

Towards Verifying Declarative Netlog Protocols with Coq

Yuxin Deng¹, Stéphane Grumbach², and Jean-François Monin^{3,4}

¹ Shanghai Jiao Tong University, China

² LNRIA-LIAMA

³ Université de Grenoble 1

⁴ CNRS-LIAMA

Abstract. Declarative languages, such as recursive rule based languages, have been proposed to program distributed applications over networks. It has been shown that they simplify greatly the code, while still offering efficient distributed execution. In this paper, we show that moreover they provide a promising approach to the verification of distributed protocols. We choose the Netlog language and use the Coq proof assistant. We first formalize the distributed computation model based on message passing with either synchronous or asynchronous behavior. We then see how the declarative rules of the protocols can be simply encoded in Coq. Finally, we develop the machine embedded on each node of the network which evaluates the rules. This framework enables us to formally verify distributed declarative protocols, as sketched on a concrete example, a breadth-first search tree construction in a distributed network.

1 Introduction

Programming distributed algorithms, such as networking protocols for instance, is a very complex task, which aims at solving global problems using local means, and requires to handle the concurrency of the processes, the delays or even the failure of the communication, as well as the limitations of both the hardware and the communication channels. Most distributed systems rely on algorithms invoking low level systems considerations. High-level abstractions have been proposed to facilitate the programming, based on graph relabeling [24, 3], rule-based languages such as [20, 14], functional languages such as Flask [23], as well as algebras for routing [13].

Rule-based languages provide a declarative programming framework which improves greatly the programmer's burden, with a code which is about two orders of magnitude shorter than standard programming languages, and has been shown to produce efficient algorithms in the case of various networking protocols in particular, by using methods developed in the field of databases for recursive languages à la Datalog [18].

In the present paper, we show that these declarative languages for distributed programming provide a new approach to the verification of distributed programs, which can naturally deal with global properties, e.g. topological properties of

a distributed data structure like a tree. To the best of our knowledge, such properties are hard to prove or even to state with usual techniques relying on labelled transition systems and temporal logics, since they essentially focus on events and their ordering.

We choose to work with the Netlog language [14], a variant of Datalog recently proposed for programming distributed algorithms. The Netlog language relies on deductive rules of the form $head \leftarrow body$, which are installed on each node of the distributed system. The rules allow to derive new facts of the form “*head*”, if their body is satisfied locally on the node. The facts derived might then be stored locally on the node or sent to other nodes in the network depending upon the rule. Netlog admits a semantics which is formally defined by distributed fixpoint, which interleaves local computation on the nodes and communication between the nodes. On each node, a local round consists of a computation phase followed by a communication phase. During the computation phase, the program updates the local data and produces messages to send. During the communication phase, the router transmits the incoming messages to the program, and routes the outgoing messages.

Our objective is to develop a framework to formally verify properties of declarative distributed programs. As to formal verification, there are roughly two kinds of approaches: *model checking* and *theorem proving*. Model checking explores the state space of a system model exhaustively to see if a desirable property is satisfied. It is largely automated and generates a counterexample if the property doesn’t hold. The state explosion problem limits the potential of model checkers for large systems. The basic idea of theorem proving is to translate a system’s specification into a mathematical theory and then construct a proof of a theorem by generating the intermediate proof steps. Theorem proving can deal with large or even infinite state spaces by using proof principles such as induction and co-induction. In this paper, we use Coq, which is a *proof assistant*, that is an interactive theorem prover, in which high level proof search commands construct formal proofs behind the scene, which are then mechanically verified. Using a proof assistant seems more relevant than model checking here since the manipulation of data plays a key role. We develop a Coq library necessary for our purposes, including (i) the formalization of the distributed system; (ii) the modeling of the embedded machine evaluating the Netlog programs; (iii) the translation of the Netlog programs; as well as a formalization of graphs and trees suitable to our needs (respectively for communication networks and our case study).

Technically, we formalize a message passing model for distributed computation. To this effect, we introduce a general framework parameterized by a network topology and an abstract type for data. We then formalize appropriate notions for defining a global behavior in terms of local rounds, in a way such that synchronous and asynchronous behaviors are obtained from the same ingredients. This provides a transition relation between configurations (or global states), on which general definitions can be applied, for example, the coinductive definition

of a run and inductive or coinductive definitions of temporal logic operators and associated proof principles.

In our Coq formalization, each body of a deductive rule is encoded in a systematic way by a tuple parameterized by a configuration, a node `Id` and the free variables of the body. In turn, each Netlog rule is formalized by an inductive type which relates a configuration to a set of data representing updates to be performed atomically at a given node, if the corresponding body is satisfied.

We test the proposed framework on a concrete protocol for constructing spanning trees over connected graphs. It is a distributed version of the classical breadth-first search algorithm (BFS) in a synchronous message passing model. It thus proceeds in rounds, in which all nodes performs some local computation and then exchange data with their neighbors before entering the next round. To show its correctness, the crucial ingredient is to formally prove the validity of the invariant that evolving from one round to another always produces a larger tree rooted at the same node. The protocol is shown to be correct for any finite connected graph.

The paper is organized as follows. In Section 2, we formalize the distributed computational model. Section 3 is devoted to the presentation of Netlog programs, and their translation in Coq. Section 4 presents the Coq formalization of the Netlog machine. Section 5 sketches the proofs of the correctness of the tree protocol. Section 6 discusses some related work, and finally we conclude in Section 7.

2 Distributed Computation Model

In this section we introduce a distributed computation model based on the message passing mechanism and then formalize it in Coq. A brief overview of Coq is delegated to Appendix A. The distributed computation model does not depend on Netlog. We just assume that the state at nodes have a type `local_data` which can evolve using simple set-theoretic operations such as union. Apart from that, the distributed computation model is quite standard. Note that it is suitable both for synchronous and asynchronous execution.

An important design choice has to be made concerning the Coq representation of finite sets of Netlog facts. Here, only a mathematical representation is needed and the simpler the better. There is no need to look for efficient representations, such as balanced binary trees; simple lists are enough for this job. Similarly, there is no reason to add programs for maintaining sorted lists or even ensuring that elements are not duplicated. In this framework, the union is simply represented by list concatenation, for instance. Representing finite sets by lists without further ado is a suitable choice as long as the only predicate we consider on lists is membership. In the sequel we use freely set-theoretic notation for list operations, e.g. \emptyset for `nil` and \in for `In`.

