Abstract—In this paper we present a general framework for the specification and analysis of declarative distributed computing applications using Answer Set Programming solvers. We first provide an operational semantics of a single computational node, based on Datalog and then show how our framework captures the semantics of a network of computational nodes working together. The framework can express several communication models (e.g. synchronous vs. asynchronous) and distributed solutions to computational tasks can be specified in terms of declarative programs that are not only executable but verifiable. Programs can be analysed with respect to a variety of properties and under different communication models. We take a simple but sufficiently interesting example in the domain of voting/gossip distributed algorithms to illustrate how a distributed computing algorithm can be specified in our framework and show examples of analysis tasks reporting on some experimental results.

I. INTRODUCTION

Recent results in the area of distributed systems and networks have promoted the use of declarative languages for the development of executable specifications of distributed systems (e.g. distributed storage systems, sensor networks, and routing protocols). Their underlying principle is to use declarative programming methods for expressing distributed states, and specifying strategies for computing and maintaining such states according to local information and communication between nodes.

With a very simple conceptual extension, enforcing that the first argument of every predicate in a declarative Datalog-based program must represent a physical location (defined as an IP address or a URI or something similar), Loo has introduced in [1] a programming model to describe and implement network routing protocols. Intuitively, nodes or routers in a network have a copy of the same set of Datalog declarative rules and given an extensional database of ground atoms, the atoms are distributed around the nodes according to their first argument. Computation starts independently bottom-up in each node, nodes send generated ground predicates to the appropriate nodes again based on the first argument of the predicate and other nodes may request information to other nodes to complete the evaluation of their rules. At the end of the computation each node has locally a routing table.

This computational model, known as Declarative Networking, has seen in the recent years several implementations of languages based on this concept [2, 3], and the implementations of protocols in these languages compare well with implementations of the same protocols in C or C++. These implementations, however, need to accommodate typical network changes (e.g. links coming up or down, communication delays, etc.) into their operational semantic, outside the logic. This is because the Datalog semantics is too poor to express state changes, and state changes need to be tracked in order to express these network changes. To address this limitation, Alvaro et al. [2] have proposed a new Declarative Networking language, called Dedalus, where all predicates are extended with an extra argument that represents time. With this extension one can easily represent states (i.e., changes of state can be reflected in changes of truth values of predicates over time), and rules can be viewed as defining state transition functions so overcoming the limitations of Datalog. With this extension Declarative Networking becomes a very general distributed computational model. However, because of the way time was modelled in [2], to avoid limiting the expressiveness of the language, their semantics allows predicates sent from one node to arrive at the receiving node as if the predicates had been sent from the future.

We build upon the ideas of Alvaro et al. and propose a declarative language that does not suffer of this paradox by not relying on a global clock but separating the clocks of each process. This important contribution to the semantics of Declarative Networking enables a very general computation model which is applicable distributed algorithms. We present, in Section II, the main features of our language and describe its operational semantics in terms of state transition systems similar to the transition systems defined by action description languages [3]. As in action languages, specifications written in our language, define a state transition function. It is standard to use logic programs to define the semantics of action domains. We are going to follow the same pattern. However, given that we can define a collection of distributed state machines (DSM) that can exchange information, if we want to analyse these machines our description must also include axioms that capture the semantics of the communication model. We show in Section III how different communication models can be easily expressed in answer set programs (ASP). We will also show examples of properties of distributed algorithms that can be proved using an ASP solver. Section IV reports experimental results on the application of our framework to the analysis of a distributed voting algorithm. Remarks on the scalability and flexibility are also given. Related work and conclusions are in Section V.
II. DISTRIBUTED STATE MACHINES: THE LANGUAGE

We model a network executing distributed algorithms as a collection of distributed state machines (DSM), where each node (e.g., host, router) in the network is abstracted as an input/output automaton that can exchange messages. Each node stores a set of named tuples in the form of $p(a_1, \ldots, a_n)$, where $p$ is the tuple name and $a_1, \ldots, a_n$ is a list of constants. These tuples represent the node’s current state, and hence are called state tuples. A node can input tuples to the state machine it hosting (e.g. the router informing the machine who are the physical neighbours). We will call these tuples input tuples. Tuples can be sent by other nodes as messages. Tuples in messages will be called transport tuples. We assume the different sets of tuple names to be mutually exclusive.

State transitions in a node are triggered when input or transport tuples are received. During a state transition, a set ADD of state tuples, a set DEL of state tuples and a set SEND of transport tuples are computed based on the existing state tuples and the tuples received. Let OLD and NEW be the set of state tuples stored by the automaton before and after the state transition, respectively, then $NEW = (OLD \setminus DEL) \cup ADD$. At the end of the state transition, tuples in SEND will be placed in the outgoing communication channel by the automaton together with the destination address of each tuple. In practice, it is often useful to allow a set of temporary tuples to be derived to support the computation of ADD, DEL and SEND. We call these transient tuples as they are not stored by the automaton after the state transition.

