Identifiers in Registers*
Describing Network Algorithms with Logic

Benedikt Bollig, Patricia Bouyer, and Fabian Reiter

LSV, CNRS, ENS Paris-Saclay, Université Paris-Saclay, France
bollig@lsv.fr, 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 \text{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 \text{np}. Subsequently, many other popular classes, such as \text{p}, \text{pspace}, and \text{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 [18], which states that on structures equipped with a total order relation, the properties decidable in \text{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 rules. If the

* Built with the knowledge package: technical terms are hyperlinked to their definitions.
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 syn-
chronously 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], [17] and [16] 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. Similarly, register contents can be copied, but no new values can be
generated. Since the original motivation for this 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 imme-
diately 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 Section 2, we formally introduce
distributed register automata in Section 3 and functional fixpoint logic in Section 4.
We then sketch the proof of Theorem 1 in Section 5, and conclude in Section 6.

## 2 Preliminaries

We denote the empty set by $\emptyset$, the set of nonnegative integers by $\mathbb{N} = \{0, 1, 2, \ldots\}$, and the set of integers by $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$. 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 $[m : n] := \{i \in \mathbb{Z} \mid m \leq i \leq n\}$ 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., \( \langle m : n \rangle := [m : n] \setminus \{m\} \). Furthermore, if we omit the first endpoint, it defaults to 0. This gives us shorthand notations such as \( [n] := [0 : n] \) and \( [n) := [0 : n) = [0 : n - 1] \).

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 \( \Sigma \)-configuration as a tuple \( C = (G, q, r) \), where \( G = (V, E) \) is a graph, called the underlying graph of \( C \), \( q : V \to Q \) is a state function that assigns to each node a state \( q \in Q \), and \( r : V \to V^R \) is a register valuation function that associates with each node a register valuation \( \rho \in V^R \). The set of all \( \Sigma \)-configurations is denoted by \( C(\Sigma) \).

If \( R = \emptyset \), then we are actually dealing with a tuple \( (G, q) \), 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 \( \{e\} \), where \( e \) is our dummy label. In this case, we identify \( (G, q) \) 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 \( (\{1\}, \emptyset) \)-configuration into a \( (\{1\}, \{r\}) \)-configuration. We say that it defines a \( (\{1\}, \emptyset) \)-\( (\{1\}, \{r\}) \)-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 \( (I, \emptyset) \)-\( (\{\text{YES, NO}\}, \emptyset) \)-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 \text{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 \( T : \mathcal{C}(\Sigma^\text{in}) \to \mathcal{C}(\Sigma^\text{out}) \) such that, if defined, \( T(G, q, r) = (G, q', r') \) for some \( q' \) and \( r' \). That is, a transduction does not modify the underlying graph but only the states and register valuations. We denote the set of all \( \Sigma^\text{in}, \Sigma^\text{out} \)-transductions by \( \mathcal{T}(\Sigma^\text{in}, \Sigma^\text{out}) \) and refer to \( \Sigma^\text{in} \) and \( \Sigma^\text{out} \) as the input and output signatures of \( T \). By extension, \( I \) and \( O \) are called the sets of input and output labels, and \( R^\text{in} \) and \( R^\text{out} \) the sets of input and output registers. Similarly, any \( \Sigma^\text{in} \)-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 \( \Sigma^\text{in} = (I, R^\text{in}) \) and \( \Sigma^\text{out} = (O, R^\text{out}) \) be two signatures. A distributed register automaton (or simply automaton) with input signature \( \Sigma^\text{in} \) and output signature \( \Sigma^\text{out} \) is a tuple \( A = (Q, R, i, \Delta, H, o) \) consisting of a nonempty finite set \( Q \) of states, a finite set \( R \) of registers that includes both \( R^\text{in} \) and \( R^\text{out} \), an input function \( i : I \to Q \), a transition maker \( \Delta \) whose specification will be given in Definition 3 below, a set \( H \subseteq Q \) of halting states, and an output function \( o : H \to O \). The registers in \( R \setminus (R^\text{in} \cup R^\text{out}) \) are called auxiliary registers.

Automaton \( A \) computes a transduction \( T_A \in \mathcal{T}(\Sigma^\text{in}, \Sigma^\text{out}) \). 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 \( C = (G, q, r) \), i.e., the underlying graph is always \( G \). As mentioned before, for a node \( v \in V \), we interpret \( q(v) \in Q \) as the current state of \( v \) and \( r(v) \in V^R \) as the current register valuation of \( v \). Abusing notation, we let \( C(v) := (q(v), r(v)) \) and say that \( C(v) \) is the local configuration of \( v \). In Figure 1, the local configuration of node 17 is \((q_1, \{r_1, r_2, r_3 \mapsto 17, 34, 98\})\).

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

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

\[
[\Delta] : (Q \times V^R)^+ \to Q \times V^R
\]

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 \( v_1, \ldots, v_d \) of \( u \)’s neighbors sorted in ascending (identifier) order, i.e., \( v_i < v_{i+1} \) for \( i \in [1 : d] \). (See Figure 1 for an example, where \( u \) corresponds to node 17.) If \( u \) is already in a halting state, i.e., if \( C(u) = (q, r) \in H \times V^R \), then its local configuration does not change anymore, i.e., \( C'(u) = C(u) \). Otherwise, we define \( C'(u) = [\Delta](C(u), C(v_1), \ldots, C(v_d)) \), which we may write more suggestively as

\[
[\Delta] : C(u) \xrightarrow{C(v_1) \ldots C(v_d)} C'(u).
\]
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).

Intuitively, node $u$ updates its own local configuration by using $\Delta$ 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 $C = (G, q, r)$ is called a halting configuration if all nodes are in a halting state, i.e., if $q(v) \in H$ for all $v \in V$. We say that $A$ halts if it reaches a halting configuration.

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

It is now obvious that $A$ defines a transduction $T_A : \mathcal{C}(\Sigma^{\text{in}}) \to \mathcal{C}(\Sigma^{\text{out}})$. If $A$ receives the input configuration $C \in \mathcal{C}(\Sigma^{\text{in}})$ and eventually halts and produces the output configuration $C' \in \mathcal{C}(\Sigma^{\text{out}})$, 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 $\{\text{YES, NO}\}, \emptyset$. 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 $T_A(C) = (G, q, r)$ such that $q(v) = \text{YES}$ for all $v \in V$. Conversely, $A$ rejects $C$ if $T_A(C) = (G, q, r)$ such that $q(v) = \text{NO}$ 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 $\Delta$ works internally.
Definition 3 (Transition maker). Suppose that $A = (Q, R, i, \Delta, H, o)$ is a distributed register automaton. Then its transition maker $\Delta = (\tilde{Q}, \tilde{R}, \tilde{i}, \tilde{\delta}, \tilde{o})$ consists of a nonempty finite set $\tilde{Q}$ of inner states, a finite set $\tilde{R}$ of inner registers that is disjoint from $\tilde{R}$, an inner initial state $\tilde{i} \in \tilde{Q}$, an inner transition function $\tilde{\delta}: \tilde{Q} \times \tilde{Q} \times 2^{(\tilde{R} \cup \tilde{R})^2} \rightarrow \tilde{Q} \times (\tilde{R} \cup \tilde{R})^R$, and an inner output function $\tilde{o}: \tilde{Q} \rightarrow \tilde{Q} \times \tilde{R}$.

Basically, a transition maker $\Delta = (\tilde{Q}, \tilde{R}, \tilde{i}, \tilde{\delta}, \tilde{o})$ is a sequential register automaton (in the spirit of [13]) that reads a nonempty sequence $(\tilde{q}_0, \tilde{r}_0), \ldots, (\tilde{q}_d, \tilde{r}_d) \in (Q \times V R)^+ \times (Q \times V R)^+$ of local configurations of $A$ in order to produce a new local configuration $(q', \rho')$. While reading this sequence, it traverses itself a sequence $(\tilde{q}_0, \tilde{r}_0), \ldots, (\tilde{q}_{d+1}, \tilde{r}_{d+1})$ 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 $\tilde{q}_0 = \tilde{i}$ and $\tilde{\rho}_0(\tilde{r}) = \bot$ for all $\tilde{r} \in \tilde{R}$. Now for $i \in [d]$, when $\Delta$ is in the inner configuration $(\tilde{q}_i, \tilde{\rho}_i)$ and reads the local configuration $(q_i, \rho_i)$, it can compare all values assigned to the inner registers and registers by $\tilde{\rho}_i$ and $\rho_i$ (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 $r, s \in \tilde{R}$ and $s \in R$, we have $r \prec_i s$ if and only if $\tilde{\rho}_i(r) \prec \rho_i(s)$, and analogously for $r \prec_i s$, where $r, s \in \tilde{R}$ and $s \in R$. In particular, if $\tilde{\rho}_i(\tilde{r}) = \bot$, then $\tilde{r}$ is incomparable with respect to $\prec_i$. Equipped with this relation, $\Delta$ 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 $\tilde{\rho}_{i+1}(\tilde{r}) = \tilde{\rho}_i(s)$ if $\tilde{\alpha}(\tilde{r}) = \tilde{s}$, and $\tilde{\rho}_{i+1}(\tilde{r}) = \rho_i(s)$ if $\tilde{\alpha}(\tilde{r}) = s$, where $r, s \in \tilde{R}$ 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 $(q', \rho')$ such that $\tilde{\alpha}(\tilde{q}_{d+1}) = (q', \beta)$ and $\beta(r) = \tilde{r}$ implies $\rho'(r) = \tilde{\rho}_{d+1}(\tilde{r})$. Here we assume without loss of generality that $\Delta$ 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 $A = (Q, R, i, \Delta, H, o)$ with input signature $\Sigma^m = \{(1, 0)\}$ and output signature $\Sigma^{out} = \{1, \{parent, root\}\}$ that computes a (breadth-first) spanning tree of its input graph $G = (V, E)$, rooted at the node with the smallest identifier. More precisely, in the computed output configuration $C = (G, q, r)$, 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 $i: \varepsilon \mapsto 1$ and $o: 3 \mapsto \varepsilon$. In addition to the output registers, each node has an auxiliary register $self$ that will always store its own identifier. Thus, we choose
Algorithm 1  Transition maker of the automaton from Example 5