A *distributed system* relies on a communication network whose topology is given by a *directed connected graph* $\mathcal{G} = (V_{\mathcal{G}}, G)$, where $V_{\mathcal{G}}$ is the set of nodes, and

G denotes the set of *communication links* between nodes. For many applications, we can also assume that the graph is symmetric, that is $G(\alpha, \beta) \Leftrightarrow G(\beta, \alpha)$.

Each node has a unique *identifier*, Id , taken from $1, 2, \dots, n$, where $n \geq 2$ is the number of nodes, and distinct local ports for distinct links incident to it. The control is fully distributed in the network, and there is no shared memory. In this high-level computation model, we abstract away detailed factors like node failures and lossy channels; if we were to formalize a more precise model, most of the data structures defined below would have to be refined.

All the nodes have the same architecture and the same behavior. Each node consists of three main components: (i) a router, handling the communication with the network; (ii) an engine, executing the local programs; and (iii) a local data store to maintain the information (data and programs) local to the node. It contains in particular the fragment of G , which relates a node to its neighbors.

In the Coq formal model, the graph is defined by a relation `edge` between nodes. This relation is itself defined by a function `neighbors` which provides the list of neighbors of a given node.

```
Variable neighbors : nat -> list nat.
Definition edge n m := m ∈ neighbors n.
```

We assume a type `local_data` for the set of facts stored on nodes as well as on communication links. This type is endowed with at least a value representing the *empty set* of facts and a binary function returning the *union* of two sets of facts. The union is used for describing incremental monotonic changes (see `local_round` and `communication` below). Non-monotonic changes such as removing facts are dealt with using an additional *set difference* function.

We also define the type `Bmsg` for “big messages”, i.e. pairs (j, t) where j is a node Id and t a set of data to be transmitted to j . The global state of the system has the type `configuration` defined as follows.

```
Variable local_data: Set.
Variable empty_ld: local_data. (* notation  $\emptyset_{ld}$  *)
Variable union_ld: local_data -> local_data -> local_data. (*  $\cup_{ld}$  *)
Definition Bmsg := nat * local_data.
Record configuration: Set:= mk_configuration {
  Cnode: nat -> local_data;
  Cedge:  $\forall$  src dst: nat, edge src dst -> local_data
}.
```

Given a configuration c and a node Id j , the *data available at j in c* is either `Cnode c j`, or `Cedge c e`, where e is an edge from some node i to j (a more complete expression would be `Cedge c i j e`, but i and j can be easily deduced from the type of e ; in what follows, we omit such implicit arguments, as is done in the formal Coq development).

We distinguish between *computation events*, performed in a node, and *communication events*, performed by nodes which cast their messages to their neighbors. On one node, a *computation phase* followed by a *communication phase* is called a *local round* of the distributed computation.

An *execution* is a sequence of alternating global configurations and rounds occurring on one node, in the case of an asynchronous system, or a sequence of alternating global configurations and rounds occurring simultaneously on each node, in the case of a synchronous system. In the latter case, the computation phase runs in parallel on all nodes, immediately followed by a parallel execution on all nodes of the corresponding communication phase.

A local round at node `loc` relates an actual configuration `pre` to a new configuration `mid` and a list `out` of big messages from `loc`. Furthermore, incoming edges are cleared. The new data `d` to be stored on `loc` is defined by a relation `new_stores` given as a parameter, and we assume that `d` depends only on the data available at `loc` in `pre`. Intuitively, the relation `new_stores` expresses that `d` consists of new facts derived from facts available at `loc` (see more details in Section 4). Similarly, `out` is defined by a relation `new_push` and satisfies similar requirements. Using relations rather than functions for `new_stores` and `new_push` deserves a special discussion provided in Section 3.

Formally, a local round is defined in Coq using the following inference rule (by convention, for such rules, free variables should be read as universally quantified over the whole rule).

$$\frac{\begin{array}{l} \exists d, \text{new_stores } \text{pre } \text{loc } d \wedge \text{Cnode } \text{mid } \text{loc} = \text{Cnode } \text{pre } \text{loc} \cup_{1d} d \\ \text{new_push } \text{pre } \text{loc } \text{out} \\ \forall \text{src } (e: \text{edge } \text{src } \text{loc}), \text{Cedge } \text{mid } e = \emptyset_{1d} \end{array}}{\text{local_round } \text{loc } \text{pre } \text{mid } \text{out}}$$

For modeling asynchronous behaviors, we also need the notion of a trivial local round at `loc`, where the local data does not change and moreover incoming edges are not cleared either.

$$\frac{\begin{array}{l} \text{Cnode } \text{mid } \text{loc} = \text{Cnode } \text{pre } \text{loc} \\ \forall \text{src } (e: \text{edge } \text{src } \text{loc}), \text{Cedge } \text{mid } e = \text{Cedge } \text{pre } e \end{array}}{\text{no_change_at } \text{loc } \text{pre } \text{mid}}$$

A communication event at node `loc` specifies that the local data at `loc` does not change and that facts from `out` are appended on edges according to their destinations.

$$\frac{\begin{array}{l} \text{Cnode } \text{post } \text{loc} = \text{Cnode } \text{mid } \text{loc} \\ \forall \text{dst } (e: \text{edge } \text{loc } \text{dst}), \text{Cedge } \text{post } e = \text{find } \text{dst } \text{out} \cup_{1d} \text{Cedge } \text{mid } e \end{array}}{\text{communication } \text{loc } \text{mid } \text{post } \text{out}}$$

The function `find` returns the fact in `out` whose destination is `dst`. Note that none of the previous three definitions specifies completely the next configuration in function of the previous one. They rather constrain a relation between two consecutive configurations by specifying what should happen at a given location. Combining these definitions in various ways allows us to define a complete transition relation between two configurations, with either a synchronous or an asynchronous behavior.

$$\frac{\begin{array}{l} \text{loc: nat; mid: configuration; out: list Bmsg} \\ \text{local_round loc pre mid out} \\ \forall \text{loc}', \text{loc} \neq \text{loc}' \rightarrow \text{no_change_at loc}' \text{ pre mid} \\ \text{communication loc mid post out} \\ \forall \text{loc}', \text{loc} \neq \text{loc}' \rightarrow \text{communication loc}' \text{ mid post } \emptyset \end{array}}{\text{async_round pre post}}$$

An asynchronous round between two configurations `pre` and `post` is given by a node `Id loc`, an intermediate configuration `mid` and a list of big messages `out` such that there is a local round relating `pre`, `mid` and `out` on `loc` while no change occurs on `loc'` different from `loc`, and a communication relates `mid` and `out` to `post` on `loc` while nothing is communicated on `loc'` different from `loc`.

$$\frac{\begin{array}{l} \text{mid: configuration; out: list Bmsg} \\ \forall \text{loc}, \exists \text{out, local_round loc pre mid out} \wedge \text{communication loc mid post out} \end{array}}{\text{sync_round pre post}}$$

A synchronous round between two configurations `pre` and `post` is given by an intermediate configuration `mid` such that for all node `Id loc`, there exists a list of big messages `out` such that there is a local round relating `pre`, `mid` and `out` on `loc` and a communication relating `mid` and `out` to `post` on `loc`.

