example: simple examples of use of Alea
- ra: tools to reason about randomised algorithms
- rda: tools to reason about randomised distributed algorithms

1.2 Install

A first draft implementation in Coq (http://coq.inria.fr/) and ssreflect (http://www.msr-inria.inria produced by our team is available at: http://www.labri.fr/perso/fontaine/RDA.

This is clearly a work in progress. Many definitions and proofs are too long and are being improved little by little.

To compile: cd <project root>/RDA/; make

You need coq8.4 and ssreflect1.4.

Your arborescence is the following:

```
<project_root>/RDA/prelude
<project_root>/RDA/graph
<project_root>/RDA/example
<project_root>/RDA/ra
<project_root>/RDA/rda
```

You will need to put an environment variable named `$ALEA_LIB` with the path of Alea directory leading to /src and /Continue (`http://www.lri.fr/~paulin/ALEA/`).

Chapter 2

Library my_ssr

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat.
Require Import fintype finset fingraph seq.
Import Prenex Implicits.
```

Lemmas to complete ssr libraries

```
Lemma leqMinus : ∀ x y, (x < y)%nat → (x ≤ y - 1)%nat.

Lemma lt_le_1 : ∀ (i n:nat), (i < n)%nat → (i ≤ n-1)%coq_nat.

Lemma irrefl_mem : ∀ (F:Type) (r:rel F) (f:F),
 irreflexive r → f \notin (r f).

Delimit Scope seq_scope with SEQ.
Open Scope seq_scope.

Section Rem.

Variables (T : eqType) (x : T).

Lemma rem_impl (s: seq_predType T) (u v:T) : u \notin s → u \notin (rem v s).

Lemma rem_mem_not : ∀ (l:seq T) (i a:T),
  i \in l → i \notin seq.rem a l → i = a.

End Rem.

Section seq.
Variables (T' : finType).
Variables (T: eqType).

Lemma index_cons : ∀ (v t:T) s, (t==v)=false →
 (index v (t::s)) = (index v s) .+1.

Lemma index_neq : ∀ (v u: T'),
 (v!=u) → index v (enum T') != index u (enum T').

Lemma index_neq_in : ∀ l (x y:T'), x \in l →
 x != y → index x l != index y l.
```

```
Lemma index_set_nth : ∀ (v w1 w2: T') x n,
  w1 != v → w2 != v →
  index v (set_nth w1 (nseq n w2) x v) = x.

Lemma rem_T : ∀ v, v \notin (rem v (enum T')).

Lemma count_notin : ∀ (x:T) s, x \notin s → count (pred1 x) s = O.

Lemma index_map2 : ∀ (T1 T2:eqType)(f:T1 → T2) (s:seq T1) (x:T1) (y:T2),
  x \in s → index x s = index y [seq f i | i ← s] → f x = y.

End seq.

Section fun1.

Require Import Arith.
Require Import Compare_dec.

Variable D : Type.
Variable def: D.
Variable i: nat.
Variable f : ordinal i → D.

Definition funbound (j:nat) : D.
Defined.

End fun1.

Section fun2.

Variable D : Type.
Variable g : nat → D.

Definition fbound {i:nat}{j:ordinal i} := g (nat_of_ord j).

End fun2.

Lemma fbound1 : ∀ D (f: nat→D) i j (H : j < i),
  f j = @fbound D f i (Ordinal H).

Lemma funbound1 : ∀ D d i (f : ordinal i → D)(x:ordinal i),
  f x = funbound _ d i f (nat_of_ord x).

Lemma disjoint_set : ∀ (T:finType) (t v:T), t != v →
  disjoint (mem [set t]) (mem [set v]).
```

Chapter 3

Library my_ssralea

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat bigop.
Require Import fintype finset fingraph seq.
Import Prenex Implicits.
Require Import my_ssr.
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Set Implicit Arguments.
```

Adjusting of bigop with Oeq

```
Lemma big_mkconds : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
  (I : Type) (r : seq I) (P : pred I) (F : I → R),
  (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
  (∀ a, (Oeq (op idx a) a)) →
  (∀ a, (Oeq (op a idx) a)) →
  (∀ a b c d, a == b → c == d → op a c == op b d) →
    Oeq (\big[op/idx]_(i ← r | P i) F i)
     (\big[op/idx]_(i ← r) (if P i then F i else idx)).
Lemma big_mkcondrs : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
  (I : Type) (r : seq I) (P Q: pred I) (F : I → R),
  (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
  (∀ a, (Oeq (op idx a) a)) →
  (∀ a, (Oeq (op a idx) a)) →
  (∀ a b c d, a == b → c == d → op a c == op b d) →
  Oeq (\big[op/idx]_(i ← r | P i && Q i) F i)
     (\big[op/idx]_(i ← r | P i) (if Q i then F i else idx)).
Lemma big_splits : ∀ (R : Type) (ordR: Ccpo.ord R)
```

```
  (idx : R) (op : R → R → R)
  (I : Type) (r : seq I) (P : pred I) (F1 F2 : I → R),
   (∀ a b, (Oeq (op a b) (op b a))) →
   (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
   (∀ a, (Oeq (op idx a) a)) →
   (∀ a, (Oeq (op a idx) a)) →
   (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op a c) (op b d))) →
    Oeq (\big[op/idx]_(i ← r | P i) op (F1 i) (F2 i))
        (op (\big[op/idx]_(i ← r | P i) F1 i)
         (\big[op/idx]_(i ← r | P i) F2 i)).
Lemma eq_bigrs : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
  (I : Type) (r : seq I) (P : pred I) (F1 F2 : I → R),
   (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op a c) (op b d))) →
   (∀ i : I, P i → F1 i == F2 i) →
    \big[op/idx]_(i ← r | P i) F1 i == \big[op/idx]_(i ← r | P i) F2 i.
Lemma bigIDs : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
  (I : Type) (r : seq I) (a P : pred I) (F : I → R),
   (∀ a b, (Oeq (op a b) (op b a))) →
   (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
   (∀ a, (Oeq (op idx a) a)) →
   (∀ a, (Oeq (op a idx) a)) →
   (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op a c) (op b d))) →
     (Oeq (\big[op/idx]_(i ← r | P i) F i)
          (op (\big[op/idx]_(i ← r | P i && a i) F i)
              (\big[op/idx]_(i ← r | P i && ~~ a i) F i))).
Lemma big_seq1s : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R→R→R)
   (I : Type) (i : I) (F : I → R),
     (∀ a, (Oeq (op a idx) a)) →
      (Oeq (\big[op/idx]_(j ← [:: i]) F j) (F i)).
Lemma big_pred1_eqs : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
  (I : finType) (i : I) (F : I → R),
  (∀ a, (Oeq (op a idx) a)) →
       (Oeq (\big[op/idx]_(j | j == i) F j) (F i)).
Lemma big_pred1s : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
  (I : finType) (i : I) (P : pred I) (F : I → R),
  (∀ a, (Oeq (op a idx) a)) →
```

```
  P =1 pred1 i → (Oeq (\big[op/idx]_(j | P j) F j) (F i)).
Lemma bigD1s : ∀ (R : Type) (ordR: Ccpo.ord R)
 (idx : R) (op : R → R → R)
 (I : finType) (j : I) (P : pred I) (F : I → R),
  (∀ a b, (Oeq (op a b) (op b a))) →
  (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
  (∀ a, (Oeq (op idx a) a)) →
  (∀ a, (Oeq (op a idx) a)) →
  (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op a c) (op b d))) →
 P j →
 Oeq (\big[op/idx]_(i | P i) F i)
     (op (F j) (\big[op/idx]_(i | P i && (i != j)) F i)).
Lemma big_catS : ∀ (R : Type) (ordR: Ccpo.ord R)
 (idx : R) (op : R → R → R)
           (I : Type) (r1 r2 : seq I) (P : pred I) (F : I → R),
  (∀ a b c : R, op a (op b c) == op (op a b) c) →
  (∀ a : R, op idx a == a) →
  (∀ a : R, op a idx == a) →
  (∀ a b c d : R, a == b → c == d → op a c == op b d) →
       \big[op/idx]_(i ← (r1 ++ r2) | P i) F i ==
       op (\big[op/idx]_(i ← r1 | P i) F i)
         (\big[op/idx]_(i ← r2 | P i) F i).
Lemma big_cat_nats : ∀ (R : Type) (ordR : Ccpo.ord R)
 (idx : R) (op : R → R → R)
 (n m p : nat) (P : pred nat) (F : nat → R),
  (∀ a b c : R, op a (op b c) == op (op a b) c) →
  (∀ a : R, op idx a == a) →
  (∀ a : R, op a idx == a) →
  (∀ a b c d : R, a == b → c == d → op a c == op b d) →
       (m ≤ n)%nat →
       (n ≤ p)%nat →
       \big[op/idx]_(m ≤ i < p | P i) F i ==
       op (\big[op/idx]_(m ≤ i < n | P i) F i)
         (\big[op/idx]_(n ≤ i < p | P i) F i).
Lemma big_nat1s : ∀ (R : Type) (ordR : Ccpo.ord R)
 (idx : R) (op : R → R → R)
           (n : nat) (F : nat → R),
  (∀ a : R, op a idx == a) →
 \big[op/idx]_(n ≤ i < n.+1) F i == F n.
Lemma big_nat_recrs : ∀ (R : Type) (ordR: Ccpo.ord R)
 (idx : R) (op : R → R → R)
```

```
          (n m : nat) (F : nat → R),
   (∀ a b c : R, op a (op b c) == op (op a b) c) →
   (∀ a : R, op idx a == a) →
   (∀ a : R, op a idx == a) →
   (∀ a b c d : R, a == b → c == d → op a c == op b d) →
   (m < n.+1)%nat →
        \big[op/idx]_(m ≤ i < n.+1) F i ==
        op (\big[op/idx]_(m ≤ i < n) F i) (F n).
Lemma big1_nats : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
  (P : nat → bool)
      (F : nat → R) (m n : nat),
   (∀ a b c : R, op a (op b c) == op (op a b) c) →
   (∀ a : R, op idx a == a) →
   (∀ a : R, op a idx == a) →
  (∀ a b c d : R, a == b → c == d → op a c == op b d) →
    (∀ i, P i && (m ≤ i < n) → Oeq (F i) idx) →
    Oeq (\big[op/idx]_(m ≤ i < n | P i) F i) idx.
Lemma big1_eqs : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
          (I : Type) (r : seq I) (P : pred I),
   (∀ a : R, op a idx == a) →
  (∀ a b c d : R, a == b → c == d → op a c == op b d) →
        \big[op/idx]_(<- r | [eta P]) (fun _ : I ⇒ idx) == idx.
Lemma big1_seqs : ∀ (R: Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
          (I : eqType) (r : seq_predType I) (P : pred I)
          (F : I → R),
(∀ a : R, op a idx == a) →
(∀ a b c d : R, a == b → c == d → op a c == op b d) →
        (∀ i : I, P i && (i \in r) → F i == idx) →
        \big[op/idx]_(i ← r | P i) F i == idx.
Print eq_bigl.
Lemma eq_bigls : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
          (I : Type) (r : seq I) (P1 P2 : pred I) (F : I → R),
        P1 =1 P2 →
        \big[op/idx]_(i ← r | P1 i) F i == \big[op/idx]_(i ← r | P2 i) F i.
Lemma eq_bigs : ∀ (R : Type) (ordR: Ccpo.ord R)
  (idx : R) (op : R → R → R)
   (I : Type) (r : seq I) (P1 P2 : pred I) (F1 F2 : I → R),
```

```
  (∀ a b c d : R, a == b → c == d → op a c == op b d )->
   P1 =1 P2 →
   (∀ i : I, P1 i → F1 i == F2 i) →
   \big[op/idx]_(i ← r | P1 i) F1 i == \big[op/idx]_(i ← r | P2 i) F2 i.

Lemma partition_bigs : ∀ (R : Type) (ordR: Ccpo.ord R)
 (idx : R) (op : R → R → R)
          (I J : finType) (P : pred I) (p : I → J)
          (Q : pred J) (F : I → R),
     (∀ a b, (Oeq (op a b) (op b a))) →
   (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
   (∀ a, (Oeq (op idx a) a)) →
   (∀ a, (Oeq (op a idx) a)) →
   (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op a c) (op b d))) →
       (∀ i : I, P i → Q (p i)) →
       \big[op/idx]_(i | P i) F i ==
       \big[op/idx]_(j | Q j) \big[op/idx]_(i | P i && (p i == j)) F i.

Lemma reindex_ontos : ∀ (R : Type) (ordR:Ccpo.ord R)
 (idx : R) (op : R → R → R)
          (I J : finType) (h : J → I) (h' : I → J)
          (P : pred I) (F : I → R),
   (∀ a b, (Oeq (op a b) (op b a))) →
   (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
   (∀ a, (Oeq (op idx a) a)) →
   (∀ a, (Oeq (op a idx) a)) →
   (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op a c) (op b d))) →
       (∀ i : I, P i → h (h' i) = i) →
       \big[op/idx]_(i | P i) F i ==
       \big[op/idx]_(j | P (h j) && (h' (h j) == j)) F (h j).

Print pair_big_dep.

Lemma pair_big_deps : ∀ (R : Type) (ordR: Ccpo.ord R)
 (idx : R) (op : R → R → R)
          (I J : finType) (P : pred I) (Q : I → pred J)
          (F : I → J → R),
     (∀ a b, (Oeq (op a b) (op b a))) →
   (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
   (∀ a, (Oeq (op idx a) a)) →
   (∀ a, (Oeq (op a idx) a)) →
   (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op a c) (op b d))) →
       \big[op/idx]_(i | P i) \big[op/idx]_(j | Q i j) F i j ==
       \big[op/idx]_(p | P p.1 && Q p.1 p.2) F p.1 p.2.

Lemma exchange_big_deps : ∀ (R : Type) (ordR: Ccpo.ord R)
```

```
 (idx : R) (op : R → R → R)
        (I J : Type) (rI : seq I) (rJ : seq J) (P : pred I)
        (Q : I → pred J) (xQ : pred J) (F : I → J → R),
  (∀ a b, (Oeq (op a b) (op b a))) →
  (∀ a b c, (Oeq (op a (op b c)) (op (op a b) c))) →
  (∀ a, (Oeq (op idx a) a)) →
  (∀ a, (Oeq (op a idx) a)) →
  (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op a c) (op b d))) →
      (∀ (i : I) (j : J), P i → Q i j → xQ j) →
      \big[op/idx]_(i ← rI | P i) \big[op/idx]_(j ← rJ | Q i j) F i j ==
      \big[op/idx]_(j ← rJ | xQ j)
         \big[op/idx]_(i ← rI | P i && Q i j) F i j.

Print big_morph.

Lemma big_morphs : ∀ (R1 R2 : Type) (ordR1:Ccpo.ord R1) (ordR2: Ccpo.ord R2)
      (f : R2 → R1) (id1 : R1)
        (op1 : R1 → R1 → R1) (id2 : R2) (op2 : R2 → R2 → R2),
   (∀ a b c d, (Oeq a b) → (Oeq c d) → (Oeq (op1 a c) (op1 b d))) →
      (∀ x y : R2, f (op2 x y) == op1 (f x) (f y)) →
      f id2 == id1 →
      ∀ (I : Type) (r : seq I) (P : pred I) (F : I → R2),
      f (\big[op2/id2]_(i ← r | P i) F i) ==
      \big[op1/id1]_(i ← r | P i) f (F i).

Lemma big_endos : ∀ (R : Type) (ordR:Ccpo.ord R)
 (f : R → R) (idx : R) (op : R → R → R),
    (∀ a b c d : R, a == b → c == d → op a c == op b d) →
      (∀ x y : R, f (op x y) == op (f x) (f y)) →
      f idx == idx →
      ∀ (I : Type) (r : seq I) (P : pred I) (F : I → R),
      f (\big[op/idx]_(i ← r | P i) F i) ==
      \big[op/idx]_(i ← r | P i) f (F i).

Lemma big_distrrs : ∀ (R : Type) (ordR: Ccpo.ord R)
 (zero : R) (times : R → R → R) (plus : R → R → R) (I : Type)
        (r : seq I) (a : R) (P : pred I) (F : I → R),
 (∀ x y : R, times a (plus x y) == plus (times a x) (times a y)) →
 (∀ a0 b c d : R, a0 == b → c == d → plus a0 c == plus b d) →
 times a zero == zero →
      times a (\big[plus/zero]_(i ← r | P i) F i) ==
      \big[plus/zero]_(i ← r | P i) times a (F i).

Lemma proba_not_null2_1 : ∀ (A:finType) (m:distr A) (f : MF A)
 (P: A → A → U),
  (0 < mu m f)%U →
```

```
  (∀ a b, (0 < P a b)%U → f a == f b) →
  ~(∀ (t:A), (Oeq (f t) 0%U)).

Lemma proba_not_null_eq1 : ∀ (T:finType) (m:distr T) (f : MF T),
 (mu m f) ==
 (mu m (fun x ⇒ \big[Uplus/0]_y (B2U(x == y) × f y)%U)).

Lemma mu_bigop1 : ∀ (T:finType) m (F:T → T → U),
(mu m) (fun x : T ⇒ \big[Uplus/0]_y (F x y)) ≤
 \big[Uplus/0]_y ((mu m) (fun x ⇒ (F x y))).

Lemma proba_not_null2_2 : ∀ (T:finType) (m:distr T) (f : MF T),
  0 < mu m f →
  ~(∀ t, (f t == 0) ∨ (Oeq (mu m (fun x ⇒ B2U (x==t))) 0)).

Hypothesis dec_zero : ∀ x : U, {x == 0}+{¬ x == 0}.

Lemma proba_not_null2 : ∀ (T:finType) (t0:T) (m:distr T) (f : MF T),
  0 < mu m f →
  ∃ t, (0 < f t)%U ∧ (0 < (mu m (fun x ⇒ B2U (x==t))))%U.
```

Chapter 4

Library my_alea

```
Require Import ssreflect ssrfun ssrbool ssrnat bigop.
Import Prenex Implicits.
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Require Import my_ssralea.
Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Set Implicit Arguments.
Open Local Scope U_scope.
```

4.1 Extra Lemmas for Alea

```
Section fixP.
Variables A B : Type.
Variable F : (A → distr B) -m> (A → distr B).
Variable q : A → B → U.
Variable PR : A → Prop.
Lemma Pfixrule_Ulub : ∀ (p : A → nat → U),
    (∀ x:A, p x O == 0)->
    (∀ (i:nat) (f:A → distr B),
        (∀ x: A, PR x → ok (p x i) (f x) (q x)) →
         ∀ x: A, PR x → ok (p x (S i)) (F f x) (q x)) →
         ∀ x: A, PR x → ok (Ulub (p x)) (Mfix F x) (q x).
Lemma Pfixrule : ∀ (p : A → nat -m> U),
    (∀ x:A, p x O == 0)->
    (∀ (i:nat) (f:A → distr B),
```

```
  (∀ x : A, PR x → ok ((p x) i) (f x) (q x)) →
   ∀ x : A, PR x →ok ((p x) (S i)) (F f x) (q x)) →
   ∀ x : A, PR x → ok (lub (p x)) (Mfix F x) (q x).
End fixP.

Lemma quarterUplus: Uplus [1/4] [1/4] == [1/2].

Lemma quarterUplusn n:
 Uplus ([1/4] × n) ([1/4] × n) == ([1/2] × n)%U.

Definition pmin2 n := match n with
 | O ⇒ 0
 | 1 ⇒ 0
 | S (S n) ⇒ pmin 1 n
end.

Instance pmin2_mon : monotonic pmin2.
Qed.

Definition Pmin2 :nat -m> U := mon pmin2.

Lemma lubp2: lub Pmin2 == 1%U.

Definition pmin1 n := match n with
 | O ⇒ 0
 | (S n) ⇒ pmin 1 n
end.

Instance pmin1_mon : monotonic pmin1.
Qed.

Definition Pmin1 :nat -m> U := mon pmin1.

Lemma lubp1: lub Pmin1 == 1%U.

Definition pcte1 (p:U) n := match n with
 | O ⇒ 0
 | (S n) ⇒ p
end.

Instance pcte1_mon : ∀ p, monotonic (pcte1 p).
Defined.

Definition Pcte1 (p:U) :nat -m> U := mon (pcte1 p).

Lemma lubpcte1 : ∀ p, lub (Pcte1 p) == p%U.

Definition pqmin (p:U) (q:U) (n:nat) := p - ( q ^ n ).

Instance pqmin_mon : ∀ p q, monotonic (pqmin p q).
Qed.

Definition Pqmin (p q:U) :nat -m> U := mon (pqmin p q).

Definition Uq1min := Pqmin 1.
```

Lemma eq_lim_Uq1min : ∀ q, q < 1 → lub (Uq1min q) == 1.

Lemma Uq1min_S : ∀ n p,
(Uq1min ([1-]p)) (S n) == p + (Uq1min ([1-]p)) n × ([1-]p).

Lemma Uq1min_0 : ∀ q, (Uq1min q) O == 0.

Lemma compn_morph : ∀ (f : U → U→ U) (x: U)
($u1$: **nat** → U) ($u2$: **nat** → U) (n: **nat**),
(∀ x y $x0$ $y0$: U, x == y → $x0$ == $y0$ → f x $x0$ == f y $y0$) →
(∀ y, $u1$ y == $u2$ y) → compn f x $u1$ n == compn f x $u2$ n.

Lemma sigma_compo : ∀ (f : **nat** → **nat** → U) (a b:**nat**),
(∀ x y,f x y == f y x) →
(sigma (fun k : **nat** ⇒ (sigma (fun l : **nat** ⇒ f k l) b)) a) ==
(sigma (fun k : **nat** ⇒ (sigma (fun l : **nat** ⇒ f k l) a)) b).

Lemma mu_cond_le : ∀ (A : Type) (m : **distr** A) (f g : MF A),
(mu m) (fconj f g) ≤ (mu m) f.

4.2 Extra Lemmas for R Alea

Section Rplus.

Require Import Rplus.
Open Scope *Rp_scope*.

Lemma Rp_double1 : ∀ x,
(2 × x) == (x + x).

Lemma N2Rp_S_plus_1 : ∀ n, N2Rp (S n) == R1 + n.

Lemma divn1 : ∀ n,
(U2Rp ([1/]1+n)) + R1 == n.+2 × (U2Rp ([1/]1+n)).

Lemma Rpsigma_const : ∀ (n : **nat**) (x : **Rp**),
(Rpsigma (fun _ : **nat** ⇒ x)) n == (n × x)%Rp.

Lemma Unth_mult_eq : ∀ x,
(U2Rp([1/]1+x) × x.+1)%Rp == R1.

Close Scope *Rp_scope*.
End Rplus.

Open Local Scope *U_scope*.
Open Local Scope *O_scope*.

Lemma sigma_mult_perm :
∀ (f : **nat** → U) n $c1$ $c2$, retract (fun k ⇒ $c1$ × (f k)) n → retract (fun k ⇒ $c2$ × (f k)) n
→ $c1$ × (sigma (fun k ⇒ $c2$ × (f k)) n) == $c2$ × (sigma (fun k ⇒ $c1$ × (f k)) n).

```
Lemma Rpsigma_U2Rp : ∀ (f : nat → U) n, retract f n
    → Rpsigma f n == sigma f n.
Hint Resolve Rpsigma_U2Rp.
```

```
Lemma sigma_dist1 : ∀ n (f:nat → U),
  [1/]1+n.+1 × (sigma (fun i ⇒ [1/]1+n × f i)) n.+1 +
  (sigma (fun i ⇒ [1/]1+n.+1 × f i)) n.+1 ==
  (sigma (fun i ⇒ [1/]1+n × f i)) n.+1.
```

```
Lemma prod_comp1 : ∀ (n m:nat) (f:nat→U),
  prod [eta f] n × prod (fun x : nat ⇒ f (x + n)%nat) m ==
  prod [eta f] (n+m)%nat.
```

```
Lemma prod_comp2 : ∀ (n:nat) (f g:U),
  prod (fun _ ⇒ f) n × prod (fun _ ⇒ g) n ==
  prod (fun _ ⇒ f × g) n.
```

```
Lemma ex_le1 : ∀ a0 a1, a0 ≤ a1 →
 ∃ x, (@Oeq U ordU (a0 + x) a1) ∧ a0 ≤ [1-] x.
```

```
Lemma id_rem0: ∀ (a b: U),
 [1/2]× (a × b) ≤ [1/2] × a *( [1/2] × a) + [1/2] × b × ([1/2] × b).
```

```
Lemma id_rem1 : ∀ (a b: U),
 a × b ≤ ([1/2] × a + [1/2] × b) × ([1/2] × a + [1/2] × b).
```

```
Lemma id_rem2 : ∀ (a b: U) (n:nat),
 prod (fun _ ⇒ a × b) n ≤ prod (fun _ ⇒[1/2] × a + [1/2] × b ) (2 × n)%nat.
```

```
Lemma prod_sigma_id2 : ∀ (n:nat) (f:nat→U),
  prod
     (fun _ : nat ⇒
      (sigma (fun j : nat ⇒ [1/]1+n × f j)) n.+1 ×
      (sigma (fun j : nat ⇒ [1/]1+n × f (j + n.+1)%nat)) n.+1) n.+1 ≤
   prod
     (fun _ : nat ⇒
      (sigma (fun j : nat ⇒ [1/]1+(2 × n).+1 × f j)) (2 × n).+2)
     (2 × n).+2.
```

4.3 Not null probability

```
Lemma proba_not_null : ∀ (A:Type) (t:A) (m:distr A) (f : MF A)
 (P: A → A → U),
  (∀ a b, 0 < P a b → f a == f b) →
  0 < mu m (fun x ⇒ P x t) → 0 < f t →
  0 < mu m f.
```

4.4 Two independent events

```
Definition indep (A:Type) (m:distr A)(f g : MF A) :=
  mu m (fconj f g) == mu m f × mu m g.

Lemma indep_cond : ∀ (A:Type) (m:distr A)(f g : MF A),
  indep m f g → ¬ 0 == mu m f → mu (Mcond m f) g == (mu m) g.

Lemma carac_prod2 : ∀ (A: Type) (m: distr A) (a b: A → U),
 indep m a b →
 mu m (fconj a b) == mu m a × mu m b.

Definition fB2U A (a : A → bool) : A → U :=
  fun x ⇒ B2U (a x).

Definition indepb (A: Type) (m: distr A) (a b: A → bool) :=
  indep m (fB2U a) (fB2U b).

Lemma carac_prodb : ∀ (A: Type) (m: distr A) (a b: A → bool),
 indepb m a b →
 mu m (fB2U (fun (x:A) ⇒ andb (a x) (b x))) == mu m (fB2U a) × mu m (fB2U b).

Lemma indepb_Munit : ∀ (A:Type) x (f g : A → bool),
 indepb (Munit x) f g.

Lemma indepb_sym : ∀ (A:Type) (m:distr A) (f g: A → bool),
 indepb m f g ↔ indepb m g f.
```

4.5 Two composed events

```
Section composed.

Definition Total {A:Type}(DA:distr A) := Oeq (mu DA (fone A)) 1%U.

Variables A B C: Type.
Variable compose : A → B → C.

Variable DA : distr A.
Variable DB : distr B.

Hypothesis HA : Total DA.
Hypothesis HB : Total DB.

Let F := Mlet DA
        (fun k ⇒ Mlet DB (fun k' ⇒ Munit (compose k k'))).

Section on_f.

Variables (fA : A → U)(fB:B→U)(fC : C→U).

Hypothesis HAB: ∀ a b, Oeq (fC (compose a b)) (fA a × fB b)%U.

Let X := mu F fC.
```

Lemma L00: X == mu DA (fun $x \Rightarrow ((fA\ x) \times$ (mu $DB\ fB$)) $\%U$).

Lemma L01 : X == (mu $DA\ fA \times$ mu $DB\ fB$)$\%U$.

End on_f.

Lemma F_total : Total F.

End composed.

4.6 Discrete sigma distributions

Section Discrete_s.

Instance discrete_s_mon : $\forall$ A (c : **nat** $\to$ U) (p : **nat** $\to$ A) (n:**nat**),
    monotonic (fun f : $A \to U \Rightarrow$ sigma (fun $k \Rightarrow c\ k \times f\ (p\ k)$) n).
Save.

Definition discrete_s A (c : **nat** $\to$ U) (p : **nat** $\to$ A) (n:**nat**): M A :=
    mon (fun f : $A \to U \Rightarrow$ sigma (fun $k \Rightarrow c\ k \times f\ (p\ k)$) n).

Lemma discrete_s_simpl : $\forall$ A (c : **nat** $\to$ U) (p : **nat** $\to$ A) f (n:**nat**),
    discrete_s $c\ p\ n\ f$ = sigma (fun $k \Rightarrow c\ k \times f\ (p\ k)$) n.

Lemma discrete_s_stable_inv : $\forall$ A (c : **nat** $\to$ U) (p : **nat** $\to$ A) (n:**nat**),
    retract $c\ n$ $\to$ stable_inv (discrete_s $c\ p\ n$).

Lemma discrete_s_stable_plus : $\forall$ A (c : **nat** $\to$ U) (p : **nat** $\to$ A) (n:**nat**),
    stable_plus (discrete_s $c\ p\ n$).

Lemma retract_le : $\forall$ ($f\ g$: **nat**$\to U$) (n:**nat**), $f \leq g$ $\to$ retract $g\ n$ $\to$
 retract $f\ n$.

Lemma discrete_s_stable_mult : $\forall$ A (c : **nat** $\to$ U) (p : **nat** $\to$ A) (n:**nat**),
    retract $c\ n$ $\to$ stable_mult (discrete_s $c\ p\ n$).

Lemma discrete_s_continuous : $\forall$ A (c : **nat** $\to$ U) (p : **nat** $\to$ A) (n:**nat**),
    continuous (discrete_s $c\ p\ n$).

Record **discr_s** (A:Type) : Type :=
    {bound_s : **nat**; coeff_s : **nat** $\to$ U;
     coeff_retr_s : retract coeff_s bound_s; points_s : **nat** $\to$ A}.
Hint Resolve coeff_retr_s.

Definition Discrete_s : $\forall$ A, **discr_s** A $\to$ **distr** A.
Defined.

Lemma Discrete_s_simpl : $\forall$ A (d:**discr_s** A),
    mu (Discrete_s d) = discrete_s (coeff_s d) (points_s d) (bound_s d).

Definition is_discrete_s (A:Type) (m: **distr** A) :=
    $\exists$ d : **discr_s** A, m == Discrete_s d.