\[
\begin{aligned}
&\text{if } \exists \text{ neighbor } nb (\text{nb.root} < \text{my.root}): \\
&\quad \text{my.state} \leftarrow 1; \quad \text{my.parent} \leftarrow \text{nb.self}; \quad \text{my.root} \leftarrow \text{nb.root} \\
&\quad \text{Rule 1} \\
&\text{else if } \text{my.state} = 1 \\
&\quad \land \forall \text{ neighbor } nb \left[ \text{nb.root} = \text{my.root} \land \\
&\quad \quad \text{nb.state} = 2 \right] \\
&\quad \text{my.state} \leftarrow 2 \\
&\quad \text{Rule 2} \\
&\text{else if } (\text{my.state} = 2 \land \text{my.root} = \text{my.self}) \lor (\text{my.parent.state} = 3): \\
&\quad \text{my.state} \leftarrow 3 \\
&\quad \text{Rule 3} \\
&\text{else do nothing}
\end{aligned}
\]

\[R = \{\text{self}, \text{parent}, \text{root}\}.\] For the sake of simplicity, we describe the transition maker \(\Delta\) 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.

(An example run and a proof of correctness can be found in Appendix A.) \(\square\)

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 $\Sigma_{\text{in}} = \{1, \{\text{parent}, \text{root}\}\}$ and output signature $\Sigma_{\text{out}} = \{\{\text{YES, NO}\}, \emptyset\}$ that decides if the underlying graph $G = (V, E)$ of its input configuration $C = (G, q, r)$ is Hamiltonian, i.e., whether $G$ contains a cycle that goes through each node exactly once. The automaton works under the assumption that $r$ encodes a valid spanning tree of $G$ in the registers $\text{parent}$ and $\text{root}$, 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 $Q = \{(\text{unvisited}, \text{visited}, \text{backtrack}) \times \{\text{idle}, \text{request}, \text{good}, \text{bad}\}\} \cup H$, with the set of halting states $H = \{\text{YES, NO}\}$. 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 $\iota$ initializes every node to the state $(\text{unvisited}, \text{idle})$, while the output function simply returns the answers chosen by the nodes, i.e., $o: \text{YES} \mapsto \text{YES}, \text{NO} \mapsto \text{NO}$. In addition to the input registers, each node has a register $\text{self}$ storing its own identifier and a register $\text{successor}$ to point to its successor in a (partially constructed) Hamiltonian path. That is, $R = \{\text{self}, \text{parent}, \text{root}, \text{successor}\}$. We now describe the algorithm in an informal way. It is, in principle, easy to implement in the transition maker $\Delta$, but a thorough formalization would be rather cumbersome.

In the first round, the root marks itself as $\text{visited}$ and updates its $\text{successor}$ register to point towards its smallest neighbor (the one with the smallest identifier). Similarly, in each subsequent round, any $\text{unvisited}$ node that is pointed to by one of its neighbors marks itself as $\text{visited}$ and points towards its smallest $\text{unvisited}$ neighbor. However, if all neighbors are already $\text{visited}$, the node instead sends the $\text{backtrack}$ signal to its predecessor and switches back to $\text{unvisited}$ (in the following round). Whenever a $\text{visited}$ node receives the $\text{backtrack}$ signal from its $\text{successor}$, it tries to update its $\text{successor}$ to the next-smallest $\text{unvisited}$ neighbor. If no such neighbor exists, it resets its $\text{successor}$ pointer to itself, propagates the $\text{backtrack}$ signal to its predecessor, and becomes $\text{unvisited}$ 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 $\text{unvisited}$ neighbor, it chooses the root as its $\text{successor}$. This way, the constructed path becomes a cycle. In order to check whether that cycle is Hamiltonian, the root now broadcast a $\text{request}$ down the spanning tree. If the $\text{request}$ reaches an $\text{unvisited}$ node, that node replies by sending the message $\text{bad}$ towards the root. On the other hand, every $\text{visited}$ leaf replies with the message $\text{good}$. While $\text{bad}$ is always forwarded up to the root, $\text{good}$ is only forwarded by nodes that receive this message from all of their children. If the root receives only $\text{good}$, then it knows that the current cycle is Hamiltonian and it switches to the halting state $\text{YES}$. The information is then broadcast through the entire graph, so that all nodes eventually accept. Otherwise, the root sends the $\text{backtrack}$ 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. □

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 \( N \) be an infinite supply of node variables and \( F \) 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 \( x \in N \) and \( f \in F \). With this, the set of formulas of first-order logic over \( I \) is given by the grammar

\[
\varphi ::= (a) t \mid s < t \mid s \varpi t \mid \neg \varphi \mid \varphi \lor \varphi \mid \exists x \varphi,
\]

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

The sets of free variables of a term \( t \) and a formula \( \varphi \) are denoted by free(\( t \)) and free(\( \varphi \)), 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 \( \varphi \) on an I-labeled graph \((G, q)\) with \( G = (V, E) \), we are given a variable assignment \( \sigma \) for the variables that occur freely in \( \varphi \). This is a partial function \( \sigma : N \cup F \rightarrow V \cup V^V \) 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 \( \sigma \), and denote them by \( x^\sigma \) and \( f^\sigma \), respectively. For a composite term \( t \), the corresponding interpretation \( t^\sigma \) under \( \sigma \) is defined in the obvious way.

We write \((G, q), \sigma \models \varphi\) to denote that \((G, q)\) satisfies \( \varphi \) under assignment \( \sigma \). If \( \varphi \) does not contain any free variables, we simply write \((G, q) \models \varphi\) and refer to the set \( P \) of I-labeled graphs that satisfy \( \varphi \) as the graph property defined by \( \varphi \). 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):

\[
\begin{align*}
(G, q), \sigma \models (a) t & \quad \text{iff} \quad q(t^\sigma) = a \quad ("t has label } a") , \\
(G, q), \sigma \models s < t & \quad \text{iff} \quad s^\sigma < t^\sigma \quad ("s is smaller than } t") , \\
(G, q), \sigma \models s \varpi t & \quad \text{iff} \quad \{s^\sigma, t^\sigma\} \in E \quad ("s \text{ and } t \text{ are adjacent}").
\end{align*}
\]
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

\[
pfp \begin{cases} f_1 : \varphi_1(f_1, \ldots, f_\ell, \text{IN, OUT}) \\ \vdots \\ f_\ell : \varphi_\ell(f_1, \ldots, f_\ell, \text{IN, OUT}) \end{cases} \psi, \tag{*}
\]

where \( f_1, \ldots, f_\ell \in \mathcal{F}, \text{IN, OUT} \in \mathcal{N}, \) and \( \varphi_1, \ldots, \varphi_\ell, \psi \) are formulas. We use the notation \( \varphi(f_1, \ldots, f_\ell, \text{IN, OUT}) \) to emphasize that \( f_1, \ldots, f_\ell, \text{IN, OUT} \) may occur freely in \( \varphi_i \) (possibly among other variables). The free variables of formula \((*)\) are given by \( \bigcup_{i \in \{1\}} \{ \text{free}(\varphi_i) \setminus \{ f_1, \ldots, f_\ell, \text{IN, OUT} \} \} \cup \{ \text{free}(\psi) \setminus \{ f_1, \ldots, f_\ell \} \} \).

The idea is that the partial fixpoint operator \( \text{pfp} \) binds the function variables \( f_1, \ldots, f_\ell \). The \( \ell \) lines in square brackets constitute a system of function definitions that provide an interpretation of \( f_1, \ldots, f_\ell \), using the special node variables \( \text{IN} \) and \( \text{OUT} \) as helpers to represent input and output values. This is why \( \text{pfp} \) also binds any free occurrences of \( \text{IN} \) and \( \text{OUT} \) in \( \varphi_1, \ldots, \varphi_\ell \), but not in \( \psi \).