Now, given an arbitrary `trans` relation, which can be of the form `sync_round`, or `async_round`, or even of some alternative form, we can coinductively define a run starting from a configuration. We have two cases: either there is a transition from configuration `pre` to configuration `post`, then any run from `post` yields a run from `pre`; or, in the opposite case, we have an empty run from `pre`. Altogether, a run from `pre` is either a finite sequence of transitions ended up with a configuration where no transition is available, or an infinite sequence of transitions, where consecutive configurations are related using `trans`.

```
CoInductive run: configuration -> Set :=
  | Rtrans: ∀ pre post, trans pre post -> run post -> run pre
  | Rterm: ∀ pre, (∀ post, ¬ trans pre post) -> run pre.
```

In order to prove properties on run, we define some temporal logic operators. In the examples considered below we need a very simple version of `always`, which is parameterized by a property `P` of configurations. In a more general setting, the parameter would be a property of runs.

$$\frac{\text{P pre} \quad \text{alw_run P r}}{\text{alw_run P (Rtrans pre post s r)}} \quad \frac{\text{P pre}}{\text{alw_run P (Rterm pre h)}}$$

It is well known that a property which holds initially and is invariant is always satisfied on a run. This fact is easily proved in the very general setting provided by Coq.

```
Definition invariant_round (P: configuration -> Prop) :=
  ∀ pre post, trans pre post -> P pre -> P post.
```

```
Lemma invar_always:
  ∀ (P: configuration -> Prop), invariant_round P ->
  ∀ ini (r: run ini), P ini -> alw_run P r.
```

3 Declarative Netlog Protocols

We next introduce the Netlog language through simple protocols for construction of routes and trees. Only the main constructs are presented. A more thorough presentation of the language can be found in [14]. Netlog relies on datalog-like recursive rules, of the form $head \leftarrow body$, which allow to derive the fact “ $head$ ” whenever the “ $body$ ” is satisfied. In contrast with other approaches to concurrency, the focus is not, primarily, on observing some output, but on the high-level data (i.e. datalog facts) contained in nodes. Imagine, for example, a program for constructing routing tables. Such tables are intended to be used by other protocols and reasoning on their *contents* is more direct than considering events.

The rules of a program are applied in parallel, and the results are computed by iterating the rules over the local instance of the node, using facts either stored on the node or pushed by a neighbor. This is formally defined in Coq by the predicate `inFact`, where `prj` is any projection from `local_data` to some `list X`.

$$\frac{x \in \text{prj}(\text{Cnode cnf loc})}{\text{inFact prj cnf loc } x} \quad \frac{e: \text{edge neighbor loc} \quad x \in \text{prj}(\text{Cedge cnf } e)}{\text{inFact prj cnf loc } x}$$

The following rules, for instance, define routes, stored in a relation $Route(Src, Hop, Dst)$, from the graph E , which from each source node, Src , and for each destination, Dst , gives the next hop, Hop , on the path to that destination.

Simple routes

$$\downarrow Route(x, y, y) \leftarrow E(@x, y). \quad (1)$$

$$\downarrow Route(x, y, z) \leftarrow E(@x, y); Route(y, u, z). \quad (2)$$

This program has the following effect, when applied on a node, say α . If there is a fact $E(\alpha, \beta)$, then $Route(\alpha, \beta, \beta)$ can be derived by Rule (1), and if there are facts $E(\alpha, \beta)$ and $Route(\beta, \gamma, \delta)$, then $Route(\alpha, \beta, \delta)$ can be derived by Rule (2). The symbol “@” in the literal $E(@x, y)$ in the body of the rules forces the variable x to be instantiated by the node Id , or, in other words, forces the rule to run on node x .

The Netlog programs are installed on each node, where they run concurrently. The facts deduced from rules can be stored on the node, on which the rules run, or sent to other nodes. The symbol \downarrow in the head of the rules means that the result has to be both stored on the local data store (\downarrow), and sent to neighbor nodes (\uparrow).

The language also contains *negation*; a node can judge if a fact or its negation is true based on its knowledge from the local data store. *Aggregation functions* can also be used in the head of rules to aggregate over all values satisfying the body of the rule. In the next example, the function *min* will be used for instance. Let us consider next, the construction of a BFS tree for synchronous systems.

The following program relies on three relation symbols: E , $onST$, and ST ; E represents the edge relation; and at any stage of the computation, $onST(\alpha)$ (respectively $ST(\alpha, \beta)$) hold iff the node α (respectively the edge (α, β)) is already on the intended tree.

Synchronous Rooted BFS Tree

$$\downarrow onST(x) \leftarrow @x = 0. \quad (3)$$

$$\left. \begin{array}{l} \downarrow onST(y) \\ \downarrow ST(min(x), y) \end{array} \right\} \leftarrow E(x, @y); onST(x); \neg onST(y). \quad (4)$$

Rule (3) runs on the unique node, say ρ , which satisfies the relation $\rho = 0$. It derives a fact $OnST(\rho)$, which is stored on ρ and sent to its neighbors. Rule (4) runs on the nodes ($@y$) at the border of the already computed tree. It chooses one parent (the one with minimal Id) to join the tree. Two facts are derived, which are both locally stored. The fact $onST(y)$ is pushed to all neighbors. Each fact $E(x, y)$ is assumed to be initially stored on node y . As no new fact $E(x, y)$ can be derived from rules (3) and (4), the consistency of E with the physical edge relation holds forever.

This algorithm aims at constructing a suitable distributed relation ST . More precisely, our objective will be to prove that the relation ST actually defines a (BFS) tree. First we define the actual `local_data` needed in the BFS protocol. The types `unary` and `binary` represent respectively sets (encoded by lists) of unary or binary facts.

```
Record bfs_data : Set := mk_bfs_data {
  onST : unary;
  E : binary;
  ST : binary}.
```

Rules are encoded in Coq according to a systematic method which should be clear from the BFS example. To be more precise, we present here a specification of the expected *semantics* of Netlog rules, as relations between a configuration, a location and incremental changes of data, consistently with what is expected in a local round (see Section 2). A translation or an interpretation of Netlog rules described using an abstract syntax (deep embedding) is left to further work: this is important but less difficult and urgent than discovering the precise shape needed for the semantics of rules – something that came into light in the process of doing proofs. The technical details are somewhat involved and cannot be provided here due to space limitation.

