Declarative Framework for Specification, Simulation and Analysis of Distributed Applications

Jiefei Ma, Frank Le, Alessandra Russo, and Jorge Lobo,

Abstract—Researchers have recently shown that declarative database query languages, such as Datalog, could naturally be used to specify and implement network protocols and services. In this paper we present a declarative framework for the specification, execution, simulation and analysis of distributed applications. Distributed applications, including routing protocols, can be specified using a Declarative Networking language, called D2C, whose semantics captures the notion of a Distributed State Machine (DSM), i.e. a network of computational nodes that communicate with each other through the exchange of data. The D2C specification can be directly executed using the DSM computational infrastructure of our framework. The same specification can be simulated and formally verified. The simulation component integrates the DSM tool within a network simulation environment and allows developers to simulate network dynamics and collect data about the execution in order to evaluate application responses to network changes. The formal analysis component of our framework, instead, complements the empirical testing by supporting the verification of different classes of properties of distributed algorithms, including convergence of network routing protocols. To demonstrate the generality of our framework, we show how it can be used to analyse two classes of network routing protocols, a path vector and a Mobile Ad-Hoc Network (MANET) routing protocol, and execute a distributed algorithm for pattern formation in multi-robot systems.

1 Introduction

Declarative Networking (DN) was first introduced in [1] as an extensible programming language infrastructure that uses database-style query languages to specify and implement network protocols and services. These languages extend the semantics of Datalog (SQL + recursion) by including constructs to specify physical distribution properties typical of networks such as where tuples in distributed database tables are generated and stored, data streams between nodes and table updates.

The realization that Datalog rules can serve as a vehicle to specify and implement routing network protocols has provided a new perspective to explore new declarative approaches for the programming of distributed applications. In particular, it has opened new avenues for the formal verification of protocols.

Contrary to imperative languages, which describe *how* computation should be executed, DN languages, such as [2], [3], [4], strive to specify *what* computation should be performed. Since the declarative code focuses on the intent versus the step-by-step descriptions of imperative code, DN makes the specifications of complex network applications very concise and intuitive, and their execution realisable through algorithms of distributed query processing (e.g. the implementation of the popular overlay multicast protocol Narada requires 16 rules, and the Chord peer-to-peer protocol requires 47 rules [5]).

- *Jiefei Ma and Alessandra Russo are with the Department of Computing, Imperial College London, UK. E-mail: {j.ma, a.russo}@imperial.ac.uk*
- *Frank Le is with IBM, US. E-mail: fle@us.ibm.com*
- *Jorge Lobo is with ICREA - UPF, Spain. E-mail: jorge.lobo@upf.edu*

In this paper we present a novel declarative networking framework (Section 2) for the development of distributed applications. At the core of this framework is a new declarative language called *D2C*. The formal underpinning of *D2C* comes from the language $\mathcal{D}$ introduced in [6]. *D2C* can be intuitively described as the Datalog-like subset of $\mathcal{D}$. Existing DN languages, such as NDLog [6], make communication between elements in a network an implicit aspect of the specification. The communication is implicitly determined by the specification of the data partition and the distributed query evaluation process. This main characteristic has the following consequences (i) leaving the designer agnostic of the actual communication model limits the type of distributed algorithms that the designer can specify since, for example, there are algorithms that are correct only if the communication is asynchronous and vice versa, and perhaps more importantly (ii) for analysis, it requires additional translation of the specification into a verification modelling language (e.g. [7], [8]). In contrast to NDLog and other existing DN languages (e.g. [2], [3], [4]), our *D2C* is a logic language that allows for (i) *explicit* process communication and state changes, and (ii) the representation of communication models (e.g. asynchronous vs. synchronous, or reliable vs. unreliable communication) within the same language used for specifying distributed applications. Even though our programs may require a few more rules, these new features of our language offer the key advantage that the same D2C specification of a distributed algorithm can be used for both execution and formal analysis. These main features have let us create a unifying environment for the simulation, execution and analysis of distributed applications.

To demonstrate the generality of our framework, we show its application to two different classes of distributed problems: network routing protocols and multi-robot system control. In the first application (Section 4.1) the framework is used to specify two classes of network protocols – a Mobile Ad-Hoc Networks (MANET) protocol and a simplified version of BGP [9], and analyse them with respect to two very different types of properties – disjoint paths and convergence over topologies with dispute wheel [10], respectively. In the second application (Section 4.2) we describe how the framework can be used to implement and execute a distributed algorithm for multi-robot pattern formation. Each robot uses a *D2C* engine to make decision on its own movements and inform neighbour robots of its new location, with the objective of cooperatively reaching a predefined pattern formation. The framework is described in Section 2. Section 2.1 presents the language *D2C*. Section 2.2 describes our current execution environment for distributed applications programmed in *D2C*. The simulation environment is summarised in Section 2.3 and the analysis component in Section 3. Sections 5 and 6 describe related work and future research.

2 The Proposed Framework

Our framework has three main components: *execution*, *simulation* and *analysis* (see Figure 1). Distributed algorithms are expressed using our D2C language. The computation model of the language is defined as a collection of *distributed state machines* (DSM), where each node (e.g., host, router) in a given network is abstracted as an input/output automaton that can exchange messages with other automata. State transitions in the automaton are triggered when it receives *external* information (e.g. from other nodes' automata). State transition functions are specified using declarative rules.

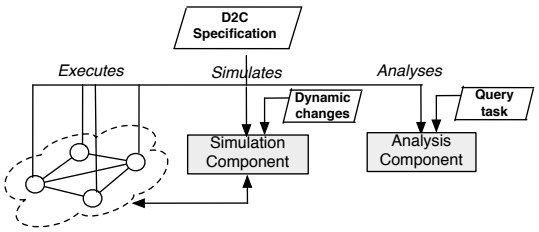


Fig. 1: D2C Framework

The D2C language can be executed using the DSM component. This assumes a network of nodes where each node may run the same or different set of D2C rules. Nodes maintain their own local data, and collaborate with other nodes by sharing data following their D2C specification. D2C specifications can be simulated to empirically evaluate the response of distributed algorithms. The simulation component integrates the DSM computational infrastructure within a simulated network topology infrastructure. It can take as input *dynamic* changes to the network topology. These changes can be either predefined or dynamically injected into the topology during the execution of the algorithm. The analysis component supports formal analysis of a D2C specification with respect to given (set of) properties (i.e. query tasks). Each analysis task comes with its own specific property to analyse, which can also be expressed in D2C. Analysis can be performed using different (pre-defined) communication models (e.g., asynchronous, synchronous).

2.1 The D2C Language

We present here the main features of our D2C language. We will use a simple distance vector routing protocol as a running example. The protocol assumes that each router knows the cost to reach each of its directly connected neighbours, and it works as follows. Each router waits for a change in the local link costs or a message from some neighbour. Upon receiving a message, the router estimates the least costs to different destinations based on the sum of the latest local link costs and the latest neighbour-to-destination costs. If the least cost to any destination has changed, the router notifies its neighbours through messages.

Each node stores a set of *named tuples* in the form of $\mathtt{p}(\mathtt{a}_1, \ldots, \mathtt{a}_n)$ where $\mathtt{p}$ is the tuple name and $\mathtt{a}_1, \ldots, \mathtt{a}_n$ (sometimes abbreviated as $\overline{\mathtt{a}}$) is a list of constants. These tuples represent the node's current state, and hence are called *state tuples*. An example of a state tuple in the distance vector routing protocol is $\mathtt{direct\_link}(\mathtt{rtr1}, 1)$. This tuple records the information of a directly connected neighbour node `rtr1` with link cost `1`. Nodes can also receive (named) tuples from outside the network. For example an administrator can execute a command "*add_link rtr2 1*" at a router node, to inform the node that a new directly connected neighbour `rtr2` exists with link cost `1`. The effect of this command would be that the router node receives the named tuple $\mathtt{add\_link}(\mathtt{rtr2}, 1)$ from outside the network (e.g. from the administrator). We will call these tuples *input tuples*. Finally, (named) tuples can be sent by nodes as messages. Tuples in messages will be called *transport tuples*. For instance, a transport tuple $\mathtt{dist\_msg}(\mathtt{rtr3}, 3)$ can be used to represent an inter-router message, containing a destination `rtr3` and cost `3`, sent from a sender router to another router. The three different sets of tuple names, state, input and transport, are assumed to be mutually exclusive.

Each state transition in an automaton is triggered when the automaton receives input or transport tuples. Our D2C language describes the state transition function using *declarative rules* and *tuple schemas*. In our language, these rules do not need to be the same at each node. In routing protocols, for instance, nodes may have different path selection or filtering policies. An input or state *tuple schema* is a tuple of the form $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ where each $\mathtt{t}_i$ is either a constant, a variable or an expression involving constants and variables. Transport tuples can be either the input or the output of a state transition.

