Convergence Verification of Declarative Distributed Systems*

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

[1] Free University of Bozen-Bolzano
[2] University of Pompeu Fabra

Abstract. Logic-based languages, such as Datalog and Answer Set Programming, have been recently put forward as a data-centric model to effectively specify and implement network services and protocols, seeing them as dynamic systems of distributed computational nodes where each node evolves an internal database and exchanges data with the other nodes of the network. This approach provides the basis for declarative distributed computing. However, a rigorous, comprehensive characterization of the decidability and complexity of verification in declarative distributed systems is yet to come. This paper charts the decidability border of the verification of convergence properties, considering the case where the network is a fixed connected graph, nodes can incorporate fresh data from the external world into the system, and can communicate asynchronously by means of reliable but unordered channels.

1 Introduction

In the past years we have seen how declarative database query languages, such as Datalog, can naturally be used to specify and implement network services and protocols [19]. The approach, referred to as *declarative networking* [5], makes the specifications of complex network protocols concise, intuitive, and directly executable through distributed query processing algorithms [25]. The compilation of the rules constituting the specification into actual implementations performs well when compared with imperative C/C++ implementations of the same protocols [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 for security and provenance in distributed query processing [29,30], in the analysis of asynchronous event systems [2], and as the core of the Webdam language for distributed Web applications [3]. We refer to these systems as *declarative distributed systems* (DDSs).

There are several variants of concrete languages for specifying DDSs [20,4,3,22], but their common denominator is *data-centricity*: computations in a single node are limited to evaluations of queries on a relational database (DB), 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 [26,22]. However, in spite of several studies on the foundations of DDSs [17,8], *a rigorous, comprehensive characterization of the decidability and complexity of verification in such systems, is yet to come.* On the one hand, verification techniques and tools for DDSs have been exploited in various settings, but providing only an empirical/experimental assessment [26,22,12,13]. On the other hand, formal models for DDSs have been mainly developed to study their ability to compute queries in a distributed manner, and not to assess their temporal/dynamic evolution [8,7,6]. Instead, in this paper we provide a step towards a formal, systematic characterization of verification of DDSs by focusing on convergence properties. 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 on the notion of *state-boundedness* [11,9], and detail the decidability frontier of verification when a bound is imposed on the system data-sources.

In the next Section 2 we introduce the DDS model, provide its execution semantics, and define the verification problem over convergence properties. In Section 3 we chart the decidability boundary of verification of convergence. Finally, Section 4 provides the conclusions.

2 Declarative Distributed Systems

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 [21]. We assume familiarity with Datalog and the stable model semantics [15] and adopt the standard conventions: variables are denoted with uppercase letters, constants by lowercase letters, and '_' is used as a placeholder for an anonymous variable (i.e., one that does not appear elsewhere in the rule).

2.1 Computational Model in a Node

A state in the state machine of a node is represented by a (relational) DB. As customary, a DB defines a set of relations conforming to a given schema. A DB schema is a finite set of relation schemas R/n, with a relation name R and an arity n, representing the number of components (or attributes) it contains. By a slight abuse of notation, sometimes we drop the arity and interchangeably use the same symbol to indicate the relation schema and its name. We fix a *countable data domain* Δ denoting an infinite set of constants. An instance of a relation schema R/n is a finite set of n-tuples over Δ, and a DB is the union of relation instances over its schema. Given a DB instance I, we denote by $\textsc{adom}(I)$ the *active domain* of I, that is the (finite) set of constants explicitly appearing in I.

State transitions occur when the node receives inputs, in the form of DBs, from external components (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 fact by fact. Thus, every node has an *input* schema $\mathcal{I}$, a *state* schema $\mathcal{S}$, and a *transport* schema $\mathcal{T}$. The latter is 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 DB I, the transition results in a new pair (S_n, T_n) of state and transport DBs.

Specifically, that mapping is specified by means of D2C, a declarative programming language introduced in [22], which is an extension of plain Datalog, enhanced with process communication and state changes capabilities. 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 in the language used to identify nodes (concretely, such domain would correspond to objects such as IP addresses or URLs). 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. Notice that the literals in the **prev** scope do not concur to determine stratification, since they are over a fixed database already computed in the previous computation step. Also annotated predicates are not involved in stratification, since they address incoming messages received in the form of an extensional DB fact. The meaning of an annotation depends on whether it appears in the head or the body of a rule: in the former case, it indicates the destination of the transport tuple, in the latter it points to the source.

[3] We can relax this requirement for negated atoms containing anonymous variables. Consider in particular a negated atom **not** $P(\vec{X}, \_, \ldots, \_)$ appearing in a rule where variables $\vec{X}$ also appear in positive atoms. Such an atom can be replaced by **not** $N(\vec{X})$, where N is a fresh predicate, provided we introduce the additional rule $N(\vec{X}) \ \ \textbf{if} \ \ P(\vec{X}, \_, \ldots, \_)$.

