Formal Proofs of Local Computation Systems

Pierre Castéran, Vincent Filou, Mohamed Mosbah

▶ **To cite this version:**

Pierre Castéran, Vincent Filou, Mohamed Mosbah. Formal Proofs of Local Computation Systems. 2009. hal-00371705

HAL Id: hal-00371705
https://hal.science/hal-00371705

Submitted on 30 Mar 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire Bordelais de Recherche en Informatique
UMR 5800 - Université Bordeaux 1, 351, cours de la Libération,
33405 Talence CEDEX, France

Research Report RR-1456-09

Formal Proofs of Local Computation Systems[1]

Pierre Castéran, Vincent Filou and Mohamed Mosbah
LaBRI,
Université Bordeaux I, ENSEIRB
351 cours de la Libération
33405 Talence, France
{casteran,filou,mosbah}@labri.fr

March 30, 2009

Formal Proofs of Local Computation Systems[†]

Pierre Castéran, Vincent Filou and Mohamed Mosbah
LaBRI,
Université Bordeaux I, ENSEIRB
351 cours de la Libération
33405 Talence, France
{casteran,filou,mosbah}@labri.fr

March 30, 2009

[*]partly supported by ANR **A3PAT** ANR-05-BLAN-0146 and **RIMEL** ANR-06-SETIN-015

[†]partly supported by ANR **A3PAT** ANR-05-BLAN-0146 and **RIMEL** ANR-06-SETIN-015

Abstract

This paper describes the structure of a library for certifying distributed algorithms, contributed by Pierre Castéran and Vincent Filou. We formalize in the Coq proof assistant the behaviour of a subclass of local computation systems, namely the **LC0** graph relabeling systems. We show how this formalization allows not only to build certified **LC0** systems, but also study the limitations of their control structure.

Contents

1 Introduction

Local computations on graphs, and particularly graph relabeling systems, have been introduced in [5] as a suitable tool for encoding distributed algorithms, for proving their correctness and for understanding their power. In this model, a network is represented by a graph whose vertices denote processors, and edges denote communication links. The local state of a processor (resp. link) is encoded by the label attached to the corresponding vertex (resp. edge). A relabeling rule is a rewriting rule which has the same underlying fixed graph for its left-hand side and its right-hand side, but with an update of the labels. According to its own state and to the states of its neighbours, each vertex may decide to realize an elementary *computation step*. After this step, the states of this vertex, of its neighbours and of the corresponding edges may change according to some specific *computation rules*.

In the *Visidia* [21] research project, we are interested in formal specification and formal proof of local computation systems. This activity consists in giving a formal semantics to these systems, develop tools for the certification of such algorithms : proof of invariants, proof of termination, and also compare the computational power of various subclasses of local computation systems or other kinds of distributed computation paradigms. The *Coq* proof assistant [18, 3] has been chosen for this task.

In this paper, we focus on a particular class of computation rules: **LC0** rules. An **LC0**-rule performs a computation on two adjacent nodes of the network. It corresponds to a rendez-vous [2, 15, 16] between two nodes. This class of computation corresponds to the client-server model employed in most network applications.

This paper describes a first modelization in *Coq* of **LC0**-relabeling systems. Section 3 presents the fundamental notions on graph Relabeling systems, illustrated by two simple examples : the distributed computation of a connected graph's spanning tree, then an election algorithm working on any tree. Section 4 shows how *Coq*'s type system has been used to represent **LC0**-relabeling systems, and a brief overview of proof techniques. Section 5 presents some properties shared by all **LC0** systems, which allow to prove some specifications are not realizable by such systems without some initial knowledge (section 6). Section 7 shows briefly the present structure of our *Coq* libraries. Finally, the directions or future work are sketched in section 8.

2 Related Work

The certification of distributed algorithms using a theorem prover has been the subject of a quantity of studies. Closest to our study, the work of Ching-Tsung Chou [7] aims at mechanicaly proving properties of distributed algorithms using Higher-Order Logic. In this work, the author uses a notion of simulation to carry out the proof of correctness of an algorithm propagating information through a network. Qiao Heiyan has been working on a methodology for proving distributed algorithms in type theory, using the Agda theorem prover [14].
The `UNITY` language has also been used [19] to prove the security properties of distributed algorithm in *Coq*. Bezem, Bol and Groote [4] used the *Coq* proof assistant for proving equalities between terms of a process algebra. Leu also cite the work of Gascard and Pierre [10] on formal proofs of parallel programs on symmetric interaction networks.

