TLP **13** (4–5): Online Supplement, July 2013. © 2013 [JIEFEI MA, FRANCK LE, DAVID WOOD, ALESSANDRA RUSSO, and JORGE LOBO]
URL: http://dx.doi.org/10.1017/S1471068413000513

A declarative approach to distributed computing: Specification, execution and analysis

JIEFEI MA
Imperial College London
(*e-mail:* {`j.ma@imperial.ac.uk`})

FRANCK LE
IBM Waston Laboratory, US
(*e-mail:* {`fle@us.ibm.com`})

DAVID WOOD
IBM Waston Laboratory, US
(*e-mail:* {`dawood@us.ibm.com`})

ALESSANDRA RUSSO
Imperial College London
(*e-mail:* {`a.russo@imperial.ac.uk`})

JORGE LOBO
ICREA - Universitat Pompeu Fabra
(*e-mail:* {`jorge.lobo@upf.edu`})

submitted 10 April 2013; revised 23 June 2013; accepted 5 July 2013

Abstract

There is an increasing interest in using logic programming to specify and implement distributed algorithms, including a variety of network applications. These are applications where data and computation are distributed among several devices and where, in principle, all the devices can exchange data and share the computational results of the group. In this paper we propose a declarative approach to distributed computing whereby distributed algorithms and communication models can be (i) specified as action theories of fluents and actions; (ii) executed as collections of distributed state machines, where devices are abstracted as (input/output) automata that can exchange messages; and (iii) analysed using existing results on connecting causal theories and Answer Set Programming. Results on the application of our approach to different classes of network protocols are also presented.

KEYWORDS: action theory, answer set programming, network protocols, distributed computing

1 Introduction

In the context of distributed applications, specifying and reasoning about distributed algorithms are hard and open problems. This is mainly because the behaviour of

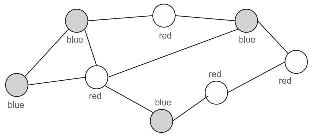

Fig. 1. Opinion network.

these applications is not simply expressed by the local computation of a program but by the *global behaviour that emerges* from the interactions of all computational nodes, which is very difficult for a programmer to visualise. Minor modifications can drastically change how the application behaves. As an illustration of these challenges consider a simple voting algorithm over a network of nodes, depicted in Figure 1. The algorithm runs as follows: each node has an initial opinion of either *good* (white color) or *bad* (grey colour), and (1) sends it to all its neighbours; (2) upon receiving a neighbour's opinion, the node decides its new opinion based on the majority opinions received from all of its neighbours so far; and (3) if the node changes its opinion, it will inform the neighbours. This program is assumed to run on every node in the network. Examples of global properties include checking whether the algorithm always converges (i.e., reaching a global state where nodes stop changing their opinion); checking if a specific global state is eventually reached (e.g., all nodes will eventually become good); or which initial state of the network will cause the algorithm to never converge. Although the algorithm that runs in each node has three simple steps, answering these questions is not obvious. Our challenge is to find appropriate computational abstractions that allow distributed algorithms to be described simply, and also amenable to efficient analysis.

A very common abstraction used to describe distributed algorithms is to view each computational node as an input/output automaton. We can find many useful/practical algorithms that can be defined using input/output automata where the transition functions are limited to polynomial computations with respect to the size of the input and the state, from routing protocols, to leader elections to commit decisions for database transactions (Lynch 1996).

A substantial body of work and results on the use of declarative languages for describing and reasoning about the effects of actions have been presented in the literature. These have been shown to be sound and complete with respect to a semantics based on automaton (e.g. (Gelfond and Lifschitz 1998)). Translations of these languages to (extended) logic programs or causal logics have also provided means to use theorem provers to tackle a variety of application problems including planning and fault diagnosis.

Building upon this body of work, we propose a declarative approach to distributed computing whereby distributed algorithms can not only be specified as action theories of fluents and actions, but also executed as collections of (input/output) automata which can exchange messages, and analysed using existing results on connecting causal theories and Answer Set Programming (ASP). By looking at a distributed computing problem from the point of view of causal theories, we can express real world network applications, such as the current de facto inter-domain routing protocol BGP, essentially as action theories of fluents and actions, and perform analysis tasks known to be NP-complete. This is done by lifting constructions from the action language $\mathscr{C}+$ to specify distributed algorithms and then use the same language to formalize network communication models. This result demonstrates that distributed systems are fundamentally action theories. This is in contrast to our earlier language (Lobo *et al.* 2012) where the communication model is expressed as a logic program. The solution proposed here is much more elegant and still takes advantage of the connection of causal theories and ASP. This is reflected in the fact that the specification of distributed algorithms is more concise, but most importantly the implementation of the automaton evaluation engine is much simpler.

By using existing results, we are able to translate the distributed algorithm and the communication model from the causal logic-based specification into logic programs, providing *a formal foundation for the implementation of distributed applications and analysis tools*. We briefly describe a system, called Distributed State Machine (DSM), for executing distributed algorithms written in our language on real network topologies, and our analysis approach, based on ASP, for checking the correctness of these algorithms with respect to given properties. We illustrate the application of our approach two classes of real-world network protocols and present results on some key verification tasks.

The paper is structured as follows. Section 2 presents the language $\mathscr{D}$ for the specification of distributed system components in terms of input/output automata. Section 3 shows how input/output automata can be composed into complex network applications and describes how $\mathscr{D}$ itself can be used to model the communication between the different distributed components. Section 4 introduces our DSM system for executing distributed applications written in $\mathscr{D}$ on real network topologies. Section 5 describes the concept of traces, how traces can be captured using logic programs and how these logic programs can be used to analyse network applications. Related work and concluding remarks can be found in Section 6.

