Verification of Fixed-Topology Declarative Distributed Systems with External Data

Diego Calvanese[1], Marco Montali[1], and Jorge Lobo[2]

[1] Faculty of Computer Science, Free-University of Bozen-Bolzano, Bolzano, Italy,
{calvanese,montali}@inf.unibz.it
[2] ICREA, University of Pompeu Fabra, Barcelona, Spain,
jorge.lobo@upf.edu

Abstract. Logic-based languages, such as Datalog and Answer Set Programming, have been recently put forward as a data-centric model to specify and implement network services and protocols. This approach provides the basis for declarative distributed computing, where a distributed system consists of a network of computational nodes, each evolving an internal database and exchanging data with the other nodes. Verifying these systems against temporal dynamic specifications is of crucial importance. In this paper, we attack this problem by considering the case where the network is a fixed connected graph, and nodes can incorporate fresh data from the external environment. As a verification formalism, we consider branching-time, first-order temporal logics. We study the problem from different angles, delineating the decidability frontier and providing tight complexity bounds for the decidable cases.

1 Introduction

It has been recently shown that declarative query languages, such as Datalog, could naturally be used to specify and implement network services and protocols [15]. The approach, referred to as *declarative networking* [3], makes the specifications of complex network protocols concise and intuitive, and directly executable through distributed query processing algorithms [20]. Applications for declarative networking go far beyond network protocols, and languages and techniques developed in this setting provide the basis for *declarative distributed computing*. This paradigm has been used widely, e.g., as the core of the Webdam language for distributed Web applications [1]. We refer to these systems as *declarative distributed systems* (DDSs).

There are several variants of concrete languages for specifying DDSs [16,2,1,18], but their common denominator is *data-centricity*: computations in a single node are limited to evaluations of queries on a relational database (RDB), and messages passed between nodes are snippets of DBs, providing a close correspondence between the programs and a formal specification in logic of their computation. This facilitates the development of program analysis tools (see, e.g., [21,18]). However, in spite of several studies on the foundations of DDSs [13,4], *a rigorous, comprehensive characterization of the decidability and complexity of verification in such systems, is yet to come.*

In this paper, we provide a stepping stone towards a formal, systematic characterization of verification of DDSs. As the basis for our investigation, we rely on the D2C

approach in [18], for which verification techniques have been developed [3]. D2C is the only proposal that decouples the declarative specification of computations inside nodes, from the inter-node communication model. Differently from DDS languages originated in Berkeley, such as NDlog or Overlog [16], whose formal semantics is not given in terms of a logical interpretation but as a transformation of the program, not specified in Datalog, D2C is strongly grounded in Datalog. The syntax of D2C programs is merely a syntactic sugar for Datalog$_{1S}$ programs with negation, whose semantics is that of Stable Models. Note that if the communication model can be described using Datalog rules, the semantics of the entire system is given by the Stable Models of a logic program, as, e.g., for the asynchronous models of WebdamLog and Dedalus, c.f. [18].

In this work, starting from [18], we provide a comprehensive formalization of DDSs and their execution semantics. Differently from previous proposals, which model DDSs as closed systems, nodes may receive data from its neighbor nodes, and also from the external environment, to account for the interaction with users and with software in the application layer. The external sources may inject fresh data into the node DBs, making the system infinite state in general. Hence, while our approach is similar, in spirit, to that of [18] and [5], it is substantially richer from the technical point of view.

In this light, the formal model proposed in this paper can be seen as a distributed version of data-aware business processes and artifact-centric systems [10,8,6]. For such systems, verification has been extensively studied over the last two decades [9], but typically focusing on a "monolithic" process operating over an RDB, without taking into account distributed, interacting dynamic components (an exception is [11]). However, our work is the first one to consider different communication models and to explicitly incorporate the topology as a parameter.

To specify interesting properties over DDSs, we use a first-order (FO) variant of CTL [8], which accounts for the evolution both of a DDS and of the node data. Notably, this logic can express fundamental properties related to *convergence of computation*.

We show that, in general, verification is undecidable even for specific convergence properties and extremely limited, single-node DDSs. To tame this strong negative result, we leverage the notion of *state-boundedness* [8,6], and detail the decidability frontier of verification when a bound is imposed on node relations storing the internal state, incoming/outgoing messages, and/or external inputs. For the decidable cases, we also provide interesting tight complexity bounds.

2 Declarative Distributed Computing

The general computational model of DDSs can be described as a network of uniquely identified nodes, each running an input/output state machine, where outputs of some machines become inputs of others [17]. We discuss how state machines and their interaction can be modeled declaratively. We assume familiarity with Datalog and the stable model semantics [12], and we adopt the standard conventions.

