A Declarative Approach to Distributed Computing: Specification, Execution and Analysis

Jiefei Ma†, Franck Le⋆, David Wood⋆, Alessandra Russo‡, Jorge Lobo†
†ICREA - Universitat Pompeu Fabra, ‡Imperial College London, ⋆IBM Watson Laboratory, US
(e-mail: j.ma@imperial.ac.uk, fle@us.ibm.com, dawood@us.ibm.com, a.russo@imperial.ac.uk, jorge.lobo@upf.edu)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

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 the 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 (ASP). This work extends our initial results on the use of an A-type language for writing network applications, by showing that it is possible to achieve similar expressiveness and generality but using only two types of non-logical symbols (fluents and actions). 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. Their underlying semantic model assumes a network of nodes where data and computation are distributed and, where nodes work together by sharing data and computation results of the group. Perhaps the best example of distributed network applications is the set of routing protocols currently used to run and control the network traffic in the Internet. These are based on very robust algorithms that are highly distributed in order to make the networks very resilient to failures. But they are also very difficult to analyse, modify and maintain. This is mainly because the behaviour of these applications is not simply expressed by the local computation of a program but by the 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 difficulties 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 (blue color) or bad (red colour) and (1) sends it to all its neighbours;
(2) upon receiving a neighbour 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, for instance, checking whether the algorithm always converges (i.e. it always reaches a global state where no more node changes its opinion); checking if a specific global state is eventually reached (e.g. all nodes will eventually become good); or which initial global 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 the appropriate computational abstractions that allow distributed algorithms to be described simply, and amenable to efficient automated analysis.

A very common abstraction that is made to describe distributed algorithms is to view each computational node as an input/output automaton (Lynch 1996). We can find many useful/practical algorithms that can be defined using input/output automata in which their 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 like, for instance, 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 that 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 are able to 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 $C$ to specify distributed algorithms and then use the same language to formalize the network communication model. 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 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 provide a translation of the distributed algorithm implementation and the communication model from the causal logic based specification into logic programs providing with a formal foundation for the implementation of distributed applications and building analysis tools. We briefly describe a system, called...
DSM, for executing distributed algorithms written in our language, 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 to 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 $\mathcal{D}$ 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 $\mathcal{D}$ itself can be used to model the communication between the different distributed components. This section also introduces our DSM system for executing distributed applications written in $\mathcal{D}$, and the concept of traces as a tool to analyse these applications. Section 5 describes 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 $\mathcal{D}$

Distributed system components are typically modelled using input/output (I/O) automata (Lynch 1996). When compared to automata defined by languages like $A$ or $C$ (Gelfond and Lifschitz 1998), an I/O automaton adds to a transition diagram defined by the automaton an extra label in each of the diagram representing the output of the transition. Thus, the underlying signature of $\mathcal{D}$ is defined by two pairwise disjoint non-empty sets $(F, A)$ of fluent names and action names. For I/O automata, in addition, the set of action names $A$ is partitioned into a set of input actions $A_I$ and communication actions $A_C$. 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 $\text{vmachine}(\text{vm1}, \text{ser1})$ and $\text{vmachine}(\text{vm2}, \text{ser3})$, or $\text{vrouter}(\text{rt11}, \text{ser2})$ and $\text{vrouter}(\text{rt22}, \text{ser2})$ 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 $\text{new\_vm}(\text{vm5}, \text{ser5})$, $\text{remove}(\text{vm2})$. A communication can be interpreted as a message that is been sent between automata. The communication action $\text{where\_is}(\text{vm1})@\text{ser5} : \text{ser3}$ can be read as server 5 is interested to know the location of virtual machine vm1 and the request is been sent to server 3.