Example 1. Consider the input predicate `echo/2`, state predicate `alive/1`, and transport predicate `say/1`. Rule `say(M)@D` **if** `echo(M,D)` models that the node sends `m` to the node with ID `d` (if it is a node neighbor) when the corresponding `echo(m,d)` fact is given as input. `M` and `D` are variables, which in this case are bound to constants `m` and `d` respectively. Rule `alive(S)` **if** `say(M)@S` models that the node adds to its state the information that node `s` is alive, whenever the transport tuple `say(m)` is received from node `s`. Rule `say(M)@Src` **if** `say(M)@Src` "echoes" the `say` tuple back to the source node. □

To support also non-deterministic transitions, D2C features the special predicate `choice` by Saccà and Zaniolo [27], where `choice(X,Y)` is a positive atom among the literals $L_1, \ldots, L_n$. Variables `X` and `Y` 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 inside the set of all values that can substitute it, so as to enforce a functional dependency `X` $\longrightarrow$ `Y`. Also the variant `choice(Y)` is admitted, to enforce a functional dependency with trivial domain. It is known that the data complexity of finding a stable model of a Datalog program with `choice` remains polynomial [16].

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 $\langle V, A \rangle$, where V are the computation nodes, and the arcs in A reflect the ability to communicate according to the physical network configuration. 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 networks with *(i)* fixed topology; *(ii)* bidirectional communication channels; *(iii)* strongly connected nodes; *(iv)* the ability for every node to communicate with itself. This class of networks can be represented by *fixed undirected, connected graphs*, where *each node has a self-loop*. From now on, we always assume that the network is in this class. 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 can use `my_name` and `neighbor` in its head.

2.3 DDS Formal Model

We now formalize DDSs adopting *homogeneous* nodes, i.e., all nodes run the same program. This also means that the local DB schemas are the same for all nodes.

Still, the behavior of different nodes might diverge over time, depending on: *(i)* their location in the network, *(ii)* the presence of non-deterministic choices in the program, and *(iii)* the data obtained from the interaction with other nodes and with the external world. A DDS $\mathcal{M}$ is a tuple $\langle \mathcal{N}, \mathcal{I}, \mathcal{T}, \mathcal{S}, \mathcal{P}, D_0 \rangle$, where:

- $\mathcal{N}$ is the network graph (obeying to the assumptions of Section 2.2);
- $\mathcal{I}$, $\mathcal{T}$, and $\mathcal{S}$ respectively 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 $\mathcal{S}$ representing the initial state of each node, and so that it assigns no tuples to `my_name` and `neighbor` (these are in fact implicitly set differently for each node, depending on the network topology); its active domain is denoted by Δ_0, while its extension with node names is denoted by $\Delta_{0,\mathcal{M}}$

Commonly, the execution semantics of dynamic systems over relational data is given in terms of a *relational transition system (RTS)*, i.e., a transition system where each state is labeled by a DB, representing the configuration of the current memory of the system in that state [28]. However, in our distributed setting, such a global memory should account for two peculiar local aspects. On the one hand, it needs to store the current local configuration of nodes in the network, and so it contains one DB per node. On the other hand, it must track the state of the network in terms of messages being exchanged (and not yet processed by their recipient nodes). The representation of this latter aspect depends on the chosen communication model and, in turn, how communication channels operate. Hence, we represent communication channels using a generic data structure $\mathfrak{C}_{\mathcal{T}}$ defined over the transport schema $\mathcal{T}$. Given a pair of connected nodes i and j in the network, each local channel configuration stores the state of the communication channel linking i and j, by instantiating $\mathfrak{C}_{\mathcal{T}}$ to store the pending messages present in the network that have been sent by i and have not yet been received/processed by j. We denote by $\mathbb{C}$ the set of all possible instantiations of $\mathfrak{C}_{\mathcal{T}}$. Once the communication model is fixed, this abstract data structure is grounded into a specific one, which must suitably reflect the functioning of the communication mode in terms of ordering and reliability. For example, $\texttt{queue}_{\mathcal{T}}$ indicates an order-preserving, reliable communication channel over pairs of nodes, while $\texttt{multiset}_{\mathcal{T}}$ represents an unordered, reliable communication channel where messages sent in a given order may be processed in reverse.

To account for those local aspects, we reorganize the global memory into many local ones attached to nodes and channels. Technically, given a DDS $\langle \mathcal{N}, \mathcal{I}, \mathcal{T}, \mathcal{S}, \mathcal{P}, D_0 \rangle$ with $\mathcal{N} = \langle V, A \rangle$, and given a data structure $\mathfrak{C}_{\mathcal{T}}$ over $\mathcal{T}$ for communication, the execution semantics of the DDS under $\mathfrak{C}_{\mathcal{T}}$ is given via a so-called *distributed RTS* (DRTS) Υ of the form $\langle \mathcal{N}, \Delta, \mathcal{S}, \mathcal{T}, \Sigma, \sigma_0, ndb, nch, \Rightarrow \rangle$, where:

- Σ is a (possibly infinite) set of states;
- $\sigma_0 \in \Sigma$ is the initial state;
- ndb is a function that, given a state $\sigma \in \Sigma$ and a node $\texttt{n} \in V$, returns a corresponding DB instance of $\mathcal{S}$ over Δ for $\texttt{n}$;

- *nch* is a function that, given a state $\sigma \in \Sigma$ and two nodes $\mathtt{n}_1, \mathtt{n}_2 \in V$ such that $\langle \mathtt{n}_1, \mathtt{n}_2 \rangle \in A$, returns an instance of $\mathfrak{C}_{\mathcal{T}}$ storing the pending messages from $\mathtt{n}_1$ to $\mathtt{n}_2$;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between states.

