Identifiers in Registers

Describing Network Algorithms with Logic

Benedikt Bollig, Patricia Bouyer, and Fabian Reiter(✉)

LSV, CNRS, ENS Paris-Saclay, Université Paris-Saclay, Cachan, France
{bollig,bouyer}@lsv.fr, fabian.reiter@gmail.com

Abstract. We propose a formal model of distributed computing based on register automata that captures a broad class of synchronous network algorithms. The local memory of each process is represented by a finite-state controller and a fixed number of registers, each of which can store the unique identifier of some process in the network. To underline the naturalness of our model, we show that it has the same expressive power as a certain extension of first-order logic on graphs whose nodes are equipped with a total order. Said extension lets us define new functions on the set of nodes by means of a so-called partial fixpoint operator. In spirit, our result bears close resemblance to a classical theorem of descriptive complexity theory that characterizes the complexity class PSPACE in terms of partial fixpoint logic (a proper superclass of the logic we consider here).

1 Introduction

This paper is part of an ongoing research project aiming to develop a *descriptive complexity* theory for *distributed computing*.

In classical sequential computing, descriptive complexity is a well-established field that connects computational complexity classes to equi-expressive classes of logical formulas. It began in the 1970s, when Fagin showed in [6] that the graph properties decidable by nondeterministic Turing machines in polynomial time are exactly those definable in existential second-order logic. This provided a logical—and thus machine-independent—characterization of the complexity class NP. Subsequently, many other popular classes, such as P, PSPACE, and EXPTIME were characterized in a similar manner (see for instance the textbooks [8,12,15]).

Of particular interest to us is a result due to Abiteboul, Vianu [1], and Vardi [19], which states that on structures equipped with a total order relation, the properties decidable in PSPACE coincide with those definable in *partial fixpoint logic*. The latter is an extension of first-order logic with an operator that allows us to inductively define new relations of arbitrary arity. Basically, this means that new relations can occur as free (second-order) variables in the logical formulas that define them. Those variables are initially interpreted as empty relations and then iteratively updated, using the defining formulas as update

M. Bojańczyk and A. Simpson (Eds.): FOSSACS 2019, LNCS 11425, pp. 115–132, 2019.
https://doi.org/10.1007/978-3-030-17127-8_7

rules. If the sequence of updates converges to a fixpoint, then the ultimate interpretations are the relations reached in the limit. Otherwise, the variables are simply interpreted as empty relations. Hence the term "partial fixpoint".

While well-developed in the classical case, descriptive complexity has so far not received much attention in the setting of distributed network computing. As far as the authors are aware, the first step in this direction was taken by Hella et al. in [10,11], where they showed that basic *modal logic* evaluated on finite graphs has the same expressive power as a particular class of *distributed automata* operating in constant time. Those automata constitute a weak model of distributed computing in arbitrary network topologies, where all nodes synchronously execute the same finite-state machine and communicate with each other by broadcasting messages to their neighbors. Motivated by this result, several variants of distributed automata were investigated by Kuusisto and Reiter in [14,18] and [17] to establish similar connections with standard logics such as the *modal μ-calculus* and *monadic second-order logic*. However, since the models of computation investigated in those works are based on anonymous finite-state machines, they are much too weak to solve many of the problems typically considered in distributed computing, such as leader election or constructing a spanning tree. It would thus be desirable to also characterize stronger models.

A common assumption underlying many distributed algorithms is that each node of the considered network is given a unique identifier. This allows us, for instance, to elect a leader by making the nodes broadcast their identifiers and then choose the one with the smallest identifier as the leader. To formalize such algorithms, we need to go beyond finite-state machines because the number of bits required to encode a unique identifier grows logarithmically with the number of nodes in the network. Recently, in [2,3], Aiswarya, Bollig and Gastin introduced a synchronous model where, in addition to a finite-state controller, nodes also have a fixed number of registers in which they can store the identifiers of other nodes. Access to those registers is rather limited in the sense that their contents can be compared with respect to a total order, but their numeric values are unknown to the nodes. (This restriction corresponds precisely to the notion of *order-invariant* distributed algorithms, which was introduced by Naor and Stockmeyer in [16].) Similarly, register contents can be copied, but no new values can be generated. Since the original motivation for the model was to automatically verify certain distributed algorithms running on ring networks, its formal definition is tailored to that particular setting. However, the underlying principle can be generalized to arbitrary networks of unbounded maximum degree, which was the starting point for the present work.

Contributions. While on an intuitive level, the idea of finite-state machines equipped with additional registers might seem very natural, it does not immediately yield a formal model for distributed algorithms in arbitrary networks. In particular, it is not clear what would be the canonical way for nodes to communicate with a non-constant number of peers, if we require that they all follow the same, finitely representable set of rules.

The model we propose here, dubbed *distributed register automata*, is an attempt at a solution. As in [2,3], nodes proceed in synchronous rounds and have a fixed number of registers, which they can compare and update without having access to numeric values. The new key ingredient that allows us to formalize communication between nodes of unbounded degree is a local computing device we call *transition maker*. This is a special kind of register machine that the nodes can use to scan the states and register values of their entire neighborhood in a sequential manner. In every round, each node runs the transition maker to update its own local configuration (i.e., its state and register valuation) based on a snapshot of the local configurations of its neighbors in the previous round. A way of interpreting this is that the nodes communicate by broadcasting their local configurations as messages to their neighbors. Although the resulting model of computation is by no means universal, it allows formalizing algorithms for a wide range of problems, such as constructing a spanning tree (see Example 5) or testing whether a graph is Hamiltonian (see Example 6).

Nevertheless, our model is somewhat arbitrary, since it could be just one particular choice among many other similar definitions capturing different classes of distributed algorithms. What justifies our choice? This is where descriptive complexity comes into play. By identifying a logical formalism that has the same expressive power as distributed register automata, we provide substantial evidence for the naturalness of that model. Our formalism, referred to as *functional fixpoint logic*, is a fragment of the above-mentioned partial fixpoint logic. Like the latter, it also extends first-order logic with a partial fixpoint operator, but a weaker one that can only define unary functions instead of arbitrary relations. We show that on totally ordered graphs, this logic allows one to express precisely the properties that can be decided by distributed register automata. The connection is strongly reminiscent of Abiteboul, Vianu and Vardi's characterization of PSPACE, and thus contributes to the broader objective of extending classical descriptive complexity to the setting of distributed computing. Moreover, given that logical formulas are often more compact and easier to understand than abstract machines (compare Examples 6 and 8), logic could also become a useful tool in the formal specification of distributed algorithms.

The remainder of this paper is structured around our main result:

Theorem 1. *When restricted to finite graphs whose nodes are equipped with a total order, distributed register automata are effectively equivalent to functional fixpoint logic.*

After giving some preliminary definitions in Sect. 2, we formally introduce distributed register automata in Sect. 3 and functional fixpoint logic in Sect. 4. We then sketch the proof of Theorem 1 in Sect. 5, and conclude in Sect. 6.

2 Preliminaries

We denote the empty set by $\emptyset$, the set of nonnegative integers by $\mathbb{N} = \{0, 1, 2, \dots\}$, and the set of integers by $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$. The cardinality of any set S is written as $|S|$ and the power set as 2^S.