In [6], Cansell and Méry show how to get local computation systems from their specification through a sequence of refinements. They use for this purpose the Event-B method and the `Rodin` tool [1]. The specification is encoded by an abstract machine which computes in one big step a final state which satisfies the specification, the distributed algorithm being obtained as the last step of the refinement chain. The class of algorithms addressed in their work is exactly the same as ours. Examples of developed algorithms using this approach can be found in [20]. In a few words, one could say that their approach is a kind of top-down development methodology, whilst ours tries to get properties from the language used to program local computation systems.

3 Graph Relabeling Systems for Encoding Distributed Algorithms

In this section, we give a few definitions of local computations, and particularly of graph relabeling systems. As usual, a network is represented by a graph whose vertices stand for processors and edges for (bidirectional) links between processors. At every time, each vertex and each edge is in some particular state and this state will be encoded by a vertex or an edge label. According to its own state and to the states of its neighbours, each vertex may decide to realize an elementary *computation step*. After this step, the states of this vertex, of its neighbours and of the corresponding edges may have changed according to some specific *computation rules*. Let us recall that graph relabeling systems satisfy the following requirements:

(C1) they do not change the underlying graph but only the labeling of its components (edges and/or vertices), the final labeling being the result,

(C2) they are local, that is, each relabeling changes only a connected subgraph of a fixed size in the underlying graph,

(C3) they are locally generated, that is, the applicability condition of the relabeling only depends on the local context of the relabeled subgraph.

For such systems, the distributed aspect comes from the fact that several relabeling steps can be performed simultaneously on "far enough" subgraphs, giving the same result as a sequential realization of them, in any order. A large family of classical distributed algorithms encoded by graph relabeling systems is given in [2]. This approach has proved itself very well suited to obtain theoretical results on the impact of the geography of a network on algorithms[17, 12, 13].

In order to make the definitions easy to read, we give in the following an example of a graph relabeling system for computing a spanning tree, and an example of a graph relabeling system to elect a node in a tree. Then, the formal definitions of local computations will be presented.

3.1 Distributed computation of a spanning tree

Let us first illustrate graph relabeling systems by considering a simple distributed algorithm which computes a spanning tree of a network. Assume that a unique given processor is in an "active" state (encoded by the label **A**), all other processors being in some "neutral" state (label **N**) and that

all links are in some "passive" state, represented by the boolean **f**. The tree initially contains the unique active vertex. At any step of the computation, an active vertex may activate one of its neutral neighbours and mark the corresponding link which gets the label **t**. This computation stops as soon as all the processors have been activated. The spanning tree is then obtained by considering all the links with label **t**.

An elementary step in this computation may be depicted as a *relabeling step* by means of the relabeling rule R, given in Figure 1, which describes the corresponding label modifications (remember that labels describe processor status):

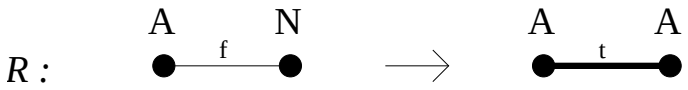

Figure 1: The relabeling rule R

Whenever an A-labeled node is linked by a **f**-labeled edge to an N-labeled node, then the corresponding subgraph may rewrite itself according to the rule.

A sample computation using this rule is given in Figure 2. Relabeling steps *may* occur concurrently on disjoint parts on the graph. When the graph is irreducible, i.e no rule can be applied, a spanning tree, consisting of **t**-edges labeled edges, is computed.

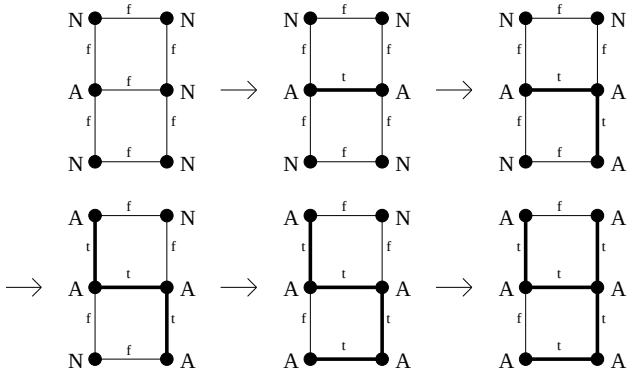

Figure 2: Distributed computation of a Spanning Tree

3.2 Vertex Election in a Tree

The aim of an election in a graph is to choose exactly one vertex among the set of all its vertices. This vertex becomes *elected* and is called the *leader* of the graph. In our framework, this problem can be formalized in the following way. We say that a relabeling system (as will defined formally in the following section) solves the election problem for a class $\mathcal{C}$ of graphs if it is noetherian and if the following conditions hold :

- Any vertex of G may know if it is already elected or beaten, or still ignores this result
- Once a vertex is beaten or elected, its situation cannot change anymore,
- There is at most one elected vertex in the graph
- When no rule can be applied, all vertices are either beaten or elected

Let us present a graph relabeling system allowing to elect a vertex in a tree (see Figure 3).