To specify the semantics of \((*)\), we first need to make some preliminary observations. As before, we consider a fixed \( I \)-labeled graph \( (G, q) \) with \( G = (V, E) \) and assume that we are given a variable assignment \( \sigma \) for the free variables of \((*)\). With respect to \( (G, q) \) and \( \sigma \), each formula \( \varphi_i \) induces an operator \( F_{\varphi_i} : (V^V)^\ell \rightarrow V^V \) that takes some interpretation of the function variables \( f_1, \ldots, f_\ell \) and outputs a new interpretation of \( f_i \), corresponding to the function graph defined by \( \varphi_i \) via the node variables \( \text{IN} \) and \( \text{OUT} \). For inputs on which \( \varphi_i \) 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 \( \sigma \) with interpretations of \( f_1, \ldots, f_\ell \), the operator \( F_{\varphi_i} \) maps \( f_1^{\hat{\sigma}}, \ldots, f_\ell^{\hat{\sigma}} \) to the function \( f_i^{\text{new}} \) such that for all \( u \in V \),

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

Here, \( \hat{\sigma}[\text{IN, OUT} \mapsto u, v] \) is the extension of \( \hat{\sigma} \) interpreting \( \text{IN} \) as \( u \) and \( \text{OUT} \) 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 \( \text{id}_V \) and each subsequent stage is obtained from its predecessor by componentwise application of the operators. More formally,

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

for \( i \in \{\ell\} \) and \( k \geq 0 \). Now, since we have not imposed any restrictions on the formulas \( \varphi_i \), 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 \( |V|^{|V|}^\ell \) (the number of \( \ell \)-tuples of functions on \( V \)).

We define the partial fixpoint \((f_{\varphi_1}^\infty, \ldots, f_{\varphi_\ell}^\infty)\) 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 $i \in \{\ell\}$,
\[
f_i^{\infty} = \begin{cases} 
  f_i^k & \text{if there exists } k \geq 0 \text{ such that } f_j^k = f_j^{k+1} \text{ for all } j \in \{\ell\}, \\
  \text{id}_\mathcal{V} & \text{otherwise}.
\end{cases}
\]

Having introduced the necessary background, we can finally provide the semantics of the formula $\text{pfp}[f_i: \phi_i]_{i \in \{\ell\}} \psi$ presented in (*):
\[
(G, q), \sigma \models \text{pfp}[f_i: \phi_i]_{i \in \{\ell\}} \psi \iff (G, q), \sigma[f_i \mapsto f_i^{\infty}]_{i \in \{\ell\}} \models \psi,
\]
where $\sigma[f_i \mapsto f_i^{\infty}]_{i \in \{\ell\}}$ is the extension of $\sigma$ that interprets $f_i$ as $f_i^{\infty}$, for $i \in \{\ell\}$.

In other words, the formula $\text{pfp}[f_i: \phi_i]_{i \in \{\ell\}} \psi$ can intuitively be read as “if $f_1, \ldots, f_\ell$ are interpreted as the partial fixpoint of $\phi_1, \ldots, \phi_\ell$, then $\psi$ 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 $\text{pfp}$ 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 $\text{pfp}[(f_i: \phi_i)_{i \in \{\ell\}}, (X_i: \theta_i)_{i \in \{m\}}] \psi$, where $f_1, \ldots, f_\ell \in \mathcal{F}$, $X_1, \ldots, X_m \in \mathcal{S}$, and $\phi_1, \ldots, \phi_\ell, \theta_1, \ldots, \theta_m, \psi$ 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. (More details can be found in Appendix B.1.)

**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 $\exists f \phi$ and $\exists X \phi$, where $f \in \mathcal{F}$, $X \in \mathcal{S}$, and $\phi$ is a formula. The semantics is the standard one. (More details can be found in Appendix B.2.)

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.

$$\exists f \left[ \forall x(f(x) \Rightarrow x) \land \forall x \exists y[f(y) = x \land \forall z(f(z) = x \Rightarrow z = y)] \land \forall X \left( \exists x(x \in X) \land \forall y(y \in X \Rightarrow f(y) \in X) \Rightarrow \forall y(y \in X) \right) \right]$$

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 Section 1), i.e., how distributed register automata can be translated into functional fixpoint logic (see Appendix C for a full proof).

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 $\varphi_A$ 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 $(G, q)$, the final interpretations of $(X_q)_{q \in Q}$ and $(f_r)_{r \in R}$ represent the halting configuration reached by $A$ on $(G, q)$. The main formula is simply

$$\varphi_A := \text{pfp} \left[ (X_q : \varphi_q)_{q \in Q} \right] \forall x \left( \exists x' \in X_q \right) \land \forall r(f_r : \varphi_r)_{r \in R} \forall x \left( \neg \left( \exists y : y \neq x \land f_r(x) = y \right) \land o(x) = \text{YES} \right)$$
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 \(\Delta\) between two consecutive synchronous rounds, using additional set and function variables to encode the inner configurations of \(\Delta\) 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. \(\square\)

Let us now consider the opposite direction and sketch how to go from functional fixpoint logic to distributed register automata (see Appendix D for a full proof).

**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 \(\varphi\) will be evaluated by a dedicated automaton \(A_{\varphi}\), 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 given spanning tree (as computed in Example 5) and are coordinated by the root in a fairly sequential manner. In \(A_{\varphi}\), each free node variable \(x\) of \(\varphi\) is represented by a corresponding input register \(x\) whose value at the root is the current interpretation \(x^\sigma\) of \(x\). Similarly, to represent a function variable \(f\), every node \(v\) has a register \(f^\sigma(v)\). The nodes also possess some auxiliary registers whose purpose will be explained below. In the end, for any formula \(\varphi\) (potentially with free variables), we will have an automaton \(A_{\varphi}\) computing a transduction \(T_{A_{\varphi}}: \mathcal{C}(I, \{\text{parent}, \text{root}\} \cup \text{free}(%(\varphi))) \rightarrow \mathcal{C}(\{\text{YES}, \text{NO}\}, \emptyset)\), where \(\text{parent}\) and \(\text{root}\) 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. Essentially, the nodes involved in the evaluation of \(\varphi\) 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 \(\varphi\) holds in the input graph under the variable assignment provided by the input registers. Centralizing \(A_{\varphi}\) in that way makes it very convenient (albeit not efficient) to evaluate composite formulas. For example, in \(A_{\varphi \lor \psi}\), the root will first run \(A_{\varphi}\), and then \(A_{\psi}\) in case \(A_{\varphi}\) returns NO.

The evaluation of atomic formulas is straightforward. So let us focus on the most interesting case, namely when \(\varphi = \text{pfp}[f_i: \varphi_i]_{i \in \ell} \psi\). The root’s program is outlined in Algorithm 2. Line 1 initializes a counter that ranges from 0 to \(n^n-1\), 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 \(A_{\text{inc}}\) 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 \(f_i^{\text{new}}\) 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 \(f_i^{\text{new}}\) to \(f_i\) (Line 3). To update \(f_i^{\text{new}}\), Line 4 calls a subroutine \text{update}(f_i^{\text{new}})\)
### Algorithm 2

For $\varphi = \text{pfp}[f_i; \varphi_i]_{i \in [1:L]} \psi$, as controlled by the root

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

whose effect is that $f_i^{\text{new}} = F_{\varphi_i}(f_i)_{i \in [1:L]}$ for all $i$, where $F_{\varphi_i} : (\mathcal{V}^V)^L \rightarrow \mathcal{V}^V$ is the operator defined in Section 4. Line 5 checks whether we have reached a fixpoint: The root asks every node to compare, for all $i$, its registers $f_i^{\text{new}}$ 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 $A_\psi$ to evaluate $\psi$ (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 $\psi$ is evaluated (Line 8). □

### 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) and ANR FREDDA (17-CE40-0013).
References

1. Serge Abiteboul and Victor Vianu. Fixpoint extensions of first-order logic and datalog-like languages. In Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS ’89), Pacific Grove, California, USA, June 5-8, 1989, pages 71–79. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39160.

2. C. Aiswarya, Benedikt Bollig, and Paul Gastin. An automata-theoretic approach to the verification of distributed algorithms. In Luca Aceto and David de Frutos-Escrig, editors, 26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1–4, 2015, volume 42 of LIPIcs, pages 340–353. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.CONCUR.2015.340.

3. C. Aiswarya, Benedikt Bollig, and Paul Gastin. An automata-theoretic approach to the verification of distributed algorithms. Inf. Comput., 259(Part):305–327, 2018. doi:10.1016/j.ic.2017.05.006.

4. Bruno Courcelle and Joost Engelfriet. Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach, volume 138 of Encyclopedia of mathematics and its applications. Cambridge University Press, 2012. URL: https://hal.archives-ouvertes.fr/hal-00646514, doi:10.1017/CBO9780511977619.

5. Reinhard Diestel. Graph Theory, 5th Edition, volume 173 of Graduate texts in mathematics. Springer, 2017. URL: http://diestel-graph-theory.com, doi:10.1007/978-3-662-53622-3.

6. Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In Richard M. Karp, editor, Complexity of Computation, volume 7 of SIAM-AMS Proceedings, pages 43–73, 1974. URL: http://www.almaden.ibm.com/cs/people/fagin/genspec.pdf.

7. Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. Bulletin of the EATCS, 119, 2016. URL: http://eatcs.org/beats/index.php/beats/article/view/411.

8. Erich Grädel, Phokion G. Kolaitis, Leonid Libkin, Maarten Marx, Joel Spencer, Moshe Y. Vardi, Yde Venema, and Scott Weinstein. Finite Model Theory and Its Applications. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2007. doi:10.1007/3-540-68804-8.

9. Stéphane Grumbach and Zhilin Wu. Logical locality entails frugal distributed computation over graphs (extended abstract). In Graph-Theoretic Concepts in Computer Science, 35th International Workshop, WG 2009, Montpellier, France, June 24-26, 2009. Revised Papers, volume 5911 of Lecture Notes in Computer Science, pages 154–165, 2009. doi:10.1007/978-3-642-11409-0_14.

10. Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempääläinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. In Darek Kowalski and Alessandro Panconesi, editors, ACM Symposium on Principles of Distributed Computing, PODC ’12, Funchal, Madeira, Portugal, July 16-18, 2012, pages 185–194. ACM, 2012. doi:10.1145/2332432.2332466.

11. Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempääläinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. Distributed Computing, 28(1):31–53, 2015. URL: https://arxiv.org/abs/1205.2051, doi:10.1007/s00446-013-0202-3.
12. Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999. doi:10.1007/978-1-4612-0539-5.

13. Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. doi:10.1016/0304-3975(94)90242-9.

14. Antti Kuusisto. Modal logic and distributed message passing automata. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, CSL 2013, September 2-5, 2013, Torino, Italy, volume 23 of LIPIcs, pages 452–468. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. doi:10.4230/LIPIcs.CSL.2013.452.

15. Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07003-1.

16. Fabian Reiter. Distributed graph automata. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015*, Kyoto, Japan, July 6-10, 2015, pages 192–201. IEEE Computer Society, 2015. URL: https://arxiv.org/abs/1408.3030, doi:10.1109/LICS.2015.27.

17. Fabian Reiter. Asynchronous distributed automata: A characterization of the modal mu-fragment. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, July 10-14, 2017, Warsaw, Poland, volume 80 of LIPIcs, pages 100:1–100:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. URL: http://arxiv.org/abs/1611.08554, doi:10.4230/LIPIcs.ICALP.2017.100.

18. Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982. doi:10.1145/800070.802186.
Algorithm 1 Transition maker of the automaton from Example 5

\begin{algorithmic}
  \If {$\exists$ neighbor \textit{nb} (\textit{nb}.\textit{root} < \textit{my}.\textit{root})} \\
  \State $\textit{my}.\textit{state} \gets 1$; $\textit{my}.\textit{parent} \gets \textit{nb}.\textit{self}$; $\textit{my}.\textit{root} \gets \textit{nb}.\textit{root}$ \\
  \EndIf

  \ElseIf {$\textit{my}.\textit{state} = 1$ \land $\forall$ neighbor \textit{nb} \left( (\textit{nb}.\textit{root} = \textit{my}.\textit{root} \And (\textit{nb}.\textit{parent} \neq \textit{my}.\textit{self} \Or \textit{nb}.\textit{state} = 2) \right)$} \\
  \State $\textit{my}.\textit{state} \gets 2$

  \ElseIf {$\left( \textit{my}.\textit{state} = 2 \land \textit{my}.\textit{root} = \textit{my}.\textit{self} \right) \lor (\textit{my}.\textit{parent}.\textit{state} = 3)$} \\
  \State $\textit{my}.\textit{state} \gets 3$

  \Else \textbf{do nothing} \EndIf
\end{algorithmic}

\section{Spanning-tree automaton}

\subsection{Run of the spanning-tree automaton}

We consider again the spanning-tree automaton of Example 5 on page 7. For convenience, we recall its transition maker in Algorithm 1 above. An example run is illustrated in Figure 2. The natural number within a node is the current content of its register \textit{root} (which, at the beginning of the run, equals the identifier of the respective node). The \textit{parent}-relation is represented by thick arrows, where we omit self loops. Moreover, white nodes are in state 1, gray ones in state 2, and black ones in state 3. Note that nodes 2 and 7 toggle between states 1 and 2 before terminating in state 3. In the broadcast phase (last row), node 0 is the first one to enter state 3. The state is then propagated to all other nodes to announce successful termination.

\subsection{Correctness of the spanning-tree automaton}

First, we observe that in every graph $G = (V,E)$, there must be a node that eventually enters state 3. Indeed, it is straightforward to show by induction that for every node $v$ at distance $i$ of node 0, if $v$ has not reached state 3 by time $i - 1$, then for every time $t \geq i$, we have $v.\textit{root} = 0$ and $v.\textit{parent}$ points to some fixed node $v'$ at distance $i - 1$ of node 0 (or distance 0 in case $v = 0$). Note that $v$ will never modify its registers again once $v.\textit{root} = 0$, because the only rule that could potentially modify the registers, i.e., Rule 1, is not applicable. Hence, if no node has reached state 3 after at most diameter($G$) rounds of communication, then the parent pointers represent a valid spanning tree rooted at node 0. (Remember that $G$ is by definition connected.) Since this tree remains forever unchanged, it is easy to verify that node 0 will eventually enter state 3, which causes all other nodes to eventually do the same, and thus the automaton to halt. What remains to be shown is that no other node reaches state 3 before node 0 does.

To this end, let us assume that $u$ is the first node to enter state 3, and that it does so in the $(t + 1)$-th round. Therefore, according to Rule 3,

\[ (u.\textit{state} = 2 \land u.\textit{root} = u). \]
Our goal is to show that $u$ is necessarily node 0.

For $i \in \mathbb{N}$, let $U_i$ be the set of nodes at distance exactly $i$ from $u$. Moreover, for notational convenience, we set $U_{-1} = U_0$ (which is \{u\}). By a similar inductive argument as above, we can easily see that for all $i \in \mathbb{N}$ and $v \in U_{i+1}$,

\[
\text{at every time } t' \in \mathbb{N}: \quad (v.\text{root} = u \implies v.\text{parent} \in U_i). \tag{b}
\]

Based on (a) and (b), we now prove the following property:

**Lemma 11.** For all $i \in [t]$, we have

\[
atime t - i: \quad \forall v \in U_i: \quad (v.\text{state} = 2 \land v.\text{root} = u). \tag{P_i}
\]

**Proof.** We proceed by induction. $P_0$ is obvious, since it coincides with our assumption (a). So suppose that $P_i$ holds. We will show $P_{i+1}$.

Take any node $v \in U_i$. By induction hypothesis, at time $t - i$, we have $v.\text{state} = 2$ and $v.\text{root} = u$. Suppose that $v$ reaches such a local configuration for the first time at time $t' \leq t - i$. Note that $v$ cannot change its local configuration between times $t'$ and $t - i$ (the only way to do so would be to choose another root, but then it could not return to root $u$ anymore). Let $v' \in U_{i+1}$ be adjacent to $v$. According to Rule 2, at time $t' - 1$, we must have $v'.\text{root} = u$. Moreover, $v'$
continues to have \( v'.root = u \) until time \( t - (i + 1) \) (otherwise, \( v \) would have a different root than \( u \) at time \( t - i \)).

Now, take any \( v' \in U_{i+1} \). We just showed that, at time \( t - (i + 1) \), we have \( v'.root = u \). It remains to show that we also have \( v'.state = 2 \) at time \( t - (i + 1) \). Let \( v \) be \( v'.parent \) at time \( t - (i + 1) \). By (b), we know that \( v \in U_i \). Let \( t' \leq t - i \) be as above, i.e., be the first point in time where \( v.state = 2 \) and \( v.root = u \). According to Rule 2, at time \( t' - 1 \), we must have \( v'.root = u \). As \( v'.root = u \) at time \( t - (i + 1) \), too, \( v'.parent \) is the same for all \( t \in [t' - 1 : t - (i + 1)] \). In particular, \( v'.parent = v \) at time \( t' - 1 \). By Rule 2, this implies \( v'.state = 2 \) at time \( t' - 1 \), and therefore also \( v'.state = 2 \) at time \( t - (i + 1) \). This proves \( P_{i+1} \). \( \square \)

To conclude, we argue that Lemma 11 implies \( u = 0 \). It suffices to observe that for every node \( v \), if \( v.root = u \) at any point in time, then \( u \leq v \). Since Lemma 11 tells us that every node \( v \) has \( u \) in its register \( root \) at some point in time, we can deduce that \( u \leq v \) for every node \( v \in V \), which implies \( u = 0 \).

### B  Syntactic sugar for functional fixpoint logic

#### B.1 Encoding sets as functions

To simplify the exposition and some of the proofs, we have defined functional fixpoint logic in such a way that the operator \( \text{pfp} \) can bind only function variables. However, since it is straightforward to encode sets of nodes as functions, we often take the liberty of writing formulas in which \( \text{pfp} \) binds both function and set variables. We now justify this formally by showing how set variables can always be eliminated.

To this end, let us fix an infinite supply \( S \) of set variables. We 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 \( S \). Naturally, the semantics is that \( "t \text{ is an element of } X" \), and we will use \( t \notin X \) as an abbreviation for \( \neg(t \in X) \). The definitions of free variables, variable assignment, interpretation, and satisfaction are generalized to set variables in the obvious way.

Since we consider only graphs that have at least two nodes and are equipped with a total order, any set of nodes \( U \) can be represented by a function that maps nodes in \( U \) to their direct successors and nodes outside of \( U \) to themselves. The reason for choosing the direct successor is simply because it is both easy to specify and well-defined for all nodes in all graphs, assuming we consider the minimum node to be the direct successor of the maximum node. The following first-order formula schema states that the node represented by term \( t \) is the direct successor of the node represented by term \( s \):

\[
\text{succ}[s, t] := (s < t \land \exists z, t < z < s) \lor \forall z (t \leq z \land z \leq s)
\]

(We write, for instance, "\( \text{succ}[x, y] \)" to instantiate this schema with node variables \( x \) and \( y \).)