Υ is *serial* if, for every state $\sigma \in \Sigma$, there exists $\sigma' \in \Sigma$ such that $\sigma \Rightarrow \sigma'$. Notice that the input schema and databases are not mentioned in the general *DRTS* definition, but will be used later to define concrete transition relations. Moreover, in case the instantiations of $\mathfrak{C}_{\mathcal{T}}$ can be represented by means of a DB, it is possible to boil down a DRTS to a standard RTS by labeling states with the disjoint union of all the state and channel DBs mentioned in the DRTS states.

To build a DRTS capturing the execution semantics of a DDS $\mathcal{M}$, one has to choose which *communication* model is used by $\mathcal{M}$ to handle the exchange of messages in the network, and how does $\mathcal{M}$ *interact with the external users* that exchange data with the computation nodes. Specifically, we consider the relevant setting where communication is *reliable* and *asynchronous*, i.e., messages are never lost and message exchanges occur independently from each other. Consistently with the fact that each transport atom in a D2C program is attached to a sender/receiver, each message consists of a single transport fact.

As for user interaction, nodes may receive a new input DB when they start processing an incoming message. We call DDSs obeying to that input-policy *interactive* DDS*s* (iDDSs). Coupling the processing of input DBs with that of incoming messages is without loss of generality, since a node may send dummy messages to itself just to signal that it is ready to process a user input.

2.4 Execution Semantics

Due to the aforementioned asynchronicity and reliability, we can assume that, at each computation step, only one node reacts to the delivery of an incoming message. Within the computed result, there may be transport facts labeled with corresponding destination nodes; these are all simultaneously emitted, and will be asynchronously received by their respective recipient nodes fact by fact. Since we assume no guarantees on the order in which messages are received, communication can be abstractly captured by equipping each node with one *message multiset* per neighbor. Hence, from now on, we always assume that the data structure for communication channel is `multiset`, and make use of the usual operators to manipulate and inspect multisets. Relying on multisets instead of sets is important to capture the fact that two distinct messages exchanged between the same two nodes and carrying exactly the same payload may be both on their way to the recipient node.

The DDS evolution is then captured by iterating through these steps:

- a non-empty multiset is non-deterministically picked, non-deterministically extracting a message M.
- the destination node performs a computation step triggered by M, possibly considering also external input data;

- the node state is updated and the produced messages are inserted in the corresponding destination multiset.

Formally, given a program P, an input DB I, a previous state DB S, and a labeled transport tuple $t@n$, we denote by $\mathit{state}(P, I, S, t@n)$ the new state database computed by the program P over $S \cup I \cup \{t@n\}$, by $\mathit{transp}(P, I, S, t@n, d)$ the set of computed output tuples (over the trasnport signature) labeled by $@d$, and by $\mathit{transp} \downarrow (P, I, S, t@n, d)$ the set of tuples in $\mathit{transp}(P, I, S, t@n, d)$ where the label $@d$ has been dropped. Let $\mathcal{M} = \langle \mathcal{N}, \mathcal{I}, \mathcal{T}, \mathcal{S}, \mathcal{P}, D_0 \rangle$ be a DDS system whose network is $\mathcal{N} = \langle V, A \rangle$. To formalize the execution semantics, we introduce a relation C-STEP$_{\mathcal{M}}$ that substantiates the generic state transition mapping introduced in Section 2.1:

$$\text{C-STEP}_{\mathcal{M}} \subseteq \prod_{n \in V} \mathbb{S} \times \prod_{a \in A} \mathbb{C} \times A \times \mathbb{I} \times \prod_{n \in V} \mathbb{S} \times \prod_{a \in A} \mathbb{C}$$

Specifically, given $S, S' \in \prod_{n \in V} \mathbb{S}$, and $C, C' \in \prod_{n \in V} \mathbb{C}$, a channel $(\mathtt{s}, \mathtt{d}) \in A$, and an input database $I \in \mathbb{I}$, we have that $\langle S, C, (\mathtt{s}, \mathtt{d}), I, S', C' \rangle \in$ C-STEP$_{\mathcal{M}}$ if and only if there exists a message tuple $t \in C_{(\mathtt{s},\mathtt{d})}$ such that:

$$S'_n = \begin{cases} \mathit{state}(P, I, S, t@s) & \text{if } n = d \\ S_n & \text{otherwise} \end{cases}$$

$$C'_{(n,m)} = \begin{cases} C_{(s,d)} \setminus \{t\} & \text{if } n = s \text{ and } m = d \\ C_{(d,m)} \cup \mathit{transp} \downarrow (P, I, S_n, t@d, m) & \text{if } n = d \\ C_{(n,m)} & \text{otherwise} \end{cases}$$