2 The language $\mathscr{D}$

Distributed system components are typically modelled using input/output (I/O) automata (Lynch 1996). An I/O automaton adds to a transition diagram extra labels representing the output of the transitions. Our approach is to build upon the language $\mathscr{C}+$ (Giunchiglia *et al.* 2004) a dialect, called $\mathscr{D}$, for describing distributed systems as compositions of I/O automata. The underlying signature of $\mathscr{D}$ is defined by two pairwise disjoint non-empty sets $(\mathbf{F}, \mathbf{A})$ of *fluent* names and *action* names. For I/O automata, in addition, the set of action names $\mathbf{A}$ is partitioned into a set of *input actions* $\mathbf{A}_I$ and *communication actions* $\mathbf{A}_C$. A *fluent literal* is a fluent name or its negation. Similarly, an *action literal* is an action named or its negation. A *state* will be any subset of fluent names. A fluent f is true in a state s if and only if $f \in s$.

As a simple illustration, assume we are managing virtual machines in a cloud system. The state of the administration node may have fluent names such as `vmachine(vm1,ser1)` and `vmachine(vm2,ser3)`, or `vrouter(rt11,ser2)` and vrouter(rt22, ser2) representing the physical server where virtual machines and routers are located. Input actions can indicate the creation or removal of virtual machines such as in new_vm(vm5, ser5), remove(vm2). A communication can be interpreted as a message that has been sent between automata. The communication action where_is(vm1)@ ser5 $\triangleright$ ser3 can be read as "server 5 sends a request to server 3 to know the location of virtual machine *vm*1". We use three types of propositions to define transition diagrams. *Static laws* of the form

$$\textbf{caused}\ \mathtt{F}\ \textbf{if}\ \mathtt{G1,\ldots,Gn} \tag{1}$$

dynamic laws of the form

$$\textbf{caused}\ \mathtt{F}\ \textbf{if}\ \mathtt{G1,\ldots,Gn}\ \textbf{after}\ \mathtt{H1,\ldots,Hm} \tag{2}$$

and *communication laws* of the form

$$\textbf{sent}\ \mathtt{A}\ \textbf{if}\ \mathtt{G1,\ldots,Gn}\ \textbf{after}\ \mathtt{H1,\ldots,Hm} \tag{3}$$

Here F and the G's are fluent literals. The H's are either fluent or action literals. A is a communication action and is called the *output communication action* of the proposition. Static and dynamic laws are similar to that in $\mathscr{C}+$ but restricted to literals. Like in $\mathscr{C}+$ as well, we write

$$\textbf{caused}\ \mathtt{F}\ \textbf{after}\ \mathtt{H}$$

when $n = 1$ and $G1$ is the constant true. As in $\mathscr{C}+$ too, one can choose in $\mathscr{D}$ the fluent literals that follow the law of inertia. Inertia is defined by dynamic laws of the form

$$\textbf{caused}\ \mathtt{F}\ \textbf{if}\ \mathtt{F}\ \textbf{after}\ \mathtt{F}$$

and they will be abbreviated by

$$\textbf{inertial}\ \mathtt{F}$$

An I/O automaton D is defined as a collection of static, dynamic and communication laws. An example of dynamic law is the proposition schema

$$\textbf{caused}\ \mathtt{vmachine(V,L)}\ \textbf{after}\ \mathtt{new\_machine(V,L)}$$

Here V and L are meta-variables that vary over all possible machine names and possible locations. Informally, this proposition schema can be read as: there is a cause for a virtual machine V to be in location L after the input action new_machine(V, L) is executed. We would like the existence or non-existence of virtual machines to persist by inertia, so we add the following propositions

$$\textbf{inertial}\ \mathtt{vmachine(V,L)}$$
$$\textbf{inertial}\ \neg\mathtt{vmachine(V,L)}$$

Inertia causes that when a new machine is created, if the machine already exists it can be located in two different places. To avoid this situation we can add the static law

$$\textbf{caused}\ \perp\ \textbf{if}\ \mathtt{vmachine(V,L1),vmachine(V,L2),L1} \neq \mathtt{L2}$$

Recall again that V, L1 and L2 are meta-variables for all possible machine names and locations, but in this case L1 must be different to L2. The following communication law

$$\textbf{sent}\ \mathtt{location(V,L)@\ ser3} \triangleright \mathtt{ser5}\ \textbf{after}\ \mathtt{vmachine(V,L),where\_is(V)@\ ser5} \triangleright \mathtt{ser3}$$

can be read as server 3 will send the location of virtual machine `V` to server 5 after server 5 asked server 3 where `V` is and server 3 knows that the location is `L`.

2.1 Semantics of $\mathscr{D}$

Since our language $\mathscr{D}$ is essentially a subset of $\mathscr{C}+$, we can use a similar translation to a logic program as described in Giunchiglia *et al.* (2004). There is only the minor distinction that in our language, our communication laws are short-hands of a combination of fluent and dynamic laws of $\mathscr{C}+$, as dynamic laws in $\mathscr{C}+$ do not allow an "after" component whereas our communication laws do. For completeness we present the full translation into logic programs. The logic program will have, in addition to elements of **F** and **A**, a copy f' and $a_c^{'}$ of each fluent f in **F** and each communication action a_c in **A**. Each static law of the form (1) in an automaton D is translated into the logic programming rules

$$\mathtt{F} \leftarrow \mathtt{not}\ \overline{\mathtt{G1}}, \ldots, \mathtt{not}\ \overline{\mathtt{Gn}}$$

$$\mathtt{F}' \leftarrow \mathtt{not}\ \overline{\mathtt{G1}'}, \ldots, \mathtt{not}\ \overline{\mathtt{Gn}'}$$

These rules say that the static laws apply to every state of the automaton. Each dynamic law of the form (2) in D is translated into the rule

$$\mathtt{F}' \leftarrow \mathtt{not}\ \overline{\mathtt{G1}'}, \ldots, \mathtt{not}\ \overline{\mathtt{Gn}'}, \mathtt{H1}, \ldots, \mathtt{Hm}$$