The alphabet for labeling the vertices is the set of natural numbers $\mathbb{N}$ augmented with a new element called "`N`". Intuitively we interpret `N` as "beaten", 0 as "elected", and any strictly positive integer d as "still ignoring the result". We do not label the graph's edges.

Initially, each vertex v is labeled with its degree in the graph. The set of rules we consider is the union of all replacements depicted in figure 3 for any $d \in \mathbb{N}$.

The intuitive behaviour of our election system can be described as follows : let us associate to any state s the subgraph G_s determined by the $\mathbb{N}$-labeled vertices of G. In fact any computation step reduces to erasing some "pendant" edge (*i.e.* having at least an extremity of degree 1 w.r.t. G_s) from G_s.

We prove two invariants of the system :

- in any reachable state s, any $\mathbb{N}$-labeled vertex measures the degree of v with respect to G_s,
- in any reachable state s, G_s is a tree.

From the invariant above, we infer that in a reachable state where no rule can be more applied (an *irreducible* state), there exists a unique vertex labeled 0 (hence elected) whilst other vertices of G are `N`-labeled (hence beaten).

Figure 3: Rules for the election of a node in a tree

3.3 Formal Definitions of Graph Relabeling Systems

We consider only connected graphs whose vertices [resp. edges] are labeled with labels from a possibly infinite alphabet L_v [resp. L_e]. We call *state* (or *labeling*) any pair s of two functions $\lambda_s \colon V(G) \to L_v$ and $\rho_s \colon E(G) \to L_e$.

Now we introduce the formal definition of a graph relabeling rule.

Definition A *(graph) relabeling rule* is a triple $r = (G_r, s_r, s'_r)$ such that (G_r, s_r) and (G_r, s'_r) are two labeled graphs. The labeled graph (G_r, s_r) is the *left-hand side* and the labeled graph (G_r, s'_r) is the *right-hand side* of r.

If the graph G_r is isomorphic to the graph $k2$ (which consists of two vertices connected by an edge), we say that the relabeling rule is of type **LC0**.

The intuitive notion of computation step will then correspond to the notion of relabeling step:

Definition A *r-relabeling step* is a 5-tuple (G, s, r, φ, s') where r is a relabeling rule and φ is both an occurrence of (G_r, s_r) in (G, s) and an occurrence of (G_r, s'_r) in (G, s').

Intuitively speaking, the labeling s' of G is obtained from s by modifying all the labels of the elements of $\varphi(G_r, s_r)$ according to the labeling s'_r. Such a relabeling step will be denoted by $(G, s) \to_{r,\varphi} (G, s')$.

This notation can be generalized to $(G, s) \to_{\mathcal{R}} (G, s')$ (making abstraction of the occurrence ϕ and the rule $r \in \mathcal{R}$), and to $(G, s) \to^*_{\mathcal{R}} (G, s')$ (reflexive and transitive closure of $\to_{\mathcal{R}}$). This last notation expresses the notion of finite computation according to the considered set $\mathcal{R}$ of rules.

The computation stops when the graph is labeled in such a way that no relabeling rule can be applied:

Definition A labeled graph (G, s) is said to be *$\mathcal{R}-$irreducible* if there exists no occurrence of (G_R, s_R) in (G, s) for every relabeling rule R in P.

Definition Finally, a *graph relabeling system* is a tuple $\mathcal{S}$ gathering the following information :

- Two alphabets for labeling vertices and edges (respectively L_v and L_e),
- A set $\mathcal{R}$ of relabeling rules,
- A class $\mathcal{C}$ of graphs, the system is assumed to work on,
- A function $\mathcal{I}$ which associates to each graph G in $\mathcal{C}$ a set $\mathcal{I}_\mathcal{G}$ of `initial states`

We will use the abbreviation $(G,s)\rightarrow^*_\mathcal{S}(G,s')$ for $(G,s)\rightarrow^*_\mathcal{R}(G,s')$. We will say that s' is *reachable from* s. If $i \in \mathcal{I}_\mathcal{G}$ and $(G,i)\rightarrow^*_\mathcal{R}(G,s')$, we will simply say that s' is *reachable*.