Lemma discrete_s_commute : $\forall$ A B ($d1$:**distr** A) ($d2$:**distr** B) (f:MF ($A \times B$)),

```
    is_discrete_s d1 → prod_distr_com d1 d2 f.
Lemma is_discrete_s_swap: ∀ A B C (d1:distr A) (d2:distr B)
 (f:A → B → distr C),
   is_discrete_s d1 →
   Mlet d1 (fun x ⇒ Mlet d2 (fun y ⇒ f x y)) ==
   Mlet d2 (fun y ⇒ Mlet d1 (fun x ⇒ f x y)).
Lemma retract_invn : ∀ n, retract (fun _ ⇒ ([1/]1+n)%U) (S n).
Lemma is_discrete_Random : ∀ (n:nat), is_discrete_s (Random n).
End Discrete_s.
```

4.7 Conditional probability

```
Section Conditionnal.
Require Import ssreflect ssrfun ssrbool eqtype ssrnat.
Require Import fintype finset fingraph seq.
Import Prenex Implicits.
Require Import my_ssr.
Require Import weird_induc.
Variables (A:Type) (B:finType) (b:B).
```

prodConj f a : Product of f applied to each element of B and a The result is in 0,1

```
Definition prodConj (f:B→MF A) (a:A) : U :=
  \big[(fun x : U ⇒ [eta Umult x])/1]_y f y a.
```

prodConjBound f j a : Product of f applied to each element of B of rank comprised between j.+1 and the cardinality of B, and a The result is in 0,1

```
Definition prodConjBound (f:B→MF A) (j:nat) (a:A) : U :=
 \big[(fun x : U ⇒ [eta Umult x])/1]_(j.+1 ≤ i < #|B|)
   f (nth b (enum B) i) a.
Lemma Mcond_prodConj : ∀ (f:B → MF A) (m: distr A),
 Term m →
 mu m (prodConj f) ==
 prod (fun i ⇒ mu (Mcond m (prodConjBound f i))
                    (f (nth b (enum B) i)) )
      #|B|.
Lemma Mcond_prodConjBound :
 ∀ (f:B→MF A) (m: distr A) (x:B) (k:nat) (P:B→nat → bool),
  Term m →

 ¬(mu m) (prodConjBound (fun y : B ⇒ Uprop.finv (f y)) k) == 0 →
```

```
 (∀ x0, f x x0 ×
      (\big[(fun x1 : U ⇒ [eta Umult x1])/1]_(k.+1 ≤ i < #|B| |
       P x i) Uprop.finv (f (nth b (enum B) i)) x0) == f x x0) →

 indep m (f x)
   (fun x0 : A ⇒
    \big[(fun x1 : U ⇒ [eta Umult x1])/1]_(k.+1 ≤ i < #|B|)
       (if ~~ P x i then Uprop.finv (f (nth b (enum B) i)) x0 else 1)) →

 mu m (f x) ≤
 mu (Mcond m
          (prodConjBound (fun y ⇒ Uprop.finv (f y))
             k))
                 (f x) .

Require Import Rplus.

Lemma prod_sigma_average : ∀ (n:nat) (f:nat→ U),
 prod (fun i ⇒ f i) n.+1 ≤
 prod (fun _ ⇒ sigma (fun i ⇒ [1/]1+n × f i) n.+1) n.+1.

Lemma sigma_inv_simpl : ∀ (n:nat) (f: nat → U),
    sigma (fun i ⇒ [1/]1+n × [1-] (f i)) (S n) == [1-] sigma (fun i ⇒ [1/]1+n × (f
i)) (S n).

Lemma prod_sigma_averagefin : ∀ (f:B→ U),
 prod (fun i ⇒ [1-] f (nth b (enum B) i))
      #|B| ≤
 prod (fun _⇒ [1-]
                (sigma (fun i ⇒ [1/]1+ #|B|.-1 × f (nth b (enum B) i)) #|B|))
      #|B|.

Lemma forall_exists_fB2U : ∀ (P: A → B → bool),
(fun x ⇒ NB2U [∃ y, P x y]) == fB2U (fun x ⇒[∀ y, ~~ P x y]).

Lemma finv_fB2U : ∀ (P: A → bool),
 (fB2U (fun y ⇒ ~~ P y)) ==
 (Uprop.finv (fB2U (fun y ⇒ P y))).

Lemma forall_prodConj_fB2U : ∀ (P: A → B → bool),
 fB2U (fun x ⇒ [∀ y, ~~ P x y]) ==
 prodConj (fun e ⇒ Uprop.finv (fB2U (fun s ⇒ P s e))).

End Conditionnal.
```

4.8 Alea/Bigop equivalence

```
Section Bigop.

Variables (A:finType) (a:A).

Definition prodOP (f:A→Rp) :=
  \big[Rpplus/R0]_y f y.

Lemma rpsigma_bigop : ∀ (f:A →Rp),
  prodOP f == Rpsigma (fun x ⇒ f (nth a (enum A) x)) #|A|.

Lemma iter_Rpplus_0 : ∀ (n:nat) (m :Rp),
 ssrnat.iter n (Rpplus m) O == Rpmult n m.

Lemma bigRpplusleq : ∀ (T:finType) (f g:T→Rp),
  (∀ v, f v ≤ g v) →
  \big[Rpplus/R0]_v (f v) ≤ \big[Rpplus/R0]_v (g v).

End Bigop.
```

Chapter 5

Library graph

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun bigop choice tuple.
Add LoadPath "../prelude".
Require Import my_ssr.
Set Implicit Arguments.
Import Prenex Implicits.
```

5.1 Introduction

This file develops the theory of finite graph represented by an edge relation over a finType V.

5.2 Definitions: Graph

V: set of vertices of the graph Adj: edge relation of the graph

```
Class Graph {V:finType} (Adj: rel V).
Section Graph.
Context `(G: Graph V Adj).
```

Nb_enum G v: the ordered sequence of the neighbours of v

```
Definition Nb_enum (G: Graph Adj)(v: V) : seq V :=
 enum (Adj v).
```

deg G v: the degree of v, i.e. the number of neighbours it has

```
Definition deg (G: Graph Adj) (v: V) : nat :=
 seq.size (Nb_enum G v).
```

nb_id G v w: the index of w in the sequence Nb_enum G v. w is said to be (nb_id v w)th neighbour of v

```
Definition nb_id (G: Graph Adj) (v w: V) : nat :=
 index w (Nb_enum G v).
```

edge_finType is the finType containing the set of edges. fste is the first member of the edge. snde is the second one.

```
Record edge : Type :=
 Edge {edgeVal : (Datatypes.prod V V);
                 EdgeValP : Adj edgeVal.1 edgeVal.2 &&
               ((enum_rank edgeVal.1) < (enum_rank edgeVal.2))%nat}.

Canonical edge_subType := Eval hnf in [subType for edgeVal by edge_rect].
Definition edge_eqMixin := Eval hnf in [eqMixin of edge by <:].
Canonical edge_eqType := Eval hnf in EqType edge edge_eqMixin.
Definition edge_choiceMixin := [choiceMixin of edge by <:].
Canonical edge_choiceType :=
  Eval hnf in ChoiceType edge edge_choiceMixin.
Definition edge_countMixin := [countMixin of edge by <:].
Canonical edge_countType :=
  Eval hnf in CountType edge edge_countMixin.
Canonical edge_subCountType := [subCountType of edge].
Definition edge_finMixin := [finMixin of edge by <:].
Canonical edge_finType := Eval hnf in FinType edge edge_finMixin.

Definition fste (x: edge) := (edgeVal x).1.
Definition snde (x: edge) := (edgeVal x).2.
```

port_finType is the finType containing the set of ports. fstp is the first member of the port. sndp is the second one.

```
Record port :=
 Port {pval : (Datatypes.prod V V); PvalP : Adj pval.1 pval.2}.

Canonical port_subType := Eval hnf in [subType for pval by port_rect].
Definition port_eqMixin := Eval hnf in [eqMixin of port by <:].
Canonical port_eqType := Eval hnf in EqType port port_eqMixin.
Definition port_choiceMixin := [choiceMixin of port by <:].
Canonical port_choiceType :=
  Eval hnf in ChoiceType port port_choiceMixin.
Definition port_countMixin := [countMixin of port by <:].
Canonical port_countType :=
  Eval hnf in CountType port port_countMixin.
Canonical port_subCountType := [subCountType of port].
Definition port_finMixin := [finMixin of port by <:].
Canonical port_finType := Eval hnf in FinType port port_finMixin.

Definition fstp (x: port) := (pval x).1.
Definition sndp (x: port) := (pval x).2.
```

outerport_set v: the set of ports whose first member is equal to v.

```
Definition outerport_set (v:V) :=
[set x:port| (fstp x) == v].
```

outerport_list v: the default sequence of ports corresponding to (outerport_set v).

```
Definition outerport_list (v:V) :=
enum (outerport_set v).
```

innerport_set v: the set of ports whose second member is equal to v.

```
Definition innerport_set (v:V) :=
[set x:port| (sndp x) == v].
```

innerport_list v: the default sequence of ports corresponding to (innerport_set v).

```
Definition innerport_list (v:V) :=
enum (innerport_set v).
```

port_id G v w: the index of the port (v,w) in the sequence (outerport_list v). (v,w) is said to be (port_id v w)th port linked to v

```
Definition port_id (G: Graph Adj) (v w: V) : nat :=
 find (fun x ⇒ sndp x == w) (outerport_list v).
```

VtoP v w p0: returns a port made with v and w if they are adjacent p0 otherwise

```
Definition VtoP (v w : V) (p0 : port_finType) : port_finType :=
 odflt p0 (insub (v, w)).
```

5.3 Lemmas: Graph

5.3.1 deg

```
Lemma deg_index_lt : ∀ (v w: V),
  Adj v w = (index w (Nb_enum G v) < deg G v).
```

```
Lemma deg_zero : ∀ (v:V),
 deg G v = 0 ↔
 ∀ (w:V), ¬Adj v w.
```

```
Lemma deg_card1 : ∀ (v:V), irreflexive Adj →
 deg G v ≤ #|V|.-1.
```

5.3.2 nb_id

```
Lemma nb_id_lt: ∀ (v w:V) (x:nat),
 ~~(x < deg G v)%nat → ~~((nb_id G v w == x) && Adj v w).
```

```
Lemma nb_id_index_lt: ∀ (v w:V),
 Adj v w = (nb_id G v w < deg G v)%nat.
```

Lemma deg_exists : ∀ (v:V) (x:**nat**),
x < deg G v → ∃ w, Adj v w ∧ nb_id G v w = x.

Lemma nb_id_e : ∀ u v v', Adj u v →
nb_id G u v == nb_id G u v' → v == v'.

5.3.3 edge

Lemma edge_fs : ∀ (e:**edge**),
fste e != snde e.

Lemma edge_fste_snde : ∀ (e:**edge**),
Adj (fste e) (snde e).

Lemma edge_eq : ∀ $e1$ $e2$,
edgeVal $e1$ = edgeVal $e2$ →
Adj (edgeVal $e1$).1 (edgeVal $e1$).2 &&
(enum_rank (edgeVal $e1$).1 < enum_rank (edgeVal $e1$).2)%nat =
Adj (edgeVal $e1$).1 (edgeVal $e1$).2 &&
(enum_rank (edgeVal $e1$).1 < enum_rank (edgeVal $e1$).2)%nat →
$e1$ = $e2$.

Lemma edge_nth_neq1 : ∀ k i e,
(k < i < #|edge_finType|)%nat →
(fste (nth e (enum edge_finType) k) ==
fste (nth e (enum edge_finType) i))%B →
snde (nth e (enum edge_finType) k) !=
snde (nth e (enum edge_finType) i).

Lemma edge_nth_neq2 : ∀ k i e,
(k < i < #|edge_finType|)%nat →
(fste (nth e (enum edge_finType) k) ==
snde (nth e (enum edge_finType) i))%B →
snde (nth e (enum edge_finType) k) !=
fste (nth e (enum edge_finType) i).

Lemma edge_nth_neq3 : ∀ k i e,
(k < i < #|edge_finType|)%nat →
(snde (nth e (enum edge_finType) k) ==
fste (nth e (enum edge_finType) i))%B →
fste (nth e (enum edge_finType) k) !=
snde (nth e (enum edge_finType) i).

Lemma edge_nth_neq4 : ∀ k i e,
(k < i < #|edge_finType|)%nat →
(snde (nth e (enum edge_finType) k) ==
snde (nth e (enum edge_finType) i))%B →

```
 fste (nth e (enum edge_finType) k) !=
   fste (nth e (enum edge_finType) i).
Lemma edge_in_V1 : ∀ e:edge_finType,
 (fste e) \in (enum V).
Lemma edge_in_V2 : ∀ e:edge_finType,
 (snde e) \in (enum V).
```

5.3.4 Port

```
Definition EtoP1 (e : edge_finType) : port_finType :=
Port (edge_fste_snde e).
Lemma port_eq : ∀ p1 p2,
 pval p1 = pval p2 →
 Adj (pval p1).1 (pval p1).2 =
 Adj (pval p1).1 (pval p1).2 →
 p1 = p2.
Lemma VtoP1 p p0 : VtoP (fstp p) (sndp p) p0 = p.
Lemma VtoP2 : ∀ v w p0, Adj v w →
 fstp (VtoP v w p0) = v.
Lemma VtoP3 : ∀ v w p0, Adj v w →
 sndp (VtoP v w p0) = w.
Lemma disjoint_outerport : ∀ v w, v != w →
  [disjoint outerport_set v & outerport_set w].
End Graph.
```

5.4 Definitions: Undirected Graph and without loop

```
Class NGraph `(Gr: Graph) := {
 gsym: symmetric Adj;
 grefl: irreflexive Adj}.
Section NGraph.
Context `(Gr: NGraph V Adj).
```

Edges on V linked thanks to Adj

```
Definition E := (@edge_finType V Adj).
```

VtoE v w e0: returns an edge made with v and w if they are adjacent e0 otherwise

```
Definition VtoE (v w : V) (e0 : E) : E :=
 odflt e0 (insub (v, w)).
```

5.5 Lemmas: Undirected Graph and without loop

5.5.1 VtoE

Lemma VtoE1 $e\ e0$: VtoE (fste e) (snde e) $e0$ = e.

Lemma VtoE2 : $\forall\ v\ w\ e0$, $Adj\ v\ w \rightarrow$
(enum_rank v < enum_rank w)%nat $\rightarrow$
(fste (VtoE $v\ w\ e0$) == v)%B.

Lemma VtoE3 : $\forall\ v\ w\ e0$, $Adj\ v\ w \rightarrow$
(enum_rank v < enum_rank w)%nat $\rightarrow$
(snde (VtoE $v\ w\ e0$) == w)%B.

5.5.2 deg

Lemma deg_card : $\forall$ (v:V),
(deg $Gr0\ v \leq$ #| V|.-1)%coq_nat.

5.5.3 Adj

Lemma adj_diff : $\forall\ u\ v$,
$Adj\ u\ v \rightarrow u$!= v.

5.5.4 Port numbering

Variable nu : $V \rightarrow$ seq V.
Hypothesis Hnu: $\forall$ ($v\ w$:V), ($Adj\ v\ w$) = (w \in ($nu\ v$)).
Hypothesis $Hnu2$: $\forall$ (v :V), uniq ($nu\ v$).

Lemma degnu1 : $\forall\ v$, size ($nu\ v$) = deg $Gr0\ v$.

Lemma degnu3 : $\forall\ v\ w\ i$,
i < deg $Gr0\ v \rightarrow$ nth v ($nu\ v$) i = $w \rightarrow$
w \in ($nu\ v$).

Lemma degnu2 : $\forall\ v\ w\ i$,
i < deg $Gr0\ v \rightarrow$ nth v ($nu\ v$) i = $w \rightarrow$
$\exists\ j$, j < deg $Gr0\ w \wedge$ nth w ($nu\ w$) j = v.

End NGraph.

Chapter 6

Library bfs

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun bigop choice tuple.
Add LoadPath "../prelude".
Require Import my_ssr.
Require Import graph.
Set Implicit Arguments.
Import Prenex Implicits.
```

6.1 Introduction

This file implements a breadth first search (BFS) on a graph described with a set of vertices V and a edge relation Adj

```
Section BFS.
Variables (V:finType) (Adj:rel V).
Hypothesis gsym: ∀ u v, Adj u v = Adj v u.
Hypothesis grefl: ∀ u, Adj u u = false.
```

connected: there is a path between two vertices of the graph

```
Definition connected := ∀ (u v:V), connect Adj u v.
```

parentF f: f is a parent function of the graph

```
Definition parentF (f: {ffun V → (option V)}) :=
 ∀ u v, f u = Some v → Adj u v.
```

Nnone v f: the set of neighbours of v which have no parent

```
Definition Nnone (v: V) (f: {ffun V → (option V)}) :=
 [set x| (Adj v x) && (f x == None)].
```

bfs n l f: the parent function made from an update of f with bfs where n is the number of visited nodes and l is the sequence of already visited nodes

```
Fixpoint bfs (n: nat) (l: seq V) (f: {ffun V → (option V)}) {struct n}
 : {ffun V → (option V)} :=
  if n is n'.+1 then
    if l is (t::q) then
       bfs n' (cat q (enum (Nnone t f)))
           (finfun (fun x ⇒ if (x \in (Nnone t f)) then (Some t) else (f x)))
     else f
  else f.
```

bfsL n lv lr: the sequence of bfs of size n lv are the already visited nodes lr the marked nodes which still has to be visited

```
Fixpoint bfsL (n: nat) (lv lr: seq V)
  {struct n} : seq V :=
  if n is n'.+1 then
    if lr is (t::q) then
       (t:: (bfsL n' (t::lv)
                 (cat q (enum [set x| (Adj t x) && (x \notin lv) &&
                            (x \notin lr)]))))
     else lr
  else lr.
```

tF v n: the parent function made from bfs where the root v has no parent `Definition tF (v:V) (n:nat):=`

```
 finfun (fun x ⇒ if x == v then None
          else (bfs n [::v]
             (finfun (fun x ⇒ if x == v then Some v else None))) x).
```

6.2 Lemmas: BFS

6.2.1 bfs

```
Lemma bfs_simpl : ∀ n l f,
 bfs n.+1 l f = if l is (t::q) then
       bfs n (cat q (enum (Nnone t f)))
           (finfun (fun x ⇒ if (x \in (Nnone t f)) then (Some t) else (f x)))
     else f.
```

```
Lemma bfs1 n: ∀ l (f:{ffun V → (option V)}) x y,
 f x = Some y → (bfs n l f) x = Some y.
```

```
Lemma bfs2 n:
 ∀ u w (l : seq V) (f : {ffun V → option V}),
   f u = None → (bfs n l f) u = Some w → Adj u w.
```

```
Lemma bfs3 n:
```

```
 ∀ u (l : seq V) (f : {ffun V → option V}),
   f u = None → (bfs n l f) u ≠ Some u.
```

```
Lemma bfs4 n : ∀ l x y f,
 (bfs n l f) x == Some y →
 (bfs n.+1 l f) x == Some y.
```

```
Lemma bfs5 n : ∀ l x y (f:{ffun V → (option V)}),
 (∀ x, x \in l → f x ≠ None) →
 (bfs n l f) x = None →
 (bfs n.+1 l f) x = Some y →
 (bfs n l f) y ≠ None.
```

```
Lemma bfs6 n : ∀ l x (f:{ffun V → (option V)}),
 (∀ x, x \in l → f x ≠ None) →
 (bfs n l f) x = None →
 (bfs n.+1 l f) x ≠ None →
 ∃ y, (bfs n l f) y ≠ None ∧
              (bfs n.+1 l f) x = Some y.
```

```
Lemma bfs7 n: ∀ l x (f:{ffun V → (option V)}),
 (∃ v, v \in l) → (∀ x, x \notin l → f x = None) →
 (∀ x, x \in l → f x ≠ None) →
  x \notin l →
  (bfs n l f) x ≠ None →
  ∃ p, ∃ v, v \in l ∧
               path.path (fun x y ⇒ (bfs n l f) y == Some x) v (p++[::x]) ∧
               seq.size p < n ∧
               uniq (v::x::p).
```

```
Lemma bfs8 p: ∀ p' n l x v (f:{ffun V → (option V)}),
  uniq (v::x::p)-> uniq(v::x::p') →
  path.path (fun x y ⇒ (bfs n l f) y == Some x) v (p++[::x]) →
  path.path (fun x y ⇒ (bfs n l f) y == Some x) v (p'++[::x]) →
  p = p'.
```

6.2.2 bfsL

```
Lemma bfsL1 n: ∀ x lv, 0<n →
 x \in bfsL n lv [:: x].
```

```
Lemma bfsL2 n: ∀ x lv lr, x \in lr →
 x \in bfsL n lv lr.
```

```
Lemma bfs_path :∀ n y (lv lr:seq V),
 #|V| ≤ #|lv| + n → y \notin (lv ++ lr) →
[disjoint lv & lr] → uniq lr →
```

reflect ($\exists$ x, (x \in lr)/\ (dfs_path (rgraph Adj) (lv)
x y)) (y \in bfsL n lv lr).

Lemma bfsP x y :
reflect (exists2 p, path (Adj) x p & y = last x p)
(y \in bfsL #| V| [::] [::x]).

Lemma bfsL3 n: $\forall$ l lv (f:{ffun V $\rightarrow$ (option V)}) x,
($\forall$ x, (x \in l $\vee$ x \in lv) $\leftrightarrow$ f x $\neq$ None) $\rightarrow$
x \in (bfsL n lv l) $\rightarrow$
(bfs n l f) x $\neq$ None.

6.2.3 tF

Lemma tF1 : $\forall$ v n, (tF v n) v = None.

Lemma tF2 : $\forall$ v x, connected $\rightarrow$ (tF v #| V|) x = None $\rightarrow$ x = v.

Lemma tF2'' : $\forall$ v x,
connect (fun x : V $\Rightarrow$ [eta Adj x]) v x $\rightarrow$ (tF v #| V|) x = None $\rightarrow$ x = v.

Lemma tF3 : $\forall$ v n, parentF (tF v n).

Lemma tF4 : $\forall$ v n x y, (tF v n) x = Some y $\rightarrow$ (tF v n) y $\neq$ Some x.

End BFS.

Section BFS2.
Variables (V:finType) (Adj:rel V).
Hypothesis $gsym$: $\forall$ u v, Adj u v = Adj v u.
Hypothesis $grefl$: $\forall$ u, Adj u u = false.

Lemma tF2' : $\forall$ (P:$V$$\rightarrow$bool),
($\forall$ u v, P u $\rightarrow$ P v $\rightarrow$ connect (fun x y $\Rightarrow$ P x && P y && Adj x y) u v) $\rightarrow$
$\forall$ v x, P v $\rightarrow$ P x $\rightarrow$
(tF (fun x y $\Rightarrow$ P x && P y && Adj x y) v #| V|) x = None $\rightarrow$ x = v.

End BFS2.

Chapter 7

Library graph_alea

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun bigop choice tuple.
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Add LoadPath "../prelude".
Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Require Import Rplus.
Require Import my_alea.
Require Import my_ssr.
Require Import my_ssralea.
Require Import graph.
Set Implicit Arguments.
Import Prenex Implicits.
```

7.1 Introduction

This file develops a relation between sum over edges and over vertices in the Positive Real of Alea

```
Section half.
Context `(NG: NGraph V Adj).
Definition E := (@edge_finType V Adj).
Variable (e0:E).
Lemma bigop_edge_half1 : ∀ (f:V→Rp),
  \big[Rpplus/R0]_(e:E) (fun a ⇒ ((f (fste a)) × (f (snde a)))%Rp) e ==
  \big[Rpplus/R0]_v (fun a : V ⇒
```

```
        (\big[Rpplus/R0]_(y | Adj a y &&
                                (enum_rank a < enum_rank y)%nat )
                                  (fun x => (f a * f x)%Rp) y ))
         v.

Lemma bigop_edge_half2 : forall (f:V->Rp),
  (2 * \big[Rpplus/R0]_(e:E) (fun a => (f (fste a)) * (f (snde a))) e)%Rp ==
  \big[Rpplus/R0]_v (fun a : V => ((f a) *
        (\big[Rpplus/R0]_(y | Adj a y) (fun x => f x) y))%Rp)
         v.

Lemma bigop_edge_R1 : forall (P Q: V-> bool)(f g:V->Rp) ( d:Rp),
( Rpmult (count P (enum V)) d == R1) ->
(forall v w, P v -> Adj v w -> Q w -> ( d <= (g w))) ->
(forall v, P v -> Rpmult (f v)
 (count (fun i : V => Adj v i && Q i) (enum V)) == R1) ->
 R1 <= \big[Rpplus/R0]_v (fun a : V => (if P a then (f a) else R0) *
        (\big[Rpplus/R0]_(y | Adj a y) (fun x => if Q x then g x else R0) y))%Rp
         v.

End half.
```

Chapter 8

Library labelling

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun tuple.
Add LoadPath "../prelude".
```

```
Set Implicit Arguments.
Import Prenex Implicits.
```

8.1 Introduction

This file is about labelling

```
Section Labelling.
```

8.2 Definitions

Locat: corresponds to the location of a label Label: the type of a label

```
Variables (Locat:finType) (Label: eqType).
```

LabelFunc: labelling function, maps a label to a location

```
Definition LabelFunc := {ffun Locat → Label}.
```

newLabel finit: Constructor of a LabelFunc returning (finit x) for a location x

```
Definition newLabel (finit: Locat → Label) : LabelFunc :=
 finfun (fun (x:Locat) ⇒ finit x).
```

update A old new: Update of a LabelFunc. If x is in A then it is updated by returning the new value from newF. Else it returns the old value from oldF

```
Definition update (A:{set Locat}) (old new: LabelFunc) : LabelFunc :=
 finfun (fun (x:Locat) ⇒ if x \in A then (new x) else (old x)).
```

8.3 Lemmas on update

Lemma update_Plocal_iff : ∀ (A:{set $Locat$}) (D Fup:LabelFunc) (w:$Locat$),
(update A D Fup) w = if (w \in A) then (Fup w) else (D w).

Lemma update_Pcomm : ∀ (A B:{set $Locat$}) (D $FupA$ $FupB$:LabelFunc),
[disjoint A & B] →
(update B (update A D $FupA$) $FupB$) =
(update A (update B D $FupB$) $FupA$).

End Labelling.

Chapter 9

Library ex1

```
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Set Implicit Arguments.
Open Local Scope U_scope.
Notation "[1/6]" := ([1/]1+5).
Lemma ex1 :
 (mu (Random 5)) (fun z : nat ⇒ if eq_nat_dec 2 z then 1%U
                                    else if eq_nat_dec 4 z then 1%U else 0%U)
  == 2 */ [1/6].
```

Chapter 10

Library dice

```
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Set Implicit Arguments.
Open Local Scope U_scope.
Notation "[1/6]" := ([1/]1+5).
Definition throw_dice :=
 Mlet (Random 5)
  (fun k => Mlet (Random 5)
    (fun k' => Munit (2 + k + k')%nat)).
Lemma throw_dice_simpl0 (f: MF nat):
  mu throw_dice f == sigma (fun i => [1/]1+5 ×
                       sigma (fun j => [1/]1+5 × f (2 + i + j)%nat) 6) 6.
Lemma throw_dice_simpl (f: MF nat) :
  mu throw_dice f == sigma (fun i =>
                       sigma (fun j => [1/]1+35 × f (2 + i + j)%nat) 6) 6.
Lemma throw_dice_11 :
 (mu throw_dice) (carac (eq_nat_dec 11)) == (2 */[1/]1+35)%U.
Lemma throw_dice_7 :
 mu throw_dice (carac (eq_nat_dec 7)) == 6 */ [1/]1+35.
Definition disjoint {A}(P Q:set A):=
   ∀ x, P x → Q x → False.
Lemma fplus_ok_carac{A} : ∀ P Pdec Q Qdec, disjoint P Q →
                      fplusok (@carac A P Pdec) (@carac A Q Qdec).
```

```
Lemma throw_dice_7_11 :
 mu throw_dice (fplus (carac (eq_nat_dec 7)) (carac (eq_nat_dec 11))) == 8 */ [1/]1+35.
```

Chapter 11

Library gen

11.1 Definition of randomised algorithm syntax

```
Inductive gen (B:Type): Type :=
  Greturn (b:B)
| Gbind (A :Type)(a:gen A)(f : A → gen B)
| Grandom (n:nat)(f : nat → gen B).
```

11.2 Definition of deterministic algorithms

```
Fixpoint Deterministic {B:Type}(e : gen B):Prop :=
match e with
  Greturn b ⇒ True
| Gbind A a f ⇒ Deterministic a ∧ ∀ b, Deterministic (f b)
| _ ⇒ False
end.
```

Chapter 12

Library op

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.
Require Import gen.
```

12.1 Definition of operational semantic

```
Definition Op (t:Type)(A:Type) := t → (A × t).
Definition Oreturn {t A}(a:A) : Op t A := fun g ⇒ (a, g).
Definition Obind {t A B} (m : Op t A) (f : A → Op t B) : Op t B :=
                    fun g ⇒ (f (m g).1) (m g).2.
Class ORandom (t:Type)(get : nat → Op t nat):={
  get_ok : ∀ n x, ( (get n x).1 ≤ n)%nat
 }.
Definition Orandom (n:nat){t:Type}{get : nat → t → nat × t}
                          (rand : ORandom t get) : Op t nat :=
 get n.
Section op.
Variable (rand_t : Type)(get : nat → rand_t → nat × rand_t).
Context (rand : ORandom rand_t get).
Fixpoint Opsem {B: Type}(m :gen B) : Op rand_t B :=
match m with Greturn b ⇒ Oreturn b
            | Gbind _ a f ⇒ Obind (Opsem a) (fun x ⇒ (Opsem (f x)))
            | Grandom n f ⇒
                     Obind (Orandom n rand)
                          (fun x ⇒ Opsem (f x))
  end.
End op.
```

Section generator.

12.2 Definition of pseudo random number generator

```
Let rand_t := nat.

Require Import div.

Let next0 m
                              a
                              c
                              (x : rand_t) : rand_t := modn (a × x + c)%nat m.

Let phi (n: nat) (x: rand_t) (max: rand_t) :=
  (n × x)%nat %/ max.

Let get0 m a c (n:nat) (x: rand_t) : nat × rand_t :=
 if (n < m.+1)%nat then
    (phi n (next0 m.+1 a c x) m, (next0 m.+1 a c x))
 else (O,O).

Instance lcg_generator (m a c : nat) : ORandom rand_t (get0 m a c).
Qed.
```

12.3 Tests

```
Example my_gen := lcg_generator (2^8.-1) 137 187.

End generator.
```

Chapter 13

Library setSem

```
Require Import Ensembles.
Require Import gen.
```

13.1 Definition of ensemblist semantic

```
Fixpoint Setsem {B: Type}(m :gen B) : Ensemble B :=
match m with
  |Greturn b ⇒ fun x ⇒ x = b
  | Gbind A a f ⇒ fun x ⇒ ∃ y, Setsem a y ∧ Setsem (f y) x
  | Grandom n f ⇒ fun x ⇒ ∃ i, (i ≤ n)%nat ∧ Setsem (f i) x
end.
```

13.2 About determinism

```
Lemma Deterministic_singleton {B:Type}(mb : gen B):
 Deterministic mb →
 ∀ b b', In _ (Setsem mb) b → In _ (Setsem mb) b' → b = b'.

Lemma Setsem1 {B:Type} (s y: B) (f: B → gen B) :
In _ (Setsem (Gbind _ _ (Greturn _ s) f)) y ↔ In _ (Setsem (f s)) y .

Section reachability.
```

13.3 Invariant

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.

Variable B:Type.

Definition Stable (P: B → Prop) (f: B → gen B) :=
```

```
 ∀ s, P s →
   ∀ s', In _ (Setsem (f s)) s' → P s'.
Definition Invariant (P: B → Prop) (f: B → gen B) (init : B) :=
 P init ∧ (Stable P f).
Definition reachFrom (f: B → gen B) (init s: B) :=
 ∃ n , In _ (Setsem (iter n (fun x ⇒ Gbind _ _ x f) (Greturn _ init))) s.
Lemma reachInd : ∀ (P:B → Prop) (f: B → gen B) (init: B) ,
 Invariant P f init →
 ∀ s, reachFrom f init s → P s.
End reachability.
```

Chapter 14

Library dist

```
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.
Require Import gen.
Require Export Cover.
Require Export Prog.
Require Export Ccpo.
Section dist.
```

14.1 Definition of distributional semantic

```
Fixpoint Distsem {B: Type}(m :gen B) : distr B :=
match m with Greturn b ⇒ Munit b
           | Gbind _ a f ⇒ Mlet (Distsem a) (fun x ⇒ (Distsem (f x)))
           | Grandom n f ⇒ Mlet (Random n) (fun x ⇒ Distsem (f x))
  end.
End dist.
```

Chapter 15

Library rdaTool_gen

Add *LoadPath* "../prelude".
Add *LoadPath* "../graph".
Add *LoadPath* "../ra".

Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun tuple.

Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import gen.

Set Implicit Arguments.
Import *Prenex Implicits*.

15.1 Introduction

Tools to reason about randomized distributed algorithms in a generic way.

Section general.

15.2 General Case

In this section, we define rounds, steps and monte carlo for algorithms which correspond to a rewriting of states over the vertices and states over locations. Locations could be ports or edges or anything else just expecting a finType.

We consider a graph with a set of vertices V and states over those vertices of type VLabel

Variable (V: finType) ($VLab$: eqType).

Locations and type of location labels

Variables (L:finType) ($LLab$:eqType).

Labelling function : VState for the vertices ; LState for the locations `Let` *VSt* := LabelFunc *V* *VLab*.
`Let` *LSt* := LabelFunc *L* *LLab*.

A vertex v can only change a part of the location which is (WriteArea v). A vertex v can only have access to a part of the location which is (ReadArea v). `Variable` *WriteArea* : *V* → {set *L*}.
`Variable` *ReadArea* : *V* → {set *L*}.

Transformation of a local computation of a vertex to a global one. The input of Lwrite is supposed to be the writeArea of the vertex. `Variable` *Vwrite* : *VLab* → *V* → *VSt*.
`Variable` *Lwrite* : (seq *LLab*) → *V* → *LSt*.

Transformation of a global computation of a vertex to a local one. Linread gives the labels of the readArea. Loutread gives the labels of the writeArea. `Variable` *Vread* : *VSt* → *V* → *VLab*.
`Variable` *Linread* : *LSt* → *V* → (seq *LLab*).
`Variable` *Loutread* : *LSt* → *V* → (seq *LLab*).

```
Hypothesis Vread1 : ∀ v w resV f,
 v != w →
(Vread resV v) =
(Vread (update [set w] resV f) v).
```

```
Hypothesis Lread1 : ∀ v w resL f,
 v != w →
  (Loutread resL v) =
  (Loutread (update (WriteArea w) resL f) v).
```

A local rule for a vertex v takes as parameters: the state of v ; the states of the writing zone ; the states of the reading zone. It gives a new state for v and new states for the writing zone. `Definition` GLocT := *VLab* → (seq *LLab*) → (seq *LLab*) →
 gen (*VLab* × seq *LLab*).