Finally, we define the *transition system* of $\mathcal{M}$, written $\Upsilon^{\text{int}}_{\mathcal{M}}$, as the DRTS $\langle \mathcal{N}, \Delta, \mathcal{S}, \mathcal{T}, \Sigma, \sigma_0, \mathit{ndb}, \mathit{nch}, \Rightarrow \rangle$, where:

- $\Sigma \subset \prod_{n \in V} \mathbb{S} \times \prod_{a \in A} \mathbb{C}$ and, for each $\sigma \in \Sigma$ of the form $\sigma = ((S_n)_{n \in V}, ((C_c)_{c \in A}))$, $\mathit{ndb}(\sigma, n) = S_n$ and $\mathit{nch}(\sigma, s, d) = C_{(s,d)}$;
- $\sigma_0 = ((D_0)_{n \in V}, (C_c)_{c \in A})$, where $C_{(s,d)} = \emptyset$ if $s \neq d$ and $C(s, d) = \{\mathtt{start}\}$ otherwise, where `start` is a special 0-ary transport tuple used to guarantee at least one computation step to each node;
- The extensions of Σ and $\Rightarrow$ are defined by simultaneous induction as follows:
  1. $\sigma_0 \in \Sigma$;
  2. if $(S, C) \in \Sigma$, then, for each $\langle S, C, (\mathtt{s}, \mathtt{d}), I, S', C' \rangle \in$ C-STEP$_{\mathcal{M}}$, it is true that $(S', C') \in \Sigma$ and $(S, C) \Rightarrow (S', C')$.
  3. if $(S, C) \in \Sigma$ and, for each $c \in A$, $C_c = \emptyset$, then $(S, C) \Rightarrow (S, C)$.

Notice that $\Rightarrow$ is guaranteed to be serial.

2.5 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 state 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. 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 *total* (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 with total correctness* if there exists a run of $\mathcal{M}$ eventually reaching a state where all nodes are quiescent, and none is faulty.
- $\mathcal{M}$ *sometimes converges with partial correctness* if there exists a run of $\mathcal{M}$ eventually reaching a state where all nodes are quiescent, and at least one is not faulty.
- $\mathcal{M}$ *always converges with total correctness* if every run in which all nodes of $\mathcal{M}$ stay non-faulty eventually converges.
- $\mathcal{M}$ *always converges with partial correctness* if every run in which at least one node of $\mathcal{M}$ stays non-faulty eventually converges.

Moreover, convergence properties can be formulated in sophisticated logics that allow to specify properties over the data flowing in the DDS and the DDS temporal behavior. A suitable language over RTSs is *FO-CTL*, which mixes first-order (FO) logic with the computation three logic (CTL). It is possible to adapt that logic to DRTSs, called $DDS-CTL$. That can be achieved by exploiting some D2C rule to transfer the previous state configuration into a copy in the current one, using the verification formula to check that the two available state configurations are identical and, finally, using temporal operators to propagate that condition in time. Thus, the next undecidability results apply also to that DDS-CTL, but also the decidability proofs can be extended to address the full language.

3 Convergence Verification of IDDS

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 is imposed on the data manipulated by the DDS, verification of convergence properties is undecidable even for a single-node DDS [9].

Taking inspiration from the well-studied notion of *state-boundedness* [11,9,10], we hence study how decidability is affected when a (pre-defined and known) bound is imposed on the different information sources of the DDS. Specifically, given an integer b, we say that a DDS $\mathcal{M}$ is *input b-bounded* if the input DB is

constrained to mention at most b values during each single step, i.e., the cardinality of the input database active domain is always at most b. In this case, unboundedly many values can still be input over time, provided that they do not accumulate in a single computation step. When the fixed value b is understood or arbitrary, we simply talk about boundedness. We define *state b-bounded* (*state-bounded*) and *transport b-bounded* (*transport-bounded*) DDSs analogously. However, this time the constraints are a consequence of the combination of the DDS program and of the constraints on the input.

A bound independent from the network size fits with the idea that the declarative programs run by nodes are not tailored to a specific network. Moreover, state-boundedness does not interfere with the information each node has about its neighbors, since our networks are fixed, thus the relation `neighbor` is trivially bounded by the fixed number of nodes (cf. Section 2.2). In fact, the next decidability results deal with *network complexity*, i.e., the complexity of the global initial configuration projected over the `neighbor` and `my_name` predicates. Thus, the *data complexity* of the initial configuration incorporates the network complexity.

Additionally, a *channel b-bounded* (*channel-bounded*) condition imposes a similar constraint on the channel multiset cardinality, i.e., in any reachable configuration there are at most b facts laying on any channel. In channel-bounded DDSs the instantiations of the multiset channel data-structure can be represented by means of a finite DB, since the multiplicity of messages is bounded and a finite domain of constants suffice to represent them. Thus, in this case, the respective DRTSs can be translated into standard RTSs. We also say that the whole DDS is *bounded* if *all* of its information sources are so, i.e., it is input-, state-, transport-, and channel-bounded at the same time.

