2012 IEEE Sixth International Symposium on Theoretical Aspects of Software Engineering

Formal Verification of Netlog Protocols

Meixian Chen
Dpt of Computer Science,
Shanghai Jiao Tong University
Shanghai, China
misam.chen@sjtu.edu.cn

Jean-François Monin
LIAMA & VERIMAG
Université de Grenoble 1 & CNRS
Grenoble, France
jean-francois.monin@imag.fr

***Abstract*—Data centric languages, such as recursive rule based languages, have been proposed to program distributed applications over networks. They greatly simplify the code, while still admitting efficient distributed execution, including on sensor networks. From previous work [1], we know that they also provide a promising approach to another tough issue about distributed protocols: their formal verification. Indeed, we can take advantage of their data centric orientation, which allows us to explicitly handle global structures such as the topology of the network. We illustrate here our approach on two non-trivial protocols and discuss its Coq implementation.**

***Keywords*-formal verification; Coq; distributed algorithms;**

I. Introduction

In the distributed setting, the certification of software systems turned out to be substantially more difficult than in the centralized setting. One reason is that the brute application of the state-oriented verification technology developed for sequential systems to distributed systems does not naturally fit all features of the latter, making things unnecessarily complex. For instance, processes actually don't read or write in arbitrary parts of the global state. And it is certainly not accidental that in the last two decades, much research on formal methods for distributed systems moved the focus from *states* to *events*. Well-developed formalisms studied in this area include process algebra, process calculi, extended communicating automata or transition systems and temporal logic. Typical issues include stuttering, fairness, non-determinism, deadlock freedom, distributed consensus and, more recently, mobility. Several verification tools for event-oriented properties have been developed and applied to different case studies, e.g., [2], [3], [4].

Still, many problems raised by distributed algorithms are hard to deal with in an event-oriented setting when, precisely, states play an essential role. A well-known example is the distributed algorithm for computing a minimum-weight spanning tree due to Gallager, Humblet and Spira [5]. Rigorous proofs made between 1987 and 2006 for GHS [6], [7], [8] are all intricate and very long (100 to 170 pages). Only [7] has been mechanically proof-checked. The point is that we need to reason on a global shape and, more generally, on data distributed on many locations. The very statement of the problem already involves a global view on pieces of data located at various locations in the system. This can be seen on previous work performed with the ACL2 proof-assistant, e.g., [9], [10]. As far as we know, all these works consider systems modeled at a rather low level of abstraction.

However, a new trend on distributed programming based on data-base concepts has been proposed [11], [12], [13], [14]. In particular, Netlog [14] provides a framework which is both data-centric and high-level: programs are expressed by means of Datalog rules augmented with information on the location and the transmission of facts. This greatly simplifies the code, while still admitting efficient distributed execution. From previous work [1], we know that such languages also provide a promising approach to the verification of distributed protocols.

We present here the encoding in Coq of the framework defined in [1], and we extend it in order to take non-monotonic features of Netlog into account. This includes a general library formalizing a distributed computation model based on message passing with either synchronous or asynchronous behavior, a representation of declarative rules of the Netlog language as well as their evaluation mechanism. As running examples, we consider a distributed algorithm which computes a breadth-first search (BFS) tree in a graph in the synchronous variant of Netlog and an optimized distributed version of Prim-Jarnìk algorithm (often referred to as Prim's algorithm [15], [16]), which computes a minimum-weight spanning tree in the asynchronous variant of Netlog. Though these case studies are simpler than GHS, they already involve subtle reasoning steps and allow us to validate our framework.

As for [1], our results are supported by a Coq development available in [17]. The main differences and improvements with relation to [1] are as follows:

- We provide the intuitive ideas on the reasoning behind the (rather dry) proof of BFS given in [1], and make it more accurate using *local* reasoning. Let us emphasize that the use of a proof-assistant was of much help when analyzing our initial proof.
- The Coq encoding of the main definitions is presented here and related to the more informal style used in [1]; additionally, we explain how Netlog rules are encoded (Section III-B).
- Non-monotonic features of Netlog are considered (the deletion of facts stored on a node): the change on the general model is small (the definition of a local round) but the impact on proof scripts is more important, our complete development about BFS was upgraded accord-

978-0-7695-4751-0/12 $26.00 © 2012 IEEE
DOI 10.1109/TASE.2012.19

ingly in order to update the methodology.
- A non-trivial distributed version of Prim's algorithm is provided, as well as a manual correctness proof. The reasoning is much more involved than for BFS; its formalization in Coq using the techniques described earlier is started.

The rest of this paper is organized as follows. Section II recalls the Netlog language, illustrates it with BFS and discusses the proof of the correctness of BFS. Section III presents our Coq formalization of Netlog. Section IV is dedicated to Prim's algorithm and its correctness proof. We conclude in Section V.

II. Proving a Distributed Algorithm in Netlog

A. The Netlog Language

Only the main constructs of Netlog 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 Netlog programs are installed on each node, where they run concurrently. Deduced facts can be stored on the same node at which they are deduced, or sent to other nodes. The rules of a program are applied in parallel: in a given state, all bodies are evaluated over the local instance of a node and results (heads) are computed, using facts either stored on the node or pushed by a neighbor – the evaluation order is irrelevant since this step is side-effect free; then the results are stored or pushed according to the specification. The symbol $\updownarrow$ 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 symbol "@" in the body of a rule forces the latter to run on a precise node. For example, in rule (2) below, the literal $E(x, @y)$ indicates that the rule runs on node y. The sequence made of the evaluation of all rule bodies followed by the updating stage on a given node is called a *local round*.