Every transport *tuple schema* must have a suffix of the form @From:To, where From and To are either constants representing node's identifiers, or variables whose possible constant values are node identifiers. Declarative rules that compute state transitions are of the form:

$$\mathtt{S}\ \textbf{if}\ \mathtt{L_1}, \ldots, \mathtt{L_k}, \textbf{not}\ \mathtt{L_{k+1}}, \ldots, \textbf{not}\ \mathtt{L_n}, \mathtt{C}$$

where S is a state or transport tuple schema, each $\mathtt{L_i}$ ($0 < i \leq n$) is an input, state or transport tuple, possibly preceded by the operator **prev**, and C a set of equality and inequality constraints involving constants or variables that appear in the rule. S is called the *head* and $\mathtt{L_1}, \ldots, \mathtt{L_k}, \textbf{not}\ \mathtt{L_{k+1}}, \ldots, \textbf{not}\ \mathtt{Ln}$ is called the *body* of the rule. Tuple schemas preceded by **prev** in the body of a rule refer to state tuples before the state transition. All other state tuples refer to the new state of a node after the transition. So the body of a rule may include mixed state tuples from states before or after the transition.

When a transport schema $\mathtt{p(t_1, ..., t_n)@From : To}$ appears in the body of a rule, it represents a transport tuple $\mathtt{p(t_1, ..., t_n)}$ sent by a node with identifier From (i.e., as the output of one of its state transitions), to a receiver node with identifier To. Similarly, when it appears in the head of a rule. It represents a transport tuple $\mathtt{p(t_1, ..., t_n)}$ sent by a node with identifier From to a receiver node with identifier To. A special constant self is used to denote the identifier of a current node. Hence, declarative rules that compute transport tuples are similar to the rules that compute state tuples with the only difference that their heads are transport tuples.

In the distance vector routing protocol, the rules that define direct_link state tuples, at router rtr1, are:

```
direct_link(Peer, Cost) if                                  (1)
    add_link(Peer, Cost)@admin:self.
direct_link(Peer, OldCost) if                               (2)
    prev direct_link(Peer, OldCost),
    not add_link(Peer, NewCost)@admin:self,
    not del_link(Peer)@admin:self.
```

Rule (1) states that if a state transition is triggered by the receipt of an input tuple add_link, then a new state tuple direct_link is added in the new state with the arguments of the input tuple. Note that the identifier of the receiver node is self since the tuple is received by the node that executes the rules. Rule (2) states that if a state tuple direct_link exists at the previous state (i.e., the state before the transition) and none of the input tuples add_link and del_link with arguments matching that of the state tuple is received, then the same state tuple should persist at the new state.

The remaining rules (3)-(6) complete the D2C description, in router rtr1, of our distance vector routing protocol. Rules (3) and (4) define the state tuples neighbour_distance (which record the neighbour-to-destination costs) after a transition. The transport tuple dist_msg(Dest, Cost)@Peer:self in rule (3) denotes the receipt at the current node rtr1 of an inter-router message sent by the node Peer. Rule (5) defines the state tuple least_distance, which records the best cost from the current router to any reachable destination, based on the sum between the direct link cost to a neighbour and the neighbour-to-destination cost. This rule is an example where the conditions refer to only the new state. In addition, the head has, as argument, the *aggregation* function #min. Its meaning is that if $\{\langle \mathtt{d_1}, \mathtt{c_1^1}\rangle, \ldots, \langle \mathtt{d_1}, \mathtt{c_1^n}\rangle, \ldots, \langle \mathtt{d_k}, \mathtt{c_k^1}\rangle, \ldots, \langle \mathtt{d_k}, \mathtt{c_k^n}\rangle\}$ are all pairs of values Dest and TotalCost that satisfy the state tuples in the body of rule (5) at the new state, then the new state must contain the state tuples least_distance with arguments $\{\langle \mathtt{d_1}, \mathtt{c_1^{min}}\rangle, \ldots, \langle \mathtt{d_k}, \mathtt{c_k^{min}}\rangle\}$ where each $\mathtt{c_i^{min}}$ is the smallest value of $\mathtt{c_i^1}, \ldots, \mathtt{c_i^n}$. Finally, rule (6) describes when and where the transport tuple in the head should be created and sent to. It states that if the best cost from the current router (i.e., rtr1) to a destination has changed (i.e., some least_distance state tuple with arguments $\langle \mathtt{d}, \mathtt{c}\rangle$ exists at the new state but not at the previous state), then a *dist_msg* type message containing these argument pairs should be sent to all direct neighbours (i.e., retrieved from the state tuple direct_link) after the transition.

```
neighbour_distance(Peer, Dest, Cost) if                     (3)
    dist_msg(Dest, Cost)@Peer:self.
neighbour_distance(Dest, OldCost) if                        (4)
    prev neighbour_distance(Dest, OldCost),
    not dist_msg(Dest, AnyCost)@Peer:self.
least_distance(Dest, #min<TotalCost>) if                    (5)
    neighbour_distance(Peer, Dest, Cost1),
    direct_link(Peer, Cost2),
    TotalCost = Cost1 + Cost2.
dist_msg(Dest, Cost)@self:Peer if                           (6)
    least_distance(Dest, Cost),
    not prev least_distance(Dest, Cost),
    direct_link(Peer, LinkCost).
```

A state transition at a node is defined by a set of D2C rules (e.g., rules (1)-(6)) and computed through evaluation of these rules. The rule evaluation computes two sets, a NEW set of the local state tuples that will hold after the execution of the transition and a $SEND$ set of the transport tuples that will be send after the computation of the state transition. The computation of the set NEW depends recursively on the same set NEW, generated during the rule evaluation, and a set $PREV$, which includes the state tuples preceded by **prev** that hold locally just before the trigger of the rule evaluation, and the input/transport tuples that have triggered the evaluation.[1] State transitions are obtained by the computation of a *fixed-point operator*, similar to the well-known *fixed-point computation* in Datalog [12].

Let $\mathcal{T}$ be a set of tuples or tuples preceded by **prev**, let $\mathcal{R}$ be the set of declarative rules specifying state

1. Note that with **prev** one can encode past time linear temporal logic to refer to earlier states (see [11]).

transitions. The *immediate consequence operator* $T_{\mathcal{R}}$:

$$T_{\mathcal{R}}(\mathcal{T}) = \{H | H \textbf{ if } L_1, \ldots, L_k, \textbf{not } L_{k+1}, \ldots, \textbf{not } Ln \in \mathcal{R} \text{ and } \{L_i, \ldots, L_k\} \subseteq \mathcal{T} \wedge \{L_{k+1}, \ldots, Ln\} \cap \mathcal{T} = \emptyset\}$$

Let $PREV$ be a set of state tuples preceded by **prev** and the received input/transport tuples triggering the transition. Suppose the set of declarative rules $\mathcal{R}$ is *stratified* [13], the set NEW is computed as follows:

$$T_{\mathcal{R}} \uparrow^0 = PREV$$
$$T_{\mathcal{R}} \uparrow^{i+1} = T_{\mathcal{R}}(T_{\mathcal{R}} \uparrow^i) \cup T_{\mathcal{R}} \uparrow^i$$

and $NEW = T_{\mathcal{R}} \uparrow^n \setminus PREV$ when $T_{\mathcal{R}} \uparrow^n = T_{\mathcal{R}} \uparrow^{n-1}$, i.e. it reaches a fixed point.

2.2 The Execution Environment

The execution module of our framework consists of an infrastructure implemented in Java, called *Distributed State Machines* (DSM), that enables the development and execution of distributed applications [14] written in D2C. It has a three-layer architecture (see Figure 2). The

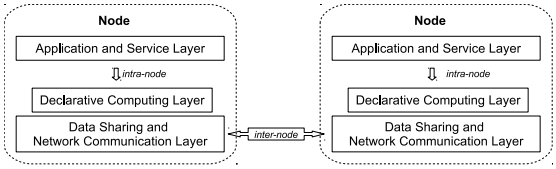


Fig. 2: Architecture of Execution Module

bottom layer, *Data Sharing and Network Communication*, is responsible for maintaining low level data representation and storage, and handling inter-node communications. The implementation is flexible enough to allow the integration of various communication mechanisms (e.g., Ethernet, Wifi), which will remain hidden from the higher layers. The middle layer, *Declarative Computing*, is where the execution of D2C programs takes place (i.e. the rule evaluation at each local node). The top layer, the *Application and Service*, is where applications/services that use results of distributed algorithms are running. For example an application that makes use of a distributed leader election algorithm that picks nodes for performing special tasks will be running in this layer.

