A Framework for Verifying Data-Centric Protocols

Yuxin Deng, Stéphane Grumbach, Jean-François Monin

A Framework for Verifying Data-Centric Protocols

Yuxin Deng[1,2⋆], Stéphane Grumbach[3], and Jean-François Monin[4]

[1] Department of Computer Science, Shanghai Jiao Tong University
[2] Laboratory of Computer Science, Chinese Academy of Sciences
[3] INRIA - LIAMA
[4] Université de Grenoble 1 - CNRS - LIAMA

Abstract. Data centric languages, such as recursive rule based languages, have been proposed to program distributed applications over networks. They simplify greatly the code, which is orders of magnitude shorter, much more declarative, while still admitting efficient distributed execution. We show that they also provide a promising approach to the verification of distributed protocols, thanks to their data centric orientation, which allows to explicitly handle global structures, such as the topology of the network, routing tables, trees, etc, as well as their properties.
We consider a framework using an original formalization in the Coq proof assistant of a distributed computation model based on message passing with either synchronous or asynchronous behavior. The declarative rules of the Netlog language for specifying distributed protocols, as well as the virtual machines for evaluating these rules, are encoded in Coq as well. We consider as a case study tree protocols, and show how this framework enables us to formally verify them in both the asynchronous and synchronous setting.

1 Introduction

Up to now, most efforts to formalize protocols or distributed algorithms and automate their verification relied on control-oriented paradigms. It is the case for instance of the "Formal Description Techniques" developed by telecom labs at the beginning of the 1980s in order to specify and verify protocols to be standardized at ITU and ISO. Two of the languages developed, 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 [35]. Several verification tools, ranging from simulation to model checking, have been developed and applied to different case studies [38, 18, 34, 39, 31, 33, 17, 9, 12].

For the verification of communication protocols based on process algebras, the idea has been to model both the implementation and the specification of a protocol as processes in a process algebra, and then to use automatic tools to

⋆ Supported by the Natural Science Foundation of China under grant No. 61033002.

check whether the former is a refinement of the latter or if they are behaviorally equivalent [5, 32]. Examples include the Concurrency Workbench [5], which is a verification tool based on CCS, FDR [32] which is based on CSP [16], ProVerif [2] which is based on the applied pi calculus [1]. Other approaches include input/output automata [26], or Unity and TLA, which combine temporal logic and transition-based specification [3, 19], and may rely as well on proof assistant technology [30, 15, 4, 20].

The common feature of all these approaches is their focus on control, in particular on 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. If this is relevant for many low-level protocols, such as transport protocols, it does not suit the needs of applications which aim at building up distributed global information, such as topological information on the network (in a physical or a virtual sense), e.g. routing tables. Such protocols are qualified as *data-centric* in the sequel. Correctness proofs of data-centric protocols are even more complex than those for *control-centric* protocols.

Data-centric rule-based languages have been recently proposed to program network protocols and distributed applications [24, 23, 22]. This approach benefits from reasonably efficient implementations, by using methods developed in the field of databases for recursive languages à la Datalog. Programs are expressed at a very high level, typically two orders of magnitude shorter than code written in usual imperative languages.

Our claim is that this framework is promising not only for the design of distributed algorithms, but for their verification as well. Up to now, only a few partial results have been demonstrated. In [36], a declarative network verifier (DNV) was presented. Specifications written in the Network Datalog query language are mapped 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 proposed in this paper. In other words, only the highly abstract centralized behaviour of a network is considered. Therefore, deep subtleties on message passing, derivation of local facts and their relationship with the intended global behaviour are absent in [36].

In the present paper, we go one essential step further. We show that it is indeed feasible to reason about the distributed behaviour of individual nodes which together yield some expected global behaviour of the whole network. We consider the Netlog language [13], which 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 fixpoint semantics which interleaves local computation on the nodes and communication between neighboring 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 be sent. During the communication phase, messages are transmitted and become available to the destination node.