Netlog comes with two flavors. In the *asynchronous* setting, a run consists in iterating the choice of a node, followed by the execution of a local round on this node. We assume fairness of executions, in order to ensure that a node able to progress eventually performs a local round. In the *synchronous* setting, local rounds are performed in parallel: bodies are evaluated simultaneously on all nodes, then updates are performed in parallel as well.

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. The *consumption* of a fact F is indicated by an exclamation mark "!" (non-monotonicity comes from this construct); this feature is exploited for Prim's algorithm in Section IV. *Aggregation functions*, such as min in the next example, can also be used in the head of rules to aggregate over all values satisfying the body of the rule.

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 $ST(\alpha, \beta)$) holds iff the node α (respectively the edge (α, β)) is already on the intended tree.

Synchronous Rooted BFS Tree

$$\updownarrow onST(x) \leftarrow @x = 0. \quad (1)$$

$$\left.\begin{array}{r} \updownarrow onST(y) \\ \downarrow ST(min(x), y) \end{array}\right\} \leftarrow E(x, @y); onST(x); \neg onST(y). \quad (2)$$

Rule (1) is enabled 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 (2) 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 (1) and (2), the consistency of E with the physical edge relation holds forever.

This algorithm aims at constructing a suitable distributed relation ST. More precisely, we prove below that the relation ST actually defines a BFS tree.

B. Proof of the Correctness of BFS

From a global perspective, it is pretty obvious that this algorithm makes up a BFS. However, the distributed implementation introduces additional details in terms of messages and synchronization. Furthermore, the fact that decisions are taken on the basis of local knowledge, which may be obsolete if it is related to a distant node, has to be taken into account. In the current version we limit our-self to *local* reasoning as far as possible: in [1], auxiliary invariants are statements universally quantified over all nodes of the network; a closer analysis of *proofs* showed then that for propagating an assertion $\forall n, A(n)$, where n stands for a node, from a configuration to the next one (after performing a transition), only a small subset of the quantified nodes is used for a given n: typically, n itself or its neighbors in the case of BFS. In the current version of the script, this remark is lifted to the level of *statements* and we try to keep locality of reasoning as far as possible. Only the very last step integrates the local invariants together in order to provide the global view. Note that we work with weaker – hence sharper – invariants, because the exact amount and structure of information needed for propagating assertions is better tracked.

The proof technique we use is to consider a transition system where each transition transforms simultaneously (our model of) the concrete distributed system and an imaginary oracle, which represents a centralized view of BFS. Note that correctness of the algorithm relative to the oracle includes

safety and liveness at the same time, since the oracle progresses at each round.

More precisely, we prove that (1) and (2) perform exactly the same computation as a suitable functional program, which operates on a data structure composed of two lists: a list of nodes generally denoted by lc and a list of arcs generally denoted by la which, intuitively, represent the expected relations $onST$ and ST. Our oracle is made of two functions respectively called `new_lloc` and `new_larc`, which respectively compute the new list of locations and of arcs to be added to a centralized configuration $\langle lc, la\rangle$ in order to reach the next one. To this effect we first compute the set `neighbors_candidates` lc of all arcs (x, y) such that x is in lc inductively on lc, removing the arcs such that y is in lc. Then, for each fixed y, we select the minimum x among $\{x \mid (x, y) \in$ `neighbors_candidates` $lc\}$. This yields `new_larc` lc, and `new_lloc` lc is obtained by mapping the second projection on `new_larc` lc.

Definition 1 (global consistency with)**.** *We say that a distributed configuration $\mathcal{C}$ is* globally ST-consistent with *a list of arcs la and a list of nodes lc iff the membership to the set of all ST facts in $\mathcal{C}$ is equivalent to the membership to la. Similarly, we say that $\mathcal{C}$ is* globally $onST$-consistent with lc *iff the membership to the set of all $onST$ facts stored in $\mathcal{C}$ is equivalent to the membership to lc.*

The main objective is to prove that global ST-consistency holds forever. The proof is by co-induction: global ST-consistency holds initially and, if $\mathcal{C}$ is globally ST-consistent with la and lc, we prove that $\mathcal{C}'$ is globally ST-consistent with la ++ `new_larc` lc and lc ++ `new_lloc` lc, where $\mathcal{C}'$ is the next distributed configuration after a synchronous round.

In order to prove the second statement, a stronger invariant is needed. The engine of BFS is $onST$, as can be seen on rules (1) and (2). In fact, $onST$-consistency is not enough because, performing rule (2) on node y requires that the *knowledge about $onST$ at y* is correct and complete with relation to $onST(x)$. The definitions we need are as follows.