`Section` round.

15.2.1 Round

Round for a randomized distributed algorithm: a local function is applied to all vertices which updates the global state `Fixpoint` GRound (*seqV*: seq *V*) (*res*: *VSt* × *LSt*)

```
 (LocalRule : GLocT)
 : gen (VSt×LSt) :=
 match seqV with
  |nil ⇒ Greturn _ res
  |h::t ⇒ Gbind _ _ (GRound t res LocalRule)
              (fun s ⇒ Gbind _ _ (LocalRule (Vread res.1 h)
                                          (Loutread res.2 h)
                                          (Linread res.2 h))
```

```
                    (fun p ⇒ Greturn _ ( (update [set h] s.1
                                                      (Vwrite p.1 h)),
                                              (update (WriteArea h) s.2
                                                      (Lwrite p.2 h))))
 end.
```

End round.

Section iterated.

15.2.2 Iteration of rounds

Let LCs be e sequence of local rules, a step is the application of each element in LCs to all vertices `Fixpoint GStep (LCs: seq GLocT) (seqV: seq V) (res: VSt× LSt) : gen (VSt×LSt) :=`

```
match LCs with
 | nil ⇒ Greturn _ res
 |a1::a2 ⇒ Gbind _ _ (GRound seqV res a1)
                              (fun y ⇒ GStep a2 seqV y)
end.
```

Monte Carlo: The iteration of a step n times `Fixpoint GMC (n:nat) (LCs : seq GLocT) (seqV : seq V) (res: VSt×LSt)`

```
 : gen(VSt×LSt) :=
 match n with
   |O ⇒ Greturn _ res
   | S m ⇒ Gbind _ _ (GStep LCs seqV res)
                       (fun y ⇒ GMC m LCs seqV y)
 end.
```

End iterated.

End general.

Section port.

15.3 Message passing algorithm: rewriting over ports.

In this section, we define rounds, steps and monte carlo for algorithms which correspond to a rewriting of states over the vertices and states over ports.

We consider a simple undirected graph with vertices in V and with an edge relation Adj. `Context ‘(NG: NGraph V Adj).`

This graph is equipped with a port numbering nu: each vertex see its neighbours following a predetermined order.

```
Variable (nu: V → seq V).
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
```

```
Hypothesis Hnu2: ∀ (v:V), uniq (nu v).
```

State over vertices are of type VLab and over ports of type Plab. We assume that the set of labels on ports are not empty. `Variable (VLab: eqType) (PLab: eqType).`
`Variable pl0:PLab.`

Labelling functions over vertices and over ports. We assume that the set of ports is not empty. `Let Pt := (@port_finType V Adj).`

```
Let VSt := LabelFunc V VLab.
Let PSt := LabelFunc Pt PLab.
```

```
Variable p0: Pt.
```

Transformations of a local computation of a vertex to a global one and of a global computation to a global one.

```
Definition WriteArea (v: V) : {set Pt} :=
 outerport_set v.

Definition Vwrite (s: VLab) (v: V) : VSt:=
  (finfun (fun x ⇒ s)).

Definition Pwrite (s: seq PLab) (v: V) : PSt :=
  (finfun (fun x⇒ nth pl0 s (index (sndp x) (nu v)))).

Definition Vread (s: VSt) (v: V) : VLab :=
 (s v).

Definition Pinread (s: PSt) (v: V) : (seq PLab) :=
 map (fun x:V ⇒ s (VtoP x v p0)) (nu v).

Definition Poutread (s: PSt) (v: V) : (seq PLab) :=
  map (fun (x:V) ⇒ s (VtoP v x p0)) (nu v).

Definition Vupdate (v:V) (s: VLab × seq PLab) (old: VSt) : VSt :=
 update [set v] old (Vwrite s.1 v).

Definition Pupdate (v:V) (s: VLab × seq PLab) (old: PSt) : PSt :=
  update (outerport_set v) old (Pwrite s.2 v).

Definition VPupdate (v:V) (s: VLab × seq PLab) (old: VSt×PSt) : VSt×PSt :=
 (Vupdate v s old.1, Pupdate v s old.2).

Lemma Vupdate_1 : ∀ v w s old,
 (Vupdate v s old) w = if (w == v) then s.1 else (old w).

Lemma Pupdate_1 : ∀ u v w s old,
Adj u w →
(Pupdate v s old) (VtoP u w p0) =
(if u == v then nth pl0 s.2 (index w (nu v)) else old (VtoP u w p0)).

Lemma VPupdate_read_1 : ∀ (v w:V) (k:VLab × seq PLab) (res:VSt×PSt),
w != v →
(Vread (VPupdate w k res).1 v) = Vread res.1 v.
```

Lemma VPupdate_read_5 : ∀ (v w: V) (k: $VLab$ × seq $PLab$) (res: VSt×PSt),
(Vread (VPupdate w k res).1 v) = if (w == v) then k.1
else (Vread res.1 v).

Lemma VPupdate_read_2 : ∀ (v w: V) (k: $VLab$×seq $PLab$) (res: VSt×PSt),
w != v →
(Poutread (VPupdate w k res).2 v) = Poutread res.2 v.

Lemma VPupdate_read_3 : ∀ (v: V) (k: $VLab$×seq $PLab$) (res: VSt× PSt),
seq.size k.2 = seq.size (nu v) →
(Poutread (VPupdate v k res).2 v) = k.2.

Lemma VPupdate_read_4 : ∀ (v: V) (k: $VLab$ × seq $PLab$) (res: VSt×PSt),
(Poutread (VPupdate v k res).2 v) =
(take (seq.size (nu v)) (k.2++(nseq (seq.size (nu v)) $pl0$))).

Lemma VPupdate_read_6 : ∀ (v w: V) (k: $VLab$ × seq $PLab$) (res: VSt×PSt),
(Poutread (VPupdate w k res).2 v) = if (w == v) then
(take (seq.size (nu v)) (k.2++(nseq (seq.size (nu v)) $pl0$)))
else Poutread res.2 v.

Lemma VPupdate_read_7 : ∀ (v: V) (k: $VLab$×seq $PLab$) (res: VSt×PSt),
(Pinread (VPupdate v k res).2 v) = Pinread res.2 v.

Lemma VPupdate_1 : ∀ v w k k' x, w != v →
VPupdate w k (VPupdate v k' x) = VPupdate v k' (VPupdate w k x).

Section round.

15.3.1 Round

Let $LocPT$:= GLocT $VLab$ $PLab$.

Round for a randomized distributed algorithm: a local function is applied to all vertices which updates the global state Definition GPRound ($seqV$: seq V)(res: VSt×PSt)(LC:$LocPT$) : **gen** (VSt×PSt):=
GRound WriteArea Vwrite Pwrite Vread Pinread Poutread $seqV$ res LC.

End round.

Section iterated.

15.3.2 Iteration of rounds

Let $LocPT$:= GLocT $VLab$ $PLab$.

Let LCs be e sequence of local rules, a step is the application of each element in LCs to all vertices Definition GPStep (LCs : seq $LocPT$)($seqV$: seq V)(res: VSt×PSt):**gen**(VSt×PSt):=
GStep WriteArea Vwrite Pwrite Vread Pinread Poutread LCs $seqV$ res.

Monte Carlo: The iteration of a step n times `Definition` GPMC $(n{:}\mathsf{nat})(LCs{:}\mathsf{seq}\ LocPT)(seqV{:}\mathsf{seq}\ V)(res{:}VSt \times PSt){:}\mathbf{gen}(VSt \times PSt){:=}$
GMC WriteArea Vwrite Pwrite Vread Pinread Poutread n LCs $seqV$ res.

`End` iterated.

`End` port.

Chapter 16

Library rdaTool_op

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.
Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import op.
Require Import rdaTool_gen.
Set Implicit Arguments.
Import Prenex Implicits.
```

16.1 Introduction

Tools to simulate randomised distributed algorithms.

```
Section port.
Context `(NG: NGraph V Adj).
Variable (nu: V → seq V).
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v:V), uniq (nu v).
Variable (VLab: eqType) (PLab: eqType).
Variable pl0:PLab.
Variable (rand_t : Type)(get : nat → rand_t → nat × rand_t).
Context (rand : ORandom _ get).
Let Pt := (@port_finType V Adj).
```

```
Let VSt := LabelFunc V VLab.
Let PSt := LabelFunc Pt PLab.

Variable p0: Pt.

Let OLocT := VLab->(seq PLab)->(seq PLab)-> Op rand_t (VLab×seq PLab).

Section finfunState.

Section One.

Fixpoint OPRound (seqV:seq V)(res:VSt×PSt) (LC:OLocT):Op rand_t (VSt×PSt):=
 match seqV with
   |nil ⇒ Oreturn res
   |h::t ⇒ Obind (OPRound t res LC)
             (fun s⇒Obind (LC (Vread res.1 h)(Poutread nu p0 res.2 h)
                                                    (Pinread nu p0 res.2 h))
              (fun p ⇒ Oreturn((update [set h] s.1 (Vwrite p.1 h)),
                                    (update (WriteArea h) s.2 (Pwrite nu pl0 p.2 h)))))
 end.

Variable Lr: OLocT.
Variable Lr': GLocT VLab PLab.

Hypothesis LocalRule1: ∀ ls l1 l2,
(Opsem rand_t get rand (Lr' ls l1 l2)) =
(Lr ls l1 l2).

Lemma OPG_eq1 : ∀ (seqV: seq V) (res: VSt×PSt),
   Opsem _ get rand (GPRound nu pl0 p0 seqV res Lr') =
   OPRound seqV res Lr.

End One.

Section iterated.

Fixpoint OPStep (LCs:seq OLocT)(seqV:seq V)(res:VSt×PSt):Op rand_t (VSt×PSt) :=
match LCs with
 | nil ⇒ Oreturn res
 |a1::a2 ⇒ Obind (OPRound seqV res a1)(fun y ⇒OPStep a2 seqV y)
end.

Fixpoint OPMC (n:nat) (LCs:seq OLocT)(seqV:seq V)(res:VSt×PSt)
 : Op rand_t (VSt×PSt):=
 match n with
    |O ⇒ Oreturn res
    | S m ⇒ Obind (OPStep LCs seqV res)
                    (fun y ⇒ OPMC m LCs seqV y)
 end.

Variable LCs : seq OLocT.
Variable LCs' : seq (GLocT VLab PLab).
```

```
Fixpoint LocalRule2 (s1:seq OLocT)
 (s2:seq(VLab->(seq PLab)->(seq PLab)->gen(VLab×seq PLab))):=
 match s1,s2 with
  |t1::q1, t2 :: q2 ⇒ (∀ ls l1 l2,
                                (Opsem rand_t get rand (t2 ls l1 l2)) = (t1 ls l1 l2))
                              ∧ (LocalRule2 q1 q2)
  |nil, nil ⇒ True
  | _ , _ ⇒ False
 end.

Hypothesis LocalRule3:LocalRule2 LCs LCs'.

Lemma OPG_eq2 : ∀ (seqV:seq V)(res:VSt×PSt),
 Opsem _ get rand (GPStep nu pl0 p0 LCs' seqV res) =1
 OPStep LCs seqV res.

Lemma OPG_eq3 : ∀ (n:nat)(seqV:seq V)(res:VSt×PSt),
 Opsem _ get rand (GPMC nu pl0 p0 n LCs' seqV res) =1
 OPMC n LCs seqV res.

End iterated.

End finfunState.

Section funState.

Let PtF := (Datatypes.prod V V).
Let VStF := V → VLab.
Let PStF := PtF → PLab.

Definition updateF (T1:finType)(T2:eqType)(A:seq T1)
 (old new: T1 → T2) : T1 → T2 :=
 fun (x:T1) ⇒ if x \in A then (new x) else (old x).

Definition VwriteF (s:VLab)(v:V): VStF :=
 (fun x ⇒ s).

Definition PwriteF (s:seq PLab)(v:V) : PStF :=
 (fun x ⇒ nth pl0 s (index x.2 (nu v))).

Definition VreadF (s:VStF)(v:V) : VLab :=
 (s v).

Definition PoutreadF (s:PStF)(v:V) : (seq PLab) :=
 map (fun (x:V) ⇒ s (v,x)) (nu v).

Definition PinreadF (s:PStF)(v:V) : (seq PLab) :=
 map (fun x:V ⇒ s (x, v)) (nu v).

Definition WriteAreaF (v: V) : seq (V × V) :=
 map (fun x ⇒ (v,x) ) (nu v).

Lemma OPF_eq1 : ∀ (resL:PSt) (resLF:PStF) (u:V),
```

```
 (∀ v w, Adj v w → resL (VtoP v w p0) = resLF (v,w)) →
 [seq resL (VtoP x u p0) | x ← nu u] = [seq resLF (x, u) | x ← nu u].
Lemma OPF_eq2 : ∀ (resL:PSt)(resLF: PStF) (u:V),
 (∀ v w, Adj v w → resL (VtoP v w p0) = resLF (v,w)) →
 [seq resL (VtoP u x p0) | x ← nu u] = [seq resLF (u, x) | x ← nu u].
Section one.
Fixpoint OPFRound (seqV:seq V)(res:VStF×PStF)(LC:OLocT): Op rand_t (VStF×PStF)
:=
 match seqV with
  |nil ⇒ Oreturn res
  |h::t ⇒ Obind (OPFRound t res LC)
    (fun s ⇒ Obind (LC (VreadF res.1 h)(PoutreadF res.2 h)(PinreadF res.2 h))
      (fun p ⇒ Oreturn((updateF (h::nil) s.1 (VwriteF p.1 h)),
                       (updateF (WriteAreaF h) s.2 (PwriteF p.2 h)))))
 end.
Variable LC : OLocT.
Lemma OPF_eq3 (seqV: seq V) : ∀ (n:rand_t)(res:VSt×PSt)(resF:VStF×PStF) ,
 (∀ v, res.1 v = resF.1 v) →
 (∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   ((OPRound seqV res LC) n).2 =
   ((OPFRound seqV resF LC) n).2 .
Lemma OPF_eq4 (l: seq V): ∀ (u:V)(n:rand_t)(res: VSt× PSt)(resF:VStF×PStF),
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
  LC (Vread res.1 u) (Poutread nu p0 res.2 u)(Pinread nu p0 res.2 u)
     (OPRound l res LC n).2 =
  LC (VreadF resF.1 u)(PoutreadF resF.2 u)(PinreadF resF.2 u)
     (OPFRound l resF LC n).2.
Lemma OPF_eq5:∀ (n:rand_t)(v:V)(seqV: seq V)(res:VSt×PSt)(resF:VStF×PStF),
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
 (((OPRound seqV res LC) n).1).1 v=
 (((OPFRound seqV resF LC) n).1).1 v.
Lemma OPF_eq6 : ∀ (n:rand_t)(v w:V)(seqV:seq V)
 (res:VSt×PSt)(resF:VStF×PStF) ,
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
 Adj v w →
 (((OPRound seqV res LC) n).1).2 (VtoP v w p0)=
 (((OPFRound seqV resF LC) n).1).2 (v,w).
```

```
End one.

Section iterated.

Fixpoint OPFStep (LCs:seq OLocT)(seqV:seq V)(res:VStF×PStF)
 : Op rand_t (VStF×PStF) :=
match LCs with
 | nil ⇒ Oreturn res
 |a1::a2 ⇒ Obind (OPFRound seqV res a1) (fun y ⇒ OPFStep a2 seqV y)
end.

Fixpoint OPFMC (n:nat)(LCs : seq OLocT)(seqV: seq V)(res:VStF×PStF)
 : Op rand_t (VStF×PStF) :=
 match n with
   |O ⇒ Oreturn res
   | S m ⇒ Obind (OPFStep LCs seqV res)
                    (fun y ⇒ OPFMC m LCs seqV y)
 end.

Variable LCs : seq OLocT.

Lemma OPF_eq7 : ∀ (n:rand_t)(seqV:seq V)(res:VSt×PSt)(resF:VStF×PStF) ,
 (∀ v, res.1 v = resF.1 v) →
 (∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   ((OPStep LCs seqV res) n).2 =
   ((OPFStep LCs seqV resF) n).2 .

Lemma OPF_eq8 : ∀ (v:V)(n:rand_t)(seqV:seq V)(res:VSt×PSt)(resF:VStF×PStF),
 (∀ v, res.1 v = resF.1 v) →
 (∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   (((OPStep LCs seqV res) n).1).1 v =
   (((OPFStep LCs seqV resF) n).1).1 v .

Lemma OPF_eq9 : ∀ (v w:V)(n:rand_t)(seqV:seq V)
 (res:VSt×PSt)(resF:VStF×PStF),
 (∀ v, res.1 v = resF.1 v) →
 (∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
 Adj v w →
   (((OPStep LCs seqV res) n).1).2 (VtoP v w p0)=
   (((OPFStep LCs seqV resF) n).1).2 (v,w).

Lemma OPF_eq10 : ∀ (m:nat)(n:rand_t)(seqV:seq V)
 (res:VSt×PSt)(resF:VStF×PStF) ,
 (∀ v, res.1 v = resF.1 v) →
 (∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   ((OPMC m LCs seqV res) n).2 =
   ((OPFMC m LCs seqV resF) n).2 .

Lemma OPF_eq11 : ∀(m:nat)(v:V)(n:rand_t)(seqV:seq V)
```

```
  (res: VSt × PSt)(resF: VStF × PStF),
  (∀ v, res.1 v = resF.1 v) →
  (∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
    (((OPMC m LCs seqV res) n).1).1 v =
    (((OPFMC m LCs seqV resF) n).1).1 v .
Lemma OPF_eq12 : ∀ (m:nat) (v w:V)(n:rand_t)(seqV:seq V)
  (res: VSt × PSt)(resF: VStF × PStF),
  (∀ v, res.1 v = resF.1 v) →
  (∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
  Adj v w →
    (((OPMC m LCs seqV res) n).1).2 (VtoP v w p0)=
    (((OPFMC m LCs seqV resF) n).1).2 (v,w).
End iterated.
Fixpoint displayOP (seqV: seq V) (m: VStF × PStF) :=
 match seqV with
   |nil ⇒ nil
   |t::q ⇒ ( (t, m.1 t) ,
          map (fun x ⇒ (x , m.2 (t , x))) (nu t)) :: (displayOP q m)
 end.
End funState.
End port.
```

Chapter 17

Library rdaTool_dist

```
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
```

```
Require Export Cover.
Require Export Prog.
Require Export Ccpo.
Require Export Rplus.
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun tuple.
```

```
Require Import my_alea.
Require Import my_ssr.
Require Import labelling.
Require Import graph.
Require Import term.
Require Import gen.
Require Import dist.
Require Import rdaTool_gen.
```

```
Set Implicit Arguments.
Import Prenex Implicits.
```

```
Open Local Scope U_scope.
Open Local Scope O_scope.
```

17.1 Introduction

This file gives tools to analyse randomised distributed algorithms

```
Section general.
```

17.2 General

```
Variable (V: finType) (VLab: eqType).
Variables (L:finType) (LLab:eqType).
Definition VSt := LabelFunc V VLab.
Definition LSt := LabelFunc L LLab.
Variable WriteArea : V → {set L}.
Variable ReadArea : V → {set L}.
Variable Vwrite : VLab → V → VSt.
Variable Lwrite : (seq LLab) → V → LSt.
Variable Vread : VSt → V → VLab.
Variable Linread : LSt → V → (seq LLab).
Variable Loutread : LSt → V → (seq LLab).
Hypothesis Vread1 : ∀ v w resV f,
 v != w →
(Vread resV v) =
(Vread (update [set w] resV f) v).
Hypothesis Loutread1 : ∀ v w resL f,
 v != w →
  (Loutread resL v) =
  (Loutread (update (WriteArea w) resL f) v).
Definition DLocT:= VLab → (seq LLab) → (seq LLab) → distr (VLab × seq LLab).
Section round.
```

17.2.1 Round

```
Fixpoint DRound (seqV:seq V)(res:VSt×LSt)(LR:DLocT) : distr (VSt×LSt) :=
match seqV with
|nil ⇒ Munit res
|h::t ⇒ Mlet (DRound t res LR)
        (fun s ⇒ Mlet (LR (Vread res.1 h) (Loutread res.2 h)(Linread res.2 h))
            (fun p ⇒ Munit((update [set h] s.1 (Vwrite p.1 h)),
                            (update (WriteArea h) s.2 (Lwrite p.2 h)))))
end.
Section gen.
```

Lemmas: Gen

```
Variable DLr: DLocT.
Variable GLr: GLocT VLab LLab.
```

```
Hypothesis LocalRule1: ∀ ls l1 l2,
(Distsem (GLr ls l1 l2)) =
(DLr ls l1 l2).

Lemma DG_eq1 : ∀ (seqV: seq V) (res: VSt×LSt),
 Distsem (GRound WriteArea Vwrite Lwrite Vread Linread Loutread seqV res GLr) =
 DRound seqV res DLr.

End gen.

Section lemmas.

Variable Lr: DLocT.
```

Lemmas: Simplification

```
Lemma DRoundcons1 : ∀ (v:V) (t:seq V) (res:VSt×LSt),
 DRound (v::t) res Lr = Mlet (DRound t res Lr)
              (fun s ⇒ Mlet (Lr (Vread res.1 v)
                                   (Loutread res.2 v)
                                   (Linread res.2 v))
                (fun p ⇒ Munit ( (update [set v] s.1 (Vwrite p.1 v)),
                                  (update (WriteArea v) s.2
                                    (Lwrite p.2 v))))).

Lemma DRoundcons2 : ∀ (v:V) (t:seq V) (res:VSt×LSt),
 is_discrete_s (Lr (Vread res.1 v)(Loutread res.2 v)(Linread res.2 v)) →
 DRound (v::t) res Lr ==
       Mlet (Lr (Vread res.1 v)(Loutread res.2 v)(Linread res.2 v))
            (fun p ⇒ Mlet (DRound t res Lr)
                (fun s ⇒ Munit ((update [set v] s.1 (Vwrite p.1 v)),
                              (update (WriteArea v) s.2 (Lwrite p.2 v))))).

Lemma DRoundcons3 : ∀ (v:V) (t:seq V) (res:VSt×LSt),
 (∀ a b c d, Lr a b c = Lr a b d) →
 (∀ w, is_discrete_s
              (Lr (Vread res.1 w)(Loutread res.2 w)(Linread res.2 w))) →
 (∀ v w, v != w → disjoint (mem (WriteArea v)) (mem (WriteArea w)))->
 v \notin t →
 DRound (v::t) res Lr ==
 Mlet (Lr (Vread res.1 v) (Loutread res.2 v)(Linread res.2 v))
      (fun p ⇒ (DRound t ((update [set v] res.1 (Vwrite p.1 v)),
                (update (WriteArea v) res.2 (Lwrite p.2 v))) Lr)).
```

Lemmas: Termination

```
Lemma DRound_total : ∀(s: seq V) (res: VSt× LSt),
```

```
(∀ w, Term (Lr (Vread res.1 w) (Loutread res.2 w)(Linread res.2 w))) →
 Term (DRound s res Lr).
```

Lemmas: Commutativity

```
Lemma DRoundCommute0 : ∀ (t:V) (s1 s2:seq V) (res:VSt×LSt),
is_discrete_s (Lr (Vread res.1 t) (Loutread res.2 t)(Linread res.2 t)) →
 (∀ k, Mlet (DRound s1 res Lr)
      (fun s : VSt × LSt ⇒
        Munit
          (update [set t] s.1 (Vwrite k.1 t),
          update (WriteArea t) s.2 (Lwrite k.2 t))) ==
   Mlet (DRound s2 res Lr)
      (fun s : VSt × LSt ⇒
        Munit
          (update [set t] s.1 (Vwrite k.1 t),
          update (WriteArea t) s.2 (Lwrite k.2 t)))) →
   DRound (t::s1) res Lr == DRound (t::s2) res Lr.
```

```
Lemma DRoundCommute1 : ∀ (v t:V) (s:seq V) (res:VSt×LSt),
is_discrete_s (Lr (Vread res.1 v) (Loutread res.2 v)(Linread res.2 v)) →
is_discrete_s (Lr (Vread res.1 t) (Loutread res.2 t)(Linread res.2 t)) →
 disjoint (mem (WriteArea t)) (mem (WriteArea v)) →
 v != t →
 DRound (v::t::s) res Lr == DRound (t::v::s) res Lr.
```

```
Lemma DRoundCommute2 : ∀ (s: seq V) (t:V) (res:VSt×LSt),
(∀ v,is_discrete_s
           (Lr (Vread res.1 v) (Loutread res.2 v)(Linread res.2 v))) →
 (∀ v w, v != w → disjoint (mem (WriteArea v)) (mem (WriteArea w))) →
 t \in s →
 DRound s res Lr == DRound (t::(rem t s)) res Lr.
```

```
Lemma DRoundCommute3 :
∀ (s1 s2: (seq V)) (res:VSt×LSt),
  (∀ v, is_discrete_s
    (Lr (Vread res.1 v) (Loutread res.2 v)(Linread res.2 v))) →
 (∀ v w, v != w → disjoint (mem (WriteArea v)) (mem (WriteArea w))) →
  perm_eq s1 s2 →
  (DRound s1) res Lr == (DRound s2) res Lr.
```

```
Section caraclocal.
```

Lemmas: Preservation local/global

Preservation of the local probability to a global one

```
Variable carac_local : V → VLab × seq LLab → U.
Variable carac_global : V → VSt × LSt → U.

Lemma FLocalGlobal :
 (∀ a b c d, Lr a b c = Lr a b d) →
 (∀ v w y resV resL,
   (Lr (Vread resV w) (Loutread resL w) (Linread resL w)) =
    Lr (Vread (update [set v] resV (Vwrite y.1 v)) w)
       (Loutread (update (WriteArea v) resL (Lwrite y.2 v)) w)
       (Linread (update (WriteArea v) resL (Lwrite y.2 v)) w)) →

 ∀ (v: V) (res:VSt×LSt) (x:U),
 (∀ w, Term (Lr (Vread res.1 w) (Loutread res.2 w)(Linread res.2 w))) →
  (∀ w, is_discrete_s
            (Lr (Vread res.1 w) (Loutread res.2 w)(Linread res.2 w))) →
 (∀ u w, u != w → disjoint (mem (WriteArea u)) (mem (WriteArea w))) →
 (∀ (v:V) (y:VLab × seq LLab) (res:VSt×LSt),
   carac_global v
     (update [set v] res.1 (Vwrite y.1 v),
     update (WriteArea v) res.2 (Lwrite y.2 v)) ==
   carac_local v y) →
(mu (Lr(Vread res.1 v)(Loutread res.2 v)(Linread res.2 v)))(carac_local v)==x→
(mu (DRound (enum V) res Lr)) (carac_global v) == x.

End caraclocal.
```

Lemmas: Independence

Here is a generalization of the independence the probability to be computed has to have some properties

```
Definition indepProp (f1 f2:VSt × LSt → bool)
 (c c': VLab × seq LLab → bool) :=
∀ t:V,
( (∀ x sn, f1 (update [set t] sn.1 (Vwrite x.1 t),
 update (WriteArea t) sn.2 (Lwrite x.2 t)) = f1 sn) ∧
 (∀ x sn, f2 (update [set t] sn.1 (Vwrite x.1 t),
 update (WriteArea t) sn.2 (Lwrite x.2 t)) = f2 sn))
∨
( (∀ x sn, f1 (update [set t] sn.1 (Vwrite x.1 t),
 update (WriteArea t) sn.2 (Lwrite x.2 t)) = c x) ∧
 (∀ x sn, f2 (update [set t] sn.1 (Vwrite x.1 t),
 update (WriteArea t) sn.2 (Lwrite x.2 t)) = f2 sn))
∨
( (∀ x sn, f1 (update [set t] sn.1 (Vwrite x.1 t),
```

```
update (WriteArea t) sn.2 (Lwrite x.2 t)) = f1 sn) ∧
(∀ x sn, f2 (update [set t] sn.1 (Vwrite x.1 t),
update (WriteArea t) sn.2 (Lwrite x.2 t)) = c' x)).
```

```
Lemma DRoundindepb : ∀ (sV:seq V) (sT:VSt×LSt)
 (f1 f2:VSt × LSt → bool) (c c':VLab × seq LLab → bool),
 (∀ w : V,
    Term (Lr (Vread sT.1 w) (Loutread sT.2 w)(Linread sT.2 w))) →
 (indepProp f1 f2 c c') →
 indepb (DRound sV sT Lr) f1 f2.
```

```
End lemmas.
End round.
```

```
Section roundlv.
```

17.2.2 Infinite iteration of Round

```
Variable Lr : DLocT.
Variable termB : (VSt×LSt) → bool.
```

DRoundLV

```
Instance DRoundLV_mon (seqV : seq V) :
 monotonic (fun f (s:VSt × LSt) ⇒
    if (termB s) then Munit s
    else Mlet (DRound seqV s Lr) (fun r ⇒ f r)).
```

```
Definition DRoundLV (seqV: seq V):=
 mon (fun f (s:VSt×LSt) ⇒
   if (termB s) then (Munit s) else
   (Mlet (DRound seqV s Lr) (fun r ⇒ f r))).
```

```
Lemma DRoundLV_simpl : ∀ f (seqV: seq V) (res:VSt×LSt),
 DRoundLV seqV f res =
 if (termB res) then Munit res
 else Mlet (DRound seqV res Lr) (fun r ⇒ f r).
```

```
Lemma DRoundLV_cont : ∀ (seqV : seq V),
 continuous (DRoundLV seqV).
```

```
Lemma DRoundLVcons1 : ∀ (v:V) (t:seq V) f (res:VSt×LSt),
 DRoundLV (v::t) f res == if (termB res) then Munit res
 else Mlet (Mlet (DRound t res Lr)
          (fun s ⇒ Mlet (Lr (Vread res.1 v)(Loutread res.2 v)(Linread res.2 v))
             (fun p ⇒ Munit ((update [set v] s.1 (Vwrite p.1 v)),
                              (update (WriteArea v) s.2 (Lwrite p.2 v))))))
```

```
            (fun r ⇒ f r).
Lemma DRoundLVcons2 : ∀ (v:V) (t:seq V) f (res:VSt×LSt),
 is_discrete_s (Lr (Vread res.1 v)(Loutread res.2 v)(Linread res.2 v)) →
 DRoundLV (v::t) f res ==
 if (termB res) then Munit res
 else Mlet (Mlet (Lr (Vread res.1 v)(Loutread res.2 v)(Linread res.2 v))
       (fun p ⇒ Mlet (DRound t res Lr)
          (fun s ⇒ Munit ((update [set v] s.1 (Vwrite p.1 v)),
                      (update (WriteArea v) s.2 (Lwrite p.2 v))))))
                 (fun r ⇒ f r).
Lemma DRoundLV_total : ∀ (s:seq V) (res: VSt×LSt) f,
(∀ w, Term (Lr (Vread res.1 w)(Loutread res.2 w)(Linread res.2 w))) →
 (∀ x, Term (f x)) →
 Term (DRoundLV s f res).
```

DRoundFixLV

```
Definition DRoundFixLV (seqV: seq V) : (VSt × LSt) → distr (VSt×LSt) :=
 Mfix (DRoundLV seqV).
Hypothesis localTerm : ∀ res,
   Term (DRound (enum V) res Lr).
Variable cardTermB : (VSt×LSt) → nat.
Variable c: U.
Let k := [1-] c.
Hypothesis hcard1 : ∀ r, (cardTermB r) = O → termB r.
Hypothesis hcard2: 0 < c.
Lemma hcard2' : k < 1.
Hypothesis hcard3 : ∀ (res: VSt×LSt),
  (0 < cardTermB res)%nat →
 c ≤ mu (DRound (enum V) res Lr)
   (fun x ⇒ B2U (lt_dec (cardTermB x) (cardTermB res))).
Hypothesis hcard4 : ∀ (res: VSt×LSt),
 mu (DRound (enum V) res Lr)
   (fun x ⇒ B2U (lt_dec (cardTermB res) (cardTermB x))) == 0.
Lemma DRoundfix_total : ∀ (res: VSt × LSt),
 Term (DRoundFixLV (enum V) res).
End roundlv.
Section iteration.
```

17.2.3 Iteration

```
Fixpoint DStep (LCs : seq DLocT) (seqV: seq V)(res: VSt× LSt) : distr (VSt×LSt) :=
match LCs with
 | nil ⇒ Munit res
 |a1 : : a2 ⇒ Mlet (DRound seqV res a1) (fun y ⇒ DStep a2 seqV y)
end.
```

```
Fixpoint DMC (n:nat) (LCs:seq DLocT)(seqV:seq V)(res:VSt×LSt) : distr(VSt×LSt) :=
 match n with
   |O ⇒ Munit res
   | S m ⇒ Mlet (DStep LCs seqV res) (fun y ⇒ DMC m LCs seqV y)
 end.
```

Lemmas: Gen

```
Variable DLCs : seq DLocT.
Variable GLCs : seq (GLocT VLab LLab).
```

```
Fixpoint LocalRule2 (s1:seq DLocT) (s2:seq(GLocT VLab LLab)) :=
 match s1,s2 with
  |t1 : : q1, t2 :: q2 ⇒ (∀ ls l1 l2, (Distsem (t2 ls l1 l2)) = (t1 ls l1 l2))
                                          ∧ (LocalRule2 q1 q2)
  |nil, nil ⇒ True
  | _ , _ ⇒ False
 end.
```

```
Hypothesis LocalRule3:LocalRule2 DLCs GLCs.
```

```
Lemma DG_eq2 : ∀ (seqV:seq V)(res:VSt×LSt),
 Distsem (GStep WriteArea Vwrite Lwrite Vread Linread Loutread GLCs seqV res) ==
 DStep DLCs seqV res.
```

```
Lemma DG_eq3 : ∀ (n:nat)(seqV:seq V)(res:VSt×LSt),
 Distsem (GMC WriteArea Vwrite Lwrite Vread Linread Loutread n GLCs seqV res)
 == DMC n DLCs seqV res.
```

Infinite iteration of Steps

```
Variable termB : VSt × LSt → bool.
Variable LCs: seq DLocT.
```

```
Instance DStepLV_mon (seqV: seq V):
 monotonic (fun f (s:VSt × LSt) ⇒
   if (termB s) then Munit s
   else Mlet (DStep LCs seqV s) (fun r ⇒ f r)).
```

```
Definition DStepLV (seqV: seq V):=
```

```
 mon (fun f (s:VSt×LSt) ⇒
  if (termB s) then (Munit s) else
  (Mlet (DStep LCs seqV s) (fun r ⇒ f r))).
Lemma DStepLV_simpl : ∀ f (seqV : seq V)(res: VSt × LSt),
 DStepLV seqV f res =
 if (termB res) then Munit res
 else Mlet (DStep LCs seqV res) (fun r ⇒ f r).
Lemma DStepLV_cont :
 ∀ seqV, continuous (DStepLV seqV).
```

DStepFixLV

```
Definition DLV (seqV: seq V): (VSt × LSt) → distr (VSt×LSt) :=
 Mfix (DStepLV seqV).
Hypothesis localTerm : ∀ seqV res,
   Term (DStep LCs seqV res).
Variable cardTermB : VSt × LSt → nat.
Variable c: U.
Let k := [1-] c.
Hypothesis hcard1 : ∀ r, (cardTermB r) = O → termB r.
Hypothesis hcard2: 0 < c.
Hypothesis hcard3 : ∀ (seqV: seq V) (res: VSt× LSt),
 (0 < cardTermB res)%nat →
 c ≤ mu (DStep LCs seqV res)
   (fun x ⇒ B2U (lt_dec (cardTermB x) (cardTermB res))).
Hypothesis hcard4 : ∀ (seqV: seq V) (res: VSt ×LSt),
 mu (DStep LCs seqV res)
   (fun x ⇒ B2U (lt_dec (cardTermB res) (cardTermB x))) == 0.
Lemma DLV_total : ∀ (seqV: seq V)(res: VSt × LSt),
 Term (DLV seqV res).
End iteration.
End general.
Section port.
```

17.3 Port algorithms

```
Context `(NG: NGraph V Adj).
Variable (nu: V → seq V).
```

```
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v:V), uniq (nu v).