2.1 Computational Model in a Node

A state in the state machine of a node is represented by a (relational) DB. State transitions occur when the node receives inputs, in the form of DBs, from external compo-

nents (such as applications running in the node, or humans), and/or from state machines running in other nodes. A state change can also produce an output in the form of a DB that can be delivered to another node. Thus, every node has an *input* schema $\mathcal{I}$, a *state* schema $\mathcal{S}$, and a schema $\mathcal{T}$, called *transport*, of the DBs used to communicate between nodes. Let $\mathbb{I}$, $\mathbb{S}$, and $\mathbb{T}$ denote all possible instances of $\mathcal{I}, \mathcal{S}$, and $\mathcal{T}$, respectively. The state transition mapping in a node is a subset of $\mathbb{S} \times \mathbb{T} \times \mathbb{I} \times \mathbb{S} \times \mathbb{T}$: given a pair (S, T) of state and transport DBs, and an input input DB I, the transition results in new pair (S_n, T_n) of state and transport DBs. In declarative distributed computing, this state transition mapping is defined using some dialect of Datalog. Intuitively, given a Datalog-like program $\mathcal{P}$ and an encoding of (S, T, I) as an extensional DB D, (S_n, T_n) is extracted from the fixpoint computation over $\mathcal{P} \cup D$.

Since plain Datalog is too poor to express state changes, we adopt D2C, a declarative programming language introduced in [18], with a semantics based on Datalog. A state transition mapping in D2C is defined by rules of the form:

$$H \ \textbf{if} \ L_1, \ldots, L_n, \textbf{prev} \ L_{n+1}, \ldots, L_m, C$$

Like in Datalog, H is the *head* atom, but in D2C it is possibly annotated with a term of the form $@t$, where t is a variable or constant from a fixed domain (of, e.g., IP addresses or URLs) used to identify nodes. The L_is are *literals* (i.e., atoms or negated atoms), again possibly annotated with a term of the form $@t$. Finally, C is a set of (in)equality constraints over variables and constants.

The predicate names follow the standard correspondence of predicates and DB tables made by Datalog. All variables appearing in H, in C, or in any negated atom inside a rule must also appear in a non-negated atom among the L_is of the same rule[3]. The name of annotated atoms must correspond to relation names in the transport schema $\mathcal{T}$ of the node, and only names that correspond to transport or state tables can appear in H. Informally, a ground instance $h \ \textbf{if} \ l_1, \ldots, l_n, \textbf{prev} \ l_{n+1}, \ldots, l_m, c$ of a rule says that h is in the current state or is an output of the current state if $l_1, \ldots, l_n$ are true in the current state, $l_{n+1}, \ldots, l_m$ were true in the previous state, and c is valid. As in Datalog with negation, we make the usual assumption on the *stratification* of negated atoms in $l_1, \ldots, l_n$ w.r.t. h, so that the transition mapping can be computed using the standard Datalog fixpoint evaluation of Datalog.

Example 1. Consider the input predicate `echo(m,d)`, where `m` is a message and `d` is a node. Rule `say(M)@D` **if** `echo(M,D)` models that the node sends `m` to `d` when the corresponding `echo` predicate is given as input. `M` and `D` are variables, which in this case are bound to constants `m` and `d` respectively. We denote variables by upper-case strings. Rule `alive(S)` **if** `say(M)@S` models that the node adds to its state the information that `s` is alive, whenever the transport tuple `say(_)` is received from node `s` (in this case, variable `S` is bound to `s`). This example shows that the meaning of an annotation depends on where it appears: if in the head of a rule, it indicates the destination of the transport tuple, if in the body, it points to the source. Thus, rule `say(M)@Src` **if** `say(M)@Src` "echoes" the `say` tuple back to the source node. ■

Notably, a D2C state transition mapping $\mathcal{P}$ can be reduced to a plain Datalog program $[\mathcal{P}]$ following the technique in [18], i.e., by *(i)* internalizing @ annotations as an

[3] This requirement can be relaxed for negated atoms containing anonymous variables.

additional relation argument; *(ii)* using two predicates (s_p and r_p) to differentiate between an outgoing and an incoming transport tuple of the same p; *(iii)* duplicating every predicate to account for the current and previous DBs. Note that the operator $[\cdot]$ can be applied also to a set of Datalog facts, i.e., an RDB. This reduction makes it possible to declaratively characterize the computations of a D2C system, including the communication model, and to simulate them with an ASP engine (for *closed* DDSs – see below).

