Scientific Annals of Computer Science vol. 26 (2), 2016, pp. 157–186
doi: 10.7561/SACS.2016.2.157

RDA

A Coq Library to Reason about Randomised Distributed Algorithms in the Message Passing Model

Allyx FONTAINE[1], Akka ZEMMARI[2]

Abstract

Distributed algorithms have received considerable attention and were studied intensively in the past few decades. Under some hypotheses on the distributed system, there is no deterministic solution to certain classical problems. Randomised solutions are then needed to solve those problems. Probabilistic algorithms are generally simple to formulate. However, their analysis can become very complex, especially in the field of distributed computing.

In this paper, we formally model in *Coq* a class of randomised distributed algorithms. We develop some tools to help proving impossibility results about classical problems and analysing this class of algorithms. As case studies, we examine the handshake and maximal matching problems. We show how to use our tools to formally prove properties about algorithms solving those problems.

Keywords: Distributed Algorithm, Randomised Algorithm, Analyses, Formal Proof, Proof Assistant.

[1]Université de Guyane, UMR Espace-Dev, France,
E-mail: allyx.fontaine@univ-guyane.fr
[2]Université de Bordeaux, LaBRI UMR CNRS 5800, France,
E-mail: akka.zemmari@labri.fr

1 Introduction

Distributed problems have received considerable attention and were studied intensively in the past few decades. Many works were conducted to study their computation power and/or to design efficient solutions using distributed algorithms. Several problems are such that deterministic solution does not exist in distributed systems. The use of randomisation makes it possible to address those problems. Generally, randomised distributed algorithms are defined in a concise way. However, their analysis remains delicate and complex, which makes their proof difficult. Model checkers give an automatic way to check whether the results of the algorithms verify a certain specification, however it proceeds exhaustively, leading to an explosion of space complexity. An alternative is to use proof assistants. They assist the user to prove properties and certify the proof at its end. The proof assistant *Coq* [20] is powerful to model and prove properties or impossibility results thanks to its higher order logic.

We have developed a *Coq* library that provides tools to reason about (randomised) distributed algorithms in anonymous networks. To illustrate how this library works, we use as case studies simple solutions (not optimal) of two problems: the handshake and maximal matching problems. The handshake problem is a building block for many distributed algorithms especially in synchronous message passing where the sender and the receiver must both be ready to communicate. A communication takes place only if the participant processors are waiting for the communication: this is termed handshake. A solution of the handshake problem gives a matching of the graph. A matching is a subset M of the set of edges of the graph such that no two edges of M have a common vertex. A matching M is said to be maximal if any edge of the graph is in M or has an extremity linked to an edge in M.

1.1 The Theoretical Model

There exists various models for distributed systems depending on the features we allow: message passing model, shared memory model, mobile agents model, communication protocol models, etc. We restrict our study to the standard message passing model for distributed computing in an anonymous network. In this section, we define theoretically the model we would like to implement in *Coq*.

The communication model consists of a point-to-point communication network described by a connected graph $G = (\mathtt{V}, \mathtt{E})$, where the ***vertices*** $\mathtt{V}$

represent network processes and the ***edges*** `E` represent bidirectional communication channels. Processes communicate by ***message passing***: a process sends a message to another by depositing the message in the corresponding channel.

We assume the system is fully ***synchronous***, namely, all processes start at the same time and time proceeds in synchronised rounds. A ***round*** of each process is composed of the following three steps. Firstly, it sends messages to its neighbours ; secondly, it receives messages from its neighbours ; thirdly, it performs some local computations. Note that we consider only ***reliable*** systems: no fault can occur on processes or communication links. This hypothesis is strong but it allows to analyse complexities that give a lower bound for systems based on weaker assumptions (and therefore more realistic).

The network $G = (\mathtt{V}, \mathtt{E})$ is ***anonymous***: unique identities are not available to distinguish the processes. We do not assume any global knowledge of the network, not even its size or an upper bound on its size. The processes do not require any position or distance information. The anonymity hypothesis is often seen for privacy reasons. In addition, each process can be integrated in a large-scale network making it difficult or impossible to guarantee the uniqueness of identifiers.

Each process knows from which channel it receives or to which it sends a message, thus one supposes that the network is represented by a connected graph with a ***port numbering function*** defined as follows (where $\mathcal{N}_G(u)$ denotes the set of vertices of G adjacent to u and $d_G(u)$ its cardinality): given a graph $G = (\mathtt{V}, \mathtt{E})$, a port numbering function ϕ is a set of local functions $\{\phi_u \mid u \in \mathtt{V}\}$ such that for each vertex $u \in \mathtt{V}$, ϕ_u is a bijection between $\mathcal{N}_G(u)$ and the set of natural numbers between 1 and $d_G(u)$.

A ***probabilistic algorithm*** is an algorithm which makes some random choices based on some given probability distributions. A ***distributed probabilistic algorithm*** is a collection of local probabilistic algorithms. Since the network is anonymous, nodes having the same degrees have the same algorithms. We assume that choices of vertices are independent. A ***Las Vegas algorithm*** is a probabilistic algorithm which terminates with a positive probability (in general 1) and always produces a correct result.

1.2 Related Works

Proof assistants are interesting tools to certify correctness because of their flexibility. Particularly, the proof assistant *Coq*, thanks to its higher order

logic, enables to prove impossibility results. For instance, Auger *et al.* [2] certify impossibility results on the mobile robot protocol with *Coq*. This work is followed by the framework designed by Courtieu *et al.* [5] to express mobile robots models, protocols, and proofs. They also certify positive results on protocol for oblivious mobile robots.

Distributed algorithms. The model we study in this paper is the message passing setting. A first step to formally prove correctness of distributed algorithm in this model is to express them in a formal language. Küfner *et al.* [13] develop a methodology based on transition rules to mechanically check proofs of correctness of fault-tolerant distributed algorithms in the asynchronous message passing model. They use the proof assistant Isabelle [17] to formally prove positive results of Consensus algorithms.

Transition systems were also used by Chou [4] who uses the *HOL* proof assistant. He shows the correctness of distributed algorithms, modelised by labelled transition systems where specifications are expressed in terms of temporal logic.

Local computations, represented by relabelling systems, are certified in Loco framework by Castéran and Filou [3]. It consists in a set of libraries on labelled graphs and graph relabelling systems. It allows the user to specify tasks, and to prove the correctness of relabelling systems with reference to these tasks and also impossibility results.

Proofs by refinement of distributed algorithms are developed by Tounsi *et al.*. They derive local computation systems from their formal specification by successive refinement steps within the `Event-B` formalism [6]. Their framework enable simulation by automatically translating the algorithm from Event-B to a code that can be executed into the visualisation tool Visidia [19].

Randomised distributed algorithms. Besides the distributive aspect, we are interested in randomised algorithms. Several approaches take into account the dual paradigm of randomised distributed systems: probabilistic aspect and non-determinism due to the response time that changes from one processor to another. They require models with non-deterministic choice between several probability distributions. These choices can be made by a scheduler or an opponent. Equivalent models are following this idea: probabilistic automata [18], decision making processes of Markov [7]. To specify properties of randomised distributed algorithms, one can use the

temporal logic with probabilistic operators and a threshold.

The *Model Checking* is a tool used to ensure system correction. However, used with probabilities, it leads to an explosion of space complexity. There are methods for reducing the explosion. A qualitative analysis of randomised distributed algorithms is feasible thanks to the model checker PRISM [14]. M. Kwiatkowska et al. [15] use *PRISM* model checker and *Cadence* proof assistant, to obtain automated proofs. Consensus protocol is proved for its non-probabilistic part with *Cadence* and for its probabilistic part with *PRISM*.