Definition 2 (local correctness of $onST$)**.** *A configuration is* $onST$-correct at node loc *if and only if, given any fact $onST(z)$ which is visible at loc (either because it is stored, or because this fact is available on a channel to loc), then it must hold on the oracle as well.*

Definition 3 (local completeness of $onST$)**.** *A configuration is said to be* $onST$-complete at node loc *if and only if, whenever $onST(loc)$ holds on the oracle, then it is stored at loc.*

The local correctness of $onST$ happens to propagate independently from its consistency (provided $onST(0) \in \mathcal{C}$ holds; this trivially holds forever). However completeness is more subtle. We need a lemma stating the completeness of evaluation of the body of rule (2): given a distributed configuration satisfying some precondition P and $onST(y) \notin \mathcal{C}$, and an edge $x \to y$ such that if $onST(x) \in \mathcal{C}$, then the body of rule (2) holds at y. What is needed for the precondition P? Completeness of $onST$ everywhere is not enough: $onST(x) \in \mathcal{C}$ yields only that $onST(x)$ is visible at x, and nothing at y. The precise additional assumption which is needed is as follows: if $onST(x)$ is stored at x, then $onST(y)$ is stored at y or $onST(x)$ is arriving at y (both things can happen simultaneously as well). We then say that edge $x \to y$ is *good*. We first remark that goodness is invariant.

Lemma 1. *If an edge $x \to y$ is good in a distributed configuration, then it is still good after a synchronous transition.*

Note that goodness is about distributed configurations (without oracle). The previous lemma is then purely about the distributed behaviors of the algorithm. The key lemma can then be stated as follows.

Definition 4 (ready at)**.** *Let y be a node. A configuration is said to be* ready at y *if it is $onST$-complete at y, $onST$-correct at y, $onST$-complete at all neighbors of y and good at all edges $x \to y$.*

Lemma 2. *Let y be a node. If a configuration is ready at y, then it is still $onST$-complete at y after a synchronous transition.*

Next we can prove that if a configuration is ready at y, then ST-correctness and ST-completeness at y is preserved by a synchronous transition. Altogether we get that if a configuration is ready, ST-correct and ST-complete *everywhere* – here we combine all local propagation properties into a global invariant – and if furthermore, $onST(0) \in \mathcal{C}$, then this conjunction still holds after a synchronous transition. This invariant allows us to conclude that the synchronous distributed algorithm defined by rules (1) and (2) behaves exactly as specified by the oracle.

III. Coq Formalization of Netlog Programs

A. The Netlog Machinery

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 :=  In m (neighbors n).
```

At this level, we make no assumption on the edge relation except finiteness, which is always satisfied in practice. In particular we don't require it to be symmetric and self edges are allowed. Additional assumptions can be introduced if needed but, for example, BFS works in the general case.

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 a value representing the *empty set* of facts and two binary functions returning respectively the *union* and the *set difference* of two sets of facts. 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. In other words, a big message represents a set of messages having the

same destination. The global state of the system has the type config defined as follows.

```
Record config: Set:= mk_config {
  Cnode: nat -> local_data;
  Cedge:
   ∀ src dst: nat, edge src dst -> local_data}.
```

A *local round at* loc (a node Id) relates an actual configuration pre to a new configuration mid and a list out of big messages from loc. Furthermore, incoming edges are cleared. After consumed facts gathered in the data d are deleted, the new data s is to be stored on loc, as well as values in out, depending only upon the data in pre. They are defined by three auxiliary functions, respectively new_deletes, new_stores and new_push, which are themselves defined on the Netlog machine on the node. The main difference in the general model given here and the one presented in [1] lies in new_deletes. We first give a definition of a local round in the style of [1]. In the notation used there, $|loc|^{cnf}$ (respectively $|x{\rightarrow}y|^{cnf}$)) represents the set of facts available at node loc (respectively at edge $x{\rightarrow}y$) in configuration cnf.

$$local\_round(loc, \mathbf{pre}, mid, out) \stackrel{\text{def}}{=} \begin{cases} \exists s, new\_stores(\mathbf{pre}, loc, s) \wedge \\ \quad \exists d, new\_deletes(\mathbf{pre}, loc, d) \wedge \\ \quad\quad |loc|^{mid} = (|loc|^{pre} - d) \cup s \\ new\_push(\mathbf{pre}, loc, out) \\ \forall\, x \in neighbors(loc), |x{\rightarrow}loc|^{mid} = \emptyset \end{cases}$$

This is formally defined in Coq using an inductive definition as follows[1].

```
Inductive local_round (loc: nat)
  (pre: config)
  (mid: config) (out: list Bmsg): Prop :=
  | mk_LR:
  (∃ s, new_stores pre loc s ∧
    ∃ d, new_deletes pre loc d ∧
    Cnode mid loc = (dif (Cnode pre loc) d) ∪ s ->
    new_push pre loc out  ->
    (∀ src (e: edge src loc),
                    Cedge mid e = empty) ->
    local_round loc srl prl pre mid out.
```

This definition expresses that a local round relating pre, mid and out at loc needs the following components: a proof that a suitable s and a suitable d exist, a proof that pre loc and out are related according to new_push, and a proof that for all nodes src related to loc by edge e, the data stored on e in configuration mid is empty.

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