During execution, a rule engine is running on each network node. Each engine takes as input the D2C rules that define the transition funtion of the local automaton, executes automaton and handles the (inter-node) automaton communication. Relational databases are the primary data structure representing an automaton's state. Each state tuple is a record in a table with name matching the tuple name, and its truth value is reflected by its presence in the table. Input tuples and transport tuples are represented as transient records (i.e., inserted when they occur, and dropped after been used) in corresponding tables.

In addition to the state, input and transport tuples, the DSM infrastructure provides two additional features to facilitate easier integration of the system with external computational environments. The first feature is the use of *system tuples*. These are special tuples that allow access from the external computational environment to the underlying layer of a DSM execution. They can be used to interact with the run-time physical topology of the network. For example, a `topology_change` system tuple can be used in a D2C rule description to detect events about changes in the network topology and define responses to such specific changes. Rule (7) at a local node `rtr1`, is an example of how system tuples could be used.

$$\texttt{dist\_msg}(\texttt{Dest}, \texttt{Cost})@\texttt{self}:\texttt{Peer} \textbf{ if } \texttt{topology\_change}(\texttt{Peer}, \text{“added”}). \quad (7)$$

The second additional feature of our DSM is the use of *action tuples*. These tuples provide a way for DSM to send commands to programs running at the Application and Service layer based to the local state, by calling predefined external methods. External method calls can be executed synchronously or asynchronous with respect to the rule evaluation of a local engine. In the first case, the rule evaluation at a local node is guaranteed to finish only after a called method has terminated, in the latter there is no ordering and rule evaluation, at a local node, may finish before or after a method returns. The following rule, for example, when evaluated makes a call to the external method `appendToFile` for each value `msg` in the received transport tuple `remoteMessage`. Note that the tuple name `my_actions_args` is a special tuple name that can be used to define the possible method calls, where the first argument is the execution type of the method (0 for synchronous and 1 otherwise), the second argument is the method name and the following arguments are the list of input parameters to the method.

$$\texttt{my\_actions\_args}(0, \texttt{appendToFile}, \text{“/tmp/removeLogs.txt”}, \texttt{Msg}) \textbf{ if } \texttt{remoteMessage}(\texttt{Msg})@\texttt{Src} : \texttt{self}. \quad (8)$$

2.3 The Simulation Environment

The simulation component of our framework builds upon the DSM and provides an environment for developers to experiment with distributed algorithms, written in D2C, over simulated networks with dynamic topologies. Its architecture (see Figure 3) includes a simula-

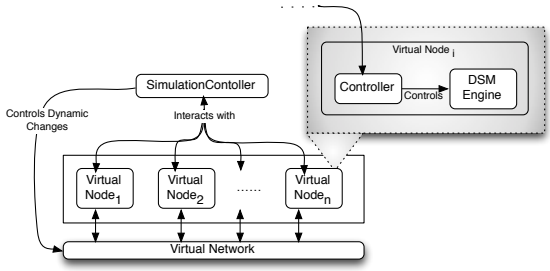


Fig. 3: Simulation Environment Architecture

tion controller node responsible for creating a virtual network, from a user-defined topology, controlling and

monitoring the local execution of the algorithms and visualising the execution traces. The local controller in each virtual node collects profiling and monitoring data from its local DSM instance, and interacts with the simulation controller, which can start, interrupt, terminate a simulation or inject network dynamics into the global virtual network. Any point in time of a detailed simulation history, gathered by the simulation controller, can be visualised. The visualization includes animation of message passing and local state tuples. In this way, protocols can be seen in action step by step and the state of the network can be inspected by the user. It can therefore be used also for debugging purposes.

Once a virtual topology is set and appropriately initialised (if required by the user) the simulation runs the D2C algorithm on the created virtual network. The execution will run until either the algorithm converges or a maximum number of *network-wide* state transitions, which have been pre-specified in the configuration of the simulation task, is reached. Convergence is inferred when no (local) transitions are executed for a predefined period of time. The maximum number of network-wide state transitions can also be set by the user, and computed during the execution using a Lamport transformation function [15]. Each virtual node has a local clock that advances by one each time a local transition takes place. The Lamport clock algorithm creates partial causal ordering between the virtual nodes' local states. These states are then topologically sorted into an ordered set, whose indexes is the global clock. Local states belonging to the same set are in the same global state.

3 The Analysis Component

The analysis component of our framework takes in input a D2C specification of a distributed algorithm and a query task that encodes the verification of a property of the algorithm. Given the declarative nature of the D2C language, any logic-based method could be used to perform this verification task. In our framework we make use of an Answer Set Programming (ASP) solver [16] in two different modes. We present first our pure ASP logic-based approach to analysis and demonstrate its effectiveness in analysing two different classes of routing protocols with respect to two different types of properties. ASP have been shown to be very effective in solving problems of high computational complexity [17], nevertheless, for properties that involved liveness conditions the search spaces are huge and we need to hybrid approach to the analysis which combines graph traversal with local ASP solver, improving the scalability of the analysis process. Both approaches make use of the same semantic model of our D2C language. This is defined declaratively as a logic program automatically generated from a given D2C specification.

3.1 Semantic Model of a D2C Specification

The declarative semantics of a D2C specification is a Datalog program extended with *location* and *time*, generated automatically from D2C rules as follows:

- for state tuples, the special variable *Self* is added as the first argument indicating the location where the tuple is evaluated;
- transport tuples that are in the head (resp. body) of a rule are mapped into a `send` (resp. `receive`) predicate that takes as first two arguments the `To` and `From`, respectively, node identifiers that appear in the transport tuple after the @ symbol, and as subsequent arguments, the transport tuple name with all its arguments. The first argument (i.e. `To` identifier) indicates the location where the transport tuple is supposed to be sent and evaluated;
- the mapping of each state and transport tuple introduces also an extra (time) variable (whose possible values are non-negative integers), as the last argument, indicating the temporal ordering of the tuples' truth:
  - if the tuple is preceded by **prev**, then `T−1` is added and **prev** is removed; otherwise, `T` is added.
  - `T` (resp. `T−1`) is added to each `send` (resp. `receive`) transport tuple.
- for each rule,
  - **if** is replaced with Datalog rule connector ":−";
  - the condition `time(T)` is added to the rule body.

For example, the semantics of rules (4) and (6) are

```
neighbour_distance(Self, Dest, OldCost, T) :−
  time(T), time(T−1),
  neighbour_distance(Self, Dest, OldCost, T−1),
  not receive(Self, Peer, neighbour_distance(Dest, AnyCost), T).
send(Peer, Self, neighbour_distance(Dest, Cost), T) :−
  time(T), time(T−1),
  least_distance(Self, Dest, Cost, T),
  not least_distance(Self, Dest, Cost, T−1),
  direct_link(Self, Peer, LinkCost, T).
```

3.2 Pure Logic-based Analysis

Our pure logic-based approach to analysis requires three main elements: the semantic model of the distributed algorithm – a logic program automatically translated from the given D2C specification as described above; a *communication model* – a set of logic programming rules describing the sending and receiving of transport tuples (e.g., synchronous or asynchronous, reliable and in-order delivery); and the semantic model of the given *query task*, which includes a logic programming formalisation of an initial network configurations (e.g., network topology, initial state of each node in the network) and the properties to be checked as logic queries. A query task is then defined as a logic query computation against the combined logic programs, performed using a state-of-the-art ASP solver called *Clingo* [18].

In what follows we describe two communication models – synchronous and asynchronous [19] – and D2C specification of network configurations.

3.2.1 Network Communication Model

In an execution environment, messages generated by a node's state transition are transferred to the destinations nodes over the network, through the communication protocol used by the execution environment (e.g. network communication layer of the DSM). However, the formal verification of a D2C specification needs the notion of connection between a sent and received message (i.e., the `send` and `receive` predicates) to be explicitly defined. The role of the network communication model is to formalise such connection in order to take into account possible nodes interactions during a verification task. Since each computational node has its own local clock, the main challenge in modelling communication is to correlate the time argument of the *send* and *receive* predicates of a transport tuple. We show in what follow how this is done and how both the *synchronous* and *asynchronous* execution models can be specified using the D2C language and two *virtual* nodes: the *execution round manager* and the *communication buffer manager*.

Synchronous Model

In this model, all nodes execute in rounds [19]. An example is illustrated in Figure 4(a). At the beginning of each round, nodes send out transport tuples, which are delivered at the recipient nodes immediately (i.e., in the same round). A state transition is performed at each node that receives at least one message. The resulting new outgoing transport tuples are sent in the next round. At each round if a node receives any message (i.e., performs a state transition), its local clock advances by one (e.g., node n_1) otherwise its clock remains the same (e.g., node n_3) and so does its state. Therefore, it is possible nodes have different local times at the same round.