In this paper, we consider only rules of type **LC0**; in fact there are other types (see [2, 16]) considered in our future developments, mainly **LC1** and **LC2**. Both of them are defined when G_r is isomorphic to a star graph (called also a ball of radius one). Roughly speaking, in an **LC1** computation step, the label attached to the center of the star is modified according to some rules depending on the labels of the star, labels of the leaves are not modified. In an **LC2** computation step, labels attached to the center and to the leaves of the star may be modified according to some rules depending on the labels of the star.

4 Types for Representing Graph Relabeling Systems

We show how the rich type system of *Coq* [18, 3] allow us to represent graph relabeling systems and their specification, and some proofs about the **LC0** class.

4.1 Labeled Graphs

Non-oriented graphs are encoded with the help of the `FSets` library, contributed by Pierre Letouzey for *Coq*'s standard library. Let us call *Pregraph* any pair G of a finite set of vertices $V(G)$ and a finite sets of edges $E(G)$. The `FSets` library is compatible with *Coq*'s *setoid* feature, thus we consider as equivalent any pair of edges (v,v') and (v',v).

Some predicates defined on `PreGraph` define the set of graphs (pregraphs which satisfy the proposition) $(E(G) \subseteq V(G) \times V(G))$, of simple graphs (self-loop free), acyclic, connected graphs, etc.

We do not provide a type for representing directly *labeled graphs*. For simplicity's sake, we define some types for labelings as follows :

- We consider two non empty types L_v (for vertices' labels) and L_v (for edges' labels). In the spanning-tree example of section 3.1, we use an inductive type `AN` with two constructors `A` and `N` for vertex-labeling and the pre-built type `bool` for edge-labeling. Notice that any type with decidable equality is well suited for labels.

- A *state* s is any pair of a vertex-labeling λ_s and an edge-labeling ρ_s, where λ_s is finite map from the type of vertices to L_v (similarly for ρ_s). In general we consider only states where the functions λ_s and ρ_s are total (we say that "s *covers* G".)

- Finally, a labeled graph is any pair (G, s) where s is a state for G.

In our *Coq* development, we denote by "`state` L_v L_e" the type of states using the alphabets L_v and L_e.

4.2 Graph Relabeling Relations

A *Graph relabeling relation* for a graph G is any binary relation $\rightarrow$ between states,such that if s covers G and $s \rightarrow s'$, then s' covers G. Notice that this condition corresponds to the `C1` requirement of section 3. Graph relabeling relations inhabit the dependent type "`GR` L_v L_e G" [1] for graph relabelings operating on G. Notice that G is in last position, so we cannot define types for labeling knowing already the graph G. The task of building a distributed algorithm using graph relabeling amounts to choose two types is building some term of type L_v and L_e, then build a term S inhabiting the dependent product "$\forall$ G, `GR` L_v L_e G".

Let us consider for instance an algorithm S of spanning-tree computation. Its correctness can be expressed as follows :

> Let G be any simple, connected graph; for any initial state i,
>
> - any rewriting sequence starting from i terminates, and
> - any irreducible state s reachable from i determines a spanning tree of G

[1] defined in the module `GRS/GraphRelabeling`

The graph G is the *parameter* of S. The predicate " to be a simple, connected graph" characterizes the *class* of graphs it is supposed to work on. Notice that this "specification" is still informal ; we must make precise the notion of initial state, and the verb "determines". For the latter, the meaning is simply "get a function r from labeled graphs to graphs, such that if s is any irreducible, reachable state w.r.t. $S(G)$, then $r(G, s)$ is a spanning tree of G."

4.3 LC0 Systems

We present a formal representation of a **LC0** System; throughout this section, the non empty types L_v and L_e are fixed; we will use the notation Σ as an abbreviation for the type "`state` L_v L_e ".

4.3.1 LC0 Rules

As said in section 3.3, **LC0** relabeling rules can be represented as a relabeling relation on the graph $k2$. Such a relation can be represented as a binary relation R on the product type $L_v \times L_e \times L_v$.

To be more precise, we encode any **LC0** set of rules as a 6-ary predicate whose arguments as are follows: By convention, we name the vertices of $k2$ as c (for "center") and v (for "voisin", french translation of "neighbour").

$\lambda_s(c)$	old label of the center c
$\lambda_s(v)$	old label of the neighbour v
$\rho_s(c, v)$	old label of the edge (c, v)
$\lambda_{s'}(c)$	new label for the center c
$\lambda_{s'}(v)$	new label for the neighbour v
$\rho_{s'}(c, v))$	new label for the edge (c, v)

In other terms, we describe a kind of before/after relation describing the state change on $k2$.