Non-determinism. The language described so far only supports *deterministic* state transitions. To support also non-deterministic transitions, we introduce the special predicate `choice`, where `choice(X, Y)` is a positive atom among the literals $L_1, \ldots, L_n$. Variable `X` must appear also in another positive atom among the L_is, and once we fix `X`, a single value for `Y` is chosen by picking it from the set of all values that can substitute `X`, so as to enforce a functional dependency $\texttt{X} \rightarrow \texttt{Y}$. Specifically, we adopt the semantics of [22] to interpret `choice`: `choice` is encoded via (a controlled form of) recursion through negation, and non-determinism is captured by the Stable Model Semantics of logic programs [12]. Example 2 below illustrates the use of `choice`.

2.2 The Network Model

A DDS relies on an underlying network $\mathcal{N}$ of communicating nodes, each running a D2C program. In distributed computing, $\mathcal{N}$ is typically represented as a graph (V, A), where V are the computation nodes, and the arcs in A reflect communication ability in the physical network. Directed arcs denote non-symmetric communication.

There is a complex spectrum of different network models, depending on the topology of the graph, the degree of mobility of nodes, and on which extent the network may vary over time. Here we consider an interesting widespread class of networks with *(i)* fixed network topology; *(ii)* bidirectional communication channels; *(iii)* strongly connected nodes; *(iv)* the number of neighbors that a node can have bounded by a constant κ; *(v)* the ability for every node to communicate with itself. This class of networks can be represented by fixed *undirected, connected graphs with bounded degree*, where *each node has a self-loop*. From now on, we always assume that the network is of this shape. To make nodes aware of their own name and that of their neighbors, each node has the following rules:

`my_name(M)` **if** **prev** `my_name(M)`. `neighbor(N)` **if** **prev** `neighbor(N)`.

This information is read-only, so no other rule has `my_name` and `neighbor` in its head.

Example 2. The following D2C program creates a *rooted spanning tree* in a network.

```
parent(P)  if  join@X, choice(X, P), prev not parent(_).
parent(P)  if  prev parent(P).
  join@N   if  parent(P),  neighbor(N), prev not parent(_).
```

The first rule says that when a node receives a join request for the first time, it picks the sender as a parent. `choice` is used to handle the case where multiple `join` requests are received simultaneously; in this case, the node non-deterministically selects one of the senders as parent. The second rule is an inertial rule to keep the tree unchanged. The last rule is recursive and propagates the join request to its neighbors. Even though termination is not explicitly detected, in a reliable order-preserving network (cf. Section 2.3) the root can safely start using the tree as soon as it sends the join tuples. ■

2.3 DDS Formal Model, Execution Semantics

We now formalize DDSs, adopting *homogeneous* nodes, i.e., all running the same program and with the same local DB schemas. Still, over time the behavior of different nodes might diverge, depending on: *(i)* the location in the network, *(ii)* nondeterministic choices, and *(iii)* the data obtained from the external world.

A DDS *system* $\mathcal{M}$ is a tuple $\langle \mathcal{N}, \mathcal{I}, \mathcal{T}, \mathcal{S}, \mathcal{P}, D_0 \rangle$, where:

- $\mathcal{N}$ is the network graph (see Section 2.2);
- $\mathcal{I}$, $\mathcal{T}$, and $\mathcal{S}$ denote the input, transport, and state schemas of every node in $\mathcal{M}$;
- $\mathcal{P}$ is the D2C program run by every node in $\mathcal{M}$;
- D_0 is a local state DB over $\mathcal{S}$, which represents the initial state of each node, without considering instances of `my_name` and `neighbor`.

As common for dynamic systems over relational data, the execution semantics of a DDS is given in terms of a *relational transition system (RTS)*, i.e., a transition system whose states are labeled by DBs [23]. In the following, Δ denotes a *countably infinite data domain* of constants/values, including also node names. Technically, an RTS Υ is a tuple $\langle \Delta, \mathcal{R}, \Sigma, \sigma_0, db, \Rightarrow \rangle$, where $\mathcal{R}$ is a DB schema, Σ a set of states, and $\sigma_0 \in \Sigma$ the initial state; db is a function that, given a state $\sigma \in \Sigma$, returns a DB instance conforming to $\mathcal{R}$ and made up of values in Δ; and $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between states. Υ is *serial* if, for each $\sigma \in \Sigma$, there is $\sigma' \in \Sigma$ s.t. $\sigma \Rightarrow \sigma'$.