To define the transition diagram we use three types of propositions:

*static laws* of the form

\[
\text{caused } F \text{ if } G_1, \ldots, G_n
\]

*dynamic laws* of the form

\[
\text{caused } F \text{ if } G_1, \ldots, G_n \text{ after } H_1, \ldots, H_m
\]

and *communication laws* of the form

\[
\text{sent } A \text{ if } G_1, \ldots, G_n \text{ after } H_1, \ldots, H_m
\]

Here $F$ and the $G$'s are fluent literals. The $H$'s are either fluent or action literals. $A$ is a communication action and $A$ is called the *output communication action* of the proposition. The static and dynamic laws are like in the action language $C$ but restricted to literals. As in $C$, we write

\[
\text{caused } F \text{ after } H
\]
when \( n = 1 \) and \( G_1 \) is the constant \text{true}. As in the language \( C \), one can choose in \( \mathcal{D} \) fluent literals follow the law of inertia. Inertia is defined by dynamic laws of the form

\[
\text{caused } F \text{ if } F \text{ after } F
\]

and they will be abbreviated by

\[
\text{inertial } F
\]

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

\[
\text{caused } \text{vmachine}(V,L) \text{ after } \text{new_machine}(V,L)
\]

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 caused for a virtual machine \( V \) to be in location \( L \) after the input action \( \text{new_machine}(V,L) \) is executed. We would like the existence or not existence of virtual machines to persist by inertia, so we add the following propositions

\[
\text{inertial } \text{vmachine}(V,L) \\
\text{inertial} \neg \text{vmachine}(V,L)
\]

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

\[
\text{caused } \bot \text{ if } \text{vmachine}(V,L_1), \text{vmachine}(V,L_2), L_1 \neq L_2
\]

Recall again that \( V, L_1 \) and \( L_2 \) are meta-variables for all possible machine names and locations but in this case \( L_1 \) most be different to \( L_2 \). The following communication law

\[
\text{sent } \text{location}(V,L)@ser5:ser3 \text{ after } \text{vmachine}(V,L), \text{where_is}(V)@ser3:ser5
\]

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

### 2.1 Semantics of \( \mathcal{D} \)

Given the similarity to \( C \), we can defined the transition diagrams of I/O automata written in \( \mathcal{D} \) using causal rules. But because we only use conjunctions of literals in the propositions of \( \mathcal{D} \), the diagram can be described using a logic program under the Answer Set Semantics. We prefer this since it reflects directly how we have built our analysis tool, i.e. using an ASP solver as core of the analysis tool. We adapt the translation from (Lifschitz and Turner 1999) to our case. The logic program will have in addition to elements of \( \mathcal{F} \) and \( \mathcal{A} \) as predicates a copy \( f' \) and copy \( a'_c \) for each fluent \( f \) in \( \mathcal{F} \) and each communication action \( a_c \) in \( \mathcal{A} \). Each static law of the form (1) in an automaton \( \mathcal{D} \) is translated into the logic programming rules

\[
\mathcal{F} \leftarrow \neg G_1, \ldots, \neg G_n \\
\mathcal{F}' \leftarrow \neg G_1', \ldots, \neg G_n'
\]

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

\[
\mathcal{F}' \leftarrow \neg G_1', \ldots, \neg G_n', H_1, \ldots, H_m
\]

As similar translations applies to the communication laws (3) in \( \mathcal{D} \)

\[
\mathcal{A}' \leftarrow \neg G_1', \ldots, \neg G_n', H_1, \ldots, H_m
\]

Note that the notation \( \mathcal{L} \) 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

\[
\begin{align*}
\neg A & \leftarrow \text{not } A \\
\neg F & \leftarrow \text{not } F \\
& \leftarrow \text{not } F', \text{not } \neg F'
\end{align*}
\]

(4)

For any set of predicates \( P \), let \( P' = \{ p' | p \in P \} \). Let \( \Pi_D \) 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 \( i \) from all the actions in \( S \cap A'_C \).

This diagram defines the transition relation \( \text{trans}(D) \subseteq F \times A \times F \times A_c \).