Variable (VLab: eqType) (PLab: eqType).
Variable pl0:PLab.

Let Pt := (@port_finType V Adj).
Let VSt := LabelFunc V VLab.
Let PSt := LabelFunc Pt PLab.

Variable p0: Pt.

Let DLocT := VLab → (seq PLab) → (seq PLab) → distr (VLab × seq PLab).

Section round.

Definition DPRound (seqV: seq V) (res: VSt× PSt) (LC:DLocT)
: distr (VSt×PSt) :=
 DRound WriteArea (@Vwrite _ VLab) (Pwrite nu pl0) (@Vread _ VLab)
                  (Pinread nu p0) (Poutread nu p0) seqV res LC.

Section gen.
```

Lemmas: Gen

```
Variable DLr: DLocT.
Variable GLr: GLocT VLab PLab.

Hypothesis LocalRule1: ∀ ls l1 l2,
(Distsem (GLr ls l1 l2)) =
(DLr ls l1 l2).

Lemma DPG_eq1 : ∀ (seqV: seq V) (res: VSt×PSt),
 Distsem (GPRound nu pl0 p0 seqV res GLr) =
 DPRound seqV res DLr.

End gen.

Section lemmas.

Variable Lr: DLocT.
```

Lemmas: Other

```
Lemma DPRound_total : ∀(s: seq V) (res: VSt× PSt),
(∀ w, Term (Lr (Vread res.1 w) (Poutread nu p0 res.2 w) (Pinread nu p0 res.2 w))) →
 Term (DPRound s res Lr).

Lemma DPRoundCommute : ∀ (s1 s2: (seq V)) (res: VSt× PSt),
  (∀ v, is_discrete_s
        (Lr (Vread res.1 v) (Poutread nu p0 res.2 v) (Pinread nu p0 res.2 v))) →
  perm_eq s1 s2 →
```

(DPRound *s1*) *res Lr* == (DPRound *s2*) *res Lr*.

End lemmas.
End round.

Section roundlv.

17.3.1 Infinite iteration of Round

Variable *Lr* : *DLocT*.
Variable *termB* : (*VSt*×*PSt*) → **bool**.

DPRoundLV

```
Definition DPRoundLV (seqV : seq V) :=
 DRoundLV WriteArea (@Vwrite _ VLab) (Pwrite nu pl0) (@Vread _ VLab)
                    (Pinread nu p0) (Poutread nu p0) Lr termB seqV.
```

DPRoundFixLV

```
Definition DPRoundFixLV (seqV: seq V) : (VSt × PSt) → distr (VSt×PSt) :=
DRoundFixLV WriteArea (@Vwrite _ VLab) (Pwrite nu pl0) (@Vread _ VLab)
                    (Pinread nu p0) (Poutread nu p0) Lr termB seqV.
```

End roundlv.

Section iteration.

17.3.2 Iteration

```
Definition DPStep (LCs : seq DLocT) (seqV: seq V)(res: VSt× PSt)
 : distr (VSt×PSt) :=
DStep WriteArea (@Vwrite _ VLab) (Pwrite nu pl0) (@Vread _ VLab)
                    (Pinread nu p0) (Poutread nu p0) LCs seqV res.
```

```
Definition DPMC (n:nat) (LCs:seq DLocT)(seqV:seq V)(res:VSt×PSt)
 : distr(VSt×PSt) :=
DMC WriteArea (@Vwrite _ VLab) (Pwrite nu pl0) (@Vread _ VLab)
                    (Pinread nu p0) (Poutread nu p0) n LCs seqV res.
```

Lemmas: Gen

Variable *DLCs* : seq *DLocT*.
Variable *GLCs* : seq (GLocT *VLab PLab*).

Hypothesis *LocalRule3*:LocalRule2 *DLCs GLCs*.

Lemma DPG_eq2 : ∀ (*seqV*:seq *V*)(*res*:*VSt*×*PSt*),

```
 Distsem (GPStep nu pl0 p0 GLCs seqV res) == DPStep DLCs seqV res.
Lemma DPG_eq3 : ∀ (n:nat)(seqV:seq V)(res:VSt×PSt),
 Distsem (GPMC nu pl0 p0 n GLCs seqV res)
 == DPMC n DLCs seqV res.
```

Infinite iteration of Steps

```
Variable termB : VSt × PSt → bool.
Variable LCs: seq DLocT.
Definition DPStepLV (seqV: seq V):=
DStepLV WriteArea (@Vwrite _ VLab) (Pwrite nu pl0) (@Vread _ VLab)
                  (Pinread nu p0) (Poutread nu p0) termB LCs seqV.
```

DPStepFixLV

```
Definition DPLV (seqV: seq V): (VSt × PSt) → distr (VSt×PSt) :=
 DLV WriteArea (@Vwrite _ VLab) (Pwrite nu pl0) (@Vread _ VLab)
                  (Pinread nu p0) (Poutread nu p0) termB LCs seqV.
Hypothesis localTerm : ∀ seqV res,
   Term (DPStep LCs seqV res).
Variable cardTermB : VSt × PSt → nat.
Variable c: U.
Let k := [1-] c.
Hypothesis hcard1 : ∀ r, (cardTermB r) = O → termB r.
Hypothesis hcard2: 0 < c.
Variable PR : VSt × PSt → bool.
Variable seqV : seq V.
Hypothesis hcard3 : ∀ (res: VSt× PSt),
(0 < (cardTermB res))%nat →
PR res →
c ≤ mu (DPStep LCs seqV res)
   (fun x ⇒ B2U (lt_dec (cardTermB x) (cardTermB res))).
Hypothesis hcard4 : ∀ (res: VSt ×PSt),
PR res →
 mu (DPStep LCs seqV res)
   (fun x ⇒ B2U (lt_dec (cardTermB res) (cardTermB x))) == 0.
Lemma DPLV_total : ∀ (res: VSt × PSt),
 PR res →
 (∀ s f, PR s ->(mu (DPStep LCs seqV s))(fun x : VSt × PSt ⇒
            B2U (PR x) × (f x)) == (mu (DPStep LCs seqV s)) f) →
```

Term (DPLV $seqV$ res).

```
End iteration.
End port.
```

Chapter 18

Library term

```
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Add LoadPath "../prelude".
Add LoadPath "../graph".

Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun tuple.

Require Export Cover.
Require Export Prog.
Require Export Ccpo.

Require Import my_alea.
Require Import my_ssr.
Require Import labelling.

Set Implicit Arguments.
Import Prenex Implicits.

Open Local Scope U_scope.
Open Local Scope O_scope.
```

18.1 Introduction

Proof of termination of a randomised distributed algorithm (not necessarily a Las Vegas). Part of this proof is from Alea Library of C. Paulin.

```
Section termfglobal.
Variable (V: finType) (VLabel: eqType).
Variables (Locat:finType) (LLabel:eqType).

Let VState := LabelFunc V VLabel.
Let LState := LabelFunc Locat LLabel.

Variable rd : VState × LState → distr (VState×LState).
```

```
Hypothesis localTerm : ∀ s,
   Term (rd s).

Variable termB : VState × LState → bool.

Instance FGlobal_mon :
 monotonic (fun f (s:VState×LState) ⇒
   if (termB s) then Munit (s)
   else Mlet (rd s) (fun r ⇒ f r)).

Definition FGlobal :=
 mon (fun f (s:VState×LState) ⇒
  if (termB s) then (Munit s) else
  (Mlet (rd s) (fun r ⇒ f r))).

Lemma FGlobal_simpl : ∀ f (s:VState ×LState),
 FGlobal f s =
 if (termB s) then Munit s
 else Mlet (rd s) (fun r ⇒ f r).

Definition fglobal : (VState×LState) → distr (VState×LState) :=
 Mfix (FGlobal).

Variable cardTermB : VState × LState → nat.

Variable c: U.
Definition k := [1-] c.

Hypothesis hcard1 : ∀ s, (cardTermB s) = O → termB s = true.

Hypothesis hcard2: 0 < c.

Lemma hcard2' : k < 1.

Variable (PR : VState×LState → bool).

Hypothesis hcard3 : ∀ (s:VState×LState),
 (0 < (cardTermB s))%nat →
 PR s →
 c ≤ mu (rd s)
   (fun x ⇒ B2U (lt_dec (cardTermB x) (cardTermB s))).

Lemma hcard3' : ∀ (s:VState×LState),
  (0 < (cardTermB s))%nat →
 PR s →
 mu (rd s)
  (finv (fun x:VState×LState ⇒ B2U (lt_dec (cardTermB x) (cardTermB s)))) ≤ k.

Hypothesis hcard4 : ∀ (s:VState×LState), PR s →
 mu (rd s)
  (fun x ⇒ B2U (lt_dec (cardTermB s) (cardTermB x))) == 0.

Lemma hcard4' : ∀ (s:VState×LState ), PR s →
```

```
 mu (rd s)
  (finv (fun x ⇒ B2U(lt_dec (cardTermB s) (cardTermB x)))) == 1.

Lemma hcardmu : ∀ s, PR s →
 (mu (rd s)
     (finv (fun x ⇒ B2U (eq_nat_dec (cardTermB x) (cardTermB s))))) ≤
  (mu (rd s)
      (fun x ⇒ B2U (lt_dec (cardTermB x) (cardTermB s)))).

Lemma hcardmu' : ∀ s, PR s →
(mu (rd s))
    (fun x ⇒ B2U (eq_nat_dec (cardTermB x) (cardTermB s))) ≤
 (mu (rd s))
    (finv (fun x ⇒ B2U (lt_dec (cardTermB x) (cardTermB s)))).

Lemma hcard5 : ∀ s a b,
  (0 < (cardTermB s))%nat →
 a ≤ b → PR s →
 k × a + [1-]k × b ≤
 mu (rd s)
  (fun x⇒B2U (eq_nat_dec (cardTermB x) (cardTermB s))) × a +
 mu (rd s)
  (fun x ⇒ B2U (lt_dec (cardTermB x) (cardTermB s))) × b.

Fixpoint pw_ (x n : nat) : U :=
  match n with O ⇒ 0
             | (S n) ⇒ match x with
                          O ⇒ 1
                       | S y ⇒ k × pw_ x n + ([1-] k) × pw_ y n
                         end
  end.

Lemma pw_decrS_x : ∀ n x, pw_ (S x) n ≤ pw_ x n.
Hint Resolve pw_decrS_x.

Lemma pw_decr_x : ∀ n x y, (x ≤ y)%nat → pw_ y n ≤ pw_ x n.
Hint Resolve pw_decr_x.

Lemma pw_incr : ∀ x n, pw_ x n ≤ pw_ x (S n).

Hint Resolve pw_incr.

Definition pw : nat → nat -m> U
    := fun x ⇒ fnatO_intro (pw_ x) (pw_incr x).

Lemma pw_pw_ : ∀ x n, pw x n = pw_ x n.

Lemma pw_simpl : ∀ x n, pw x n =
    match n with O ⇒ 0
              | (S n) ⇒ match x with
```

```
                                      O ⇒ 1
                                    | S y ⇒ k × pw x n + ([1-] k) × pw y n
                                    end
     end.
Lemma pwS_simpl : ∀ x n, pw (S x) (S n) = k × pw (S x) n + [1-]k × (pw x n).
Lemma lim_pw_one : ∀ x, lub (pw x) == 1.
Lemma termglobal : ∀ (s: VState×LState),
 PR s →
(∀ s f, PR s ->(mu (rd s))(fun x : VState × LState ⇒
              B2U (PR x) × (f x)) == (mu (rd s)) f) →
 Term (fglobal s).
End termfglobal.
```

Chapter 19

Library symBreak

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat.
Require Import fintype finset fingraph seq finfun bigop choice tuple.
Import Prenex Implicits.
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Require Import Rplus.
Require Import my_alea.
Require Import my_ssr.
Require Import my_ssralea.
Require Import graph.
Require Import labelling.
Require Import rdaTool_gen.
Require Import rdaTool_dist.
Set Implicit Arguments.
```

19.1 Introduction

Definition and analysis of the symmetry break over a graph composed of one edge.

State of a vertex: (Active/Inactive ; Sequence of drawn bits) State of a port : Drawn bit Local Computation : If the received message is different from the sent one then stop Else draw a bit, record it in the state and send it

```
Section symBreak.
```

```
Definition V : finType := bool_finType.

Definition u1 := true.
Definition u2 := false.

Definition Adj : (rel V) :=
 (fun x y ⇒ match x,y with
             |u1,u2 ⇒ true
             |u2,u1 ⇒ true
             |_,_ ⇒ false
            end).

Definition nu (v: V) : seq V :=
 match v with
  |u1 ⇒ (u2::nil)
  |u2 ⇒ (u1::nil)
 end.

Context `(NG: NGraph V Adj).

Definition VLabel : eqType := (prod_eqType bool_eqType
 (seq_eqType bool_eqType)).

Definition Pt := (@port_finType V Adj).
Definition PLabel : eqType := bool_eqType.

Definition VState := LabelFunc V VLabel.
Definition PState := LabelFunc Pt PLabel.

Lemma edge_ft : Adj (u2,u1).1 (u2,u1).2.

Lemma edge_tf : Adj (u1,u2).1 (u1,u2).2.

Definition pft:= Port edge_ft.
Definition ptf:= Port edge_tf.

Lemma VtoPft : (VtoP u2 u1 pft) = pft.

Lemma VtoPtf : (VtoP u1 u2 pft) = ptf.

Lemma enumbool1: (enum bool_finType) = (u1::u2::nil).

Lemma update1 (x0:VState) (x1 x2 :bool × seq bool):
update [set u1]
  (update [set u2] x0
       (Vwrite x1 u2))
                  (Vwrite x2 u1) =
  [ffun y ⇒ if y then x2 else x1].

Lemma update2 (x0: PState)
 (x1 x2 : seq bool):
 update (WriteArea u1)
   (update (WriteArea u2) x0
```

```
(Pwrite nu false x1 u2))
                  (Pwrite nu false x2 u1)
 = [ffun y ⇒
 if (fstp y) then (match x2 with
   | nil ⇒ false
   | x :: _ ⇒ x
   end) else ( match x1 with
   | nil ⇒ false
   | x :: _ ⇒ x
   end)].
Definition LocalComp (lv:VLabel) (lpout:seq PLabel) (lpin:seq PLabel):
  distr (VLabel × seq PLabel) :=
 if (head false lpin == head false lpout) then
   Mif Flip
       (Munit ((false, (true::lv.2)), [::true]))
       (Munit ((false, (false::lv.2)), [::false]))
 else Munit ((true, lv.2), nil).
Lemma Local_total : ∀ lv lpout lpin,
 Term (LocalComp lv lpout lpin).
Definition termB (s: VState × PState) : bool :=
 (s.1 true).1 && (s.1 false).1.
Definition Fsb :=
 DPRoundLV nu false pft LocalComp termB (enum V).
Definition symBreak :=
DPRoundFixLV nu false pft LocalComp termB (enum V).
Open Local Scope U_scope.
Open Local Scope O_scope.
Definition Musb (q: VState × PState → U) :
 MF (VState × PState) -m> MF (VState × PState).
Lemma Musb_simpl : ∀ q f x,
  Musb q f x =
 if (termB x) then q x
 else if ( x.2 pft == x.2 ptf) then
  [1/4] × (f
  ([ffun y ⇒ if y then (false, true :: (x.1 true).2)
     else (false, true :: (x.1 false).2)], [ffun⇒ true])) +
  [1/4] × (f
  ([ffun y ⇒ if y then (false, false :: (x.1 true).2)
     else (false, true :: (x.1 false).2)],[ffun y ⇒ ~~ fstp y])) +
  [1/4] × (f
```

```
  ([ffun y ⇒ if y then (false, true :: (x.1 true).2)
     else (false, false :: (x.1 false).2)],[ffun y ⇒ fstp y])) +
  [1/4] × (f
  ([ffun y ⇒ if y then (false, false :: (x.1 true).2)
     else (false, false :: (x.1 false).2)], [ffun⇒ false]))
 else (f
  ([ffun y ⇒ if y then (true, (x.1 true).2) else (true, (x.1 false).2)],
   [ffun⇒ false])).
```

```
Lemma Musb_eq: ∀ (q: VState × PState → U) f l1 l2,
 mu (Fsb f (l1,l2)) q == Musb q (fun y ⇒ mu (f y) q) (l1,l2).
```

```
Lemma Sb_eq : ∀ q l,
 mu (symBreak l) q == mufix (Musb q) l.
```

```
Lemma Sb_commute : ∀ q,
    mu_muF_commute_le Fsb (fun _⇒q) (Musb q).
```

Terminaison

```
Lemma Sb_term1 n f:
      (iter (Musb (fone (VState × PState)))) n.+2
         (f, [ffun y⇒ fstp y]) == 1.
```

```
Lemma Sb_term2 n f:
      (iter (Musb (fone (VState × PState)))) n.+2
         (f, [ffun y⇒ ~~fstp y]) == 1.
```

```
Lemma Sb_term : ∀ l, ¬termB l →
 l.2 pft = l.2 ptf →
 Term (symBreak l).
```

At the end , different labels

```
Definition neq_c (l: VState) :=
 (l true).2 ≠ (l false).2.
```

```
Lemma neq_c_dec : ∀ l : VState × PState,
 {(neq_c l.1)} + {¬ (neq_c l.1)}.
```

```
Lemma Sb_breaks1: ∀ n (x:VState),
(iter (Musb (carac neq_c_dec))) n.+2
      ([ffun y ⇒ if y
                  then (false, false :: (x true).2)
                  else (false, true :: (x false).2)], [ffun y ⇒
      ~~ fstp y]) == 1.
```

```
Lemma Sb_breaks2: ∀ n (x:VState),
(iter (Musb (carac neq_c_dec))) n.+2
      ([ffun y ⇒ if y
                  then (false, true :: (x true).2)
```

```
                    else (false, false :: (x false).2)], [ffun y ⇒
      fstp y]) == 1.
```

```
Lemma Sb_breaks: ∀ (x:VState × PState),
 ¬termB x →
 x.2 pft = x.2 ptf →
 mu (symBreak x) (carac neq_c_dec)==1.
```

```
Definition lg (l: VState) := (seq.size (l true).2).
```

```
Lemma ltlg_dec : ∀ (k:nat) (l : VState × PState),
 {(k < (lg l.1))%nat} + {¬(k <(lg l.1))%nat}.
```

```
Lemma fst_simpl : ∀ (T1 T2:Type) (l1:T1) (l2:T2),
(l1 ,l2).1 = l1.
```

```
Lemma snd_simpl : ∀ (T1 T2:Type) (l1:T1) (l2:T2),
(l1 ,l2).2 = l2.
```

```
Lemma continuousFsb : continuous Fsb.
```

```
Lemma prob_ltlg01 n (x:VState×PState) l:
   (l ≤ seq.size (x.1 true).2)%nat →
  (iter (Musb (carac (ltlg_dec l))) n.+2
          ([ffun y ⇒ if y
                    then (false, false :: (x.1 true).2)
                    else (false, true :: (x.1 false).2)],
                      [ffun y⇒ ~~ fstp y]))
 == 1.
```

```
Lemma prob_ltlg02 n (x:VState × PState) l:
      (l ≤ seq.size (x.1 true).2)%nat →
(iter (Musb (carac (ltlg_dec l))) n.+2
       ([ffun y ⇒ if y
                    then (false, true :: (x.1 true).2)
                    else (false, false :: (x.1 false).2)], [ffun y ⇒
      fstp y])) == 1.
```

```
Lemma prob_ltlg0 (x:VState × PState):
 ¬termB x →
 x.2 pft = x.2 ptf →
(mu (symBreak x)) (carac (ltlg_dec (0 + seq.size (x.1 true).2))) == 1.
```

```
Lemma prob_ltlg1 (x:VState ×PState) k:
(mu
          (Mfix Fsb
             ([ffun y ⇒ if y
                         then (false, false :: (x.1 true).2)
                         else (false, true :: (x.1 false).2)],
```

```
          [ffun y ⇒ if fstp y then false else true])))
      (carac (ltlg_dec (k + (seq.size (x.1 true).2).+1))) ==
carac (ltlg_dec (k + (seq.size (x.1 true).2).+1))
     ([ffun y ⇒ if y
                then (true, false :: (x.1 true).2)
                else (true, true :: (x.1 false).2)],
     [ffun y ⇒ if fstp y then false else false]).

Lemma prob_ltlg2 (x:VState ×PState) k:
(mu
       (Mfix Fsb
          ([ffun y ⇒ if y
                     then (false, true :: (x.1 true).2)
                     else (false, false :: (x.1 false).2)],
          [ffun y ⇒ if fstp y then true else false])))
      (carac (ltlg_dec (k + (seq.size (x.1 true).2).+1))) ==
carac (ltlg_dec (k + (seq.size (x.1 true).2).+1))
     ([ffun y ⇒ if y
                then (true, true :: (x.1 true).2)
                else (true, false :: (x.1 false).2)],
     [ffun y ⇒ if fstp y then false else false]).

Lemma prob_ltlg k: ∀ (x:VState × PState),
 ¬termB x →
 x.2 pft = x.2 ptf →
mu (symBreak x)
 (carac (ltlg_dec (k+ (seq.size (x.1 true).2))%nat)) == [1/2]^k.

Lemma sumgHalf :
 islub (sumg [1/2]) 2.

Lemma expectancySb: ∀ (x:VState × PState),
 ¬termB x →
 x.2 pft = x.2 ptf →
(islub (Rpsigma (fun k ⇒ mu (symBreak x)
 (carac (ltlg_dec (k+ (seq.size (x.1 true).2))%nat))))
  2).

End symBreak.
```

Chapter 20

Library handshake_spec

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun tuple.
Require Import Ensembles.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import setSem.
Require Import rdaTool_gen.
Set Implicit Arguments.
Import Prenex Implicits.
```

20.1 Introduction

This files describes the specification of any handshake problem

```
Section hsSpec.
```

20.2 Specification of the handshake problem in a global view

V: set of vertices. Adj: edge relation NG: undirected non-loop graph.

```
Context `(NG: NGraph V Adj).
```

synch: type of function describing the other vertex with which a vertex is in handshake

```
Definition synch := V → option V.
```

synchAdj s: each pairs in a handshake are adjacent

```
Definition synchAdj (s: synch) := ∀ (v:V),
 match (s v) with
  |Some w ⇒ Adj v w
  |_ ⇒ true
 end.
```

synchSym s: each member of a pairs in a handshake are in a handshake with the other

```
Definition synchSym (s: synch) := ∀ (v:V),
 match (s v) with
  |Some w ⇒ (s w) == Some v
  |_ ⇒ true
 end.
```

partialMatching s: s is adj and sym `Definition matching (s: synch) :=`
`(synchAdj s) ∧ (synchSym s).`

hsBetween s v w: there is a handshake between v and w

```
Definition hsBetween (s: synch) (v w:V) :=
 (Adj v w) && (s v == Some w) && (s w == Some v).
```

hsExists, there exists v and w such that they are in a handshake

```
Definition hsExists (s: synch) := ∃ v w,
 hsBetween s v w.
```

```
End hsSpec.
```

20.3 In a common graph: Definitions

```
Section commonGraph.
```

V: set of vertices of the graph. Adj: edge relation of the graph. NG: undirected non-loop graph. VLabel: type of labels on vertices. LLabel: type of labels on ports. vunit, lunit: default values. LR lv slp: local computation from a local view.

```
Context ‘(NG: NGraph V Adj).

Variables (VLabel: eqType) (PLabel:eqType).

Variables (vunit:VLabel) (lunit:PLabel).

Variable LR : seq(VLabel → (seq PLabel) ->(seq PLabel) →
  gen (VLabel × seq PLabel)).

Let VSt := LabelFunc V VLabel.
Let PSt := LabelFunc (@port_finType V Adj) PLabel.

Variable p0 : (@port_finType V Adj).

Variable (nu:V → seq V).
```

```
Hypothesis Hnu : ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).
Definition UniformView (s: VSt× PSt) :=
 ∀ v w, seq.size (nu v) = seq.size (nu w) →
              (Vread s.1 v)= (Vread s.1 w) ∧
              (Pinread nu p0 s.2 v) = (Pinread nu p0 s.2 w) ∧
              (Poutread nu p0 s.2 v) = (Poutread nu p0 s.2 w).
Definition Uniform (s: VSt× PSt) :=
 (∀ v1 v2, (s.1 v1)= (s.1 v2)) ∧
 (∀ p1 p2, (s.2 p1) = (s.2 p2)).
Lemma uniformUniformView : ∀ (s: VSt× PSt),
Uniform s → UniformView s.
```

nextState sigma pSeq: a round over sigma with the pSeq order

```
Definition nextState (sV: seq V)(sigma:VSt× PSt):=
GPStep nu lunit p0 LR sV sigma.
```

We assume there is a function, hsPort, wich tells from a local view if there is a handshake for a vertex v and on which port

```
Variable hsPort : VLabel → seq PLabel → seq PLabel → option nat.
Definition hsPortR (sigma: VSt ×PSt) (v:V) :=
 (hsPort (Vread sigma.1 v) (Poutread nu p0 sigma.2 v) (Pinread nu p0 sigma.2 v)) .
Hypothesis hsp1 : ∀ (sigma: VSt ×PSt) (v:V) i,
  (hsPortR sigma v) = Some i →
  i < (deg Gr v).
```

assNeigh v: returns None if v is not in handshake or Some w, if w is in handshake with w

```
Definition assNeigh (v: V) (sigma:VSt × PSt) : (option V) :=
 match (hsPortR sigma v) with
  |Some i ⇒ Some (nth v (nu v) i)
  |_ ⇒ None
 end.
```

consistent: hsPort is symmetrical

```
Definition consistent (sigma: VSt × PSt) :=
 synchSym (fun v ⇒ assNeigh v sigma).
```

hsEventually: there exists a state where there is at least one handshake and which is reachable from the initState `Definition hsEventually initS sV :=`

```
 ∃ sigma,
    reachFrom _ (nextState sV) initS sigma ∧
     (@hsExists _ Adj (fun v ⇒ assNeigh v sigma )).
```

Lemmas Lemma assNeigh1 : ∀ v w s,
uniq (nu v) → size (nu v) = deg Gr v →
assNeigh v s = Some w →
hsPortR s v = Some (index w (nu v)).

End commonGraph.

Section specAlg.

V: set of vertices of the graph. Adj: edge relation of the graph. NG: undirected non-loop graph. VLabel: type of labels on vertices. LLabel: type of labels on ports. vunit, lunit: default values. LR lv slp: local computation from a local view.

Variables ($VLabel$: eqType) ($PLabel$:eqType).

Variables ($vl0$:$VLabel$) ($pl0$:$PLabel$).

Let $State$ (V:finType) (Adj:rel V) := **Datatypes.prod** (LabelFunc V $VLabel$)
(LabelFunc (@port_finType V Adj) $PLabel$).

Let pfT (V:finType) (Adj:rel V) := (@port_finType V Adj).

Record **hsAlgo** :=
{
Local rules HsR : seq($VLabel$ → (seq $PLabel$) ->(seq $PLabel$) →
gen ($VLabel$ × seq $PLabel$));

Handshake function HsP : $VLabel$ → seq $PLabel$ → seq $PLabel$ → **option nat**;

Initial state HsI : ∀ (V:finType) (Adj: rel V)(Gr:**Graph** Adj)(NG: **NGraph** Gr), $State$ Adj;

HsI1: ∀ (V:finType) (Adj: rel V) (Gr:**Graph** Adj)(NG: **NGraph** Gr) (nu:V → seq V)
(Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) ($Hnu2$: ∀ (v :V), uniq (nu v))
($p0$: pfT Adj),
consistent $p0$ nu HsP (HsI NG);

HsI2 : ∀ (V:finType) (Adj: rel V) (Gr:**Graph** Adj) (NG: **NGraph** Gr)(nu:V → seq V)
(Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) ($Hnu2$: ∀ (v :V), uniq (nu v)) ,
Uniform (HsI NG);

HsP1 : ∀ (V:finType) (Adj: rel V) (Gr:**Graph** Adj) (NG: **NGraph** Gr)(nu:V → seq V)
(Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) ($Hnu2$: ∀ (v :V), uniq (nu v))
($p0$: pfT Adj) (s:$State$ Adj) (v:V) (i:**nat**),
(hsPortR $p0$ nu HsP s v) = Some i → i < (deg Gr v);

HsRind: ∀ (V:finType) (Adj: rel V) (Gr:**Graph** Adj) (NG: **NGraph** Gr)(nu:V → seq V)
(Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) ($Hnu2$: ∀ (v :V), uniq (nu v))
($p0$: pfT Adj),
Stable _ (fun s ⇒ consistent $p0$ nu HsP s) (nextState $pl0$ HsR $p0$ nu (enum V))
}.

Definition hsRealisation (A: **hsAlgo**) :=
∀ (V:finType) (Adj: rel V) (Gr:**Graph** Adj)(NG: **NGraph** Gr)(nu:V → seq V)
(Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) ($Hnu2$: ∀ (v :V), uniq (nu v))
($p0$: pfT Adj),
hsEventually $pl0$ (HsR A) $p0$ nu (HsP A) (HsI A NG) (enum V).

Variable A: **hsAlgo**.
Hypothesis Aok : hsRealisation A.

Fixpoint Adet (l:seq($VLabel$ → (seq $PLabel$) ->(seq $PLabel$) →
gen ($VLabel$ × seq $PLabel$))) :=
match l with
|nil ⇒ **True**
|t : : q ⇒ (∀ lv $lp1$ $lp2$, Deterministic (t lv $lp1$ $lp2$)) ∧ (Adet q)
end.

End specAlg.

Chapter 21

Library handshake_det

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
```

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun tuple.
```

```
Require Import Ensembles.
```

```
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import setSem.
Require Import rdaTool_gen.
Require Import handshake_spec.
```

```
Set Implicit Arguments.
Import Prenex Implicits.
```

21.1 Introduction

This files describes the proof of the following lemma: there is no deterministic algorithm which solves the handshake problem

```
Section Witness.
```

21.2 Description of the witness graph

```
Definition Vw : finType := ordinal_finType 3.
```

```
Definition Adjw : rel Vw := (fun x y => x != y).
```

```
Context `(wNG: NGraph Vw Adjw).
```

```
Lemma Hv0 : 0 < 3.