Our objective is to develop a framework to formally verify properties of Netlog 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 does not 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.

We use the *proof assistant*, Coq, which is an interactive theorem prover, in which high level proof search commands construct formal proofs behind the scene, which are then mechanically verified. Coq has been successfully applied to ensure reliability of hardware and software systems in various fields, such as multiplier circuits [29], concurrent communication protocols [11], self-stabilizing population protocols [8], devices for broadband protocols [27], and compilers [21], to name a few.

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 (iv) a formalization of graphs and trees suitable to our needs.

As a proof of concept, we experimented the proposed framework on concrete protocols for constructing spanning trees over connected graphs. Such protocols have been shown to be correct on theoretical models. This is the case for instance of the well-known distributed algorithm for computing a minimum-weight spanning tree due to Gallager, Humblet and Spira [10]. The rigorous proofs made between 1987 and 2006 [37, 14, 28] are all intricate and very long (100 to 170 pages). Only [14] has been mechanically proof-checked.

Our objective is to carry on proofs for protocols written in a programming language (Netlog) which is implemented and runs on real distributed systems, and not only in theoretical models, and to concentrate on a data-centric approach. The protocols proceed in rounds, where one node (in the asynchronous model) or all nodes (in the synchronous model) perform some local computation, update their local data and then exchange data with their neighbors before entering the next round.

We have proven an initial simple protocol for defining spanning trees. Furthermore, in the synchronous message passing model, we show that we obtain a distributed version of the classical breadth-first search (BFS) algorithm. To show its correctness, the crucial ingredient is to formally prove the validity of the invariant that states the relationship between the centralized and the distributed

version of the protocol, as well as the propagation of information on the distributed version. This is non trivial and requires modeling how distributed tasks cooperate together to form the invariant. We claim that the proposed techniques establish foundations for proving more complex tree protocols such as GHS [10].

The paper is organized as follows. In Section 2, the distributed computation as formalized in Coq is presented. Section 3 is devoted to the presentation of the Netlog language. Section 4 contains the sketches of the proofs of the correctness of the tree protocol (a complete Coq script is available in [6]).

2 Distributed Computation Model of Netlog

In this section we introduce a distributed computation model based on the message passing mechanism, which is suitable both for synchronous and asynchronous execution. Corresponding formal definitions in Coq can be found in [6, 7]. The distributed computation model described below does not depend on Netlog. In our formalization, we just assume that the states at nodes have a type *local_data* which can evolve using simple set-theoretic operations such as union.

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 $1, 2, \cdots, n$, where n 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. The router queues the incoming messages on the reception queue and the message to push produced by the engine on the emission queue.

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.

The contents of node *loc* (the database of facts stored at *loc*) for a configuration *cnf* is denoted by $|loc|^{cnf}$, or just $|loc|$ when the configuration is clear from the context. Similarly, the set of messages arriving a node y from node x is denoted by $|x{\rightarrow}y|^{cnf}$ or $|x{\rightarrow}y|$.

A local round at node *loc* relates an actual configuration *pre* to a new configuration *mid* and a list *out* of messages emitted from *loc*. Furthermore, incoming edges are cleared – intuitively, this represents the consumption of messages, once their contents has been used to elaborate *mid* and *out*. 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*, that is, $|loc|^{pre}$ and all the $|x{\rightarrow}loc|^{pre}$ such that there is an edge from x to *loc*. Intuitively, the relation *new_stores* expresses that d consists of new facts derived from facts available at *loc*. Similarly, *out* is defined by a relation *new_push* and satisfies similar requirements. Formally, a local round is defined by the following conjunction.