In our analysis, a key observation is that the notion of *uniformity* [11] can be straightforwardly recast for DDSs. Intuitively, uniformity (corresponding to *genericity* in databases) 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. Technically, we recast the notion of uniformity in [11] to the case of DDSs as follows. Given two DBs D_1, D_2 over schema $\mathcal{R}$ with $|\textsc{adom}(D_1)| = |\textsc{adom}(D_2)|$, we say that D_1 and D_2 are *isomorphic* if there exists a bijection $h : \Delta_1 \longrightarrow \Delta_2$, with $\textsc{adom}(D_1) \subseteq \Delta_1$ and $\textsc{adom}(D_2) \subseteq \Delta_2$, such that for every relation $R/n \in \mathcal{R}$ and every fact $R(\mathtt{d}_1, \ldots, \mathtt{d}_n) \in D_1$, we have $R(h(\mathtt{d}_1), \ldots, h(\mathtt{d}_n)) \in D_2$. With some abuse of notation, in this case we write $D_2 = h(D_1)$. With this notion at hand, given a channel-bounded DDS $\mathcal{M}$ with DRTS $\Upsilon_{\mathcal{M}} = \langle \mathcal{N}, \Delta, \mathcal{S}, \mathcal{T}, \Sigma, \sigma_0, ndb, nch, \Rightarrow \rangle$, we say that $\Upsilon_{\mathcal{M}}$ is *uniform* if for every $\sigma, \sigma_{next}, \sigma' \in \Sigma$ and every pair $D = ((S_n)_{n \in V}, ((C_c)_{c \in A})) \in \prod_{n \in V} \mathbb{S} \times \prod_{a \in A} \mathbb{C}$, where C_c is a DB instance of the multiset data-structure over Δ: if

1. $\sigma \Rightarrow \sigma_{next}$;
2. the number of constants mentioned in σ and σ_{next} together is the same as that of constants mentioned in σ' and D;

3. there exists a bijection $\Delta \longrightarrow \Delta$ that fixes $\Delta_{0,\mathcal{M}}$ and maps each node DB and channel DB of σ and σ_{next} into those of σ' and D, thus enforcing an isomorphism over all components of the various states and D;

then $D \in \Sigma$ and $\sigma' \Rightarrow D$. Making use of uniformity, we get:

Theorem 1. *Verification of convergence properties over bounded i*DDS*s is decidable in* PSPACE *in the network size.*

Proof (sketch). First of all, it can be easily proven that DDSs are uniform. Uniformity comes from the fact that: *(i)* D2C is based on Datalog, which, as virtually all DB query languages, enjoys genericity; *(ii)* external inputs are provided "uniformly", i.e., whenever an input DB is delivered to a node, there is an alternative execution in which the node receives an isomorphic variant of the same input. For a uniform dynamic system, with a pre-defined bound on the size of its DBs, thus including the RTS version of the DRTSs of bounded-DDS, verification of *FO-CTL* properties, including translations of convergence, is decidable with a PSPACE (tight) bound in data complexity [11], which in our case comprises also the size of the network. The intuition behind the decidability proof in [11], is that given an RTS $\Upsilon^{\mathrm{i}}_{\mathcal{M}}$ and a *FO-CTL* property Φ, a finite domain $\Delta_f \subset \Delta$ (with $\Delta_{0,\mathcal{M}} \subseteq \Delta_f$) can be found such that $\Upsilon^{\mathrm{i}}_{\mathcal{M}} \models \Phi$ iff $\Theta^{\mathrm{i}}_{\mathcal{M}} \models \Phi$, where: *(i)* $\Upsilon^{\mathrm{i}}_{\mathcal{M}}$ is the RTS version of the DRTS of $\mathcal{M}$; *(ii)* $\Theta^{\mathrm{i}}_{\mathcal{M}}$ is the RTS obtained by applying the same construction for $\Upsilon^{\mathrm{i}}_{\mathcal{M}}$ but using the finite domain Δ_f instead of Δ. Note that $\Theta^{\mathrm{i}}_{\mathcal{M}}$ is finite-state, hence the standard on-the-fly model checking algorithm for CTL can be applied to check whether Φ holds.

We next show that bounding the channel multisets is necessary towards decidability of verification over iDDSs.

Theorem 2. *Verification of convergence over input- and state-bounded i*DDS*s is undecidable, even when the* DDS *employs: (i) a single-node network; (ii) a single unary,* 1*-bounded input relation; (iii)* 0*-ary state relations; (iv) two* 2*-ary transport relations.*

Proof (sketch). The proof is via a reduction from the undecidable halting problem of deterministic 2-counter machines [23] to convergence of iDDSs as in the theorem statement. The 2-counter machine $\mathfrak{M}$ states are encoded in the DDS state by means of 0-ary state predicates (state-flags). The counters are encoded in the node self-loop channel as the lengths of two cyclic graphs whose edges are specified by the transport relations `counter1/2` and `counter2/2`, respectively. The initial DDS state DB contains the state-flag encoding the initial state of $\mathfrak{M}$.

At the first computation step the node, say named `me`, sends to himself two messages: `counter1(me, me)` and `counter2(me, me)`. Then, the program triggers the increment and conditional decrement instructions according to the current DDS state-flag.