Push rules follow always the same scheme which can be abstracted as follows. Assuming an update `u` related to a configuration `pre` and location `loc` by a relation R given as parameter, the list of `Bmsg` broadcasted from `loc` is made of all pairs (dst, u) such that dst is a neighbor of `loc`, as formalized by:

$$\frac{R \text{ pre } loc \ u}{\text{push } R \text{ pre } loc \ (\text{map } (\text{fun } dst \Rightarrow (dst, u)) \ (\text{neighbors } loc))}$$

A common situation is when a fact has to be both stored and broadcasted, as happens with Rules (3) and (4) of the BFS example. R is then obtained from the corresponding store rule.

Rule (3) leads to two definitions in Coq, for both the store (\downarrow) and the push (\uparrow) consequences. The stored part is coded as follows:

$$\frac{\begin{array}{l} \text{ST upd} = \emptyset \\ \text{E upd} = \emptyset \\ \text{loc} = 0 \rightarrow \text{onST upd} = \{\text{loc}\} \quad \text{loc} \neq 0 \rightarrow \text{onST upd} = \emptyset \end{array}}{\text{compute_phase_store_onST_initial} \text{ pre loc upd}}$$

The push part is then coded by:

Definition `compute_phase_push_onST_initial` :=
`push compute_phase_store_onST_initial`

For Rule (4), we first introduce the inductive definition corresponding to its body $E(x, @y); \text{onST}(x); \neg \text{onST}(y)$ as follows:

$$\frac{\text{in_E cnf loc } (x, y) \quad \text{in_onST cnf loc } x \quad \neg \text{in_onST cnf loc } y}{\text{tree_body cnf loc } x \ y}$$

Here, `in_E` is a specialization of `inFact` to `E`, and similarly for `in_onST` (see the beginning of Section 3). In other words, `in_E cnf loc (x, y)` holds iff a fact `E (x, y)` is available in configuration `cnf` at node `loc`. We then introduce three definitions corresponding to the two derived facts, together with the two modes store and push for the first one.

$\downarrow \text{OnST}(y) \leftarrow E(x, @y); \text{onST}(x); \neg \text{onST}(y)$ is coded by:

$$\frac{\begin{array}{l} \text{ST upd} = \emptyset \\ \text{E upd} = \emptyset \\ \forall y, y \in \text{onST upd} \leftrightarrow ((\exists x, \text{tree_body pre loc } x \ y) \wedge y = \text{loc}) \end{array}}{\text{compute_phase_store_onST_tree} \text{ pre loc upd}}$$

$\uparrow \text{OnST}(y) \leftarrow E(x, @y); \text{onST}(x); \neg \text{onST}(y)$ is coded by:

Definition `compute_phase_push_onST_tree` :=
`push compute_phase_store_onST_tree`

$\downarrow \text{ST}(\text{min}(x), y) \leftarrow E(x, @y); \text{onST}(x); \neg \text{onST}(y)$ is coded by:

$$\frac{\begin{array}{l} \text{onST upd} = \emptyset \\ \text{E upd} = \emptyset \\ \text{let } P d := \text{tree_body pre loc } x \ \text{loc } \text{in} \\ \forall x, P x \rightarrow \forall m, \text{is_min pre } m \ P \rightarrow \text{ST upd} = \{(m, \text{loc})\} \quad \forall x, \neg P x \rightarrow \text{ST upd} = \emptyset \end{array}}{\text{compute_phase_store_ST_tree} \text{ pre loc upd}}$$

Here `is_min pre m P` means that `m` is a smallest element `x` satisfying `P x`, but it does not ensure the existence of such an `x`.

One may wonder why we use relations everywhere instead of functions: relations are more general but less handy than functions especially in proofs. This matter of fact is indeed driven by the relational nature of datalog, on which Netlog is based: facts may be derived or not according to the body of rules and

available facts. Moreover, the sequential composition of functions with relations provides relations, hence even when trying to use functions in previous attempts, e.g. for `new_stores` and `new_push` in `local_round`, eventually turned out to be not general enough.

4 The Netlog Machine in Coq

Recall that in our computation model each node has an embedded Netlog machine, which implements a precise semantics formally specified in Appendix B. Our Coq formalization of the Netlog machine defines a specialization of the distributed computation model given in Section 2. This model is expressed in terms of two abstract relations, `new_stores` and `new_push`. The present section explains their definitions, according to the model given in the previous section.

We have seen in Section 3 how Netlog rules are represented in Coq in some examples. For store rules the type is `configuration -> nat -> local_data -> Prop` and for push rules `configuration -> nat -> list Bmsg -> Prop`. In order to manipulate them explicitly in a general setting, we assume two types `store_rule_name` and `push_rule_name`, as well as an appropriate semantics for each rule.

```
Variable store_rule_name push_rule_name : Set.
Variable store_sem_of : store_rule_name ->
  configuration -> nat -> local_data -> Prop.
Variable push_sem_of : push_rule_name ->
  configuration -> nat -> list Bmsg -> Prop.
```

Then, from a configuration `pre` and a given list of `store_rule_name`, the future local data to be stored at node `loc` is defined by:

$$\frac{}{\text{stores_of_list } pre \text{ } loc \emptyset \emptyset_{1d}} \quad \frac{\text{store_sem_of } r \text{ } pre \text{ } loc \text{ } updr \quad \text{stores_of_list } pre \text{ } loc \text{ } l \text{ } updl}{\text{stores_of_list } pre \text{ } loc \text{ } (r::l) \text{ } (updr \cup_{1d} updl)}$$

The list of `Bmsg` to be sent from `loc` in configuration `pre` is defined similarly by a predicate called `push_of_list`. A local round is defined by just applying the previous definitions to a list `srl` of store rule names and a list `prl` of push rule names, where each element of `store_rule_name` occurs exactly once; similarly for `push_rule_name`. Here is the formal definition.

```
Variable srl : list store_rule_name.
Variable prl : list push_rule_name.
Definition new_stores_mach :=
  fun cnf loc d => stores_of_list cnf loc srl d.
Definition new_push_mach :=
  fun cnf loc l => push_of_list cnf loc prl l.
Definition local_round_mach := local_round new_stores_mach new_push_mach.
```

For instance, for BFS, we provide the following rule names and semantics (again, we omit the definition for push rule names, which are similar).

```

Inductive bfs_store_rule_name : Set :=
  store_onST_initial | store_onST_tree | store_ST_tree.
Definition bfs_store_sem_of (r : bfs_store_rule_name) :=
  match r with
  | store_onST_initial => compute_phase_store_onST_initial
  | store_onST_tree => compute_phase_store_onST_tree
  | store_ST_tree => compute_phase_store_ST_tree
  end.
Definition bfs_store_rule_order :=
  store_onST_tree :: store_ST_tree :: store_onST_initial :: nil.