Similar translation applies to the communication laws (3) in D

$$\mathtt{A}' \leftarrow \mathtt{not}\ \overline{\mathtt{G1}'}, \ldots, \mathtt{not}\ \overline{\mathtt{Gn}'}, \mathtt{H1}, \ldots, \mathtt{Hm}$$

Note that the notation $\overline{\mathtt{L}}$ in the body of the rules above represents the complementary literal of `L`, and `not` corresponds to negation as failure. The translation is completed by adding the following set of rules for each action name `A`, and each fluent name `F`

$$\neg\mathtt{A} \leftarrow \mathtt{not}\ \mathtt{A}$$

$$\neg\mathtt{F} \leftarrow \mathtt{not}\ \mathtt{F}$$

$$\leftarrow \mathtt{not}\ \mathtt{F}', \mathtt{not}\ \neg\mathtt{F}' \tag{4}$$

For any set of predicates P, let $P' = \{p' | p \in P\}$. Let Π_D be the logic program obtained from an I/O automaton D. Given two states s_1 and s_2, there is an arc from s_1 to s_2 in the diagram of D if and only if there is a subset E of the actions **A** and answer set S of $\Pi_D \cup s_1 \cup E$ such that $S \cap F' = s_2'$. The labels in the arc are the set E as input and the set O of output actions that result from removing the $'$ from all the actions in $S \cap \mathbf{A}_C'$. This diagram defines the transition relation $trans(D) \subseteq \mathbf{F} \times \mathbf{A} \times \mathbf{F} \times \mathbf{A}_c$.

3 Modeling distributed processing with $\mathscr{D}$

In a distributed application, each I/O automaton represents a component of the application which is defined by the composition of its components. Following (Lynch 1996), a composition is realized by identifying actions in different automata with the same name as the same action and compositions are allowed if the automata participating in the composition are *compatible*. A collection of automata is *compatible* if and only if:

- All the sets of output actions are pair-wise disjoint.

This condition means that although a communication action can appear in any automaton, it can only be the output of a single automaton. In this paper we will assume that the number of automata in any composition is finite and all the compositions are made with compatible sets of automata. Given I/O automata $D_1, \dots, D_n$, the composition $\Delta(D_1, \dots, D_n)$ of these automata is an I/O automaton defined as follows

1. $(s_1, \dots, s_n)$ is a state of $\Delta(D_1, \dots, D_n)$ iff s_i is a state of D_i, for $1 \leqslant i \leqslant n$ and
2. $((s_1, \dots, s_n), E, (r_1, \dots, r_n), O) \in trans(\Delta(D_1, \dots, D_n))$ iff
   $(s_i, E_i, r_i, O_i) \in trans(D_i)$, for $1 \leqslant i \leqslant n$, and $E = \bigcup E_i, O = \bigcup O_i$.

The interesting definition for composed automata is not the diagram that the composition defines but the interactions between its components. This interaction is captured by the concept of traces. An *elementary trace* of a composed automaton $\Delta(D_1, \dots, D_n)$, is a (possibly infinite) sequence $s_0, O_0, s_1, O_1, s_2, O_2, \dots$ such that $(s_{i-1}, O_{i-1}, s_i, O_i) \in trans(\Delta(D_1, \dots, D_n))$ for every $i \geqslant 1$ in the sequence. Note that, except for O_0, all the O_i's are sets of output actions that are used to move to the next state in the trace. This captures a possible computation of the system where input actions are only executed at the beginning of the computation. Non-elementary traces are defined similarly, except that input actions are allowed at any step. That is, a *non-elementary trace* of a composed automaton Δ, is a (possibly infinite) sequence $s_0, O_0, s_1, O_1, s_2, O_2, \dots$ such that $(s_{i-1}, O_{i-1}, s_i, O_i \cap \mathbf{A}_c) \in trans(\Delta)$ for every $i \geqslant 1$ in the sequence.

Let us take the voting algorithm as an illustration. The specification of the I/O automaton D_i that will run in the node i is given below:

inertial `neighbour(X)`, $\neg$`neighbour(X)`. (5)

caused `neighbour(X)` **after** `add_neighbour(X, i)`. (6)

caused $\neg$`neighbour(X)` **after** `del_neighbour(X, i)`. (7)

caused `neighbour_opinion(X, V)` **after** `receive_vote(V, X, i)`. (8)

caused `neighbour_opinion(X, OldV)` **after** (9)
`neighbour_opinion(X, OldV)`, $\neg$`receive_vote(AnyV, X, i)`.

caused `my_opinion(good)` **if** `num_goods(N), num_bads(M)`, $N \geqslant M$. (10)

caused `my_opinion(bad)` **if** `num_goods(N), num_bads(M)`, $M > N$. (11)

caused `num_good(#count<X>)` **if** `neighbour_opinion(X, good)`. (12)

caused `num_bad(#count<X>)` **if** `neighbour_opinion(X, bad)`. (13)

sent `send_vote(V, i, X)` **if** `neighbour(X)`, (14)
`my_opinion(V)` **after** `my_opinion(OldV)`, $V \neq OldV$.