In analogy to the commonly used notation for real intervals, we define the notation $[m:n] \coloneqq \{i \in \mathbb{Z} \mid m \leq i \leq n\}$ for any $m, n \in \mathbb{Z}$ such that $m \leq n$. To indicate that an endpoint is excluded, we replace the corresponding square bracket with a parenthesis, e.g., $(m:n] \coloneqq [m:n] \setminus \{m\}$. Furthermore, if we omit the first endpoint, it defaults to 0. This gives us shorthand notations such as $[n] \coloneqq [0:n]$ and $[n) \coloneqq [0:n) = [0:n-1]$.

All graphs we consider are finite, simple, undirected, and connected. For notational convenience, we identify their nodes with nonnegative integers, which also serve as unique identifiers. That is, when we talk about the *identifier* of a node, we mean its numerical representation. A *graph* is formally represented as a pair $G = (V, E)$, where the set V of *nodes* is equal to $[n)$, for some integer $n \geq 2$, and the set E consists of undirected *edges* of the form $e = \{u, v\} \subseteq V$ such that $u \neq v$. Additionally, E must satisfy that every pair of nodes is connected by a sequence of edges. The restriction to graphs of size at least two is for technical reasons; it ensures that we can always encode Boolean values as nodes.

We refer the reader to [5] for standard graph theoretic terms such as *neighbor*, *degree*, *maximum degree*, *distance*, and *spanning tree*.

Graphs are used to model computer networks, where nodes correspond to processes and edges to communication links. To represent the current configuration of a system as a graph, we equip each node with some additional information: the current state of the corresponding process, taken from a nonempty finite set Q, and some pointers to other processes, modeled by a finite set R of registers.

We call $\Sigma = (Q, R)$ a *signature* and define a *Σ-configuration* as a tuple $C = (G, \mathfrak{q}, \mathfrak{r})$, where $G = (V, E)$ is a graph, called the *underlying* graph of C, $\mathfrak{q}\colon V \to Q$ is a *state function* that assigns to each node a state $q \in Q$, and $\mathfrak{r}\colon V \to V^R$ is a *register valuation function* that associates with each node a *register valuation* $\rho \in V^R$. The set of all Σ-configurations is denoted by $\mathbb{C}(\Sigma)$. Figure 1 on page 6 illustrates part of a $(\{q_1, q_2, q_3\}, \{r_1, r_2, r_3\})$-configuration.

If $R = \emptyset$, then we are actually dealing with a tuple $(G, \mathfrak{q})$, which we call a *Q-labeled graph*. Accordingly, the elements of Q may also be called *labels*. A set P of labeled graphs will be referred to as a *graph property*. Moreover, if the labels are irrelevant, we set Q equal to the singleton $\mathbb{1} \coloneqq \{\varepsilon\}$, where ε is our dummy label. In this case, we identify $(G, \mathfrak{q})$ with G and call it an *unlabeled* graph.

3 Distributed Register Automata

Many distributed algorithms can be seen as *transducers*. A leader-election algorithm, for instance, takes as input a network and outputs the same network, but with every process storing the identifier of the unique leader in some dedicated register r. Thus, the algorithm transforms a $(\mathbb{1}, \emptyset)$-configuration into a $(\mathbb{1}, \{r\})$-configuration. We say that it defines a $(\mathbb{1}, \emptyset)$-$(\mathbb{1}, \{r\})$-transduction. By the same token, if we consider distributed algorithms that *decide* graph properties (e.g., whether a graph is Hamiltonian), then we are dealing with a $(I, \emptyset)$-$(\{\text{YES}, \text{NO}\}, \emptyset)$-transduction, where I is some set of labels. The idea is that a graph will be accepted if and only if every process eventually outputs YES.

Let us now formalize the notion of transduction. For any two signatures $\Sigma^{in} = (I, R^{in})$ and $\Sigma^{out} = (O, R^{out})$, a *$\Sigma^{in}$-$\Sigma^{out}$-transduction* is a *partial* mapping $T\colon \mathbb{C}(\Sigma^{in}) \to \mathbb{C}(\Sigma^{out})$ such that, if defined, $T(G, \mathfrak{q}, \mathfrak{r}) = (G, \mathfrak{q}', \mathfrak{r}')$ for some $\mathfrak{q}'$ and $\mathfrak{r}'$. That is, a transduction does not modify the underlying graph but only the states and register valuations. We denote the set of all Σ^{in}-Σ^{out}-transductions by $\mathbb{T}(\Sigma^{in}, \Sigma^{out})$ and refer to Σ^{in} and Σ^{out} as the *input* and *output signatures* of T. By extension, I and O are called the sets of *input* and *output labels*, and R^{in} and R^{out} the sets of *input* and *output registers*. Similarly, any Σ^{in}-configuration C can be referred to as an *input configuration* of T and $T(C)$ as an *output configuration*.

Next, we introduce our formal model of distributed algorithms.

Definition 2 (Distributed register automaton). *Let $\Sigma^{in} = (I, R^{in})$ and $\Sigma^{out} = (O, R^{out})$ be two signatures. A* distributed register automaton *(or simply* automaton*) with input signature Σ^{in} and output signature Σ^{out} is a tuple $A = (Q, R, \iota, \Delta, H, o)$ consisting of a nonempty finite set Q of* states*, a finite set R of* registers *that includes both R^{in} and R^{out}, an* input function *$\iota\colon I \to Q$, a transition maker Δ whose specification will be given in Definition 3 below, a set $H \subseteq Q$ of* halting states*, and an* output function *$o\colon H \to O$. The registers in $R \setminus (R^{in} \cup R^{out})$ are called* auxiliary registers*.*

Automaton A computes a transduction $T_A \in \mathbb{T}(\Sigma^{in}, \Sigma^{out})$. To do so, it runs in a sequence of synchronous rounds on the input configuration's underlying graph $G = (V, E)$. After each round, the automaton's global configuration is a (Q, R)-configuration $C = (G, \mathfrak{q}, \mathfrak{r})$, i.e., the underlying graph is always G. As mentioned before, for a node $v \in V$, we interpret $\mathfrak{q}(v) \in Q$ as the current state of v and $\mathfrak{r}(v) \in V^R$ as the current register valuation of v. Abusing notation, we let $C(v) := (\mathfrak{q}(v), \mathfrak{r}(v))$ and say that $C(v)$ is the *local configuration* of v. In Fig. 1, the local configuration node 17 is $(q_1, \{r_1, r_2, r_3 \mapsto 17,\ 34,\ 98\})$.

For a given input configuration $C = (G, \mathfrak{q}, \mathfrak{r}) \in \mathbb{C}(\Sigma^{in})$, the automaton's *initial configuration* is $C' = (G, \iota \circ \mathfrak{q}, \mathfrak{r}')$, where for all $v \in V$, we have $\mathfrak{r}'(v)(r) = \mathfrak{r}(v)(r)$ if $r \in R^{in}$, and $\mathfrak{r}'(v)(r) = v$ if $r \in R \setminus R^{in}$. This means that every node v is initialized to state $\iota(\mathfrak{q}(v))$, and v's initial register valuation $\mathfrak{r}'(v)$ assigns v's own identifier (provided by G) to all non-input registers while keeping the given values assigned by $\mathfrak{r}(v)$ to the input registers.