To increment a counter, say counter 1, the node extracts a single constant from the current input DB, checks whether it is fresh with respect to the counter, and, in that case, puts the fresh constant in the appropriate cyclic graph. To

perform the freshness test, the node expects to receive (and then sends it back on the channel), fact by fact, the `counter1` cyclic-graph in the right order, i.e. from `counter1(me, _)` to `counter1(_, me)`. In case the input constant was not available in the input DB, i.e., the DB was empty, or it already appeared in the incoming message, i.e., it was not fresh, or the incoming message is a `counter2` or an unexpected `counter1` message, the node enters in an error state.

To perform a conditional decrement on a counter, say again counter 1, the node expects to receive a `counter1` message. If it is a `counter1(me, me)` message, then the node sends back the message and detects that the counter is zero. If the message is a different `counter1` message, the node stores it in the state, expects to receive the next `counter1` edge and then sends back only one edge where the middle constant has been dropped. Again, in case unexpected messages are delivered, the node enters in an error state.

After performing each increment or decrement instruction, the node transitions to the next state. In case the error state is reached, the node starts sending at each step a `foo/0` message, whose reception triggers a random state transition (excluding the final state of $\mathfrak{M}$ and the DDS current state-flag). Thus, $\mathfrak{M}$ terminates if and only if the DDS sometimes (always) converges with total (partial) correctness.

Considering Theorem 2, in the following we assume that DDSs are channel-bounded. Notice that channel-boundedness implies transport-boundedness, since at each computation step the whole transport DB is always sent in the reliable channel. This means that it makes no sense to study the verification of DDSs that are channel-bounded but not transport-bounded. Thus, we consider now what happens when the DDS is input- and transport-bounded, but have unconstrained state.

Theorem 3. *Verification of convergence over input- and channel-bounded i*DDS*s is undecidable, even when i*DDS *employs: (i) a single computation node; (ii) a single unary, 1-bounded input relation; (iii) Two 1-ary state relations. (iv) a single 0-ary transport relation;*

Proof (sketch). The proof is similar to that of theorem 2, but the counters of $\mathfrak{M}$ are now encoded in the state of $\mathcal{M}$ by means of two state predicates `counter1/1` and `counter2/1`. The node continually sends to himself a `wakeup/0` message, unless it reaches the final state of $\mathfrak{M}$. Thus, the channel gets empty, causing the DDS to trivially converge, if and only if the DDS reaches the final state of $\mathfrak{M}$.

In this case, freshness and zero checks can be done in one step by means of a couple of D2C rules that inspect the state DB. In case the freshness check fails or the input constant is not available, the node enters in a state error as above, triggering a run that never converges. Hence, $\mathfrak{M}$ terminates if and only if the DDS sometimes (always) converges with total (partial) correctness.

We investigate now what happens when the state DBs and channels are bounded, but the input is not. This is a subtle case: unboundedly many input data can be delivered to a node, but not inserted into its state/transport. In fact:

Theorem 4. *Verification of convergence properties over state- and channel-bounded i*DDS*s is in* PSPACE *in the network size.*

Proof. Let $\mathcal{M}$ be an iDDSs, with DRTS $\langle \mathcal{N}, \Delta, \mathcal{S}, \mathcal{T}, \Sigma, \sigma_0, ndb, nch, \Rightarrow \rangle$, whose state and transport are, together, bounded by b, and let φ be a convergence property. While Υ could be infinite, we can specify and build a finite DRTS Υ' equivalent to Υ under φ. Thus, to check whether $\Upsilon \models \varphi$ amounts to check whether $\Upsilon' \models \varphi$.

Let $vars(\varphi)$ and Con be the set of variables and constants, respectively, occurring in φ. Fix a set $C \subset \Delta$ of $2b + |vars(\varphi)| + |Con|$ constants, disjoint from those in $\Delta_{0\mathcal{M}}$, i.e., the initial state active domain, and call $\Delta' = \Delta_{0,\mathcal{M}} \cup C$. Then, Υ' is the DRTS $\langle \mathcal{N}, \Delta, \mathcal{S}, \mathcal{T}, \Sigma', \sigma_0, ndb', nch', \Rightarrow' \rangle$ such that: *(i)* Σ' is the set of all the states $\sigma \in \Sigma$ such that the set of constants mentioned in σ are at most b and all contained in Δ'; *(ii)* ndb', nch', and $\Rightarrow'$ are the restrictions to Σ' of ndb, nch, and $\Rightarrow$ respectively. Since both the schemas $\mathcal{S}$ and $\mathcal{T}$, and the domain Δ' are finite, also Σ' is finite, so that Υ' is a finite DRTS. Moreover, since $|\Delta'| \geq 2b + |Con| + |vars(\varphi)|$ and Υ is uniform (see proof 1), it is possible to adapt Th. 3.18 in [11] to obtain that the RTS versions of Υ and Υ' are equivalent under φ. In fact, recall that those RTS versions exist because, since the channels are bounded, the instances of the channel multisets can be encoded in a DB. We now show a procedure to effectively build Υ' from $\mathcal{M}$: we will compute a number of grounded versions of the program of $\mathcal{M}$, which can be recursively applied starting from σ_0 up to a fix-point to build Υ'.