The state transition functions are defined 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. A tuple schema looks very much like a tuple $p(t_1, \ldots, t_n)$ with the difference that each $t_i$ is either a constant, a variable or an expression involving constants and variables. Rules that compute a tuple in $ADD$ are of the form:

$$H \text{ after } L_1, \ldots, L_n, c_1, \ldots, c_m (1)$$

where $H$ is a tuple schema for a state tuple, and $L_1, \ldots, L_n$ is a list of constants. An informal reading of this rule is: if the body of the rule is satisfied with respect to the current state then the state tuple that matches the head will be in $ADD$ (and hence will be stored in $NEW$). Rules that compute tuples in $DEL$ are of the form:

$$\text{forget } H \text{ if } L_1, \ldots, L_n, c_1, \ldots, c_m (2)$$

where $H$ is a tuple schema for a state tuple, and $L_1, \ldots, L_n, c_1, \ldots, c_m$ is defined as in (1). An informal reading of this rule is: if the body of the rule is satisfied with respect to the current state then the state tuple that matches the head will be in $DEL$ (and hence will not be “copied” to $NEW$).

Similarly, rules that compute transient tuples are of the form:

$$H \text{ if } L_1, \ldots, L_n, c_1, \ldots, c_m (3)$$

where $H$ is a tuple schema for a transient tuple, and $L_1, \ldots, L_n, c_1, \ldots, c_m$ is defined as in (1) except that $n \geq 0$ (i.e., it can have an empty body). An informal reading of this rule is: if the body of the rule is satisfied with respect to the current state, then the transient tuple that matches the head must also be considered part of the current state. Note that if this rule has an empty body (i.e., $n = m = 0$), then the tuple that matches the head will temporarily be in every state.

Recall that transport tuples are exchanged between nodes and can be either the input or the output of a state transition. We require that every transport tuple (schema) that appears in a rule must have a suffix of the form $@\text{ID}$, where $\text{ID}$ is either a constant representing a node’s identifier, or a variable whose possible constant values are node identifiers. When a transport tuple (schema) $p(t_1, \ldots, t_n)@\text{ID}$ appears in the body of a rule, it represents a transport tuple received (i.e., as the input of the state transition), and $\text{ID}$ indicates the sender of the tuple. Transport tuple schemas appear only in the head of rules that compute tuples in $SEND$. These are of the form:

$$\text{send } H@\text{ID} \text{ after } L_1, \ldots, L_n, c_1, \ldots, c_m (4)$$

where $H$ is a tuple schema for a transport tuple, and $L_1, \ldots, L_n, c_1, \ldots, c_m$ is defined as in (1). In contrast to transport tuple schemas appearing in the body of rules, the $\text{ID}$ in the head of a rule indicates the destination of the tuple. An informal reading of this rule is: if the body of the rule is satisfied with respect to the current state, then the transport tuple that matches the head will be in $SEND$.

To exemplify the different types of rules, we will describe a Colour Voting algorithm, which works as follows. Each node in a mesh network is assigned an initial colour (either red or blue). The nodes exchange information of their current colour with their neighbours, and decide their new colour based on the majority colour of their neighbours. To specify this algorithm in our language, we use the schemas given in the table below.

<table>
<thead>
<tr>
<th>input</th>
<th>init_neighbour(X)</th>
<th>X is a node’s neighbour.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>init_colour(C)</td>
<td>C is the node’s initial colour.</td>
</tr>
<tr>
<td>state</td>
<td>my_colour(C)</td>
<td>C is the node’s current colour.</td>
</tr>
<tr>
<td></td>
<td>neighbour_colour(X,C)</td>
<td>stored colour of neighbour X is C.</td>
</tr>
<tr>
<td></td>
<td>neighbour(X)</td>
<td>X is a neighbour.</td>
</tr>
<tr>
<td>transient</td>
<td>new_colour(C)</td>
<td>the node updates its colour to C.</td>
</tr>
<tr>
<td></td>
<td>latest_colour(X,C)</td>
<td>latest colour of neighbour X is C.</td>
</tr>
<tr>
<td></td>
<td>num_reds(N)</td>
<td>N is the number of red neighbours.</td>
</tr>
<tr>
<td></td>
<td>num_blues(N)</td>
<td>N is the number of blue neighbours.</td>
</tr>
<tr>
<td>transport</td>
<td>vote_colour(C)</td>
<td>C is the sender’s latest colour.</td>
</tr>
</tbody>
</table>
In our example we assume that the network topology does not change during the algorithm execution, i.e., once a neighbour is added it cannot be removed. Thus, there is only one rule defining neighbour(X):

\[ \text{neighbour}(X) \text{ after } \text{init neighbour}(X). \]  \hspace{1cm} (5)

The newColour(C) is a transient schema for computing the new colour of the current node when a change to the colour is needed. The node then stores this colour in the state using the state tuple (schema) myColour(C) and informs its neighbours of the change using the transport tuple (schema) voteColour(C). The relevant rules are:

\[
\text{newColour}(C) \text{ if initColour}(C), \\
\text{not myColour(red)}, \text{not myColour(blue)}. \\
\text{newColour}(\text{red}) \text{ if} \\
\text{numReds}(N), \text{numBlues}(M), N > M, \text{myColour(blue)}. \\
\text{newColour}(\text{blue}) \text{ if} \\
\text{numReds}(N), \text{numBlues}(M), N < M, \text{myColour(red)}. \\
\text{myColour}(C) \text{ after newColour}(C), \\
\text{forget myColour(OldC)} \text{ if myColour(OldC)}, \text{newColour}(C). \\
\text{send voteColour}(C) \otimes X \text{ after newColour}(C), \text{neighbour}(X). \\
\]  \hspace{1cm} (6)