Each subsequent configuration is obtained by running the transition maker Δ synchronously on all nodes. As we will see, Δ computes a function

$$[\![\Delta]\!]\colon (Q \times V^R)^+ \to Q \times V^R$$

that maps from nonempty sequences of local configurations to local configurations. This allows the automaton A to transition from a given configuration C to the next configuration C' as follows. For every node $u \in V$ of degree d, we consider the list $v_1, \dots v_d$ of u's neighbors sorted in ascending (identifier) order, i.e., $v_i < v_{i+1}$ for $i \in [1 : d)$. (See Fig. 1 for an example, where u corresponds to node 17.) If u is already in a halting state, i.e., if $C(u) = (q, \rho) \in H \times V^R$,

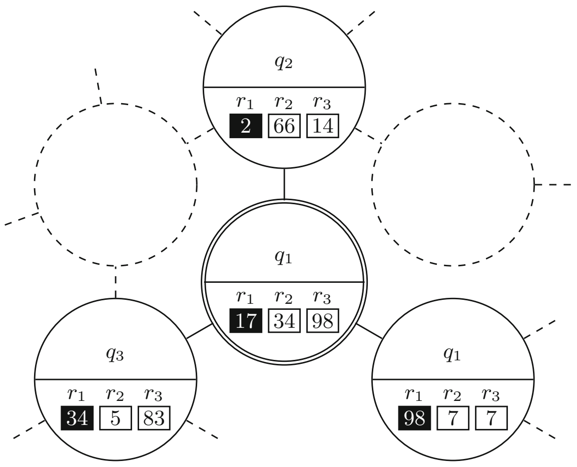

Fig. 1. Part of a configuration, as seen by a single node. Assuming the identifiers of the nodes are the values represented in black boxes (i.e., those stored in register r_1), the automaton at node 17 will update its own local configuration (q_1, $\{r_1, r_2, r_3 \mapsto$ 17, 34, 98$\}$) by running the transition maker on the sequence consisting of the local configurations of nodes 17, 2, 34, and 98 (in that exact order).

then its local configuration does not change anymore, i.e., $C'(u) = C(u)$. Otherwise, we define $C'(u) = [\![\Delta]\!]\big(C(u), C(v_1), \ldots, C(v_d)\big)$, which we may write more suggestively as

$$[\![\Delta]\!] \colon C(u) \xmapsto{C(v_1),\ldots,C(v_d)} C'(u).$$

Intuitively, node u updates its own local configuration by using Δ to scan a snapshot of its neighbors' local configurations. As the system is synchronous, this update procedure is performed simultaneously by all nodes.

A configuration $C = (G, \mathfrak{q}, \mathfrak{r})$ is called a *halting configuration* if all nodes are in a halting state, i.e., if $\mathfrak{q}(v) \in H$ for all $v \in V$. We say that A *halts* if it reaches a halting configuration.

The output configuration produced by a halting configuration $C = (G, \mathfrak{q}, \mathfrak{r})$ is the Σ^{out}-configuration $C' = (G, o \circ \mathfrak{q}, \mathfrak{r}')$, where for all $v \in V$ and $r \in R^{out}$, we have $\mathfrak{r}'(v)(r) = \mathfrak{r}(v)(r)$. In other words, each node v outputs the state $o(\mathfrak{q}(v))$ and keeps in its output registers the values assigned by $\mathfrak{r}(v)$.

It is now obvious that A defines a transduction $T_A \colon \mathbb{C}(\Sigma^{in}) \to \mathbb{C}(\Sigma^{out})$. If A receives the input configuration $C \in \mathbb{C}(\Sigma^{in})$ and eventually halts and produces the output configuration $C' \in \mathbb{C}(\Sigma^{out})$, then $T_A(C) = C'$. Otherwise (if A does not halt), $T_A(C)$ is undefined.

Deciding graph properties. Our primary objective is to use distributed register automata as decision procedures for graph properties. Therefore, we will focus on automata A that halt in a finite number of rounds on *every* input configuration, and we often restrict to input signatures of the form $(I, \emptyset)$ and the output signature $(\{\text{YES}, \text{NO}\}, \emptyset)$. For example, for $I = \{a, b\}$, we may be interested in the set of I-labeled graphs that have exactly one a-labeled node v (the "leader"). We stipulate that A *accepts* an input configuration C with underlying graph $G = (V, E)$ if $T_A(C) = (G, \mathfrak{q}, \mathfrak{r})$ such that $\mathfrak{q}(v) = \text{YES}$ for *all* $v \in V$. Conversely, A *rejects* C if $T_A(C) = (G, \mathfrak{q}, \mathfrak{r})$ such that $\mathfrak{q}(v) = \text{NO}$ for *some* $v \in V$. This corresponds to the usual definition chosen in the emerging field of *distributed decision* [7]. Accordingly, a graph property P is *decided* by A if the automaton accepts all input configurations that satisfy P and rejects all the others.

It remains to explain how the transition maker Δ works internally.

Definition 3 (Transition maker). *Suppose that $A = (Q, R, \iota, \Delta, H, o)$ is a distributed register automaton. Then its* transition maker $\Delta = (\tilde{Q}, \tilde{R}, \tilde{\iota}, \tilde{\delta}, \tilde{o})$ *consists of a nonempty finite set $\tilde{Q}$ of* inner states*, a finite set $\tilde{R}$ of* inner registers *that is disjoint from R, an* inner initial state $\tilde{\iota} \in \tilde{Q}$*, an* inner transition function $\tilde{\delta}\colon \tilde{Q} \times Q \times 2^{(\tilde{R} \cup R)^2} \to \tilde{Q} \times (\tilde{R} \cup R)^{\tilde{R}}$*, and an* inner output function $\tilde{o}\colon \tilde{Q} \to Q \times \tilde{R}^R$*.*