Remarks: In terms of theory of causal explanation (McCain et al. 1997), input actions do not need cause, they have been called exogenous but communication actions do. McCain and Turner (McCain et al. 1997) have already mentioned the possibility of these actions which can be called events. These actions are the output of automata and might trigger the state transition of other automata. In the next section we will see how this inter-automata communication can be modelled.

3 Modeling distributed processing with \( D \)

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 the 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 the following condition holds:

- 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 \( D_1, \ldots, D_n \), the composition \( \Delta(D_1, \ldots, D_n) \) of these automata is an I/O automaton defined as follows

1. \((s_1, \ldots, s_n)\) is a state of \( \Delta(D_1, \ldots, D_n) \) iff \( s_i \) is a state of \( D_i \), for \( 1 \leq i \leq n \) and
2. \(((s_1, \ldots, s_n), E, (r_1, \ldots, r_n), O) \in \text{trans}(\Delta(D_1, \ldots, D_n)) \) iff \((s_i, E_i, r_i, O_i) \in \text{trans}(D_i) \), for \( 1 \leq i \leq n \), 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. A elementary trace of a composed automaton \( \Delta(D_1, \ldots, D_n) \), is a not necessarily finite sequence \( s_0, O_0, s_1, O_1, s_2, O_2, \ldots \) such that \((s_{i-1}, O_{i-1}, s_i, O_i) \in \text{trans}(\Delta(D_1, \ldots, D_n)) \) for every \( i \geq 1 \) 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 \( \Delta \), is a not necessarily finite sequence \( s_0, O_0, s_1, O_1, s_2, O_2, \ldots \) such that \((s_{i-1}, O_{i-1}, s_i, O_i \cap A_c) \in \text{trans}(\Delta) \) for every \( i \geq 1 \) 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

\begin{align*}
\text{inertial neighbour}(X), & \quad \neg \text{neighbour}(X). \\
\text{caused neighbour}(X) & \text{after add\_neighbour}(X, i). \\
\text{caused} & \quad \neg \text{neighbour}(X) \text{after del\_neighbour}(X, i). \\
\text{caused neighbour\_opinion}(X, V) & \text{after receive\_vote}(V, X, i). \\
\text{caused neighbour\_opinion}(X, \text{OldV}) & \text{after} \\
& \quad \text{neighbour\_opinion}(X, \text{OldV}), \neg \text{receive\_vote}(\text{AnyV}, X, i). \\
\text{caused my\_opinion}(\text{good}) & \text{if \ num\_goods}(N), \text{num\_bads}(M), N \geq M. \\
\text{caused my\_opinion}(\text{bad}) & \text{if \ num\_goods}(N), \text{num\_bads}(M), M > N. \\
\text{caused num\_good}(\# \text{count}<X>) & \text{if \ neighbour\_opinion}(X, \text{good}). \\
\text{caused num\_bad}(\# \text{count}<X>) & \text{if \ neighbour\_opinion}(X, \text{bad}). \\
\text{sent send\_vote}(V, i, X) & \text{if \ neighbour}(X), \\
& \quad \text{my\_opinion}(V) \text{after my\_opinion}(\text{OldV}), V \neq \text{OldV}.
\end{align*}

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 found in (Baral 2003). The point is to have copies of this automaton in each node of the network and just change the identifier $i$. 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 send\_vote($V, S, D$), and all the communication actions in the rest of the propositions are of the form 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

$$\text{send received\_vote}(V, S, D) \text{ after send\_vote}(V, S, D)$$

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 detail the content of the initial state $s_{0}$ and the actions in the trace

$$\langle \{ \text{my\_opinion}(\text{good}) \}, \{ \text{my\_opinion}(\text{bad}) \}, \emptyset \rangle, \{ \text{add\_neighbour}(i, j), \text{add\_neighbour}(j, i) \}, s_{1}, \{ \text{send\_vote}(\text{good}, i, j), \text{send\_vote}(\text{bad}, j, i) \}, s_{2}, \{ \text{received\_vote}(\text{good}, i, j), \text{received\_vote}(\text{bad}, j, i) \}, s_{3}, \{ \text{send\_vote}(\text{good}, i, j), \text{send\_vote}(\text{bad}, i, j) \}, s_{4}, \ldots$$