In addition to storing its own colour, the local node also stores the colour of its neighbours:

\[
\text{neighbourColour}(X, C) \text{ after} \\
\text{voteColour}(C) \otimes X, \text{not neighbourColour}(X, C). \\
\text{forget neighbourColour}(X, \text{OldC}) \text{ if} \\
\text{neighbourColour}(X, \text{OldC}), \text{voteColour}(C) \otimes X, \\
\text{OldC} \neq C. \\
\]  \hspace{1cm} (7)

Finally, the number of red and blue neighbours are computed based on the transient tuple (schema) latestColour(X, C). The latest colour of the current node is the colour currently stored. The local node of neighbour X is either the one received from a transport tuple sent by X, or the one stored at the current state if no transport tuple is received from X. The relevant rules are:

\[
\text{latestColour}(X, C) \text{ if currentHost}(X), \text{myColour}(C), \\
\text{latestColour}(X, C) \text{ if voteColour}(C) \otimes X, \\
\text{latestColour}(X, C) \text{ if} \\
\text{neighbourColour}(X, C), \\
\text{not voteColour(red)} \otimes X, \text{not voteColour(blue)} \otimes X. \\
\text{numReds} (#\text{count}(X)) \text{ if latestColour}(X, \text{red}). \\
\text{numBlues} (#\text{count}(X)) \text{ if latestColour}(X, \text{blue}). \\
\]  \hspace{1cm} (8)

In the above last two rules, the operator #\text{count}(X) is used. This is an example of an expression involving a variable in the argument of a tuple schema. #\text{count} is an aggregation operation. In the last rule, #\text{count}(X) is equal to the total number of constant values for X extracted from the tuples matching latestColour(X, blue) in the current state. Other aggregation operations supported by our language are #\text{sum}(X), #\text{min}(X) and #\text{max}(X), which return the sum, the smallest value and the biggest value of X, respectively (in these cases the possible values for X must be numerical). The other operators allowed in expressions are arithmetic operations with their standard semantics.

The full specification for the Colour Voting algorithm is therefore the set of rules (5)–(19). Each node that participates in execution of the algorithm will be running these rules.

A. Distributed State Machine: the semantics

The semantics of a single node can be described by a (Datalog+time) logic program by mapping each tuple to a predicate with new arguments representing location and time. First, we introduce two sorts: the identifier sort (used to uniquely identify a node in the network) and the time sort (which is the set of non-negative integers plus a special constant null). Secondly, for each tuple type we introduce one or two corresponding predicates. This is illustrated in Table I, where \( \pi \) represents a list of terms.

Let \( B = l_1(\bar{V}_1), \ldots, l_n(\bar{V}_n), c_1, \ldots, c_n \) be the body of a declarative rule. The translated body \( B(\text{ID}, T) \) is obtained from \( B \) by translating each \( l_i(\bar{V}_i) \) as follows. If \( p(\bar{V}) \) is a non-transport tuple schema of \( l_i(\bar{V}_i) \), then it is replaced with the predicate \( p(\text{ID}, \bar{V}, T) \); otherwise let \( p(\bar{V}) \@ \text{Src} \) be the transport tuple schema of \( l_i(\bar{V}_i) \), it is replaced with the predicate \( \text{receive}_p(\text{ID}, \bar{V}, T) \). Note that in \( B(\text{ID}, T) \) all predicates have the same first argument (i.e., the location) and last argument (i.e., the time).

The declarative rule \( p(\bar{V}) \) after \( B \) for a state tuple schema \( p(\bar{V}) \) is translated to a Datalog+time rule:

\[ p(\text{ID}, \bar{V}, T + 1) := B(\text{ID}, T) \]  \hspace{1cm} (20)

and \( \text{forget } p(\bar{V}) \) if \( B \) is translated to a Datalog+time rule:

\[ \text{forget}_p(\text{ID}, \bar{V}, T) := B(\text{ID}, T) \]  \hspace{1cm} (21)

Similarly, the declarative rule \( p(\bar{V}) \) if \( B \) for a transient tuple schema \( p(\bar{V}) \) is translated to a Datalog+time rule:

\[ p(\text{ID}, \bar{V}, T) := B(\text{ID}, T) \]  \hspace{1cm} (22)

To capture the default persistence nature of state tuples, for each state tuple schema \( p(\bar{V}) \), where \( V \) are all variables, an inertia rule is added:

\[ p(\text{ID}, \bar{V}, T + 1) := p(\text{ID}, \bar{V}, T), \text{not forget}_p(\text{ID}, \bar{V}, T) \]  \hspace{1cm} (23)

There is no such inertia rule for transient tuples. Finally, the declarative rule \( \text{send } p(\bar{V}) \@ \text{Dest} \) after \( B \) for a transport tuple schema \( p(\bar{V}) \) is translated to a Datalog+time rule:

\[ \text{send}_p(\text{ID}, \bar{V}, \text{Dest}, T + 1) := B(\text{ID}, T) \]  \hspace{1cm} (24)

We will denote with \( M_{id} \) the set of Datalog+time rules associated with a node identified by \( id \), and with \( M \) the union of all the rules \( M_{id} \) for all the \( id \). Note that if the set of rules is the same in every node then \( M = M_{id} \).