Let us take for instance the spanning tree rule : It is represented by an inductive predicate :

```
Inductive R : AN → AN → bool → AN → AN → bool → Prop :=
R_intro : R A N false A A true.
```

4.3.2 Embedding into a Graph Relabeling Relation

Let R be a set of **LC0**-rules over the types L_v and L_e. It is easy to embed R into a relabeling relation over any graph G :

Let s and s' be two states covering G; $(G, s) \rightarrow_{\mathcal{R}} (G, s')$ if there exists two adjacent vertices c and v of G, such that

- the proposition $R(\lambda_s(c), \lambda_s(v), \rho_s(c, v), \lambda_{s'}(c), \lambda_{s'}(v), \rho_{s'}(c, v))$ holds, and
- s and s' coincide everywhere except on the vertices c,v and the edge (c, v)

4.3.3 Graph Classes and Initial States

A **LC0** relabeling system is supposed to perform some computations on some class $\mathcal{C}$ of graphs, starting from some initial state. In section 4.2 we presented a graph class as any predicate P defined on graphs.

In additon to its sets of rules, a graph relabeling system must determine a set of possible initial states for any graph G which belongs to $\mathcal{C}$.

4.4 A Type for LC0 Systems

Putting all this information together, we can define a type **LC0** for **LC0**-systems as a dependent structure containing the following fields :

- Two non empty types L_v and L_e,
- A graph class $\mathcal{C}$: `PreGraph`$\rightarrow$`Prop`,
- A family of sets of initial states $\mathcal{I}$: `PreGraph`$\rightarrow(\Sigma\rightarrow$`Prop`$)$
- A **LC0**-set of rules $\mathcal{R} : L_v \rightarrow L_e \rightarrow L_v \rightarrow L_v \rightarrow L_e \rightarrow L_v \rightarrow$ `Prop`.

4.5 Proof Techniques for LC0 Systems

Let $\mathcal{S}$ be a **LC0**-system described as above. There are a few kinds of statement one may have to prove about $\mathcal{S}$.

4.5.1 Invariants

An invariant of $\mathcal{S}$ is any relation I on states such that $(G, i) \rightarrow^*_{\mathcal{R}} (G, s)$ implies $I\, i\, s$ for any graph G of the considered class $\mathcal{C}$, any initial state $i \in \mathcal{I}_G$.

Let us consider for instance the invariants of the spanning tree algorithm: For writing these invariants, we need an auxiliary definition : Let G be a

graph, and s be some state, then $G_A(s)$ is the subgraph of G containing all the vertices of G such that $\lambda_s(v) = \mathtt{A}$ and the edges of G such that $\rho_s(e) = \mathbf{t}$.

Thus the invariants we prove are the following ones :

edge_inv: For each edge of e of G, if at least one of e's extremities is labeled by `N` in the state s, then $\rho_s(e) = \mathtt{f}$,

tree_inv: $G_A(s)$ is a tree,

ExA: There exists at least some vertex v of G such that $\lambda_s(v) = \mathtt{A}$.

Proof techniques for invariants are quite usual : we prove that they are satisfied by any initial state, and preserved by the relation $\rightarrow_{\mathcal{R}}$. The proof of this preservation is helped by specialized inversion tactics, which infer from some hypothesis $(G, s) \rightarrow_{\mathcal{R}} (G, s')$ the equalities of section 4.3.2.

4.5.2 Termination

The usual proof technique for termination can be very simplified when dealing with **LC0**-systems. If we associate to any label l of L_v and L_e some value $\mu(l)$ in a well-founded domain $(D, <)$, and if $+$ is an associative, commutative and monotone operation on D, then it is enough to check the inequality $\mu(x') + \mu(y') + \mu(z') < \mu(x) + \mu(y) + \mu(z)$, whenever $R\ x\ y\ z\ x'\ y'\ z'$ holds. For instance, if we associate 1 to the label `N`, and 0 to any other label, the proof of termination of the spanning-tree computation is simply a proof of $0 < 1$.

5 Properties Shared by all LC0 Systems

It is important to consider *generic* properties of the **LC0** class of local computation systems. This is made possible by *Coq*'s type system, which allows to quantify on the types for labeling, on the function which builds initial states, and the sets of **LC0**-rules.

First, **LC0**-systems are consistent w.r.t. the theoretical model by Godard and Métivier [11] : If two labeled graphs (G, s) and (G', s') are isomorphic, then any **LC0**-system S has the same behaviour on both labeled graphs.

Other properties implement a common scheme : from some computation on a labeled graph (G, s), build a computation for another appropriate labeled graph (G_1, s_1) (with respect to the same relabeling system).