Now, let us consider a formula of the form \( \text{pfp}[(f_{i} : \varphi_{i})_{i \in (\ell)}].(X : \vartheta)] \psi \), where \( X \in S \), \( f_{1}, \ldots, f_{\ell} \in F \), and \( \varphi_{1}, \ldots, \varphi_{\ell}, \vartheta, \psi \) are formulas. We assume that the
variable \textit{out} does not occur freely in \( \vartheta \). On an intuitive level, since \( X \) is a set variable, the set membership defined by \( \vartheta \) is with respect to a single special variable, namely \textit{in}. A node \( v \) lies in the corresponding set precisely if \( \vartheta \) is satisfied by interpreting \( \textit{in} \) as \( v \). The stages of the partial fixpoint induction are computed as before, the only novelty being that \( X \) is initialized to the empty set (while the function variables \( f_1, \ldots, f_\ell \) are still initialized to the identity function). If the sequence of stages does not converge to a fixpoint, the partial fixpoint is the same as the initial stage (i.e, \( \emptyset \) for set variables and \( \text{id}_V \) for function variables).

We can easily eliminate \( X \) by replacing it with a fresh function variable \( f_X \). More precisely, we rewrite the preceding formula as \( \text{pfp}_{f_1} \ldots f_\ell [ (f_X : \vartheta') ] \psi' \).

To transform \( (\varphi_i)_{i \in [\ell]} \) and \( \psi \) into their new forms \( (\varphi'_i)_{i \in [\ell]} \) and \( \psi' \), it suffices to replace every occurrence of an atomic formula of the form \( t \in X \) with its encoded representation \( f_X(t) \neq t \). That is, for instance, \( \psi' := \psi[t \in X \mapsto f_X(t) \neq t] \). For the subformula \( \vartheta' \), we additionally have to ensure that \( f_X \) is interpreted as a function that maps a node to its direct successor if and only if it is contained in the new interpretation of \( X \). Thus, we define

\[
\vartheta' := \vartheta[t \in X \mapsto f_X(t) \neq t] \land \text{succ[IN, OUT]}.\]

Notice that our encoding scheme preserves the partial fixpoint of the original formula. In particular, if the sequence of stages does not converge, then \( f_X \) is interpreted as \( \text{id}_V \), which represents the empty set.

\textbf{B.2 Quantification over functions}

Since partial fixpoint inductions allow us to iterate over various interpretations of a function variable, they provide a way of expressing (second-order) quantification over functions. To make this more convenient, we add function quantifiers as “syntactic sugar” to the language of functional fixpoint logic and show how they can be converted into \texttt{pfp} operators.

Consider a formula of the form \( \exists f \varphi \), where \( f \) is a function variable and \( \varphi \) is a formula. Obviously, the semantics is that “there exists a function \( f \) such that \( \varphi \) holds”. In the following, we simulate this existential quantification by a partial fixpoint induction that iterates over all possible interpretations of \( f \). If there exists no interpretation satisfying \( \varphi \), then the sequence of stages of our induction cycles forever through all functions, never reaching a fixpoint. Otherwise, it ends up looping on the first function that satisfies \( \varphi \). This function therefore becomes the interpretation assigned to \( f \) by the partial fixpoint operator.

To perform the iteration described above, we need to define a cyclic order on the set of all functions. This is easy because we restrict ourselves to graphs whose nodes are totally ordered. Thus, each function on a given input graph \( G = (V, E) \) can be thought of as a \(|V|\)-digit number written in base \(|V|\), where the \( i \)-th least significant digit represents the value taken by the function at the \( i \)-th smallest node. Based on this, the direct successor of a function is simply the function that corresponds to its number incremented by 1. Additionally, to make the order cyclic, we stipulate that the smallest function is the direct successor of the largest function; in other words, there is an “integer overflow”.

It only remains to implement these ideas in functional fixpoint logic. Without loss of generality, we may assume that \( \varphi \) does not contain any free occurrences of IN and OUT.\(^1\) Hence, we can rewrite \( \exists f \varphi \) as the partial fixpoint formula pfp\([f : \psi]\) \( \varphi \), where

\[
\psi := \bigwedge \left[ \left[ \varphi \lor \exists x, y \left( x < \text{IN} \land f(x) < y \right) \Rightarrow \text{OUT} = f(\text{IN}) \right] \land \left[ \neg \left[ \varphi \lor \exists x, y \left( x < \text{IN} \land f(x) < y \right) \Rightarrow \text{succ}[f(\text{IN}), \text{OUT}] \right] \right] \right].
\]

The subformula \( \psi \) distinguishes between three cases in order to determine how to update the interpretation of \( f \). First, as stated in the first line of the big conjunction, if the current interpretation already satisfies \( \varphi \), then it is maintained. Intuitively, this can be read as: “the new value OUT of \( f \) at IN is equal to its current value \( f(\text{IN}) \)”. Second, even if \( f \) (or rather the number representing it) has to be incremented, the function may retain its current value at some nodes. In fact, “the value of digit IN remains unchanged as long as there is some less significant digit \( x \) that can still be incremented”. Third, as stated in the second line of the big conjunction, if none of the two previous cases apply, then “the value of digit IN must be incremented if possible and otherwise reset”. We express this using the formula schema succ\([s, t]\) from Section B.1 to ensure that “the new value OUT is the direct successor of the current value \( f(\text{IN}) \).”

Now, when we evaluate the formula pfp\([f : \psi]\) \( \varphi \) on a structure \((G, q), \sigma\), there are two possibilities. Either there exists no interpretation of \( f \) that satisfies \( \varphi \), in which case the partial fixpoint induction does not reach a fixpoint. This means that \( f \) defaults to id\(_V\) and pfp\([f : \psi]\) \( \varphi \) evaluates to false. Or \( \varphi \) can be satisfied, in which case the induction reaches a satisfying fixpoint. This fixpoint is chosen as the interpretation of \( f \), and pfp\([f : \psi]\) \( \varphi \) evaluates to true. In both cases, the result is the same as when evaluating \( \exists f \varphi \) on \((G, q), \sigma\).

### C From automata to logic

We now prove the first part of Theorem 1 (stated in Section 1), by showing that functional fixpoint logic is at least as expressive as distributed register automata:

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

The rest of this section is devoted to the proof of Proposition 9. We consider a distributed register automaton \( A = (Q, R, i, \Delta, H, o) \) with transition maker \( \Delta = (Q, \tilde{R}, i, \tilde{\delta}, \tilde{o}) \) and assume that \( A \) decides a graph property \( P \) over label set \( I \). In the remainder of this section, we construct a formula \( \varphi_A \) of functional fixpoint logic that defines \( P \). As our exposition goes into full detail, it is a bit lengthy.

\(^1\) Otherwise, we replace \( \exists f \varphi \) with the equivalent formula \( \exists x, y \left( x = \text{IN} \land y = \text{OUT} \land \exists f \varphi[\text{IN}, \text{OUT} \Rightarrow x, y] \right) \), where \( \varphi \)’s free occurrences of IN and OUT are substituted with fresh node variables \( x \) and \( y \).
To make it clear how all the pieces fit together, we present the construction in a top-down manner.

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 \( q \). Furthermore, 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 will ensure that on any \( I \)-labeled graph \((G, q)\), the interpretations \((X_q)_{q \in Q}\) and \((f_r)_{r \in R}\) represent the halting configuration reached by \( A \) on \((G, q)\).

Hence, the final formula is simply

\[
\varphi_A := \text{pfp} \left[ \left( X_q : \varphi_q \right)_{q \in Q} \right] \forall x \left( \bigvee_{p \in H} \left( o(p) = \text{YES} \right) \wedge \bigvee_{x \in X_p} \right),
\]

which states that all nodes end up in a halting state that outputs \text{YES}.

### C.1 Simulating the automaton

The real work is now to construct the formulas \((\varphi_q)_{q \in Q}\) and \((\varphi_r)_{r \in R}\) 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. Note that in doing so, we make sure that the infinite sequence of stages reaches a fixpoint (given our assumption that the automaton is a decider and thus eventually halts).

For each state \( q \in Q \), we update the set variable \( X_q \) with the formula

\[
\varphi_q := \left( \bigvee_{a \in I} \left( \text{IN} \wedge \bigwedge_{p \in Q} \left( \text{IN} \not\in X_p \right) \right) \right) \vee \varphi^\Delta_q \vee \left( \text{IN} \in X_q \right),
\]

Note that this formula makes use of the syntactic sugar introduced in Subsection B.1, which allows us to update set variables without any explicit reference to the variable \( \text{OUT} \). The first disjunct of \( \varphi_q \) ensures that stage 1 of the partial fixpoint induction corresponds to the initial configuration of the automaton. It does so by stating that a node (represented by the variable \( \text{IN} \)) will belong to \( X_q \) in the next stage if its input label \( a \) maps to \( q \) and it does not belong to any set \( X_p \) in the current stage. The latter part holds only in stage 0, where all set variables are initialized to \( \emptyset \). In the second disjunct, we use the subformula \( \varphi^\Delta_q \) defined below to ensure that if node \( \text{IN} \) is currently active, i.e., in a non-halting state, then \( \text{IN} \) switches to state \( q \) when executing the transition maker \( \Delta \). Finally, in case \( q \) is a halting state, the third disjunct of \( \varphi_q \) states that \( \text{IN} \) remains in \( q \) if it is already there. This formalizes the fact that halting states are never left.