Building an RTS from a DDS $\mathcal{M}$ poses three main challenges, resulting in variants of verification: *(i)* how to construct a global view of $\mathcal{M}$ from the local node DBs; *(ii)* which communication model $\mathcal{M}$ adopts; *(iii)* how $\mathcal{M}$ interacts with the environment.

Global View. To obtain a global view of $\mathcal{M} = \langle \mathcal{N}, \mathcal{I}, \mathcal{T}, \mathcal{S}, \mathcal{P}, D_0 \rangle$, we group all local DBs into a unique, global DB. To do so, we obtain schemas $\mathcal{I}_G$, $\mathcal{S}_G$, and $\mathcal{T}_G$ from the corresponding local ones, by adding in the first position of each relation an extra argument, keeping track of the node to which a relation tuple belongs. This allows us to define the *initial global DB* $D_{0,\mathcal{M}}$ of $\mathcal{M}$, as the global DB over $\mathcal{S}_G$ where each node stores D_0, and the information about its own name and that of its neighbors, according to $\mathcal{N} = \langle V, A \rangle$. Specifically $D_{0,\mathcal{M}} = \{p(\mathtt{n}, \mathtt{t}) \mid p(\mathtt{t}) \in D_0,\ \mathtt{n} \in V\} \cup \{\mathtt{my\_name}(\mathtt{n}, \mathtt{n}) \mid \mathtt{n} \in V\} \cup \{\mathtt{neighbor}(\mathtt{n}, \mathtt{n}') \mid (\mathtt{n}, \mathtt{n}') \in A\}$. We denote by $\Delta_{0,\mathcal{M}} \subset \Delta$ the set of values explicitly mentioned in $D_{0,\mathcal{M}}$, i.e., its *active domain*.

Also the local D2C program $\mathcal{P}$ can be turned into a global program $\mathcal{P}_G$, by: *(i)* introducing an additional node variable S in every predicate of $\mathcal{P}$, and *(ii)* adding atom $\mathtt{my\_name}(\mathtt{S}, \mathtt{S})$ to the body of each rule (so as to "bind" S). We denote by $[\mathcal{P}_G]$ the Datalog program obtained from $\mathcal{P}_G$ via the procedure in Section 2.1.

Communication Model. A communication model specifies how messages are exchanged by nodes, i.e., how output transport tuples are delivered to the destination. We assume *reliable* communication, i.e., messages are never lost. In a given computation step, we also assume that for each pair of neighbor nodes `s` and `d`, `s` concretely sends at most one message to `d`. The actual data payload of such messages consists of all transport tuples produced by `s` with destination `d`.

We consider here *asynchronous* communication, which decouples senders from receivers, as well as message exchanges: each message is delivered independently from the others, without any guarantee about "when" it will be received. However, we assume that all messages sent from a node to another node are delivered in the *same relative*

order. Hence, asynchronous communication can be abstractly captured by equipping every node with one *message queue* per neighbor. The DDS evolution is then captured by iterating through these steps: *(i)* nondeterministically pick a non-empty queue, extracting its front message M; *(ii)* the destination node performs a computation step triggered by M, possibly considering external input data; *(iii)* the node state is updated and the produced messages are inserted in the corresponding destination queues.

Environment Interaction. In addition to considering *closed* DDS, which do not receive any external input, we distinguish two modes:

(i) *autonomous* DDS (aDDS): nodes may receive an input DB only at startup, and then the input remains fixed throughout the execution;

(ii) *interactive* DDS (iDDS): nodes may receive a new input DB when they are active and are about to process an incoming message.

The coupling of input and transport DBs is w.l.o.g., since nodes may send dummy messages to themselves just to process a new input.

Asynchronous, Interactive Execution Semantics. We show how to build the RTS for a DDS $\mathcal{M} = \langle \mathcal{N}, \mathcal{I}, \mathcal{T}, \mathcal{S}, \mathcal{P}, D_0 \rangle$ with network $\mathcal{N} = \langle V, A \rangle$. Given a global DB D over $\mathcal{T}_G \uplus \mathcal{S}_G$, $transp(D)$ and $state(D)$ denote the projections of D over transport and state relations, respectively. As pointed out before, nodes are conceptually equipped with queues for incoming messages, whose relational representation obeys the special (global) schema $\mathcal{Q}_G$. We abstract away from the concrete relational representation of queues. Given a global queue DB Q over $\mathcal{Q}_G$ and a pair of neighbor nodes $\mathtt{s}, \mathtt{d} \in V$, notation $in^{Q}_{\mathtt{s}\to\mathtt{d}} \subseteq Q$ denotes the portion of Q that refers to the queue used by $\mathtt{d}$ to store messages from $\mathtt{s}$.[4] Given a global DB D over $\mathcal{T}_G \uplus \mathcal{S}_G \uplus \mathcal{Q}_G$, $queues(D)$ denotes the projection of D that maintains only the queue relations. Then, given a global DB D over $\mathcal{S}_G \uplus \mathcal{T}_G \uplus \mathcal{Q}_G$, we introduce the following preliminary notions:

– The *enqueue result of* D, written $enqueueMsg_{\mathcal{M}}(D)$, is the global queue DB over $\mathcal{Q}_G$ that results from the queues in D by incorporating all the output transport tuples into the corresponding input queues:

$$enqueueMsg_{\mathcal{M}}(D) = \big\{ in^{D}_{\mathtt{s}\to\mathtt{d}}.enqueue(m_{\mathtt{s}\to\mathtt{d}}) \mid (\mathtt{s},\mathtt{d}) \in A, \text{ and } m_{\mathtt{s}\to\mathtt{d}} = \{\mathtt{p}(\mathtt{x}) \mid \mathtt{s\_p}(\mathtt{s},\mathtt{d},\mathtt{x}) \in D\}\big\}$$

– The *active nodes* in D are those nodes that have at least a non-empty queue in D:

$$active_{\mathcal{M}}(D) = \{\mathtt{d} \mid \text{there exists } \mathtt{s} \text{ such that } (\mathtt{s},\mathtt{d}) \in A \text{ and } \neg(in^{D}_{\mathtt{s}\to\mathtt{d}}.isEmpty())\}$$

– Given a node $\mathtt{d} \in active_{\mathcal{M}}(D)$ and a neighbor $\mathtt{s}$ of $\mathtt{d}$ s.t. $\neg(in^{D}_{\mathtt{s}\to\mathtt{d}}.isEmpty())$, the *dequeue result of* D *w.r.t.* $\mathtt{s}$ *and* $\mathtt{d}$, written $dequeueMsg^{\mathtt{s}\to\mathtt{d}}_{\mathcal{M}}(D)$, is a pair $\langle Q_{new}, T \rangle$, where, given $\langle newin_{\mathtt{s}\to\mathtt{d}}, m_{\mathtt{s}\to\mathtt{d}} \rangle = in^{D}_{\mathtt{s}\to\mathtt{d}}.dequeue()$:

- Q_{new} is the global queue DB obtained by extracting the front message from queue $in^{D}_{\mathtt{s}\to\mathtt{d}}$, while maintaining all other queues unaltered:

$$Q_{new} = (queues(D) \setminus in^{D}_{\mathtt{s}\to\mathtt{d}}) \cup newin_{\mathtt{s}\to\mathtt{d}}$$

- T is the global transport DB obtained by transforming the tuples contained inside message $m_{\mathtt{s}\to\mathtt{d}}$ into corresponding input transport tuples for $\mathtt{d}$:

$$T = \{\mathtt{r\_p}(\mathtt{d},\mathtt{s},\mathtt{x}) \mid \mathtt{p}(\mathtt{x}) \in m_{\mathtt{s}\to\mathtt{d}}\}$$

[4] Recall that every node is connected to itself, hence there is a special queue where each node stores the messages sent to itself.

Let $\mathbb{Q}_G$ be the set of possible DB instances over $\mathcal{Q}_G$. To formalize the asynchronous execution semantics, we introduce a relation C-STEP$_\mathcal{M} \subseteq \mathbb{S}_G \times \mathbb{T}_G \times \mathbb{Q}_G \times V \times \mathbb{I}_G \times \mathbb{S}_G \times \mathbb{T}_G \times \mathbb{Q}_G$ that substantiates the generic state transition mapping introduced in Section 2.1 by adding the current and next queue state, as well as the single node asynchronously triggering the transition. In particular, given two global DBs D, D_{new} over $\mathcal{S}_G \uplus \mathcal{T}_G \uplus \mathcal{I}_G$, a node $\mathtt{n} \in V$, and a global input DB I over $\mathcal{I}_G$, we have that $\langle D, \mathtt{n}, I, D_{new}\rangle \in$ C-STEP$_\mathcal{M}$ if and only if $\mathtt{n} \in \mathit{active}_\mathcal{M}(D)$, and there exists a neighbor node $\mathtt{s}$ of $\mathtt{n}$ with $\neg \mathit{in}^D_{\mathtt{s}\to\mathtt{n}}.\mathit{isEmpty}()$ and a stable model $\mathfrak{M}$ for the program $[\mathcal{P}_G] \cup [D'] \cup T \cup I|_\mathtt{n}$ such that $D_{new} = \mathit{state}(\mathfrak{M}) \cup \mathit{transp}(\mathfrak{M}) \cup \mathit{enqueueMsg}_\mathcal{M}(\mathfrak{M} \cup Q_n)$, where $\langle Q_n, T\rangle = \mathit{dequeueMsg}^{\mathtt{s}\to\mathtt{n}}_\mathcal{M}(D)$. Finally, we define the *asynchronous transition system* of $\mathcal{M}$, written $\Upsilon^{\text{int}}_\mathcal{M}$, as the RTS $\langle \Delta, \mathcal{R}, \Sigma, \sigma_0, db, \Rightarrow\rangle$, where:

- $\mathcal{R} = \mathcal{S}_G \uplus \mathcal{T}_G \uplus \mathcal{Q}_G$; $\quad \Delta_{0,\mathcal{M}} \subset \Delta$;
- $db(\sigma_0) = D_{0,\mathcal{M}} \cup \bigcup_{\mathtt{n}\in V} \mathit{in}^\emptyset_{\mathtt{n}\to\mathtt{n}}.\mathit{enqueue}(\mathtt{start}())$, where $\mathtt{start} \in \mathcal{T}_G$ is considered as a special, 0-ary transport tuple that is initially inserted in every self-queue of the system, so as to trigger the first computation step of the corresponding node;
- Σ and $\Rightarrow$ are defined by simultaneous induction as the smallest sets s.t. $\sigma_0 \in \Sigma$ and
  1. given $\sigma \in \Sigma$ s.t. $db(\sigma) = D$ and $\mathit{active}_\mathcal{M}(D) \neq \emptyset$, for every global DB D_{new} over $\mathcal{S}_G \uplus \mathcal{T}_G$, every node $\mathtt{n} \in V$, and every global input DB I over $\mathcal{I}_G$, if $\langle D, \mathtt{n}, I, D_{new}\rangle \in$ C-STEP$_\mathcal{M}$ then $\sigma_{new} \in \Sigma$ and $\sigma \Rightarrow \sigma_{new}$, with $db(\sigma_{new}){=}\hat{D}$;
  2. given $\sigma \in \Sigma$ s.t. $db(\sigma) = D$ and $\mathit{active}_\mathcal{M}(D) = \emptyset$ (i.e., the system is quiescent), we have $\sigma \Rightarrow \sigma$.

Observe that, also in this case, $\Rightarrow$ is guaranteed to be serial.

Execution Semantics for Autonomous DDSs. In the case of autonomous DDSs, the execution semantics needs to be modified so as to take into account that the input is provided only at the beginning of the computation, and then stays rigid over time. Clearly, each computation may be associated with a different initial input. We can then imagine that the execution semantics is given, in this case, by an infinite set of RTSs, each associated to a different global input DB. Given a DDS $\mathcal{M}$, we write $[\![\Upsilon^{\text{aut}}_\mathcal{M}]\!]$ to denote the set of RTSs obtained by variying the rigid input DB under the autonomous semantics.

3 Static Analysis of DDS

To specify interesting properties over DDSs, we need a logic that captures: *(i)* the DDS dynamics, *(ii)* queries over the data stored in the node DBs, and *(iii)* conditions about the evolution of such data over time. The first requirement calls for temporal logics, the second for FO logic, and the third for their combination, where FO quantification interacts with temporal modalities. Several logics have been introduced for that purpose (see, e.g., [10,6]). Here we rely on *FO-CTL* in [8], which combines FOL under the active domain semantics with CTL, fully supporting FO quantification across states. Given schema $\mathcal{R}$ and a finite set Δ_0 of values, the syntax of *FO-CTL* is:

$$\Phi ::= Q \mid \exists x.\Phi \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \mathsf{EX}\Phi \mid \mathsf{E}(\Phi_1 \mathsf{U} \Phi_2) \mid \mathsf{A}(\Phi_1 \mathsf{U} \Phi_2)$$

where Q is a FO formula over $\mathcal{R}$ and Δ_0. *FO-CTL* formulae are interpreted over a (serial) RTS relative to a state. We refer to [8] for the full semantics of *FO-CTL*, and just recall that E stands for "existence of a path", A for "all paths", X for "next state", and U for "until".