Lemma Hv1 : 1 < 3.

Lemma Hv2 : 2 < 3.

Definition v0 : Vw := (Ordinal Hv0).
Definition v1 : Vw := (Ordinal Hv1).
Definition v2 : Vw := (Ordinal Hv2).

Definition sVw := ([::v0;v1;v2] ).

Definition nuw (v:Vw) :=
 match (val v) with
  |0 ⇒ (v1::v2::nil)
  |1 ⇒ (v2::v0::nil)
  |2 ⇒ (v0::v1::nil)
  |_ ⇒ nil
 end.
```

Lemmas `Lemma enumV1 : (enum Vw) = ord_enum 3.`

```
Lemma enumV : (enum Vw) = ([::v0;v1;v2] ).

Lemma Gind : ∀ (P:Vw→Prop),
 P v0 → P v1 → P v2 →
 ∀ v, P v.

Lemma v012 : ∀ v, v == v0 ∨ v == v1 ∨ v == v2.

Lemma v210 : ∀ v, v0 == v ∨ v1 == v ∨ v2 == v.

Lemma degree_2 : ∀ v, deg Gr v = 2.

Lemma nu1 : ∀ v, uniq (nuw v).

Lemma nu2 : ∀ v, size (nuw v) = (deg Gr v).

Lemma nu3 : ∀ v w, Adjw v w = (w \in nuw v).
```

21.3 No algorithm

```
Variables (VLabel: eqType) (PLabel:eqType).

Variables (vunit:VLabel) (lunit:PLabel).

Let VSt := LabelFunc Vw VLabel.
Let PSt := LabelFunc (@port_finType Vw Adjw) PLabel.

Variable p0 : (@port_finType Vw Adjw).

Variable A : (hsAlgo VLabel lunit).
```

**Induction step `Section Ind.`

```
Variable Msigma: gen (VSt × PSt).
```

```
Hypothesis Hsigma1 : ∀ s, In _ (Setsem Msigma) s → UniformView p0 nuw s.
Hypothesis Hsigma2 : ∀ s, In _ (Setsem Msigma) s → consistent p0 nuw (HsP A) s.
Let HsP0 s v := (@HsP1 _ _ _ A _ _ _ wNG _ nu3 nu1 p0 s v).
Lemma nohs00 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v0 sigma != Some v0.
Lemma nohs10 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v1 sigma != Some v0.
Lemma nohs20 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v2 sigma != Some v0.
Lemma nohs01 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v0 sigma != Some v1.
Lemma nohs11 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v1 sigma != Some v1.
Lemma nohs21 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v2 sigma != Some v1.
Lemma nohs02 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v0 sigma != Some v2.
Lemma nohs12 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v1 sigma != Some v2.
Lemma nohs22 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 assNeigh p0 nuw (HsP A) v2 sigma != Some v2.
Lemma nohs3 (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
∀ v w,
 assNeigh p0 nuw (HsP A) v sigma != Some w.
Lemma nohs (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 ∀ v,
 assNeigh p0 nuw (HsP A) v sigma = None.
Lemma NoHs (sigma:VSt×PSt) (Hsig:In _ (Setsem Msigma) sigma) :
 ~(@hsExists _ Adjw (fun v ⇒ assNeigh p0 nuw (HsP A) v sigma)).
Lemma Unif_aux1 : ∀ (y:VSt×PSt) (y' : VSt) k,
y' = update [set v0]
            (update [set v1]
               (update [set v2] y.1 (Vwrite k.1 v2),
               update (WriteArea v2) y.2 (Pwrite nuw lunit k.2 v2)).1
               (Vwrite k.1 v1),
            update (WriteArea v1)
              (update [set v2] y.1 (Vwrite k.1 v2),
              update (WriteArea v2) y.2 (Pwrite nuw lunit k.2 v2)).2
```

```
                (Pwrite nuw lunit k.2 v1)).1 (Vwrite k.1 v0) →
(∀ v w:Vw, Vread y.1 v = Vread y.1 w) →
∀ v w : Vw, Vread y' v = Vread y' w.

Lemma Unif_aux2 : ∀ (y:VSt×PSt) (y' : PSt) k,
y' = update (WriteArea v0)
            (update [set v1]
                (update [set v2] y.1 (Vwrite k.1 v2),
                update (WriteArea v2) y.2 (Pwrite nuw lunit k.2 v2)).1
                (Vwrite k.1 v1),
            update (WriteArea v1)
               (update [set v2] y.1 (Vwrite k.1 v2),
               update (WriteArea v2) y.2 (Pwrite nuw lunit k.2 v2)).2
               (Pwrite nuw lunit k.2 v1)).2 (Pwrite nuw lunit k.2 v0) →
(∀ v w:Vw, Poutread nuw p0 y.2 v = Poutread nuw p0 y.2 w) →
 ∀ v w, Poutread nuw p0 y' v = Poutread nuw p0 y' w.

Lemma Unif_aux3 : ∀ (y:VSt×PSt) (y' : PSt) k,
y' = update (WriteArea v0)
            (update [set v1]
                (update [set v2] y.1 (Vwrite k.1 v2),
                update (WriteArea v2) y.2 (Pwrite nuw lunit k.2 v2)).1
                (Vwrite k.1 v1),
            update (WriteArea v1)
               (update [set v2] y.1 (Vwrite k.1 v2),
               update (WriteArea v2) y.2 (Pwrite nuw lunit k.2 v2)).2
               (Pwrite nuw lunit k.2 v1)).2 (Pwrite nuw lunit k.2 v0) →
(∀ v w:Vw, Pinread nuw p0 y.2 v = Pinread nuw p0 y.2 w) →
 ∀ v w, Pinread nuw p0 y' v = Pinread nuw p0 y' w.

Lemma UniformViewStablehs : Adet (HsR A) →
 ∀ s',
 In _ (Setsem (Gbind _ _ Msigma
     (fun x ⇒ nextState lunit (HsR A) p0 nuw sVw x))) s' →
 UniformView p0 nuw s'.

End Ind.

Lemma NotReal : Adet (HsR A) →
 ¬ (hsRealisation A).
Print reachInd.

Qed.

End Witness.
```

Chapter 22

Library handshake_gen

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun tuple.
Require Import Ensembles.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import setSem.
Require Import rdaTool_gen.
Set Implicit Arguments.
Import Prenex Implicits.
```

22.1 Introduction

The handshake algorithm is the following: each vertex v chooses a neighbour c(v) v sends 1 to c(v) and 0 to its other neighbour if v receives 1 from c(v) there is a handshake.

The message passing is simulated by a labelling on the ports If v has chosen c(v), the port p(v,c(v)) is relabelled 1.

```
Section genAlgo.
Context `(NG: NGraph V Adj).
Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).
Let Pt := (@port_finType V Adj).
Variable p0 : Pt.
```

Let $VLabel$: eqType := option_eqType nat_eqType.
Let $PLabel$: eqType := bool_eqType.

Let $VState$:= LabelFunc V $VLabel$.
Let $PState$:= LabelFunc Pt $PLabel$.

22.2 Auxiliairy functions

numberNeigh lpin: number of neighbours according a local view

```
Definition numberNeigh (lpin: seq PLabel) : nat :=
 size lpin.
```

rand_sendChosen k lpin : the sequence of size lpin composed of false elements except the kth wichi is true

```
Fixpoint rand_sendChosen (k:nat) (lpin: seq PLabel) : seq PLabel :=
 match lpin with
  |t::q => match k with
            |0 => (false::(rand_sendChosen 0 q))
            |1 => (true::(rand_sendChosen 0 q))
            |S k' => (false::(rand_sendChosen k' q))
            end
  | nil => nil
 end.
```

```
Lemma rand_sendChosen_size : ∀ l i,
 size (rand_sendChosen i l) = size l.
```

```
Lemma rand_sendChosen_count : ∀ (k : nat) (lpin : seq bool_eqType),
     count id (rand_sendChosen k.+1 lpin) ≤ 1.
```

```
Lemma rand_sendChosen_index : ∀ (k : nat) (lpin : seq bool_eqType),
k < seq.size lpin →
index true (rand_sendChosen k.+1 lpin) = k.
```

```
Lemma rand_sendChosen_index2 : ∀ (k : nat) (lpin : seq bool_eqType),
 seq.size lpin ≤ k→
index true (rand_sendChosen k.+1 lpin) = seq.size lpin.
```

```
Lemma rand_sendChosenlpin : ∀ lpin1 lpin2 n,
seq.size lpin1 = seq.size lpin2 →
rand_sendChosen n lpin1 = rand_sendChosen n lpin2.
```

```
Lemma rand_sendChosen0 : ∀ l,
 rand_sendChosen 0 l = nseq (size l) false.
```

```
Lemma rand_sendChosen_nth1 : ∀ (V0:finType) lp (w:V0) l,
 size l = size lp →
 size l ≠ 0 →
 w \in l →
```

nth false (rand_sendChosen (index w l).+1 lp)(index w l).

Lemma rand_sendChosen_nth2 : ∀ ($V0$:finType) lp (v w:$V0$) l,
size l = size lp →
size $l \neq 0$ →
(v == w)=false →
nth false (rand_sendChosen (index w l).+1 lp)(index v l) = false.

agreed lpout lpin : returns true if ith element of lpin is true where i is the index of the first element at true in lpout else returns false

```
Fixpoint agreed (lpout:seq PLabel) (lpin:seq PLabel) : bool :=
 match lpout,lpin with
  |true::q, true::q' ⇒ true
  |true::q, false::q' ⇒ false
  |false::q, _::q' ⇒ agreed q q'
  |_ , _ ⇒ false
 end.
```

Lemma agreed_1 v : ∀ (y:$VState$×$PState$),
agreed (Poutread nu $p0$ y.2 v) (Pinread nu $p0$ y.2 v) = true →
true \in (Poutread nu $p0$ y.2 v) .

Lemma agreed_2 v : ∀ (y:$VState$×$PState$) w i,
nth v (nu v) i = w→
count id (Poutread nu $p0$ y.2 w) ≤ 1 →
index true (Poutread nu $p0$ y.2 v) = i →
agreed (Poutread nu $p0$ y.2 v) (Pinread nu $p0$ y.2 v) = true →
v \in (nu w) →
agreed (Poutread nu $p0$ y.2 w) (Pinread nu $p0$ y.2 w) = true.

Lemma agreed_3 v : ∀ (y:$VState$×$PState$) w i j,
agreed (Poutread nu $p0$ y.2 v) (Pinread nu $p0$ y.2 v) = true →
index true (Poutread nu $p0$ y.2 v) = i → nth v (nu v) i = w →
nth w (nu w) j = v → j < deg Gr w →
count id (Poutread nu $p0$ y.2 w) ≤ 1 →
index true (Poutread nu $p0$ y.2 w) = j.

Lemma agreed_4 u : ∀ (x:$VState$×$PState$) v,
Adj v u →
index v (nu u) = index true (Poutread nu $p0$ x.2 u) →
index u (nu v) = index true (Poutread nu $p0$ x.2 v) →
agreed (Poutread nu $p0$ x.2 u) (Pinread nu $p0$ x.2 u).

22.3 Local algorithm

Definition randHSLoc (lv:$VLabel$) ($lpout$ $lpin$: seq $PLabel$) : **gen** ($VLabel$ ×seq $PLabel$) :=

```
match (numberNeigh lpin) with
    |O ⇒ Greturn _ (None, nil)
    |S n ⇒ Grandom _ n
        (fun k ⇒ Greturn _ (None,rand_sendChosen k.+1 lpin))
  end.
```

22.4 Global algorithm

```
Definition randHSRound (seqV: seq V) (res: VState × PState):=
 GPRound nu false p0 seqV res randHSLoc.

End genAlgo.
```

Chapter 23

Library handshake_op

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".

Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.

Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import op.
Require Import rdaTool_op.
Require Import handshake_gen.

Set Implicit Arguments.
Import Prenex Implicits.
```

23.1 Simulation of handshake algorithm

```
Section HS.

Variable (rand_t : Type)(get : nat → rand_t → nat × rand_t).
Context (rand : ORandom _ get).

Let VLabel : eqType := option_eqType nat_eqType.
Let PLabel : eqType := bool_eqType.

Definition OHSLoc (lv:VLabel) (lpout lpin: seq PLabel)
 : Op rand_t (VLabel ×seq PLabel) :=
  match (numberNeigh lpin) with
   |O ⇒ Oreturn (None, nil)
   |S n ⇒ Obind (Orandom n rand)
      (fun k ⇒ Oreturn (None,rand_sendChosen k.+1 lpin))
```

```
  end.
Context `(NG: NGraph V Adj).
Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).
Let Pt := (@port_finType V Adj).
Variable p0 : Pt.
Let VState := LabelFunc V VLabel.
Let PState := LabelFunc Pt PLabel.
Definition OHSRound (seqV: seq V)(res: VState × PState) :=
 OPRound nu false p0 seqV res OHSLoc.
Section gen.
Lemma OPGHS_eq1 : ∀ (lv:VLabel) (lp1 lp2: seq PLabel) ,
 Opsem _ get rand (randHSLoc lv lp1 lp2) =
 OHSLoc lv lp1 lp2.
Lemma OPGHS_eq2 : ∀ (seqV: seq V) (res:VState×PState),
 Opsem _ get rand (randHSRound nu p0 seqV res) =
 OHSRound seqV res.
End gen.
Section simulation.
Definition OHSRoundF (seqV: seq V) (res: (V → VLabel) × (V × V → PLabel)) :=
 OPFRound nu false seqV res OHSLoc.
Lemma OHSF_eq1 : ∀ (seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v n,
 seqV = seqVF →
 (∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   ((OHSRound seqV res n).1).1 v =
   ((OHSRoundF seqVF resF n).1).1 v.
Lemma OHSF_eq2 : ∀ (seqV seqVF : seq V)(res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v w n,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
 Adj v w →
   ((OHSRound seqV res n).1).2 (VtoP v w p0) =
   ((OHSRoundF seqVF resF n).1).2 (v,w).
 Lemma OHSF_eq3 : ∀ (seqV seqVF : seq V) (res: VState×PState)
```

```
 (resF : (V → VLabel) × (V × V → PLabel) ) n,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   (OHSRound seqV res n).2 =
   (OHSRoundF seqVF resF n).2.
End simulation.

End HS.

Section simulation.
```

Definition of the graph

```
Inductive V : Type :=
 |v0 : V
 |v1 : V
 |v2 : V
 |v3 : V.

Definition eqV := (fun x y : V ⇒
 match x,y with
|v0,v0 ⇒ true
|v1,v1 ⇒ true
|v2,v2⇒true
|v3,v3 ⇒ true
|_,_ ⇒ false
end).

Lemma eqVP : Equality.axiom eqV.

Canonical V_eqMixin := EqMixin eqVP.
Canonical V_eqType:= Eval hnf in EqType V V_eqMixin.

Lemma V_pickleK : pcancel (fun v : V ⇒ match v with |v0 ⇒ O |v1 ⇒ 1%nat |v2 ⇒
2 |v3 ⇒ 3 end)
 (fun x : nat ⇒ match x with |0 ⇒ Some v0 | 1 ⇒ Some v1 |2 ⇒ Some v2 | 3 ⇒ Some v3
| _ ⇒ None end).

Fact V_choiceMixin : choiceMixin V.

Canonical V_choiceType := Eval hnf in ChoiceType V V_choiceMixin.

Definition V_countMixin := CountMixin V_pickleK.
Canonical V_countType := Eval hnf in CountType V V_countMixin.

Definition venum := (v0:: v1:: v2:: v3:: nil).
Lemma V_enumP : Finite.axiom venum.

Definition V_finMixin := Eval hnf in FinMixin V_enumP.
Canonical V_finType := Eval hnf in FinType V V_finMixin.
```

```
Lemma card_V : #|{: V}| = 4.

Definition Adj : rel V := (fun x y => match x, y with
 |v0,v1 |v0,v3 |v1,v0 |v1,v2 |v1,v3 |v2,v1 |v2,v3 |v3,v0 |v3,v1 |v3,v2 => true
 | _,_ => false
 end).

Lemma AdjSym : symmetric Adj.

Lemma AdjIrrefl : irreflexive Adj.

Lemma enumV : (enum V_finType) = ([::v0;v1;v2;v3] ).

Context `(NG: NGraph V_finType Adj).

Lemma Nb_enumv0 : Nb_enum Gr v0 = (v1::v3::nil).

Lemma degv0 : (deg Gr v0) = 2.

Definition nu (v: V) : seq V :=
 match v with
   |v0 => [::v1;v3]
   |v1 => [::v0;v2;v3]
   |v2 => [::v1;v3]
   |v3 => [::v1;v2;v0]
end.

Lemma nuAdj_eq : ∀ u w,
Adj u w = (w \in nu u).

Lemma hp0 : Adj (v0,v1).1 (v0,v1).2.

Definition p0 := Port hp0.
```

Definition of the labelling

```
Let VLabel : eqType := option_eqType nat_eqType.
Let PLabel : eqType := bool_eqType.

Definition initV : (LabelFunc V_finType VLabel) :=
finfun (fun x:V => None).

Definition initP : (LabelFunc (@port_finType V_finType Adj) PLabel) :=
 finfun (fun x => true).

Definition init := (initV, initP).

Definition initVF : (V -> VLabel) :=
 (fun x:V => None).

Definition initPF : ((V×V) -> PLabel) :=
 (fun x => true).

Definition initF := (initVF, initPF).

Lemma init_eq1 : ∀ v, init.1 v = initF.1 v.

Lemma init_eq2 : ∀ v w,
```

```
  Adj v w → init.2 (VtoP v w p0) = initF.2 (v,w).
```

Equivalence

```
Lemma OHSF_eq4 : ∀ v n,
   ((OHSRound my_gen nu p0 (enum V_finType) init n).1).1 v =
   ((OHSRoundF my_gen nu [::v0;v1;v2;v3] initF n).1).1 v.

Lemma OHSF_eq5 : ∀ v w n,
  Adj v w →
   ((OHSRound my_gen nu p0 (enum V_finType) init n).1).2 (VtoP v w p0) =
   ((OHSRoundF my_gen nu [::v0;v1;v2;v3] initF n).1).2 (v,w).

 Lemma OHSF_eq6 : ∀ n,
   (OHSRound my_gen nu p0 (enum V_finType) init n).2 =
   (OHSRoundF my_gen nu [::v0;v1;v2;v3] initF n).2.
```

Computation

```
Let R1 := (OHSRoundF my_gen nu [::v0;v1;v2;v3] initF) 6.

Check (R1).

Eval vm_compute in (R1.1.1 v3).
Eval vm_compute in (R1.1.2 (v3,v1)).
Eval vm_compute in (R1.1.2 (v3,v2)).
Eval vm_compute in (R1.1.2 (v3,v0)).

Eval vm_compute in (R1.1.1 v0).
Eval vm_compute in (R1.1.2 (v0,v1)).
Eval vm_compute in (R1.1.2 (v0,v3)).
Eval vm_compute in (R1.1.2 (v0,v0)).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] R1.1).

End simulation.
```

Chapter 24

Library handshake_dist

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat.
Require Import fintype finset fingraph seq finfun bigop choice tuple.
Import Prenex Implicits.

Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".

Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Require Import Rplus.
Require Import my_alea.
Require Import my_ssr.
Require Import my_ssralea.
Require Import graph.
Require Import graph_alea.
Require Import labelling.
Require Import bfs.
Require Import gen.
Require Import dist.
Require Import rdaTool_gen.
Require Import rdaTool_dist.
Require Import handshake_gen.

Set Implicit Arguments.

Open Local Scope U_scope.
Open Local Scope O_scope.

Section Handshake.
```

24.1 The graph

```
Context `(NG: NGraph V Adj).
Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).
Definition E := (@edge_finType V Adj).
Variable e0:E.
Definition Pt := (@port_finType V Adj).
Definition p0 := (EtoP1 e0).
Definition VLab : eqType := option_eqType nat_eqType.
Definition PLab : eqType := bool_eqType.
Definition VSt := LabelFunc V VLab.
Definition PSt := LabelFunc Pt PLab.
```

24.2 Local Algorithm

```
Definition DHSLoc (lv:VLab) (lpout lpin: seq PLab)
 : distr (VLab ×seq PLab) :=
  match (numberNeigh lpin) with
   |O ⇒ Munit (None, nil)
   |S n ⇒ Mlet (Random n)
      (fun k ⇒ Munit (None,rand_sendChosen k.+1 lpin))
  end.
Section gen.
```

24.2.1 Proofs of the equivalence with the generic algorithm

```
Lemma DPGHS_eq1 : ∀ (lv:VLab) (lp1 lp2: seq PLab) ,
 Distsem (randHSLoc lv lp1 lp2) =
 DHSLoc lv lp1 lp2.
End gen.
```

24.2.2 Local Analysis

DHSLoc can be decomposed in a sum of computations around each port

```
Lemma is_discrete_DHSLoc : ∀ (lv:VLab) (lpout:seq PLab)
 (lpin:seq PLab),
  is_discrete_s (DHSLoc lv lpout lpin).
```

DHSLoc terminates

```
Lemma DHSLoc_total : ∀ (lv:VLab) (lpout:seq PLab) (lpin:seq PLab),
 Term (DHSLoc lv lpout lpin).
```

The probability for a vertex to choose the ith neighbour is 1/(deg v)
carac_lc_eq returns true if i is equal to the choice of v i.e. it returns true if v chooses its ith neighbour else false

```
Definition carac_lc_eq : nat → seq PLab → VLab × seq PLab → U :=
 fun (i: nat) (lpin:seq PLab) (s: VLab × seq PLab) ⇒
  B2U (i == (index true s.2)).

Lemma DHSLoc_eq : ∀ (lv:VLab)(lpout lpin:seq PLab)(k: nat),
 (k < seq.size lpin)%nat →
 (mu (DHSLoc lv lpout lpin)) (carac_lc_eq k lpin) ==
 [1/]1+((seq.size lpin).-1).
```

24.3 Global Algorithm

DHS seqV res : at the end of the algorithm DHS, each vertices in seqV has made a choice among its neighbours and has updated its choice in res `Definition DHS (seqV: seq V) (res: VSt × PSt): distr (VSt × PSt) :=`
` DPRound nu false p0 seqV res DHSLoc.`

```
Section genRound.
```

24.3.1 Proofs of the equivalence with the generic algorithm

```
Lemma DPGHS_eq2 : ∀ (seqV: seq V) (res:VSt×PSt),
 Distsem (randHSRound nu p0 seqV res) =
 DHS seqV res.

End genRound.
```

24.3.2 Analysis

Termination

DHS terminates whichever the sequence of vertices on which DHS is applied

```
Lemma DHS_total : ∀(s: seq V) (res: VSt×PSt),
 Term (DHS s res).
```

Probability to choose a neighbour, local view

The probability for a vertex to choose the ith neighbour (i.e. ith neighbour is labelled true) is 1/(deg v) carac_hs_eqNat returns true if i is equal to the local choice of v extracted from the global labelling function i.e. it returns true if v chooses its ith neighbour else false

```
Definition carac_hs_eqNat : V → seq PLab → nat → VSt × PSt → U :=
 fun (v:V) (lpin:seq PLab) (i: nat) (s: VSt × PSt) ⇒
 B2U (i ==
 index true (Poutread nu p0 s.2 v)).
```

```
Lemma DHS_degv_aux1 : ∀ (v:V) (i:nat) (lpin:seq PLab) (y:VLab × seq PLab)
(sn:VSt × PSt),
 seq.size y.2 = seq.size (nu v) →
 carac_lc_eq i lpin y ==
 carac_hs_eqNat v lpin i (VPupdate nu false v y sn).
```

```
Lemma DHS_size1 : ∀ a b c,
 seq.size b = seq.size c →
 (mu (DHSLoc a b c)) (fun x ⇒ B2U(seq.size x.2 != seq.size c)) == 0.
```

```
Lemma DHSLtac1 a b c d f:
 seq.size b = seq.size c →
 (mu (DHSLoc a b c)) (fplus f
       (fun x ⇒ B2U(seq.size x.2 != seq.size c))) == d →
 (mu (DHSLoc a b c)) f == d.
```

```
Lemma DHS_degv_aux2 : ∀ (v:V) (x:VLab×seq PLab) (s:VSt×PSt)(i:nat),
  seq.size x.2 = seq.size (nu v) →
  (mu (DHS (seq.rem v (enum V)) s))
        (fun x0 : LabelFunc V VLab × LabelFunc port_finType bool_eqType ⇒
         carac_hs_eqNat v (Pinread nu p0 s.2 v) i (VPupdate nu false v x x0)) ==
         carac_hs_eqNat v (Pinread nu p0 s.2 v) i (VPupdate nu false v x s).
```

```
Lemma DHS_degv_local : ∀ (v:V)(i:nat)(s:VSt×PSt),
 (i < (deg Gr v))%nat →
 (mu (DHS (enum V) s)) (carac_hs_eqNat v (Pinread nu p0 s.2 v) i) ==
 [1/]1+((deg Gr v).-1).
```

Probability to choose a neighbour, global view

The probability for a vertex to choose the vertex w which is a neighbour is 1/(deg v)

carac_hs_eqV returns true if v chooses w else false

```
Definition hs_eqVB (v w:V) (s:VSt×PSt) :=
 index w (nu v) ==
 index true (Poutread nu p0 s.2 v).
```

```
Definition carac_hs_eqV : V → V → VSt×PSt → VSt×PSt →U :=
 fun (v w: V) (inits s:VSt×PSt) ⇒
   B2U (hs_eqVB v w s).
```

```
Lemma carac_hs_iff : ∀ (v w: V) (inits:VSt ×PSt) (i:nat),
 index w (nu v) = i →
 carac_hs_eqV v w inits ==
 carac_hs_eqNat v (Pinread nu p0 inits.2 v) i.
```

```
Lemma DHS_degv_global : ∀ (v w: V) (s:VSt×PSt),
 Adj v w →
 (mu (DHS (enum V) s)) (carac_hs_eqV v w s) == [1/]1+((deg Gr v).-1).
```

Probability of having a handshake on an edge

The probability for an edge (v,w) having a handshake on it is 1/(deg v * deg w)

carac_hs_edge returns true if v chooses w and w chooses v else false

```
Definition hs_edgeB (e:E) (s:VSt×PSt) : bool :=
(hs_eqVB (fste e) (snde e) s) && (hs_eqVB (snde e) (fste e) s).
```

```
Definition carac_hs_edge : E → VSt×PSt → U :=
 fun (e:E) ⇒
  fB2U (fun (s:VSt×PSt) ⇒ hs_edgeB e s).
```

carac_hs_edge returns true if v chooses w, w chooses v and v and w are in the connex composant of the edge eth else false

```
Definition eth :=
 nth e0 (enum E) 0.
```

```
Definition carac_hs_edge0 : E → VSt×PSt → U :=
 fun (e:E) ⇒
  fB2U (fun (s:VSt×PSt) ⇒ hs_edgeB e s &&
 connect (fun v w ⇒ Adj v w) (fste eth) (fste e)).
```

```
Lemma indepbDHS_hs : ∀ (e:E) (inits:VSt×PSt),
 indepb (DHS (enum V) inits)
     (hs_eqVB (fste e) (snde e))
     (hs_eqVB (snde e) (fste e)).
```

```
Lemma DHS_dege : ∀ (e:E) (s:VSt×PSt),
 mu (DHS (enum V) s) (carac_hs_edge e) ==
   [1/]1+((deg Gr (fste e)).-1) ×
   [1/]1+((deg Gr (snde e)).-1).
```

Probability for having at least one vertex

```
Require Import Rplus.
```

hs_glob s returns true if there is a handshake in the graph else false

```
Definition hs_glob_ex (s:VSt×PSt) : bool :=
  [∃ x, hs_edgeB x s].
```

```
Definition hs_glob_ex0 (s:VSt×PSt) : bool :=
  [∃ x, hs_edgeB x s &&
   connect (fun x y ⇒ Adj x y ) (fste eth) (fste x) ].
```

carac_hs_glob s returns 1 if there is a handshake in the graph else 0

```
Definition carac_hs_glob_ex : VSt×PSt → U :=
 fB2U (fun (s:VSt×PSt) ⇒ hs_glob_ex s).

Definition carac_hs_glob_ex0 : VSt×PSt → U :=
 fB2U (fun (s:VSt×PSt) ⇒ hs_glob_ex0 s).

Definition hscte := prod (fun _ ⇒ [1-] ([1/2] × [1/]1+(#|E|.-1))) #|E|.
```