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). The following I/O automaton defines an 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 to the corresponding communication links.

\[ \text{sent} \text{receive} \text{vote}(V, \text{From}, \text{To}) \text{if} \text{dequeued}(\text{From}, \text{To}) \text{after} \text{buffer}(V, \text{From}, \text{To}, 0). \]  \hspace{1cm} (15)

\[ \text{caused buffer}(V, \text{From}, \text{To}, \text{Pos}) \text{if} \text{dequeued}(\text{From}, \text{To}) \text{after} \text{buffer}(V, \text{From}, \text{To}, \text{Pos}). \]  \hspace{1cm} (16)

\[ \text{caused buffer}(V, \text{From}, \text{To}, \text{Pos} - 1) \text{if} \text{dequeued}(\text{From}, \text{To}) \text{after} \text{buffer}(V, \text{From}, \text{To}, \text{Pos}). \]  \hspace{1cm} (17)

\[ \text{caused buffer}(V, \text{From}, \text{To}, \text{NextPos}) \text{if} \text{next_queue_pos}(\text{From}, \text{To}, \text{NextPos}), \text{NextPos} < \kappa \]  \hspace{1cm} (18)

\[ \text{after} \text{send}\text{vote}(V, \text{From}, \text{To}). \]

\[ \text{caused next_queue_pos}(\text{From}, \text{To}, \#\text{count}<\text{Pos}>) \text{if} \text{dequeued}(\text{From}, \text{To}) \text{after} \text{buffer}(V, \text{From}, \text{To}, \text{Pos}). \]  \hspace{1cm} (19)

\[ \text{caused next_queue_pos}(\text{From}, \text{To}, \#\text{count}<\text{Pos}>) \text{if} \text{dequeued}(\text{From}, \text{To}) \text{after} \text{buffer}(V, \text{From}, \text{To}, \text{Pos}). \]  \hspace{1cm} (20)

\[ \text{caused activated}(\#\text{chosen}<\text{To}>) \text{after} \text{buffer}(V, \text{From}, \text{To}, 0). \]  \hspace{1cm} (21)

\[ \text{caused dequeued}(\#\text{chosen}<\text{From}>, \text{To}) \text{if} \text{activated}(\text{To}) \text{after} \text{buffer}(V, \text{From}, \text{To}, 0). \]  \hspace{1cm} (22)

The constant \( \kappa \) in Proposition (18) fixes the maximum size of the buffers. 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 \( \text{chosen}\). In particular, Proposition (21) non-deterministically selects one receiver node \( \text{To} \) from a set of buffered messages (i.e., \( \text{after buffer}(V, \text{From}, \text{To}, 0) \)) 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 \( \text{From} \) to fix a communication link to dequeue. The combined effect of (21) and (22) guarantees that at any step there are only one \text{activated} fluent and only one \text{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):

\[ \text{caused dequeued}(\text{From}, \text{To}) \text{if} \text{activated}(\text{To}) \text{after} \text{buffer}(V, \text{From}, \text{To}, 0). \]  \hspace{1cm} (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):

\[ \text{caused activated}(\text{To}) \text{after} \text{buffer}(V, \text{From}, \text{To}, 0). \]  \hspace{1cm} (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 \text{next_queue_pos} tuples will always have 0 as the next position for any \text{From} and \text{To}).

\[ \text{We are again abusing notation but translations to causal theories and logic programs of non-deterministic choice can be found in (Turner 1999) and (Baral 2003).} \]
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 Figure 2).