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

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.

$$no\_change\_at(loc, pre, mid) \stackrel{\text{def}}{=\!=} \begin{cases} |loc|^{mid} = |loc|^{pre} \\ \forall\ x \in neighbors(loc), |x{\rightarrow}loc|^{mid} = |x{\rightarrow}loc|^{pre} \end{cases}$$

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.

$$communication(loc, mid, post, out) \stackrel{\text{def}}{=\!=} \begin{cases} |loc|^{post} = |loc|^{mid} \\ \forall\ y \in neighbors(loc), |loc{\rightarrow}y|^{post} = find(y, out) \ \cup \ |loc{\rightarrow}y|^{mid} \end{cases}$$

The function *find* returns the fact in *out* whose destination is y. 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.

$$async\_round(pre, post) \stackrel{\text{def}}{=\!=} \exists\, loc\ mid\ out \begin{cases} local\_round(loc, pre, mid, out) \\ \forall\ loc', loc \neq loc' \Rightarrow no\_change\_at(loc', pre, mid) \\ communication(loc, mid, post, out) \\ \forall\ loc', loc \neq loc' \Rightarrow communication(loc', mid, post, \emptyset) \end{cases}$$

An asynchronous round between two configurations *pre* and *post* is given by a node Id *loc*, an intermediate configuration *mid* and a list of 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*.

$$sync\_round(pre, post) \stackrel{\text{def}}{=} \exists\, mid,\ \forall\, loc,\ \exists\, out \begin{cases} local\_round(loc, pre, mid, out) \\ communication(loc, mid, post, out) \end{cases}$$

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 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 co-inductively 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*. 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 parametrized by a property P of configurations. In a more general setting, the parameter would be a property of runs. 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.

3 Data Centric Protocols

In this section, we introduce the Netlog language through examples of simple protocols for defining trees. Only the main constructs of the language are presented. A more thorough presentation can be found in [13]. 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.

We first recall classical Datalog, whose programs run in a centralized setting over relational structures, and which allow to define invariants that will be used as well in the proofs in the distributed setting. We assume that the language contains *negation* as well as *aggregation functions*, which can be used in the head of rules to aggregate over all values satisfying the body of the rule. For instance, the function *min* will be used in the next example.

Let us start with the program, **BFS-seq**, which computes BFS trees. It runs on an instance of a graph represented by a binary relation E, and a unique node satisfying $root(x)$. The derived relations $onST$, and ST, are such that $onST(\alpha)$

holds for a node α already on the tree, and $ST(\alpha, \beta)$ holds for an edge (α, β) already in the BFS tree.

BFS-seq in Datalog

$$onST(x) \leftarrow Root(x). \tag{1}$$

$$\left.\begin{matrix} ST(\min(x), y) \\ onST(y) \end{matrix}\right\} \leftarrow E(x, y); onST(x); \neg onST(y). \tag{2}$$

The evaluation of the program is iterated in a inflationary manner, by accumulating the results till a fixpoint, which defines its semantics, is reached. At the first step, the root node is included in the relation $onST$ using the first rule. At the n^{th} step, nodes at distance $n-1$ from the root are added in $onST$, and an arc of the tree is added in ST for each of them, by choosing the parent with minimal Id. The fixpoint is reached when all nodes are on the tree.

Minimum spanning trees can be defined in Datalog with arithmetic. Let us consider first the definition corresponding to Prim's algorithm [25]. We assume weighted graphs, $G = (V, E, \omega)$, where the weight $\omega : E \rightarrow \mathbb{R}^+$, satisfies $\omega(u, v) = \omega(v, u)$ for every edge $(u, v) \in E$. As usual, to simplify the algorithm, we assume that ω is a 1-1 mapping, so the weights of any pair of edges are distinct. Prim's algorithm starts from a (root) node, and construct successive fragments of the MST, by adding the minimal outgoing edge to the fragment at each step.

The sequential Datalog program can be written with three rules as follows. The symbol "!" denotes the consumption of the fact used in the body of the rule, which is deleted after the application of the rule.