J. Hurd et al. [10] formalise in higher order logic the language *pGCL* used to reason on probabilistic choices or choices made by an adversary. They prove the mutual exclusion algorithm of Rabin: consider N processes, sometimes some of them need to access a critical zone; the algorithm consists in electing one of them. However they do not model the processor concurrently but use an interpretation consisting in reducing the number of processes to 1.

In our model, we consider that the algorithm operates in rounds, applying a local algorithm to each vertex. This removes the non-determinism due to the asynchrony. Up to our knowledge, our work is the first to certify impossibility results on distributed problems as well as positive results on randomised algorithms in the distributed message passing setting.

1.3 Our Contribution

We are interested in obtaining formal proofs of distributed algorithms, including randomised algorithms. To do so, we use the *Coq* proof assistant, library *Alea* [1] and plugin *ssreflect* [9]. We first define, in Section 3, the algorithm class of anonymous distributed algorithms according to the model previously described. We explain why our definitions are valid. Our main contribution is the tools we developed to enable the user to analyse anonymous distributed algorithms described in Section 4.

Section 5.2 illustrates how to use those tools by analysing solutions for problems. First, we show that randomisation may be required to solve distributed problems in particular the handshake problem. Hence, we formally prove an impossibility result which is: "there is no deterministic algorithm in this class that solves the handshake problem". This also proves that there is no deterministic algorithm that solves the (maximal) matching problem. Then we implement a solution of the handshake problem and we prove that this solution is correct. Then, we analyse the handshake and the maximal matching problems by proving some probabilistic properties.

We believe this is the first work that deals with the formal modelling and proof of anonymous synchronous randomised distributed message passing algorithms. The layer of distributive and randomised aspect is managed by the library. The only thing the user has to do is to define the local algorithm he/she wants to study. Furthermore, proofs for the analysis of algorithms on this class can be lighten thanks to the general tools available in the library.

Examples such as algorithms that solve the handshake problem and the maximal matching problem are certified. Lemmas and theorems, presented in frame in our paper, are denoted by their name in the *Coq* development available at [8].

2 Preliminaries

Different evaluations of the same probabilistic expression lead to different values. Hence, the probabilistic expression e represents a set of values. To reason about such expressions in a functional language, a solution consists in studying the distribution of this expression rather than its result.

In *Alea*, a probabilistic expression $(e : \tau)$ is interpreted as a distribution whose type is $(\tau \to [0,1]) \to [0,1]$. This monadic type is denoted `distr` τ. We will use the notation $(\mu\ e)$ to represent the associated measure of expression e. Let Q be a property and let $\mathbb{1}_Q$ be its characteristic function. The probability that the result of the expression e satisfies Q is represented by $(\mu\ e)\ \mathbb{1}_Q$.

To construct monadic expressions, *Alea* provides the following functions:

- `Munit a`: returns the Dirac distribution at point `a`;
- `Mlet x = d1 in d2`: evaluates `d1`, links the result to `x` and then evaluate `d2` where `d1` and `d2` are random expressions (not necessarily of the same type);
- `Random n`: from a natural number `n`, this function returns a number between `0` and `n` with a uniform probability $1/(n+1)$.

Most of the proofs presented in this paper are based under both transformations:

```
Lemma Munit_simpl [1]:
∀ (P: τ) (f: τ → [0,1]), (μ (Munit P)) f = (f P).

Lemma Mlet_simpl [1]:
∀ (P: distr τ) (Q: τ → distr τ') (f: τ' → [0,1]),
  (μ ( Mlet x = P in (Q x) )) f = (μ P) (fun x ⇒ (μ (Q x)) f).
```

3 Our Formal Model

Our aim is to give to the user the possibility to define his/her own anonymous randomised distributed algorithms. First, we define the distributed systems that can be studied with our library. Then, we give the syntax that can be used to define randomised distributed algorithms. Once the algorithms are defined, the user can do tests by evaluating them, prove correctness and analyse them. We define semantics and several functions to express the algorithms in order to ensure that they belong to the class we described.

3.1 Formal Distributed Systems

As stated in the introduction, synchronous anonymous message passing model can be represented by a connected graph $G = (\mathtt{V}, \mathtt{E})$ with a port numbering function ϕ. To encode the graph in *Coq*, we use an adjacency function `Adj` that, given two vertices, returns a boolean saying either they are connected or not. In the latter, we mainly reason on graphs denoted $G = (\mathtt{V}, \mathtt{Adj})$. The edge that links two adjacent vertices v and w is denoted by $\{v, w\}$.

We model the **port numbering function** $\phi : \mathtt{V} \mapsto (seq\ \mathtt{V})$ as the ordered sequence of the neighbours of a vertex. For all v, $\phi(v) = [v_1, v_2]$ means that v has two neighbours: the first one is v_1 and the second one is v_2. Two axioms (stated as hypotheses each port numbering function has to ensure) are required: the function ϕ only links adjacent vertices and does not contain duplicated vertices:

```
Hypothesis Hϕ1 : ∀ v w, Adj v w = v ∈ (ϕ w).
Hypothesis Hϕ2 : ∀ v, uniq (ϕ v).
```

Each process sends a message to its neighbour by putting it in the corresponding link. A port, pair of vertices, represents the link whereby a vertex put its message. We define `P` as the set of ports. Thus, if v sends a message to its ith neighbour, it sends its message by the port (v, w) where w is the ith element of the sequence $(\phi\ v)$.

We model the exchange of messages, in a global way, by a **port labelling function** over the graph G. The set of labels over ports is denoted Ψ. A port labelling function $\psi : \mathtt{P} \mapsto \Psi$ maps a port to its associated label. The state of each process is represented by a label $(\lambda\ v)$ associated to the corresponding vertex $v \in \mathtt{V}$. Hence, each vertex has a status represented by a **vertex labelling function** $\lambda : \mathtt{V} \mapsto \Lambda$ where Λ is the set of labels over

the vertices.

Consider $\sigma = (\lambda, \psi)$ the pair of labelling functions which maps a vertex (resp. a port) to its state. The type of such a pair, the **global state** of the graph, will be denoted by `State`.

For instance, see Figure 1, v_2 only distinguishes its four neighbours but it knows nothing about its identity or the identities of its neighbours. We can see, with a global view, that v_5 is the fourth neighbour of v_2 according to ϕ; the fact that v_2 sends a message m to its fourth neighbour consists in replacing the label m_4 in Figure 2 of the port (v_2, v_5) by m.

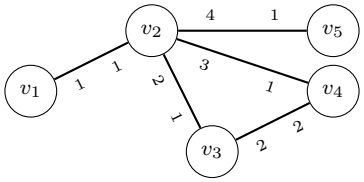


Figure 1: Graph supplied with a port numbering ϕ such that $(\phi\ v_1) = [v_2]$, $(\phi\ v_2) = [v_1, v_3, v_4, v_5]$, $(\phi\ v_3) = [v_2, v_4]$, $(\phi\ v_4) = [v_2, v_3]$, $(\phi\ v_5) = [v_2]$.