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. Here is where applications written in $\mathcal{D}$ are executed.\(^2\) 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 transient 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 Figure 3), the infrastructure provides an engine (which realises the three-layer architecture) running on each network node. Each engine takes as input a $\mathcal{D}$ specification, automatically executes the local I/O automaton and handles the inter-automaton (inter-node) communications. 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. The events of receiving new messages or input data, which are represented as transient records (i.e., inserted when they occur, and dropped after use) in their corresponding tables, can trigger local state transitions. During a transition, the new local state is computed, based on the $\mathcal{D}$ specification and the current local state, using a variant of the semi-naive fixed-point 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 used our approach to build 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 resources in a data center environment. Testings have been conducted in developing a universal proxy for P2P file sharing protocols, which have 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 cases is

\(^2\) The current implementation does not support constraint propositions, i.e. proposition with $\bot$ as it effect.
equivalent to finding an empty \( O_i \), for some \( i \). In the synchronous case too, 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 than 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 and let \((F_i, A_i)\) 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 \( T \) of time names corresponding to an initial segment of the natural numbers. The program \( \Pi(D_i) \) uses atoms of the form \( f^j_i \) and \( a_i \), for each \( t \) in \( T \), each \( f \) in \( F_i \), and each \( a \) in \( A_i \) (note that there is no super-index \( i \) in the actions). The translation follows the same pattern than 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 logic programming rules

\[
P^j_i \leftarrow \neg \overline{G^{j}}_{t}^{T}, \ldots, \neg \overline{G^{n}}_{t}^{T}
\]
for each \( t \) in \( T \). Each dynamic law of the form (2) in \( D_i \) is translated into the rules

\[
P^{j}_{t+1} \leftarrow \neg \overline{G^{j}}_{t+1}^{T}, \ldots, \neg \overline{G^{n}}_{t+1}^{T}, \pi(H1), \ldots, \pi(Hm)
\]

Here, \( \pi(Hj) = Hj^k_i \), if \( Hj \) is a fluent literal; otherwise \( \pi(Hj) = Hj \). A similar translations applies to the communication laws (3) in \( D_i \)

\[
A_{t+1} \leftarrow \neg \overline{G^{i}}_{t+1}^{T}, \ldots, \neg \overline{G^{n}}_{t+1}^{T}, \pi(H1), \ldots, \pi(Hm)
\]
In all cases, this is done for all possible values \( t \) in \( T \). To this set we also add, for each action name \( A \) and each fluent name \( F \) and each time name \( t \), the rules

\[
\neg A_t \leftarrow \neg A_c \quad \neg F_t^i \leftarrow \neg F_c^i
\]

Let \( \Pi(\Delta(D_1, \ldots, D_n)) = \bigcup_{1 \leq i \leq \epsilon} \Pi(D_i) \). Given a \( t \in T \), and an \( i, 1 \leq i \leq \epsilon, \) for any set of fluent symbols \( s \) and any set of actions \( O \), let \( (s)_t^i = \{ f^j_i \mid f \in s \} \) and \( (O)_t = \{ a_i \mid a \in O \} \).

**Proposition 1**

For a composed automaton \( \Delta(D_1, \ldots, D_n) \), and \( T \) of size \( t \),

\[
(s_0,1, \ldots, s_0,n), O_0, (s_1,1, \ldots, s_1,n), O_1, \ldots, (s_k,1, \ldots, s_k,n), O_k
\]

is an elementary trace of the automaton if and only if

\[
\bigcup_{i=1}^{n} (s_0,i)_0 \bigcup_{i=1}^{n} (s_1,i)_1 \bigcup_{i=1}^{n} (s_1,i)_1 \bigcup_{i=1}^{n} (s_k,i)_k \bigcup_{i=1}^{n} (O_k)_k
\]
is an answer set of

\[
\Pi(\Delta(D_1, \ldots, D_n)) = \bigcup_{i=1}^{n} (s_0,i)_0
\]

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 tasks are possible, and more details of the tasks follow (the protocol specification in $\mathcal{D}$ can be found in Appendix A).

**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 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 (Sobrinho and Griffin 2010).