To "synchronise" the sender's local time and the receiver's local time in the same round, we assume a virtual *execution round manager* node R that keeps track of each node's local clock among the rounds (Figure 4(b)). All outgoing transport tuples of computational nodes

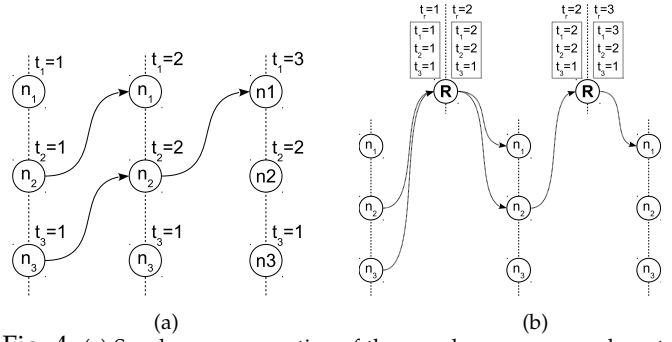


Fig. 4: (a) Synchronous execution of three nodes n_1, n_2, n_3, where t_i is the local clock of n_i. Vertical dashed lines separate execution rounds. Arcs represent inter-node communications. (b) Synchronous execution through an execution round manager (denoted R) with local clock t_r.

are "routed" through the execution round manager node R. By "observing" the network traffic, R can infer and record the local clock values of each node. The behaviour of R (referred as `round` from now on) can be specified using the following set of D2C rules, where the special tuples (and predicates) $node\_send$ and $node\_receive$ denote, respectively, the sending and the receiving of a transport tuple by computational nodes. The tuple $node\_send$ can also be considered as an input tuple to `round`'s automaton, which is automatically generated. The semantic encoding of these rules as Datalog program (see below) uses a sort T to denote the local clock of R, which essentially captures the execution-round number of the synchronous execution.

$$node\_receive(\texttt{Dest}, \texttt{Src}, \texttt{Msg}, \texttt{Td}) \textbf{ if} \quad (9)$$
$$\textbf{prev } node\_send(\texttt{Dest}, \texttt{Src}, \texttt{Msg}, \texttt{Ts}),$$
$$\textbf{prev } \texttt{vclock}(\texttt{Dest}, \texttt{Td}).$$
$$\texttt{clock\_advanced}(\texttt{Dest}) \textbf{ if} \quad (10)$$
$$node\_receive(\texttt{Dest}, \texttt{Src}, \texttt{Msg}, \texttt{Ts}).$$
$$\texttt{vclock}(\texttt{Dest}, \texttt{Td}+1) \textbf{ if} \quad (11)$$
$$\textbf{prev } \texttt{vclock}(\texttt{Dest}, \texttt{Td}),$$
$$\texttt{clock\_advanced}(\texttt{Dest}).$$
$$\texttt{vclock}(\texttt{Dest}, \texttt{Td}) \textbf{ if} \quad (12)$$
$$\textbf{prev } \texttt{vclock}(\texttt{Dest}, \texttt{Td}),$$
$$\textbf{not } \texttt{clock\_advanced}(\texttt{Dest}).$$

Rule (9) states that every message is sent and received in the same round. If in some round a node, $Dest$, receives a transport tuple, then its local clock must be advanced in the current round. This is described by rule (10). Therefore, at the end of a round its local clock, recorded by the manager, is either incremented (11) or stays the same (12), depending on whether the node receives any message.

```
node_receive(round, Dest, Src, Msg, Td, T) :-
    time(T), time(T−1),
    node_send(round, Dest, Src, Msg, Ts, T−1),
    vclock(round, Dest, Td, T−1).
clock_advanced(round, Dest, T) :-
    time(T),
    node_receive(round, Dest, Src, Msg, Ts, T).
vclock(round, Dest, Td+1, T) :-
    time(T), time(T−1),
    vclock(round, Dest, Td, T−1),
    clock_advanced(round, Dest, T).
vclock(round, Dest, Td, T) :-
    time(T), time(T−1),
    vclock(round, Dest, Td, T−1),
    not clock_advanced(round, Dest, T).
```

The semantic model of the synchronous execution will also automatically include rules (13) and (14) below to link the special predicates $node\_receive$ and $node\_send$ of R with the transport predicates $receive$ and $send$ of the computational nodes. The body of rule (13), the sender Src and its local sent time Ts are used to recover the round number (i.e., the local clock of R). Note that the last negative body literal is necessary to avoid repeatedly

and incorrectly generating *node_send* when the local clock of Src is not advancing (i.e., Src is idle).

```
node_send(round, Dest, Src, Msg, Ts, T) :-                    (13)
    time(T),
    send(Dest, Src, Msg, Ts),
    vclock(round, Src, Ts, T),
    not vclock(round, Src, Ts, T−1).
receive(Dest, Src, Msg, Td) :-                                (14)
    time(T),
    node_receive(round, Dest, Src, Msg, Td, T).
```

Asynchronous Model

The main difference between the asynchronous model (an example is given in Figure 5) and the synchronous model is that a message sent may be received after an arbitrary number of rounds (e.g., the communication between n_3 and n_2).

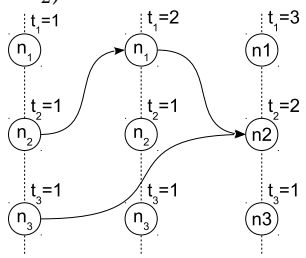


Fig. 5: Asynchronous execution of three nodes n_1, n_2, n_3.

To specify the asynchronous model, another virtual node – the *communication buffer manager* `buffer` – is used in addition to the execution round manager. The D2C rules for `buffer`'s automaton are as follows.

```
queue_elem(Src, Dest, Msg, Pos) if                            (15)
    not dequeued(Src, Dest),
    prev queue_elem(Src, Dest, Msg, Pos).
queue_elem(Src, Dest, Msg, Pos−1) if                          (16)
    dequeued(Src, Dest),
    prev queue_elem(Src, Dest, Msg, Pos),
    Pos > 0.
queue_elem(Src, Dest, Msg, NextPos) if                        (17)
    next_queue_position(Src, Dest, NextPos),
    queue_capacity(Src, Dest, K),
    NextPos < K,
    msg(Dest, Src, Msg)@Src.
prev_queue_size(Src, Dest, #count<Pos>) if                    (18)
    prev queue_elem(Src, Dest, Msg, Pos).
next_queue_position(Src, Dest, Size) if                       (19)
    not dequeued(Src, Dest),
    prev_queue_size(Src, Dest, Size).
next_queue_position(Src, Dest, Size−1) if                     (20)
    dequeued(Src, Dest),
    prev_queue_size(Src, Dest, Size).
activated(#choice<Dest>) if                                   (21)
    prev queue_elem(Src, Dest, Msg, 0).
dequeued(#choice<Src>, Dest) if                               (22)
    activated(Dest),
    prev queue_elem(Src, Dest, Msg, 0).
msg(Dest, Src, Msg)@Dest if                                   (23)
    dequeued(Src, Dest),
    prev queue_elem(Src, Dest, Msg, 0).
heartbeat@buffer if                                           (24)
    queue_elem(Src, Dest, Msg, 0).
```

It is assumed that there is one message buffer queue for each pair of sender and receiver. Rule (15)–(17) specify the effect of the dequeue and enqueue operations and Rule (18)–(20) specify the size of the queue and the next available position for new message. In the default asynchronous execution, we assume that only one non-empty buffer queue can be dequeued at a time, which is captured by Rules (21)–(23). This effectively restricts that each node can receive at most one message at a time. If such restriction needs to be relaxed in the analysis task, the user can customise Rule (22) by removing the use of the *choice* aggregation function for the Src argument. Similarly, the situations of message lost or delivered out-of-order can be specified by modifying Rules (21)–(23). Finally, Rule (24) is a way to trigger a dequeue process by `buffer` in the next round if there is no outgoing transport tuples from the computational nodes but there is still non-empty queue.