Rpsigma_hs

```
Lemma conncount1 : ∀ w v,
   connect (fun v0 : V ⇒ [eta Adj v0]) v w →
   (deg Gr w ≤
     (count (connect (fun v0 : V ⇒ [eta Adj v0]) v) (enum V)).-1)%coq_nat.

Lemma conncount2 : ∀ v,
 connect (fun v : V ⇒ [eta Adj v]) (fste eth) v →
count (fun i : V ⇒Adj v i && connect (fun v0 : V ⇒ [eta Adj v0]) (fste eth) i) (enum
V)
 = deg Gr v.

Lemma Rpsigma_hs : ∀ (res:VSt×PSt),
 U2Rp([1/2]) ≤
   (Rpsigma (fun k : nat ⇒
        (mu (DHS (enum V) res)) (carac_hs_edge0 (nth e0 (enum E) k))))
    #|E|.
```

hs1

```
Definition parentFunc (k:nat) := (@tF _ Adj (fste eth) #| V|).

Definition choiceFunc (k:nat) : {ffun V → V} :=
 finfun (fun x ⇒ match parentFunc k x with
                   |Some y ⇒ y
                   |None ⇒ snde eth
                 end).

Definition coverTree (k: nat) : {ffun Pt → bool} :=
 finfun (fun x ⇒ (choiceFunc k (fstp x)) == sndp x).

Definition subinit (initState:PSt) : PSt :=
 finfun (fun p ⇒ if (connect (fun x y ⇒ Adj x y ) (fste eth) (fstp p))
  then initState p else
  (nth false (rand_sendChosen 1 (Pinread nu p0 initState (fstp p)))
   (index (sndp p) (nu (fstp p))))).

Lemma forall_port : ∀ (s1 s2:VSt×PSt) ,
```

```
 (∀ (v:V),
    s1.1 v = s2.1 v ∧
   (∀ w, Adj v w → (s1.2 (VtoP v w p0) = s2.2 (VtoP v w p0)))) →
  s1 = s2.
```

```
Lemma hs1_aux11 : ∀ a l (x:VSt×PSt) (x0:VLab × seq PLab),
a \notin l →
(if [∀ v,(v \in a :: l)==>((update [set a] x.1 (Vwrite x0.1 a)) v == None)&&
   [∀ w,Adj v w ==>((update (WriteArea a) x.2 (Pwrite nu false x0.2 a))
   (VtoP v w p0) ==(subinit (coverTree 0)) (VtoP v w p0))]] then 1 else 0) ==
(B2U ((x0.1 == None) &&
      [∀ w, Adj a w ==> (nth false x0.2 (index w (nu a)) ==
 (subinit (coverTree 0)) (VtoP a w p0))]])*
(B2U ( [∀ v,(v \in l) ==>(x.1 v == None) &&
 [∀ w, Adj v w ==> (x.2 (VtoP v w p0) ==
          (subinit (coverTree 0)) (VtoP v w p0))]])).
```

```
Lemma hs1_aux12:
∀ res,
0 <(mu (DHS (enum V) res))
     (fun x ⇒ if [∀ v, (v \in (enum V)) ==>
                 ( ((x.1 v) == None) &&
 [∀ w, Adj v w ==> ((x.2 (VtoP v w p0)) ==
     ((subinit (coverTree 0)) (VtoP v w p0)))])]
  then 1 else 0).
```

```
Lemma hs1_aux1 : ∀ res,
0 <
   (mu (DHS (enum V) res))
     (fun x : VSt × PSt ⇒
      if x == (finfun (fun _ ⇒ None), subinit (coverTree 0)) then 1 else 0).
```

```
Lemma hs1_aux2 : ∀ k, (k < #|E|)%coq_nat →
0 <
   \big[(fun x : U ⇒ [eta Umult x])/1]_(k.+1 ≤ i < #|E|)
      finv (carac_hs_edge0 (nth e0 (enum E) i))
        ([ffun⇒ None], subinit (coverTree 0)).
```

```
Lemma hs1 : ∀ res,
∀ k, (k < #|E|)%coq_nat →
¬ (mu (DHS (enum V) res)) (fun a : VSt×PSt ⇒
      \big[(fun x : U ⇒ [eta Umult x])/1]_(k.+1 ≤ i < #|E|)
         finv (carac_hs_edge0 (nth e0 (enum E) i)) a) == 0.
```

hs2 Lemma hs_loc_neigh : ∀ e1 e2 x,

```
 (hs_edgeB e1 x) ->
 ((fste e1 == fste e2) && (snde e1 != snde e2)) ||
 ((fste e1 == snde e2) && (snde e1 != fste e2)) ||
 ((snde e1 == fste e2) && (fste e1 != snde e2)) ||
 ((snde e1 == snde e2) && (fste e1 != fste e2)) ->
 (hs_edgeB e2 x) = false.
Lemma hs2 : ∀ k, (k < #|E|)%coq_nat ->
 ∀ x0 : VSt×PSt,
 carac_hs_edge0 (nth e0 (enum E) k) x0 ×
 \big[(fun x1 : U ⇒ [eta Umult x1])/1]_(k.+1 ≤ i < #|E| |
 (fste (nth e0 (enum E) k) == fste (nth e0 (enum E) i)) ||
 (fste (nth e0 (enum E) k) == snde (nth e0 (enum E) i)) ||
 (snde (nth e0 (enum E) k) == fste (nth e0 (enum E) i)) ||
 (snde (nth e0 (enum E) k) == snde (nth e0 (enum E) i)))
 finv (carac_hs_edge0 (nth e0 (enum E) i)) x0 ==
 carac_hs_edge0 (nth e0 (enum E) k) x0.
```

hs3 Lemma carac_hs_loc_iff : $\forall(e{:}E)(v{:}V)(sn{:}VSt\times PSt)(x{:}VLab\times seq\ PLab)$,

```
 carac_hs_edge0 e (VPupdate nu false v x sn) =
 match (fste e == v), (snde e == v) with
  |true, true ⇒ B2U false
  |true, false⇒ B2U (
            (index (snde e) (nu v) == index true
               (take (seq.size (nu v)) (x.2 ++ nseq (seq.size (nu v)) false)))
           && (index v (nu (snde e)) == index true (Poutread nu p0 sn.2 (snde e) ))
           && (connect (fun v0 : V ⇒ [eta Adj v0]) (fste eth) v))
  |false, true⇒ B2U (
            (index v (nu (fste e)) == index true (Poutread nu p0 sn.2 (fste e)))
           && (index (fste e) (nu v) == index true
               (take (seq.size (nu v)) (x.2 ++ nseq (seq.size (nu v)) false)))
           && (connect (fun v0 : V ⇒ [eta Adj v0]) (fste eth) (fste e)))
  |false, false ⇒ carac_hs_edge0 e sn
 end.
Lemma hs3_aux : ∀ res (ek: E) (r:seq E),
 (fste ek \in (enum V)) -> (snde ek \in (enum V)) ->
 indep (DHS (enum V) res) (carac_hs_edge0 ek)
   (fun x0 : VSt × PSt ⇒
    \big[(fun x1 : U ⇒ [eta Umult x1])/1]_(e <- r)
       (if ~~
           ((fste ek == fste e) ||
            (fste ek == snde e) ||
            (snde ek == fste e) ||
```

```
            (snde ek == snde e))
        then finv (carac_hs_edge0 e) x0
        else 1)).

Lemma hs3 : ∀ res k, (k < #|E|)%coq_nat →
 indep (DHS (enum V) res) (carac_hs_edge0 (nth e0 (enum E) k))
   (fun x0 : VSt×PSt ⇒
    \big[(fun x1 : U ⇒ [eta Umult x1])/1]_(k.+1 ≤ i < #|E|)
       (if ~~
           ((fste (nth e0 (enum E) k)
             == fste (nth e0 (enum E) i)) ||
            (fste (nth e0 (enum E) k)
             == snde (nth e0 (enum E) i)) ||
            (snde (nth e0 (enum E) k)
             == fste (nth e0 (enum E) i)) ||
            (snde (nth e0 (enum E) k)
             == snde (nth e0 (enum E) i)))
        then finv (carac_hs_edge0 (nth e0 (enum E) i)) x0
        else 1)).
```

DHS_deg

```
Lemma DHS_deg_aux : ∀ initState,
(mu (DHS (enum V) initState)) (prodConj edge_finType
   (fun e : edge_finType ⇒ finv (fB2U
     (fun s : VSt × PSt ⇒ hs_edgeB e s &&
          connect (fun x : V ⇒ [eta Adj x]) (fste eth) (fste e)))))
 ≤
 hscte.

Lemma DHS_deg : ∀ initState,
 [1-] hscte
 ≤ (mu (DHS (enum V) initState)) (carac_hs_glob_ex).

End Handshake.
```

Chapter 25

Library handshake_rand

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun tuple.
Require Import Ensembles.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import setSem.
Require Import rdaTool_gen.
Require Import handshake_gen.
Require Import handshake_spec.
Set Implicit Arguments.
Import Prenex Implicits.
```

25.1 Introduction

We prove that there exists a randomised algorithms which solve the handshake algorithm with a non-null probability

```
Section hsalgo.
```

To have an hsAlgo, we consider the algorithm defined in handshake_gen

25.2 Definitions and proofs of hypotheses

```
Let VLabel := option_eqType nat_eqType.
Let PLabel := bool_eqType.
```

```
Let vunit : VLabel := None.
Let lunit : PLabel := true.

Let VSt (V: finType) := LabelFunc V VLabel.
Let Pt (V:finType) (Adj: rel V) := (@port_finType V Adj).
Let PSt (V:finType) (Adj: rel V) := LabelFunc (Pt Adj) PLabel.
Let State (V:finType) (Adj:rel V) := Datatypes.prod (LabelFunc V VLabel)
 ( LabelFunc (@port_finType V Adj) PLabel).

Definition rand_HsR : seq (VLabel → seq PLabel → seq PLabel → gen (VLabel × seq
PLabel)) :=
(randHSLoc::nil).

Definition rand_HsP (lv:VLabel) (lpout lpin :seq PLabel) : option nat :=
 if (agreed lpout lpin) then Some (index true lpout) else None.

Definition rand_HsI (V:finType) (Adj : rel V) (Gr:Graph Adj)(NG: NGraph Gr): State
Adj :=
  (finfun [ffun v ⇒ None] , finfun [ffun p ⇒ false]).

Lemma rand_HsI1 (V:finType) (Adj : rel V) (Gr:Graph Adj)(NG: NGraph Gr)(nu : V →
seq V)
     (Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) (Hnu2: ∀ (v :V), uniq (nu v))
     (p0 : Pt Adj) :
   consistent p0 nu rand_HsP (rand_HsI NG).

Lemma rand_HsI2 (V:finType) (Adj : rel V) (Gr:Graph Adj)(NG: NGraph Gr)(nu : V →
seq V)
     (Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) (Hnu2: ∀ (v :V), uniq (nu v)) :
     Uniform (rand_HsI NG).

Lemma rand_HsP1 (V:finType) (Adj: rel V) (Gr:Graph Adj)(NG: NGraph Gr)(nu:V → seq
V)
 (Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) (Hnu2: ∀ (v :V), uniq (nu v))
 (p0: Pt Adj) (s: State Adj) (v:V) (i:nat) :
 hsPortR p0 nu rand_HsP s v = Some i → i < deg Gr v.

Lemma HS1 (V:finType) (Adj: rel V) (nu:V → seq V)
 (Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v))) (Hnu2: ∀ (v :V), uniq (nu v))
  (p0: Pt Adj) :
∀ s s' w,
Setsem (GRound WriteArea (Vwrite (VLab:=VLabel)) (Pwrite nu false) (Vread (VLab:=VLabel))

(Pinread nu p0) (Poutread nu p0) (enum V) s randHSLoc) s' →
count id (Poutread nu p0 s'.2 w) ≤ 1 .

Lemma rand_HsRind (V:finType) (Adj: rel V) (Gr:Graph Adj)(NG: NGraph Gr)(nu:V →
seq V)
```

(*Hnu*: ∀ (*v w*:*V*), (*Adj v w*) = (*w* \in (*nu v*))) (*Hnu2*: ∀ (*v* :*V*), uniq (*nu v*))
(*p0*: *Pt Adj*) :
Stable _
(consistent *p0 nu* rand_HsP) (nextState false rand_HsR *p0 nu* (enum *V*)).

Definition rand_hs : (**hsAlgo** *VLabel* false) :=
(Build_hsAlgo rand_HsI1 rand_HsI2 rand_HsP1 rand_HsRind).

Lemma NonADet : ¬ Adet (HsR rand_hs).

Section Correct.

25.3 Correction

Context ‘(*NG*: **NGraph** *V Adj*).

Variable *nu* : *V* → seq *V*.
Hypothesis *Hnu*: ∀ (*v w*:*V*), (*Adj v w*) = (*w* \in (*nu v*)).
Hypothesis *Hnu2*: ∀ (*v* :*V*), uniq (*nu v*).

Let *Ptf* := *Pt Adj*.

Definition E1 := (@edge_finType *V Adj*).
Variable (*e0*:E1).
Definition p1 := (EtoP1 *e0*).

Let *VState* := LabelFunc *V VLabel*.
Let *PState* := LabelFunc *Ptf PLabel*.

Let *hsr* := (HsR rand_hs).
Let *hsp* := (HsP rand_hs).
Let *hsi* := (HsI rand_hs *NG*).
Let *hsi1* := (HsI1 rand_hs *NG Hnu Hnu2* p1).
Let *hsi2* := (HsI2 rand_hs *NG Hnu Hnu2*).
Let *hsp1* := (@HsP1 _ _ _ rand_hs _ _ _ *NG* _ *Hnu Hnu2* p1).
Let *hsrind* := (@HsRind _ _ _ rand_hs _ _ _ *NG* _ *Hnu Hnu2* p1).

Lemma rand_HsI_choice (*v*:*V*) :
agreed (Poutread *nu* p1 *hsi*.2 *v*)
(Pinread *nu* p1 *hsi*.2 *v*) = false.

Lemma rand_HsI3 :
@matching _ *Adj* (fun *v* ⇒ assNeigh p1 *nu hsp v hsi*).

Lemma rand_HSInvariant_matching :
Invariant _ (fun *s* ⇒ @matching _ *Adj* (fun *v* ⇒ assNeigh p1 *nu hsp v s*))
(nextState false *hsr* p1 *nu* (enum *V*)) *hsi*.

Definition f (*V0*:finType) (*Adj0*:rel *V0*) *nu0* (*l*:seq *V0*) (*p0*:@port_finType *V0 Adj0*):=

```
finfun (fun x:@port_finType V0 Adj0 ⇒
if ((fstp x) \in l) then
   if ((fstp x) == (fstp p0)) then if ((sndp x) == (sndp p0)) then true
                                  else false
   else if ((fstp x) == (sndp p0)) then if ((sndp x) == (fstp p0)) then true
                                  else false
   else if (index (sndp x) (nu0 (fstp x)) == O) then true
                                               else false
else false).
```

Lemma Real : hsRealisation rand_hs.

Section proba.

25.4 Analyse

Add *Rec LoadPath* "$ALEA_LIB/ALEA/src" as *ALEA*.

```
Require Import my_alea.
Require Import dist.
Require Import rdaTool_dist.
Require Import handshake_dist.
```

Set Implicit Arguments.

```
Open Local Scope U_scope.
Open Local Scope O_scope.
```

```
Lemma rand_hsexists :
  [1-] (@hscte _ Adj) ≤ mu (Distsem (GPStep nu false p1 hsr (enum V) hsi))
           (fun x ⇒ B2U ([∃ v, [∃ w,
            @hsBetween _ Adj (fun v ⇒ assNeigh p1 nu hsp v x) v w]])).
```

End proba.
End Correct.
End hsalgo.

Chapter 26

Library hsAct_gen

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.
Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import rdaTool_gen.
Set Implicit Arguments.
Import Prenex Implicits.
```

26.1 Introduction

The handshake algorithm is the following: each vertex v chooses a neighbour c(v) v sends 1 to c(v) and 0 to its other neighbour if v receives 1 from c(v) there is a handshake.

The message passing is simulated by a labelling on the ports If v has chosen c(v), the port p(v,c(v)) is relabelled 1.

The graphe contains vertices which are either active or inactive. We consider that new handshakes can only occur in the active subgraph.

```
Section HS.
```

26.2 The graph

```
Context `(NG: NGraph V Adj).
Variable nu : V → seq V.
```

```
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).
Let Pt := (@port_finType V Adj).
Variable p0 : Pt.
Let VLabel : eqType := option_eqType nat_eqType.
Let PLabel : eqType := bool_eqType.
Let VState := LabelFunc V VLabel.
Let PState := LabelFunc Pt PLabel.
```

26.3 Activity

```
Definition activeL (lv:VLabel) :=
  lv == None.
Definition numberActive (lpin: seq PLabel) : nat :=
 count (fun x ⇒ x==true) lpin.
Fixpoint sendChosen (k:nat) (lpin: seq PLabel) : seq PLabel :=
 match lpin with
   |t::q ⇒ match k with
              |0 ⇒ (false::(sendChosen 0 q))
              |1 ⇒ if t then (true::(sendChosen 0 q))
                      else (false::(sendChosen 1 q))
              |S k' ⇒ if t then (false::(sendChosen k' q))
                        else (false::sendChosen k q)
              end
   | nil ⇒ nil
 end.
Lemma sendChosen_size : ∀ k lpin,
 seq.size (sendChosen k lpin) = seq.size lpin.
Lemma sendChosen_memT : ∀ k lpin,
 (k < numberActive lpin)%nat →
  true \in sendChosen k.+1 lpin.
Lemma sendChosen_count : ∀ k lpin,
 (count id (sendChosen k.+1 lpin) ≤ 1)%nat.
Lemma sendChosen_countk : ∀ k lpin,
 (k < numberActive lpin)%nat →
 count id (sendChosen k.+1 lpin) = 1.
```

26.4 Algorithms

```
Definition HSLoc (lv:VLabel) (lpout lpin: seq PLabel) : gen (VLabel ×seq PLabel) :=
if (activeL lv) then
  match (numberActive lpin) with
   |O ⇒ Greturn _ (Some (seq.size lpout),nseq (seq.size lpout) false)
   |S n ⇒ Grandom _ n
       (fun k ⇒ Greturn _ (lv,sendChosen k.+1 lpin))
  end
else Greturn _ (lv,lpout).

Definition HSRound (seqV: seq V) (res: VState × PState):=
 GPRound nu false p0 seqV res HSLoc.

End HS.
```

Chapter 27

Library hsAct_op

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".

Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.

Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import op.
Require Import rdaTool_op.
Require Import hsAct_gen.

Set Implicit Arguments.
Import Prenex Implicits.
```

27.1 Introduction

This file is the simulation of the handshake algorithm over a graph containing active or inactive vertices.

```
Section HS.

Variable (rand_t : Type)(get : nat → rand_t → nat × rand_t).
Context (rand : ORandom _ get).

Let VLabel : eqType := option_eqType nat_eqType.
Let PLabel : eqType := bool_eqType.

Definition OHSLoc (lv:VLabel) (lpout lpin: seq PLabel)
 : Op rand_t (VLabel ×seq PLabel) :=
if (activeL lv) then
  match (numberActive lpin) with
```

```
    |O ⇒ Oreturn (Some (seq.size lpout),nseq (seq.size lpout) false)
    |S n ⇒ Obind (Orandom n rand)
        (fun k ⇒ Oreturn (lv,sendChosen k.+1 lpin))
  end
else Oreturn (lv,lpout).

Context `(NG: NGraph V Adj).

Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).

Let Pt := (@port_finType V Adj).
Variable p0 : Pt.

Let VState := LabelFunc V VLabel.
Let PState := LabelFunc Pt PLabel.

Definition OHSRound (seqV: seq V)(res: VState × PState) :=
 OPRound nu false p0 seqV res OHSLoc.

Section gen.

Lemma OPGHS_eq1 : ∀ (lv:VLabel) (lp1 lp2: seq PLabel) ,
 Opsem _ get rand (HSLoc lv lp1 lp2) =
 OHSLoc lv lp1 lp2.

Lemma OPGHS_eq2 : ∀ (seqV: seq V) (res:VState×PState),
 Opsem _ get rand (HSRound nu p0 seqV res) =
 OHSRound seqV res.

End gen.

Section simulation.

Definition OHSRoundF (seqV: seq V) (res: (V → VLabel) × (V × V → PLabel)) :=
 OPFRound nu false seqV res OHSLoc.

Lemma OHSF_eq1 : ∀ (seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v n,
 seqV = seqVF →
 (∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
    ((OHSRound seqV res n).1).1 v =
    ((OHSRoundF seqVF resF n).1).1 v.

Lemma OHSF_eq2 : ∀ (seqV seqVF : seq V)(res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v w n,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
 Adj v w →
```

```
    ((OHSRound seqV res n).1).2 (VtoP v w p0) =
    ((OHSRoundF seqVF resF n).1).2 (v,w).
 Lemma OHSF_eq3 : ∀ (seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) n,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
    (OHSRound seqV res n).2 =
    (OHSRoundF seqVF resF n).2.
End simulation.
End HS.
Section simulation.
    Definition of the graph
Inductive V : Type :=
 |v0 : V
 |v1 : V
 |v2 : V
 |v3 : V.
Definition eqV := (fun x y : V ⇒
 match x,y with
|v0,v0 ⇒ true
|v1,v1 ⇒ true
|v2,v2⇒true
|v3,v3 ⇒ true
|_,_ ⇒ false
end).
Lemma eqVP : Equality.axiom eqV.
Canonical V_eqMixin := EqMixin eqVP.
Canonical V_eqType:= Eval hnf in EqType V V_eqMixin.
Lemma V_pickleK : pcancel (fun v : V ⇒ match v with |v0 ⇒ O |v1 ⇒ 1%nat |v2 ⇒
2 |v3 ⇒ 3 end)
 (fun x : nat ⇒ match x with |0 ⇒ Some v0 | 1 ⇒ Some v1 |2 ⇒ Some v2 | 3 ⇒ Some v3
| _ ⇒ None end).
Fact V_choiceMixin : choiceMixin V.
Canonical V_choiceType := Eval hnf in ChoiceType V V_choiceMixin.
Definition V_countMixin := CountMixin V_pickleK.
Canonical V_countType := Eval hnf in CountType V V_countMixin.
Definition venum := (v0:: v1:: v2:: v3:: nil).
```

```
Lemma V_enumP : Finite.axiom venum.

Definition V_finMixin := Eval hnf in FinMixin V_enumP.
Canonical V_finType := Eval hnf in FinType V V_finMixin.

Lemma card_V : #|{: V}| = 4.

Definition Adj : rel V := (fun x y ⇒ match x, y with
 |v0,v1 |v0,v3 |v1,v0 |v1,v2 |v1,v3 |v2,v1 |v2,v3 |v3,v0 |v3,v1 |v3,v2 ⇒ true
 | _,_ ⇒ false
 end).

Lemma AdjSym : symmetric Adj.

Lemma AdjIrrefl : irreflexive Adj.

Lemma enumV : (enum V_finType) = ([::v0;v1;v2;v3] ).

Context `(NG: NGraph V_finType Adj).

Lemma Nb_enumv0 : Nb_enum Gr v0 = (v1::v3::nil).

Lemma degv0 : (deg Gr v0) = 2.

Definition nu (v: V) : seq V :=
 match v with
   |v0 ⇒ [::v1;v3]
   |v1 ⇒ [::v0;v2;v3]
   |v2 ⇒ [::v1;v3]
   |v3 ⇒ [::v1;v2;v0]
end.

Lemma nuAdj_eq : ∀ u w,
Adj u w = (w \in nu u).

Lemma hp0 : Adj (v0,v1).1 (v0,v1).2.

Definition p0 := Port hp0.
```

Definition of the labelling

```
Let VLabel : eqType := option_eqType nat_eqType.
Let PLabel : eqType := bool_eqType.

Definition initV : (LabelFunc V_finType VLabel) :=
finfun (fun x:V ⇒ None).

Definition initP : (LabelFunc (@port_finType V_finType Adj) PLabel) :=
 finfun (fun x ⇒ true).

Definition init := (initV, initP).

Definition initVF : (V → VLabel) :=
 (fun x:V ⇒ None).

Definition initPF : ((V×V) → PLabel) :=
 (fun x ⇒ true).

Definition initF := (initVF, initPF).
```

```
Lemma init_eq1 : ∀ v, init.1 v = initF.1 v.

Lemma init_eq2 : ∀ v w,
 Adj v w → init.2 (VtoP v w p0) = initF.2 (v,w).
```

Equivalence

```
Lemma OHSF_eq4 : ∀ v n,
   ((OHSRound my_gen nu p0 (enum V_finType) init n).1).1 v =
   ((OHSRoundF my_gen nu [::v0;v1;v2;v3] initF n).1).1 v.

Lemma OHSF_eq5 : ∀ v w n,
 Adj v w →
   ((OHSRound my_gen nu p0 (enum V_finType) init n).1).2 (VtoP v w p0) =
   ((OHSRoundF my_gen nu [::v0;v1;v2;v3] initF n).1).2 (v,w).

 Lemma OHSF_eq6 : ∀ n,
   (OHSRound my_gen nu p0 (enum V_finType) init n).2 =
   (OHSRoundF my_gen nu [::v0;v1;v2;v3] initF n).2.
```

Computation

```
Let R1 := (OHSRoundF my_gen nu [::v0;v1;v2;v3] initF) 6.

Check (R1).

Eval vm_compute in (R1.1.1 v3).
Eval vm_compute in (R1.1.2 (v3,v1)).
Eval vm_compute in (R1.1.2 (v3,v2)).
Eval vm_compute in (R1.1.2 (v3,v0)).

Eval vm_compute in (R1.1.1 v0).
Eval vm_compute in (R1.1.2 (v0,v1)).
Eval vm_compute in (R1.1.2 (v0,v3)).
Eval vm_compute in (R1.1.2 (v0,v0)).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] R1.1).

End simulation.
```

Chapter 28

Library hsAct_dist

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat.
Require Import fintype finset fingraph seq finfun bigop choice tuple.
Import Prenex Implicits.

Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".

Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Require Import Rplus.
Require Import my_alea.
Require Import my_ssr.
Require Import my_ssralea.
Require Import graph.
Require Import graph_alea.
Require Import labelling.
Require Import bfs.
Require Import gen.
Require Import dist.
Require Import rdaTool_gen.
Require Import rdaTool_dist.
Require Import hsAct_gen.

Set Implicit Arguments.

Open Local Scope U_scope.
Open Local Scope O_scope.
```

28.1 Introduction

This file is the analysis of the solution of the handshake problem over an active subgraph.

Section Handshake.

28.2 The graph

```
Context `(NG: NGraph V Adj).
Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).
Definition E := (@edge_finType V Adj).
Variable e0:E.
Definition Pt := (@port_finType V Adj).
Definition p0 := (EtoP1 e0).
Definition VLab : eqType := option_eqType nat_eqType.
Definition PLab : eqType := bool_eqType.
Definition VSt := LabelFunc V VLab.
Definition PSt := LabelFunc Pt PLab.
```

28.3 Some other definitions

```
Definition activeG (v:V) (s:VSt × PSt):=
  s.1 v == None.
Definition inactiveG (v:V) (s:VSt × PSt):=
 ∃ i, s.1 v == Some i.
Fixpoint index_ithActive_aux (lpin:seq PLab) (i res:nat) :=
 match lpin with
   |nil ⇒ res
   |t::q ⇒ if t then
              match i with |O ⇒ res |S n ⇒ (index_ithActive_aux q n res.+1) end
              else (index_ithActive_aux q i res.+1)
 end.
Definition index_ithActive (lpin:seq PLab) (i:nat) :=
 index_ithActive_aux lpin i 0.
Lemma nthActive0 : ∀ l n m,
index_ithActive_aux l n m.+1 = (index_ithActive_aux l n m).+1.
Lemma nthActive1 : ∀ lpin k,
```

index_ithActive *lpin* *k* = index true (sendChosen *k*.+1 *lpin*).

Lemma nthActive2 : ∀ *lpin* *k* *x*, (*k* < numberActive *lpin*)%*nat* →
$k \neq x$ → index_ithActive *lpin* *k* ≠ index_ithActive *lpin* *x*.

28.4 Local Algorithm

```
Definition DHSLoc (lv:VLab) (lpout lpin: seq PLab)
 : distr (VLab ×seq PLab) :=
if (activeL lv) then
  match (numberActive lpin) with
   |O ⇒ Munit (Some (seq.size lpout),nseq (seq.size lpout) false)
   |S n ⇒ Mlet (Random n)
      (fun k ⇒ Munit (lv,sendChosen k.+1 lpin))
  end
else Munit (lv,lpout).
```

Section gen.

28.4.1 Proofs of the equivalence with the generic algorithm

Lemma DPGHS_eq1 : ∀ (*lv*:VLab) (*lp1* *lp2*: seq PLab) ,
Distsem (HSLoc *lv* *lp1* *lp2*) =
DHSLoc *lv* *lp1* *lp2*.

End gen.

28.4.2 Local Analysis

DHSLoc can be decomposed in a sum of computations around each port

Lemma is_discrete_DHSLoc : ∀ (*lv*:VLab) (*lpout*:seq PLab)
(*lpin*:seq PLab),
is_discrete_s (DHSLoc *lv* *lpout* *lpin*).

DHSLoc terminates

Lemma DHSLoc_total : ∀ (*lv*:VLab) (*lpout*:seq PLab) (*lpin*:seq PLab),
Term (DHSLoc *lv* *lpout* *lpin*).

The number generated by (DHSLoc v) are inferior or equal to the number of active

Definition carac_lc_size : seq PLab → VLab × seq PLab → *U* :=
fun (*lpin*:seq PLab) (*s*:VLab × seq PLab) ⇒
B2U ((index true *s*.2) < seq.size *lpin*)%*nat*.

Lemma DHSLoc_size : ∀ (*lv*:VLab) (*lpout*:seq PLab) (*lpin*:seq PLab),
(0 < numberActive *lpin*)%*nat* →

```
activeL lv →
mu (DHSLoc lv lpout lpin) (carac_lc_size lpin) == 1.
```

The probability for a vertex to choose the ith neighbour is 1/(deg v)

carac_lc_eq returns true if i is equal to the choice of v i.e. it returns true if v chooses its ith neighbour else false

```
Definition carac_lc_eq : nat → seq PLab → VLab × seq PLab → U :=
 fun (i: nat) (lpin:seq PLab) (s: VLab × seq PLab) ⇒
  B2U ( (index_ithActive lpin i) == (index true s.2)).
```

```
Lemma DHSLoc_eq : ∀ (lv:VLab)(lpout lpin:seq PLab)(k n: nat),
 (k < n.+1)%nat →
 numberActive lpin = n.+1 →
 activeL lv →
 (mu (DHSLoc lv lpout lpin)) (carac_lc_eq k lpin) == [1/]1+n.
```

28.5 Global Algorithm

DHS seqV res : at the end of the algorithm DHS, each vertices in seqV has made a choice among its neighbours and has updated its choice in res

```
Definition DHS (seqV: seq V) (res: VSt × PSt): distr (VSt × PSt) :=
 DPRound nu false p0 seqV res DHSLoc.
```

```
Section genRound.
```

28.5.1 Proofs of the equivalence with the generic algorithm

```
Lemma DPGHS_eq2 : ∀ (seqV: seq V) (res:VSt×PSt),
 Distsem (HSRound nu p0 seqV res) =
 DHS seqV res.
```

```
End genRound.
```

28.5.2 Analysis

Termination

DHS terminates whichever the sequence of vertices on which DHS is applied

```
Lemma DHS_total : ∀(s: seq V) (res: VSt×PSt),
 Term (DHS s res).
```

Probability to choose a neighbour, local view

The probability for a vertex to choose the ith neighbour (i.e. ith neighbour is labelled true) is 1/(deg v) carac_hs_eqNat returns true if i is equal to the local choice of v extracted from the global labelling function i.e. it returns true if v chooses its ith neighbour else false

```
Definition carac_hs_eqNat : V → seq PLab → nat → VSt × PSt → U :=
 fun (v:V) (lpin:seq PLab) (i: nat) (s: VSt × PSt) ⇒
 B2U (index_ithActive lpin i ==
 index true (Poutread nu p0 s.2 v)).
```

```
Lemma DHS_degv_aux1 : ∀ (v:V) (i:nat) (lpin:seq PLab) (y:VLab × seq PLab)
(sn:VSt × PSt),
 seq.size y.2 = seq.size (nu v) →
 carac_lc_eq i lpin y ==
 carac_hs_eqNat v lpin i (VPupdate nu false v y sn).
```

```
Lemma DHS_size1 : ∀ a b c,
 seq.size b = seq.size c →
 (mu (DHSLoc a b c)) (fun x ⇒ B2U(seq.size x.2 != seq.size c)) == 0.
```

```
Lemma DHSLtac1 a b c d f:
 seq.size b = seq.size c →
 (mu (DHSLoc a b c)) (fplus f
      (fun x ⇒ B2U(seq.size x.2 != seq.size c))) == d →
 (mu (DHSLoc a b c)) f == d.
```

```
Lemma DHS_degv_aux2 : ∀ (v:V) (x:VLab×seq PLab) (s:VSt×PSt)(i:nat),
  seq.size x.2 = seq.size (nu v) →
  (mu (DHS (seq.rem v (enum V)) s))
        (fun x0 : LabelFunc V VLab × LabelFunc port_finType bool_eqType ⇒
         carac_hs_eqNat v (Pinread nu p0 s.2 v) i (VPupdate nu false v x x0)) ==
         carac_hs_eqNat v (Pinread nu p0 s.2 v) i (VPupdate nu false v x s).
```

```
Lemma DHS_degv_local : ∀ (v:V)(i n:nat)(s:VSt×PSt),
 (i < n.+1)%nat →
 numberActive (Pinread nu p0 s.2 v) = n.+1 →
 activeG v s →
 (mu (DHS (enum V) s)) (carac_hs_eqNat v (Pinread nu p0 s.2 v) i) ==
 [1/]1+n.
```

Probability to choose a neighbour, global view

The probability for a vertex to choose the vertex w which is a neighbour is 1/(deg v)

carac_hs_eqV returns true if v chooses w else false

```
Definition hs_eqVB (v w:V) (s:VSt×PSt) :=
 index w (nu v) ==
 index true (Poutread nu p0 s.2 v).
```

```
Definition carac_hs_eqV : V → V → VSt×PSt → VSt×PSt →U :=
 fun (v w: V) (inits s:VSt×PSt) ⇒
```

```
  B2U (hs_eqVB v w s).
```

```
Lemma carac_hs_iff : ∀ (v w: V) (inits:VSt ×PSt) (i:nat),
 index w (nu v) = index_ithActive (Pinread nu p0 inits.2 v) i →
 carac_hs_eqV v w inits ==
 carac_hs_eqNat v (Pinread nu p0 inits.2 v) i.
```

```
Definition is_neigh_active (v w:V) (s:VSt×PSt) :=
(nth false (Pinread nu p0 s.2 v) (index w (nu v))).
```

```
Lemma is_neigh_active1 : ∀ (v w:V) (s:VSt×PSt),
 is_neigh_active v w s →
 ∃ i,
 (i < numberActive (Pinread nu p0 s.2 v))%nat ∧
 (index w (nu v) = index_ithActive (Pinread nu p0 s.2 v) i).
```

```
Lemma is_neigh_active2 : ∀ (v w:V) (s:VSt×PSt),
 is_neigh_active v w s →
 Adj v w.
```

```
Lemma DHS_degv_global : ∀ (v w: V) (s:VSt×PSt) (n:nat),
  numberActive (Pinread nu p0 s.2 v) = n.+1 →
  is_neigh_active v w s →
  activeG v s →
 (mu (DHS (enum V) s)) (carac_hs_eqV v w s) == [1/]1+n.
```

```
Section initState.
```