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 addition, the metric of a route with metric $(b_1, d_1)$, exported over a link with metric $(b_2, d_2)$, becomes $(\text{min}(b_1, b_2), d_1 + d_2)$. For each node, we adopted the right local algorithm (Sobrinho and Griffin 2010) to compute the best path (see code in Appendix A). Answer set program solvers revealed the persistent forwarding loop $3\rightarrow 4\rightarrow 5\rightarrow 3$ 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 potentially 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.
First, it requires that 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 loop or a convergent trace), it cannot scale up well for queries that require reasoning 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. Therefore, to scale up the analysis for this type of queries, we use a hybrid approach which first constructs the transition diagram of the composed automaton using an ASP solver and then computes the queries as graph analysis. Such approach avoids the re-computation of the same global states and hence makes performance improvement possible. In fact, with this approach we managed to check convergence for BGP with dispute wheel of up to 7 nodes, size that is twice larger than those considered by (Musuvathi and Engler 2004; Wang et al. 2012). Some other experimental results are shown in Table 1 (description of the configurations can be found in Appendix B and BGP in Appendix A).

<table>
<thead>
<tr>
<th>Configurations</th>
<th>Total Nodes</th>
<th>Converge?</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGP with dispute wheel size of 2</td>
<td>3</td>
<td>Sometimes</td>
<td>0.079s</td>
</tr>
<tr>
<td>BGP with dispute wheel size of 3</td>
<td>4</td>
<td>Never</td>
<td>3.173s</td>
</tr>
<tr>
<td>BGP with dispute wheel size of 4</td>
<td>5</td>
<td>Sometimes</td>
<td>25.784s</td>
</tr>
<tr>
<td>BGP with dispute wheel size of 5</td>
<td>6</td>
<td>Never</td>
<td>4m</td>
</tr>
<tr>
<td>BGP with dispute wheel size of 6</td>
<td>7</td>
<td>Sometimes</td>
<td>102m</td>
</tr>
<tr>
<td>BGP with dispute wheel size of 7</td>
<td>8</td>
<td>Never</td>
<td>4503m</td>
</tr>
<tr>
<td>BGP Conf. B</td>
<td>5</td>
<td>Always</td>
<td>123m</td>
</tr>
<tr>
<td>BGP with dispute wheel size of 4</td>
<td>11</td>
<td>Sometimes</td>
<td>503m</td>
</tr>
</tbody>
</table>

Table 1. BGP Convergence Analysis Results

6 Discussion and related work

Although in the past 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). Declarative Networking introduces the idea of using Datalog-like languages (as supposed to full logic programs) as the computational model for distributed computing. But, 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 declarative networking languages were not declarative enough (Mao 2009). The consequence has been a mismatch between the syntax and semantics of the languages: although the syntax of the languages is Datalog, their 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 limitation Alvaro et al in (Alvaro et al. 2011) modified the original proposal by adding a second argument to every predicate to represent time and called the new language Dedalus. 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 (Alvaro et al. 2011). This limitation makes debugging and analysis also difficult.
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 the introduction, although our theories in (Lobo et al. 2012) described state transition systems, their model is fundamentally different from the theories of actions that are built on the basis of two types of non-logical symbols: fluents and actions. This has been resolved with the results in this paper.

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., (Pontelli et al. 2012; Baral et al. 2010)) 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 more computationally complex problem than many planning tasks. This is reflected in the size of networks we are able to analyse for BGP convergence: a maximum of 8 nodes. Although limited, with our approach we have been able to double the size of the problems tackled in the state of the art (Wang et al. 2012). This has been possible by doing analysis directly over the state diagram generated by the system as whole, instead of analysing answer sets. We still need an ASP solver to generate the graph. 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 asynchronous event systems (Abiteboul et al. 2011), and management of web applications (Abiteboul et al. 2011) and we are planning 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).
References


Appendix A Protocol Specification in \( \mathcal{D} \)