Figure 6 illustrates the use of the two managers for asynchronous execution modelling: the transport tuples are re-directed to and from the communication buffer manager at consecutive rounds. Finally, Datalog rule (13) is replaced by (25)–(27) to model the transport tuples redirection (to `buffer`) and routing (through `round`):

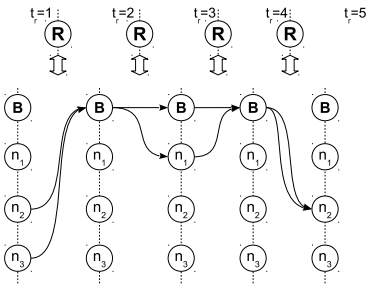


Fig. 6: Interactions between the execution round manager and the communication buffer manager.

```
node_send(round, buffer, Src, msg(Dest, Src, Msg), Ts, T) :-   (25)
    time(T), send(Dest, Src, Msg, Ts),
    Src ≠ buffer, Dest ≠ buffer,
    vclock(round, Src, Ts, T), not vclock(round, Src, Ts, T−1).
```

```
node_send(round, Dest, Src, Msg, Ts, T) :-                    (26)
    time(T),
    send(Dest, buffer, msg(Dest, Src, Msg), Ts),
    Dest ≠ buffer,
    vclock(round, buffer, Ts, T),
    not vclock(round, Src, Ts, T−1).
node_send(round, buffer, buffer, heartbeat, Ts, T) :-         (27)
    time(T),
    send(buffer, buffer, heartbeat, Ts),
    vclock(round, buffer, Ts, T),
    not vclock(round, Src, Ts, T−1).
```

3.2.2 Analysis Task

Our analysis task includes two parts. The first part specifies the network configuration. For example, in the routing protocols domain, to analyze the configuration of Figure 7, the network topology will need to be specified (e.g., `node(1, 0)`, `neighbour(1, 0, 0)`, `neighbour(1, 2, 0)`). The last argument in all these four tuples represents the time where the tuples became true; these tuples will persist until an explicit removal is made. New tuple instances about the topology can be added at different times (e.g. a new node joins the network during execution) as described in Section 2.1. We consider in this paper only analysis tasks that are performed on static networks, so only configuration tuples with time 0 are included. Other properties about the topology that are protocol specific, can also be specified. For instance, in the case of the BGP protocol, network policies can be specified: e.g., the preference at node 3 of paths 3420 and 30 can be expressed through the following tuples:

```
manual_local_preference(3, 0, 100, 0)
manual_local_preference(3, 420, 200, 0)
```

where 200 and 100 are local preference values (the higher, the more preferred) and as explained above, the last argument, 0, is the starting time where these tuples are considered true. The second part of the analysis task is the specification of the query schemas that capture the specific properties of the distributed algorithm that we want to analyse. As part of this specification is the description of the notion of time. Each extended tuple in the Datalog rules translated from the protocol model and network communication model has a time argument. Recall from Section 3.1 that a condition `time(T)` is added to the body of Datalog rules in order to give a fixed domain to these time arguments. This condition is defined in Datalog as `time(0..k)`, which gives the domain of time as $[0, k]$ where `k` is a positive integer that can be set by the analyst using **maxtime** `k`.

4 Various Applications

In this section we present two types of applications. The first type is the analysis of two classes of routing protocols. The second is a running system in the domain of multi-robot pattern formation.

4.1 Analysis of Routing Protocols

Since the early ARPANET, deployed routing protocols have been discovered to cause severe routing anomalies [20]. Even BGP, the current *de facto* inter-domain routing protocol, has been shown to be vulnerable to persistent route oscillations [9]. There is clearly a need for formal analysis techniques in order to verify properties of routing protocols, and configurations. We illustrate how our framework allows users to specify different properties, and verify them on the actual D2C description. Such analysis can be particularly useful to researchers when designing new routing protocols, to software engineers for verifying the implementation, and to operators for validating configurations. We consider two properties: convergence and disjoint paths.

Convergence

A routing protocol converges when it reaches a state where every node stops updating its routing or forwarding table. In the absence of convergence, nodes may continuously change state impacting the stability of the overall routing system, and potentially affecting the delivery of traffic to the intended destinations. As such, convergence is one of the most important properties, and every routing protocol should ideally always converge. However, despite its importance, BGP, a path-vector type of routing protocol, has been shown to possibly violate this property: a set of BGP policies (i.e. routing preferences set in each node) can result in persistent route oscillations. Griffin *et al.* [9] have identified sufficient conditions – the lack of dispute wheels – to guarantee convergence in BGP. Yet in the presence of dispute wheels, a protocol may diverge. Griffin *et al.* have further demonstrated that given a set of BGP policies, determining whether a stable state exists is NP-complete. Given that the existence of a stable state does not imply that the network always converges, verifying whether a set of BGP policies – that includes a dispute wheel – always converges is arguably a more difficult problem.

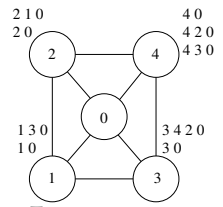


Fig. 7: BGP configuration.

We have specified in D2C a simplified version of BGP [2]. The BGP router nodes communicate via UPDATE messages, each of which may either advertise or withdraw a path to a destination by the sender. The message recipient updates its locally stored paths to the destination accordingly, and select a best path based on a pre-defined set of policies (e.g., path lengths, preference of next hop).

2. Available in the appendix of this paper

If the selected best path has changed, a new UPDATE message is sent to the neighbours. Figure 7 depicts a BGP configuration, where node 0 is the destination, and paths next to a node indicate path selection policy of that node (node 2 prefers path $2 \rightarrow 1 \rightarrow 0$ to path $2 \rightarrow 0$).

Given a set of policies, our analysis framework can determine whether the network always converges. For routing protocols such as BGP where routers generate new messages only in response to received messages or topology changes (which generate the initial messages), we know that the protocols converge when all the communication links become empty. Consequently, to prove the convergence property, it suffices to find a round in which all the communication links become empty (i.e., no node receives any message). To verify it, we incrementally increase a parameter, *maximum round*, which specifies the maximum local clock of the execution round manager, and check if in the final global state there is any received message. The parameter is initially set to a given non-negative integer. A good initial value would be the minimum network rounds in which the routing protocol is expected to converge in the best case scenario. If this value cannot be determined then it would be initially set to 1. During the analysis task it will be incremented by 1 each time the convergence query is called, until either the query fails or a limit is reached. Note that the initial value of the parameter does not affect the correctness of the verification result. To check convergence the following query is used:

```
divergent :-                                                    (28)
    max_round(MaxTime),
    node_receive(round, Dest, Src, Msg, Td, MaxTime).
:- not divergent.                                               (29)
```

Given the above query, the ASP solver searches for traces that satisfies the Integrity Constraint (29), i.e., traces of messages sequence which present received message(s) at *maximum round*. If successful, the communication sequence provides a divergent execution trace. If the solver returns empty, all message sequences result in empty communication links after *maximum round*, and we can conclude that the network always converges. We have tested a network configuration proposed in [21] (see Figure 7). Starting at *maximum round* = 11, the ASP solver returns empty indicating that all message sequences converge after 11 rounds.

Node-disjoint paths

A third class of analysis that we have considered regards the discovery of disjoint paths by routing protocols for mobile ad-hoc networks (MANET). Paths are node-disjoint if, besides the source and destination nodes, they share no common nodes. Finding node-disjoint paths is particularly important in MANETs for network throughput optimization, or for fast restoration after a network failure: given a pair of source and destination nodes, two node-disjoint paths can be used as an active and a backup paths. As soon as the active path fails, traffic is redirected to the backup path. Zhang *et al.* have proposed in [22] a routing protocol to establish node-disjoint paths in MANET. Their protocol is proved to be node-disjoint, namely to compute only paths that are node-disjoint. We have implemented a simplified version of this routing protocol and used our analysis framework to verify whether given any topology with node-disjoint paths, the protocol can (i) find at least two of them and (ii) does not compute paths that are not node-disjoint.

Similar to the forwarding loops analysis, the discovery of node-disjoint paths can be performed through the transitivity closure of the forwarding tables after the protocol converges. To prove whether, for any given topology and source and destination nodes, the protocol can discover at least two node-disjoint paths if they exist, we check whether it is possible to discover some topology in which node-disjoint paths exist but the protocol is not able to identify them. This is captured by the following two analysis queries:

```
:- not has_disjoint_paths                                       (30)
:- discover_disjoint_paths.                                     (31)
```

where the predicates `has_disjoint_paths` and `discover_disjoint_paths` compute respectively the physical paths in the topology and the paths discovered by the protocol that are node-disjoint (note that a discovered path must also be a physical path). Solutions to these queries would therefore give communication sequence where no node-disjoint paths are discovered by the protocol even though they exist. The predicate `discovered_path` computes paths between the given source and destination nodes that are discovered by the protocol, using the same definition of `path` as that described in the forwarding loop analysis. The built-in predicate `share_intermediate_node(Path1, Path2)` checks whether two physical paths, `Path1` and `Path2`, constructed through the transitive closure over the physical links in the topology, are node-disjoint. Formally, these predicates are defined as follows:

```
has_disjoint_paths :-                                           (32)
    source(From), destination(To),
    physical_path(From, To, Path1),
    physical_path(From, To, Path2),
    Path1!=Path2,
    not share_intermediate_node(Path1, Path2).
discover_disjoint_paths :-                                      (33)
    discovered_path(Path1), discovered_path(Path2),
    Path1!=Path2,
    not share_intermediate_node(Path1, Path2).
```

In summary, to answer queries (30) and (31) the ASP solver searches for traces where the protocol fails to discover disjoint paths while the actual topology contains at least two. If the solver returns empty, then we can conclude that the protocol verifies the node-disjoint property for the given topology. If not, the solver provides a counter-example.