We assume that initial state is coherent according to the activity of vertices `Variable initState : VSt × PSt.`

```
Hypothesis initState1 : ∀ (v:V), (activeG v initState) →
 (Poutread nu p0 initState.2 v) = nseq (seq.size (nu v)) true.
```

```
Hypothesis initState2 : ∀ v, (inactiveG v initState) →
 (Poutread nu p0 initState.2 v) = nseq (seq.size (nu v)) false.
```

Probability of having a handshake on an edge

The probability for an edge (v,w) having a handshake on it is 1/(deg v * deg w)
carac_hse returns true if v chooses w and w chooses v else false

```
Definition hs_edgeB (e:E) (s:VSt×PSt) : bool :=
(hs_eqVB (fste e) (snde e) s) && (hs_eqVB (snde e) (fste e) s).
```

```
Definition carac_hs_edge : E → VSt×PSt → U :=
 fun (e:E) ⇒
  fB2U (fun (s:VSt×PSt) ⇒ hs_edgeB e s).
```

```
Definition nactv (v:V) :=
 (numberActive (Pinread nu p0 initState.2 v)).
```

```
Definition nactvdecr (v:V) :=
 (numberActive (Pinread nu p0 initState.2 v)).-1.

Lemma activeG1 : ∀ v w,
 activeG w initState →
 Adj v w →
 is_neigh_active v w initState.

Lemma activeG2 : ∀ v w,
 activeG w initState →
 Adj v w →
 ∃ n, numberActive (Pinread nu p0 initState.2 v) = n.+1.

Lemma activeG3 : ∀ (i:E),
 activeG (fste i) initState →
 (0 < (numberActive (Pinread nu p0 initState.2 (snde i))))%nat.

Lemma activeG4 : ∀ (i:E),
 activeG (snde i) initState →
 (0 < (numberActive (Pinread nu p0 initState.2 (fste i))))%nat.

Lemma indepbDHS_hs : ∀ (e:E) (inits:VSt×PSt),
 indepb (DHS (enum V) inits)
     (hs_eqVB (fste e) (snde e))
     (hs_eqVB (snde e) (fste e)).

Lemma DHS_dege : ∀ (e:E),
 activeG (fste e) initState →
 activeG (snde e) initState →
 mu (DHS (enum V) initState) (carac_hs_edge e) ==
    [1/]1+(nactvdecr (fste e)) ×
    [1/]1+(nactvdecr (snde e)).
```

Probability for having at least one vertex

```
Require Import Rplus.
```

hs_glob s returns true if there is a handshake in the graph else false

```
Definition hs_glob (x:E) (inits s:VSt×PSt):=
 (activeG (fste x) inits) && (activeG (snde x) inits) && (hs_edgeB x s).

Definition hs_glob_ex (inits s:VSt×PSt) : bool :=
   [∃ x, hs_glob x inits s].
```

carac_hs_glob s returns 1 if there is a handshake in the graph else 0

```
Definition carac_hs_glob (x:E) (inits:VSt×PSt): VSt×PSt → U :=
 fB2U (fun (s:VSt×PSt) ⇒ hs_glob x inits s).

Definition carac_hs_glob_ex (inits:VSt×PSt): VSt×PSt → U :=
```

```
  fB2U (fun (s:VSt×PSt) ⇒ hs_glob_ex inits s).
Definition hscte := prod (fun _ ⇒ [1-] ([1/2] × [1/]1+(#|E|.-1))) #|E|.
```

Rpsigma_hs

```
Lemma subsetmem1 : ∀ (l l':seq V) (a:V),
  a \notin l' → l' \subset a::l →
  l' \subset l.

Lemma rem_mem_not : ∀ (l:seq V) (i a:V),
   i \in l → i \notin seq.rem a l → i = a.

Lemma remsubsetcons : ∀ (l l':seq V) a, uniq l' →
  l' \subset a :: l → a \in l' → seq.rem a l' \subset l.

Lemma map_nseq_eq1 : ∀ (l:seq V) f a,
  [seq f x | x ← l] = nseq (seq.size l) true →
  [seq f x | x ← seq.rem a l] =
    nseq (seq.size (seq.rem a l)) true.

Lemma numberActive1 : ∀ l l', perm_eq l l' →
  numberActive l = numberActive l'.

Lemma numberActive2 : ∀ l a v,
  a \in l →
    numberActive [seq initState.2 (VtoP x0 v p0) | x0 ← l] =
    ( (initState.2 (VtoP a v p0) == true) +
      numberActive [seq initState.2 (VtoP x0 v p0) | x0 ← seq.rem a l])%nat.

Lemma activeinactive : ∀ a s,
  activeG a s = false → inactiveG a s.

Lemma activeinit1 : ∀ v w,
  v \in nu w → activeG w initState = false →
  initState.2 (VtoP w v p0) = false.

Lemma activeinit2 : ∀ v w,
  v \in nu w → activeG w initState = true →
  initState.2 (VtoP w v p0) = true.

Lemma numberActive3 : ∀ v w,
  v \in nu w →
  initState.2 (VtoP v w p0) →
  (0 < numberActive [seq initState.2 (VtoP x1 w p0) | x1 ← nu w])%nat.

Lemma nactvdecr2' : ∀ v,
activeG v initState → (0 < nactv v)%nat →
    count (fun i : V ⇒ (i \in nu v) && (activeG i initState &&
  (0 < nactv i)%nat))
      (enum V) = (nactvdecr v).+1.

Lemma nactvdecr2 : ∀ v,
```

```
activeG v initState → (0 < nactv v)%nat →
   count (fun i : V ⇒ Adj v i && (activeG i initState && (0 < nactv i)%nat))
     (enum V) = (nactvdecr v).+1.

Lemma nactvdecr1 : ∀ n w,
 activeG w initState →
 (0 < nactv w)%nat →
 count (fun x ⇒ activeG x initState && (0 < nactv x)%nat) (enum V) = n.+1 →
 (nactvdecr w ≤ n)%nat.

Lemma Rpsigma_hs : ∀ (e:E) ,
 (0 < (count (fun x ⇒ activeG x initState && (0<nactv x)%nat )
(enum V)))%nat →
 U2Rp([1/2]) ≤
   (Rpsigma (fun k : nat ⇒
       (mu (DHS (enum V) initState)) (carac_hs_glob (nth e0 (enum E) k)
initState)))
    #|E|.
```

hs1

```
Definition eth (k:nat) :=
 nth e0 (enum E) k.

Definition AdjAct : rel V := (fun x y ⇒
 (activeG x initState) && (activeG y initState) && (Adj x y)).

Definition parentFunc (k:nat) := (@tF _ AdjAct (fste (eth k)) #|V|).

Definition choiceFunc (k:nat) : {ffun V → V} :=
 finfun (fun x ⇒ match parentFunc k x with
                     |Some y ⇒ y
                     |None ⇒ snde (eth k)
                    end).

Definition coverTree (k: nat) : {ffun Pt → bool} :=
 finfun (fun x ⇒ if activeG (fstp x) initState && activeG (sndp x) initState
                   then (choiceFunc k (fstp x)) == sndp x
                   else false).
Print Vupdate.
Lemma hs1_aux11 : ∀ l v k x x0,
v \notin l →
(if [∀ v0, (v0 \in v :: l) ==> ((Vupdate v x0 x.1) v0 == initState.1 v0) &&
      [∀ w,Adj v0 w ==>((Pupdate nu false v x0 x.2 ) (VtoP v0 w p0) ==
                (coverTree k) (VtoP v0 w p0))]] then 1 else 0) ==
       (B2U ((x0.1 == initState.1 v) &&
               [∀ w, Adj v w ==> (nth false x0.2 (index w (nu v)) ==
 (coverTree k) (VtoP v w p0))]))*
      (B2U ( [∀ v0,(v0 \in l) ==>
```

```
                ( x.1 v0 == initState.1 v0) &&
                [∀ w, Adj v0 w ==> (x.2 (VtoP v0 w p0) ==
                     (coverTree k) (VtoP v0 w p0))]])).
Lemma DHS_inactive : ∀ (l1 l2:seq V) (s:VSt×PSt) (f: MF (VSt×PSt) ),
 (∀ v, inactiveG v s → (fun x ⇒ f (VPupdate nu false v (s.1 v,
Poutread nu p0 s.2 v)
x)) == f ) →
 count (activeG^~ s) l2 = O →
 mu (DHS (l1++l2) s) f ==
 mu (DHS l1 s) f.
Lemma VPupdate_id : ∀ a (s:VSt×PSt),
 (∀ v : V, inactiveG v s →
                Poutread nu p0 s.2 v = nseq (seq.size (nu v)) false)->
 inactiveG a s →
 (VPupdate nu false a (s.1 a, Poutread nu p0 s.2 a) s) = s.
Lemma DHS_inactive' : ∀ (l:seq V) (s:VSt×PSt),
(∀ v : V, inactiveG v s →
                Poutread nu p0 s.2 v = nseq (seq.size (nu v)) false)->
(∀ u, u \in l → inactiveG u s) →
 mu (DHS l s) == mu (Munit s).
Lemma DHS_deg_connect_aux24 : ∀ (e:E) s,
activeG (fste e) s → activeG (snde e) s →
(∀ u v, activeG u s → activeG v s →
  connect (fun x y ⇒ activeG x s && activeG y s && Adj x y ) u v) →
∀ v,
 activeG v s →
 ∃ w, Adj v w ∧ activeG w s.
Lemma numberActive_conn : ∀ (e:E) ,
activeG (fste e) initState → activeG (snde e) initState →
(∀ u v, activeG u initState → activeG v initState →
  connect (fun x y ⇒ activeG x initState && activeG y initState && Adj x y )
u v) →
∀ v,
activeG v initState →
(0 < (numberActive (Pinread nu p0 initState.2 v)))%nat.
Lemma choiceFunc1 : (∀ u v : V,
          activeG u initState →
          activeG v initState →
          connect
            (fun x y : V ⇒
             activeG x initState && activeG y initState && Adj x y) u v) →
```

```
 ∀ v k, activeG (fste (eth k)) initState →
 activeG v initState → Adj v (choiceFunc k v).

Lemma choiceFunc2 :
 ∀ v k,
 activeG (snde (eth k)) initState → activeG (choiceFunc k v) initState.

Lemma sendChosen3 : ∀ l i, (i < numberActive l)%nat →
 nth false (sendChosen i.+1 l) (index_ithActive l i) = true.

Lemma sendChosen4 : ∀ l l' l'' x,
(sendChosen x.+1 l) = (l''++true::l') → true \notin l'.

Lemma sendChosen2 : ∀ l x i,
 nth false (sendChosen x.+1 l) i →
 index_ithActive l x = i.

Lemma index_eq : ∀ (l:seq V) x y, x \in l → index x l = index y l →
 x = y.

Lemma hs1_aux12:
(∀ u v, activeG u initState → activeG v initState →
  connect (fun x y ⇒ activeG x initState && activeG y initState && Adj x y )
u v) →
 (0 < count (fun x ⇒ activeG x initState) (enum V))%nat →
∀ k, activeG (fste (eth k)) initState →
 activeG (snde (eth k)) initState →
0 <(mu (DHS (enum V) initState))
     (fun x ⇒ if [∀ v, (v \in (enum V)) ==>
                 ( ((x.1 v) == initState.1 v) &&
  [∀ w, Adj v w ==> ((x.2 (VtoP v w p0)) ==
     ((coverTree k) (VtoP v w p0)))])]
  then 1 else 0).

Lemma forall_port : ∀ (s1 s2:VSt×PSt) ,
 (∀ (v:V),
    s1.1 v = s2.1 v ∧
   (∀ w, Adj v w → (s1.2 (VtoP v w p0) = s2.2 (VtoP v w p0)))) →
  s1 = s2.

Lemma hs1_aux1 :
(∀ u v, activeG u initState → activeG v initState →
  connect (fun x y ⇒ activeG x initState && activeG y initState && Adj x y )
u v) →
 (0 < count (fun x ⇒ activeG x initState) (enum V))%nat →
∀ k, activeG (fste (eth k)) initState → activeG (snde (eth k)) initState →
0 <(mu (DHS (enum V) initState))
     (fun x ⇒ if x == (initState.1,coverTree k) then 1 else 0).
```

Lemma hs1_aux2' :
 (∀ *u v*, activeG *u initState* → activeG *v initState* →
  connect (fun *x y* ⇒ activeG *x initState* && activeG *y initState* && *Adj x y*)
u v) →
∀ *i k*,
 (k < i < #|E|)%*nat* → activeG (fste (eth *k*)) *initState* →
 (carac_hs_glob (nth *e0* (enum E) *i*)) *initState* (*initState*.1, coverTree *k*) == 0.

Lemma hs1_aux2 :
 (∀ *u v*, activeG *u initState* → activeG *v initState* →
  connect (fun *x y* ⇒ activeG *x initState* && activeG *y initState* && *Adj x y*)
u v) →
∀ *k*,
activeG (fste (eth *k*)) *initState* →
 0 <
 \big[(fun *x* : *U* ⇒ [eta *Umult x*])/1]_(*k*.+1 ≤ *i* < #|E|)
    finv (carac_hs_glob (nth *e0* (enum E) *i*) *initState*) (*initState*.1, coverTree *k*).

Lemma hs1 : ∀ (*e*:E), activeG (fste *e*) *initState* →
activeG (snde *e*) *initState* →
 (∀ *u v*, activeG *u initState* → activeG *v initState* →
  connect (fun *x y* ⇒ activeG *x initState* && activeG *y initState* && *Adj x y*)
u v) →
∀ *k*, (*k* < #|E|)%*coq_nat* →
activeG (fste (nth *e0* (enum E) *k*)) *initState* →
activeG (snde (nth *e0* (enum E) *k*)) *initState* →
¬ (mu (DHS (enum *V*) *initState*)) (fun *a* : VSt×PSt ⇒
      \big[(fun *x* : *U* ⇒ [eta *Umult x*])/1]_(*k*.+1 ≤ *i* < #|E|)
         finv (carac_hs_glob (nth *e0* (enum E) *i*) *initState*) *a*) == 0.

hs2 Lemma hs_loc_neigh : ∀ *e1 e2 s x*,
 hs_glob *e1 s x* →
 ((fste *e1* == fste *e2*) && (snde *e1* != snde *e2*)) ||
 ((fste *e1* == snde *e2*) && (snde *e1* != fste *e2*)) ||
 ((snde *e1* == fste *e2*) && (fste *e1* != snde *e2*)) ||
 ((snde *e1* == snde *e2*) && (fste *e1* != fste *e2*)) →
 (hs_glob *e2 s x*) = false.

Lemma hs2 : ∀ *k*, (*k* < #|E|)%*coq_nat* →
 ∀ *x0* : VSt×PSt,
 carac_hs_glob (nth *e0* (enum E) *k*) *initState x0* ×
 \big[(fun *x1* : *U* ⇒ [eta *Umult x1*])/1]_(*k*.+1 ≤ *i* < #|E| |
 (fste (nth *e0* (enum E) *k*) == fste (nth *e0* (enum E) *i*)) ||
 (fste (nth *e0* (enum E) *k*) == snde (nth *e0* (enum E) *i*)) ||

```
 (snde (nth e0 (enum E) k) == fste (nth e0 (enum E) i)) ||
 (snde (nth e0 (enum E) k) == snde (nth e0 (enum E) i)))
 finv (carac_hs_glob (nth e0 (enum E) i) initState) x0 ==
 carac_hs_glob (nth e0 (enum E) k) initState x0.
```

hs3 Lemma carac_hs_loc_iff : ∀(e:E)(v:V)(sn:VSt×PSt)(x:VLab×seq PLab),

```
 carac_hs_glob e initState (VPupdate nu false v x sn) =
 match (fste e == v), (snde e == v) with
  |true, true ⇒ B2U false
  |true, false⇒ B2U ((activeG v initState && (activeG (snde e) initState) &&
           ((index (snde e) (nu v) == index true
             (take (seq.size (nu v)) (x.2 ++ nseq (seq.size (nu v)) false)))
         && (index v (nu (snde e)) == index true (Poutread nu p0 sn.2 (snde e) )))))
  |false, true⇒ B2U ((activeG (fste e) initState && (activeG v initState) &&
           ((index v (nu (fste e)) == index true (Poutread nu p0 sn.2 (fste e)))
         && (index (fste e) (nu v) == index true
             (take (seq.size (nu v)) (x.2 ++ nseq (seq.size (nu v)) false))))))
  |false, false ⇒ carac_hs_glob e initState sn
 end.
Lemma hs3_aux : ∀ (ek: E) (r:seq E),
 (fste ek \in (enum V)) → (snde ek \in (enum V)) →
 indep (DHS (enum V) initState) (carac_hs_glob ek initState)
   (fun x0 : VSt × PSt ⇒
    \big[(fun x1 : U ⇒ [eta Umult x1])/1]_(e ← r)
      (if ~~
          ((fste ek == fste e) ||
           (fste ek == snde e) ||
           (snde ek == fste e) ||
           (snde ek == snde e))
       then finv (carac_hs_glob e initState) x0
       else 1)).
Lemma hs3 : ∀ k, (k < #|E|)%coq_nat →
 indep (DHS (enum V) initState) (carac_hs_glob (nth e0 (enum E) k) initState)
   (fun x0 : VSt×PSt ⇒
    \big[(fun x1 : U ⇒ [eta Umult x1])/1]_(k.+1 ≤ i < #|E|)
      (if ~~
          ((fste (nth e0 (enum E) k)
             == fste (nth e0 (enum E) i)) ||
           (fste (nth e0 (enum E) k)
             == snde (nth e0 (enum E) i)) ||
           (snde (nth e0 (enum E) k)
             == fste (nth e0 (enum E) i)) ||
```

```
            (snde (nth e0 (enum E) k)
             == snde (nth e0 (enum E) i)))
        then finv (carac_hs_glob (nth e0 (enum E) i) initState) x0
        else 1)).
```

DHS_deg

```
Lemma DHS_deg_aux :
 (∀ u v, activeG u initState → activeG v initState →
 connect (fun x y ⇒ activeG x initState && activeG y initState && Adj x y )
    u v)
→
 (0 < count (fun x : V ⇒ activeG x initState && (0 < nactv x)%nat)
(enum V))%nat →
(mu (DHS (enum V) initState)) (prodConj edge_finType
   (fun e : edge_finType ⇒ finv (fB2U
      (fun s : VSt × PSt ⇒ hs_glob e initState s))))
 ≤
 hscte.
```

```
Lemma DHS_deg :
(∀ u v, activeG u initState → activeG v initState →
 connect (fun x y ⇒ activeG x initState && activeG y initState && Adj x y )
   u v) →
 (0 < count (fun x : V ⇒ activeG x initState && (0 < nactv x)%nat)
 (enum V))%nat →
 [1-] hscte
 ≤ (mu (DHS (enum V) initState)) (carac_hs_glob_ex initState).
```

For a connected subgraph

```
Definition subinit1 (e:E) : VSt :=
 finfun (fun v ⇒ if (connect (fun x y ⇒ activeG x initState && activeG y
 initState && Adj x y ) (fste e) v) then initState.1 v else Some O).
```

```
Definition subinit2 (e:E) : PSt :=
 finfun (fun p ⇒ if (connect (fun x y ⇒ activeG x initState && activeG y
 initState && Adj x y ) (fste e) (fstp p)) then initState.2 p else false).
```

```
Lemma connectProp : ∀ f (x y:V),
 (∀ w, f w y = false) →
 x ≠ y → connect f x y → false.
```

```
Lemma DHS_deg_exconn : ∀ (e':E), activeG (fste e') initState →
activeG (snde e') initState →
∃ (e:E) (s:VSt×PSt),
activeG (fste e) s ∧
 activeG (snde e) s ∧
 (∀ v : V, inactiveG v s →
```

```
    Poutread nu p0 s.2 v = nseq (seq.size (nu v)) false) ∧
 (∀ u v, activeG u s → activeG v s →
      connect (fun x y ⇒ activeG x s && activeG y s && Adj x y ) u v) ∧
 (∀ v:V, activeG v s → s.1 v = initState.1 v) ∧
 (∀ v x:V, activeG v s → Adj v x →
      s.2 (VtoP v x p0) = initState.2 (VtoP v x p0))/\
 (∀ v x:V, inactiveG v initState → Adj v x →
     s.2 (VtoP v x p0) = initState.2 (VtoP v x p0))/\
 (∀ v w, Adj v w → activeG v s → activeG w initState → activeG w s).

Section sdeg_conn.

Variables (s:VSt×PSt) (e:E).
Hypothesis s4 : ∀ v : V, inactiveG v s → Poutread nu p0 s.2 v = nseq
 (seq.size (nu v)) false.
Hypothesis s5 : activeG (fste e) s.
Hypothesis s6 : activeG (snde e) s.
Hypothesis s7 : ∀ u v, activeG u s → activeG v s → connect
 (fun x y ⇒ activeG x s && activeG y s && Adj x y ) u v.
Hypothesis s8 : ∀ v:V, activeG v s → s.1 v = initState.1 v.
Hypothesis s9 : ∀ v x:V, activeG v s → Adj v x → s.2 (VtoP v x p0) =
initState.2 (VtoP v x p0).
Hypothesis s11 : ∀ v x:V, inactiveG v initState → Adj v x → s.2
(VtoP v x p0) = initState.2 (VtoP v x p0).
Hypothesis s12 : ∀ v w, Adj v w → activeG v s → activeG w initState →
activeG w s.

Lemma activeNone : ∀ s v, activeG v s → s.1 v = None.

Lemma s1 :∀ x, activeG x s → activeG x initState.

Lemma s3 : ∀ v : V, activeG v s → Poutread nu p0 s.2 v =
nseq (seq.size (nu v)) true.

Lemma DHS_deg_connect_aux1 :
 (mu (DHS(enum V) initState))
  (fun x : VSt × PSt ⇒ B2U (hs_glob_ex s x)) ≤
 (mu (DHS (enum V) initState))
    (fun x : VSt × PSt ⇒ B2U (hs_glob_ex initState x)).

Lemma DHS_deg_connect_aux22 : ∀ v, activeG v s →
 (∀ x, x \in nu v → (activeG x s ↔ activeG x initState)).

Lemma DHS_deg_connect_aux23 : ∀ v, activeG v s →
Pinread nu p0 initState.2 v = Pinread nu p0 s.2 v.

Lemma DHS_deg_connect_aux25 : ∀ v, activeG v s →
 (0 <numberActive (Pinread nu p0 initState.2 v))%nat.

Lemma DHS_deg_connect_aux21 : ∀ f v,
```

```
 (∀ a, inactiveG a s → ∀ x x0, f s (VPupdate nu false a x x0) ==
    f s x0) →
(∀ a x, activeG a s → activeG a initState →
   f s (VPupdate nu false a x initState) == f s (VPupdate nu false a x s)) →
(f s initState == f s s) →
(mu (DHSLoc (Vread initState.1 v) (Poutread nu p0 initState.2 v)
    (Pinread nu p0 initState.2 v))) (fun x0 ⇒ (f s (VPupdate
   nu false v x0 initState)))
==
 (mu (DHSLoc (Vread s.1 v) (Poutread nu p0 s.2 v) (Pinread nu p0 s.2 v)))
      (fun x0 ⇒ (f s (VPupdate nu false v x0 s))).
Lemma inactive_active : ∀ x s', activeG x s' → ¬inactiveG x s'.
Lemma DHS_deg_connect_aux2 : ∀ f,
  (∀ a, inactiveG a s → ∀ x x0, (f s (VPupdate nu false a x x0))
 == (f s x0)) →
(∀ (l:seq (V × (VLab × seq PLab))), (∀ a, a \in l → activeG a.1 s) →
                    (∀ a, a \in l → activeG a.1 initState) →
    f s (foldr (fun x s' ⇒ VPupdate nu false x.1 x.2 s') initState l) ==
    f s (foldr (fun x s' ⇒ VPupdate nu false x.1 x.2 s') s l)) →
(f s initState == f s s) →
 (mu (DHS (enum V) initState)) (fun x : VSt × PSt ⇒ (f s x)) ==
 (mu (DHS (enum V) s)) (fun x : VSt × PSt ⇒ (f s x)).
Lemma DHS_deg_connect :
 (mu (DHS (enum V) s)) (fun x : VSt × PSt ⇒ B2U (hs_glob_ex s x)) ≤
 (mu (DHS (enum V) initState)) (fun x : VSt × PSt ⇒ B2U
   (hs_glob_ex initState x)).
End sdeg_conn.
End initState.
Section whole.
Variable initState : VSt × PSt.
Hypothesis initState1 : ∀ (v:V), (activeG v initState) →
 (Poutread nu p0 initState.2 v) = nseq (seq.size (nu v)) true.
Hypothesis initState2 : ∀ v, (inactiveG v initState) →
 (Poutread nu p0 initState.2 v) = nseq (seq.size (nu v)) false.
Lemma initsub : ∀ (s:VSt×PSt) (e:E),
(∀ v : V,activeG v s → Poutread nu p0 s.2 v= nseq (seq.size (nu v)) true) →
activeG (fste e) s →
activeG (snde e) s →
 (0 < count (fun x : V ⇒ activeG x s && (0 < nactv s x)) (enum V))%nat.
Lemma DHS_deg_whole :
 (0 < count (fun x : V ⇒ activeG x initState && (0 < nactv initState x)%nat)
```

```
(enum V))%nat →
  [1-] hscte
  ≤ (mu (DHS (enum V) initState)) (carac_hs_glob_ex initState).
End whole.
End Handshake.
```

Chapter 29

Library maxmatch_gen

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.
Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import rdaTool_gen.
Require Import hsAct_gen.
Require Import handshake_gen.
Set Implicit Arguments.
Import Prenex Implicits.
```

29.1 Introduction

The maximal matching algorithm is the following: State of a vertex: None (= active) / Some i (matching ith neighbours or nobody if i>deg) State of a port : Bit messages Algorithm : 2 stages, the first to udate the activity, the second to choose a neighbour We consider a graph with active and inactive vertices. Handshakes happen in the active subgraph. At the beginning, every vertex is active and sent 0-messages (saying no choice are made) Only active vertex does the 2 computations. Local Computation 2: If the number of active neighbours is null then state becomes inactive (isolated vertex), send 0 to all neighbours Else, choose a neighbour and send 1 to it and 0 to the other Local Computation 1: If the received message and the sent message is equal to 1 (2 neighbours mutually choosen) then state become inactive (keep messages as before) Else, stay active and send 1 to all neighbours

At the end, every vertex is inactive, handshakes are represented by 1-labelled edges

```
Section MaxMatch.
```

29.2 The graph

```
Context `(NG: NGraph V Adj).

Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).

Let Pt := (@port_finType V Adj).
Variable p0 : Pt.

Let VLabel : eqType := option_eqType nat_eqType.
Let PLabel : eqType := bool_eqType.

Let VState := LabelFunc V VLabel.
Let PState := LabelFunc Pt PLabel.

Definition MMLoc1 (lv:VLabel) (lpout:seq PLabel) (lpin:seq PLabel):
  gen (VLabel × seq PLabel) :=
 if (activeL lv) then
   if (agreed lpout lpin) then
       Greturn _ ( Some (index true lpout) , nseq (seq.size lpout) false)
    else Greturn _ (None, nseq (seq.size lpout) true)
 else Greturn _ (lv, lpout).

Definition MMLoc2 (lv:VLabel) (lpout:seq PLabel) (lpin:seq PLabel):
  gen (VLabel × seq PLabel) :=
 HSLoc lv lpout lpin.

Definition MMStep (seqV : seq V ) (res: VState×PState) :=
 GPStep nu false p0 (MMLoc2::MMLoc1::nil) seqV res.

Definition MMMC (n:nat) (seqV : seq V ) (res: VState×PState) :=
 GPMC nu false p0 n (MMLoc2::MMLoc1::nil) seqV res.