```

In practice, suppose that we need to exploit an assumption

```
bsr : bfs_synchronous_round cnf_pre cnf_post
```

First we specialize `bsr` to a suitable location `x`. Then expanding and destructing the definitions given in the current section yields an environment containing:

```

upd0, upd1, upd2 : bfs_data
Hupd0 : compute_phase_store_onST_tree pre loc upd0
Hupd1 : compute_phase_store_ST_tree pre loc upd1
Hupd2 : compute_phase_store_onST_initial pre loc upd2
eCnode : Cnode post loc = Cnode pre loc  $\cup_{1d}$  upd0  $\cup_{1d}$  upd1  $\cup_{1d}$  upd2  $\cup_{1d}$   $\emptyset_{1d}$ 

```

We are then in position to reason by cases on the facts contained in `Cnode post loc`, using knowledge specified by the representation of Netlog rules on the corresponding updates `upd0`, `upd1`, `upd2` or on the facts previously in `Cnode pre loc`.

5 Verification of a Tree Protocol

The correctness proof of the BFS protocol is based on its consistency with a centralized version of the protocol. Given a tree made of node `Id`'s `lloc` and edges `larc`, we define the arcs to be added by considering the neighbors of nodes in `lloc` which are not in `lloc`. We get `new_lloc lloc` and `new_larc lloc` by simple functional programs.

Our main theorem states that a synchronous round in the distributed synchronous version corresponds to the computation performed by `new_larc`. It is actually not enough to consider only `ST` facts, since `onST` plays a prominent role in the construction of new facts. Our invariant for the distributed BFS protocol is strengthened by `correct_onST cnf lc` which states that in configuration `cnf`, the facts `onST` available on any node are consistent with `lc`.

```
Definition correct_onST cnf lc :=  $\forall$  loc y, in_onST cnf loc y -> y  $\in$  lc.
```

Then we prove the following propagation properties.

Lemma `invar_In_0_lc` : $\forall lc, 0 \in lc \rightarrow 0 \in (\text{new_lloc } lc ++ lc)$.

Lemma `propag_correct_onST` :
 $\forall pre\ post, \text{bfs_synchronous_round } pre\ post \rightarrow$
 $\forall lc, 0 \in lc \rightarrow \text{correct_onST } pre\ lc \rightarrow$
 $\text{correct_onST } post\ (\text{new_lloc } lc ++ lc)$.

Lemma `propag_consistent_with` :
 $\forall pre\ post, \text{bfs_synchronous_round } pre\ post \rightarrow$
 $\forall lc\ la, 0 \in lc \rightarrow \text{correct_onST } pre\ lc \rightarrow$
 $\text{consistent_with } (\text{global_ST } pre)\ la \rightarrow$
 $\text{consistent_with } (\text{global_ST } post)\ (\text{new_larc } lc ++ la)$.

Here, `global_ST` represents the union of all ST facts on the network. The most difficult part is to prove Lemma `propag_correct_onST`, which requires a careful use of `correct_onST pre lc` simultaneously on several neighboring locations. It follows that the conjunction of the propagated properties is an invariant.

Definition `invar_cnf_lc_la` :=
 $\text{consistent_with } (\text{global_ST } cnf)\ la \wedge \text{correct_onST } cnf\ lc \wedge 0 \in lc$.

Theorem `consistent_with_invariant` :
 $\forall pre\ post, \text{bfs_synchronous_round } pre\ post \rightarrow$
 $\forall lc\ la, \text{invar } pre\ lc\ la \rightarrow$
 $\text{invar } post\ (\text{new_lloc } lc ++ lc)\ (\text{new_larc } lc ++ la)$.

Then proving properties on the distributed algorithm for BFS boils down to prove them on the centralized version provided by `(new_larc, new_larc)`, i.e traditional technology about functional programs. For example we have the following theorem, where `tree_mult lloc larc` holds if the list of nodes `lloc` and the list of arcs `larc` constitute a tree, which can be inductively defined in the usual way.

Theorem `invariant_new_lloc_larc` :
 $\forall lloc\ larc, \text{tree_mult } lloc\ larc \rightarrow$
 $\text{tree_mult } (\text{new_lloc } lloc ++ lloc)\ (\text{new_larc } lloc ++ larc)$.

Combining `consistent_with_invariant` and `invariant_new_lloc_larc` yields that after a synchronous round, the set of facts ST in the network still represents a tree if it was already a tree before this round. That is, `is_mtree (global_inST cnf)` holds for any configuration `cnf` in a run.

Lemma `invariant_inST_is_mtree` :
`invariant_round`
 $(\text{synchronous_round_mach } \text{bfs_store_rule_order } \text{bfs_push_rule_order})$
 $(\text{fun } cnf \Rightarrow \text{is_mtree } (\text{global_inST } cnf))$.

We check that initially the ST (reduced to the root) is a tree, then by applying a general result given in `distr_comp` on invariant properties, we get that in any synchronous run starting from the initial configuration, ST is always a tree.

Theorem `always_inST_is_mtree` :
 $\forall cnfi, \text{is_initial_bfs } cnfi \rightarrow$
 $\forall (r : \text{bfs_run } cnfi), \text{alw_run } (\text{fun } c \Rightarrow \text{is_mtree } (\text{global_inST } c))\ r$.

Besides this global property, one may (and should) wonder whether $ST(x, y)$ facts are located on relevant nodes, i.e. child nodes y in our case, so that this information could be used by a higher layer protocol for transmitting data towards the root. This is actually a simple consequence of Rules (3) and (4), since they ensure that $ST(x, y)$ can only be stored on y . This is easy to formalize in our framework.