Given a rule in a D2C program, we call *input-*, *state-*, and *transport-variables* those variables occurring only in input-, state-, and transport-predicates respectively. While the semantics of D2C requires a preliminary full grounding of the rules under the state and the input DBs active domains, we start by grounding only the state- and transport-variables with constants in the (finite) full domain Δ' of Υ', resulting in the program P'. Its variables occur only in input-predicates, hence the heads are fully grounded in Δ'. Nevertheless, the resulting program is suitable to compute all the transitions in Υ' since, by construction, each state in Υ' has a DB over Δ'.

While we should ground also the input-variables with constants in the unconstrained input domains, we can avoid it by noting that they act like variables in a Boolean query over the input, independent of the grounded part of the program. Specifically, for each semi-grounded rule ρ in P' with at least one variable, consider the existential closure q_ρ of the conjunction of all input literals in ρ. These q_i form a finite family of existential sentences. The effect of an input DB I is just to discard those rules ρ such that $I \not\models q_\rho$. To capture the effects of all possible input DBs at once, we initialize a table with one column for each q_ρ and populate it with all the distinct rows r_j containing a possible sequence of the symbols $\top$ and $\bot$. For each row r_j in the table, consider the conjunction of: *(i)* all q_ρ such that the corresponding cell contains $\top$, and *(ii)* of all $\neg q_\rho$ such that the corresponding cell contains $\bot$. Written in negative normal form, this is a conjunction of existential and universal sentences, which can be transformed into a prenex formula Ξ_j of the form $\exists^{\leq n} \forall^{\leq n}$, where n is the number

of all variables in P' times the number of queries q_ρ. This is a fragment of the Bernays-Schönfinkel class enjoying a finite satisfiablity problem in NEXPTIME in the length of the sentence, but in $O(1)$ in the network size. If the sentence Ξ_j is not satisfiable in the finite, then there is no input DB enabling the effects described by r_j, thus it has to be deleted from the table. Otherwise, it has to be retained. After this pruning, for each row r_j consider the fully-grounded subprogram P'_j of P' containing only those rules ρ, pruned of the input predicates, such that the corresponding cell in r_j contains $\top$. If the body results empty, fill it with true.

Given a state σ' in Υ', the finite family of programs P'_j, with no input predicates, capture the effects of all the infinitely many input DBs. Thus, to compute a successor of a given configuration in Σ', it is sufficient to compute a stable model of a program P'_j over the configuration DB, which can be done in PTIME in data complexity. Thus, by applying on-the-fly verification techniques for finite RTSs against FO-CTL, which can express convergence translated over those RTSs, we can check Υ' in NPSPACE in data complexity, which amounts to PSPACE.

4 Conclusions

In the wide spectrum of declarative distributed computing, we have formalized and studied verification of convergence in the important case of reliable communication with unordered asynchronous communication and interactive input policy. While the problem is undecidable in general, decidability can be regained by imposing boundedness conditions on the channels and state DBs.

We foresee two main lines of future research. First, we plan to build on our foundational results to implement verification techniques for DDS cases with decidable verification . To this aim, we will rely on existing ASP techniques for D2C, and in particular on the implementation in [18], which can simulate runs of DDSs according to our formalization. Remarkably, those cases are all channel-bounded, and thus pose no problem for the implementation of the multiset channels. However, great care should be put in handling the message passing mechanism, since the reception of a random incoming message requires to analyze many different branches, resulting in a bottleneck. Moreover, the technique in the proof for theorem 4 could be exploited to abstract away the interaction policy, thus avoiding the necessity of providing random input DBs at each step. Second, we want to study how our verification results carry over the setting in which the network topology can change, both with respect to node connection, and for what concerns the creation and deactivation of computation nodes. In this light, it is worth noting that, following the approach in [24], the results here presented can be seamlessly generalized to the case where node connections are arbitrarily changed over time, and nodes can be created and deactived, provided that their overall number does not exceed a pre-defined bound. On the other hand, the case where unboundedly many computation nodes can be created is left for interesting future work. In fact, we intend to leverage parameterized verification [1,14] to study data-centric distributed systems whose topology can change over

time in an unbounded, but controlled way, i.e., respecting certain patterns (see, for example, [13], where constraints about topology structures are added to the specification).

We are also interested in exploring the complexity of weaker properties besides convergence, like safety and liveness.