5.1 Extending a Computation to a Bigger Graph

The first case (lemma `LC0:extends_computation`), deals with the case where (G, s) is a labeled subgraph of (G_1, s_1) (intuitively (G_1, s_1) is obtained from (G, s) by adding some labeled vertices and edges). Then for any computation $(G, s) \rightarrow_{\mathcal{R}} (G, s')$ there exists a computation $(G_1, s_1) \rightarrow_{\mathcal{R}} (G_1, s'_1)$ where (G, s') is a labeled subgraph of (G_1, s'_1). In other words, the **LC0**-system S can "ignore" the extra edges and vertices and perform the same computations in G and G_1.

This property is used in Section 6.1 for proving that no **LC0**-system can be used to compute the degree of the underlying graph's vertices.

Notice that this property is clearly false for **LC1** and **LC2**-systems, since their rules' guards explore the neighbourhood of any vertex. It is then possible to consider some **LC1** or **LC2**-guard which becomes false if some new neighbour is added to some vertex.

5.2 Composing Computations

Let (G, s) be a labeled graph. Let us consider two disjoint labeled-subgraphs of G : (G_1, s_1) and (G_2, s_2), and two computations $(G_1, s_1) \rightarrow_{\mathcal{R}} (G_1, s'_1)$ and $(G_2, s_2) \rightarrow_{\mathcal{R}} (G_2, s'_2)$. Then we can build a computation $(G, s) \rightarrow_{\mathcal{R}} (G, s')$ where (G_1, s'_1) and (G_2, s'_2) are labeled subgraphs of (G, s').

This property is used in the proof of 6.2.

When we extend this property to **LC1** or **LC2** systems, we will have to consider some stronger constraints than "G_1 and G_2 are disjoint" : if we allow some edge (v_1, v_2) to link G_1 and G_2, any state change in v_1 can affect rule guards in G_2. Thus G_1 and G_2 will have to be "distant enough".

6 Computational Power of LC0 Systems

LC0 algorithms form only a subfamily of local computation systems. It is thus interesting to consider the limitations of this class of algorithms. In this section, we present two examples of specifications, which cannot be realized by any **LC0**-system under some "reasonable" conditions (a term that will be explicited below).

6.1 Computing the Vertices' Degree

Let us consider for instance the following statement :

No **LC0** algorithm can compute the degree of the underlying graph's vertices

We have to make this statement more precise in order to prove it :

There exists no types L_v and L_e, no function $f : L_v \rightarrow$option $\mathbb{N}$, no set of **LC0**-rules, no "reasonable" family of set of initial states $\mathcal{I}$, such that for any graph G, :

- for any reachable state s, any vertex v of G, if $f(\lambda_s(v)) =$ `Some` d, then the degree of v in G is d,
- for any initial state i, there exists some state s reachable from i, such that $f(\lambda_s(v)) =$ `Some` d, for some d.

It remains to make more precise which sets of initial states we have to consider. We say that a family $\mathcal{I}$ of type "`PreGraph`$\rightarrow(\Sigma\rightarrow$`Prop`$)$" is *extendable* if for any pair of graphs G and G_1 such that G is a subgraph of G_1, then any state i in $\mathcal{I}_G$ can be extended in a state $i_1 \in \mathcal{I}_{G'}$.

Intuitively, any initial state for a graph G can be built incrementally. This is a the case of randomly built initial states, or initial states which attribute a constant label to each vertex [resp. edge].

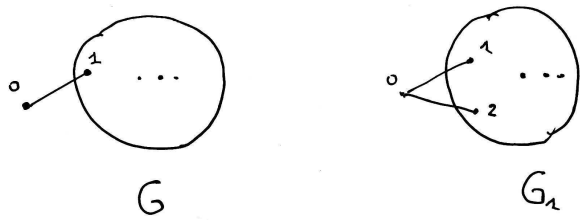

Figure 4: The degree of 0 in G is not equal to the degree of 0 in G_1

The proof is done by contradiction. Suppose that some **LC0**-system allows to compute the degree of any vertex of any graph, with an extendable family I of sets of initial states. Let us consider the graphs G and G_1 as in figure 4. Then there exists some computation starting from some initial state $i \in \mathcal{I}_G$, reaching some state s, where $f(\lambda_s(0) =$ `Some`$(\deg_G(0))$. Then there exists some initial state i_1 for G_1 which extends i, and, by the property

5, some state s_1 which extends s and is reachable from i_1. The following equalities show that our hypotheses lead to a contradiction.

$$\begin{aligned} \texttt{Some } 2 &= \texttt{Some}(\deg_{G_1}(0)) \\ &= \lambda_{s_1}(0) \;=\; \lambda_s(0) \\ &= \texttt{Some}(\deg_{G_1}(0)) \;=\; \texttt{Some } 1 \end{aligned}$$

6.1.1 Important Remark

Notice that the statement we proved deals with a local knowledge of the each vertex's degree at any time of the computation. Let us consider the following change of our specification :

- for any irreducible reachable state s, any vertex v of G, if $f(\lambda_s(v)) = \texttt{Some } d$, then the degree of v in G is d,
- for any initial state i, there exists some state s reachable from i, such that $f(\lambda_s(v)) = \texttt{Some } d$, for some d.