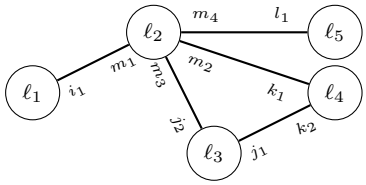


Figure 2: Graph supplied with a vertex labelling function λ and a port labelling function ψ such that $(\lambda\ v_1) = \ell_1$, $(\lambda\ v_2) = \ell_2$, $(\lambda\ v_3) = \ell_3$, $(\lambda\ v_4) = \ell_4$, $(\lambda\ v_5) = \ell_5$ and $(\psi\ (v_2, v_1)) = m_1$, $(\psi\ (v_2, v_4)) = m_2$, $(\psi\ (v_2, v_3)) = m_3$, $(\psi\ (v_2, v_4)) = m_4$, $(\psi\ (v_1, v_2)) = i_1$, $(\psi\ (v_3, v_4)) = j_1$, $(\psi\ (v_3, v_2)) = j_2$, $(\psi\ (v_4, v_2)) = k_1$, $(\psi\ (v_4, v_3)) = k_2$, $(\psi\ (v_5, v_2)) = l_1$.

A processor sends (write a message on the corresponding port) and receives (reads the corresponding port) messages. We define the *writing (resp. reading) area* of a vertex v as the set of port labels it is able to update (resp. to read), that is labels associated to ports of the form (v, w) (resp. (w, v)) where w is a neighbour of v. We define the *local view* of a vertex v as the triple composed by its local state, the sequence of local states of the port in its writing and in its reading area given with the order of ϕ. The local

view of a vertex corresponds to the local information it owns. We define two local functions for a vertex to model received message from all neighbours (`read`) and sent messages to all neighbours (`write`):

- `read:State` $\times$ `V` $\to \Lambda \times$ (`seq` Ψ) $\times$ (`seq` Ψ): consider σ and v, (`read` σ v) returns the local view of v, *i.e.*, the local state of v, the local state of its reading area, and the local state of its writing area, each extracted from σ.

- `write:State`$\times$`V`$\times\Lambda \times$ (`seq` Ψ) $\to$ `State`: consider σ, v, λ, ψ, (`write` σ v λ ψ) returns the new global state obtained from the old one σ such that the local state of v is updated by λ and the one of its writing area by the sequence ψ.

For example, according to Figures 1 and 2, `read` applied to vertex v_2 returns $(\ell_2, [i_1, j_2, k_1, l_1], [m_1, m_3, m_2, m_4])$ and `write` applied to the triplet $(v_2, \ell, [m_1, m, m_2, m_4])$ updates the graph by changing the label ℓ_2 into ℓ and the label m_3 into m.

3.2 Syntax and Semantics

Randomisation appears in local computations of type $(\Lambda \times$ (`seq` $\Psi))$ made by a vertex. Local computations of all vertices of the graph create altogether a random global state of type `State`. We define the inductive type for randomisation `FR`, that will be used to construct random local computations of type `FR` $(\Lambda \times$ (`seq` $\Psi))$ and global random states of type `FR State`. In Haskell [11], monads are structures that represent computations and the way they can be combined. To express randomisation in *Coq* in a monadic form, we state three functions `Freturn`, `Fbind`, and `Frandom`.

```
Inductive FR (B:Type): Type :=
  | Freturn (b:B)
  | Fbind {A :Type}(a:FR A)(f : A → FR B)
  | Frandom (n:nat)(f : nat → FR B).
```

To improve the readability of the code, we define the following abbreviations. Let `stmts` be a statement block, `n` be an integer, and `f` a function:

```
Fbind x (fun v⇒{<stmts>}) ↔ Flet v = x in {<stmts>}.
Frandom n f  ↔ Flet x = (random n) in f.
```

Once one has set out a randomised algorithm thanks to our syntax, one would like to simulate, to prove the correctness or to analyse this algorithm.

For those purposes, we define three semantics that interpret their input, a monad of type `FR`, in an operational, set, or distributional way. The operational semantics `Opsem`, that takes as a parameter a random number generator, is used to evaluate computations. The set monad `Setsem` is used to handle the set of transitional and final results of a randomised algorithm. We can then prove properties of correctness by reasoning on this set. The distributional monad `Distsem` is used to reason about distribution. We define it according to the monad of *Alea* by using the operators `Munit`, `Mlet`, and `Random` [1].

3.2.1 Operational Semantics

We define an operational monad `Op` to evaluate computations. It takes as a parameter a random number generator. The first step consists in defining the three operators for a monad: the type constructor `Op`, the return function `Oreturn`, and the binding function `Obind`. We add the random function `Orandom` after being ensured that it returns a result lesser than its input.

```
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 : forall n x, ( (get n x).1 ≤ n)}.

Definition Orandom (n: nat) {t: Type} {get: nat → t → nat * t}
  (rand: ORandom t get): Op t nat:=
  get n.
```

We give here the semantic definition where `get` is a pseudo-random number generator and `rand` is the random function.

```
Variable (rand_t: Type) (get: nat → rand_t → nat * rand_t).
Context (rand : ORandom rand_t get).

Fixpoint Opsem {B: Type}(m:FR B) : Op rand_t B :=
  match m with
    |Freturn b ⇒ Oreturn b
    |Fbind _ a f ⇒ Obind (Opsem a) (fun x ⇒ (Opsem (f x)))
    |Frandom n f ⇒ Obind (Orandom n rand) (fun x ⇒ Opsem (f x))
   end.
```

To execute our algorithms, we implement a pseudo-random number

generator using the linear congruential method of Lehmer. The pseudo-random number sequence is defined as [12]: $X_{n+1} = (aX_n + c)\ mod\ m$ where m is the modulo, a the multiplier, c the increment and X_0 the seed. We choose standard values: $m = 255$, $a = 137$ and $c = 187$. The seed is a parameter of the generator.

3.2.2 Set Semantics

We define the set monad to handle the set of transitional and final results of a randomised algorithm. We can then prove properties of correctness by reasoning on this set. Here is the semantics:

```
Fixpoint Setsem {B: Type}(m :FR B) : Ensemble B :=
  match m with
    |Freturn b ⇒ fun x ⇒ x = b
    |Fbind A a f ⇒ fun x ⇒ exists y, Setsem a y ∧ Setsem (f y) x
    |Frandom n f ⇒ fun x ⇒ exists i, (i ≤ n) ∧ Setsem (f i) x
  end.
```

3.2.3 Distributional Semantics

We define the distributional monad according to the monad of *Alea*. For this, we use the operator `Munit`, `Mlet`, and `Random` [1].

```
Fixpoint Distsem {B: Type}(m :FR B) : distr B :=
  match m with
    |Freturn b ⇒ Munit b
    |Fbind _ a f ⇒ Mlet (Distsem a) (fun x ⇒ (Distsem (f x)))
    |Frandom n f ⇒ Mlet (Random n) (fun k ⇒ (Distsem (f k )))
  end.
```

3.3 Randomised Distributed Algorithms

We model a distributed algorithm by local algorithms executed by each processes during a round. We represent local algorithms by rewriting rules. From the knowledge of its local view, a vertex v can rewrite its own state and its writing area by applying a local computation of type `FLocT`. A round, `FRound`, is the state obtained from the application of a local computation to all vertices. Note that the updating of the global state is not made concurrently but sequentially. We justify, in Section 4.1, this choice.