A.1 BGP Protocol and Convergence Query

```plaintext
% ### BGP PROTOCOL SPECIFICATION ###

inertial_neighbour(Peer).
inertial_neighbour(Peer).
caused_neighbour(X)
after add_neighbour(X, i).
caused_neighbour(X)
after del_neighbour(X, i).

inertial_route_policy(Dest, Path, LocalPref).
inertial_route_policy(Dest, Path, LocalPref).
caused_route_policy(Dest, Path, LocalPref)
after add_route_policy(Dest, Path, LocalPref).
caused_route_policy(Dest, Path, LocalPref)
after del_route_policy(Dest, Path, LocalPref).

% update of the received routes table
caused_route_candidate(Dest, NextHop, Path)
if route_policy(Dest, Path, LocalPref)
after receive(update(Dest, Path, NextHop, i)).
caused_route_candidate(Dest, NextHop, Path)
if not route_updated(Dest, NextHop),
after route_candidate(Dest, NextHop, Path).

caused_route_updated(Dest, NextHop)
after receive(update(Dest, Path, NextHop, i)).
caused_route_updated(Dest, NextHop)
after receive(withdraw(Dest, NextHop, i)).

% computing best route
caused_best_route(Dest, BestPath)
if best_next_hop(Dest, BestNextHop),
route_candidate(Dest, BestNextHop, BestPath).
caused_has_route(Dest)
if route_candidate(Dest, NextHop, Path).

% #max<> is an aggregation function takes all the possible values of % "LocalPref" and returns the maximum one
caused_best_local_pref(Dest, #max<LocalPref>)
if has_route(Dest),
routecandidate(Dest, NextHop, Path),
route_policy(Dest, Path, LocalPref).

% #min<> is an aggregation function takes all the possible values of % "Cost" and returns the minimum one
caused_best_cost(Dest, #min<Cost>)
if has_route(Dest),
best_local_pref(Dest, MaxLocalPref),
routecandidate(Dest, NextHop, Path),
route_policy(Dest, Path, MaxLocalPref),
% #length<> is a function takes a path and returns its length
```
A Declarative Approach to Distributed Computing

Cost = # length <Path>.
cause best_next_hop(Dest, #min<NextHop>)
    if has_route(Dest),
        best_local_pref(Dest, MaxLocalPref),
        best_cost(Dest, MinCost),
        route_candidate(Dest, NextHop, Path),
        route_policy(Dest, Path, MaxLocalPref),
        # length <Path> == MinCost.

% handling outgoing routes
sent update(Dest, NewPath, i, Peer)
    if best_route(Dest, Path),
        NewPath := # append<i, Path>,
        neighbour(Peer),
        # member<> is a function returns 0 if "Peer" is not a member
        % of "NewPath", and returns 1 otherwise.
        # member<Peer, NewPath> == 0,
        after ~best_route(Self, Dest, Path, t-1).

sent withdraw(Dest, i, Peer)
    if best_route(Dest, Path),
        NewPath := # append<i, Path>,
        neighbour(Peer),
        # member<Peer, NewPath> == 1,
        after ~best_route(Self, Dest, Path, t-1).

send(withdraw(Dest), i, Peer)
    if ~has_route(Dest),
        neighbour(Peer),
        after best_route(Dest, OldPath).

% ### CONVERGENCE QUERY ###
has_non_empty_buffer(T) :-
    buffer(V, From, To, 0, T).

convergent_time(T) :-
    time(T),
    T > 0,
    not has_non_empty_buffer(T),
    T1 = T -1,
    has_non_empty_buffer(T1).

A.2 Link State Protocol and Forwarding Loop Query

% ### LINK STATE PROTOCOL SPECIFICATION ###
inertial neighbour(Peer).
inertial ~neighbour(Peer).
caused neighbour(X)
    after add_neighbour(X, i).
caused ~neighbour(X)
    after del_neighbour(X, i).

% store received links
inertial stored_link(From, To, Band, Dist).
caused stored_link(From, To, Band, Dist)
    after receive(link(From, To, Band, Dist, Sender, i)).