In this example specification, there are two types of input actions: `add_neighbour (X, i)` and `del_neighbour(X, i)`, and two types of communication actions: `send_vote (V, i, X)` and `receive_vote(V, X, i)`, and the rest are fluents. Note that the `i` is not a meta-variable; it will not be replaced. It is used to identify the node. Propositions (6)–(7) describe how the input actions (i.e., inserted by the administrator) can affect fluents of the form `neighbour(X)`. In Propositions (12)–(13) we are abusing notation and doing aggregation. Translations into logic program rules of this simple kind of aggregation can be done systematically. This is described in detail in Section 2.1.14 of (Baral 2003). The point is to have copies of this automaton in each node of the network and just change the identifier i for the proper identifier. So, if we have another node j and its automaton D_j, we can compose D_i and D_j. However, notice that output actions of these automata are of the form $\texttt{send\_vote(V, S, D)}$, and all the communication actions in the rest of the propositions are of the form $\texttt{receive\_vote(V, S, D)}$. Hence, as it is, there is no real interaction between the automata and all the elementary traces of $\Delta(D_i, D_j)$ have a single step. To make the connection, we can define another I/O automaton to model the communication between D_i and D_j and add it to the composition. For example, we can define the automaton C_S with a single communication law schema

$$\textbf{sent}\ \texttt{received\_vote(V, S, D)}\ \textbf{after}\ \texttt{send\_vote(V, S, D)}$$

This is actually modelling synchronous communication: all messages sent are passed simultaneously to all the receiving automata which will evaluate them simultaneously in the next step. The following is an elementary trace of $\Delta(D_i, D_j, C_s)$. We show only in details the content of the initial state s_0 and the actions in the trace

$$(\{\texttt{my\_opinion(good)}\}, \{\texttt{my\_opinion(bad)}\}, \emptyset), \{\texttt{add\_neighbour(i, j)}, \texttt{add\_neighbour(j, i)}\}, s_1,$$
$$\{\texttt{send\_vote(good, i, j)}, \texttt{send\_vote(bad, j, i)}\}, s_2, \{\texttt{received\_vote(good, i, j)},$$
$$\texttt{received\_vote(bad, j, i)}\}, s_3, \{\texttt{send\_vote(good, j, i)}, \texttt{send\_vote(bad, i, j)}\}, s_4, \ldots)$$

Note that more instances of D_i, representing more nodes in the graph, can be composed without changes in C_S. This method of representing communication using I/O automata is common in the formalization of distributed systems (Lynch 1996). And hence, the idea here is to use the same language to describe different communication models, e.g., (un)reliable and/or (a)synchronous, etc. For example, the following I/O automaton defines a reliable asynchronous model, called the fully interleaved model, as at each step only one node is activated. In this model, we describe each directional communication link as a message queue. At each step, a node is non-deterministically selected to activate. Upon activation, the node non-deterministically selects a non-empty communication link to dequeue, and processes the dequeued message (i.e., as the received action). If after the step, a non-empty set of messages (i.e., output action) are generated, they will be enqueued the corresponding communication links

$$\textbf{sent}\ \texttt{receive\_vote(V, From, To)}\ \textbf{if}\ \texttt{dequeued(From, To)}\ \textbf{after}\ \texttt{buffer(V, From, To, 0)}. \tag{15}$$

$$\textbf{caused}\ \texttt{buffer(V, From, To, Pos)}\ \textbf{if}\ \neg\texttt{dequeued(From, To)}\ \textbf{after}\ \texttt{buffer(V, From, To, Pos)}. \tag{16}$$

$$\textbf{caused}\ \texttt{buffer(V, From, To, Pos} - 1)\ \textbf{if}\ \texttt{dequeued(From, To)}\ \textbf{after}\ \texttt{buffer(V, From, To, Pos)}, \texttt{Pos} > 0. \tag{17}$$

$$\textbf{caused}\ \texttt{buffer(V, From, To, NextPos)}\ \textbf{if}\ \texttt{next\_queue\_pos(From, To, NextPos)}, \texttt{NextPos} < \kappa\ \textbf{after}\ \texttt{send\_vote(V, From, To)}. \tag{18}$$

$$\textbf{caused}\ \texttt{next\_queue\_pos(From, To,}\ \#count\texttt{<Pos>)}\ \textbf{if}\ \neg\texttt{dequeued(From, To)}\ \textbf{after}\ \texttt{buffer(V, From, To, Pos)}. \tag{19}$$

$$\textbf{caused}\ \texttt{next\_queue\_pos(From, To, }\#\mathit{count}\texttt{<Pos>)}\ \textbf{if}\ \texttt{dequeued(From, To)} \qquad (20)$$
$$\textbf{after}\ \texttt{buffer(V, From, To, Pos), Pos} > \texttt{0.}$$
$$\textbf{caused}\ \texttt{activated(}\#\mathit{chosen}\texttt{<To>)}\ \textbf{after}\ \texttt{buffer(V, From, To, 0).} \qquad (21)$$
$$\textbf{caused}\ \texttt{dequeued(}\#\mathit{chosen}\texttt{<From>, To)}\ \textbf{if}\ \texttt{activated(To)}\ \textbf{after}\ \texttt{buffer(V, From, To, 0).} \qquad (22)$$

Propositions (15) and (17) describe the effects of a dequeue operation: the first message (i.e., at position 0) in the buffer will be sent to the destination; the remaining messages are "shifted" one position to the front. If no dequeue operation is performed by the buffer, then all the messages remain at their position (i.e., Proposition (16)). The constant κ in Proposition (18) fixes the maximum size of the buffers. Note that during an enqueue operation, if the buffer is already full (i.e., the next available position is bigger than κ), then the message will not be added to the buffer (i.e., dropped). Propositions (21) and (22) capture the non-deterministic selection of an activating node and a dequeuing communication link, respectively, by the use of the non-deterministic operator *chosen*<>.[1] In particular, Proposition (21) non-deterministically selects one receiver node *To* from a set of buffered messages (i.e., **after** `buffer(V, From, To, 0)`) at the head of their corresponding queues (this guarantees that the activated node has at least one communication link to dequeue), and Proposition (22) non-deterministically selects a sender node `From` to fix a communication link to dequeue. The combined effect of (21) and (22) guarantees that at any step there are only one `activated` fluent and only one `dequeued` fluent which are true. This captures the behaviour of fully interleaved execution of any given protocol. Variants of the model can be obtained by modifying rules (22) and/or (21). For another example, if we want to capture that upon activation a node is allowed to dequeue all the non-empty incoming communication links and process multiple messages, then we can replace Proposition (22) with (23):

$$\textbf{caused}\ \texttt{dequeued(From, To)}\ \textbf{if}\ \texttt{activated(To)}\ \textbf{after}\ \texttt{buffer(V, From, To, 0).} \qquad (23)$$

Alternatively, if we want to allow all nodes with available incoming messages to be activated at each step (e.g., to maximize parallelism), we can replace rule (21) with (24):

$$\textbf{caused}\ \texttt{activated(To)}\ \textbf{after}\ \texttt{buffer(V, From, To, 0).} \qquad (24)$$

When both rules (23) and (24) are used, the constrained asynchronous model mimics exactly the synchronous model (it is easy to check that in this case the `next_queue_pos` tuples will always have 0 as the next position for any `From` and `To`).