Let `LCs` be a sequence of local computations, then a step, `FStep`, corresponds to the application of rounds taking successively as input the local

computations of LCs. The execution of a distributed algorithm with a maximum of n steps is modelled by the function (FMC n LCs s init) where s is an enumeration of V and init is the initial global state.

```
Definition FLocT := Λ → (seq Ψ) → (seq Ψ) → FR(Λ*seq Ψ).
Fixpoint FRound (s:seq V)(res: State)(LCs:FLocT):FR State:=
  match s with     |nil ⇒ Freturn res
    |v::t ⇒Flet s=(FRound t res LCs) in Flet p=(LCs (read res v)) in
           Freturn (write s v p)
  end.
Fixpoint FStep (LCs:seq FLocT)(s:seq V)(res:State):FR State:=
  match LCs with     | nil ⇒ Freturn res
    |a1::a2 ⇒ Flet y = (FRound s res a1) in (FStep a2 s y)
  end.
Fixpoint FMC(n:nat) (LCs:seq FLocT) (s:seq V)(init:State):FR State:=
   match n with      |0 ⇒ Freturn init
     | S m ⇒ Flet y = (FStep LCs s init) in (FMC m LCs s y)
   end.
```

According to the semantics, the result of the distributed algorithm (of type FR State) is either a possible global state that can be obtained from the algorithm with a random number generator (operational semantics); the set of all global states that the algorithm can produce (set semantics); or the distribution of global states resulting from the algorithm (distributional semantics).

To define an algorithm, the user has to write the local algorithm of type FLocT and use the functions FRound, FStep or FMC. According to what he/she wants to study, the user chooses the appropriate semantics.

4 General Results

In this section, we only use the distributional semantics. To ensure readability, let E: FR B be a randomised expression of type B, then instead of writing (Distsem (Freturn E)), we write Dreturn E ((Dreturn E) is a simplification of (Distsem (Freturn E))). Similarly, we introduce new functions beginning with D (instead of F) as distributional: Dlet, DRound, etc. First, we show why our model is valid. Then we described the following proof techniques: permutability, composition, non-null probability and termination. Let LC be a local computation and LCs be a sequence of local computations.

4.1 Validity of Our Model

We have seen that the sending of messages is implemented by updating the state σ a vertex after another. The function `read` applied to a vertex v only gives the information about the reading area of v. The function `write` applied to a vertex v updates the global state by only rewriting the writing area of v. They are both deterministic. As the writing areas are pairwise disjoint (relabellings do not overlap), two calls of `write`, each applied to a different vertex, permute. It is equivalent to apply this function first to a vertex v and then to a vertex w or vice-versa.

```
Lemma write_comm: ∀ v w, v ≠ w →
  (write (write σ w c2) v c1) = (write (write σ v c1) w c2).
```

As our system is distributed, several vertices can relabel their writing area at the same time. However, it is simpler to reason on such algorithm if they are sequential. That is why we have implemented the global function with parameter the sequence of vertices `(enum V)`. It describes sequentially the application of the local function that would be applied simultaneously on all the vertices. We then have to show that the results obtained from the application of the local algorithm on vertices in a sequential way do not depend on the order of the vertices on which it is applied. This property is ensured thanks to the permutability of function `write`. We have ensured that the result will be the same than the one obtained if vertices would execute this algorithm at the same time by proving the lemma `DRoundCommute3`.

`Lemma DRoundCommute3:` Let σ be a global state of G and `LC` be a discrete local computation (*i.e*, rewritable into a sum). Let lv be a sequence of vertices of G. Let lv' be a permutation of lv, then:

```
DRound lv σ LC = DRound lv' σ LC.
```

4.2 Permutability

We have proved (see Lemma `DRoundCommute3`) that for all sequences s_1 and s_2 such that s_2 is a permutation of s_1, `(DRound` s_1`)` has the same output as `(DRound` s_2`)`. Therefore, if we consider the labelling σ and the sequence $(v :: s)$ where v is a vertex and s is a sequence of vertices where v does not appear, it is the same to include the result of the local function applied to v in `(DRound` s σ`)` than to include the result of `(DRound` s σ`)` into the result of the local

function applied to v. Formally:

```
Lemma DRoundcons2: ∀ s σ, (∀ w, is_discrete (LC (read σ w))) →
  DRound s σ LC =
  if (null s) then σ else Dlet c = LC (read σ (head s) ) in
     write (DRound (tail s) σ LC) (head s) c
```

The proof of this lemma is based on the discretisation of the measure of the local computation LC, that is its rewriting into a finite sum.

4.3 Composition

A way to prove properties on function DRound is to proceed by induction on the sequence of vertices. For example, we have proved that the function terminates with probability one, assuming that LC terminates:

```
Lemma DRound_total: ∀ σ s, (∀ v, Term (LC (read v))) →
(μ (DRound s σ LC)) 𝕀 = 1.
```

Proof: By induction on s. Assume that the property is checked for s', we show that it is verified for $s = (v :: s')$, that is:
∀ σ s' v,(μ (DRound v :: s' σ)) 𝕀 = 1.

Using the definition of DRound, this expression becomes:

```
∀ σ s' v, (μ (Dlet r = DRound s' σ in
Dlet c = LR (read σ v) in write r v c)) 𝕀 = 1.
```

Transformations of Lemmas Munit_simpl and Mlet_simpl give:

```
∀ σ s' v, (μ  (DRound s' σ)) (fun r ⇒ (μ ( LR (read σ v))) 𝕀) = 1.
```

We assumed that LC terminates, then as the definition of the characteristic function $\mathbb{1}$ is (fun $x \Rightarrow 1$), from the following equation, yields the result:

```
∀ σ v, (μ DRound s' σ) 𝕀 = 1.
```

□

A general technique appeared in this proof. The expression can be decomposed into the measure of one vertex and the measure for the remaining. Therefore, if we want to prove a property about a vertex v, we can use this technique.

4.4 Non-null Probability

The probability that an event occurs in a randomised algorithm is not null if there is a possible execution of the algorithm whereby this event is verified.

Therefore, to show that a probability is non-null, it suffices to highlight a witness.

`Lemma proba_not_null:` Let A be a randomised algorithm and E an event. Let t be a witness, if $(\mu\ A)\ \mathbb{1}_{.=t} > 0$ and $(E\ t) > 0$ then $(\mu\ A)\ \mathbb{1}_{(E\ t)} > 0$.

4.5 Termination

A randomised distributed algorithm repeats a step until a certain property is verified by the labelling graph. In general, this property is that all the vertices stop to interact with others, *i.e.* until all vertices are inactive. That leads us to consider the algorithm with a property of termination `TermB`.

```
Fix DLV (sV: seq V)(σ: State) (LCs: seq DLocT) (TermB: State → bool) :
distr (State) :=    if (TermB σ) then Dreturn σ
   else Dlet r = (DStep LCs sV σ) in DLV sV r LCs TermB
```

In *Coq* we need to highlight a variant which decrements at each round in order to prove the termination. However there exists some algorithms which terminate with probability 1 but in which some executions could possibly be infinite. To deal with this kind of programs, there is, in *Alea*, a tool to handle limits of sequences of distributions. Hence, when a recursive function is introduced, we interpret it as a fix point and then compute the least upper bound of the sequence.