To implement the above-mentioned subformula \( \varphi^\Delta_q \), we use another partial fixpoint induction, which simulates the behavior of the transition maker \( \Delta \) between two consecutive synchronous rounds. This second induction is thus nested within the main induction of \( \varphi_A \). Similarly to before, we introduce a set variable \( Y_q \) for each inner state \( q \) and a function variable \( g_r \) for each inner register \( r \). These variables serve to encode the inner configurations of the transition maker at each node, in the same way as the variables \((X_q)_{q \in Q}\) and \((f_r)_{r \in R}\) encode the
local configurations of the automaton. Furthermore, we introduce a function variable \( g_q \) to represent the transition maker’s reading head at each node and a set variable \( Y_{\text{end}} \) to represent the set of nodes that have finished scanning their neighborhood. We will make sure that once the nested partial fixpoint has been reached, the variables \((Y_q)_{\bar{q}\in Q}\) and \((g_r)_{\bar{r}\in R}\) represent the final inner configurations reached by the nodes. Relying on that, we define

\[
\varphi^\Delta_q := \left( \bigvee_{p\in Q \setminus H} \text{IN} \in X_p \right) \land \text{pfp} \left[ \begin{array}{cc}
\{Y_q : \psi_q\}_{\bar{q}\in \bar{Q}} \\
g_r : \psi_r \\
Y_{\text{end}} : \psi_{\text{end}}
\end{array} \right] \bigvee \text{IN} \in Y_{\bar{q}}.
\]

The first conjunct simply checks that \( \text{IN} \) is currently in a non-halting state, which means that it is allowed to run the transition maker \( \Delta \). The second conjunct expresses that after running the transition maker, \( \text{IN} \) is in some inner state \( \bar{q} \) from which \( \Delta \) outputs state \( q \).

Continuing with our top-down approach, we first complete the discussion of the outer partial fixpoint induction of \( \varphi_A \) before turning to the nested one. For each register \( r \in R \), the formula with which we update \( f_r \) is

\[
\varphi_r := \varphi^\Delta_r \lor \left( \bigvee_{P \in H} \text{IN} \in X_P \land \text{OUT} = f_r(\text{IN}) \right).
\]

Here, the subformula \( \varphi^\Delta_r \) states that \( \text{IN} \) is still in a non-halting state and writes the identifier of \( \text{OUT} \) to its register \( r \) after running the transition maker. The second disjunct of \( \varphi_r \) covers the case where \( \text{IN} \) has already reached a halting state, forcing it to maintain its current valuation of \( r \). Note that neither of the two disjuncts is satisfied if all set variables are interpreted as the empty set, as is the case in stage 0 of the (outer) partial fixpoint induction. Therefore, \( \varphi_r \) does not define a functional relationship in stage 0, which means that the interpretation of \( f_r \) in stage 1 defaults to the identity function. Rather conveniently, this is precisely what we want, i.e., stage 1 represents the initial configuration of the automaton. Here we rely on the assumption that \( A \) has no input registers (because it decides a graph property) and therefore initializes the registers of each node to the node’s own identifier.

The implementation of \( \varphi^\Delta_r \) is very similar to that of \( \varphi^\Delta_q \). In particular, it uses the same nested induction to simulate \( \Delta \):

\[
\varphi^\Delta_r := \left( \bigvee_{p\in Q \setminus H} \text{IN} \in X_p \right) \land \text{pfp} \left[ \begin{array}{cc}
\{Y_q : \psi_q\}_{\bar{q}\in \bar{Q}} \\
g_r : \psi_r \\
Y_{\text{end}} : \psi_{\text{end}}
\end{array} \right] \bigvee \{\text{IN} \in Y_{\bar{q}} \land \text{OUT} = g_r(\text{IN})\}.
\]

The only difference to \( \varphi^\Delta_q \) is the big disjunction within the scope of the pfp operator. It stipulates that \( \text{IN} \) ends up in some inner state \( \bar{q} \) that causes the transition maker to update \( \text{IN} \)’s register \( r \) to the identifier of \( \text{OUT} \). For this to be true, the identifier of \( \text{OUT} \) must be stored in the inner register \( \bar{r} \) that is used to update \( r \).
C.2 Simulating the transition maker

We now come to the nested partial fixpoint induction. As briefly mentioned above, it uses two helper variables $g$ and $Y_{\text{end}}$ to keep track of the transition maker’s scanning process. In each stage of the induction, $g$ is interpreted as a function that maps each node $u$ to the node $v$ that is currently scanned by (the transition maker at) $u$. In other words, this function gives us the current position of each node’s reading head. Since every node starts by reading its own local configuration, it comes in handy that $g$ is initialized to the identity function at stage 0 of the nested induction. The function is then updated with the following formula:

$$\psi := \text{IN} \iff \text{OUT} \land \Bigg[ \begin{array}{l} g_{i}(\text{IN}) = \text{IN} \land \forall x (\text{IN} \iff x \Rightarrow \text{OUT} \leq x) \\ g_{i}(\text{IN}) \neq \text{IN} \land g_{i}(\text{IN}) < \text{OUT} \land \forall x (\text{IN} \iff x \land g_{i}(\text{IN}) \leq x \Rightarrow \text{OUT} \leq x) \\ g_{i}(\text{IN}) \neq \text{IN} \land g_{i}(\text{IN}) = \text{OUT} \land \forall x (\text{IN} \iff x \Rightarrow x \leq \text{OUT}) \end{array} \Bigg]$$

Here, the first conjunct ensures that the reading head of IN can be moved only to a neighbor of IN, while the big disjunction below is responsible for selecting the smallest neighbor that has not yet been visited. The first line of the disjunction covers the initial step, where the reading head is still at IN and must be moved to the smallest of all neighbors. The second line corresponds to the case where the head is moved from one neighbor to the next-smallest one. Finally, the third line states that once the head has reached the greatest neighbor, it remains there. This is important to ensure that the sequence of stages converges to a fixpoint. Also note that $\psi$ always defines a total function because every node has at least one neighbor. (This follows from our restriction to connected graphs with at least two nodes.)

Since the reading head of the transition maker remains at the last-visited node, we must prevent that node from being processed more than once. This is the purpose of the set variable $Y_{\text{end}}$, which will be interpreted as the set of all nodes that have finished scanning their neighborhood. If IN’s reading head reaches the last neighbor in stage $i$, then IN is added to $Y_{\text{end}}$ in stage $i + 1$:

$$\psi_{\text{end}} := g_{i}(\text{IN}) \neq \text{IN} \land \forall x (\text{IN} \iff x \Rightarrow x \leq g_{i}(\text{IN}))$$

Having formalized how the transition maker $\Delta$ scans its input, we now make use of the variables $g_{i}$ and $Y_{\text{end}}$ to define how it updates its inner configuration. For each inner state $\tilde{q} \in \tilde{Q}$, the set variable $Y_{\tilde{q}}$ is updated with the formula

$$\psi_{\tilde{q}} := \left( \text{IN} \in Y_{\text{end}} \land \text{IN} \in Y_{\tilde{q}} \right) \lor \left( \text{IN} \notin Y_{\text{end}} \land \bigvee \varphi_{\text{curr}}(\tilde{p}, \tilde{p}, <) \right).$$

The first disjunct states that $\Delta$ remains in $\tilde{q}$ if it has reached the end of its input and is currently in $\tilde{q}$, whereas the second disjunct describes an inner transition...
to \( \bar{q} \) in case \( \Delta \) has not yet terminated. For such an inner transition to take place, \( \bar{q} \) must be the inner state that is obtained when applying \( \delta \) to the current inner state \( \bar{p} \) of \( \text{IN} \), the current state \( p \) of \( g_1(\text{IN}) \), and the relation \( \prec \) that compares the inner register values of \( \text{IN} \) with the register values of \( g_1(\text{IN}) \). Our formula expresses this as a disjunction over all possible choices of \( \bar{p}, p, \) and \( \prec \) that lead to \( \bar{q} \), using the subformula \( \vartheta^{\text{curr}}(\bar{p}, p, \prec) \) to check whether \( \bar{p}, p, \) and \( \prec \) do indeed correspond to the current inner configuration of \( \text{IN} \) and local configuration of \( g_1(\text{IN}) \).

Implementing \( \vartheta^{\text{curr}}(\bar{p}, p, \prec) \) for any \( \bar{p} \in \bar{Q}, p \in Q, \) and \( \prec \in 2^{(\bar{R} \cup R)^2} \) is straightforward:

\[
\vartheta^{\text{curr}}(\bar{p}, p, \prec) := \vartheta^{\text{curr}} \land g_1(\text{IN}) \in X_p \land \\
\bigwedge_{\bar{s} \in \bar{R}, s \in R, \bar{s} \prec s} g_2(\text{IN}) < f_\bar{s}(g_1(\text{IN})) \land \bigwedge_{\bar{s} \in \bar{R}, s \in R, \bar{s} \prec s} f_\bar{s}(g_1(\text{IN})) < g_2(\text{IN}) \land \\
\bigwedge_{\bar{s}, \bar{s}' \in \bar{R}, s, s' \prec \bar{s}, s'} g_2(\text{IN}) < g_{s'}(\text{IN}) \land \bigwedge_{s, s' \in R, s \prec s'} f_s(g_1(\text{IN})) < f_{s'}(g_1(\text{IN}))
\]

In the first line, we state that \( \bar{p} \) is the inner state of \( \text{IN} \) and \( p \) is the state of \( g_1(\text{IN}) \). For the former, we use the little helper formula \( \vartheta^{\text{curr}} \) defined below. In the remaining two lines, we check that all inequalities specified by \( \prec \) are satisfied.