4 Implementation and system execution

We have developed an infrastructure in Java called *Distributed State Machines* (DSM) for the development and execution of distributed applications (Lobo *et al.* 2012). It has a three-layer architecture (see Fig. 2).

[1] We are again abusing notation: translations to causal theories and logic programs of non-deterministic choice can be found in Turner (1999) and Baral (2003).

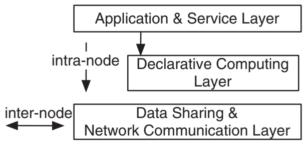
Application & Service Layer
intra-node
Declarative Computing Layer
inter-node
Data Sharing & Network Communication Layer

Fig. 2. System architecture.

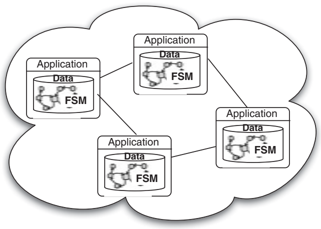

Fig. 3. DSM execution.

At the bottom, there is the Data Sharing and Network Communication Layer, which maintains low level data representation and storage, and handles inter-node communications. Various communication mechanisms (e.g., Ethernet, Wifi) or protocols (e.g., UDP, TCP) can be implemented, which remain hidden from the higher layers. The middle layer is called the Declarative Computing Layer. Applications written in $\mathscr{D}$ are executed in this layer.[2] At the Application and Service Layer, we find the application (or service) that might use the results of the distributed computation. For example, the router application that uses the forwarding tables maintained by the declarative rules to decide where to route the data packets, or the application that uses the fact that the node has been elected leader to start performing special operations.

During deployment and execution (see Fig. 3), the infrastructure provides an engine running on each network node. Each engine takes as input a $\mathscr{D}$ specification, executes the local I/O automaton and handles the inter-automaton (inter-node) communication. Relational databases are used as the primary data structure representing an automaton's state, i.e., each fluent refers to a record in a table with the matching fluent name, and its truth value is reflected by its presence in the table. Events executing communication or input actions, which can trigger local state transitions, are represented as transient records (i.e., inserted when they occur, and dropped after used) in their corresponding tables. During a transition, the new local state is computed, based on the $\mathscr{D}$ specification and the current local state, using a variant of the semi-naive evaluation algorithm of Datalog, and then stored. Any communication action computed during the transition is handled and sent out by the communication layer implementation of the engine.

We have developed various distributed applications, for instance, for the control of video analytics in a surveillance context, tracking of assets in a sensor field, fault management and resiliency in networked appliances, and for management of

[2] The current implementation does not support constraint propositions, i.e. proposition with $\perp$ as effect.

resources in a data center environment. Testing has been conducted in developing a universal proxy for P2P file sharing protocols, which has shown that the versatility of the declarative rule language allows us to quickly adapt the proxy to multiple protocols and easily reconfigure the proxy to support either caching or access control.

5 Analysis of distributed systems

Most properties of distributed systems can be reduced to properties of their traces. For example, questions of convergence in our voting example for the synchronous case are equivalent to finding an empty O_i, for some i. Given an initial configuration of the network, its elementary trace is unique, so showing divergence is equivalent to finding i and j, $i \neq j$, such as $s_i = s_j$ and $O_i = O_j$. For the asynchronous case, expressing the same properties is more complicated because a single network configuration may have multiple elementary traces. In fact, for the voting algorithm the same initial configuration can lead to both converging and diverging traces, but the same methods for the synchronous case can be used to check convergence or divergence of individual traces.

Besides the intrinsic value of declarative programming of distributed applications (Loo *et al.* 2009), the other contribution of this paper is to provide the formal foundations on which analysis tools can be developed. This is done by showing a 1-to-1 correspondence between traces of a composed automaton and the answer sets of a logic program obtained from the individual components of the composition.

Let $\Delta(D_1, \ldots, D_n)$ be the composition of n I/O automata. Let $(\mathbf{F}_i, \mathbf{A}_i)$ be the set of fluent and action names defining D_i. To define the logic programs $\Pi(D_i)$, for each D_i in the composition, we will need a set $\mathbf{T}$ of *time names* corresponding to an initial segment of the natural numbers. $\Pi(D_i)$ uses atoms of the form f^i_t and a_t, for each t in $\mathbf{T}$, each f in $\mathbf{F}_i$, and each a in $\mathbf{A}_i$ (note that super-indices are not used in actions). The translation follows the same pattern as the program defining the semantics of a single I/O automaton. Each static law of the form (1) in D_i is translated into the rules

$$\mathtt{F}^{\mathtt{i}}_{\mathtt{t}} \leftarrow \mathtt{not}\ \overline{\mathtt{G1}^{\mathtt{i}}}_{\mathtt{t}}, \ldots,\ \mathtt{not}\ \overline{\mathtt{Gn}^{\mathtt{i}}}_{\mathtt{t}}$$

for each $\mathtt{t}$ in $\mathbf{T}$. Each dynamic law of the form (2) in D_i is translated into the rules

$$\mathtt{F}^{\mathtt{i}}_{\mathtt{t+1}} \leftarrow \mathtt{not}\ \overline{\mathtt{G1}^{\mathtt{i}}}_{\mathtt{t+1}}, \ldots, \mathtt{not}\ \overline{\mathtt{Gn}^{\mathtt{i}}}_{\mathtt{t+1}}, \pi(\mathtt{H1}), \ldots, \pi(\mathtt{Hm})$$