`Lemma termglobal:` For all randomised updating of a global state to another `rd : State → distr State`, for all global state σ, for all ended property `TermB`, for all variant `(cardTermB: State → nat)`, for all real c between 0 and 1 and for all state property `(PR:State→bool)`, if:

1. ∀ `s, Term (rd s)`
2. ∀ `s, cardTermB s = 0 → TermB s = true`
3. $0 < c$
4. ∀ `s, 0<cardTermB s →` `PR s →` `c≤` μ `(rd s)` $\mathbb{I}_{(cardTermB\ .<cardTermB\ s)}$
5. ∀ `s, PR s →` μ `(rd s)` $\mathbb{I}_{(cardTermB\ s<cardTermB\ .)} = 0$
6. ∀ `s f, PR s →` μ `(rd s)` $\mathbb{I}_{PR.\ \wedge\ f.}$ `=` μ `(rd s)` $\mathbb{I}_f$
7. `PR` σ

then: `Term (fglobal rd TermB` σ`)`.

From this lemma, we obtain the Lemma `DPLV_total` saying that the function `DLV` terminates by taking as input for the state transformation (`rd`) the function `(DStep LCs (enum V))`. Thus, to prove that a Las Vegas algorithm terminates with probability 1, it suffices to show that the probability for a certain variant (such that, if it is null, it implies the termination) to

decrement is non-null and to increase is null after a step `(DStep LCs (enum V))`. The property `PR` specify the global states with a property always true: the two last hypotheses mean that it has no impact on the probability computations and that it is verified by the initial state.

5 Applications

As a case study, we focus on the Handshake problem. We first prove an impossibility result that implies randomisation is required. We define a randomised solution and we prove its correctness. Then, we analyse this solution. As a generalisation of this problem, we analyse a solution to the maximal matching problem.

5.1 Correctness of an Handshake Solution

5.1.1 Handshake Specification

In this subsection, we specify the handshake problem by defining what specifications an algorithm (structure `hsAlgo`) solving this problem has to ensure. We assume important hypotheses on the graph: it must contain at least an edge (otherwise no handshake can occur) and the graph is uniform. We define an algorithm that solves the handshake problem as a structure containing:

- `HsR`: a sequence of local computations (that each node executes in successive rounds);
- `HsP`: a local handshake function (from a local view of the vertex, this function returns the port which the vertex is in handshake with or `None` if it is not in handshake);
- `HsI`: an initial state for the graph.

Hypotheses on the above components are the following:

- `HsI1`: the initial state is *consistent*, *i.e.*, for each v, if v is in handshake with one of its neighbours (say w), then w is also in handshake with v;
- `HsI2` : the initial state is *uniform*, *i.e.* each vertex has the same label and each port also;

- `HsP1` : the global handshake function (obtained from `HsP`) applied to a vertex v returns numbers lesser than the degree of v;

- `HsRind`: consistency is preserved by a step of the algorithm.

```
Record hsAlgo Λ Ψ :={ (** Local rules *) HsR:seq (FLocT Λ Ψ);

(** Handshake function *)
HsP:Λ → seq Ψ → seq Ψ → option nat;

(** Initial state *)
HsI:∀ V Adj G, State Adj;

(** Hypotheses *)
HsI1:∀ V Adj G δ (Hδ1:∀ v w, Adj v w = w ∈(δ v)) (Hδ2:∀ v, uniq(δ v)),
  consistent δ HsP (HsI G);

HsI2:∀ V Adj G,   Uniform (HsI G);

HsP1:∀ V Adj G δ (Hδ1:∀ v w, Adj v w=w ∈(δ v) (Hδ2:∀ v, uniq(δ v)) σ v i,
  (hsPortR δ HsP σ v) = Some i → i < (deg G v);

HsRind:∀ V Adj G δ (Hδ1:∀ v w,Adj v w = w ∈(δ v)) (Hδ2:∀ v,uniq(δ v)),
  Stable (fun σ ⇒ consistent δ HsP σ) (nextState HsR δ).
}.
```

We define a handshake between two vertices via the property `hsBetween`. The existence of such a handshake is defined in `hsExists`. The property `hsEventually` specifies whether a handshake occurs or not from the initial state. The aim of this algorithm is to realise handshakes (`hsRealisation:Λ Ψ (A: hsAlgo Λ Ψ)`), *i.e.*, for any graph, there is an execution in which one reachable state contains a handshake.

Let `s` be the handshake function (function that maps each vertex v to `None` to specify that the vertex is not in handshake and `Some` w to specify that there is a handshake between v and w).

```
Definition hsBetween s v w:= (Adj v w)&&(s v==Some w)&&(s w==Some v).
Definition hsExists s := ∃ v w, hsBetween s v w.

Definition hsEventually LR δ hsPort initState:=
  ∃ σ,  reachFrom (nextState LR δ) initState σ ∧
  hsExists(assNeigh hsPort δ σ).

Definition hsRealisation Λ Ψ (A: hsAlgo Λ Ψ) :=
  ∀ V Adj G δ (Hδ1:∀ v w, Adj v w = w ∈ (δ v)) (Hδ2:∀ v, uniq(δ v)),
   hsEventually (HsR A) δ (HsP A) (HsI A G).
```

5.1.2 Impossibility Result

We have seen that the difference between deterministic algorithms and randomised algorithms is the use of `random`. We show in this section the interest of randomised algorithms by proving that there is no deterministic algorithm that solves the handshake problem. The property `Deterministic {B:Type}(e : FR B)` is defined as below. The algorithms are expressed via computational rules, so we defined a property `Adet (l:seq FLocT)` verifying that all the computational rules are deterministic.

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

Fixpoint Adet (l:seq FLocT) :=
  match l with
   |nil ⇒ True
   |t::q ⇒ ∀ lv lp₁ lp₂, Deterministic(t lv lp₁ lp₂) ∧ (Adet q)
  end.
```

We have proven the following lemma: in our model, there is no deterministic distributed algorithm that solves the handshake problem for any graph G.

```
Lemma NotReal: ∀ Λ Ψ(A: hsAlgo Λ Ψ),
  Adet(HsR A) →  ∼(hsRealisation A).
```

The proof is based on the stability of the uniform view in the graph. Let $G = (\mathtt{V}, \mathtt{Adj})$ be a simple undirected graph supplied with a port numbering function ϕ (verifying the two hypotheses `Hϕ1` and `Hϕ2`). We define a uniform view as follow: for each pair of vertices with the same degree, their reading areas are equal.