Basically, a transition maker $\Delta = (\tilde{Q}, \tilde{R}, \tilde{\iota}, \tilde{\delta}, \tilde{o})$ is a sequential register automaton (in the spirit of [13]) that reads a nonempty sequence $(q_0, \rho_0), \ldots, (q_d, \rho_d) \in (Q \times V^R)^+$ of local configurations of A in order to produce a new local configuration (q', ρ'). While reading this sequence, it traverses itself a sequence $(\tilde{q}_0, \tilde{\rho}_0), \ldots, (\tilde{q}_{d+1}, \tilde{\rho}_{d+1})$ of *inner configurations*, which each consist of an inner state $\tilde{q}_i \in \tilde{Q}$ and an *inner register valuation* $\tilde{\rho}_i \in (V \cup \{\bot\})^{\tilde{R}}$, where the symbol $\bot$ represents an undefined value. For the initial inner configuration, we set $\tilde{q}_0 = \tilde{\iota}$ and $\tilde{\rho}_0(\tilde{r}) = \bot$ for all $\tilde{r} \in \tilde{R}$. Now for $i \in [d]$, when Δ is in the inner configuration $(\tilde{q}_i, \tilde{\rho}_i)$ and reads the local configuration (q_i, ρ_i), it can compare all values assigned to the inner registers and registers by $\tilde{\rho}_i$ and ρ_i (with respect to the order relation on V). In other words, it has access to the binary relation $\prec_i \subseteq (\tilde{R} \cup R)^2$ such that for $\tilde{r}, \tilde{s} \in \tilde{R}$ and $r, s \in R$, we have $\tilde{r} \prec_i r$ if and only if $\tilde{\rho}_i(\tilde{r}) < \rho_i(r)$, and analogously for $r \prec_i \tilde{r}$, $\tilde{r} \prec_i \tilde{s}$, and $r \prec_i s$. In particular, if $\tilde{\rho}_i(\tilde{r}) = \bot$, then $\tilde{r}$ is incomparable with respect to $\prec_i$. Equipped with this relation, Δ transitions to $(\tilde{q}_{i+1}, \tilde{\rho}_{i+1})$ by evaluating $\tilde{\delta}(\tilde{q}_i, q_i, \prec_i) = (\tilde{q}_{i+1}, \tilde{\alpha})$ and computing $\tilde{\rho}_{i+1}$ such that $\tilde{\rho}_{i+1}(\tilde{r}) = \tilde{\rho}_i(\tilde{s})$ if $\tilde{\alpha}(\tilde{r}) = \tilde{s}$, and $\tilde{\rho}_{i+1}(\tilde{r}) = \rho_i(s)$ if $\tilde{\alpha}(\tilde{r}) = s$, where $\tilde{r}, \tilde{s} \in \tilde{R}$ and $s \in R$. Finally, after having read the entire input sequence and reached the inner configuration $(\tilde{q}_{d+1}, \tilde{\rho}_{d+1})$, the transition maker outputs the local configuration (q', ρ') such that $\tilde{o}(\tilde{q}_{d+1}) = (q', \tilde{\beta})$ and $\tilde{\beta}(r) = \tilde{r}$ implies $\rho'(r) = \tilde{\rho}_{d+1}(\tilde{r})$. Here we assume without loss of generality that Δ guarantees that $\rho'(r) \neq \bot$ for all $r \in R$.

Remark 4. Recall that $V = [n)$ for any graph $G = (V, E)$ with n nodes. However, as registers cannot be compared with constants, this actually represents an arbitrary assignment of unique, totally ordered identifiers. To determine the smallest identifier (i.e., 0), the nodes can run an algorithm such as the following.

Example 5 (Spanning tree). We present a simple automaton $A = (Q, R, \iota, \Delta, H, o)$ with input signature $\Sigma^{in} = (\mathbb{1}, \emptyset)$ and output signature $\Sigma^{out} = (\mathbb{1}, \{\mathit{parent}, \mathit{root}\})$ that computes a (breadth-first) spanning tree of its input

Algorithm 1. Transition maker of the automaton from Example 5

```
if $\exists$ neighbor $nb$ $(nb.root < my.root)$:                                     } Rule 1
    $my.state \leftarrow 1$;   $my.parent \leftarrow nb.self$;   $my.root \leftarrow nb.root$

else if $my.state = 1$                                                                  } Rule 2
        $\wedge\ \forall$ neighbor $nb \begin{bmatrix} nb.root = my.root\ \wedge \\ (nb.parent \neq my.self\ \vee\ nb.state = 2) \end{bmatrix}$:
    $my.state \leftarrow 2$

else if $(my.state = 2 \wedge my.root = my.self) \vee (my.parent.state = 3)$:          } Rule 3
    $my.state \leftarrow 3$

else do nothing
```

graph $G = (V, E)$, rooted at the node with the smallest identifier. More precisely, in the computed output configuration $C = (G, \mathfrak{q}, \mathfrak{r})$, every node will store the identifier of its tree parent in register *parent* and the identifier of the root (i.e., the smallest identifier) in register *root*. Thus, as a side effect, A also solves the leader election problem by electing the root as the leader.

The automaton operates in three phases, which are represented by the set of states $Q = \{1, 2, 3\}$. A node terminates as soon as it reaches the third phase, i.e., we set $H = \{3\}$. Accordingly, the (trivial) input and output functions are $\iota\colon \varepsilon \mapsto 1$ and $o\colon 3 \mapsto \varepsilon$. In addition to the output registers, each node has an auxiliary register *self* that will always store its own identifier. Thus, we choose $R = \{\mathit{self}, \mathit{parent}, \mathit{root}\}$. For the sake of simplicity, we describe the transition maker Δ in Algorithm 1 using pseudocode rules. However, it should be clear that these rules could be relatively easily implemented according to Definition 3.

All nodes start in state 1, which represents the tree-construction phase. By Rule 1, whenever an active node (i.e., a node in state 1 or 2) sees a neighbor whose *root* register contains a smaller identifier than the node's own *root* register, it updates its *parent* and *root* registers accordingly and switches to state 1. To resolve the nondeterminism in Rule 1, we stipulate that *nb* is chosen to be the neighbor with the smallest identifier among those whose *root* register contains the smallest value seen so far.

As can be easily shown by induction on the number of communication rounds, the nodes have to apply Rule 1 no more than diameter(G) times in order for the pointers in register *parent* to represent a valid spanning tree (where the root points to itself). However, since the nodes do not know when diameter(G) rounds have elapsed, they must also check that the current configuration does indeed represent a single tree, as opposed to a forest. They do so by propagating a signal, in form of state 2, from the leaves up to the root.

By Rule 2, if an active node whose neighbors all agree on the same root realizes that it is a leaf or that all of its children are in state 2, then it switches to state 2 itself. Assuming the *parent* pointers in the current configuration already represent a single tree, Rule 2 ensures that the root will eventually be notified of this fact (when all of its children are in state 2). Otherwise, the *parent* pointers represent a forest, and every tree contains at least one node that has a neighbor outside of the tree (as we assume the underlying graph is connected).

Depending on the input graph, a node can switch arbitrarily often between states 1 and 2. Once the spanning tree has been constructed and every node is in state 2, the only node that knows this is the root. In order for the algorithm to terminate, Rule 3 then makes the root broadcast an acknowledgment message down the tree, which causes all nodes to switch to the halting state 3. $\square$

Building on the automaton from Example 5, we now give an example of a graph property that can be decided in our model of distributed computing. The following automaton should be compared to the logical formula presented later in Example 8, which is much more compact and much easier to specify.

Example 6 (Hamiltonian cycle). We describe an automaton with input signature $\Sigma^{in} = (\mathbb{1}, \{\mathit{parent}, \mathit{root}\})$ and output signature $\Sigma^{out} = (\{\text{YES}, \text{NO}\}, \emptyset)$ that decides if the underlying graph $G = (V, E)$ of its input configuration $C = (G, \mathfrak{q}, \mathfrak{r})$ is Hamiltonian, i.e., whether G contains a cycle that goes through each node exactly once. The automaton works under the assumption that $\mathfrak{r}$ encodes a valid spanning tree of G in the registers *parent* and *root*, as constructed by the automaton from Example 5. Hence, by combining the two automata, we could easily construct a third one that decides the graph property of Hamiltonicity.