The computational model of a single machine: We are going to assume that the standard definition of stratification over negation and aggregation applies to the collection of logic programming rules defining transient tuples (i.e. tuples defined by (22)). The (local) stratification of the other rules is instead guaranteed because the heads of the rules are always in a different time \((+1)\) than the literals in the body. Hence, given an OLD set of state tuples in a node identified by \( id \) at time \( t \), and a set of input tuples \( I \) and transport tuples \( M \) that are
received by node id at time t, the set NEW can be calculated as follows:
1) Let OLD(id, t) = \{ p(id, π, t) | p(π) ∈ OLD \}.
2) Let I(id, t) = \{ p(id, π, t) | p(π) ∈ I \}.
3) Let M(id, t) = \{ p(id, π, id’, t) | p(π)@id’ ∈ M \}.
4) Let A be the unique answer set associated with M(id) ∪ OLD(id, t) ∪ I(id, t) ∪ M(id, t).
5) NEW = \{ p(π)|p(id, π, t + 1) ∈ A, π a state predicate \}.
6) SEND = \{ (p(π)|@id, t+1) | send_p(id, π, id1, t) \}.

The reader can observe that the evaluation of the transition function can be done using a simple Datalog program with aggregates since there are only two states mentioned in the rules, the current state and the successor state.\footnote{Supported by this observation, we have an implementation of a DSM system in which we have used a relational database as the core of the rule evaluation engine.}

Our goal is to provide a semantics for the overall system of distributed state machines so that we can do analysis of distributed programs. For that we need to create a bridge between the rules of nodes with different identifiers. This bridge is done through the receive_p and the send_p predicates: for each transport tuple schema p(\overline{ν}), where \overline{ν} are all variables, the following two rules are added:

\begin{align}
\text{receive}_p(Dest, \overline{V}, Src, SendT, RecvT) & : - \quad (25) \\
\text{send}_p(Src, \overline{V}, Dest, SendT), \\
\text{choice}((Src, \overline{V}, Dest, SendT), (RecvT)) & , \\
\text{receive}_p(Dest, \overline{V}, Src, RecvT) & : - \quad (26) \\
\text{receive}_p(Dest, \overline{V}, Src, SendT, RecvT).
\end{align}

<table>
<thead>
<tr>
<th>Tuple Type</th>
<th>Tuple</th>
<th>Predicate</th>
<th>Intuitive Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>p(id, \overline{v})</td>
<td>\text{p(id, \overline{v}, t)}</td>
<td>it holds iff p(id, \overline{v}) is stored by node id at the state associated with time t.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{forget_p(id, \overline{v}, t)}</td>
<td>it holds iff p(id, \overline{v}) is in the set \text{DEL} of node id at the state associated with time t.</td>
</tr>
<tr>
<td>input</td>
<td>p(id, \overline{v})</td>
<td>\text{p(id, \overline{v}, t)}</td>
<td>it holds iff p(id, \overline{v}) is manually inserted to node id at the state associated with time t.</td>
</tr>
<tr>
<td>transient</td>
<td>p(id, \overline{v})</td>
<td>\text{p(id, \overline{v}, t)}</td>
<td>it holds iff p(id, \overline{v}) is derived by node id at the state associated with time t.</td>
</tr>
<tr>
<td>transport</td>
<td>p(id, \overline{v}@id1)</td>
<td>\text{send_p(id, \overline{v}, id1, t)}</td>
<td>it holds iff p(id, \overline{v}) is (in the head) with destination id1 is in the set \text{SEND} of node id at the state associated with time t.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{receive_p(id, \overline{v}, id1, t)}</td>
<td>it holds iff p(id, \overline{v}) (in the body) is sent by node id1 at id1's state associated with (send) time t1 and received by node id at id's state associated with (receive) time t.</td>
</tr>
</tbody>
</table>


\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Tuple Type & Tuple & Predicate & Intuitive Meaning \\
\hline
state & p(id, \overline{v}) & \text{p(id, \overline{v}, t)} & it holds iff p(id, \overline{v}) is stored by node id at the state associated with time t. \\
& & \text{forget_p(id, \overline{v}, t)} & it holds iff p(id, \overline{v}) is in the set \text{DEL} of node id at the state associated with time t. \\
input & p(id, \overline{v}) & \text{p(id, \overline{v}, t)} & it holds iff p(id, \overline{v}) is manually inserted to node id at the state associated with time t. \\
transient & p(id, \overline{v}) & \text{p(id, \overline{v}, t)} & it holds iff p(id, \overline{v}) is derived by node id at the state associated with time t. \\
transport & p(id, \overline{v}@id1) & \text{send_p(id, \overline{v}, id1, t)} & it holds iff p(id, \overline{v}) is (in the head) with destination id1 is in the set \text{SEND} of node id at the state associated with time t. \\
& & \text{receive_p(id, \overline{v}, id1, t)} & it holds iff p(id, \overline{v}) (in the body) is sent by node id1 at id1's state associated with (send) time t1 and received by node id at id's state associated with (receive) time t. \\
\hline
\end{tabular}
\caption{Corresponding Predicates for Tuples}
\end{table}