```
Definition UniformView V Adj G δ Hδ1 Hδ2 σ :=
  ∀ v w, |δ v| = |δ w| → read σ v = read σ w.
```

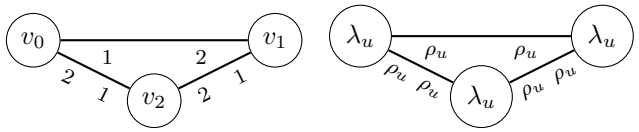
We now detail the relevant steps of the proof of Lemma `NotReal`.

Lemma 1 *In our model, there is no deterministic distributed algorithm that solves the handshake problem for any graph G supplied with the port numbering ϕ.*

The development based on the same name used in the proof is available on the web page of the library *RDA* [8].

Proof: We want to prove that, regardless of the graph and its supplied port numbering, there is no deterministic distributed algorithm that solves the handshake problem. In this purpose, we proceed by contradiction assuming that there exists such an algorithm.

Let A be an algorithm that solves the handshake problem irrespective of the graph and the port numbering. We then prove that there exists a labelled graph G and a port numbering ϕ such that this algorithm does not produce a handshake, which is a contradiction. For this, we consider the graph described in Figure 3(a). We show that whatever is the reached state, no handshake can occur.



(a) $(\mathsf{V}, \mathsf{E}, \phi)$ (b) Init Global State

Figure 3: Witness for the impossibility proof

- Initialisation. By hypothesis, the initial state is uniform and consistent.
- Stability. We prove (*Lemma UniformViewStablehs*) that the uniform view is preserved by a step of the algorithm. Hence we obtain an invariant.
- Induction. Let σ be a state such that it has a uniform view and it is consistent. We prove that (*Lemma NoHs*) for every local view of each vertex, `hsPort` is equal to `None`. The proof is based on the fact that if a vertex v makes a handshake with another one, saying the first, then this first has to be synchronised with v. The port numbering does not allow this configuration.
- Conclusion. In summary, we know that the uniform view and the consistency are invariant. We have proved that for an arbitrary state σ which is consistent and uniform, there is no handshake. We can deduce from Lemma `reachInd` that no handshake can be done during a round and then by the execution of this class of algorithms.

□

5.1.3 The Randomised Algorithm

The algorithm `randHSLoc` is defined as follows: each vertex v chooses uniformly at random one of its neighbours $c(v)$, sends 1 to $c(v)$ and 0 to the others. There is a handshake between v and $c(v)$ if v receives 1 from $c(v)$. Vertex labels are of type `option nat` and those for ports are of type `bool`. We consider a graph $G = (\mathtt{V}, \mathtt{Adj})$ supplied with a port numbering ϕ (verifying the two hypotheses `Hϕ1` and `Hϕ2` in Section 3.3). We denote by `State` the type of the global state of the graph (given by the two labelling functions).

To define the local algorithm, we use the function `(randSendChosen` n l`)` that returns a boolean sequence of size the size of l where each component takes the value 0 except the nth that takes value 1. Vertex labels do not interfere. The simulation of this algorithm is given in the following subsection.

```
Definition randHSLoc (λ:Λ) (ψ_out ψ_in:seq Ψ) : FR (Λ × seq Ψ) :=
 match |ψ_in| with |0 ⇒ Freturn (None, nil)       (*isolated vertex*)
   |S n ⇒ Flet k=(random n) in Freturn(None,randSendChosen(k+1) ψ_in)
 end.
```

We define a round for the handshake as:

```
Definition randHSRound (σ: State):=
   FRound δ σ randHSLoc.
```

5.1.4 Simulation of the Randomised Algorithm

Thanks to the operational semantics, we simulate the algorithm `randHSLoc`. The simulation is launched for the graph of Figure 4. Figure 4 precises the port numbering and Figure 5 the local states of the ports. We simulate the algorithm with a seed equal to 6. The obtained result (see Figure 6) corresponds to what we expect. We can see that there is a handshake between v_1 and v_2.

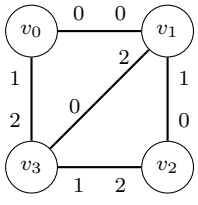


Figure 4: Port numbering.

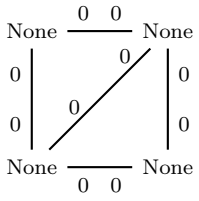


Figure 5: Initial state.

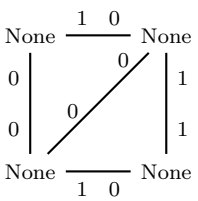


Figure 6: Result state.

5.1.5 Correctness of the Randomised Algorithm

Formal Definition of the Randomised Algorithm. To define formally our algorithm, we first define the components (`randHsR`, `randHsP`, `randHsI`). The rule sequence `randHsR` corresponds to a single local rule `randHSLoc`. The function `randHsP` returns `None` if the vertex is not in a handshake or `Some i` if the vertex is in handshake with its ith neighbour. For this, we define the function `agreed` that returns true if the rank of label 1 in the writing area corresponds to the rank of label 1 in the reading area (*i.e.*, if there exists an edge labelled on those two ports with 1). The initial state `randHsI` is the one where all labels are valued at `None` and all the labels of the ports at 0.

```
Definition randHsR : FLocT Λ Ψ := (randHSLoc::nil).

Definition randHsP λ ψ_out ψ_in : option nat :=
  if (agreed ψ_out ψ_in) then Some (index true ψ_out)
  else None.

Definition randHsI V Adj : State Adj:=
  ([ffun v⇒None],[ffun p⇒false]).
```

We prove the properties that every handshake algorithm must satisfy: consistency of the initial state (`randHSI1`), uniformity of the initial state (`randHsI2`), domain of the handshake function (`randHsP1`) and stability of consistency by a computation step (`randHsRind`).

```
Lemma rda.handshake_rand.randHsI1 V Adj G δ
 (Hδ1: ∀ v w,Adj v w=w ∈(δ v)) (Hδ2: ∀ v,uniq(δ v)) p0:
  consistent randHsP (randHsI G) δ.

Lemma rda.handshake_rand.randHsI2 V Adj G:
   Uniform (randHsI G).

Lemma rda.handshake_rand.randHsP1 V Adj G δ
 (Hδ1: ∀ v w,Adj v w=w ∈(δ v)) (Hδ2: ∀ v,uniq(δ v)) p0 σ v:
  hsPortR randHsP σ v = Some i → i < deg G v.

Lemma rda.handshake_rand.randHsRind V Adj G δ
 (Hδ1:∀ v w,Adj v w=w ∈(δ v)) (Hδ2:∀ v,uniq(δ v)) p0:
   Stable (fun σ ⇒ (consistent randHsP σ δ)) (nextState randHsR δ).
```

We then build the algorithm:

```
Definition randhs: (hsAlgo Λ Ψ) :=
 (Build_hsAlgo randHsI1 randHsI2 randHsP1randHsRind).
```

Finally, we prove that the only hypothesis that differs is the determinism:

```
Lemma NonADet: ~ Adet (HsR randhs).
```

Matching Invariant. A solution of the handshake problem gives a matching of the graph. Matching comprised two aspect: adjacency and symmetry. Let $G = (\mathtt{V}, \mathtt{Adj})$ be a simple undirected graph. Let s be a function of type $\mathtt{V} \to \mathtt{seq\ V}$ that maps each vertex v to `None` to specify that the vertex is not in handshake and `Some` w to specify that there is a handshake between v and w. A matching is defined as follow:

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

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

Definition matching s := (synchAdj s) ∧ (synchSym s).
```

We prove in Lemma `randHsInvariant_matching` that `randhs` always produces a `matching`. This property is easily deduced from the consistency of the algorithm.

```
Lemma randHsInvariant_matching:
Invariant (fun σ ⇒ matching
                     (fun v ⇒ assNeigh(HsP randhs)
                     v σ δ))
          (nextState (HsR randhs) δ)
          (HsI randhs G).
```

Handshake occurring. We prove (Lemma `Real`) that there exists at least an execution of the algorithm that realises a handshake.

```
Lemma Real : hsRealisation randhs.
```

To prove this lemma, as there exists an edge $\{u, v\}$ in the graph, we consider the labelling such that the two ports of this edge are labelled 1 and the other port of u and v are labelled 0. For the other vertices, by default the first port is labelled 1 and the other 0. We show that this labelling can be obtained from an execution of the algorithm `randhs` and that this labelling contains a handshake (on the edge $\{u, v\}$).