The reason for using the helper formula \( \vartheta^{\text{curr}} \) is that the inner state of \( \text{IN} \) is not represented explicitly in stage 0 of the nested induction. Therefore, if \( \text{IN} \) does not belong to any set \( Y_{\bar{q}} \), we assume that it is in the inner initial state \( \bar{i} \):

\[
\vartheta^{\text{curr}}:= \begin{cases} 
\text{IN} \in Y_{\bar{p}} \lor \bigwedge_{\bar{q} \in \bar{Q}} \text{IN} \notin Y_{\bar{q}} & \text{if } \bar{p} \neq \bar{i}, \\
\text{IN} \in Y_{\bar{p}} & \text{otherwise.}
\end{cases}
\]

Note that since we represent each inner register \( r \) by a function variable \( g_r \), our encoding does not take into account that inner registers can hold the undefined value \( \perp \). In particular, in stage 0 of the nested induction, \( g_r \) is interpreted as the identity function, whereas the transition maker initializes \( r \) to \( \perp \). However, we may assume without loss of generality that when the transition maker starts in its initial inner configuration \((\bar{i}, \{ \bar{r} \mapsto \perp \}_{\bar{r} \in \bar{R}})\) and reads the first local configuration \((p, \rho)\) of its input sequence, then it updates each of its inner registers to some value provided by \( \rho \) (and thus distinct from \( \perp \)) that depends only on \( p \) and the order relation between the register values of \( p \). More formally, we assume that for every state \( p \in Q \) and all binary relations \( \prec, \prec' \subseteq (\bar{R} \cup R)^2 \) such that \( r \prec s \) if and only if \( r \prec' s \) for all \( r, s \in R \), we are guaranteed that \( \delta(\bar{i}, p, \prec) = \delta(\bar{i}, p, \prec') = (\bar{q}, \bar{\alpha}) \) such that \( \bar{\alpha}(\bar{r}) \in \bar{R} \) for all \( \bar{r} \in \bar{R} \). On that basis, we may substitute the initial inner configuration at node \( v \) with the indistinguishable inner configuration \((\bar{i}, \{ \bar{r} \mapsto v \}_{\bar{r} \in \bar{R}})\), which is precisely what we do in stage 0.
To complete our construction, it only remains to specify how the inner registers are updated. For each \( \tilde{r} \in \tilde{R} \), we define the formula

\[
\psi_{\tilde{r}} := \left( \text{IN} \in Y_{\text{end}} \land \text{OUT} = g_{\tilde{r}}(\text{IN}) \right) \lor
\left[ \text{IN} \not\in Y_{\text{end}} \land \left( \bigvee_{p \in \tilde{Q}, p \in Q, \prec \in (\tilde{R} \cup R)^2, s \in R} g_{\tilde{r}}(p, p, \prec) = g_{\tilde{s}}(\text{IN}) \right) \lor \left( \bigvee_{p \in \tilde{Q}, p \in Q, \prec \in (\tilde{R} \cup R)^2, s \in R} g_{\tilde{s}}(\text{IN}) = f_s(g_{\tilde{r}}(\text{IN})) \right) \right],
\]

whose basic structure is very similar to that of \( \psi_{\tilde{q}} \). The first line just states that the transition maker \( \Delta \) retains the current value of \( \tilde{r} \) if it has reached the end of its input. The second line covers the case where \( \Delta \) has to update \( \text{IN} \)'s inner register \( \tilde{r} \) to a new value \( \text{OUT} \), based on what it sees from the current inner configuration of \( \text{IN} \) and the current local configuration of \( g_{\tilde{r}}(\text{IN}) \). When \( \Delta \) evaluates its inner transition function \( \tilde{\delta} \) on the currently seen inner state \( \tilde{p} \), state \( p \), and relation \( \prec \), there are two possibilities: either the new value of \( \tilde{r} \) is obtained from some inner register \( \tilde{s} \) of \( \text{IN} \), or it is obtained from some register \( s \) of \( g_{\tilde{r}}(\text{IN}) \). These two possibilities are expressed by the two big disjunctions in the second line of \( \psi_{\tilde{r}} \).

## D From logic to automata

We will now take the opposite direction and go from formulas to automata. Hence, this section is devoted to the proof of the following result:

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

We proceed by induction, i.e., we construct automata for subformulas, which will then be put together to produce automata for composed formulas. The invocation of automata as subroutines will be more convenient if they are centralized. Intuitively, this means that a dedicated root initiates an execution and, at the end, collects an acknowledgment from all the other processes before terminating. Consider the set of registers \( R_{\text{tree}} = \{ \text{parent}, \text{root} \} \) and suppose \( C \in C(Q, R) \) is a configuration such that \( R_{\text{tree}} \subseteq R \). We call \( C \) a spanning-tree configuration if the valuations of parent and root form a spanning tree in \( C \) as computed by the algorithm from Example 5.

**Definition 12 (Centralized automaton).** A centralized automaton is a distributed register automaton \( A \) with input registers \( R^{\text{in}} \) and output registers \( R^{\text{out}} \) such that \( R_{\text{tree}} \subseteq (R^{\text{in}} \setminus R^{\text{out}}) \) and all runs of \( A \) starting in a spanning-tree input configuration satisfy the following properties:

1. A non-root node can only leave its initial local configuration after its parent changed its local state for the first time.
2. The run eventually halts and the root is the last node to terminate (i.e., no node enters a halting state after the root does).

3. The registers in \((R^{in} \setminus R^{out})\) are not modified during the run.

Essentially, the notion of centralized automaton provides a way to combine and reason about distributed register automata in a fully sequential way. Given a centralized automaton \(A_1 = (Q_1, R_1, \iota_1, \Delta_1, H_1, o_1)\) with input registers \(R_1^{in}\) and output registers \(R_1^{out}\), we can construct another centralized automaton \(A_2 = (Q_2, R_2, \iota_2, \Delta_2, H_2, o_2)\) that uses \(A_1\) as a subroutine. To this end, we make sure that \(Q_2\) is a Cartesian product of the form \(S \times \{\text{main}, \text{call}\} \times Q_1\) and that \(R_2\) includes \(R_1\). In any execution of \(A_2\), every node is initially in the main mode (by which we mean that the relevant component of its state is main). While in this mode, a node behaves according to the algorithm implemented by \(A_2\). The algorithm must be such that at some point in time, all nodes except the root of the spanning tree are idle, and the current configuration of \(A_2\) yields a desired initial configuration of \(A_1\) if we project all states to their \(Q_1\)-component and consider only those registers that belong to \(R_1\). Then, the root can launch a nested execution of \(A_1\) by entering the call mode. While in this mode, a node must simulate automaton \(A_1\), using only the \(Q_1\)-component of its state and its \(R_1\)-registers. Whenever a node in the main mode sees that one of its neighbors has entered the call mode, it does the same. Since \(A_1\) is centralized, by Definition 12 (1), this approach yields a faithful simulation of \(A_1\), despite the fact that the nodes do not enter the call mode simultaneously. Moreover, by Definition 12 (2), once the root has reached a halting state of \(A_1\), it knows that the automaton has halted everywhere. The root can thus switch back to the main mode and resume executing \(A_2\). The other nodes do the same as soon as one of their neighbors has done so. As a result, automaton \(A_2\) can use the output configuration produced by \(A_1\). Since by Definition 12 (3), \(A_1\) does not modify the registers in \(R_1^{in} \setminus R_1^{out}\), we can rely on them being the same as before calling \(A_1\). In particular, the spanning tree represented by the registers in \(R^{tree}\) remains unchanged.

Note that while the preceding approach may lead to very inefficient algorithms from the perspective of distributed computing, it is sufficient for the purposes of this paper, since we only want to characterize the fundamental expressive power of distributed register automata.

As our proof of Proposition 10 will proceed by induction on the structure of formulas, we will have to cope with the interpretations of free node and function variables. To this end, we are going to encode interpretations into configurations. Consider a spanning-tree configuration \(C = ((V, E), q, r) \in C(Q, R)\), with root \(v_0\), such that the set \(R\) contains variables from \(\mathcal{N} \cup \mathcal{F}\). Then, \(C\) defines a variable assignment \(\sigma\) as follows: For a node variable \(x \in R\), we let \(x^\sigma = r(v_0)(x)\). That is, the interpretation of variable \(x\) is the value of register \(x\) at the root. Moreover, each register \(f \in R \cap \mathcal{F}\) encodes an interpretation \(f^\sigma\) of the function variable \(f\) by letting \(f^\sigma(v) = r(v)(f)\) for all \(v \in V\).
The next preparatory proposition states that there is a centralized automaton that evaluates a term with respect to the variable assignment encoded in the input configuration.

**Proposition 13 (Automata for terms).** Let $t$ be a term. There is a centralized automaton $A_t$ computing a transduction

$$T_A_t : \mathcal{C}(1, R_{\text{tree}} \cup \text{free}(t)) \to \mathcal{C}(1, \{t\})$$

such that, given a spanning-tree input configuration $C = (G, q, r)$ with $G = (V, E)$ and root $v_0 \in V$ encoding a variable assignment $\sigma$, the automaton eventually outputs a configuration $C' = (G, q, r')$ satisfying $r'(v_0)(t) = t^\sigma$.