MST-Prim-seq in Datalog

$$\left.\begin{matrix} onST(x) \\ MWOE(min(m)) \end{matrix}\right\} \leftarrow Root(x); E(x, y, m). \tag{3}$$

$$\left.\begin{matrix} ST(x, y) \\ onST(y) \end{matrix}\right\} \leftarrow onST(x); \neg onST(y); E(x, y, m); !MWOE(m). \tag{4}$$

$$MWOE(min(m)) \leftarrow onST(x); \neg onST(y); E(x, y, m); \neg MWOE(m). \tag{5}$$

The evaluation of this program alternates two phases, (i) computation of the minimal outgoing edge's weight, MWOE, and when it is obtained, (ii) addition of the corresponding unique edge.

Let us consider now, Netlog programs, which are installed on each node, where they run concurrently. 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. In contrast with other approaches to concurrency, the focus is not primarily on monitoring events, but data (i.e. Datalog facts) contained in nodes.

The facts deduced from rules can be stored on the node, on which the rules run, or sent to other nodes. The symbol in the head of the rules means that the

result has to be either stored on the local data store (↓), sent to neighbor nodes (↑), or both (↕). The facts received on a node are used to trigger the rules, but do not get stored on that node.

The evaluation of the body of a rule is always performed on a given node. A fact is then considered to hold if and only if it occurs on this node. The *negation* of a fact holds if the fact does not occur on the node where the computation is performed.

The following program, which constructs a spanning tree over a distributed system, relies as above 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.

Spanning Tree Protocol in Netlog

$$\updownarrow onST(x) \quad \leftarrow @x = 0. \tag{6}$$

$$\left.\begin{array}{r}\updownarrow onST(y)\\ \downarrow ST(min(x), y)\end{array}\right\} \leftarrow E(x, @y); onST(x); \neg onST(y). \tag{7}$$

Rule (6) runs on the unique (rrot) 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 (7) 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 (6) and (7), the consistency of E with the physical edge relation holds forever. This algorithm aims at constructing suitable distributed relations $onST$ and ST. In Section 4, we will prove that they actually define a tree; moreover, in the synchronous setting they define a BFS tree.

The translation of the sequential Datalog program defining a spanning tree, to a distributed program is almost trivial. It suffices to add communication instructions since the program runs locally. The translation to a distributed program is more complex for the **Minimal Spanning tree protocol**. Indeed, rules (4) and (5) are not local, and require communication between remote nodes. In a network, the root can orchestrate the distributed computation, by alternating phases of (i) computation of the MWOE by a convergecast into the current spanning tree, and (ii) addition of the new edge with minimal weight to the tree.

The next program (together with two simple rule modules for the convergecast and the edge addition) defines the minimal spanning tree in Netlog. The facts $AddEdge(x, m)$ and $GetMWOE(x)$ are triggering rule modules (of half a dozen rules) that perform respectively a traversal of the tree to add the edge with minimal outgoing weight, and a convergecast to obtain the new minimal outgoing weight. These rule modules (omitted for space reason) can be used in other protocols requiring non local actions. We have tested them in particular in a concurrent version of the Minimal Spanning Tree, the GHS, where there is no initial root and all nodes concurrently start building MST fragments.

$$\left.\begin{array}{c} onST(x) \\ UpMWOE(x, min(m)) \\ GetMWOE(x) \end{array}\right\} \leftarrow Root(x); E(x, y, m). \quad (8)$$

$$\downarrow AddEdge(x, m) \leftarrow Root(x); !GetMWOE(x); !UpMWOE(x, m). \quad (9)$$

$$\downarrow GetMWOE(x) \leftarrow Root(x); !AddEdge(x, m); !UpEdge(x, m). \quad (10)$$

4 Verifying Tree Protocols

We conduct the verification in two settings. In the asynchronous case, we prove that the previous protocol for spanning tree eventually constructs a spanning tree, while in the synchronous case, we prove that this protocol constructs actually a spanning tree by doing a breadth-first search in the network. We briefly sketch the first case study and then give a more detailed discussion for the second one which involves a much more difficult proof.