To perform this analysis we generated all possible bidirectional connected graphs, for the given number of nodes, remove isomorphic graphs, and considered every possible pair of source and destination nodes [3]. In so doing we considered all topologies that could be generated from a given number of nodes, and possible pairs of source and destination nodes. We jointed each of them together with the simplified implementation of the protocol in [22] in turn and checked queries (31) and (32). We then were able to discover topologies where the protocol fails to discover node-disjoint paths. In particular, the ASP solver has provided a communication sequence, for the topology depicted in Figure 8 with node 1 being the source and node 6 being the destination, in which the two (physical) node-disjoint paths 1–4–3–6 and 1–2–5–6 are not detected by the protocol.

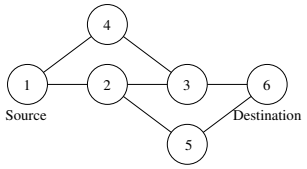


Fig. 8: Topology with Disjoint Paths

Furthermore, the discovered paths 1–2–3–6 and 1–2–5–6 (extracted from the analysis result) are not node-disjoint. Hence, we have been able to prove that the protocol fails to discover disjoint paths despite the presence of physical disjoint paths and that the discovered paths may not be node-disjoint, violating Theorem 1 in [22].

4.1.1 Scaling up the analysis task

Although our analysis framework is flexible enough to verify different protocol properties, for analysis tasks of general properties that involve reasoning over a set of global states or reasoning over a set of execution traces (as it is, for instance, the case for the convergence property), the scalability of our pure logic-based approach is bound by a computational bottleneck limitation of the ASP solver. Analysis tasks of this type include for instance finding oscillation during protocol execution, or answering whether a protocol can *always* converge. This is because: (1) the global state at a given time in an execution trace is represented by a set of tuples with their respective logical time, and comparing repetition of global states in ASP is expensive (in terms of speed); (2) in order to answer *always* type of queries, all possible logic models have to be computed in advance, some of which may differ from only a small set of tuples (i.e., the memory requirement is big). Furthermore, only *finite* logic models can be computed and although the number of global states is finite, a logical model might not be since the number of rounds captured by a logical model can be infinite (repeating global states). Hence, the parameter *maximum round* was used to make sure of having only finite logical models.

3. Isomorphic topologies and the cases where the source and destination nodes have direct link are skipped.

To increase the scalability of the analysis process, we have developed an hybrid approach for analysis. Inspired by model checking [23], the analysis task is turned into a (*global state transition*) graph traversal problem. First, each communication link between any pair of nodes is modelled as a FIFO message buffer queue possibly with a fixed capacity. A *local state of a queue* is represented by a set of tuples $\texttt{queue\_elem}(\texttt{Src}, \texttt{Dest}, \texttt{Msg}, \texttt{Pos})$, where `Msg` is a transport tuple, `Src` and `Dest` are the location identifiers of the sender and receiver, respectively, and `Pos` indicates the position (starting from 0) in the queue where transport tuple `Msg` is stored. Thus, $\texttt{From}-\texttt{To}$ can uniquely identify a queue. Note that this is in effect the local state of the communication buffer manager. Secondly, a *local state of a computational node* is presented by a set of state tuples of that node. Thus, a *global state* consists of all the local states of the nodes and the links. Thirdly, given a global state, a successor state is generated as follows: for each non-empty link (i.e., has `queue_elem` tuple) (1) the link is dequeued (i.e., remove the `queue_elem` with $\texttt{Pos} = 0$); (2) the new local state of the dequeued message receiver is computed and replaces the old local state in the global state; (3) the newly generated transport tuples are enqueued into the corresponding links (as new $queue\_elem$ tuples).

Finally, given an initial global state, a global transition graph can be constructed iteratively until no more new global states are generated. If the set of possible state tuples and transport tuples is finite (which is often the case in real application), the transition graph will be finite and the construction process will terminate. In the rest of this section, we show how convergence queries can be answered by analysing the transition graph. We will focus on the BGP protocol, for which a number of analysis techniques have been proposed. This allows us to compare our approach with existing work[4].

Convergence Analysis for BGP

Various techniques have been proposed for identifying and checking the sufficient condition for the convergence of routing protocols (see, for example, [24] and [10] and the references therein). However, these approaches analyse an abstracted model of the protocol instead of the actual implementation, and only work when sufficient conditions have been identified, e.g., the absence of dispute wheels guarantees BGP convergence, but in the presence of dispute wheels, BGP may sometimes converge, always converge, or always diverge. Furthermore, these approaches do not reveal transient oscillations of states during the execution of a converging protocol which is important for the stability of the network during dynamic changes nor give example traces when a protocol diverges. Therefore, they cannot answer questions such as time to converge or provide hints for protocol correction. Our hybrid analysis approach can address these issues and answer the following queries:

- *Does the protocol sometimes converge?* A global state

4. Note that the analysis approach is independent of the protocol.

with all the links empty does not have any successor. Thus, it is called a *terminal* or *convergent* state. The analysis reports yes if and only if there exists some convergent state in the global transition graph. An convergent (execution) trace is a path from the initial state to the convergent state. The longest (shortest) convergent trace gives maximum (minimum) time to converge.

- *Can there be oscillations during execution?* This is equivalent to the question of whether the graph contains cycles, which can be solved by cycle detection algorithm for directed graphs.
- *Does the protocol sometimes oscillate permanently?* There are two types of cycles: *transient* and *persistent*. In a transient cycle, every state has a path to at least one convergent state. With some *fairness* [25] assumption, the protocol execution can eventually get out of the cycle and reach some convergent state. Such cycles represent temporary oscillations and can be extracted. In a persistent cycle, no state has path to any convergent state. This implies that once the protocol execution enters such cycle it can never come out no matter what. Therefore, such cycles represent permanent oscillations, which can also be extracted. In fact, it is easy to show that a non-terminal state without path to any terminal state must belong to a persistent cycle. We call such non-terminal state a *divergent* state.
- *Does the protocol always/never converge?* The protocol always converges under fairness assumption if the transition graph is devoid of persistent cycles, and it never converges if the global transition graph is devoid of convergent state.

We have implemented this approach and have tested it with different BGP configurations. Table 1 shows the results of testing network configurations with a single dispute wheel of various sizes. Column 1 represents the size of the dispute wheel (including the origin node). Column 2 shows the total number of global states, convergent states and permanently divergent states. Columns 3 and 5 give the total transitions and time taken to construct the whole transition system [5] and to compute the answers to all of the aforementioned convergence queries. Column 4 are the results of the tests: *sometimes* means it has both convergent states and permanently divergent states, *never* means it has no convergent state, and *always* means it has no permanently divergent state. We note that the time to build the entire global transition system may seem large, but to answer questions such as whether a configuration can converge, a trace would typically be produced in a much quicker time. In other words, the indicated time is an upper bound.

5. Note we could use query specific graph traversal algorithms to avoid constructing full graphs and hence reduce the analysis time significantly. For example, a depth-first search for *sometimes converge* query can stop as soon as a convergent state is found. For finding persistent oscillation, an algorithm for computing strongly connected components can be used.

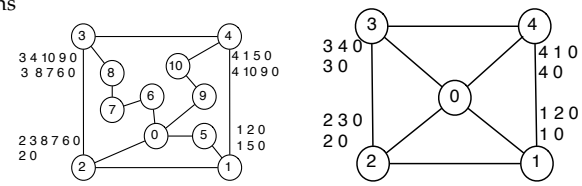
Nodes	States	Transitions	Converges	Time
3	17	22	Sometimes	0.079s
4	673	2007	Never	3.173s
5	5207	20356	Sometimes	25.784s
6	39537	192445	Never	4m
7	306749	1787535	Sometimes	102m
8	2410160	16379733	Never	4503m

TABLE 1: Convergence Analysis on BGP with Dispute Wheel: total time includes both global transition graph construction time and traversal time

Table 2 shows the results for the surprise gadget [26], and other randomly generated topologies (Figure 9) with a dispute wheel.