```
Inductive no_change_at (loc: nat)
  (pre mid: config): Prop :=
  | mk_NCA:
    Cnode mid loc = Cnode pre loc  ->
    (∀ src (e: edge src loc),
    Cedge mid e = Cedge pre e) ->
    no_change_at loc pre 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.

```
Inductive communication (loc: nat)
  (mid: config) (out: list Bmsg)
  (post: config): Prop :=
  | mk_comm:
    Cnode post loc = Cnode mid loc ->
    (∀ (dst: nat) (e: edge loc dst),
     Cedge post e = find dst out ∪ Cedge mid e) ->
    communication loc mid out post.
```

Here, 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 as a 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.

```
Inductive async_round (pre post:config):Prop :=
  | mk_AR:
      ∀ loc: nat, ∀ mid: config,
      ∀ out, local_round loc pre mid out ->
      (∀ loc', loc <> loc' ->
         no_change_at loc' pre mid) ->
      communication loc mid out post ->
      (∀ loc', loc <> loc' ->
         communication loc' mid nil post) ->
      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. We can define a synchronous round using similar lines.

```
Inductive sync_round (pre post:config):Prop :=
  | mk_SR:
     ∀ mid: config,
     (∀ loc: nat,
        ∃ out, local_round loc pre mid out ∧
            communication loc mid out post) ->
     sync_round pre post.
```

B. Encoding of Netlog Rules

Available facts are either stored on the node or pushed by a neighbor. This is formally defined in Coq as follows.

```
Inductive inFact (X:Set)
     (prj: local_data -> list X)
     (cnf: config) (loc: nat) : X -> Prop :=
```

[1]In Coq, all data structures boil down to inductive definitions, even if there is no recursions or if there is only one constructor (tuples). In recent versions of Coq special keywords are available but, as a matter of taste, we choose to stick to the use of Inductive. However, we sometimes use Record for tuples, when we need to conveniently name the projection functions (fields) at once.

```
  | Node_in: ∀ x, In x (prj (Cnode cnf loc)) ->
             inFact prj cnf loc x
  | Edge_in: ∀ neighbor (e: edge neighbor loc),
             ∀ x, In x (prj (Cedge cnf e)) ->
             inFact prj cnf loc x.
```

The actual `local_data` needed in the BFS protocol is just a triple.

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

Rules are encoded in Coq according to a systematic method, which is illustrated on rule (2). We first introduce the inductive definition corresponding to its body $E(x, @y); onST(x); \neg onST(y)$ as follows:

```
Inductive tree_body (cnf: config bfs_data)
                    (loc: nat) (x y:nat) : Prop :=
 | TB : in_E cnf loc (x,y) -> in_onST cnf loc x ->
        ¬in_onST cnf loc  y ->
        tree_body cnf loc x y.
```

Here, `in_E` is a specialization of `inFact` to `E`, and similarly for `in_onST`. Netlog rules are then encoded along the following scheme.

$\downarrow OnST(y) \leftarrow E(x, @y); onST(x); \neg onST(y)$ is encoded by:

```
Inductive compute_phase_store_onST_tree
    (pre : config bfs_data) (upd : bfs_data)
    (loc : nat) : Prop :=
  | Cstore_onST_tree :
    ST upd = nil -> E upd = nil ->
    (∀ x, tree_body pre loc x loc <->
                        In loc (onST upd)) ->
    compute_phase_store_onST_tree pre upd loc.
```

$\uparrow OnST(y) \leftarrow E(x, @y); onST(x); \neg onST(y)$ is encoded by:

```
Inductive compute_phase_push_onST_tree
    (pre : config bfs_data)
    (updest : list (nat * bfs_data))
    (loc : nat) : Prop :=
| Cpush_onST_tree :
  (∀ d u,
     In (d, u) updest <->
        compute_phase_store_onST_tree pre u loc
          ∧ edge loc d) ->
  compute_phase_push_onST_tree pre updest loc.
```

IV. Prim's algorithm

The asynchronous version of Netlog is used here. We assume a connected weighted graph, $G = (V, E, w, R)$, where V is the set of nodes, E is the set of Edges, the weight $w : E \to R+$, satisfies $w(u, v) = w(v, u)$ for all edge (u, v) in E, and $R \in V$ a distinguished node called the root. In addition, we assume that w is injective. In other words, weights of different edges are distinct and then the MST is unique.

The original algorithm [15], [16] starts from the root R and constructs successive fragments of the MST, by adding to the current fragment its minimal outgoing edge at each step. This algorithm is easy to translate to centralized Netlog.

As for BFS, the spanning tree is represented by facts $OnST(a)$ and $ST(a, b)$. Moreover, we define an intermediate relation $MWOE(x)$ to designate the minimal out-going weight at the current stage. The centralized Prim's algorithm can be expressed as follows. We need a non-monotonic feature of Netlog. More precisely, the symbol "!" indicates the consumption of a fact in the body: this fact will be deleted when committing the rules.

MST-Prim-Seq

$$\left.\begin{array}{r} OnST(x) \\ MWOE(min(m)) \end{array}\right\} \leftarrow \left\{\begin{array}{l} Root(x) \\ E(x, y, m) \end{array}\right. \quad (3)$$

$$\left.\begin{array}{r} ST(x, y) \\ OnST(y) \end{array}\right\} \leftarrow \left\{\begin{array}{l} OnST(x) \\ \neg OnST(y) \\ E(x, y, m) \\ !MWOE(m). \end{array}\right. \quad (4)$$

$$MWOE(min(m)) \leftarrow \left\{\begin{array}{l} OnST(x) \\ \neg OnST(y) \\ E(x, y, m) \\ \neg MWOE(m). \end{array}\right. \quad (5)$$

After initialization (rule (3)), this program alternately executes two phases: compute the minimal outgoing edge's weight m (given by $MWOE(m)$ in rule (5)), then find the corresponding edge and add it to the MST (rule (4)).

The verification of the correctness of the centralized version of Prim's algorithm is not difficult [15], [16].

A. Distributed Prim's algorithm in Netlog

In the design of a distributed implementation of the above algorithm, the ternary relation $E(x, y, m)$ of the centralized version is represented by binary facts $Weight(y, m)$ locally stored at x; as for BFS, $ST(x, y)$ is stored on the node y to remember its parent x, and $OnST(x)$ is stored on x.

The main issue is the representation of rule (5): all m satisfying its body have to be collected. The obvious place for computing $min(m)$ is the root. A simple idea consists of broadcasting queries from the root to the leaves, asking the $MWOE$ from each of them, then returning the results to the root. Rule (4) can be represented in a similar way, by broadcasting an invitation to insert the new $MWOE$ and waiting for an acknowledgement from all leaves – only one leaf will actually perform the insertion, the others will just send an acknowledgement back to the root.

However, we are able to achieve the same result with much less messages. To this effect, each node maintains suitable additional information, namely facts $WTable(y, m)$, so that queries and invitations can be directed along the suitable branch of the current spanning tree. Intuitively, a fact $WTable(y, m)$ stored on some node x of the spanning tree is intended to indicate the weight of the outgoing edge of smallest weight beyond y, when the edge (x, y) is on the spanning tree. A further subtle optimization is performed when adding a new leaf to the spanning tree: no check is made on the status ($OnST$ or $\neg OnST$) of its neighbors. Therefore, $WTable(y, m)$ contains the weight of the MWAE (minimum weight alive edge) below y, rather than the $MWOE$.

Definition 5 (dead, alive)**.** *An edge (a, b) such that a is in $OnST$ is either* alive*,* dead*, or a member of ST: it is initially* alive *(as soon as a becomes a member of $OnST$) and becomes*

dead *when a transmits to b an invitation to become a member of $OnST$ – the corresponding $WTable$ entry at a is then removed. Later, (a, b) will turn to ST if $\neg OnST(b)$ or will definitely remain dead if $OnST(b)$, without this having any significance with regard to values of MWAE.*

Note that the same message is used for inviting a node b to join the spanning tree and for querying the next MWAE below b. This yields the following Netlog program.

MST-Prim-Dis

$$\left.\begin{array}{r}\downarrow OnST(loc)\\ \downarrow WTable(y,n)\\ \uparrow Down(loc,@loc,min(n))\end{array}\right\} \leftarrow \left\{\begin{array}{l}Root(@loc)\\ Weight(y,n)\end{array}\right. \quad (6)$$

$$\uparrow Down(loc,@y,m) \leftarrow \left\{\begin{array}{l}Down(x,@loc,m)\\ !WTable(y,m)\\ \neg Weight(x,m)\end{array}\right. \quad (7)$$

$$\left.\begin{array}{r}\downarrow OnST(loc)\\ \downarrow ST(x,loc)\\ \downarrow WTable(z,n)\\ \uparrow Up(@x,loc,min(n))\end{array}\right\} \leftarrow \left\{\begin{array}{l}Down(x,@loc,m)\\ \neg OnST(loc)\\ Weight(z,n)\\ n \neq m\end{array}\right. \quad (8)$$

$$\uparrow Up(@x,loc,dead) \leftarrow \left\{\begin{array}{l}Down(x,@loc,m)\\ OnST(loc)\\ Weight(x,m)\end{array}\right. \quad (9)$$