Handshake Analyse. We prove (Lemma `rand_hsexists`) that the probability (measure μ on distributional semantics) to obtain at least one handshake

(event `hsExists`) is greater than the constant $1 - e^{-1/2}$. We make a deeper study in the next section.

```
Lemma rand_hsexists: ∀ σ,
1 − e^{−1/2} ≤  μ (Distsem (randHSRound σ))
             (fun s⇒hsExists(assNeigh randHsP δ s)).
```

5.2 The Handshake Algorithm in *Coq*

The local computation we consider here (`DHSLoc`) is similar to `randHSLoc` except that we applied it only on active vertices, that is on the active subgraph.

```
Definition DHSLoc (λ:Λ) (ψ_out ψ_in: seq Ψ): dist (Λ*seq Ψ) :=
  if (active λ) then
    match (numberActive ψ_in) with
      |O ⇒ Dreturn (Some |ψ_out|,nseq |ψ_out|) false)
      |S n ⇒ Dlet k = (Random n) in
                Dreturn (λ,sendChosen k.+1 ψ_in)
    end
  else Dreturn (λ,ψ_out).

Definition DHSRound (sV: seq V)(σ:State) := DRound sV σ DHSLoc.
```

The local computation of the handshake for a vertex v consists in choosing a number k between 0 and $d(v) - 1$ via the function random and in labelling 1 the port linked to the chosen neighbour and in labelling 0 the other ports. Thus, the generated state is obtained from a `State` σ by changing the value of the ports linked to v by $\mathbf{0}$, except the port (v, w) put at $\mathbf{1}$ where w is the kth port of v. Active vertices are required to construct the maximal matching of the next section. We denote by `DHS` the global algorithm based on the local algorithm `DHSLoc`.

In the following sections, we prove specific results about the algorithm `DHSLoc`. The proofs are facilitated thanks to the general results stated in Section 4.

5.2.1 Permutability

Lemma `DRoundcons2` directly implies permutability. Indeed, the hypothesis of this lemma is based on the discretisation of the measure of the local function, that is its rewriting into a finite sum. The measure of our local function is discretisable since it is directly defined from `random` whose distribution is a finite sum.

5.2.2 Composition

From the general Lemma `DRound_total`, we have seen that if we want to prove a property about a vertex v, an expression can be decomposed into the measure of one vertex and the measure for the remaining. To illustrate this fact, we prove Lemma `DHS_degv_global`. Let $P(v,w)$ be the property *"v chooses w"*. We denote by `sV` the sequence of vertices in the graph.

`Lemma DHS_degv_global:` $\forall\ G\ \sigma\ \{v,w\},\ (\mu$ `(DHS sV` σ `))` $\mathbb{I}_{P(v,w)} = 1/d(v)$.

As the order is irrelevant, sV can be rewrited into $v :: (sV \backslash v)$, we can apply the composition technique.

5.2.3 Analysis of the success

Let $\mathcal{HS}(e)$ denote the event "there is a handshake on the edge e". We define $\mathcal{H}(e)$ as the characteristic function of $\mathcal{HS}(e)$, *i.e.*, a boolean set to **1** if there is a handshake on e and to **0** otherwise.

The goal of our establishment of a model is to write the formal proof of results from [16]. Mainly, we prove formally this main theorem:

`Theorem DHS_deg:` $\forall$ `sV` σ, μ `(DHS (enum V)` σ`)` $(\exists\ e,\ \mathcal{H}(e)) \geq 1 - e^{-1/2}$.

We now detail the relevant steps of the proof of Theorem `DHS_deg`. Thanks to the formal proof in `Coq`, we realised that a proof obligation that did not appear in the original proof was required: the probability that no handshake occur in any edges is not null. To prove it, we use the general result of Lemma `proba_not_null`, the witness is a spanning tree in a connected component of the graph. In the sequel, we will use the symbol $\mathbb{P}$ as an abbreviation of the distribution $\mu($`DHS (enum V)` $\sigma)$. Theorem `DHS_deg` becomes:

Theorem 1 *"The probability $\mathbb{P}(\exists\ e \in \mathsf{E},\ \mathcal{H}(e))$ to have at least one handshake after the execution of `DHS` is greater or equal than the constant $1-e^{-1/2}$."*

Proof: From one hand:

$$\mathbb{P}(\exists\ e \in \mathsf{E},\ \mathcal{H}(e)) \quad = 1 - \mathbb{P}(\textstyle\prod_{j=1}^{m}\ \overline{\mathcal{H}(e_j)}).$$

On the other hand:

$$\begin{aligned}
\mathbb{P}(\textstyle\prod_{j=1}^{m}\ \overline{\mathcal{H}(e_j)}) &= \textstyle\prod_{i=1}^{m}(1 - \mathbb{P}(\mathcal{H}(e_i) | \prod_{j=1}^{i-1} \overline{\mathcal{H}(e_j)})) \\
&\leq \textstyle\prod_{i=1}^{m}(1 - \mathbb{P}(\mathcal{H}(e_i))) \\
&\qquad \text{(cf. Lemma 2)} \\
&\leq \textstyle\prod_{i=1}^{m}(1 - \frac{\sum_{j=1}^{m} \mathbb{P}(\mathcal{H}(e_j))}{m}) \\
&\leq (1 - \tfrac{\frac{1}{2}}{m})^m \ (*) \\
&\leq e^{-\frac{1}{2}}.
\end{aligned}$$

Remark 1 *(*) This step consists in proving:* $\sum_{j=1}^{m} \mathbb{P}(\mathcal{H}(e_j)) \leq \frac{1}{2}$.
And $\sum_{j=1}^{m} \mathbb{P}(\mathcal{H}(e_j)) = \sum_{j=1}^{m} \frac{1}{d(e_j^1)*d(e_j^2)}$ *where* $e_j = (e_j^1, e_j^2)$.
The well-known result $\sum_{j=1}^{m} \frac{1}{d(e_j^1)*d(e_j^2)} \geq \frac{1}{2}$ *leads us to conclude.*

□

The following lemma (Lemma 2) is the proof of the second step described above.

Lemma 2 *(prelude.my_alea.Mcond_prodConjBound).*
Let $\delta(e, e')$ *be the boolean whose value is* 1 *if the edges* e *and* e' *are not adjacent and* 0 *otherwise.*
If the following hypotheses hold

1. $\forall i \in 1..m,\ \mathbb{P}(\prod_{j=i+1}^{m} \overline{\mathcal{H}(e_j)}) \neq 0$
2. $\forall i \in 1..m,\ \mathcal{HS}(e_i)$ *and* $\bigwedge_{j=i+1|\delta(e_i,e_j)}^{m} \overline{\mathcal{HS}(e_j)}$ *are independent*
3. $\forall i \in 1..m, \mathcal{H}(e_i) * \prod_{j=i+1|\delta(e_i,e_j)}^{m} (\overline{\mathcal{H}(e_j)}) = \mathcal{H}(e_i)$

then for any i *in* $1..m$ *and any edge* e,

$$\mathbb{P}(\mathcal{H}(e)) \leq \mathbb{P}(\mathcal{H}(e) | \prod_{j=i+1}^{m} \overline{\mathcal{H}(e_j)}).$$

Proof: The proof of this lemma is based on a partition of E: edges which are adjacent to e and those which are not:
let $A = \prod_{j=i+1|\delta(e_i,e_j)}^{m} \overline{\mathcal{H}(e_j)}$ and $B = \prod_{j=i+1|\sim\delta(e_i,e_j)}^{m} \overline{\mathcal{H}(e_j)}$.