Conf. Desc.	Nodes	Converges	Time
Surprise gadget	5	Always	123m
Topology A	11	Sometimes	503m
Topology B	5	Sometimes	29.275s

TABLE 2: Convergence Analysis on BGP with Different Configurations



(a) Topology A (b) Topology B

Fig. 9: BGP Gadgets

Applying the reduction proposed by Wang et al. [27], we could analyse even larger networks. The analysis time in fact mainly grows with the size of the dispute wheels. Hence, our hybrid approach allows the analysis of BGP over topologies with dispute wheels of up to 8 nodes, size that is twice larger than those considered by existing work. In [27], BGP instances are instead reduced to "bad gadgets" dispute wheels of up to 4 nodes only. Once the pre-step of reduction is applied, the analysis in [27] uses a rewriting logic approach [28], which can only work for small networks and may identify false positives. Protocols written in previous versions of DN languages could only be analysed by translating into "logic" (e.g. [29], [8]). But no proof of the correctness of the translation has ever been given.

4.2 Pattern Formation in Multi-Robot Systems

Advances in robotics have witness the deployment of multiple robots that can perform team tasks. One of the challenges within this context is the pattern formation problem, that is how a group of robots can collaboratively achieve a desired formation and maintain it. Application areas of pattern formation include search and rescue operations, land mine removal, and unmanned aerial vehicles (UAVs). In a decentralized environment, robots are required to create and maintain formation using just local communication and sensor data. Our framework is particularly suited for distributed management and control. It provides the networking infrastructure needed to execute distributed algorithms where (sensor) data and computation are distributed among

several devices and devices can exchange data and share computational results of the group.

We have substantiated this claim by applying our framework to a specific example of pattern formation: decentralised control for multiple robots to align themselves uniformly between two fixed points. This alignment problem is defined as follows: a set of robots are assumed to be placed in a straight line between two fixed walls but with unequal distances between them. Using only their light sensor data and their wifi they have to cooperatively accomplish the task of positioning themselves uniformly along the line (i.e. the distance between any two robots is approximately the same). The algorithm starts with each robot sending its location to everyone. When a robot receives the location of both its neighbours, it calculates the middle position between the two neighbours, it moves there and sends its new location to its neighbours. The algorithm continues until no new message has been exchanged. At that point the algorithm should converge to a state where all robots are uniformly distributed along the line. We have developed a D2C implementation of such algorithm, where all robots are assumed to be "equal", i.e. they run the same D2C set of rules (hence the same algorithm) and their computation is based only on their sensor data and communication received.

We have used Lego MINDSTORMS NXT Robots. Each robot includes a NXT Intelligent Brick, two Servo Motors with built-in rotation sensor to measure speed and distance, two Ultrasonic Sensors that enable the robot to detect an object and measure its proximity. Ultrasonic sensors are used for discovering robot's neighbours. Each robot doesn't know in advance its neighbours but it detects them around it and finds out their location, which is then used in its computation. In addition, the decision of what move to make is done by the execution of D2C rules. The D2C rules make use of action tuples to determine specific method calls and relative parameters to be executed by the NXT Intelligent Brick.

A demo video of the robots execution is available online as supplemental material to this paper.

5 Discussion and Related Work

The declarative networking programming model was introduced in [30]. Several implementations of languages based on this concept have been developed (e.g. [2]). These implementations, however, accommodate typical network changes (e.g. links coming up or down, communication delays, etc.) outside the logic into their operational semantics. To address this limitation, Alvaro et al [4] proposed a new Declarative Networking language, called Dedalus, where all predicates are extended with an extra argument that represents *time*. With this argument, notions of state and state changes can be captured but because of the way time was modelled, the semantics of the Dedalus allows for time paradox: predicates, sent from one node to another, can arrive at the receiving node as if the predicate had been sent from the future. Our initially proposed language [31] also uses a time argument but avoids the limitations and paradox of Dedalus by not relying on a global clock but separating the clocks of each process whilst allowing a semantics expressed in term of Datalog + time. The D2C language proposed in this paper expands and generalises our initial proposal [31] by allowing declarative networking rules, describing state transition, to refer in their conditions to both previous and new state without the need of creating auxiliary predicates and introducing additional rules for generating intermediate states. The distinction between transient predicates and permanent predicates introduced in our initial proposal has been dropped in our new D2C language. Consequently, the body of our D2C rules can now refer to both the previous state and the new state and do not need to creating auxiliary (transient) predicates and introducing rules for generating them. Therefore the description of distributed algorithms in our proposed D2C language is more compact than that in our initial language; and the computational model of state transitions much simpler. In our previous DN framework, the computation of a state transition was generating two sets, ADD and DEL, of state tuples and a set SEND of transport tuples from only existing state tuples and tuples received. Let PREV and NEW be the set of state tuples stored by the automaton before and after the state transition, respectively, then NEW = (PREV $\setminus$ DEL)$\cup$ADD. At the end of a state transition, tuples in SEND are placed in the outgoing communication channel by the automaton together with the destination address of each tuple. In practice, temporary tuples may have been derived to support the computation of ADD, DEL and SEND. In our new proposed language, the computation of a new state has been simplified: NEW no longer depends on ADD and DEL, but recursively on NEW and PREV. As a result, the semantics of the new declarative rules becomes very close to a well-known fixed-point semantics for logic programs and has allowed us to define the evaluation of D2C rules using a form of fixed-point operator eliminating the need to use constraints to define the semantics (see Eq. (4) in [6]).

Formal analysis techniques for network routing protocols can be broadly classified into three groups: those based on model checking (e.g. [32], [33], [34])), which normally verify properties by exhaustive search over all possible executions; those based on theorem proving (e.g. [35]), which instead make use of formal specifications of routing protocols and prove their correctness using mathematical proofs for arbitrary network topologies; and the algebraic approaches (e.g. [36], [37]), which provide algebraic meta-theories of routing protocols and capture properties as algebraic constraints to be verified.

Similarly to existing model checking approaches, (e.g. [32], [33], [34]), both our approaches for convergence analysis are based on checking a state transition graph. But, whereas in existing model-checking approaches a state is represented as a fixed vector of variables and can only be computed based on a derived abstract

specification of the protocol (e.g. `Promela` model of a BGP protocol [32]), in our approaches a state in the transition graph is represented as a set of tuples that can be computed directly from the protocol implementation. We do not need to express the protocol implementation into an abstract specification. In principle, our transition graph can be translated into a model checking transition graph to be used by a model checking tool, but this could lead to state explosion (e.g., if our state is a subset of N possible tuples, then a corresponding model checking state will have 2^N variables). Furthermore, contrary to existing model checking approaches, where a convergence property is expressed in a formal language (e.g., LTL), and a uniform search algorithm is used for traversing the transition graph, in our approach, different convergence properties are turned into different corresponding graph analysis tasks, and task specific algorithms are adopted to efficiently compute the answers. This allows larger network to be checked.

In theorem prover approaches a verification proof is constructed by repeatedly applying inference rules to axioms or to previously proved theorems. But, designing the proof strategy is the responsibility of the user. They require and rely on human involvement to guide the search for proofs (e.g., how does one identify the right lemmas and rewrite rules to use?). As a result, *expertise* and *user intervention* are typically cited as theorem provers' main limitations. In [35], proofs are constructed using an hybrid of SPIN model checker and HOL theorem prover. The HOL is first used to prove abstractions, then, SPIN is applied on the abstracted system. Our approach is more general as it enables the analysis of properties in specific configurations for which sufficient conditions, or ways to proof the desired properties, have not yet been established. For example, we have showed how we can analyze the convergence of a BGP configuration which includes a dispute wheel. While the absence of dispute wheels has been proven to be a sufficient condition for correctness, BGP configurations which include a dispute wheel may always converge, sometimes converge, or always diverge. Our approach allows users to analyze such BGP configurations.

Algrebraic structures are also among alternative approaches for modelling and reasoning about routing protocols [38], [39], [40]. More recently, Sobrinho has introduced a new routing algebra framework [36], [37] to address rich policy control of contemporary routing protocols (e.g., BGP). The abstractions of these methods have allowed researchers to identify fundamental properties to achieve convergence, loop freedom, and path optimality for distance and link state routing protocols [36], [37]. Although these sufficient conditions have been proven to be very useful to guide the design of new routing mechanisms, or fix existing routing protocols (e.g., [41], [42]), this abstract approach may disregard seemingly innocuous details of a routing protocol implementation that could ultimately violate desirable properties. Moreover, representing complex routing mechanisms in algebraic terms can be challenging. Our framework is complementary to the routing algebraic framework as it allows users to specify and verify properties of routing protocols within the same representation formalism.