We can easily build a **LC0**-algorithm for computing the degree of each vertex.

- Vertices are labeled by a natural number (initially 0), and edges by booleans (initially **f**),
- The unique rule applies to any edge labeled by **f**, increments its extremities, and relabels this edge with **t**.

Notice that this example doesn't contradict at all the result shown in section 6.1. There is a big difference between a local knowledge of a vertex's degree, and using a global knowledge about irreducibility of all the graph's state.

6.2 Initial States for an Election Algorithm

In section 3.2, we presented a **LC0** election algorithm. Its correction relies on the fact that its initial states encode every vertex's degree in the graph.

A very natural question is : "is this initial knowledge necessary" ? A partial answer is given if we determine some properties of the function $\mathcal{I}$ which associates to any tree T the set of initial states allowed for starting the computation.

Let $\mathcal{I}$ be a family of set of initial states allowing to build a configuration like in section 5.2, where s, [resp. s_1, s_2] are initial states taken in $\mathcal{I}_G$ [resp. $\mathcal{I}_{G_1}$, $\mathcal{I}_{G_2}$.

Let us assume that some **LC0** election algorithm S works for, say, trees, using such a family $\mathcal{I}$. Consider the graph P_4, decomposed in two disjoints

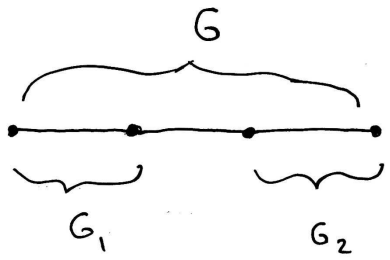

Figure 5: A graph with two disjoint subgraphs

subgraphs isomorphic to P_2 linked by an edge (fig 5). Applying the property in section 5.2, we can easily reach a state where two vertices are elected, which contradicts the election specification.

Among the families $\mathcal{I}$ we can consider for that purpose, we can cite :

- Any state defined over the types L_v and L_e can be an initial state
- There are some constants $l_v : L_v$ and $l_e : L_e$ such that every vertex [resp. edge] is labeled l_v [resp. l_e].
- Any state where no vertex is elected or beaten yet.

7 About the Development in *Coq*

The libraries on graph relabeling systems work with the version 8.2 of the *Coq* proof assistant, and can be downloaded at www.labri.fr/∼casteran/Penda. These libraries are still in development, and some definitions, theorem statements and specialized tactics will be subject to change. Some proofs are very long and will be shortened too. The present (March 2009) representation of unordered edges led to very long proofs. It will be replaced in the next release by another representation relying on reflection.

We present briefly the structure of this library, which will be quite stable, and will be extended to wider classes of local computation systems. The library is divided in four parts :

7.1 Library *Prelude*

This part contains some additional definitions and lemmas, extending *Coq*'s standard library.

We mainly extend Letouzey's `FSets` and `FMaps` libraries . The module `AccP` is a slight extension to `Wellfounded`, dealing with relations of the form $P\,a \wedge R\,a\,b \wedge P\,b$.

Finally, `Monoids` defines the structure of [commutative] monoid on a setoid.

7.2 Library *Graphs*

In our development, we consider unoriented finite graphs, represented with finite sets of vertices and edges. Usual notions such as pathes, cycles, connexity are also defined. Since mechanizing graph theory is not the main purpose of our work, we often use in our proofs of algorithms some folklore lemmas which will be proven when there are more time or people to do it. At present the module `Graph_Admitted` contains only two lemmas, used in the proof of the election algorithm:

- `Acyclic_leaf` : Any non empty, acyclic graph G has at least a vertex of degree 0 or 1,
- `remove_leaf_tree` : Let G be a tree, x and y two adjacent vertices of G, where x is a leaf of G. Then removing the edge (x, y) from G results in a tree.

The module `GraphLabeling` defines the notion of labeled graph.

7.3 Library *GRS* (Graph Relabeling Systems

The sublibrary `GRS` contain general definitions about Graph Relabeling relationsé: invariants, termination, etc.