End MaxMatch.
```

Chapter 30

Library maxmatch_op

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".

Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.

Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import op.
Require Import rdaTool_op.
Require Import handshake_gen.
Require Import hsAct_gen.
Require Import hsAct_op.
Require Import maxmatch_gen.

Set Implicit Arguments.
Import Prenex Implicits.
```

30.1 Introduction

This file contains a simulation of the maximal matching algorithm described in maxmatch_gen

```
Section HS.

Variable (rand_t : Type)(get : nat → rand_t → nat × rand_t).
Context (rand : ORandom _ get).

Let VLabel : eqType := option_eqType nat_eqType.
Let PLabel : eqType := bool_eqType.

Definition OMMLoc1 (lv:VLabel) (lpout lpin: seq PLabel)
```

```
 : Op rand_t (VLabel ×seq PLabel) :=
 if (activeL lv) then
   if (agreed lpout lpin) then
       Oreturn ( Some (index true lpout) , nseq (seq.size lpout) false)
    else Oreturn (None, nseq (seq.size lpout) true)
 else Oreturn (lv, lpout).

Definition OMMLoc2 (lv:VLabel) (lpout lpin: seq PLabel)
 : Op rand_t (VLabel ×seq PLabel) :=
 OHSLoc rand lv lpout lpin.

Variables (V:finType) (Adj: rel V).
Context `(NG: NGraph V Adj).

Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).

Let Pt := (@port_finType V Adj).
Variable p0 : Pt.

Let VState := LabelFunc V VLabel.
Let PState := LabelFunc Pt PLabel.

Definition OMMStep (seqV: seq V)(res: VState × PState) :=
 OPStep nu false p0 (OMMLoc2::OMMLoc1::nil) seqV res.

Definition OMMMC (n: nat) (seqV: seq V)(res: VState × PState) :=
 OPMC nu false p0 n (OMMLoc2::OMMLoc1::nil) seqV res.

Section gen.

Lemma OPGMM_eq1 : ∀ (lv:VLabel) (lp1 lp2: seq PLabel) ,
 Opsem _ get rand (MMLoc1 lv lp1 lp2) =
 OMMLoc1 lv lp1 lp2.

Lemma OPGMM_eq2 : ∀ (lv:VLabel) (lp1 lp2: seq PLabel) ,
 Opsem _ get rand (MMLoc2 lv lp1 lp2) =
 OMMLoc2 lv lp1 lp2.

Lemma OPGMM_eq3 : ∀ (seqV: seq V) (res:VState×PState),
 Opsem _ get rand (MMStep nu p0 seqV res) =1
 OMMStep seqV res.

Lemma OPGMM_eq4 : ∀ (n:nat) (seqV: seq V) (res:VState×PState),
 Opsem _ get rand (MMMC nu p0 n seqV res) =1
 OMMMC n seqV res.

End gen.

Section simulation.

Definition OMMStepF (seqV: seq V) (res: (V → VLabel) × (V × V → PLabel)) :=
```

```
 OPFStep nu false (OMMLoc2::OMMLoc1::nil) seqV res.
Lemma OMMF_eq1 : ∀ (seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v n,
 seqV = seqVF →
 (∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   ((OMMStep seqV res n).1).1 v =
   ((OMMStepF seqVF resF n).1).1 v.
Lemma OMMF_eq2 : ∀ (seqV seqVF : seq V)(res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v w n,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
 Adj v w →
   ((OMMStep seqV res n).1).2 (VtoP v w p0) =
   ((OMMStepF seqVF resF n).1).2 (v,w).
 Lemma OMMF_eq3 : ∀ (seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) n,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   (OMMStep seqV res n).2 =
   (OMMStepF seqVF resF n).2.
Definition OMMMCF (n:nat) (seqV: seq V) (res: (V → VLabel) × (V × V → PLabel))
:=
 OPFMC nu false n (OMMLoc2::OMMLoc1::nil) seqV res.
Lemma OMMF_eq4 : ∀ (n:nat) (seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v r,
 seqV = seqVF →
 (∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   ((OMMMC n seqV res r).1).1 v =
   ((OMMMCF n seqVF resF r).1).1 v.
Lemma OMMF_eq5 : ∀ (n:nat) (seqV seqVF : seq V)(res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v w r,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
 Adj v w →
   ((OMMMC n seqV res r).1).2 (VtoP v w p0) =
   ((OMMMCF n seqVF resF r).1).2 (v,w).
```

```
Lemma OMMF_eq6 : ∀ (n:nat)(seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) r,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
    (OMMMC n seqV res r).2 =
    (OMMMCF n seqVF resF r).2.
```

```
End simulation.
```

```
End HS.
```

```
Section simulation.
```

Definition of the graph

```
Inductive V : Type :=
 |v0 : V
 |v1 : V
 |v2 : V
 |v3 : V.
```

```
Definition eqV := (fun x y : V ⇒
 match x,y with
|v0,v0 ⇒ true
|v1,v1 ⇒ true
|v2,v2⇒true
|v3,v3 ⇒ true
|_,_ ⇒ false
end).
```

```
Lemma eqVP : Equality.axiom eqV.
```

```
Canonical V_eqMixin := EqMixin eqVP.
Canonical V_eqType:= Eval hnf in EqType V V_eqMixin.
```

```
Lemma V_pickleK : pcancel (fun v : V ⇒ match v with |v0 ⇒ O |v1 ⇒ 1%nat |v2 ⇒
2 |v3 ⇒ 3 end)
 (fun x : nat ⇒ match x with |0 ⇒ Some v0 | 1 ⇒ Some v1
      |2 ⇒ Some v2 | 3 ⇒ Some v3 | _ ⇒ None end).
```

```
Fact V_choiceMixin : choiceMixin V.
```

```
Canonical V_choiceType := Eval hnf in ChoiceType V V_choiceMixin.
```

```
Definition V_countMixin := CountMixin V_pickleK.
Canonical V_countType := Eval hnf in CountType V V_countMixin.
```

```
Definition venum := (v0:: v1:: v2:: v3:: nil).
Lemma V_enumP : Finite.axiom venum.
```

```
Definition V_finMixin := Eval hnf in FinMixin V_enumP.
```

```
Canonical V_finType := Eval hnf in FinType V V_finMixin.

Lemma card_V : #|{: V}| = 4.

Definition Adj : rel V := (fun x y => match x, y with
 |v0,v1 |v0,v3 |v1,v0 |v1,v2 |v1,v3 |v2,v1 |v2,v3 |v3,v0 |v3,v1 |v3,v2 => true
 | _,_ => false
 end).

Lemma AdjSym : symmetric Adj.

Lemma AdjIrrefl : irreflexive Adj.

Lemma enumV : (enum V_finType) = ([::v0;v1;v2;v3] ).

Context `(NG: NGraph V_finType Adj).

Lemma Nb_enumv0 : Nb_enum Gr v0 = (v1::v3::nil).

Lemma degv0 : (deg Gr v0) = 2.

Definition nu (v: V) : seq V :=
 match v with
   |v0 => [::v1;v3]
   |v1 => [::v0;v2;v3]
   |v2 => [::v1;v3]
   |v3 => [::v1;v2;v0]
end.

Lemma nuAdj_eq : ∀ u w,
Adj u w = (w \in nu u).

Lemma hp0 : Adj (v0,v1).1 (v0,v1).2.

Definition p0 := Port hp0.
```

Definition of the labelling `Let VLabel : eqType := option_eqType nat_eqType.`
`Let PLabel : eqType := bool_eqType.`

```
Definition initV : (LabelFunc V_finType VLabel) :=
finfun (fun x:V => None).

Definition initP : (LabelFunc (@port_finType V_finType Adj) PLabel) :=
 finfun (fun x => true).

Definition init := (initV, initP).

Definition initVF : (V → VLabel) :=
 (fun x:V => None).

Definition initPF : ((V×V) → PLabel) :=
 (fun x => true).

Definition initF := (initVF, initPF).

Lemma init_eq1 : ∀ v, init.1 v = initF.1 v.
```

```
Lemma init_eq2 : ∀ v w,
 Adj v w → init.2 (VtoP v w p0) = initF.2 (v,w).
```

Equivalence

```
Lemma OMMF_eq7 : ∀ v n,
   ((OMMStep my_gen nu p0 (enum V_finType) init n).1).1 v =
   ((OMMStepF my_gen nu [::v0;v1;v2;v3] initF n).1).1 v.

Lemma OMMF_eq8 : ∀ v w n,
 Adj v w →
   ((OMMStep my_gen nu p0 (enum V_finType) init n).1).2 (VtoP v w p0) =
   ((OMMStepF my_gen nu [::v0;v1;v2;v3] initF n).1).2 (v,w).

 Lemma OHSF_eq9 : ∀ n,
   (OMMStep my_gen nu p0 (enum V_finType) init n).2 =
   (OMMStepF my_gen nu [::v0;v1;v2;v3] initF n).2.

Lemma OMMF_eq10 : ∀ n v r,
   ((OMMMC my_gen nu p0 n (enum V_finType) init r).1).1 v =
   ((OMMMCF my_gen nu n [::v0;v1;v2;v3] initF r).1).1 v.

Lemma OMMF_eq11 : ∀ n v w r,
 Adj v w →
   ((OMMMC my_gen nu p0 n (enum V_finType) init r).1).2 (VtoP v w p0) =
   ((OMMMCF my_gen nu n [::v0;v1;v2;v3] initF r).1).2 (v,w).

 Lemma OHSF_eq12 : ∀ n r,
   (OMMMC my_gen nu p0 n (enum V_finType) init r).2 =
   (OMMMCF my_gen nu n [::v0;v1;v2;v3] initF r).2.
```

Computation

```
Let R1 := (OMMStepF my_gen nu [::v0;v1;v2;v3] initF) 6.

Check (R1).

Eval vm_compute in (R1.1.1 v3).
Eval vm_compute in (R1.1.2 (v3,v1)).
Eval vm_compute in (R1.1.2 (v3,v2)).
Eval vm_compute in (R1.1.2 (v3,v0)).

Eval vm_compute in (R1.1.1 v0).
Eval vm_compute in (R1.1.2 (v0,v1)).
Eval vm_compute in (R1.1.2 (v0,v3)).
Eval vm_compute in (R1.1.2 (v0,v0)).

Eval vm_compute in (R1.1.1 v1).
Eval vm_compute in (R1.1.2 (v1,v2)).
Eval vm_compute in (R1.1.2 (v1,v3)).
Eval vm_compute in (R1.1.2 (v1,v0)).

Eval vm_compute in (R1.1.1 v2).
```

```
Eval vm_compute in (R1.1.2 (v2,v1)).
Eval vm_compute in (R1.1.2 (v2,v3)).
Eval vm_compute in (R1.1.2 (v2,v0)).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] R1.1).

Let R2 (n:nat) := (OMMMCF my_gen nu n [::v0;v1;v2;v3] initF) 6.

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (R2 1).1).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (R2 2).1).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (R2 3).1).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (R2 6).1).

End simulation.
```

Chapter 31

Library maxmatch_dist

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat.
Require Import fintype finset fingraph seq finfun bigop choice tuple.
Import Prenex Implicits.
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Add LoadPath "../prelude".
Add LoadPath "../ra".
Add LoadPath "../graph".
Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Require Import Rplus.
Require Import my_alea.
Require Import my_ssr.
Require Import my_ssralea.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import dist.
Require Import rdaTool_gen.
Require Import rdaTool_dist.
Require Import handshake_gen.
Require Import hsAct_gen.
Require Import hsAct_dist.
Require Import maxmatch_gen.
Set Implicit Arguments.
```

31.1 Introduction

This file contains the analysis of the maximal matching algorithm described in maxmatch_gen

```
Section MaxMatch.
```

31.2 Definitions

```
Context `(NG: NGraph V Adj).

Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).

Definition Pt := (@port_finType V Adj).

Definition E := (@edge_finType V Adj).
Variable e0:E.

Definition p0 := (EtoP1 e0).

Definition VState := LabelFunc V VLab.
Definition PState := LabelFunc Pt PLab.

Variable initState : VState × PState.

Hypothesis initState1 : ∀ (v:V), (activeG v initState) →
 (Poutread nu p0 initState.2 v) = nseq (seq.size (nu v)) true.

Hypothesis initState2 : ∀ v, (inactiveG v initState) →
 (Poutread nu p0 initState.2 v) = nseq (seq.size (nu v)) false.

Hypothesis initState3 :
(0 < count (fun x0 : V ⇒ activeG x0 initState && (0 < nactv nu e0 initState x0))
     (enum V))%nat.

Definition DMMLoc1 (lv:VLab) (lpout:seq PLab) (lpin:seq PLab):
  distr (VLab × seq PLab) :=
 if (activeL lv) then
   if (agreed lpout lpin) then
       Munit ( Some (index true lpout) , nseq (seq.size lpout) false)
     else Munit (None, nseq (seq.size lpout) true)
 else Munit (lv , lpout).

Definition DMMLoc2 (lv:VLab) (lpout:seq PLab) (lpin:seq PLab):
  distr (VLab × seq PLab) :=
 DHSLoc lv lpout lpin.

Definition DPRLC1 s x:=
(DPRound nu false p0 s x DMMLoc1).
```

```
Definition DPRLC2 s x :=
(DPRound nu false p0 s x DMMLoc2).

Definition DMMStep (seqV: seq V) (res: VState×PState) :=
  DPStep nu false p0 (DMMLoc2::DMMLoc1::nil) seqV res.

Definition DMMMC (n:nat) (seqV: seq V) (res: VState×PState) :=
  DPMC nu false p0 n (DMMLoc2::DMMLoc1::nil) seqV res.
```

31.3 Equivalence

```
Lemma DPGMM_eq1 : ∀ (lv:VLab) (lp1 lp2: seq PLab) ,
  Distsem (MMLoc1 lv lp1 lp2) =
  DMMLoc1 lv lp1 lp2.

Lemma DPGMM_eq2 : ∀ (lv:VLab) (lp1 lp2: seq PLab) ,
  Distsem (MMLoc2 lv lp1 lp2) =
  DMMLoc2 lv lp1 lp2.

Lemma DPGMM_eq3 : ∀ (seqV: seq V) (res:VState×PState),
  Distsem (MMStep nu p0 seqV res) ==
  DMMStep seqV res.

Lemma DPGMM_eq4 : ∀ (n:nat) (seqV: seq V) (res:VState×PState),
  Distsem (MMMC nu p0 n seqV res) ==
  DMMMC n seqV res.
```

31.4 Lemmas

```
Lemma DMMLoc1_total : ∀ lv lpout lpin,
  Term (DMMLoc1 lv lpout lpin).

Lemma DMMLoc2_total : ∀ lv lpout lpin,
  Term (DMMLoc2 lv lpout lpin).

Lemma DPRLC1_total : ∀ s x,
  Term (DPRLC1 s x).

Lemma DPRLC2_total : ∀ s x,
  Term (DPRLC2 s x).

Lemma DMMStep_total : ∀ s res,
  Term (DMMStep s res).

Definition termB (f: VState× PState) : bool :=
  [∀ v, ~~ activeL (f.1 v)].

Definition DMMStepLV (s: seq V) :=
```

```
 DPStepLV nu false p0 termB (DMMLoc2::DMMLoc1::nil) s.
Lemma DMMStepLV_cont : ∀ s, continuous (DMMStepLV s).
Definition DMMLV (s: seq V) :=
 DPLV nu false p0 termB (DMMLoc2::DMMLoc1::nil) s.
Open Local Scope U_scope.
Open Local Scope O_scope.
Definition numberActiveGlob (resV: VState) :=
#|[set x | activeL (resV x)]|.
Definition hct := [1-] (@hscte V Adj).
Lemma numberActiveGlob_dec1 : ∀ x s,
 (numberActiveGlob x < numberActiveGlob s)%nat →
 ∃ v, activeL (s v) ∧ ~~ activeL (x v).
   Search _ (_.+1 == _ .+1).
Lemma L11_aux : ∀ x,
(mu (DPRLC2 (enum V) x)) (fun x0 ⇒
         if [∀ v, ~~ activeL (x.1 v) ==> ~~ activeL (x0.1 v)]
         then 1
         else 0) == 1.
Lemma L12_aux : ∀ res,
   (mu (DPRLC1 (enum V) res))
     (fun x ⇒
      B2U [∀ v, ~~ activeL (res.1 v) ==> ~~ activeL (x.1 v)]) == 1.
Lemma L1_aux : ∀ res,
 mu (DMMStep (enum V) res)
   (finv (fun x:VState×PState ⇒
  if [∀ v, ~~(activeL (res.1 v)) ==> ~~(activeL (x.1 v))] then 1 else 0)) == 0.
Lemma L21_aux : ∀ s res,
 1 ≤
   (mu (DPRLC1 s res))
     (fun x ⇒ B2U ([set x0 | activeL (x.1 x0)] \subset
   [set x0 | activeL (res.1 x0)])).
Lemma L22_aux : ∀ s res,
 1 ≤
   (mu (DPRLC2 s res))
     (fun x ⇒ B2U ([set x0 | activeL (x.1 x0)] \subset
   [set x0 | activeL (res.1 x0)])).
Lemma is_discrete_DMMLOC1 :
 ∀ (x:VSt×PSt) (v:V),
   is_discrete_s
```

```
    (DMMLoc1 (Vread x.1 v) (Poutread nu p0 x.2 v) (Pinread nu p0 x.2 v)).
Lemma L2_aux : ∀ res,
[∀ v,
          activeG v res ==>
          (Poutread nu p0 res.2 v == nseq (seq.size (nu v)) true)] &&
       [∀ v,
          ~~ activeG v res ==>
          (Poutread nu p0 res.2 v == nseq (seq.size (nu v)) false)] &&
       (0 <
        count (fun x0 : V ⇒ activeG x0 res && (0 < nactv nu e0 res x0)%nat) (enum
V))%nat →
 hct ≤
   (mu (DMMStep (enum V) res)
     (fun x ⇒if (lt_dec (numberActiveGlob x.1) (numberActiveGlob res.1)) then 1
      else 0)).
   Search _ sendChosen seq.size. Search _ count sendChosen.
Lemma L3_aux : ∀ (res:VSt×PSt) (x:V), res.1 x = None →
nactv nu e0 res x = O →
  1 ≤ (mu (DPStep nu false p0 [:: DMMLoc2; DMMLoc1] (enum V) res))
     (fun x : LabelFunc V VLab × LabelFunc port_finType PLab ⇒
      B2U (lt_dec (numberActiveGlob x.1) (numberActiveGlob res.1))).
Lemma Umult_lt_1 : ∀ x y, x < 1 → x × y < 1.
Lemma DMMLV_term :
  Term (DMMLV (enum V) initState).
End MaxMatch.
```

Chapter 32

Library mis_gen

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.
Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import rdaTool_gen.
Set Implicit Arguments.
Import Prenex Implicits.
```

32.1 Introduction

We define the MIS algorithm according three local rules : sends draw numbers to neighbours, if it is the maximal then enters the MIS and send 1, if received 1 then enters the complementary.

```
Section MIS.
```

32.2 The graph

```
Context `(NG: NGraph V Adj).
Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).
Let Pt := (@port_finType V Adj).
```

```
Variable p0 : Pt.

Let VLabel : eqType := option_eqType bool_eqType.
Let PLabel : eqType := nat_eqType.

Let VState := LabelFunc V VLabel.
Let PState := LabelFunc Pt PLabel.

Variable (c:nat).

Definition active (lv: VLabel) : bool :=
 lv == None.

   number between 1 and c+1  Fixpoint getRandSeq (l: seq PLabel) : gen (seq PLabel) :=
 match l with
  |nil ⇒ Greturn _ nil
  |t :: q ⇒ Gbind _ _ (getRandSeq q) (fun l' ⇒ Grandom _ c (fun x ⇒ Greturn _ (x.+1::l')))
 end.

Definition MISLoc1 (lv:VLabel) (lpout:seq PLabel) (lpin:seq PLabel):
  gen (VLabel × seq PLabel) :=
if (active lv) then
  Gbind _ _ (getRandSeq lpin) (fun l ⇒ Greturn _ (lv, l))
else Greturn _ (lv, nseq (seq.size lpin) 0).

Fixpoint supNeigh (lpout:seq PLabel) (lpin:seq PLabel): bool :=
 match lpout,lpin with
  |t::q, t'::q' ⇒ (t' < t)%nat && (supNeigh q q')
  | nil , nil ⇒ true
  | _ , _ ⇒ false
end.

 Definition MISLoc2 (lv:VLabel) (lpout:seq PLabel) (lpin:seq PLabel):
  gen (VLabel × seq PLabel) :=
if (active lv) then
   if (supNeigh lpout lpin) then
       Greturn _ (Some true, nseq (seq.size lpin) 1)
   else Greturn _ (None, nseq (seq.size lpin) 0)
else Greturn _ (lv, nseq (seq.size lpin) 0).

Definition hasRec1 (l: seq PLabel) :=
 has (fun x ⇒ x == 1) l.

 Definition MISLoc3 (lv:VLabel) (lpout:seq PLabel) (lpin:seq PLabel):
  gen (VLabel × seq PLabel) :=
if (active lv) then
   if (hasRec1 lpin) then
       Greturn _ (Some false, nseq (seq.size lpin) 0)
   else Greturn _ (None, nseq (seq.size lpin) 1)
else Greturn _ (lv, nseq (seq.size lpin) 0).
```

```
Definition MISStep (seqV : seq V ) (res: VState×PState) :=
 GPStep nu O p0 (MISLoc1::MISLoc2::MISLoc3::nil) seqV res.

Definition MISMC (n:nat) (seqV : seq V ) (res: VState×PState) :=
 GPMC nu O p0 n (MISLoc1::MISLoc2::MISLoc3::nil) seqV res.

End MIS.
```

Chapter 33

Library mis_op

```
Add LoadPath "../prelude".
Add LoadPath "../graph".
Add LoadPath "../ra".
```

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq.
Require Import fintype path finset fingraph finfun choice tuple.
```

```
Require Import my_ssr.
Require Import graph.
Require Import labelling.
Require Import op.
Require Import rdaTool_op.
Require Import mis_gen.
```

```
Set Implicit Arguments.
Import Prenex Implicits.
```

33.1 Introduction

This file contains the simulation if the MIS algorithm described in mis_gen.

```
Section MIS.
```

```
Variable (rand_t : Type)(get : nat → rand_t → nat × rand_t).
Context (rand : ORandom _ get).
```

```
Let VLabel : eqType := option_eqType bool_eqType.
Let PLabel : eqType := nat_eqType.
```

```
Variable (c: nat).
```

```
Fixpoint OgetRandSeq (l: seq PLabel) : Op rand_t (seq PLabel) :=
 match l with
  |nil ⇒ Oreturn nil
```

```
  |t ::q ⇒ Obind (OgetRandSeq q) (fun l' ⇒ Obind (Orandom c rand) (fun x⇒ Oreturn
(x.+1::l')))
 end.

Definition OMISLoc1 (lv:VLabel) (lpout lpin: seq PLabel)
 : Op rand_t (VLabel ×seq PLabel) :=
 if (active lv) then
  Obind (OgetRandSeq lpin) (fun l ⇒ Oreturn (lv, l))
else Oreturn (lv, nseq (seq.size lpin) 0).

Definition OMISLoc2 (lv:VLabel) (lpout:seq PLabel) (lpin:seq PLabel):
  Op rand_t (VLabel ×seq PLabel) :=
if (active lv) then
   if (supNeigh lpout lpin) then
       Oreturn (Some true, nseq (seq.size lpin) 1)
   else Oreturn (None, nseq (seq.size lpin) 0)
else Oreturn (lv, nseq (seq.size lpin) 0).

Definition OMISLoc3 (lv:VLabel) (lpout:seq PLabel) (lpin:seq PLabel):
  Op rand_t (VLabel ×seq PLabel) :=
if (active lv) then
   if (hasRec1 lpin) then
       Oreturn (Some false, nseq (seq.size lpin) 0)
   else Oreturn (None, nseq (seq.size lpin) 1)
else Oreturn (lv, nseq (seq.size lpin) 0).

Variables (V:finType) (Adj: rel V).
Context `(NG: NGraph V Adj).

Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).

Let Pt := (@port_finType V Adj).
Variable p0 : Pt.

Let VState := LabelFunc V VLabel.
Let PState := LabelFunc Pt PLabel.

Definition OMISStep (seqV : seq V ) (res: VState×PState) :=
 OPStep nu O p0 (OMISLoc1::OMISLoc2::OMISLoc3::nil) seqV res.

Definition OMISMC (n:nat) (seqV : seq V ) (res: VState×PState) :=
 OPMC nu O p0 n (OMISLoc1::OMISLoc2::OMISLoc3::nil) seqV res.

Section gen.

Lemma OgetRandSeq_eq1 : ∀ l,
 Opsem rand_t get rand (getRandSeq c l)=
 OgetRandSeq l.
```

```
Lemma OPGMIS_eq1 : ∀ (lv:VLabel) (lp1 lp2: seq PLabel) ,
 Opsem _ get rand (MISLoc1 c lv lp1 lp2) =
 OMISLoc1 lv lp1 lp2.

Lemma OPGMIS_eq2 : ∀ (lv:VLabel) (lp1 lp2: seq PLabel) ,
 Opsem _ get rand (MISLoc2 lv lp1 lp2) =
 OMISLoc2 lv lp1 lp2.

Lemma OPGMIS_eq3 : ∀ (lv:VLabel) (lp1 lp2: seq PLabel) ,
 Opsem _ get rand (MISLoc3 lv lp1 lp2) =
 OMISLoc3 lv lp1 lp2.

Lemma OPGMIS_eq4 : ∀ (seqV: seq V) (res:VState×PState),
 Opsem _ get rand (MISStep nu p0 c seqV res) =1
 OMISStep seqV res.

Lemma OPGMIS_eq5 : ∀ (n:nat) (seqV: seq V) (res:VState×PState),
 Opsem _ get rand (MISMC nu p0 c n seqV res) =1
 OMISMC n seqV res.

End gen.

Section simulation.

Definition OMISMCF (n:nat) (seqV: seq V) (res: (V → VLabel) × (V × V → PLabel))
:=
 OPFMC nu O n (OMISLoc1::OMISLoc2::OMISLoc3::nil) seqV res.

Lemma OMISF_eq1 : ∀ (n:nat) (seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v r,
 seqV = seqVF →
 (∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
   ((OMISMC n seqV res r).1).1 v =
   ((OMISMCF n seqVF resF r).1).1 v.

Lemma OMISF_eq2 : ∀ (n:nat) (seqV seqVF : seq V)(res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) v w r,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
 Adj v w →
   ((OMISMC n seqV res r).1).2 (VtoP v w p0) =
   ((OMISMCF n seqVF resF r).1).2 (v,w).

 Lemma OMISF_eq3 : ∀ (n:nat)(seqV seqVF : seq V) (res: VState×PState)
 (resF : (V → VLabel) × (V × V → PLabel) ) r,
 seqV = seqVF →
(∀ v, res.1 v = resF.1 v) →
(∀ v w, Adj v w → res.2 (VtoP v w p0) = resF.2 (v,w)) →
```

```
    (OMISMC n seqV res r).2 =
    (OMISMCF n seqVF resF r).2.
End simulation.

End MIS.

Section simulation.

    Definition of the graph

Inductive V : Type :=
 |v0 : V
 |v1 : V
 |v2 : V
 |v3 : V.

Definition eqV := (fun x y : V ⇒
 match x,y with
|v0,v0 ⇒ true
|v1,v1 ⇒ true
|v2,v2⇒true
|v3,v3 ⇒ true
|_,_ ⇒ false
end).

Lemma eqVP : Equality.axiom eqV.

Canonical V_eqMixin := EqMixin eqVP.
Canonical V_eqType:= Eval hnf in EqType V V_eqMixin.

Lemma V_pickleK : pcancel (fun v : V ⇒ match v with |v0 ⇒ O |v1 ⇒ 1%nat |v2 ⇒
2 |v3 ⇒ 3 end)
 (fun x : nat ⇒ match x with |0 ⇒ Some v0 | 1 ⇒ Some v1
       |2 ⇒ Some v2 | 3 ⇒ Some v3 | _ ⇒ None end).

Fact V_choiceMixin : choiceMixin V.

Canonical V_choiceType := Eval hnf in ChoiceType V V_choiceMixin.

Definition V_countMixin := CountMixin V_pickleK.
Canonical V_countType := Eval hnf in CountType V V_countMixin.

Definition venum := (v0:: v1:: v2:: v3:: nil).
Lemma V_enumP : Finite.axiom venum.

Definition V_finMixin := Eval hnf in FinMixin V_enumP.
Canonical V_finType := Eval hnf in FinType V V_finMixin.

Lemma card_V : #|{: V}| = 4.

Definition Adj : rel V := (fun x y ⇒ match x, y with
 |v0,v1 |v0,v3 |v1,v0 |v1,v2 |v1,v3 |v2,v1 |v2,v3 |v3,v0 |v3,v1 |v3,v2 ⇒ true
 | _,_ ⇒ false
```

```
  end).
Lemma AdjSym : symmetric Adj.
Lemma AdjIrrefl : irreflexive Adj.
Lemma enumV : (enum V_finType) = ([::v0;v1;v2;v3] ).
Context `(NG: NGraph V_finType Adj).
Lemma Nb_enumv0 : Nb_enum Gr v0 = (v1::v3::nil).
Lemma degv0 : (deg Gr v0) = 2.
Definition nu (v: V) : seq V :=
 match v with
   |v0 ⇒ [::v1;v3]
   |v1 ⇒ [::v0;v2;v3]
   |v2 ⇒ [::v1;v3]
   |v3 ⇒ [::v1;v2;v0]
end.
Lemma nuAdj_eq : ∀ u w,
Adj u w = (w \in nu u).
Lemma hp0 : Adj (v0,v1).1 (v0,v1).2.
Definition p0 := Port hp0.
```

Definition of the labelling

```
Let VLabel : eqType := option_eqType bool_eqType.
Let PLabel : eqType := nat_eqType.
Definition initV : (LabelFunc V_finType VLabel) :=
finfun (fun x:V ⇒ None).
Definition initP : (LabelFunc (@port_finType V_finType Adj) PLabel) :=
 finfun (fun x ⇒ O).
Definition init := (initV, initP).
Definition initVF : (V → VLabel) :=
 (fun x:V ⇒ None).
Definition initPF : ((V×V) → PLabel) :=
 (fun x ⇒ O).
Definition initF := (initVF, initPF).
Lemma init_eq1 : ∀ v, init.1 v = initF.1 v.
Lemma init_eq2 : ∀ v w,
 Adj v w → init.2 (VtoP v w p0) = initF.2 (v,w).
```

Equivalence

```
Definition c := 2^8.-1.
Lemma OMMF_eq4 : ∀ n v r,
```

```
    ((OMISMC my_gen c nu p0 n (enum V_finType) init r).1).1 v =
    ((OMISMCF my_gen c nu n [::v0;v1;v2;v3] initF r).1).1 v.

Lemma OMISF_eq5 : ∀ n v w r,
  Adj v w →
    ((OMISMC my_gen c nu p0 n (enum V_finType) init r).1).2 (VtoP v w p0) =
    ((OMISMCF my_gen c nu n [::v0;v1;v2;v3] initF r).1).2 (v,w).

 Lemma OMISF_eq6 : ∀ n r,
    (OMISMC my_gen c nu p0 n (enum V_finType) init r).2 =
    (OMISMCF my_gen c nu n [::v0;v1;v2;v3] initF r).2.
```

Computation

```
Let R1 := OMISLoc1 my_gen c.
Let RR1 := (OPFRound nu O [::v0;v1;v2;v3] initF R1).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (RR1 6).1).
Eval vm_compute in ( (RR1 6).2).

Let R2 := @OMISLoc2 nat.
Let RR2 := (OPFRound nu O [::v0;v1;v2;v3] (RR1 6).1 R2).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (RR2 12).1).
Eval vm_compute in ( (RR2 12).2).

 Let R3 := @OMISLoc3 nat.
Let RR3 := (OPFRound nu O [::v0;v1;v2;v3] (RR2 12).1 R3).

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (RR3 12).1).
Eval vm_compute in ( (RR3 12).2).

Let R (n:nat) := (OMISMCF my_gen c nu n [::v0;v1;v2;v3] initF) 6.

Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (R 1).1).
Eval vm_compute in (displayOP nu [::v0;v1;v2;v3] (R 2).1).

End simulation.
```

Chapter 34

Library mis_dist

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat.
Require Import fintype finset fingraph seq finfun bigop choice tuple.
Import Prenex Implicits.
Add Rec LoadPath "$ALEA_LIB/ALEA/src" as ALEA.
Add Rec LoadPath "$ALEA_LIB/Continue".
Add LoadPath "../prelude".
Add LoadPath "../ra".
Add LoadPath "../graph".
Require Export Prog.
Require Export Cover.
Require Import Ccpo.
Require Import Rplus.
Require Import my_alea.
Require Import my_ssr.
Require Import my_ssralea.
Require Import graph.
Require Import labelling.
Require Import gen.
Require Import dist.
Require Import rdaTool_gen.
Require Import rdaTool_dist.
Require Import mis_gen.
Set Implicit Arguments.
```

34.1 Introduction

This file contains the analysis of the MIS described in mis_gen

```
Section MIS.
```

34.2 Definitions

```
Context `(NG: NGraph V Adj).
Variable nu : V → seq V.
Hypothesis Hnu: ∀ (v w:V), (Adj v w) = (w \in (nu v)).
Hypothesis Hnu2: ∀ (v :V), uniq (nu v).
Definition Pt := (@port_finType V Adj).
Variable (p0: Pt).
Let VLab : eqType := option_eqType bool_eqType.
Let PLab : eqType := nat_eqType.
Definition VState := LabelFunc V VLab.
Definition PState := LabelFunc Pt PLab.
Variable (c: nat).
Fixpoint DgetRandSeq (l: seq PLab) : distr (seq PLab) :=
 match l with
  |nil ⇒ Munit nil
  |t ::q ⇒ Mlet (DgetRandSeq q) (fun l' ⇒ Mlet (Random c) (fun x⇒ Munit (x.+1::l')))
 end.
Definition DMISLoc1 (lv:VLab) (lpout:seq PLab) (lpin:seq PLab):
  distr (VLab × seq PLab) :=
 if (active lv) then
  Mlet (DgetRandSeq lpin) (fun l ⇒ Munit (lv, l))
else Munit (lv, nseq (seq.size lpin) O).
Definition DMISLoc2 (lv:VLab) (lpout:seq PLab) (lpin:seq PLab):
  distr (VLab × seq PLab) :=
 if (active lv) then
    if (supNeigh lpout lpin) then
       Munit (Some true, nseq (seq.size lpin) 1)
   else Munit (None, nseq (seq.size lpin) O)
else Munit (lv, nseq (seq.size lpin) O).
Definition DMISLoc3 (lv:VLab) (lpout:seq PLab) (lpin:seq PLab):
  distr (VLab × seq PLab) :=
 if (active lv) then
   if (hasRec1 lpin) then
       Munit (Some false, nseq (seq.size lpin) O)
   else Munit (None, nseq (seq.size lpin) 1)
else Munit (lv, nseq (seq.size lpin) O).
Definition DMISStep (seqV : seq V ) (res: VState×PState) :=
 DPStep nu O p0 (DMISLoc1::DMISLoc2::DMISLoc3::nil) seqV res.
Definition DMISMC (n:nat) (seqV : seq V ) (res: VState×PState) :=
```

DPMC nu O p0 n (DMISLoc1::DMISLoc2::DMISLoc3::nil) seqV res.

34.3 Equivalence

`Lemma` *DgetRandSeq_eq1* : ∀ *l*,
 Distsem (*getRandSeq c l*)=
 DgetRandSeq l.

`Lemma` *DPGMIS_eq1* : ∀ (*lv*:*VLab*) (*lp1 lp2*: *seq PLab*) ,
 Distsem (*MISLoc1 c lv lp1 lp2*) =
 DMISLoc1 lv lp1 lp2.

`Lemma` *DPGMIS_eq2* : ∀ (*lv*:*VLab*) (*lp1 lp2*: *seq PLab*) ,
 Distsem (*MISLoc2 lv lp1 lp2*) =
 DMISLoc2 lv lp1 lp2.

`Lemma` *DPGMIS_eq3* : ∀ (*lv*:*VLab*) (*lp1 lp2*: *seq PLab*) ,
 Distsem (*MISLoc3 lv lp1 lp2*) =
 DMISLoc3 lv lp1 lp2.

`Lemma` *DPGMIS_eq4* : ∀ (*seqV*: *seq V*) (*res*:*VState*×*PState*),
 Distsem (*MISStep nu p0 c seqV res*) ==
 DMISStep seqV res.

`Lemma` *DPGMIS_eq5* : ∀ (*n*:*nat*) (*seqV*: *seq V*) (*res*:*VState*×*PState*),
 Distsem (*MISMC nu p0 c n seqV res*) ==
 DMISMC n seqV res.

`End` *MIS*.