In both cases we expect to show that the relation ST determines a spanning tree. However, this relation is distributed on the nodes and the Netlog protocol reacts only to a locally visible part of relations ST, $onST$ and E. The expected property is then stated in terms of the *union* of all ST facts available on the network.

4.1 Spanning Tree in the Asynchronous Case

We have to check that when adding a new fact $ST(x, y)$ at some node loc then x is already on the tree while y is not yet. This is basically entailed by the body of the last rule, but additional properties are needed in order to ensure this rigorously. We use the following ones:

1. The E relation corresponds exactly to the edges.
2. An $onST(z)$ fact arriving at a node y is already stored on the sender x.
3. If an $onST(x)$ fact is stored on a node loc, then $x = loc$.
4. The $onST$ relation grows consistently with ST ($onST$ is actually the engine of the algorithm), and these two relations define a tree.

The first three properties are separately proved to be invariant. The last property is included in a predicate `is_tree`(o, s), which intuitively means that the union of all $onST$ facts o and the union of all ST facts s are consistent and they define a tree. We prove that if at the beginning of a round the first three properties together with `is_tree`(o, s) hold, then at the end of the round `is_tree`(o, s) still holds. The conjunction of all the four properties then constitutes an invariant of the protocol.

We check that the initial configuration generates a tree, then we have that in all configurations of any asynchronous run starting from the initial configuration,

ST has the shape of a tree. This safety property is formalized in Coq (the script is available online [6]).

Liveness, i.e. each node is eventually a member of $onST$, can be easily proved, provided the graph is finite and connected, and a *fairness* property is assumed in order to discard uninteresting runs where an inactive node is continuously chosen for each local round, instead of another node having an enabled rule. The proof is by induction on the finite cardinality of the set $\overline{onST}$ of nodes which do not satisfy $onST$. If at some point of a run this set is non-empty, then at least one of its members is a neighbor of the current tree due to connectivity. By fairness, this node eventually performs a local round and is no longer in $\overline{onST}$. Formalizing such arguments involving liveness and fairness properties of infinite behaviors of distributed systems has already been done in Coq [8]. The issue of termination is simpler in the synchronous setting, since fairness is no more needed to remove fake stuttering steps.

4.2 BFS in the Synchronous Case

For our second case study, the correctness proof of the BFS protocol, we prove that in the synchronous setting, the union of ST facts is the same as the one which would be computed by a centralized algorithm $\mathcal{O}$ (the oracle) running rules (1) and (2) on a reference version of the global relations $onST$ and ST. This is subtler than one may expect at first sight, because decisions taken on a given node do not depend on the global relations $onST$ and ST, but only on the visible part, which is made of the locally stored facts and of the arriving messages. Moreover, the information contained in an arriving $onST(x)$ fact is ephemeral: this fact is not itself stored locally (only its consequences $onST(y)$ and $ST(m, y)$ are stored) and it will never be sent again. Indeed this information is available exactly at the right time. We therefore make a precise reasoning on the consistency of stored and transmitted facts with the computation that would be performed by the oracle $\mathcal{O}$.

We denote by $\mathcal{C}$ the database of facts managed by $\mathcal{O}$. Our main theorem states that a synchronous round in the distributed synchronous version corresponds to a step of computation performed by $\mathcal{O}$ on $\mathcal{C}$. The proof relies necessarily on a suitable characterization of the body of rule (7), which depends on the presence and the absence of facts $onST$. Therefore we need first to prove that facts $onST$, as computed by distributed rules (6) and (7), are the ones computed by $\mathcal{O}$ and conversely – this is respectively called correctness and completeness of $onST$ (definitions 3 and 6).