Example 3. Given the standard abbreviation $\mathsf{AG}\Phi = \neg\mathsf{E}(\mathsf{true}\mathsf{U}\neg\Phi)$, the formula $\mathsf{AG}(\forall n, p.\mathtt{Parent}(n,p) \rightarrow \mathsf{AG}\ \mathtt{Parent}(n,p))$ expresses that `Parent` is rigid: whenever n stores that p is its parent, then it will keep this information forever. This property is satisfied by any DDS using the D2C program of Example 2. □

Given a DDS $\mathcal{M}$, we assume, without loss of generality, that the finite set Δ_0 over which Φ is defined coincides with $\Delta_{0,\mathcal{M}}$. Given a closed *FO-CTL* property Φ, verifying whether Φ holds in $\mathcal{M}$ gives rise to a different model checking problems, each one obtained by fixing a certain execution semantics for $\mathcal{M}$. Specifically, for aDDSs it is interesting to study whether the DDS satisfies a given *FO-CTL* property *independently from the given input* (that is, for every possible input DB provided at the system startup). This verification problem resembles the classical verification problem for relational transducers and their distributed variants [10,11]. We consequently obtain that:

- $\mathcal{M}$ verifies Φ under the interactive semantics, if $\Upsilon^{\text{int}}_{\mathcal{M}} \models \Phi$;
- $\mathcal{M}$ verifies Φ under the autonomous semantics, if $\Upsilon \models \Phi$ for every RTS $\Upsilon \in [\![\Upsilon^{\text{aut}}_{\mathcal{M}}]\!]$.

Convergence Properties. Convergence is a generalization of termination for DDSs, indicating that the distributed computation run by the whole DDS eventually reaches a stable situation where all nodes are quiescent, i.e., do not change anymore their DBs. This typically occurs when no messages are exchanged. However, we want to rule out those cases in which quiescence is reached because one or more nodes stop their computation due to an error (e.g., because the node has received an unexpected message/input, or has entered an undesired state). We assume that a node declares that it is *faulty* by inserting the special flag `error` in its state. This, in turn, corresponds to a global unary relation, where `error(n)` indicates that node `n` is faulty. Under this assumption, D2C programs employ `error` to indicate under which circumstances a node becomes faulty.

Convergence properties are then defined by mixing two dimensions: *(i)* number of faulty nodes in a run, with the two extreme cases of *global* (no faulty node) vs *partial* (at least one non-faulty node) correctness; *(ii)* quantification over runs, considering the case in which the DDS *sometimes* (i.e., for at least one run) vs *always* (i.e., for all runs) converges. Given a DDS $\mathcal{M}$, this gives rise to four variants of convergence:

- $\mathcal{M}$ *sometimes converges when totally/partially correct* if there is a run of $\mathcal{M}$ reaching a state where all nodes are quiescent, and all are/at least one is not faulty.
- $\mathcal{M}$ *always converges when totally/partially correct* if every run of $\mathcal{M}$ in which all nodes stay/at least one node stays non-faulty eventually converges.

All four convergence properties can be formalized in *FO-CTL*. Moreover, in *FO-CTL* we can model variants of convergence properties that better reflect the conditions under which the DDS is desired to converge. In this way, checking convergence is reduced to *FO-CTL* model checking, for which well known techniques are available.

4 Verification of DDSs

We investigate the decidability frontier of verification over DDSs. To stress the tightness of our results, we establish undecidability for convergence properties, and decidability for full *FO-CTL*. While input, state, and transport schemas are fixed in advance, the extension of such relations is not, and could grow unboundedly over time. If no bound

Table 1: Summary of decidability and complexity results for verification of DDSs

Type of DDS	Input bounded	State/transport bounded		
		N/Y	Y/N	Y/Y
Interactive	N	undecidable	undecidable	PSPACE-complete
	Y	undecidable	undecidable	PSPACE-complete
Autonomous	N	undecidable	undecidable	undecidable
	Y	not applicable	not applicable	PSPACE-complete
Closed	not applicable	not applicable	not applicable	PSPACE-complete

is imposed on the data manipulated by the DDS, verification of convergence properties (and even propositional reachability) is undecidable even for a single-node DDS [6][5].

Building on the notion of *state-boundedness* [8,6,7], we study how decidability is affected when we impose a (pre-defined) bound on the different information sources of the DDS. Specifically, we say that a DDS $\mathcal{M}$ is *input-bounded* it there is a bound on the number of values that can appear in each single input DB. In this case, unboundedly many values can still be input over time, provided they do not accumulate in a single computation step. The notions of *state-boundedness*, *transport-boundedness*, and (for asynchronous DDSs) *queue-boundedness* are analogous. A bound independent from the network size fits with the idea that the programs run by nodes are not tailored to a specific network. Also, state-boundedness does not interfere with the information each node has about its neighbors, since our networks are neighbor-bounded (cf. Section 2.2).