The main purpose of a communication model is to describe how the send and receive tuples are related. Various assumptions can be made for the communication channel, such as whether a message can be lost, duplicated or received out of order. There are two basic models for distributed algorithms: \text{synchronous} and \text{asynchronous}. In a synchronous communication model, all the nodes’ local clocks are synchronised (i.e., each local clock is the same as a global clock), and all the nodes send out tuples, receive tuples and update states at the same time, respectively. In an asynchronous communication model nodes’ local clocks are not synchronised.

The communication model assumptions are often specified using \text{integrity constraints}. Implementations of ASP solvers require finite domains. However, in our case the time sort has an infinite number of possible values. To restrict this we assume that there is a pre-specified special value \text{MaxTime} such that the time sort (for analysis) is the set of integers between 0 and \text{MaxTime} (i.e., no need for the special constant \text{null}). The
algorithm execution (during analysis) is affected by MaxTime as follows. Each node can go through at most MaxTime state transitions, which are indexed from 0 to MaxTime − 1. A node may receive tuples at a terminating state indexed with MaxTime, but these tuples do not trigger a new state transition (i.e., the tuples are considered to be received too late and ignored, or be lost during communication). A node may also send tuples at MaxTime (i.e., the transport tuples in SEND computed at MaxTime − 1), but these tuples will not be received by the destination node (or received at time Maxtime local to the destination). Note that the terminating state of each node is particularly useful when we perform analysis on algorithm convergence, e.g., the algorithm converges when there are no tuples sent or received at each node’s terminating state (i.e., all the communication channels are empty). This will be further discussed in Section III-B. We now describe how synchronous and asynchronous communication models are specified in ASP.

1) Interfacing Predicates: The specifications for a distributed algorithm and the adopted communication model are loosely coupled through a set of interfacing predicates. These are communication(Src, Dest, SendT, RecvT), message_sent(X, T) and message_received(X, T). These are defined using the transport predicates (i.e., corresponding predicate for transport tuple schema) from the algorithm specification. For example, for each transport predicate receive_p(ID, V, ID1, T1, T) we have the following two rules:

\[
\text{communication}(\text{ID}, \text{ID1}, \text{T1}, \text{T}) \rightarrow \text{receive_p}(\text{ID1}, \text{V}, \text{ID}, \text{T}, \text{T}) \quad (27)
\]

\[
\text{message_required}(\text{ID}, \text{V}) \rightarrow \text{receive_p}(\text{ID}, \text{V}, \text{ID1}, \text{T}, \text{T}) \quad (28)
\]

Similarly, for each transport predicate send_p(ID, V, ID1, T) there is a rule:

\[
\text{message_sent}(\text{ID}, \text{V}) \rightarrow \text{send_p}(\text{ID}, \text{V}, \text{ID1}, \text{T}) \quad (29)
\]

A communication model is a set of rules and integrity constraints specified using these and the transport predicates.

2) Synchronous Model: The simplest communication model is a synchronous model, in which all the nodes’ local clocks are the same and each message sent will be received at the same state. Note that the transport tuples are computed in the state right before that in which they are sent, so we may assume the messages sent are delivered immediately. This model has only one integrity constraint:

\[
\rightarrow \text{communication}(\text{Src}, \text{Dest}, \text{SendT}, \text{RecvT}), \text{RecvT} = \text{SendT} \quad (30)
\]

This integrity constraint effectively forces the implementation of the choice predicate in (25) to assign the value SendT to RecvT. If we want to allow a message to be lost during transmission, it can be easily done by modifying (30) to be:

\[
\rightarrow \text{communication}(\text{Src}, \text{Dest}, \text{SendT}, \text{RecvT}), \\
\quad \text{RecvT} = \text{SendT}, \text{get_maxtime}(\text{MaxTime}), \text{RecvT} = \text{MaxTime} \quad (31)
\]

Thus, a synchronous communication model, C, is defined by the set of rules and integrity constraints \{(25), (26), (27), (28), (29), (30)\} or the set \{(25), (26), (27), (28), (29), (31)\}.

Note that in the synchronous model, a node’s local clock may advance by 1 even though it does not receive any transport or input tuple.

3) Asynchronous Model: The main difference between an asynchronous and a synchronous model is that in the former the nodes’ local clocks are not synchronised. Each node’s local time is understood by that node only. Therefore, we cannot guarantee at what state a node may receive a tuple that is sent to it. This poses significant challenges in the modelling.