The automaton $A = (Q, R, \iota, \Delta, H, o)$ presented here implements a simple backtracking algorithm that tries to traverse G along a Hamiltonian cycle. Its set of states is $Q = \big(\{\mathit{unvisited}, \mathit{visited}, \mathit{backtrack}\} \times \{\mathit{idle}, \mathit{request}, \mathit{good}, \mathit{bad}\}\big) \cup H$, with the set of halting states $H = \{\text{YES}, \text{NO}\}$. Each non-halting state consists of two components, the first one serving for the backtracking procedure and the second one for communicating in the spanning tree. The input function ι initializes every node to the state $(\mathit{unvisited}, \mathit{idle})$, while the output function simply returns the answers chosen by the nodes, i.e., $o\colon \text{YES} \mapsto \text{YES},\ \text{NO} \mapsto \text{NO}$. In addition to the input registers, each node has a register *self* storing its own identifier and a register *successor* to point to its successor in a (partially constructed) Hamiltonian path. That is, $R = \{\mathit{self}, \mathit{parent}, \mathit{root}, \mathit{successor}\}$. We now describe the algorithm in an informal way. It is, in principle, easy to implement in the transition maker Δ, but a thorough formalization would be rather cumbersome.

In the first round, the root marks itself as *visited* and updates its *successor* register to point towards its smallest neighbor (the one with the smallest identifier). Similarly, in each subsequent round, any *unvisited* node that is pointed to by one of its neighbors marks itself as *visited* and points towards its smallest *unvisited* neighbor. However, if all neighbors are already *visited*, the node instead sends the *backtrack* signal to its predecessor and switches back to *unvisited* (in the following round). Whenever a *visited* node receives the *backtrack* signal from its *successor*, it tries to update its *successor* to the next-smallest *unvisited* neighbor. If no such neighbor exists, it resets its *successor* pointer to itself, propagates the *backtrack* signal to its predecessor, and becomes *unvisited* in the following round.

There is only one exception to the above rules: if a node that is adjacent to the root cannot find any *unvisited* neighbor, it chooses the root as its *successor*.

This way, the constructed path becomes a cycle. In order to check whether that cycle is Hamiltonian, the root now broadcast a *request* down the spanning tree. If the *request* reaches an *unvisited* node, that node replies by sending the message *bad* towards the root. On the other hand, every *visited* leaf replies with the message *good*. While *bad* is always forwarded up to the root, *good* is only forwarded by nodes that receive this message from all of their children. If the root receives only *good*, then it knows that the current cycle is Hamiltonian and it switches to the halting state YES. The information is then broadcast through the entire graph, so that all nodes eventually accept. Otherwise, the root sends the *backtrack* signal to its predecessor, and the search for a Hamiltonian cycle continues. In case there is none (in particular, if there is not even an arbitrary cycle), the root will eventually receive the *backtrack* signal from its greatest neighbor, which indicates that all possibilities have been exhausted. If this happens, the root switches to the halting state NO, and all other nodes eventually do the same. $\square$

4 Functional Fixpoint Logic

In order to introduce functional fixpoint logic, we first give a definition of first-order logic that suits our needs. Formulas will always be evaluated on *ordered*, undirected, connected, I-labeled graphs, where I is a fixed finite set of labels.

Throughout this paper, let $\mathcal{N}$ be an infinite supply of *node variables* and $\mathcal{F}$ be an infinite supply of *function variables*; we refer to them collectively as *variables*. The corresponding set of *terms* is generated by the grammar $t ::= x \mid f(t)$, where $x \in \mathcal{N}$ and $f \in \mathcal{F}$. With this, the set of *formulas* of *first-order logic* over I is given by the grammar

$$\varphi ::= \langle a \rangle\, t \mid s < t \mid s \leftrightarrow t \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\, \varphi,$$

where s and t are terms, $a \in I$, and $x \in \mathcal{N}$. As usual, we may also use the additional operators $\wedge$, $\Rightarrow$, $\Leftrightarrow$, $\forall$ to make our formulas more readable, and we define the notations $s \leq t$, $s = t$, and $s \neq t$ as abbreviations for $\neg(t < s)$, $(s \leq t) \wedge (t \leq s)$, and $\neg(s = t)$, respectively.

The sets of *free variables* of a term t and a formula φ are denoted by free(t) and free(φ), respectively. While node variables can be bound by the usual quantifiers $\exists$ and $\forall$, function variables can be bound by a partial fixpoint operator that we will introduce below.

To interpret a formula φ on an I-labeled graph $(G, \mathfrak{q})$ with $G = (V, E)$, we are given a *variable assignment* σ for the variables that occur freely in φ. This is a partial function $\sigma \colon \mathcal{N} \cup \mathcal{F} \to V \cup V^V$ such that $\sigma(x) \in V$ if x is a free node variable and $\sigma(f) \in V^V$ if f is a free function variable. We call $\sigma(x)$ and $\sigma(f)$ the *interpretations* of x and f under σ, and denote them by x^σ and f^σ, respectively. For a composite term t, the corresponding interpretation t^σ under σ is defined in the obvious way.

We write $(G, \mathfrak{q}), \sigma \models \varphi$ to denote that $(G, \mathfrak{q})$ *satisfies* φ under assignment σ. If φ does not contain any free variables, we simply write $(G, \mathfrak{q}) \models \varphi$ and refer to the set P of I-labeled graphs that satisfy φ as the graph property *defined* by φ. Naturally enough, we say that two devices (i.e., automata or formulas) are *equivalent* if they specify (i.e., decide or define) the same graph property and that two classes of devices are equivalent if their members specify the same class of graph properties.

As we assume that the reader is familiar with first-order logic, we only define the semantics of the atomic formulas (whose syntax is not completely standard):

$$\begin{array}{llll}
(G,\mathfrak{q}),\sigma \models \langle a\rangle\, t & \text{iff} & \mathfrak{q}(t^\sigma) = a & (\text{“}t \text{ has label } a\text{”}), \\
(G,\mathfrak{q}),\sigma \models s < t & \text{iff} & s^\sigma < t^\sigma & (\text{“}s \text{ is smaller than } t\text{”}), \\
(G,\mathfrak{q}),\sigma \models s \leftrightarrow t & \text{iff} & \{s^\sigma, t^\sigma\} \in E & (\text{“}s \text{ and } t \text{ are adjacent”}).
\end{array}$$

We now turn to *functional fixpoint logic*. Syntactically, it is defined as the extension of first-order logic that allows us to write formulas of the form

$$\text{pfp}\begin{bmatrix} f_1\colon\ \varphi_1(f_1,\dots,f_\ell,\textsc{in},\textsc{out}) \\ \vdots \\ f_\ell\colon\ \varphi_\ell(f_1,\dots,f_\ell,\textsc{in},\textsc{out}) \end{bmatrix}\psi\,, \qquad (*)$$

where $f_1,\dots,f_\ell \in \mathcal{F}$, $\textsc{in},\textsc{out} \in \mathcal{N}$, and $\varphi_1,\dots,\varphi_\ell,\psi$ are formulas. We use the notation "$\varphi_i(f_1,\dots,f_\ell,\textsc{in},\textsc{out})$" to emphasize that $f_1,\dots,f_\ell,\textsc{in},\textsc{out}$ may occur freely in φ_i (possibly among other variables). The free variables of formula $(*)$ are given by $\bigcup_{i\in(\ell]}\big[\text{free}(\varphi_i)\setminus\{f_1,\dots,f_\ell,\textsc{in},\textsc{out}\}\big] \cup \big[\text{free}(\psi)\setminus\{f_1,\dots,f_\ell\}\big]$.