6 Related work

Declarative languages have been first used in the context of networks for sensor networks. TinyDB [22] and Cougar [10] offer the possibility to write distributed queries in SQL. More interestingly, recursive query languages have been used to express communication network algorithms such as routing protocols [20] and declarative overlays [19]. Distributed query languages thus provide new means to express complex network problems such as node discovery [2], route finding, path maintenance with quality of service [5], topology discovery, including physical topology [4], secure networking [1], or adaptive MANET routing [17].

Using formal techniques for verifying communication protocols is far from being a new idea. “Formal Description techniques” were developed by telecommunication laboratories from the beginning of the 1980s in order to specify and verify protocols to be standardized at ITU and ISO. Three languages came out. Two of them, Estelle and SDL, are based on asynchronous communicating automata, while LOTOS is a process algebra based on CCS and CSP extended with algebraic data types [27]. Other approaches include input/output automata [21], or Unity and TLA, which combine temporal logic and transition-based specification [7, 15]. A common feature to these approaches is their focus on control, in particular how to deal with behaviors in a distributed framework. Typical issues include non-determinism, deadlock freedom, stuttering, fairness, distributed consensus and, more recently, mobility. Data is generally considered as an abstract object not really related to the behavior. This is relevant for many low-level protocols, such as transport protocols. However, this does not suit the needs of applications which aim at building up a distributed global information, such as topological information on the network (in a physical or virtual sense), as in routing tables, for example. To our knowledge, such problems have not been attacked by means of the above mentioned approaches. An explanation may be that the pieces of data involved in distributed computations are embedded in different components of the global configuration, and the previous formalisms make it difficult to isolate them or to consider them as a whole in reasoning. A clear feature of the current paper, compared with those previous approaches, is the emphasis on manipulating data in formal reasoning, which also drove us to use Coq as the verification tool. Beyond formal verification of distributed protocols, Coq has been successfully applied to ensure reliability of hardware and software systems in various fields, such as multiplier circuits [26], concurrent communication protocols [12], self-stabilizing population protocols [11], devices for broadband protocols [25], and compilers [16] to name a few.

Closely related to our work is [28], where a declarative network verifier (DNV) was presented which maps specifications written in the Network Datalog query language into logical axioms which can be used in theorem provers like PVS to validate protocol correctness. The reasoning based on DNV is for Datalog specifications of (eventually distributed) algorithms, but not for distributed versions of Datalog such as the one provided by Netlog. In other words, it only considers the highly abstract centralized behaviour of a network. In contrast, our development in this paper is to reason about the distributed behaviour of individual nodes which together yield some expected global behaviour of the whole network. Therefore, we need to involve deep subtleties on message passing and derivation of local facts, which are all absent in [28].

7 Conclusion

We developed a Coq library for verifying declarative protocols expressed in a rule-based language. This library includes the formalization of the distributed computation environment with the communication network, where both the synchronous and the asynchronous models of communication are formalized in very similar ways. The library also includes the embedded machine which evaluates the Netlog programs on each node. The Netlog programs are translated into straightforward Coq definitions. As a preliminary result we proved a topological property of a distributed data structure – a tree – constructed by a simple but subtle program. To our knowledge, such properties are difficult to handle in other approaches to the verification of distributed programs. From this experiment, we are in position to define a deep embedding for systematically deriving Coq encodings from the abstract syntax of Netlog rules, as well as dedicated tactics for handling tedious steps specific to Netlog, and then plan to further verify declarative protocols for routing, election, naming, and other fundamental distributed problems.

References

1. M. Abadi and B. T. Loo. Towards a declarative language and system for secure networking. In *Proc. NETB'07*, pages 1–6. USENIX Association, 2007.
2. G. Alonso, E. Kranakis, C. Sawchuk, R. Wattenhofer, and P. Widmayer. Probabilistic protocols for node discovery in ad hoc multi-channel broadcast networks. In *Proc. ADHOC-NOW'03*, 2003.
3. M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. Graph relabelling systems: a tool for encoding, proving, studying and visualizing - distributed algorithms. *ENTCS*, 51, 2001.
4. Y. Bejerano, Y. Breitbart, M. N. Garofalakis, and R. Rastogi. Physical topology discovery for large multi-subnet networks. In *Proc. INFOCOM'03*, 2003.
5. Y. Bejerano, Y. Breitbart, A. Orda, R. Rastogi, and A. Sprintson. Algorithms for computing qos paths with restoration. *IEEE/ACM Trans. Netw.*, 13(3), 2005.
6. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

7. K. M. Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., 1988.
8. Coq. The coq proof assistant reference manual. <http://coq.inria.fr/V8.1p13/refman/index.html>.
9. Coq. The Coq user contributions. <http://coq.inria.fr/contribs-eng.html>.
10. A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The cougar project: a work-in-progress report. *SIGMOD Record*, 32(4):53–59, 2003.
11. Y. Deng and J.-F. Monin. Verifying self-stabilizing population protocols with coq. In *Proc. TASE'09*, pages 201–208. IEEE Computer Society, 2009.
12. E. Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems*. PhD thesis, ENS Lyon, 1996.
13. T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proc. ACM SIGCOMM'05*, 2005.
14. S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *Proc. PADL'10*, volume 5937 of *LNCS*, pages 88–103, 2010.
15. L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
16. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL'06*, pages 42–54. ACM, 2006.
17. C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo. A declarative perspective on adaptive manet routing. In *Proc. PRESTO '08*, pages 63–68. ACM, 2008.
18. B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proc. ACM SIGMOD'06*, 2006.
19. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proc. SOSP'05*, 2005.
20. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM '05*, 2005.
21. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
22. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1), 2005.
23. G. Mainland, G. Morrisett, and M. Welsh. Flask: staged functional programming for sensor networks. In *Proc. ICFP'08*, 2008.
24. Y. Métivier and E. Sopena. Graph relabelling systems: A general overview. *Computers and Artificial Intelligence*, 16(2), 1997.
25. J.-F. Monin. Proving a real time algorithm for ATM in Coq. In *Types for Proofs and Programs*, volume 1512 of *LNCS*, pages 277–293. Springer, 1998.
26. C. Paulin-Mohring. Circuits as streams in coq: Verification of a sequential multiplier. In *Proc. TYPES'96*, volume 1158 of *LNCS*, pages 216–230. Springer, 1996.
27. K. J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL*. John Wiley & Sons, Inc., 1993.
28. A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative network verification. In *PADL '09: Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, pages 61–75, Berlin, Heidelberg, 2009. Springer-Verlag.