First of all, if one wants to guarantee that the choice predicate assigns the right values for message delivery time so that no node can receive a message that is sent in the future we need to implement the Lamport timestamps [9]. This can be easily specified through the following rules:

\[
\text{local_state_ts}(\text{ID}, \text{T}) := \\
\quad \text{message_required}(\text{ID}, \text{T}), \quad \text{get_maxtime}(\text{MaxTime}), \text{T} = \text{MaxTime} \quad (32)
\]

\[
\text{local_state_ts}(\text{ID}, \text{T}) := \\
\quad \text{message_required}(\text{ID}, \text{T}), \quad \text{get_maxtime}(\text{MaxTime}), \text{T} = \text{MaxTime} \quad (33)
\]

\[
\rightarrow \text{globally_precedes}(\text{ID}, \text{LT1}, \text{ID}, \text{LT2}) := \\
\quad \text{local_state_ts}(\text{ID}, \text{LT1}), \text{local_state_ts}(\text{ID}, \text{LT2}), \text{LT1} < \text{LT2} \quad (34)
\]

\[
\rightarrow \text{globally_precedes}(\text{ID}, \text{LT1} - 1, \text{ID2}, \text{LT2}) := \\
\quad \text{communication}(\text{ID}, \text{ID2}, \text{LT1}, \text{LT2}) \quad (35)
\]

\[
\rightarrow \text{globally_precedes}(\text{ID}, \text{LT1}, \text{ID}, \text{LT3}), \\
\quad \text{globally_precedes}(\text{ID}, \text{LT3}, \text{ID}, \text{LT2}) \quad (36)
\]

The first rules simply collect the local state timestamps as a set of (identifier, time) pairs, based on the interactions between the nodes. The predicate globally_precedes(ID1, LT1, ID2, LT2) can be interpreted as the event with timestamp (ID1,LT1) is logically before the event with timestamp (ID2,LT2). This is defined by three cases: First, (34) says that the a local event globally precedes another local event if the former occurs before the latter at the same node (i.e. having a smaller local time). Second, (35) says that if there is a message passing between two nodes, then the send event globally precedes the receive event. Note that in the head of (35), the use of LT1 − 1 instead of LT1 for the second argument is due to the fact that transport tuples are sent at the next state after they are computed. By using LT1 − 1 we allow a node to receive and process a transport tuple at the same time it sends out transport tuples computed at the previous state. Third, (36) computes the transitive closure of globally_precedes. Finally, the integrity constraint for guaranteeing correct local time assignments by choice is:

\[
\rightarrow \text{globally_precedes}(\text{ID}, \text{LT}, \text{ID}, \text{LT}) \quad (37)
\]

which means no event can globally precede itself. With the above integrity constraint, several communication invariants can be specified. For example, the assumption that “a message sent too late by a node (i.e., at local time MaxTime) will not be processed by the destination” is:

\[
\rightarrow \text{communication}(\text{Src}, \text{Dest}, \text{maxtime}, \text{RecvT}), \\
\quad \text{get_maxtime}(\text{MaxTime}), \text{RecvT} = \text{MaxTime} \quad (38)
\]
The assumption that “messages sent by a source node to a destination node must be delivered in order” is:

\[
\begin{align*}
\text{message received}(\text{Receiver}, \text{RecvTime}), \\
\text{time}(\text{SomeTime}, \text{SomeTime} < \text{RecvTime}), \\
\text{not message received}(\text{Receiver}, \text{SomeTime}).
\end{align*}
\]

(40)

The assumption that “each node must process at least one message before its communication queue becomes empty” is:

\[
\begin{align*}
\text{message received}(\text{Receiver}, \text{RecvTime}), \\
\text{time}(\text{SomeTime}, \text{SomeTime} < \text{RecvTime}), \\
\text{not message received}(\text{Receiver}, \text{SomeTime}).
\end{align*}
\]

The assumption that “node must process at least one message before its communication queue becomes empty” is:

\[
\begin{align*}
\text{message received}(\text{Receiver}, \text{RecvTime}), \\
\text{time}(\text{SomeTime}, \text{SomeTime} < \text{RecvTime}), \\
\text{not message received}(\text{Receiver}, \text{SomeTime}).
\end{align*}
\]

(39)

Depending on the desired properties, an asynchronous communication model, \( C \), will include the rules \{(25), (26), (27), (28), (29), (37)\} any specific set of rules and constraints (e.g., \{(38), (39), (40)\}) that corresponds to the appropriate model.

### B. Query Language

The query language for algorithm analysis allows narratives and queries to be specified. Narratives are observations made from a node state. Queries are either hypothetical assertions about the effects of the distributed algorithm execution based on the observations, or assertions about the past states which the algorithm has executed.

The basic components of narratives and queries are called **axiom schemas**, which are of the following four forms:

\[
\begin{align*}
\text{P holds at ID}.T, \\
\text{I inserted at ID}.T, \\
\text{M from ID'} received at ID}.T, \\
\text{M to ID'} sent at ID}.T.
\end{align*}
\]

where \( P \) is a either a state or transient tuple schema, \( I \) is an input tuple schema, \( M \) is a transport tuple schema, \( ID \) and \( ID' \) are constants or variables of the identifier sort, and \( T \) is a constant or variable of the time sort.