The idea is that the *partial fixpoint operator* pfp binds the function variables $f_1,\dots,f_\ell$. The ℓ lines in square brackets constitute a system of function definitions that provide an interpretation of $f_1,\dots,f_\ell$, using the special node variables $\textsc{in}$ and $\textsc{out}$ as helpers to represent input and output values. This is why pfp also binds any free occurrences of $\textsc{in}$ and $\textsc{out}$ in $\varphi_1,\dots,\varphi_\ell$, but not in ψ.

To specify the semantics of $(*)$, we first need to make some preliminary observations. As before, we consider a fixed I-labeled graph $(G,\mathfrak{q})$ with $G=(V,E)$ and assume that we are given a variable assignment σ for the free variables of $(*)$. With respect to $(G,\mathfrak{q})$ and σ, each formula φ_i induces an operator $F_{\varphi_i}\colon (V^V)^\ell \to V^V$ that takes some interpretation of the function variables $f_1,\dots,f_\ell$ and outputs a new interpretation of f_i, corresponding to the function graph defined by φ_i via the node variables $\textsc{in}$ and $\textsc{out}$. For inputs on which φ_i does not define a functional relationship, the new interpretation of f_i behaves like the identity function. More formally, given a variable assignment $\hat{\sigma}$ that extends σ with interpretations of $f_1,\dots,f_\ell$, the operator F_{φ_i} maps $f_1^{\hat{\sigma}},\dots,f_\ell^{\hat{\sigma}}$ to the function f_i^{new} such that for all $u \in V$,

$$f_i^{\text{new}}(u) = \begin{cases} v & \text{if } v \text{ is the unique node in } V \text{ s.t. } (G,\mathfrak{q}),\hat{\sigma}[\textsc{in},\textsc{out}\mapsto u,v] \models \varphi_i, \\ u & \text{otherwise.} \end{cases}$$

Here, $\hat{\sigma}[\text{IN}, \text{OUT} \mapsto u, v]$ is the extension of $\hat{\sigma}$ interpreting IN as u and OUT as v.

In this way, the operators $F_{\varphi_1}, \ldots, F_{\varphi_\ell}$ give rise to an infinite sequence $(f_1^k, \ldots, f_\ell^k)_{k \geq 0}$ of tuples of functions, called *stages*, where the initial stage contains solely the identity function id_V and each subsequent stage is obtained from its predecessor by componentwise application of the operators. More formally,

$$f_i^0 = \mathrm{id}_V = \{u \mapsto u \mid u \in V\} \qquad \text{and} \qquad f_i^{k+1} = F_{\varphi_i}(f_1^k, \ldots, f_\ell^k),$$

for $i \in (\ell]$ and $k \geq 0$. Now, since we have not imposed any restrictions on the formulas φ_i, this sequence might never stabilize, i.e, it is possible that $(f_1^k, \ldots, f_\ell^k) \neq (f_1^{k+1}, \ldots, f_\ell^{k+1})$ for all $k \geq 0$. Otherwise, the sequence reaches a (simultaneous) fixpoint at some position k no greater than $|V|^{|V| \cdot \ell}$ (the number of ℓ-tuples of functions on V).

We define the *partial fixpoint* $(f_1^\infty, \ldots, f_\ell^\infty)$ of the operators $F_{\varphi_1}, \ldots, F_{\varphi_\ell}$ to be the reached fixpoint if it exists, and the tuple of identity functions otherwise. That is, for $i \in (\ell]$,

$$f_i^\infty = \begin{cases} f_i^k & \text{if there exists } k \geq 0 \text{ such that } f_j^k = f_j^{k+1} \text{ for all } j \in (\ell], \\ \mathrm{id}_V & \text{otherwise.} \end{cases}$$

Having introduced the necessary background, we can finally provide the semantics of the formula $\mathrm{pfp}[f_i \colon \varphi_i]_{i \in (\ell]}\, \psi$ presented in $(*)$:

$$(G, \mathfrak{q}), \sigma \models \mathrm{pfp}[f_i \colon \varphi_i]_{i \in (\ell]}\, \psi \qquad \text{iff} \qquad (G, \mathfrak{q}), \sigma[f_i \mapsto f_i^\infty]_{i \in (\ell]} \models \psi,$$

where $\sigma[f_i \mapsto f_i^\infty]_{i \in (\ell]}$ is the extension of σ that interprets f_i as f_i^∞, for $i \in (\ell]$. In other words, the formula $\mathrm{pfp}[f_i \colon \varphi_i]_{i \in (\ell]}\, \psi$ can intuitively be read as

"if $f_1, \ldots, f_\ell$ are interpreted as the partial fixpoint of $\varphi_1, \ldots, \varphi_\ell$, then ψ holds".

Syntactic Sugar

Before we consider a concrete formula (in Example 8), we first introduce some "syntactic sugar" to make using functional fixpoint logic more pleasant.

Set variables. According to our definition of functional fixpoint logic, the operator pfp can bind only function variables. However, functions can be used to encode sets of nodes in a straightforward manner: any set U may be represented by a function that maps nodes outside of U to themselves and nodes inside U to nodes distinct from themselves. Therefore, we may fix an infinite supply $\mathcal{S}$ of *set variables*, and extend the syntax of first-order logic to allow atomic formulas of the form $t \in X$, where t is a term and X is a set variable in $\mathcal{S}$. Naturally, the semantics is that "t is an element of X". To bind set variables, we can then write partial fixpoint formulas of the form $\mathrm{pfp}\big[(f_i \colon \varphi_i)_{i \in (\ell]}, (X_i \colon \vartheta_i)_{i \in (m]}\big]\, \psi$, where $f_1, \ldots, f_\ell \in \mathcal{F}$, $X_1, \ldots, X_m \in \mathcal{S}$, and $\varphi_1, \ldots, \varphi_\ell, \vartheta_1, \ldots, \vartheta_m, \psi$ are formulas. The stages of the partial fixpoint induction are computed as before, but each set variable X_i is initialized to $\emptyset$, and falls back to $\emptyset$ in case the sequence of stages does not converge to a fixpoint.

Quantifiers over functions and sets. Partial fixpoint inductions allow us to iterate over various interpretations of function and set variables and thus provide a way of expressing (second-order) quantification over functions and sets. Since we restrict ourselves to graphs whose nodes are totally ordered, we can easily define a suitable order of iteration and a corresponding partial fixpoint induction that traverses all possible interpretations of a given function or set variable. To make this more convenient, we enrich the language of functional fixpoint logic with second-order quantifiers, allowing us to write formulas of the form $\exists f\, \varphi$ and $\exists X\, \varphi$, where $f \in \mathcal{F}$, $X \in \mathcal{S}$, and φ is a formula. Obviously, the semantics is that "there exists a function f, or a set X, respectively, such that φ holds".