Here, $\pi(\mathtt{Hj}) = \mathtt{Hj}^{\mathtt{i}}_{\mathtt{t}}$, if $\mathtt{Hj}$ is a fluent literal; otherwise $\pi(\mathtt{Hj}) = \mathtt{Hj}_{\mathtt{t}}$. Similar translation applies to the communication laws (3) in D_i

$$\mathtt{A}_{\mathtt{t+1}} \leftarrow \mathtt{not}\ \overline{\mathtt{G1}^{\mathtt{i}}}_{\mathtt{t+1}}, \ldots, \mathtt{not}\ \overline{\mathtt{Gn}^{\mathtt{i}}}_{\mathtt{t+1}}, \pi(\mathtt{H1}), \ldots, \pi(\mathtt{Hm})$$

In all cases, this is done for all possible values $\mathtt{t}$ in $\mathbf{T}$. To this set we also add, for each action name $\mathtt{A}$ and each fluent name $\mathtt{F}$ and each time name $\mathtt{t}$, the rules

$$\neg\mathtt{A}_{\mathtt{t}} \leftarrow \mathtt{not}\ \mathtt{A}_{\mathtt{t}} \qquad\qquad \neg\mathtt{F}^{\mathtt{i}}_{\mathtt{t}} \leftarrow \mathtt{not}\ \mathtt{F}^{\mathtt{i}}_{\mathtt{t}}$$

Let $\Pi(\Delta(D_1, \ldots, D_n)) = \bigcup_{1 \leqslant i \leqslant n} \Pi(D_i)$. Given a $t \in \mathbf{T}$, and an i, $1 \leqslant i \leqslant n$, for any set of fluent symbols s and any set of actions O, let $(s)^i_t = \{f^i_t | f \in s\}$ and $(O)_t = \{a_t | a \in O\}$.

Proposition 1
For a composed automaton $\Delta(D_1, \ldots, D_n)$, and $\mathbf{T}$ of size t,

$$(s_{0,1}, \ldots, s_{0,n}), O_0, (s_{1,1}, \ldots, s_{1,n}), O_1, \ldots, (s_{k,1}, \ldots, s_{k,n}), O_k$$

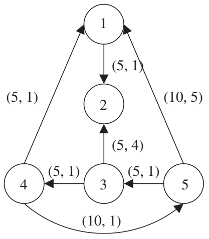

Fig. 4. Link-state.

is an elementary trace of the automaton if and only if

$$\left(\bigcup_{i=1}^{n}(s_{0,i})_0^i\right) \cup (O_0)_0 \cup \left(\bigcup_{i=1}^{n}(s_{1,i})_1^i\right) \cup (O_1)_1 \cup \cdots \cup \left(\bigcup_{i=1}^{n}(s_{k,i})_k^i\right) \cup (O_k)_k$$

is an answer set of

$$\Pi(\Delta(D_1,\ldots,D_n)) \cup \left(\bigcup_{i=1}^{n}(s_{0,i})^i\right) \cup (O_0)_0$$

This proposition is limited to elementary traces but a similar proposition can be written to cover general traces too. We have used this connection between traces and logic programs to build tools supported by ASP solvers to analyze different properties of distributed algorithms, and more specifically routing protocols. Routing protocols allow nodes to populate their routing tables and learn the forwarding paths so that traffic can be routed and delivered to the intended destination.

A routing protocol must satisfy two properties. First, it must *converge*: in the absence of topology change, all nodes should ultimately reach a consistent view of the network. Second, the resulting forwarding paths must be *devoid of loops*. Although these two properties are critical, verifying them has been a challenging problem. By using ASP, such analysis is possible (see online Appendix A for protocol specification in $\mathscr{D}$):

Forwarding loops: Once a protocol converges (described next), the forwarding table of each node in the network becomes stable. From the union of these tables, we can compute the transitivity closure with respect to different destinations, and detect the presence of loops in the forwarding tables. To illustrate it, we implemented the link state protocol suggested in Sobrinho and Griffin (2010): link state protocols have each node flood its topological information and is commonly believed not to result in loops since every participant node can reconstruct a global view of the entire network topology. However, (Sobrinho and Griffin 2010) demonstrated that depending on the metrics (i.e., how weights are assigned to paths), and the adopted path computation algorithm, persistent forwarding loops can actually also happen in link state protocols.

As depicted in Figure 4, each link is labelled with a tag (b, d) where b represents the bandwidth, and d the distance. As specified in Sobrinho and Griffin (2010), nodes prefer path with the largest available bandwidth, and among paths of identical bandwidth, select the one with the shortest distance to the destination. Formally, metric (b_1, d_1) is *better than* metric (b_2, d_2) if $b_1 > b_2$, or if $b_1 = b_2$ and $d_1 < d_2$. In

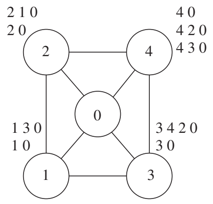

Fig. 5. Conf. B.

addition, the metric of a route with metric (b_1, d_1), exported over a link with metric (b_2, d_2), becomes $(min(b_1, b_2), d_1 + d_2)$. For each node, we adopted the *right local* algorithm (Sobrinho and Griffin 2010) to compute the best path (see code in online Appendix A.2). Answer set program solvers revealed the persistent forwarding loop 3–4–5–3 for destination 2.