A **narrative axiom** is a variable-free axiom schema and a **narrative** is a finite collection of narrative axioms. For a concrete example of a narrative axiom (or axiom to simplify notation), let us consider again the Colour Voting algorithm, which is to be analysed (for convergence) with respect to a network configuration shown in Figure 1.

The set of axioms describing such network configuration is:

\[
\begin{align*}
\text{neighbour}(2) \text{ holds at } 1.0, \\
\text{my colour(red) holds at } 1.0, \\
\text{vote colour(red) to } 2 \text{ sent at } 1.0, \\
\text{vote colour(red) to } 4 \text{ sent at } 1.0.
\end{align*}
\]

A **query definition rule** is a rule for the form:

\[
\begin{align*}
\text{if } \text{given } L_1, \ldots, L_n, \ c_1, \ldots, c_n \\
\text{then } \text{has unprocessed message given } \\
\text{vote colour(C)@Dest sent at ID}.MaxTime, \\
\text{has unprocessed message given } \\
\text{vote colour(C)@Src received at ID}.MaxTime, \\
\text{converged given not has unprocessed message.}
\end{align*}
\]

where \( \text{has unprocessed message} \) and \( \text{converged} \) are query predicates. Therefore, three different queries \( \text{always converged}, \text{sometimes converged} \), and \( \text{never converged} \) can be asked to check whether the algorithm can always converge, sometimes converge or never converge, respectively, within \( m \) steps given a particular communication model and network configuration (as a narrative).

### C. Formal Semantics of Narratives and Queries

Narratives and queries can also be translated into Data-log-time rules to describe their semantics.

First, each axiom schema, \( S \), can be translated to a corresponding predicate, \( \text{tr} (S) \), using the following table:

<table>
<thead>
<tr>
<th>Schema Axiom</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(V) \text{ holds at ID}.T )</td>
<td>( p(ID, V, T) )</td>
</tr>
<tr>
<td>( p(V) \text{ inserted at ID}.T )</td>
<td>( p(ID, V, T) )</td>
</tr>
<tr>
<td>( p(V) \text{ from ID} received at ID}.T )</td>
<td>( receive_p(ID, V, ID', T) )</td>
</tr>
<tr>
<td>( p(V) \text{ to ID} sent at ID}.T )</td>
<td>( send_p(ID, V, ID', T) )</td>
</tr>
</tbody>
</table>

Each narrative axiom, \( L \), can then be translated into a fact, \( \text{tr}(L) \), (i.e., a rule with an empty body) following the table mapping. Similarly, each query definition rule can be

![Network diagram](image-url)  

Fig. 1. A network with four nodes: Each line represents a bidirectional communication channel between two nodes. Node 1 and Node 3 are initially assigned to red; Node 2 and Node 4 are initially assigned to blue.
translated into a rule by simply replacing given with :- and translating each schema axiom in the body according to the table. Given a narrative $N$ and a set of query definition rules, $Q$, we denote by $tr(N)$ and $tr(Q)$, the set of facts and logic programming rules that result from the translation.

**Definition 1:** Let $\mathcal{M}$ be the set of rules associated with a collection of DSM programs, $C$ a communication model with $\text{MaxTime} = m$, $N$ a narrative and $Q$ a set of query definition rules. Let $\Pi = \mathcal{M} \cup C \cup tr(N) \cup tr(Q)$.

- A query sometimes$_n L_1, \ldots, L_n$ is true if and only if there exists an answer set $M$ of $\Pi$ such that $M \models tr(L_1), \ldots, tr(L_n)$.
- A query always$_n L_1, \ldots, L_n$ is true if and only if for every answer set $M$ of $\Pi$ it holds that $M \models tr(L_1), \ldots, tr(L_n)$.
- A query never$_n L_1, \ldots, L_n$ is true if and only if the query sometimes$_n L_1, \ldots, L_n$ is false.

**IV. Example Analysis**

Distributed algorithms can be classified as asynchronous or synchronous. Synchronous means that the participants send and receive data at agreed on times. As an example, a voting protocol where every participant announces its colour to its neighbors at 5 PM of each day is synchronous. In contrast, transmissions in asynchronous distributed algorithms are not synchronized by an external clock. Routing protocols, or the postal mail services, where the messages may have arbitrary delays, are asynchronous. Our framework can analyze properties of both classes of distributed algorithms.

First, we consider the voting protocol previously described and assume a synchronous communication model. We note that the algorithm is deterministic. We address distributed algorithms with non deterministic behaviors later. Because both the algorithm and the communication models are deterministic, a given initial state of the network always produces the same outcome. As such, we can study properties of the initial network states and answer questions such as: which initial configurations result in a divergent state? Formally, given a network topology and initial colour setting, we say that a network converges if there exists a time after which no node changes its colour. The network diverges otherwise. ASP can answer the above question by computing and analyzing all the possible network executions. To verify convergence, we simply need to find a $\text{MaxTime}$ such that there is no more communication between the nodes. ASP can identify these networks through the following query:

$$\text{converged} :- \text{message\_received(AnyNode, MaxTime)}.$$ 

As for identifying the cases that diverge, it suffices to identify a network execution with a global state at time $t_2$, which is identical with a global state at $t_1$ ($t_2 < t_1$). Due to the deterministic properties of the distributed algorithm, the cycle that led to the repetition will reproduce continuously and the network will never converge. ASP can identify the repetition of global states in a network execution with an implementation of repeated states:

$$\text{repeated\_states} \leftarrow \exists T_1, T_2. \forall N_1, N_2, C. [\text{receive\_vote\_colour}(N_d, C, N_1, T_1) \leftarrow \text{receive\_vote\_colour}(N_d, C, N_2, T_2) \lor \forall N, N_1, C, C_1 \leftarrow \text{my\_colour}(N, C, T_1) \land \text{my\_colour}(N, C, T_2) \land \text{neighbour\_colour}(N, N_1, C, T_1) \land \text{neighbour\_colour}(N, N_2, C, T_2)]]$$

As such, to identify all the initial configurations of $N$ nodes that converge (or diverge), we can apply the following procedure: First, we generate all the network topologies of $N$ nodes. Then, for each initial configuration, we set $\text{MaxTime}$ to 1, and relies on ASP to find a model that satisfies both : -converged and : -repeated_state (i.e., the execution either has converged, or diverges). As long as ASP finds a model, we increment $\text{MaxTime}$, and resubmit the ASP query. When ASP no longer finds a solution, the absence of model indicates that we have explored all the network executions, and can enumerate all the cases that converge and all those that diverge. Figure 1 depicts an example of an initial configuration that diverges. Table II summarizes the number of networks (from 3 to 8 nodes) that converge, and the number of networks that diverge.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Graphs</th>
<th>Convergent Configs.</th>
<th>Divergent Configs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>94</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>661</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>112</td>
<td>7042</td>
<td>126</td>
</tr>
<tr>
<td>7</td>
<td>853</td>
<td>107443</td>
<td>1741</td>
</tr>
<tr>
<td>8</td>
<td>11117</td>
<td>2513729</td>
<td>32223</td>
</tr>
</tbody>
</table>

**TABLE II**

**Synchronous Example Statistics**

Asynchronous communication models may cause non deterministic outcomes: an initial configuration may result in more than one final state. For example, for the same initial configuration, a network execution may diverge, while another may converge. To illustrate this, we reconsider the previously presented voting protocol but assume an asynchronous communication model (e.g., the nodes may be communicating by email which can present arbitrary delays). We focus on the configuration depicted in Figure 1. Although this configuration was previously shown to diverge, ASP identified a network execution (Figure 2) that converges. In summary, while the configuration of Figure 1 always diverges in a synchronous mode of communication, it can converge or diverge when considering an asynchronous mode of communication. ASP can analyze the convergence and divergence properties of different configurations. However, asynchronous distributed algorithms can result in a large number of states to explore. The scalability of the approach is an open question, and we are currently investigating solutions to address it. For example, we note that to answer the initial question (which configurations result in a divergent state?), we may not need to enumerate all the possible network topologies as many of them are isomorphic. Similarly, we believe that taking advantage of other characteristics of the properties to be verified can help
with the scalability issues. In addition, other techniques like abduction, planning and model checking should be explored — many analyses are computationally hard and we should make use of all the tools available now that we have formal logic representation of these problems.

![Diagram](image_url)

Fig. 2. A converged (all red) execution trace within \( \text{MaxTime} = 4 \)

V. RELATED WORK AND FINAL REMARKS

Our work builds on the programming language Dedalus of Alvaro et al [2]. In [10], Dedalus is used as the basis for a programming language that mixes declarative and imperative specifications and some high level analysis of concurrency and synchronisation is done. Several analysis systems have been developed for declarative networking. They are limited by the restricted semantics of declarative networking languages as in the work described in [11] and in many cases they are specific for the analysis of routing protocols as in [12] and [13]. Datalog has been used to analyze distributed discrete event systems described as Petri nets [14]. This is an interesting area to explore next. The description of transition systems using logic rules is not limited to action languages. Relational transducers [15] have been used to define state transition systems mainly with the goal of describing web-services and service composition and not distributed computing. We are the first ones to use ASP and we believe that our model and ASP are a very good combination for the study of many distributed programs.

We implemented a parser which translates DSM specifications and queries into their corresponding answer set program and integrity constraints. Clingo [8] was used as the underlying ASP solver for model computation. We have also used this implementation to check properties of two different classes of network routing protocols: path-vector based protocols and link state protocols [16], [17]. In path vector protocols we are able to prove protocol convergence under conditions for which no analytical conditions for convergence are known [18], and checking the convergence manually is virtually impossible.

There are many directions of research that can be followed. We will mention a few. Our query language still has several drawbacks. The query definition rules are very general but the model does not provide too much guidance to the programmer about what kind of rules are needed and how to write them. Adding explicitly temporal operators might be useful. We are also investigating queries that incorporate some type of abductive reasoning [19]. For example, an interesting query for the voting algorithm is to find assignments of colours for the nodes of a given region in the graph are mostly red, are questions of interest for an analyst.

ACKNOWLEDGMENT

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

REFERENCES