A specialized module `LC0` is devoted to the **LC0** class of graph relabeling systems, and contains specialized tactics, as well as the proofs of the theorems presented in section 5.

7.4 Library *Examples*

The Examples directory contains some cases studies. Two of them are examples of algorithm certification through invariant proofs.

7.4.1 Spanning Tree Computation

The module `SpanningTree_Specification` contains a formal specification of any graph relabeling system in which any irreducible states allows to get a spanning tree of the input graph.

The module `SpanningTree_LC0` contains a certified implementation of this specification, which works on any simple and connected input graph.

7.4.2 Vertex Election

The module `Election_Specification` contains a formal specification of any election algorithm. It is the formal translation of the informal presentation of section 3.2.

The correction of the algorithm of election in a tree is in the module `Election_Tree_LC0`.

7.4.3 Limitations of LC0 Rules

The module `No_Degree_LC0` contains the formal proof presented in section 6.1 and `No_Election_LC0` the proof presented in section 6.2.

8 Conclusion and Future Work

This works shows how to design a framework for reasoning on a wide class of local computation systems. Such a framework allows two kinds of activities :

- Formal proofs of particular distributed algorithms,
- Proofs of properties of entire classes of such systems, allowing expressive power comparison, including some proofs of the impossibility to realize a given specification within a given class of systems.

This work will be continued in the following directions :

- Automation of parts of the proofs, including termination [8], proof by reflection,

- Extension to more classes of local computation systems, more generally to other formalisms for distributed algorithmics,
- Use of the Why [9] technology for generating proof obligations,
- Taking into account algorithm composition, probabilistic algorithms, etc.

Acknowledgement We thank Yves Bertot, Dominique Cansell, Pierre Letouzey, Dominique Méry, Yves Métivier and Xavier Urbain for the helpful discussions and for many ideas used in this work.

References

[1] *Event-B and the Rodin Platform.* www.event-b.org.

[2] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI, 2002. http://www.labri.fr/visidia/.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[4] Marc Bezem, Roland Bol, and Jan Friso Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Aspects of Computing*, 9:1–48, 1997.

[5] M. Billaud, P. Lafon, Y. Métivier, and E. Sopena. Graph rewriting systems with priorities. *Lecture notes in computer science*, 411:94–106, 1989.

[6] Dominique Cansell and Dominique Méry. *Logics of Specification Languages*, chapter The Event-B Modelling Method: Concepts and Case Studies, pages 47–152. Springer Verlag, 2007.

[7] Chin-Tsun Chou. Mechanical verification of distributed algorithms in higher-order logic. *The Computer Journal*, 38(2), 1995.

[8] Evelyne Contéjean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

[9] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification.

[10] Eric Gascard and Laurence Pierre. Formal proof of applicatons distributed in symmetric interconnexion networks. *Parallel Processing Letters*, 13(1):3–18, 2003.

[11] E. Godard and Y. Métivier. A characterization of classes of graphs recognizable by local computations with initial knowledge. In 8^{th} *International Conference on Structural Information and Communication Complexity (SIROCCO01)*, pages 179–193. Carleton university press, 2001.

[12] E. Godard and Y. Métivier. A characterization of families of graphs in which election is possible. In *Foundations of Software Science and Computation Structures (FOSSACS)*, Lecture notes in computer science. Springer-Verlag, to appear.

[13] E. Godard, Y. Métivier, and A. Muscholl. Characterizations of classes of graphs recognizable by local computations. *Theory of Computing Systems*, 37:2:249–293, 2004.

[14] Qiao Haiyan. Testing and proving distributed algorithms in constructive type theory. In *Tests and Proof*, volume 4454 Lecture Notes in Computer Science, pages 79–94. Springer, 2004.

[15] I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Mathematical Systems Theory*, 28:41–65, 1995.

[16] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H.J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

[17] A. Mazurkiewicz. Distributed enumeration. *Information Processing Letters*, 61:233–239, 1997.

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

[19] X. Thirioux and G. Padiou. Etude comparative de deux techniques de preuves de programmes unity. In *AFADL'97*.

[20] Mohamed Tounsi, Ahmed Hadj Kacem, Mohamed Mosbah, and Dominique Méry. A Refinement Approach for Proving Distributed Algorithms : Examples of Spanning Tree Problems. In *Integration of Model based Formal Methods and Tools(IMFMT 2009)*, Düsseldorf Allemagne, 02 2009.

[21] ViSiDiA. http://www.labri.fr/visidia. LaBRI Distributed Algorithms Group, 2006.