In our analysis, a key observation is that the notion of *uniformity* [8] (corresponding to *genericity* in databases) can be straightforwardly recast for DDSs. Intuitively, uniformity states that the dynamics of a DDS $\mathcal{M}$ are invariant under permutation of values in the node DBs, modulo the finite subset $\Delta_{0,\mathcal{M}}$ of Δ: the system exhibits the same behavior (modulo renaming of values) when nodes compute over isomorphic DBs. Making use of uniformity, we are able to show the upper bounds in Table 1. Undecidability is proved via reductions from the halting problem for 2-counter machines.

5 Conclusions

In the wide spectrum of declarative distributed computing, we have studied verification in the important case of reliable communication with order-preserving asynchronous strategies. We plan to build on our foundational results to implement verification techniques for DDSs, relying on existing ASP techniques for D2C, and in particular on the implementation in [14], which can simulate runs of DDSs according to our formalization. We also want to study how to extend our results to networks whose topology can change. Building on the approach in [19], we can deal with the case of an overall bounded number of nodes, while the case where unboundedly many computation nodes can be created is left for interesting future work.

Acknowledgements. This work is supported by the Spanish Ministry of Economy and Competitiveness under grants MDM-2015-0502 and TIN2016-81032-P, and by the REKAP project funded by unibz through the 2017 research budget.

[5] Observe that, in the single-node case, partial and global correctness, coincide.

References

1. S. Abiteboul, M. Bienvenu, A. Galland, and É. Antoine. A rule-based language for web data management. In *Proc. of PODS*, pages 293–304. ACM Press, 2011.
2. P. Alvaro, T. J. Ameloot, J. M. Hellerstein, W. Marczak, and J. Van den Bussche. A declarative semantics for Dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley, 2011.
3. T. J. Ameloot. Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. *SIGMOD Record*, 43(2):5–16, 2014.
4. T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *JACM*, 60(2):15:1–15:38, 2013.
5. T. J. Ameloot, J. Van den Bussche, W. R. Marczak, P. Alvaro, and J. M. Hellerstein. Putting logic-based distributed systems on stable grounds. CoRR Technical Report abs/1507.05539, arXiv.org e-Print archive, 2015. Available at `http://arxiv.org/abs/1507.05539`.
6. B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *Proc. of PODS*, 2013.
7. B. Bagheri Hariri, D. Calvanese, M. Montali, and A. Deutsch. State-boundedness in data-aware dynamic systems. In *Proc. of KR*, pages 458–467. AAAI Press, 2014.
8. F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of agent-based artifact systems. *JAIR*, 51:333–376, 2014.
9. D. Calvanese, G. De Giacomo, and M. Montali. Foundations of data aware process analysis: A database theory perspective. In *Proc. of PODS*, pages 1–12, 2013.
10. A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *JCSS*, 73(3):442–474, 2007.
11. A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven web services. In *Proc. of PODS*, pages 90–99, 2006.
12. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of ICLP*, pages 1070–1080. The MIT Press, 1988.
13. J. M. Hellerstein. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.
14. J. Lobo, D. Wood, D. Verma, and S. Calo. Distributed state machines: A declarative framework for the management of distributed systems. In *Proc. of CNSM*, pages 224–228, 2012.
15. 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*, 52(11):87–95, 2009.
16. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *Operating Systems Rev.*, 39(5):75–90, 2005.
17. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
18. J. Ma, F. Le, D. Wood, A. Russo, and J. Lobo. A declarative approach to distributed computing: Specification, execution and analysis. *TPLP*, 13:815–830, 2013.
19. M. Montali, D. Calvanese, and G. De Giacomo. Verification of data-aware commitment-based multiagent systems. In *Proc. of AAMAS*, pages 157–164, 2014.
20. V. Nigam, L. Jia, B. T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. *Computer Languages, Systems & Structures*, 38(2):158–180, 2012.
21. 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. *Computer Communication Rev.*, 41(4):440–441, 2011.
22. D. Sacca and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. of PODS*, pages 205–217, 1990.
23. M. Y. Vardi. Model checking for database theoreticians. In *Proc. of ICDT*, volume 3363 of *LNCS*, pages 1–16. Springer, 2005.