As a consequence, it is possible to express any graph property definable in *monadic second-order logic*, the extension of first-order logic with set quantifiers.

Corollary 7. *When restricted to finite graphs equipped with a total order, functional fixpoint logic is strictly more expressive than monadic second-order logic.*

The strictness of the inclusion in the above corollary follows from the fact that even on totally ordered graphs, Hamiltonicity cannot be defined in monadic second-order logic (see, e.g., the proof in [4, Prp. 5.13]). As the following example shows, this property is easy to express in functional fixpoint logic.

Example 8 (Hamiltonian cycle). The following formula of functional fixpoint logic defines the graph property of Hamiltonicity. That is, an unlabeled graph G satisfies this formula if and only if there exists a cycle in G that goes through each node exactly once.

$$\exists f \left[\begin{array}{l} \forall x \big(f(x) \leftrightarrow x\big) \;\wedge\; \forall x\, \exists y \big[f(y) = x \;\wedge\; \forall z \big(f(z) = x \;\Rightarrow\; z = y\big)\big] \;\wedge \\ \forall X \Big(\big[\exists x (x \in X) \;\wedge\; \forall y \big(y \in X \Rightarrow f(y) \in X\big)\big] \;\Rightarrow\; \forall y (y \in X) \Big) \end{array} \right]$$

Here, $x, y, z \in \mathcal{N}$, $X \in \mathcal{S}$, and $f \in \mathcal{F}$. Intuitively, we represent a given Hamiltonian cycle by a function f that tells us for each node x, which of x's neighbors we should visit next in order to traverse the entire cycle. Thus, f actually represents a directed version of the cycle.

To ensure the existence of a Hamiltonian cycle, our formula states that there is a function f satisfying the following two conditions. By the first line, each node x must have exactly one f-predecessor and one f-successor, both of which must be neighbors of x. By the second line, if we start at any node x and collect into a set X all the nodes reachable from x (by following the path specified by f), then X must contain all nodes. □

5 Translating Between Automata and Logic

Having introduced both automata and logic, we can proceed to explain the first part of Theorem 1 (stated in Sect. 1), i.e., how distributed register automata can be translated into functional fixpoint logic.

Proposition 9. *For every distributed register automaton that decides a graph property, we can construct an equivalent formula of functional fixpoint logic.*

Proof (sketch). Given a distributed register automaton $A = (Q, R, \iota, \Delta, H, o)$ deciding a graph property P over label set I, we can construct a formula φ_A of functional fixpoint logic that defines P. For each state $q \in Q$, our formula uses a set variable X_q to represent the set of nodes of the input graph that are in state q. Also, for each register $r \in R$, it uses a function variable f_r to represent the function that maps each node u to the node v whose identifier is stored in u's register r. By means of a partial fixpoint operator, we enforce that on any I-labeled graph $(G, \mathfrak{q})$, the final interpretations of $(X_q)_{q\in Q}$ and $(f_r)_{r\in R}$ represent the halting configuration reached by A on $(G, \mathfrak{q})$. The main formula is simply

$$\varphi_A \coloneqq \text{pfp}\begin{bmatrix}(X_q \colon \varphi_q)_{q\in Q}\\ (f_r \colon \varphi_r)_{r\in R}\end{bmatrix} \forall x \Big(\bigvee_{p \in H \colon o(p) = \text{YES}} x \in X_p \Big),$$

which states that all nodes end up in a halting state that outputs YES.

Basically, the subformulas $(\varphi_q)_{q\in Q}$ and $(\varphi_r)_{r\in R}$ can be constructed in such a way that for all $i \in \mathbb{N}$, the $(i+1)$-th stage of the partial fixpoint induction represents the configuration reached by A in the i-th round. To achieve this, each of the subformulas contains a nested partial fixpoint formula describing the result computed by the transition maker Δ between two consecutive synchronous rounds, using additional set and function variables to encode the inner configurations of Δ at each node. Thus, each stage of the nested partial fixpoint induction corresponds to a single step in the transition maker's sequential scanning process. □

Let us now consider the opposite direction and sketch how to go from functional fixpoint logic to distributed register automata.

Proposition 10. *For every formula of functional fixpoint logic that defines a graph property, we can construct an equivalent distributed register automaton.*

Proof (sketch). We proceed by structural induction: each subformula φ will be evaluated by a dedicated automaton A_φ, and several such automata can then be combined to build an automaton for a composite formula. For this purpose, it is convenient to design *centralized* automata, which operate on a givenspanning tree (as computed in Example 5) and are coordinated by the root in a fairly sequential manner. In A_φ, each free node variable x of φ is represented by a corresponding input register x whose value at the root is the current interpretation x^σ of x. Similarly, to represent a function variable f, every node v has a register f storing $f^\sigma(v)$. The nodes also possess some auxiliary registers whose purpose will be explained below. In the end, for any formula φ (potentially with free variables), we will have an automaton A_φ computing a transduction $T_{A_\varphi} \colon \mathbb{C}(I, \{\textit{parent}, \textit{root}\} \cup \text{free}(\varphi)) \to \mathbb{C}(\{\text{YES}, \text{NO}\}, \emptyset)$, where *parent* and *root* are supposed to constitute a spanning tree. The computation is triggered by the root, which means that the other nodes are waiting for a signal to wake up.

Algorithm 2. A_φ for $\varphi = \mathrm{pfp}[f_i\colon \varphi_i]_{i\in[1:\ell]}\,\psi$, as controlled by the root

```
1 init($A_{\mathrm{inc}}$)
2 repeat
3     @every node do for $i \in [1:\ell]$ do $f_i \leftarrow f_i^{\mathrm{new}}$
4     for $i \in [1:\ell]$ do $\mathit{update}(f_i^{\mathrm{new}})$
5     if @every node $(\forall i \in [1:\ell] : f_i^{\mathrm{new}} = f_i)$ then goto 8
6 until execute($A_{\mathrm{inc}}$) returns NO     /* until global counter at maximum */
7 @every node do for $i \in [1:\ell]$ do $f_i \leftarrow \mathit{self}$
8 execute($A_\psi$)
```

Essentially, the nodes involved in the evaluation of φ collect some information, send it towards the root, and go back to sleep. The root then returns YES or NO, depending on whether or not φ holds in the input graph under the variable assignment provided by the input registers. Centralizing A_φ in that way makes it very convenient (albeit not efficient) to evaluate composite formulas. For example, in $A_{\varphi\vee\psi}$, the root will first run A_φ, and then A_ψ in case A_φ returns NO.