The first direction is not very hard (proposition 5). Completeness requires more attention. The issue is to ensure that, given an edge from x to y, such that $onST(x) \in \mathcal{C}$ but $onST(y) \notin \mathcal{C}$, the body of rule (7) holds at y in order to ensure that rule (7) will derive $onST(y)$ as expected by rule (2) at the next step. If we just assume correctness and completeness of $onST$, we get $onST(x)$ only on x, while we need it on y. Therefore a stronger invariant is needed. The key is the introduction of the notion of a *good* edge (definition 7) which says that 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). Here are the main steps. Additional properties, such as the establishment of the invariant in the initial configuration (actually: after one synchronous round) are available in [7, 6].

Notation Let φ be a fact; here φ can have the shape $E(x, y)$ or $onST(x)$ or $ST(x, y)$. The presence of a fact φ in a database d is denoted by $\varphi \in d$. The set of facts $ST(x, y)$ in d is denoted by d_{ST}, and use a similar convention for $onST$ end E. The database of facts stored at node loc is denoted by $|loc|$. Similarly, the database of facts arriving a node y from node x is denoted by $|x{\rightarrow}y|$. Statements such as $onST(z) \in |loc|$ are about a given configuration cnf or even an extended configuration $\langle cnf, \mathcal{C}\rangle$, and should be written $cnf, \mathcal{C} \Vdash onST(z) \in |loc|$. In general cnf is clear from the context and we just write P instead of $cnf, \mathcal{C} \Vdash P$. When we consider a synchronous round, i.e., a transition between two consecutive configurations pre and $post$, we write $P \xrightarrow{sr} Q$ for $pre \Vdash P \;\Rightarrow\; post \Vdash Q$. Similarly, for oracle transitions and transitions between extended configurations, we write respectively $P \xrightarrow{o} Q$ for $\mathcal{C} \Vdash P \;\Rightarrow\; \mathcal{C}' \Vdash Q$ and $P \xrightarrow{sro} Q$ for $pre, \mathcal{C} \Vdash P \;\Rightarrow\; post, \mathcal{C}' \Vdash Q$.

Definition 1 *A configuration satisfies* received-onST-already-stored *if and only if for all edges $x{\rightarrow}y$, if $onST(z) \in |x{\rightarrow}y|$, then $z = x$ and $onST(z) \in |x|$.*

Proposition 2 *After a transition, a configuration always satisfies* received-onST-already-stored.

Proof. By inspection of store and push rules (6) and (7).

Correctness of *onST*

Definition 3 *An extended configuration $\langle cnf, \mathcal{C}\rangle$ satisfies* correct-onST *if and only if for all location loc of cnf, if some fact $onST(z)$ is visible at loc, then $onST(z) \in \mathcal{C}$.*

Proposition 4 $onST(0) \in \mathcal{C} \xrightarrow{o} onST(0) \in \mathcal{C}$.

Proof. By inspection of oracle rules (1) and (2).

Proposition 5 $onST(0) \in \mathcal{C}$, *correct-onST* $\xrightarrow{sro}$ *correct-onST.*

Proof. By inspection of the consequences of rule (7) and using proposition 2.

Completeness of *onST* The notion of completeness needed is much more precise than the converse of correct-onST: the location where $onST(z)$ is stored has to be known. This is especially clear in the proof of lemma 10.

Definition 6 *An extended configuration $\langle cnf, \mathcal{C}\rangle$ satisfies* complete-onST-node *if and only if for all x, if $onST(x) \in \mathcal{C}$, then $onST(x)$ is stored at x.*

Definition 7 *An edge $x \to y$ is* good *in a given configuration if and only if, if $onST(x) \in |x|$, then $onST(y) \in |y|$ or $onST(x) \in |x \to y|$. A configuration satisfies* all-good *if and only if all its edges are good.*

The following proposition is about non-extended configurations, i.e. it is purely about the distributed aspect of the BFS algorithm.

Proposition 8 *received-onST-already-stored, all-good* $\xrightarrow{sr}$ *all-good*

The main use of goodness is the completeness of the evaluation of the body of rule (7).