Convergence: The Border Gateway Protocol (BGP) is the current de facto inter-domain routing protocol. Despite its important role, conflicting BGP policies can violate the convergence property and cause permanent route oscillations. The absence of specific patterns in the network topology and policies known as "dispute wheels" has been proved to be a sufficient condition for correctness. However, in the presence of a dispute wheel, determining whether a network can converge is NP-complete. Applying answer set programs, given a BGP configuration with dispute wheels – such as the one depicted in Figure 5 – we can answer whether it sometimes or always converges. In Figure 5, the vertical list next to each node represents the path ranking preference: e.g., node 4 prefers the path (4, 0) to the path (4, 2, 0), which is in turn preferred to (4, 3, 0). This BGP network includes a dispute wheel but always converges: there exists some i such that for all $j > i$, O_j is empty.

However, there are two limitations of the pure ASP approach for analysing convergence. The time sort to be finite, and hence when the convergence query fails (to find some i), we cannot tell whether it is due to protocol divergence or due to the time domain being too small. Secondly, though ASP is good for computing reachability queries (e.g., finding forwarding loops) it does not scale up for queries needing to reason over a set of traces. For example, it takes more than 7 hours for answering "always converge" to the above BGP configuration with 5 nodes. To do the analysis for this type of queries, we used a hybrid approach. We first construct the transition diagram of the composed automaton and then compute the queries as graph analysis. We used C++ to iteratively build the transition graph and ASP to compute all children states given a global state (which is the union of all local network node states and all the communication buffers). Such approach avoids the re-computation of the same global states (note that ASP cannot recognise the same state occurring at different time points) and hence makes performance improvement possible. With this approach we have checked BGP convergence in networks with dispute wheels of up to 7 nodes, which is twice larger than those considered by (Musuvathi and Engler 2004; Wang *et al.* 2012). Other experimental results are shown in Figure 6 (the configurations can be found in online Appendix B and BGP in Appendix A.1).

BGP Configurations	Total nodes	Converge?	Time
Dispute wheel size of 2	3	Sometimes	0.079s
Dispute wheel size of 3	4	Never	3.173s
Dispute wheel size of 4	5	Sometimes	25.784s
Dispute wheel size of 5	6	Never	4min
Dispute wheel size of 6	7	Sometimes	102min
Dispute wheel size of 7	8	Never	4503min
Conf. B	5	Always	123min
Dispute wheel size of 4	11	Sometimes	503min

Fig. 6. BGP convergence analysis results.

Remarks: we did not use model checking (MC) for the analysis of the composed I/O automaton because the state representations are different: a state is represented as a fixed set of variables in MC and as a set of fluents/actions in $\mathscr{D}$. Translating our analysis problem into model checking problem not only may cause state explosion (e.g., treating each fluent/action as a boolean variable) but also departs from our original objective of doing analysis directly on the implementation.

6 Discussion and related work

There has been a lot of work in distributed logic programming (references can be found as far back as the early 80's). The work presented here builds upon the recent results on Declarative Networking (Loo *et al.* 2009) (DN). DN introduces the idea of using Datalog-like languages (as opposed to full logic programs) as the computational model for distributed computing. Although the initial proposal of using Datalog was a leap forward toward a declarative approach for the specification of routing protocols, many operational properties of distributed asynchronous computation (e.g. process communication and state changes) were implicit in the implementations (Pérez and Rybalchenko 2009). In other words initial DN languages were not declarative enough (Mao 2009). The consequence has been a mismatch between the syntax and semantics of the languages: the syntax is Datalog, but the semantics are not. Hence, building analysis tools has not been easy. The limitation arises because the semantics of Datalog is too poor to express state changes. In an attempt to address this problem Alvaro et al modified the original proposal by adding a second argument to every predicate to represent time and called the new language Dedalus (Alvaro *et al.* 2011). With time states can be represented. However, Alvaro et al did not think in terms of state machine transitions, and because of the way time was modelled, in order to avoid limiting the expressiveness of the language, their semantics allows message sent from one node to another, to arrive at the receiving node as if the message had been sent from the future. This makes programming confusing. Furthermore, providing a declarative semantics to Dedalus has not been easy either and has required going out of logic programs. This limitation makes debugging and analysis also difficult. Another language proposal for declarative specifications of distributed computations is Netlog (Grumbach and Wang 2010). Motivated by the limitations of Dedalus, the authors of Netlog developed an extension of Datalog and a new fix-point semantics for which they built an implementation. Although inspired by the semantics of logic programs, this is a new programming language, and all the work of tools for program analysis are open research issues. In contrast to extending Datalog, (Lopes *et al.* 2010) proposed a Prolog-based system for declarative distributed system development, for which system analysis is not easy.

Our key insight, introduced in Lobo *et al.* (2012), has been to exploit the similarities between the computational model of the Datalog approach to distributed computation and theories of action to help us close the gap between syntax and semantics. However, as mentioned in Section 1, although our theories in Lobo *et al.* (2012) described state transition systems, their model is fundamentally different from the theories of actions built on the basis of two types of non-logical symbols: fluents and actions. This has been resolved with the results in this paper. As a consequence, we can model communication channels as I/O automata as oppose to integrity constraints over times. This allows us to develop a new analysis approach that gives much better performance results.

Recent work on modular logic programming in the context of ASP (see (Krennwallner and Gelfond 2011)) is somewhat related to our result in Proposition 1 of obtaining a single logic program from the composition of I/O automata. The goal of modularity is to call programs that have predicates defined externally, but new semantic notions need to be defined to accommodate the modular composition. In contrast, our composition is simpler and results in a single logic program with standard Answer Set semantics.

A line of research to be mentioned is reasoning about actions in multi-agent domains. In a series of papers Baral, Gelfond, Ponetielli and Son (e.g., (Baral *et al.* 2010; Pontelli *et al.* 2012)) have been developing a theory of reasoning about knowledge of interactive agents. Their aim is to study how individual agents can reason and plan about the knowledge of the other agents, in contrast to the work of distributed network applications where we want the theories to be exactly the programs that we deploy and run in the network nodes, and the reasoning is about the programs, i.e. the behaviour of the application as a whole in the network. In planning for multi-agent domains, the actions are supposed to be implemented by somebody else, and the domain specification just has a specification of the effects of the actions relevant for the planning, not the implementation.