$$\downarrow WTable(y,m) \leftarrow \left\{\begin{array}{l}Up(@loc,y,m)\\ \neg WTable(\_,m)\\ m \neq dead.\end{array}\right. \quad (10)$$

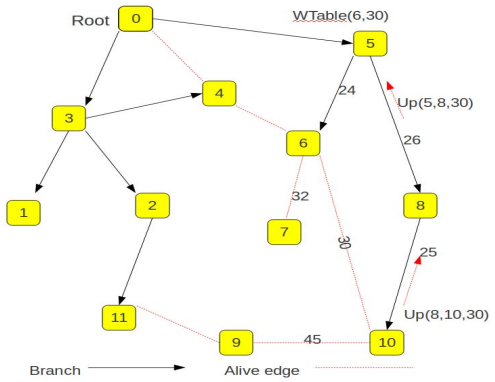
$$\uparrow Up(@x,loc,min(m,n)) \leftarrow \left\{\begin{array}{l}Up(@loc,y,m)\\ WTable(z,n)\\ \neg WTable(\_,m)\\ ST(x,loc)\end{array}\right. \quad (11)$$

$$\uparrow Down(loc,@loc,min(m,n)) \leftarrow \left\{\begin{array}{l}Up(@loc,y,m)\\ WTable(z,n)\\ \neg WTable(\_,m)\\ Root(loc)\end{array}\right. \quad (12)$$

$$\uparrow Down(loc,@y,m) \leftarrow \left\{\begin{array}{l}Up(@loc,y,m)\\ WTable(\_,m)\end{array}\right. \quad (13)$$

Facts $Down$ and Up are messages. After initialization of the algorithm at the root (rule (6)) $Down$ goes from the root (rules (6) and (12)) along the branch of the spanning tree leading to the MWAE (rule (7)), until it turns the selected alive edge to an ST edge (rule (8)) or to a dead edge (rule (9)); by convention, the minimum of the empty set of weights is the special value $dead$ assumed to be greater than any weight; then, if there is no outgoing edge from loc when triggering rule (8), the aggregation $min(n)$ evaluates to $dead$. Rule (9) runs when discovering that the invited node links two $OnST$ nodes. Note that rule (7) deletes the relevant $WTable$ entry because the information contained there becomes unreliable.

Then, an Up message carrying the minimum weight alive edge (or $dead$, the maximum weight value, if no one exists) is sent back to the root (rule (11)), restoring $WTable$ (rule (10)) and, eventually, generating a new $Down$ at the root (rule (12)). Note that $WTable$ is not restored when the corresponding branch is dead. In rules (11) and (12), $min(m, n)$ represents the least value among m and the min-aggregation of all n satisfying the body.



Node 5 first emitted an invitation $Down$ to node 6; the conflict appears in node 5 when receiving $Up(5, 8, 30)$ after node 10 joined the ST fragment, and is solved by rule (13) which eventually removes alive edge 30.

Figure 1. Fake invitation

Rules (10) to (12) assume that the weight m carried by an arriving Up is not already stored in $WTable$. Rule (10) guarantees that $WTable$ stores a distinct weight, in order to ensure uniqueness of the destination of $Down$ in a future execution of rule (7). However, the distributed setting makes it possible for a MWAE to arrive from two different branches to the same node: this happens when, and only when, this edge links two $OnST$ nodes and has not yet been recognized as a dead edge (see Figure 1). The fake invitation rule (13) fixes this issue, by enforcing the eventual triggering of rule (9) and appropriate cleaning.

B. Certification of Distributed Prim's Algorithm

This section presents the main ideas of the proof. Our goal is to verify that the program satisfies two properties: safety – each new joining edge is the minimal outgoing edge of the current fragment, which means that the distributed ST relation is consistent with the expected centralized view; and liveness – the minimal weight outgoing edge of the current fragment will eventually be added to the ST. To this effect we prove various technical lemmas. A key step states that $WTable(d, w)$ records the minimal weight w among alive edges in branch d and Up coming from d carries the minimal weight w among alive edges of branch d. As a corollary, $Down$ generated at the root contains the MWAE of the current fragment. Another lemma states that rule (13) behaves as expected. Combining them we get:

Theorem 1 (Safety)**.** *Each new joining edge is the minimal outgoing edge of the current spanning tree.*

Additionally, we can prove a liveness property.

Theorem 2 (Liveness)**.** *The minimal weight outgoing edge of the current fragment will eventually be added to the ST.*

Corollary 3 (Termination)**.** *All nodes will eventually belong to the spanning tree.*

C. Detailed Proofs

Lemma 4. *(i) If a node x is a member of $OnST$, then $OnST(x)$ is stored on x. (ii) Any node which is not a member of $OnST$ contains an empty $WTable$. (iii) Running rules* (6) *and* (8) *on some node creates of a new $WTable$ on this node.*

Proof: (i) and (iii) are easy inductive invariants, by inspection of the rules; note that initially, no node is a member of $OnST$ and all $WTable$ are empty. (iii) is a consequence of (i) and (ii). ■

Lemma 5. *In any node, the facts stored in $WTable$ contain distinct values in their second argument.*

Proof: Only rules (6), (8) *and* (10) *may store values in $WTable$. For rules* (6), (8) *the third argument is a weight, we then use the fact that all weights are distinct as well as Lemma 4 (iii). For rule* (10)*, the conclusion follows from the second literal of the body.* ■

Lemma 6. *In any configuration there is at most one message among $Down$ and Up.*

Proof: This is not difficult to check by chasing the rules of the program. Lemma 5 and rule (7) *guarantee that only one $Down$ is sent along the suitable branch.* ■

Lemma 7. *(A) $WTable(d, w)$ records the minimal weight w among all the alive edges of the branch d. (B) $Up(x, d, w)$ coming from node d carries the minimal weight w among all the alive edges of branch d.*