A Formal Verification with Coq

Coq is one of the most popular proof assistants for formal verification. It is based on a constructive type theoretic setting, called the *Calculus of (co-)Inductive*

Constructions (CIC), which can be summarized both as a polymorphic typed lambda-calculus enriched with universes, inductive and co-inductive types and a language for describing mathematical definitions and proofs [8, 6]. These two aspects are actually related thanks to the well-known Curry-Howard-De Bruijn isomorphism, which maps propositions to types and proofs to functional objects or strongly normalizing programs.

Let us illustrate some concepts and the syntax of Coq by a few examples. One of the most commonly used data structure is `list`. Let `A` be a type, a list whose elements are of type `A` can be inductively defined, with the usual constructors `nil` and `cons`:

```
Inductive list : Type :=
  | nil : list
  | cons : A -> list -> list.
```

Total recursive functional programs can be defined for lists. For example, the `In` predicate defined below checks if `a` occurs in the list `l`.

```
Fixpoint In (a:A) (l:list) {struct l} : Prop :=
  match l with
  | nil => False
  | b :: m => b = a \/ In a m
  end.
```

In order to ensure termination, a structurally decreasing argument is specified by `struct l`. Here we meet the realm of propositions `Prop`. A predicate over natural numbers for instance has type `nat -> Prop`. Given such a predicate `P`, a proof `p0` of `P 0` and a proof `step` of `forall n, P n -> P (S n)`, we can construct a proof of `P n` for all natural numbers `n`, using the following functional (primitive recursive) program:

```
Fixpoint natind (n:nat) {struct n}: P n := match n return (P n) with
  | 0 => p0
  | S q => step q (natind q)
  end.
```

The type of `natind` is `forall n, P n`, that is a *dependent type*, since the type of the result depends on the value of the argument; `step`, seen as a function from numbers `n` and proofs of `P n` and returning a proof of `P (S n)`, has a slightly more complex dependent type. In the `match` construct itself, the type of the result depends on the branch – it could be `P 0` or `P (S q)` for some `q`. Abstracting `P`, `p0` and `step` in `natind` yields a proof of the usual induction principle over natural numbers. As a function, it illustrates some important features of the type theory of Coq: polymorphism, inductive and dependent types.

Other constructs used in this paper, such as *records*, are special cases of inductive types (i.e with only one constructor; *fields* are just projections). When defining inductive types, dependent types can also be used for constructors. It is

especially convenient for formalizing algebraic structures (a carrier, operations and algebraic laws) and we use them extensively in the sequel. For example, graphs can be defined below, with two fields: `Vert` for vertices and `Edge` for edges.

```
Record Graph : Type := mkGraph {
  Vert : Type;                (* vertices *)
  Edge : Vert -> Vert -> Prop (* edges *)
}.
```

Like functions, relations are widely used mathematical concepts in formal verification. As an example, let `R` be a binary relation over natural numbers. Its transitive closure can be defined as follows.

```
Inductive TC (R: nat->nat->Prop): nat -> nat -> Prop :=
| TCO : forall x y, R x y -> TC R x y
| TCrec : forall x y z, R x z -> TC R z y -> TC R x y.
```

Some frequently used types such as `nat` and `list` are available in the standard library of Coq; by importing relevant packages, we can directly use the operations (e.g. `In` seen above) associated with lists. However, many other types are not in the library, and in this case they need to be defined from scratch. We have seen `Graph` for graphs above. We next consider trees which will be used to reason about algorithms for constructing spanning trees on connected graphs. We define abstract trees inductively as follows (from now on, we use the notation \forall instead of `forall` in order to save space; types of variables which can be easily inferred from the context are not explicitly given):

```
Variable Carrier : Set.
Inductive tree : list Carrier -> list (Carrier * Carrier) -> Prop :=
| root :  $\forall x$ : Carrier, tree (x :: nil) nil
| leaf :  $\forall lv\ le$ , tree lv le ->
   $\forall x\ y$ : Carrier, In x lv ->  $\neg$  In y lv ->
  tree (y :: lv) ((x,y) :: le).
```

A tree is built upon a set of vertices (represented by a list `lv`) and a set of edges (represented by a list `le` of pairs) by:

- the tree reduced to its root and no edge;
- from a tree upon `lv` and `le`, adding a new pendant vertex `y` and edge (x, y) such that $x \in lv$, $y \notin lv$, we obtain a tree upon `lv` extended with `y` and `le` extended with (x, y) .

Altogether, the features of Coq allow us to formalize mathematical theories in a typed and precise but still very general setting. Coq offers an environment where users can state mathematical definitions using types, concrete objects, functions over them, and then interactively prove theorems. Obvious proof steps are automated, but clever ones, e.g. inductive arguments or intermediate sub-goals, require user interactions.

B The embedded Netlog machine

Netlog programs are running on the nodes of the network. They produce facts to store as well as facts to be sent to other nodes. They are evaluated by a machine which implements a precise semantics [14], which has been defined by fixpoints in a way which is classical for rule-based languages such as Datalog. We present the semantics of a subset of the Netlog language of interest in this paper. The language is restricted to a subset of the language constructs, and moreover, the rules are applied only once at each round, unlike in [14] where a local fixpoint is computed at each local round.

We assume that all variables range over the sort (\mathbb{N}, \leq) , of the natural numbers. Given a finite set V of variables, a *valuation* over V is a mapping from V to \mathbb{N} . Let $\mathcal{V}(Var(r))$ be the set of valuations σ over $Var(r)$, the set of variables of a rule r .

Let an *instance* I be a finite interpretation of the relations of some schema S , which contains E , as well as some other relation symbols, such as *Route*, *ST*, etc. depending upon the programs. The *satisfaction* of the literals in the body of rule r by instance I and valuation σ is defined in a classical way, except for the universal literal, where: $(I, \sigma) \models \neg R(t_1, \dots, -, \dots, t_n)$ iff $R(\sigma(t_1), \dots, C, \dots, \sigma(t_n)) \notin I$, for any constant C . Assume the body of r , $body_r$, is $L_1; \dots; L_\ell$. We have $(I, \sigma) \models body_r$ iff $(I, \sigma) \models L_i$, for each $i \in [1, \ell]$.