Definition 9 *We say that an extended configuration $\langle cnf, \mathcal{C} \rangle$ is* ready *if and only if (i) it satisfies correct-onST, complete-onST-node and (ii) cnf satisfies all-good.*

Lemma 10 *Given an extended configuration satisfying ready, and an edge $x \to y$ such that $onST(x) \in \mathcal{C}$ but $onST(y) \notin \mathcal{C}$, the body of rule (7) holds at y.*

The propagation of the completeness of *onST* follows.

Proposition 11 *ready* $\xrightarrow{sro}$ *complete-onST-node.*

Correctness and completeness of *ST*

Definition 12 *Let cnf be a given configuration. We say that $\langle cnf, \mathcal{C} \rangle$ satisfies* same-ST *if and only if the union of all ST facts contained in some node of cnf is the same as set of facts ST in $\mathcal{C}$.*

Proposition 13 *ready, same-ST* $\xrightarrow{sro}$ *same-ST.*

Main theorem Our invariant is the following conjunction.

Definition 14 *An extended configuration $\langle cnf, \mathcal{C} \rangle$ satisfies* invar *if and only if it satisfies $onST(0) \in \mathcal{C}$, received-onST-already-stored, ready and same-ST.*

Theorem 15 *invar* $\xrightarrow{sro}$ *invar.*

Proof. Immediate use of propositions 2, 4, 5, 11 and 13.

Finally, we observe that invar is established after one synchronous round from the initial configuration, and that same-ST holds in the initial configuration. As a consequence, same-ST holds forever, as expected.

Besides this global property, one may 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 (6) and (7), since they ensure that $ST(x, y)$ can only be stored on y. This is formally proved in our framework.

5 Conclusion

We developed a framework for verifying data-centric protocols expressed in a rule-based language. We have shown that both the synchronous and the asynchronous models of communication can be formalized in very similar ways from common building blocks, that can be easily adapted to other communication models.

Our framework has been implemented as a Coq library, which includes the formalization of the distributed computation environment with the communication network, as well as the embedded machine which evaluates the Netlog programs on each node. The Netlog programs are translated into straightforward Coq definitions. The proofs, sketched in the paper have been fully formalized in Coq [7]. As a preliminary result we proved a topological property of a distributed data structure – a tree – constructed by a simple but subtle program.

Figures on the size of our current Coq development are given in Table 1. The detail of justifications such as "by inspection of rules (6) and (7)" requires in general many bureaucratic proofs steps. From previous experience with Coq, we know that most of them can be automated using dedicated tactics, so that the user can focus entirely on the interesting part of the proof. The representation of Netlog rules was obtained in a systematical way and could be automated as well, using a deep embedding.

The distributed algorithm considered here as a case study was not as trivial as it may appear at first sight, though it can be expressed in a few lines of Netlog. It was sufficient to cover essential issues of a data-centric distributed algorithm, in particular the relationship between local transformations and global properties. Such properties are difficult to handle and even to state in event-centric approaches to the verification of distributed programs.

The advantage of the techniques we have developed is that they constitute a natural and promising open framework to handle other distributed data-centric algorithms. We are currently working on proofs for minimum spanning trees, and plan to further verify protocols for routing, election, naming, and other fundamental distributed problems.

Distributed computation model	180
Netlog	1300
Tree definitions and properties	80
Translation of rules	50
Proofs on centralized ST algorithm (Rules (1) and (2))	360
Proofs on (asynchronous) ST (Rules (6) and (7))	1100
Proofs on (synchronous) BFS (Rules (6) and (7))	1300

Table 1. Size of coq scripts (in number of lines)