Proof: (A) and (B) obviously hold in the initial configuration. Now, assume (A) and (B) in a given configuration. Let us first show that (A) holds in the next configuration obtained after a round. $WTable$ could be changed in three possible ways. (i) creation (rules (6) *or* (8)*): for each neighbor d, the corresponding branch starting contains just the edge to d, which is alive and stored in $WTable$ according to rules* (6) *and* (8)*, then (A) holds in the next configuration; (ii) deletion: if a node a passes $Down$ to d using rule* (7)*, the corresponding $WTable$ entry for d is deleted at a and the edge (a, d) is no longer alive, hence (A) holds in the next configuration; (iii) updating: when node a receives Up from node d by rule* (10)*, Up carries the minimal weight m of branch d by assumption (B) on the current configuration, and m is stored in $WTable$ for branch d if m is not $dead$, ensuring (A) in the next configuration.*

Next let us show that (B) holds as well in the next configuration. Up is emitted from in two possible ways: (i) with a dead value in rule (9)*, indicating that there is no more alive edge; (ii) with the minimal weight w among alive edges below itself, by rules* (8)*, and* (11)*; using assumption (A) for all $OnST$ node from this node, we get that (B) holds in the next configuration.* ■

Corollary 8 (MWAE)**.** *The value w carried by a message $Down(root, root, w)$ generated in rule* (12) *is actually the MWAE of the current spanning tree.*

Lemma 9. *Only an $OnST$ node can pass a $Down$ message.*

Proof: $Down$ is passed along $WTable$, which only exists in $OnST$ nodes by Lemma 4 (ii). ■

Lemma 10 (ST)**.** *(i) When a new $ST(x, loc)$ is added, then x is an $OnST$ node and loc is a $\neg OnST$ node; this happens only by triggering rule* (8)*. (ii) The set of ST facts makes up a tree.*

Proof: For (i), only rule (8) *can add a fact $ST(x, loc)$, i.e., when a $\neg OnST$ node loc receives $Down$ from node x. Lemma 9 tells us that moreover x is an $OnST$ node. (ii) is an inductive invariant, using (i) when rule* (8) *is triggered.* ■

Lemma 11. *$Down$ generated in rule* (13) *carries the weight of an edge connecting two $OnST$ nodes.*

Proof: Rule (13) *is executed at node loc upon reception of an Up from branch y carrying some weight m already stored in $WTable$ for another branch x. By uniqueness of weights in the graph, and since ST is a tree by Lemma 10 (ii), m is the weight of some alive edge (a, b), by Lemma* (7)*(A) for branch y, and of the symmetric alive edge (b, a), by Lemma* (7)*(B) for branch x. By definition 5, a and b are then in $OnST$.* ■

Proof of Theorem 1: By Lemma 10 (i), any edge (a, b) joining ST is an outgoing edge from the current fragment, and its weight w is carried by a $Down$ message. In addition, this $Down$ cannot have been generated by rule (13), because, from Lemma 11, b would be $\neg OnST$. This $Down$ was then generated by rule (12) and, by Corollary 8, w is the weight of the minimal alive edge of the current tree. Altogether (a, b) is actually the MWOE of the current spanning tree.

Lemma 12. *When a node receives Up with a non dead value w, it stores w in $WTable$.*

Proof: Trivial from rule (10)*.* ■

Lemma 13. *When a node loc receives $Down(x, loc, w)$ from node x and $Weight(x, loc) \neq w$ (body of rule* (7)*), then at node loc, $WTable$ must contain an entry with value w.*

Proof: If x passes $Down$ to loc, we know that, before $Down$ is sent, there is a $WTable$ entry with parameters loc and w weight value stored at x. Since $Weight(x, loc) \neq w$, $WTable(y, w)$ was generated by an Up coming from branch loc, i.e., by rule (8)*, or* (11)*, from Lemma 12, then it also stored a $WTable$ entry with value w at node loc. This entry could not be removed because rule* (7) *had no opportunity to be enabled on node loc, as a consequence of Lemma 6.* ■

Lemma 14 (run)**.** *While there is an alive edge in the graph, there is exactly one message among Up and $Down$ and the program runs without deadlock.*

Proof: In the initial configuration, exactly one $Down$ message is generated by rule (6)*. Then by inspection of the rules, Lemmas* (6), (7) *(A) and* (10) *(ii), we always have exactly one message $Down$ or Up in the following configurations, except, at the root, when the $WTable$ is empty, which means that the set of alive edges is empty by Lemma* (7) *(A). Now we need to prove that when a message $Down$ or Up exists, there is no deadlock (a rule can be triggered). This follows from (A) and (B) below.*

(A) A message $Down$ exists in three situations: (i) it is a newly generated message by rule (6)*,* (12)*, or* (13)*; triggering rule* (6) *or* (12) *immediately stores the value w carried by $Down$ in the local $WTable$, while triggering* (13) *has the same effect by Lemma 12; then $Down$ will be passed down to the right branch using rule* (7)*. (ii) a node loc receives $Down$ from node x, and $Weight(x, loc) \neq w$; again, rule* (7) *can be applied from Lemma 13; (iii) a node loc receives $Down$ from node x, and $weight(x, y) = w$ ($Down$ reaches the target edge); a rule among* (8) *and* (9) *is triggered, generating Up at the same time.*

(B) From rules (8) *and* (9)*, we see that after a node passes $Down$ to the suitable edge, it will receive Up from the same edge. If a node receives Up with weight value w, either there is already a $WTable$ with the same w stored locally, then it consumes Up and generates $Down$ by rule* (13)*; or it passes Up to its parent along ST. When $Root$ receives Up and gets a non-empty $WTable$, it triggers rule* (12)*.* ■