The valuation of the head, $head_r$, of rule r can now be defined. The aggregation functions, which can only occur in the head of rules, require some care. Let $Var^{Agg}(head_r)$ be the set of simple variables in the head, which are not arguments of aggregation functions. Let $\tau \in \mathcal{V}(Var^{Agg}(head_r))$. We extend τ to $\mathcal{V}(Var(r))$ with respect to interpretation I , as:

$$[\tau]_{I,r} = \{\sigma \mid \sigma \in \mathcal{V}(Var(r)), \sigma(x) = \tau(x), \text{ for all } x \in dom(\tau), \text{ and } (I, \sigma) \models body_r\}.$$

In the sequel, we assume that $[\tau]_{I,r} \neq \emptyset$. We define $\tau(head_r)$ as follows. First, if $head_r$ contains only simple variables and is of the form: $R(x_1, \dots, x_n)$, then $\tau(head_r) = R(\tau(x_1), \dots, \tau(x_n))$. More generally, if it is of the form:

$$R(x_1, \dots, x_n, agg(y_1), \dots, agg(y_m))$$

where agg denotes an aggregate function on multi-sets, and $\{\{ \}$ denotes multi-set, then

$$\tau(head_r) = R(\tau(x_1), \dots, \tau(x_n), agg\{\{\sigma(y_1) \mid \sigma \in [\tau]_{I,r}\}\}, \dots, agg\{\{\sigma(y_m) \mid \sigma \in [\tau]_{I,r}\}\}).$$

We can now define the set of positive consequences of a program P over an instance I , $\Delta_P^+(I)$, as well as the set of consumed facts, $\Delta_P^-(I)$.

$$\Delta_P^+(I) = \{\tau(head_r) \mid r \in P, \tau \in \mathcal{V}(Var^{Agg}(head_r)), [\tau]_{I,r} \neq \emptyset\}.$$

$$\Delta_P^-(I) = \{R(\sigma(t_1), \dots, \sigma(t_n)) \mid r \in P, (I, \sigma) \models body_r, !R(t_1, \dots, t_n) \text{ in } body_r\}.$$

It is not hard to see that $\Delta_P^-(I) \subseteq I$.

Let us now distinguish between P_{\downarrow} the subset of store rules, and P_{\uparrow} of push rules in P . Note that store-and-push rules belong to both sets.

We describe the behavior of a Netlog program on one node, say α . At each local round, it takes as input the local data on α and the data pushed by neighbor nodes to α , (`local_round`) and produces updated local data, and data to be pushed to each of its neighbor (`communication`). The node also forwards messages, that are not used in the local computation. Its interaction with the rest of the network is defined by the *communication function*: $\mathcal{R}^{\alpha}(\ell)$, which maps each local round ℓ to the set of incoming messages on node α at local round ℓ .

Note that at each local round, the router sorts the incoming messages into two sets $\mathcal{L}^{\alpha}(\ell)$, of *received facts*, and $\mathcal{F}^{\alpha}(\ell)$, of *messages to forward* to other nodes depending upon their destination: $\mathcal{L}^{\alpha}(\ell)$ contains the facts extracted from messages received from other nodes, with destination α , “*nbg*” (the neighbor of the sender), or “*all*” (the message is broadcasted to all nodes). $\mathcal{F}^{\alpha}(\ell)$ contains the messages received from other nodes, with a destination different from α or destination “*all*”, which will be forwarded further to other nodes.

$$\begin{aligned}\mathcal{F}^{\alpha}(0) &= \emptyset; \\ \mathcal{F}^{\alpha}(\ell) &= \{(dest, fact) \mid (dest, fact) \in \mathcal{R}^{\alpha}(\ell); dest \notin \{\alpha, nbg\}\}; \\ \mathcal{L}^{\alpha}(\ell) &= \{fact \mid (dest, fact) \in \mathcal{R}^{\alpha}(\ell); dest \in \{\alpha, nbg, all\}\}.\end{aligned}$$

The computation relies on two *operators*, associated to program P , (i) for the data to store locally, Ψ_P^{\downarrow} , and (ii) for the data to push to other nodes, Ψ_P^{\uparrow} . They take as input the local instance I , and the received facts L .

- $\Psi_P^{\downarrow}(I, L) = \Delta_{P_{\downarrow}}^+(I \cup L) \cup (I \setminus \Delta_{P_{\downarrow}}^-(I \cup L))$ defines the *store operator*, producing facts to store.
- Ψ_P^{\uparrow} defines the *push operator*, producing messages to push:

$$\Psi_P^{\uparrow}(I \cup L) = \left\{ (dest, fact) \left| \begin{array}{l} fact \in \Delta_{P_{\uparrow}}^+(I \cup L); \text{ and} \\ \text{if } fact \text{ contains an address term } @\beta \\ \text{or } @all, \text{ then resp. } dest = \beta \text{ or } all; \\ \text{otherwise } dest = nbg. \end{array} \right. \right\}$$

When a local round ℓ starts, the node α has a local instance $\mathcal{I}^{\alpha}(\ell)$, and has received facts $\mathcal{L}^{\alpha}(\ell)$, and messages to forward $\mathcal{F}^{\alpha}(\ell)$. It then starts its computation, and produces a new local instance $\mathcal{I}^{\alpha}(\ell + 1) = \Psi_P^{\downarrow}(\mathcal{I}^{\alpha}(\ell), \mathcal{L}^{\alpha}(\ell))$, and a set of messages to push $\mathcal{P}^{\alpha}(\ell) = \Psi_P^{\uparrow}(\mathcal{I}^{\alpha}(\ell), \mathcal{L}^{\alpha}(\ell))$, which is then sorted by destination.

Let us now consider the communication between nodes. The messages to push are accumulated in $\mathcal{P}^{\alpha}(\ell)$. Their routes will be computed according to the knowledge node α has of the *Route* relation.

In the case of synchronous systems without failure, there is an explicit correspondence between the incoming and outgoing sets of messages.

Proposition 1. [14] *For synchronous systems without failure, we have for $l \geq 0$:*

$$\mathcal{R}^\alpha(0) = \emptyset,$$

$$\mathcal{R}^\alpha(\ell + 1) = \left\{ (dest, fact) \left| \begin{array}{l} \exists \beta \text{ s.t. } E(\beta, \alpha) \in I^\beta(\ell); \\ (dest, fact) \in \mathcal{P}^\beta(\ell); \text{ and} \\ \text{if } dest \notin \{\alpha, nil, all\} \text{ then} \\ Route(\beta, \alpha, dest) \in I^\beta(\ell) \end{array} \right. \right\}.$$

In the case of asynchronous systems, the function \mathcal{R}^α depends upon the distributed system, and in general might differ between two executions. The semantics is thus defined up to the system of communication function \mathcal{R}^α for each node α .

The semantics is defined as the local data store obtained on each node of the network, when no communication occurs anymore in the network. The termination is thus only implicit and globally defined. Clearly, programs can very well not terminate.