% forward received links
sent link(From, To, Band, Dist, i, Peer)
    if neighbour(Peer),
        ~stored_link(From, To, Band, Dist),
        Peer != Sender,
        after receive(link(From, To, Band, Dist, Sender, i)).
% compute best paths
caused best_path_to(Dest, Path, Band, Dist)
    if best_path_to(Mid, PathSoFar, BandSoFar, DistSoFar),
        stored_link(Mid, Dest, BandOfLink, DistOfLink),
        #member<Dest, PathSoFar> == 0,
        Path = #append<PathSoFar, Dest>,
        Band = #min<BandSoFar, BandOfLink>,
        Dist = DistSoFar + DistOfLink,
    -better_path_to(Dest, Band, Dist).

cause better_path_to(Dest, Band, Dist)
    if best_path_to(SomeMid, SomePath, SomeBand, SomeDist),
        stored_link(SomeMid, Dest, AnotherBand, AnotherDist),
        #member<Dest, SomePath> == 0,
        NewBand = #min<AnotherBand, SomeBand>,
        NewDist = AnotherDist + SomeDist,
        NewDist > Band.

cause better_path_to(Dest, Band, Dist)
    if best_path_to(SomeMid, SomePath, SomeBand, SomeDist),
        stored_link(SomeMid, Dest, AnotherBand, AnotherDist),
        #member<Dest, SomePath> == 0,
        Band == #min<AnotherBand, SomeBand>,
        NewDist = AnotherDist + SomeDist,
        NewDist < Dist.

% generate the forwarding table
caused forwarding_table(Dest, NextHop)
    if best_path_to(Dest, Path, Band, Dist),
        Dest != i,
        NextHop = #list_nth_elem<2, Path>.

% ### FORWARDING LOOP QUERY IN ASP ###
q_arc(Node, NextHop, Dest) :-
    convergent_time(T),
    forwarding_table(Node, Dest, NextHop).

q_path(X, Y, #init_list<Y>, Dest) :-
    q_arc(X, Y, Dest),
    node(Dest).
q_path(X, Y, #append<Y, Path>, Dest) :-
    q_path(X, Z, Path, Dest),
    q_arc(Z, Y, Dest),
    #member<Y, Path> == 0.

q_cycle(Dest, #append<Path, From>) :-
    q_path(From, To, Path, Dest),
    q_arc(To, From, Dest).

Appendix B BGP Configurations

![BGP with wheel size of 2](image_url)

Fig. B1. BGP with wheel size of 2
Figure B1 is a BGP configuration involving three nodes, where 0 is the origin (i.e., the destination of all packets) and 1 and 2 form a dispute wheel (Griffin et al. 2002). To see why it is called a dispute wheel, consider the following scenario where both 1 and 2 receive a route “0” from the origin. Since 1 does not have any route to the origin initially, it will select “1 → 0” (i.e., appends itself to the route) as its best route, and sends it to 2. Similarly, 2 chooses “2 → 0” as its best route and sends it to 1. Now 1 has two route candidates to the origin: “1 → 0” and “1 → 2 → 0”. According to its local preference, it will choose “1 → 2 → 0” as its new best route and then informs 2 to withdraw any route containing 1 (so that it will not contain a forwarding loop). Similar, 2 does the same thing by selecting a new best route “2 → 1 → 0” and informing 1 to withdraw any route containing 2. Finally, after the two nodes withdraw their longer candidate routes to the origin, they have to choose “1 → 0” and “2 → 0”, respectively, as their latest best routes, and hence start another route switching cycle.

![Fig. B2. BGP with wheel size of 3](image)

We can increase the wheel size of Figure B1 by adding an extra node with appropriate local preference policy between 1 and 2, e.g., to obtain Figure B2. We have done this for up to wheel size of 7, and the first six rows in Table 1 shows the experimental results for these configurations. The final row shows the result for a BGP configuration depicted in Figure B3. This configuration contains 11 nodes but the dispute wheel size is only 4.

![Fig. B3. Big BGP with wheel size of 4](image)