As for verification of DN protocols, approaches based on theorem-proving include DNV [8] and FVN [43]. DNV, in particular, automatically translates NDLog specifications into axiomatic schemas that are used by a semi-automated theorem prover. The analysis requires input from the user on selection of proof strategies. Our analysis is, instead, fully automated as it does not relay upon a general purpose theorem prover, but on a solver that is specific to set of extensions of Datalog sufficient for our system. All heuristics are already embedded in the solver implementation. The FSR toolkit [44] focuses on routing algebra, from which declarative networking programs can be automatically generated, and analyses them for strict monotonicity property. This property is, however, only sufficient but not necessary condition for proving convergence. Our analysis tool can be directly applied to our declarative executable protocol specifications and to configurations for which sufficient conditions have not yet been identified. For example, we can analyze the convergence properties of a BGP configuration with a dispute wheel, or the discovery of disjoint paths by a MANET protocol.

Rewriting logic have also been used [29] for simulating and performing state exploration of BGP instances to check for route oscillation. Analysis can be applied to topologies with up to 9 nodes, but they cannot distinguish between temporary and permanent oscillations. Our approach is more general. By integrating Answer Set Solving and graph algorithms, it scales the *always converge* analyses for BGP to network topologies that include dispute wheels that have twice the size of those considered by other existing analysis approaches. It does not generate false positive answers, and can distinguish between transient and permanent routing anomalies, analysis not supported in previous work [27], [45].

6 Conclusion and Future Work

We have presented our declarative framework for specification, simulation and analysis of distributed applications. We have focused in particular on the analysis features of our framework assuming a fixed topology. In our future work we aim to extend our analysis component to support protocol analysis with dynamic topology changes.

There are surprisingly many practical distributed algorithms that can be expressed by our DN computational model, and hence target for our framework. We also plan to use our approach in two emerging areas of network management: Software Defined Networks [46] and Named Data Networking [47].

References

[1] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking," *Commun. ACM*, vol. 52, no. 11, pp. 87–95, 2009.

[2] ——, "Declarative networking: language, execution and optimization," in *SIGMOD Conference*, 2006, pp. 97–108.

[3] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking," *CACM*, vol. 52, no. 11, pp. 87–95, 2009.

[4] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears, "Dedalus: Datalog in time and space," in *Datalog, LNCS 6702*, 2011, pp. 262–281.

[5] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 75–90.

[6] J. Ma, F. Le, D. Wood, A. Russo, and J. Lobo, "A declarative approach to distributed computing: Specification, execution and analysis," *TPLP*, vol. 13, no. 4-5, pp. 815–830, 2013.

[7] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu, "Formally verifiable networking," 2009.

[8] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky, "Declarative network verification," in *PADL, LNCS 5418*, 2009, pp. 61–75.

[9] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," in *IEEE/ACM Trans. on Networking*, 2002.

[10] T. Griffin, F. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *IEEE Trans. on Networking 10*, pp. 232–243, 2002.

[11] J. Lobo, J. Ma, A. Russo, E. Lupu, S. Calo, and M. Sloman, "Refinement of history-based policies," in *Logic programming, knowledge representation, and nonmonotonic reasoning*. Springer, 2011, pp. 280–299.

[12] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[13] K. R. Apt and R. Bol, "Logic programming and negation: A survey," *Journal of Logic Programming*, vol. 19, pp. 9–71, 1994.

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

[15] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[16] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider, "Potassco: The Potsdam answer set solving collection," *aicom*, vol. 24, no. 2, pp. 105–124, 2011.

[17] G. Brewka, T. Eiter, and M. Truszczynski, "Answer set programming at a glance," *Commun. ACM*, vol. 54, no. 12, pp. 92–103, 2011.

[18] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "*Clingo* = ASP + control: Preliminary report," in *Technical Communications of the Twenty-ninth International Conference on Logic Programming (ICLP'14)*, vol. arXiv:1405.3694v1.

[19] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[20] A. Khanna and J. Zinky, "The revised arpanet routing metric," in *ACM SIGCOMM*, 1989.

[21] T. G. Griffin and G. Wilfong, "An analysis of bgp convergence properties," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, 1999, pp. 277–288.

[22] Y. Zhang, G. Wang, Q. Hu, Z. Li, and J. Tian, "Design and performance study of a topology-hiding multipath routing protocol for mobile ad hoc networks," in *IEEE INFOCOM*, 2012, pp. 10–18.

[23] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[24] T. G. Griffin and J. L. Sobrinho, "Metarouting," in *SIGCOMM*, 2005.

[25] L. Gao and J. Rexford, "Stable internet routing without global coordination," *IEEE/ACM Transactions on Networking (TON)*, vol. 9, no. 6, pp. 681–692, 2001.

[26] T. G. Griffin and G. Wilfong, "An analysis of bgp convergence properties," in *SIGCOMM*, 1999.

[27] A. Wang, C. Talcott, A. J. T. Gurney, B. T. Loo, and A. Scedrov, "Reduction-based formal analysis of bgp instances," in *TACAS'12*. Springer-Verlag, 2012, pp. 283–298.

[28] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, no. 1, pp. 73–155, 1992.

[29] A. Wang, C. Talcott, L. Jia, B. T. Loo, and A. Scedrov, "Analyzing bgp instances in maude," in *FMOODS/FORTE, LNCS*, no. 6722, 2011, pp. 334–348.

[30] B. T. Loo, "The design and implementation of declarative networks," Ph.D. dissertation, EECS Department, University of California, Berkeley, December 2006.

[31] J. Lobo, J. Ma, A. Russo, and F. Le, "Declarative distributed computing," in *Correct Reasoning*, 2012, pp. 454–470.

[32] C. Huadmai, "Verification of routing policies by using model checking technique," in *IEEE IDAACS*, vol. 2, 2011, pp. 711–716.

[33] K. Bhargavan, D. Obradovic, Carl, and A. Gunter, "Formal verification of standards for distance vector routing protocols," *Journal of the ACM*, vol. 49, pp. 538–576, 2002.

[34] F. De Renesse and A. Aghvami, "Formal verification of ad-hoc routing protocols using spin model checker," in *IEEE MELECON 2004*, vol. 3, 2004, pp. 1177–1182.

[35] K. Bhargavan, D. Obradovic, Carl, and A. Gunter, "Formal verification of standards for distance vector routing protocols," *Journal of the ACM*, vol. 49, pp. 538–576, 2002.

[36] J. a. L. Sobrinho, "Algebra and algorithms for qos path computation and hop-by-hop routing in the internet," *IEEE/ACM Trans. Netw.*, vol. 10, no. 4, pp. 541–550, 2002.

[37] ——, "Network routing with path vector protocols: theory and applications," in *ACM SIGCOMM '03*, 2003, pp. 49–60.

[38] B. Carre, "Graphs and networks," *Oxford University Press*, 1979.

[39] M. Gondran and M. Minoux, "Graphs and algorithms," *Wiley*, 1984.

[40] ——, "Graphs, dioids, and semirings: New models and algorithms," *Springer*, 2008.

[41] T. G. Griffin and J. L. Sobrinho, "Metarouting," in *ACM SIGCOMM '05*, 2005, pp. 1–12.

[42] F. Le, G. G. Xie, and H. Zhang, "Theory and new primitives for safely connecting routing protocol instances," in *ACM SIGCOMM '10*, 2010, pp. 219–230.

[43] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu, "Formally verifiable networking," in *ACM SIGCOMM HotNets-VIII*, 2009.

[44] Y. Ren, W. Zhou, A. Wang, L. Jia, A. J. Gurney, B. T. Loo, and J. Rexford, "Fsr: Formal analysis and implementation toolkit for safe inter-domain routing," in *ACM SIGCOMM (demonstration)*, 2011, pp. 440–441.

[45] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *NSDI*, 2004, pp. 440–441.

[46] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

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

Jiefei Ma is a postdoctoral researcher at the Department of Computing, Imperial College London, where he obtained his Master of Engineering degree in Computing in 2007, and his Ph.D. in 2012. His research interests include computational logic, machine learning, distributed systems and robotic vision.

Frank Lee is a researcher in the Cloud-based Networks group at IBM T. J. Watson. He received a Ph.D. from Carnegie Mellon University, and a diplome d'ingenieur from the Ecole Nationale Superieure des Telecommunications de Bretagne in France.

Alessandra Russo is a Reader in Applied Computational Logic at the department of Computing, Imperial College London, where she currently leads the Structured and Probabilistic Knowledge Engineering research group. She received her Ph.D. in Computer Science from Imperial College in 1996. She is Editor-in-Chief of IET Software journal and associate editor of ACM Computing Survey.

Jorge Lobo is an ICREA research Professor in the Department of Information and Communication Technologies at UPF, Barcelona and visiting professor at Imperial College London. He received a Ph.D. in Computer Science from the University of Maryland at College Park and a MSc and a BE from Simon Bolivar University in Venezuela. He is an ACM Distinguished Scientist.