Proof of Theorem 2: Let us say that an oriented edge (a, b) is *outside* when it is alive or when a is not $OnST$. From definition 5, an edge is either outside, dead or in ST. When a $Down$ message reaches a leaf b from node a, rule (8) or (9) is triggered; new edges (b, x) may become alive if b becomes $OnST$, which does not change the cardinality of outside edges; at the same time (a, b) turns from alive to either dead or ST; altogether, the number of outside edges decreases by one.

Now, by Lemma 14, a configuration which contains alive edges contains a message among Up and $Down$, and this message propagates along ST, eventually reaching the root (for Up) or a leaf (for $Down$) since ST is a tree by Lemma 10. In both cases, a $Down$ message is eventually generated and reaches a leaf where the number of outside edges is decremented. The theorem results from the finiteness of the initial number of outside edges.

V. Conclusion

It is well-known that distributed algorithms may have very subtle behaviors, which make their design and proofs rather delicate. Moreover, once a proof is done, we still have to consider its maintenance. It is very easy to go from a correct system to a mistaken one through seemingly innocuous changes and, in practice, implementations of correctly designed protocols commonly introduce such modifications.

The experience reported here supports the view that rule-based languages such as Netlog provide a helpful level of abstraction not only for implementing data-centric distributed algorithms, but also for reasoning effectively about them. This claim is seconded by our Coq formal model of Netlog which allows us to design and perform very accurate proofs. Our experience on BFS showed the robustness of a fully formalized case study with relation to two non-trivial modifications (in that case: not of the algorithm itself, which is very short, but of the underlying model): overall, less than 5% of the code had to be changed in order to recover the desired results.

Our experience on Prim's algorithm is less advanced, since the Coq formalization has only been carried out on the model. However, in this case, Netlog already turned out to be a suitable framework for designing correctness proofs. The two central characteristics of Netlog in this respect are its high level of expression and its data-centric features.

Our current Coq development contains about 7000 lines of Coq [17]: 1200 for the general model and common libraries, and 5800 for the case studies about BFS and Prim's algorithm. We estimate that completing the formalization in Coq of our proof of Prim's algorithm would require about two months. Future work will include the study of GHS and other kinds of data-centric protocols.

References

[1] Y. Deng, S. Grumbach, and J.-F. Monin, "A Framework for Verifying Data-Centric Protocols," in *FMOODS/FORTE 2011*, ser. LNCS, R. Bruni and J. Dingel, Eds., vol. 6722. Reykjavik, Iceland: Springer, June 6-9 2011, pp. 106–120.

[2] C. Kirkwood and M. Thomas, "Experiences with specification and verification in LOTOS: a report on two case studies," in *WIFT'95*. IEEE Computer Society, 1995, p. 159.

[3] F. Regensburger and A. Barnard, "Formal Verification of SDL Systems at the Siemens Mobile Phone Department," in *TACAS'98*. Springer, 1998, pp. 439–455.

[4] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis, "A toolbox for the verification of LOTOS programs," in *ICSE'92*. ACM, 1992, pp. 246–259.

[5] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 66–77, 1983.

[6] J. L. Welch, L. Lamport, and N. Lynch, "A lattice-structured proof of a minimum spanning," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, ser. PODC'88. New York, NY, USA: ACM, 1988, pp. 28–43. [Online]. Available: http://doi.acm.org/10.1145/62546.62552

[7] W. H. Hesselink, "The Verified Incremental Design of a Distributed Spanning Tree Algorithm: Extended Abstract," *Formal Asp. Comput.*, vol. 11, no. 1, pp. 45–55, 1999.

[8] Y. Moses and B. Shimony, "A New Proof of the GHS Minimum Spanning Tree Algorithm," in *DISC*, ser. Lecture Notes in Computer Science, S. Dolev, Ed., vol. 4167. Springer, 2006, pp. 120–135.

[9] J. S. Moore, "An acl2 proof of write invalidate cache coherence," in *CAV*, ser. Lecture Notes in Computer Science, A. J. Hu and M. Y. Vardi, Eds., vol. 1427. Springer, 1998, pp. 29–38.

[10] F. Verbeek and J. Schmaltz, "Formal specification of networks-on-chips: Deadlock and evacuation." in *International Conference on Design Automation and Test Europe (DATE'10)*. Dresden, Germany: IEEE, March 2010.

[11] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, "Declarative routing: extensible routing with declarative queries," in *ACM SIGCOMM'05*, 2005.

[12] 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 *ACM SIGMOD'06*, 2006.

[13] C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo, "A declarative perspective on adaptive manet routing," in *PRESTO'08*. ACM, 2008, pp. 63–68.

[14] S. Grumbach and F. Wang, "Netlog, a Rule-Based Language for Distributed Programming," in *PADL'10*, ser. LNCS, vol. 5937, 2010, pp. 88–103.

[15] V. Jarnìk, "O jistèm problému minimàlnìm [about a certain minimal problem]," *Práce Moravské Přìrodovědecké Společnosti*, vol. 6, pp. 57–63, 1930, (in Czech).

[16] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, pp. 1389–1401, 1957.

[17] M. Chen, Y. Deng, and J.-F. Monin, "Coq Script for Netlog Protocols," http://www-verimag.imag.fr/~monin/Proof/NetlogCoq/netlogcoq.tar.gz.