Our approach is closer in goals to, for example, the work in Chandy *et al.* (2011) for distributed program verification. In general, program verification is a computationally harder problem than many planning tasks. This is reflected in the size of networks we are able to analyse for BGP convergence: 8 nodes so far. Although limited, with our approach we have been able to double the size of the problems tackled when compared to the state of the art (Wang *et al.* 2012). One important piece of work missing is the characterization of the computational complexity of the different analysis tasks.

Nevertheless, in the constraint model of our approach, we might be able to borrow ideas and results from planning since some analysis problems can be casted as planning problems. For example, Propositions (21) and (22) can be re-written in such a way that an omniscient agent can pick what message from the tops of the buffers to process next and model this as an action that the agent can take. Hence, for analysis, we could ask questions like, given a initial state, is there a sequence of actions that can take us to a particular state. The sequence of action will represent a trace of the execution.

So far we have done analysis over elementary traces, but analysis of traces where input actions are allowed will be extremely valuable since they model topological changes of the network. Planning techniques might also be useful here since analysis may require finding a sequence of input actions that causes certain behaviour in the system.

There are surprisingly many practical algorithms that can be expressed by this simple computational model, and hence target for our implementation and analysis. The computational approach has been used, in addition to the implementation of routing network protocols, in a diversity of areas such as in security and provenance of distributed database query processing (Marczak *et al.* 2010; Zhou *et al.* 2011), analysis of event systems and management of web applications (Abiteboul *et al.* 2011). We also plan to use our approach in two emerging areas of network management: Software Defined Networks (McKeown *et al.* 2008) and Named Data Networking (Zhang *et al.* 2010).

Acknowledgement

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

Abiteboul, S., Bienvenu, M., Galland, A. and Rousset, M.-C. 2011. Distributed datalog revisited. In *Datalog reloaded*, Springer, 252–261.

Alvaro, P., Ameloot, T. J., Hellerstein, J. M., Marczak, W. R. and den Bussche, J. V. 2011. A declarative semantics for dedalus. Tech. Rep. UCB/EECS-2011-120, UC Berkley.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press.

Baral, C., Gelfond, G., Son, T. C. and Pontelli, E. 2010. Using answer set programming to model multi-agent scenarios involving agents' knowledge about other's knowledge. In *AAMAS*, 259–266.

Chandy, K. M., Go, B., Mitra, S., Pilotto, C. and White, J. 2011. Verification of distributed systems with local–global predicates. *Formal Aspects of Computing 23,* 5, 649–679.

Gelfond, M. and Lifschitz, V. 1998. Action languages. *Electronic Transactions on Artificial Intelligence 2*, 193–210.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N. and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence 153,* 1, 49–104.

Grumbach, S. and Wang, F. 2010. Netlog, a rule-based language for distributed programming. In *PADL*, 88–103.

Krennwallner, T. and Gelfond, M. 2011. Promoting modular nonmonotonic logic programs. In *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, Vol. 11. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 274–279.

Lobo, J., Ma, J., Russo, A. and Le, F. 2012. Declarative distributed computing. In *Correct Reasoning*, 454–470.

Lobo, J., Wood, D., Verma, D. C. and Calo, S. B. 2012. Distributed state machines: A declarative framework for the management of distributed systems. In *CNSM*, 224–228.

Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T. and Stoica, I. 2009. Declarative networking. *CACM 52,* 11, 87–95.

Lopes, N. P., Navarro, J. A., Rybalchenko, A. and Singh, A. 2010. Applying prolog to develop distributed systems. *Theory and Practice of Logic Programming 10,* 4-6 (July), 691–707.

Lynch, N. A. 1996. *Distributed Algorithms.* Morgan Kaufmann.

Mao, Y. 2009. On the declarativity of declarative networking. *Operating Systems Review 43,* 4, 19–24.

Marczak, W. R., Huang, S. S., Bravenboer, M., Sherr, M., Loo, B. T. and Aref, M. 2010. Secureblox: customizable secure distributed data processing. In *Proceedings of the 2010 International Conference on Management of Data.* ACM, 723–734.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S. and Turner, J. 2008. Openflow: Enabling innovation in campus networks. *SIGCOMM Computer Communication Review 38,* 2 (March), 69–74.

Musuvathi, M. and Engler, D. R. 2004. Model checking large network protocol implementations. In *NSDI*, 440–441.

Pérez, J. A. N. and Rybalchenko, A. 2009. Operational semantics for declarative networking. In *PADL*, 76–90.

Pontelli, E., Son, T. C., Baral, C. and Gelfond, G. 2012. Answer set programming and planning with knowledge and world-altering actions in multiple agent domains. In *Correct Reasoning*, 509–526.

Sobrinho, J. L. and Griffin, T. G. 2010. Routing in equilibrium. In *Mathematical Theory of Networks and System.*

Turner, H. 1999. A logic of universal causation. *Artificial Intelligence 113,* 1-2, 87–123.

Wang, A., Talcott, C., Gurney, A. J. T., Loo, B. T. and Scedrov, A. 2012. Reduction-based formal analysis of bgp instances. In *Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, Springer-Verlag, Berlin, Heidelberg, 283–298.

Zhang, L., Estrin, D., Burke, J., Jacobson, V., Thornton, J. D., Smetters, D. K., Zhang, B., Tsudik, G., Massey, D., Papadopoulos, C., *et al.* 2010. Named data networking (ndn) project. *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC.*

Zhou, W., Fei, Q., Sun, S., Tao, T., Haeberlen, A., Ives, Z., Loo, B. T. and Sherr, M. 2011. Nettrails: a declarative platform for maintaining and querying provenance in distributed systems. In *Proceedings of the 2011 international conference on Management of data*, ACM, 1323–1326.