The evaluation of atomic formulas is straightforward. So let us focus on the most interesting case, namely when $\varphi = \mathrm{pfp}[f_i\colon \varphi_i]_{i\in(\ell]}\,\psi$. The root's program is outlined in Algorithm 2. Line 1 initializes a counter that ranges from 0 to $n^{\ell n}-1$, where n is the number of nodes in the input graph. This counter is distributed in the sense that every node has some dedicated registers that together store the current counter value. Every execution of A_{inc} will increment the counter by 1, or return NO if its maximum value has been exceeded. Now, in each iteration of the loop starting at Line 2, all registers f_i and f_i^{new} are updated in such a way that they represent the current and next stage, respectively, of the partial fixpoint induction. For the former, it suffices that every node copies, for all i, the contents of f_i^{new} to f_i (Line 3). To update f_i^{new}, Line 4 calls a subroutine $\mathit{update}(f_i^{\mathrm{new}})$ whose effect is that $f_i^{\mathrm{new}} = F_{\varphi_i}((f_i)_{i\in(\ell]})$ for all i, where $F_{\varphi_i} : (V^V)^\ell \to V^V$ is the operator defined in Sect. 4. Line 5 checks whether we have reached a fixpoint: The root asks every node to compare, for all i, its registers f_i^{new} and f_i. The corresponding truth value is propagated back to the root, where *false* is given preference over *true*. If the result is *true*, we exit the loop and proceed with calling A_ψ to evaluate ψ (Line 8). Otherwise, we try to increment the global counter by executing A_{inc} (Line 6). If the latter returns NO, the fixpoint computation is aborted because we know that it has reached a cycle. In accordance with the partial fixpoint semantics, all nodes then write their own identifier to every register f_i (Line 7) before ψ is evaluated (Line 8). $\square$

6 Conclusion

This paper makes some progress in the development of a descriptive distributed complexity theory by establishing a logical characterization of a wide class of network algorithms, modeled as distributed register automata.

In our translation from logic to automata, we did not pay much attention to algorithmic efficiency. In particular, we made extensive use of centralized subroutines that are triggered and controlled by a leader process. A natural question for future research is to identify cases where we can understand a distributed architecture as an opportunity that allows us to evaluate formulas faster. In other words, is there an expressive fragment of functional fixpoint logic that gives rise to efficient distributed algorithms in terms of running time? What about the required number of messages? We are then entering the field of automatic *synthesis of practical distributed algorithms* from logical specifications. This is a worthwhile task, as it is often much easier to declare what should be done than how it should be done (cf. Examples 6 and 8).

As far as the authors are aware, this area is still relatively unexplored. However, one noteworthy advance was made by Grumbach and Wu in [9], where they investigated distributed evaluation of first-order formulas on bounded-degree graphs and planar graphs. We hope to follow up on this in future work.

Acknowledgments. We thank Matthias Függer for helpful discussions. Work supported by ERC *EQualIS* (FP7-308087) (http://www.lsv.fr/~bouyer/equalis) and ANR *FREDDA* (17-CE40-0013) (https://www.irif.fr/anr/fredda/index).

References

1. Abiteboul, S., Vianu, V.: Fixpoint extensions of first-order logic and datalog-like languages. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989), Pacific Grove, California, USA, 5–8 June 1989, pp. 71–79. IEEE Computer Society (1989). https://doi.org/10.1109/LICS.1989.39160
2. Aiswarya, C., Bollig, B., Gastin, P.: An automata-theoretic approach to the verification of distributed algorithms. In: Aceto, L., de Frutos-Escrig, D. (eds.) 26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, 14 September 2015. LIPIcs, vol. 42, pp. 340–353. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015). https://doi.org/10.4230/LIPIcs.CONCUR.2015.340
3. Aiswarya, C., Bollig, B., Gastin, P.: An automata-theoretic approach to the verification of distributed algorithms. Inf. Comput. **259**(Part 3), 305–327 (2018). https://doi.org/10.1016/j.ic.2017.05.006
4. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach. Encyclopedia of Mathematics and Its Applications, vol. 138. Cambridge University Press, Cambridge (2012). https://hal.archives-ouvertes.fr/hal-00646514. https://doi.org/10.1017/CBO9780511977619
5. Diestel, R.: Graph Theory. GTM, vol. 173. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-53622-3
6. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. In: Karp, R.M. (ed.) Complexity of Computation. SIAM-AMS Proceedings, vol. 7, pp. 43–73 (1974). http://www.almaden.ibm.com/cs/people/fagin/genspec.pdf
7. Feuilloley, L., Fraigniaud, P.: Survey of distributed decision. Bull. EATCS **119** (2016). http://eatcs.org/beatcs/index.php/beatcs/article/view/411
8. Grädel, E., et al.: Finite Model Theory and Its Applications. Texts in Theoretical Computer Science. An EATCS Series, 1st edn. Springer, Heidelberg (2007). https://doi.org/10.1007/3-540-68804-8

9. Grumbach, S., Wu, Z.: Logical locality entails frugal distributed computation over graphs (extended abstract). In: Paul, C., Habib, M. (eds.) WG 2009. LNCS, vol. 5911, pp. 154–165. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11409-0_14
10. Hella, L., et al.: Weak models of distributed computing, with connections to modal logic. In: Kowalski, D., Panconesi, A. (eds.) ACM Symposium on Principles of Distributed Computing, PODC 2012, Funchal, Madeira, Portugal, 16–18 July 2012, pp. 185–194. ACM (2012). https://doi.org/10.1145/2332432.2332466
11. Hella, L., et al.: Weak models of distributed computing, with connections to modallogic. Distrib. Comput. **28**(1), 31–53 (2015). https://arxiv.org/abs/1205.2051. http://dx.doi.org/10.1007/s00446-013-0202-3
12. Immerman, N.: Descriptive Complexity. Texts in Computer Science. Springer, New York (1999). https://doi.org/10.1007/978-1-4612-0539-5
13. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994). https://doi.org/10.1016/0304-3975(94)90242-9
14. Kuusisto, A.: Modal logic and distributed message passing automata. In: Rocca, S.R.D. (eds.) Computer Science Logic 2013 (CSL 2013), Torino, Italy, 2–5 September 2013, LIPIcs, vol. 23, pp. 452–468. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013). https://doi.org/10.4230/LIPIcs.CSL.2013.452
15. Libkin, L., et al.: Elements of Finite Model Theory. Texts in Theoretical Computer Science. An EATCS Series, 1st edn. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07003-1
16. Naor, M., Stockmeyer, L.J.: What can be computed locally? SIAM J. Comput. **24**(6), 1259–1277 (1995). https://doi.org/10.1137/S0097539793254571
17. Reiter, F.: Distributed graph automata. In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, 6–10 July 2015, pp. 192–201. IEEE Computer Society (2015). https://arxiv.org/abs/1408.3030. https://doi.org/10.1109/LICS.2015.27
18. Reiter, F.: Asynchronous distributed automata: a characterization of the modal MU-fragment. In: Chatzigiannakis, I., Indyk, P., Kuhn, F., Muscholl, A. (eds.) 44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, Warsaw, Poland, 10–14 July 2017. LIPIcs, vol. 80, pp. 100:1–100:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). http://arxiv.org/abs/1611.08554. https://doi.org/10.4230/LIPIcs.ICALP.2017.100
19. Vardi, M.Y.: The complexity of relational query languages (extended abstract). In: Lewis, H.R., Simons, B.B., Burkhard, W.A., Landweber, L.H. (eds.) Proceedings of the 14th Annual ACM Symposium on Theory of Computing, San Francisco, California, USA, 5–7 May 1982, pp. 137–146. ACM (1982). https://doi.org/10.1145/800070.802186