References

1. Abdulla, P.A., Delzanno, G., Rezine, A.: Approximated parameterized verification of infinite-state processes with global conditions. Formal Methods in System Design **34**(2), 126–156 (2009)
2. Abiteboul, S., Abrams, Z., Haar, S., Milo, T.: Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In: Proc. of the 24th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS). pp. 358–367. ACM Press (2005)
3. Abiteboul, S., Bienvenu, M., Galland, A., Antoine, É.: A rule-based language for web data management. In: Proc. of the 30th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS). pp. 293–304. ACM Press (2011)
4. Alvaro, P., Ameloot, T.J., Hellerstein, J.M., Marczak, W., Van den Bussche, J.: A declarative semantics for Dedalus. Tech. Rep. UCB/EECS-2011-120, EECS Department, University of California, Berkeley (Nov 2011), http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-120.html
5. Ameloot, T.J.: Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. SIGMOD Record **43**(2), 5–16 (2014)
6. Ameloot, T.J., Geck, G., Ketsman, B., Neven, F., Schwentick, T.: Parallel-correctness and transferability for conjunctive queries. In: Proc. of the 34th ACM SIGACT SIGMOD SIGAI Symp. on Principles of Database Systems (PODS). pp. 47–58 (2015)
7. Ameloot, T.J., Ketsman, B., Neven, F., Zinn, D.: Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the CALM-conjecture. In: Proc. of the 33rd ACM SIGACT SIGMOD SIGAI Symp. on Principles of Database Systems (PODS). pp. 64–75 (2014)
8. Ameloot, T.J., Neven, F., Van den Bussche, J.: Relational transducers for declarative networking. J. of the ACM **60**(2), 15:1–15:38 (2013)
9. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Proc. of the 32nd ACM SIGACT SIGMOD SIGAI Symp. on Principles of Database Systems (PODS). pp. 163–174 (2013)
10. Bagheri Hariri, B., Calvanese, D., Deutsch, A., Montali, M.: State-boundedness in data-aware dynamic systems. In: Proc. of the 14th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR). AAAI Press (2014)
11. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of agent-based artifact systems. J. of Artificial Intelligence Research **51**, 333–376 (2014). https://doi.org/10.1613/jair.4424
12. Chen, C., Jia, L., Xu, H., Luo, C., Zhou, W., Loo, B.T.: A program logic for verifying secure routing protocols. In: 34th IFIP Int. Conf. on Formal Techniques for Distributed Objects, Components and Systems (FORTE 2014). Lecture Notes in Computer Science, vol. 8461, pp. 117–132. Springer (2014)

13. Chen, C., Loh, L.K., Jia, L., Zhou, W., Loo, B.T.: Automated verification of safety properties of declarative networking programs. In: Proc. of the 17th Int. Symposium on Principles and Practice of Declarative Programming (PPDP). pp. 79–90 (2015)
14. Delzanno, G., Sangnier, A., Traverso, R.: Parameterized verification of broadcast networks of register automata. In: Proc. of the 7th Int. Workshop on Reachability Problems (RP). Lecture Notes in Computer Science, vol. 8169, pp. 109–121. Springer (2013)
15. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of the 5th Int. Conf. on Logic Programming (ICLP). pp. 1070–1080. The MIT Press (1988)
16. Giannotti, F., Pedreschi, D.: Datalog with non-deterministic choice computes NDB-PTIME. J. of Logic Programming **35**(1), 79–101 (1998)
17. Hellerstein, J.M.: The declarative imperative: Experiences and conjectures in distributed logic. SIGMOD Record **39**(1), 5–19 (2010)
18. Lobo, J., Wood, D., Verma, D., Calo, S.: Distributed state machines: A declarative framework for the management of distributed systems. In: Proc. of the 8th Int. Conf. on Network and Service Management (CNSM). pp. 224–228 (2012)
19. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Communications of the ACM **52**(11), 87–95 (2009)
20. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I.: Implementing declarative overlays. Operating Systems Review **39**(5), 75–90 (2005)
21. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
22. Ma, J., Le, F., Wood, D., Russo, A., Lobo, J.: A declarative approach to distributed computing: Specification, execution and analysis. Theory and Practice of Logic Programming **13**, 815–830 (2013)
23. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall (1967)
24. Montali, M., Calvanese, D., De Giacomo, G.: Verification of data-aware commitment-based multiagent systems. In: Proc. of the 13th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS). pp. 157–164 (2014)
25. Nigam, V., Jia, L., Loo, B.T., Scedrov, A.: Maintaining distributed logic programs incrementally. Computer Languages, Systems & Structures **38**(2), 158–180 (2012)
26. Ren, Y., Zhou, W., Wang, A., Jia, L., Gurney, A.J., Loo, B.T., Rexford, J.: FSR: Formal analysis and implementation toolkit for safe inter-domain routing. Computer Communication Review **41**(4), 440–441 (2011)
27. Saccà, D., Zaniolo, C.: Stable models and non-determinism in logic programs with negation. In: Proc. of the 9th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS). pp. 205–217 (1990)
28. Vardi, M.Y.: Model checking for database theoreticians. In: Proc. of the 10th Int. Conf. on Database Theory (ICDT). Lecture Notes in Computer Science, vol. 3363, pp. 1–16. Springer (2005)
29. Zaychik Moffitt, V., Stoyanovich, J., Abiteboul, S., Miklau, G.: Collaborative access control in WebdamLog. In: Proc. of the ACM SIGMOD Int. Conf. on Management of Data. pp. 197–211. ACM (2015)
30. Zhou, W., Mapara, S., Ren, Y., Li, Y., Haeberlen, A., Ives, Z., Loo, B.T., Sherr, M.: Distributed time-aware provenance. Proc. of the VLDB Endowment **6**(2), 49–60 (2012)