We can write, thanks to hypothesis 1., the expression: $\frac{\mathbb{P}(B)}{\mathbb{P}(A*B)}$. Then, we have proved that:

$$1 \leq \frac{\mathbb{P}(B)}{\mathbb{P}(A * B)}.$$

Hypothesis 2. leads us to:

$$\mathbb{P}(\mathcal{H}(e)) \leq \frac{\mathbb{P}(\mathcal{H}(e) * B)}{\mathbb{P}(A * B)}.$$

Finally, hypothesis 3. gives us the result:

$$\mathbb{P}(\mathcal{H}(e)) \leq \frac{\mathbb{P}(\mathcal{H}(e) * A * B)}{\mathbb{P}(A * B)}.$$

□

In Lemma 2, a non-null probability is required as an hypothesis to achieve the proof (hypothesis 1). Lemma 3 is a proof that this hypothesis is checked. From the general Lemma proba_not_null, it appears that we only need to highlight a witness that satisfies the expression.

Lemma 3 *(rda.handshake.hs1).*
For any subset $S = \{e_{i+1}, \ldots, e_m\}$ of edges, the probability that no handshake occurs in S is not null, that is:

$$\forall i \in 1..m,\ \mathbb{P}(\prod_{j=i+1}^{m} \overline{\mathcal{H}(e_j)}) \neq 0.$$

Proof: Consider the set of edges $\mathsf{E} = e_1, \ldots, e_m$.

To show that this probability is not null, we highlight a witness which is a possible execution of the algorithm and in which there is no handshake on the edges $e_{i+1}, \ldots, e_m$ (Lemma `proba_not_null`) where $i \in 1..m$.

We proved that it is always possible to construct a parent function representing a rooted tree of any connected graph $G = (\mathsf{V}, \mathsf{E})$ such that the root is an extremity of the edge e_1. From this parent function we make a total function where the root is mapped to the other extremity of the edge e_1. This labelling can be obtained by our algorithm. Moreover, it ensures that there will be no handshake in the graph except maybe in e_1 which is not a problem because we only consider edges e_{i+1}, .., e_m for $i \in 1..m$. That is why we need in hypothesis to have at least one edge. □

5.3 The Maximal Matching Algorithm

Here is the definition of the maximal matching algorithm. We show that this algorithm terminates with probability 1. This algorithm consists in iterating the handshake algorithm (`DMMLoc2`) only by considering the active vertices where vertices in handshake becomes inactive (`DMMLoc1`). At the beginning, every vertex is active. At the end, every vertex is inactive (`termB`).

```
Definition DMMLoc1 (λ:Λ) (ψ_out ψ_in:seq Ψ) : FR (Λ × seq Ψ) :=
   if (active λ) then
     if (agreed ψ_out ψ_in) then
           Dreturn ( Some (index true ψ_out) , ψ_out)
     else Dreturn (None, map (fun x ⇒ true) ψ_out)
   else Dreturn (λ, ψ_out).

Definition DMMLoc2 (λ:Λ) (ψ_out ψ_in:seq Ψ) : FR (Λ × seq Ψ) :=
   DHSLoc λ ψ_out ψ_in .

Definition termB (f: State) : bool :=
   [∀ v, active (f.1 v)].

Definition DMMLV (sV: seq V) (σ: State) :=
  DLV sV σ (DMMLoc1::DMMLoc2::nil) termB.
```

The general lemma `DPLV_total` (see Lemma `termglobal`) implies the specific results: this algorithm terminates with probability 1.

```
Theorem DMMLV_term: ∀ σ, μ (DMMLV (enum V) σ) 𝕀 = 1.
```

Proof: To prove this lemma, we used the general result `termglobal`. We first show that the probability to have a handshake during a round is strictly positive which means that the number of active decrements with a non null probability. Hence as a variant we take the number of active vertices. The property always true `PR` in our labelling is that every active vertex sends 1 to all of its neighbours and every inactive vertex sends 0. We only have to prove the 7 hypotheses of Lemma `termglobal`. □

6 Conclusion

We develop on this paper tools to reason about (randomised) distributed algorithms in anonymous networks. We prove negative results but also we prove properties over randomised algorithms which solve handshake and maximal matching problems. More particularly, for the handshake problem, we analyse the probability of at least a handshake in a round. We then iterate this algorithm to construct a maximal matching. We prove that this algorithm terminates with probability 1. Many of the techniques used in this paper can be applied to analyse solutions for other similar problems like symmetry break, local election algorithms and distributed computing of maximal independent sets. One of the future works consists in proving properties about time complexity by providing tools to handle the number of rounds.

Acknowledgement

The authors are grateful to P. Castéran who follows this work all along. We thank him for his first proof in *Coq* of the impossibility result stated in Section 5.1.1 and for the development of the semantics that is the base of their development. They also thank C. Paulin-Mohring and A. Mahboubi for their help using *Alea* and *ssreflect* respectively.

References

[1] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 74(8):568–589, 2009. doi:10.1016/j.scico.2007.09.002.

[2] C. Auger, Z. Bouzid, P. Courtieu, S. Tixeuil, and X. Urbain. Certified impossibility results for byzantine-tolerant mobile robots. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013*, pages 178–190, 2013. doi:10.1007/978-3-319-03089-0_13.

[3] P. Castéran and V. Filou. Tasks, types and tactics for local computation systems. *Studia Informatica Universalis, Hermann*, 9(1):39–86, 2011.

[4] C.-T. Chou. Mechanical verification of distributed algorithms in higher-order logic. *The Computer Journal*, 38:158–176, 1995. doi:10.1007/3-540-58450-1_41.

[5] P. Courtieu, L. Rieg, S. Tixeuil, and X. Urbain. Certified universal gathering in R^2 for oblivious mobile robots. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016*, pages 187–200, 2016. doi:10.1007/978-3-662-53426-7_14.

[6] Deploy European Community Project, www.event-b.org. *Event-B and the Rodin Platform.*

[7] C. Derman. *Finite state Markovian decision processes.* Mathematics in science and engineering. Academic Press, Orlando, FL, USA, 1970.

[8] A. Fontaine and A. Zemmari. Rda: A Coq Library on Randomised Distributed Algorithms. www.allyxfontaine.com/rda.

[9] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010. doi:10.6092/issn.1972-5787/1979.

[10] J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanized in *HOL*. *Electronic Notes in Theoretical Computer Science*, 112:95–111, 2005. doi:10.1016/j.entcs.2004.01.021.

[11] Monads in Haskell. http://www.haskell.org/haskellwiki/monad.

[12] D. E. Knuth. *The art of computer programming, volume 2: seminumerical algorithms*. Addison-Wesley, Boston, MA, USA, 1981.

[13] P. Küfner, U. Nestmann, and C. Rickmann. Formal verification of distributed algorithms - from pseudo code to checked proofs. In *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012*, pages 209–224, 2012. doi:10.1007/978-3-642-33475-7_15.

[14] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation / TOOLS*, pages 200–204, 2002. doi:10.1007/3-540-46029-2_13.

[15] M. Z. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In *Computer Aided Verification (CAV'01)*, pages 194–206, 2001. doi:10.1007/3-540-44585-4_17.

[16] Y. Métivier, N. Saheb, and A. Zemmari. Analysis of a randomized rendez-vous algorithm. *Information and Computation*, 184(1):109–128, 2003. doi:10.1016/S0890-5401(03)00054-3.

[17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

[18] A. Pogosyants and R. Segala. Formal verification of timed properties for randomized distributed algorithms. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC'95)*, pages 174–183, 1995. doi:10.1145/224964.224984.

[19] DAMPAS Project. Visidia. http://visidia.labri.fr.

[20] Coq Development Team. *The Coq Proof Assistant Reference Manual*. coq.inria.fr.

© Scientific Annals of Computer Science 2016