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. POPL'01*, volume 36, pages 104–115. ACM, 2001.
2. B. Blanchet. Automatic Verification of Correspondences for Security Protocols. *Journal of Computer Security*, 17(4):363–434, 2009.
3. K. M. Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., 1988.
4. B. Chetali. Formal Verification of Concurrent Programs Using the Larch Prover. *IEEE Transactions on Software Engineering*, 24:46–62, 1998.
5. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrency systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.
6. Y. Deng, S. Grumbach, and J.-F. Monin. Coq script for Netlog protocols. `http://www-verimag.imag.fr/~monin/Proof/NetlogCoq/netlogcoq.tar.gz`.
7. Y. Deng, S. Grumbach, and J.-F. Monin. Verifying Declarative Netlog Protocols with Coq: a First Experiment. Research Report 7511, INRIA, 2011.
8. Y. Deng and J.-F. Monin. Verifying Self-stabilizing Population Protocols with Coq. In *Proc. TASE'09*, pages 201–208. IEEE Computer Society, 2009.
9. J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proc. ICSE '92*, pages 246–259. ACM, 1992.
10. R. G. Gallager, P. A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.
11. E. Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems*. PhD thesis, ENS Lyon, 1996.
12. R. Gotzhein and J. Bredereke, editors. *Formal Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1*, volume 69 of *IFIP Conference Proceedings*. Chapman & Hall, 1996.
13. 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.
14. W. H. Hesselink. The Verified Incremental Design of a Distributed Spanning Tree Algorithm: Extended Abstract. *Formal Asp. Comput.*, 11(1):45–55, 1999.
15. B. Heyd and P. Crégut. A Modular Coding of UNITY in COQ. In *Proc. TPHOLs'96*, pages 251–266. Springer, 1996.
16. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
17. C. Jard, J. F. Monin, and R. Groz. Development of Veda, a Prototyping Tool for Distributed Algorithms. *IEEE Trans. Softw. Eng.*, 14(3):339–352, 1988.
18. C. Kirkwood and M. Thomas. Experiences with specification and verification in LOTOS: a report on two case studies. In *Proc. WIFT '95*, page 159. IEEE Computer Society, 1995.
19. L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
20. T. Långbacka. A HOL Formalisation of the Temporal Logic of Actions. In *Proc. TPHOL'94*, pages 332–345. Springer, 1994.
21. 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.
22. 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.

23. 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.
24. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM '05*, 2005.
25. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.
26. N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
27. 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.
28. Y. Moses and B. Shimony. A New Proof of the GHS Minimum Spanning Tree Algorithm. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2006.
29. 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.
30. L. C. Paulson. Mechanizing UNITY in Isabelle. *ACM Trans. Comput. Logic*, 1(1):3–32, 2000.
31. F. Regensburger and A. Barnard. Formal Verification of SDL Systems at the Siemens Mobile Phone Department. In *Proc. TACAS '98*, pages 439–455. Springer, 1998.
32. A. W. Roscoe. *Model-checking CSP*, chapter 21. Prentice-Hall, 1994.
33. S. M. Shahrier and R. M. Jenevein. SDL Specification and Verification of a Distributed Access Generic opticalNetwork Interface for SMDS Networks. Technical report, University of Texas at Austin, 1997.
34. M. Törö, J. Zhu, and V. C. M. Leung. SDL specification and verification of universal personal computing: with Object GEODE. In *Proc. FORTE XI / PSTV XVIII '98*, pages 267–282. Kluwer, B.V., 1998.
35. K. J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL*. John Wiley & Sons, Inc., 1993.
36. A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative Network Verification. In *Proc. PADL '09*, pages 61–75. Springer, 2009.
37. 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*, PODC '88, pages 28–43, New York, NY, USA, 1988. ACM.
38. J.-P. Wu and S. T. Chanson. Translation from LOTOS and Estelle Specifications to Extended Transition System and its Verification. In *Proc. FORTE '89*, pages 533–549. North-Holland Publishing Co., 1990.
39. W. Zhang. Applying SDL Specifications and Tools to the Verification of Procedures. In *Proc. SDL '01*, pages 421–438. Springer, 2001.