**Proof.** Let $t = f_\ell(\ldots f_2(f_1(x)) \ldots)$. Note that free$(t) = \{f_1, \ldots, f_\ell, x\}$. The case $\ell = 0$ is trivial, so suppose $\ell \geq 1$. For $i \in [1: \ell]$, let $t_i = f_i(\ldots f_1(x) \ldots)$.

Given a spanning-tree input configuration $C = ((V, E), q, r)$, the automaton $A_t$ will use auxiliary registers $t_i$ to compute and store the intermediary values $t_{\sigma i} = f_{\sigma i}(\ldots f_{\sigma 1}(x^\sigma) \ldots)$. The register $t = t_\ell$ is the only output register. We compute the values for $t_1, \ldots, t_\ell$ successively. As an invariant, we maintain that, after computing $t_{\sigma i}$, the root stores $t_{\sigma i}$ in its register $t_i$.

We reserve an auxiliary register self for storing the own identifier of a node. To compute $t_{\sigma i+1}$, we proceed as follows. Starting from the root, the value of $t_\sigma$ is propagated to all nodes along the spanning tree down to the leaves. The leaves, in turn, initiate a back-propagation phase, where an internal node waits for signals from all its children, notifies its parent, and then terminates. However, during this back-propagation, the node whose identifier coincides with $t_i$ includes the contents of its register $f_{i+1}$ in its message to its parent node, which propagates it further, until the value reaches the root. The latter stores it in its register $t_{i+1}$. \[\square\]

As further preparation for the inductive translation of formulas, we describe a subroutine that allows the root to increment a register by one.

**Proposition 14 (Register incrementation).** Let $r$ be a register. There is a centralized automaton $A_{r++}$ computing a transduction

$$T_{A_{r++}} : \mathcal{C}(1, R_{\text{tree}} \cup \{r\}) \to \mathcal{C}(1, \{r\})$$

such that, given a spanning-tree input configuration $C = (G, q, r)$ with $G = (V, E)$ and root $v_0 \in V$, the automaton eventually outputs a configuration $C' = (G, q, r')$ satisfying $r'(v_0)(r) = \min\{r(v_0)(r) + 1, |V| - 1\}$.

**Proof.** First, the current value $w$ of $v_0$’s register $r$ is broadcast through the entire graph. Then, every node $u$ sends to its parent in the spanning tree the smallest value greater than $w$ among the node identifiers in the subtree rooted at $u$ (provided that this subtree contains such an identifier). This procedure starts at the leaves and works its way up to the root. There, the smallest value obtained must be $w + 1$. \[\square\]

We are now ready to transform any formula into an equivalent automaton.
Proposition 15. Let $\varphi$ be a formula of functional fixpoint logic over some finite set of labels $I$. There is a centralized automaton $A_\varphi$ computing a transduction

$$T_{A_\varphi} : \mathbb{C}(I, R_{\text{tree}} \cup \text{free}(\varphi)) \rightarrow \mathbb{C}([\text{YES}, \text{NO}], \emptyset)$$

such that, given a spanning-tree input configuration $C = (G, q, r)$ with $G = (V, E)$ and root $v_0$ encoding a variable assignment $\sigma$, the automaton eventually outputs a configuration $C' = (G, q', r')$ satisfying $q'(v_0) = \text{YES}$ if and only if $(G, q), \sigma \models \varphi$.

Proof. We proceed by induction on the structure of formulas.

Case $\varphi = (a) t$. Given a spanning-tree input configuration $C = ((V, E), q, r)$ encoding a variable assignment $\sigma$, the automaton $A_\varphi$ first applies, as a subroutine, $A_t$ from Proposition 13. The latter eventually outputs a configuration in $\mathbb{C}(I, \{t\})$ such that the value of register $t$ at the root is $t^\sigma$. Similarly to the construction of Proposition 13, the root then broadcasts $t^\sigma$ to all other nodes along the spanning tree, down to the leaves. During a subsequent back-propagation phase, node $t^\sigma$ checks whether its label is equal to $a$. Accordingly, it sends either YES or NO to its parent, which propagates it further until it reaches the root.

Case $\varphi = s < t$. As a subroutine, the root first launches $A_s$ and then $A_t$ so that it eventually stores, in (auxiliary) registers $s$ and $t$, the interpretation $s^\sigma$ and $t^\sigma$, respectively. The root then compares both values and outputs the corresponding truth value.

Case $\varphi = s \rightarrow t$. Like in the previous case, the root first launches $A_s$ and then $A_t$ so that it eventually stores $s^\sigma$ and $t^\sigma$ in registers $s$ and $t$, respectively. Both are then propagated along the spanning tree. During a subsequent back-propagation, node $s^\sigma$ checks whether $t^\sigma$ is in its neighborhood. The corresponding truth value is forwarded back to the root.

Case $\varphi = \neg \psi$. As a subroutine, $A_{\neg \psi}$ first applies $A_\psi$ to the given spanning-tree input configuration. Upon termination, the root will just flip the node label from YES to NO or vice versa.

Case $\varphi = \varphi_1 \lor \varphi_2$. The automaton $A_\varphi$ first applies $A_{\varphi_1}$ as a subroutine. If the root outputs YES, then it stops. Otherwise, $A_{\varphi_2}$ is launched.

Case $\varphi = \exists x \psi$. Automaton $A_\varphi$ has an auxiliary register $x$. It makes use of the subroutine $A_{x^+}$ from Proposition 14, which increments the value of $x$ by one every time it is called. The root, initially storing its own identifier 0 in $x$, starts by launching $A_\psi$. If the latter outputs YES (at the root), then the root outputs YES and stops. Otherwise, using $A_{x^+}$, the root will increment its register $x$ by one, launch $A_\psi$ again, and so on. If no increment is possible anymore, $A_\varphi$ outputs NO.
Identifiers in Registers – Describing Network Algorithms with Logic

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

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

Algorithm 3  update($f_i^{\text{new}}$), as controlled by the root

1. $\text{for } \text{in } \in V \text{ do}$
2.   $\text{out}^{\prime} \leftarrow \text{in}; \text{found } \leftarrow \text{false}$
3. $\text{for } \text{out } \in V \text{ do}$
4.   if execute($A_{\varphi_i}$) returns YES /* $\varphi_i[\text{in, out}]$ satisfied */
5.     then if $\text{found } = \text{false}$
6.       then $\text{out}^{\prime} \leftarrow \text{out}; \text{found } \leftarrow \text{true}$
7.     else $\text{out}^{\prime} \leftarrow \text{in}; \text{goto } 8^{\prime}$
8. $\text{in } \text{do } f_i^{\text{new}} \leftarrow \text{root.out}^{\prime}$

Case $\varphi = \text{pfp}[f_i: \varphi_i]_{i \in [1:\ell]} \psi$. First of all, recall that $\text{free}(\varphi)$ is the union of all sets $\text{free}(\varphi_i) \setminus \{f_1, \ldots, f_i, \text{IN, OUT}\}$ and $\text{free}(\psi) \setminus \{f_1, \ldots, f_i\}$. We are interested in a transduction $T_{A_{\varphi}} : \mathbb{C}(I, R_{\text{free}} \cup \text{free}(\varphi)) \rightarrow \mathbb{C}(\{\text{YES}, \text{NO}\}, \emptyset)$. To implement the partial fixpoint computation, we introduce auxiliary registers $f_1, \ldots, f_i$ (for the current interpretation) and $f_1^{\text{new}}, \ldots, f_i^{\text{new}}$ (for the next interpretation). We also use auxiliary registers IN and OUT, as well as a register self for storing the own identifier of each node. Furthermore, to test whether the partial fixpoint computation has reached a cycle with a period greater than one, we include an additional set of auxiliary registers that will allow us to implement a global counter. Since there are precisely $n^{\ell n}$ different $\ell$-tuples of functions on an input graph with $n$ nodes, it is sufficient to count from 0 to $n^{\ell n} - 1$. The counter is taken for granted for now, but it will be explained below, and we shall define a helper automaton $A_{\text{inc}}$ that allows us to increment it by one.

In the following, in the interest of clarity, we abstract away from many implementation details and only describe our construction informally. Recall that $A_{\varphi}$ is a centralized automaton, which means that it is controlled by the root. The root’s program is given by Algorithm 2, presented as pseudo code. First, the counter is initialized to zero in Line 1 (see below for an explanation). Then, in every iteration of the loop starting at Line 2, all registers $f_i$ and $f_i^{\text{new}}$ 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 $f_i^{\text{new}}$ to $f_i$ (Line 3). To update $f_i^{\text{new}}$, Line 4 calls the subroutine provided by Algorithm 3 (which will be explained in the
next paragraph). As a result of this algorithm, we have $f_{i}^{\text{new}} = F_{\bar{\varphi}}((f_{i})_{i \in (\ell)})$ for all $i$, where $F_{\bar{\varphi}}: (V^V)^{\ell} \rightarrow V^V$ is the operator defined in Section 4. Line 5 checks whether we have reached a fixpoint: The root asks every node to compare, for all $i$, its registers $f_{i}^{\text{new}}$ 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 $A_{\psi}$ to evaluate $\psi$ (Line 8). Otherwise, we try to increment the global counter by executing $A_{\text{inc}}$ (Line 6). If the latter returns YES, i.e., incrementation was possible, then another iteration takes place. However, if $A_{\text{inc}}$ returns NO, then the counter has reached its maximum. This implies that we have not reached (and will not reach) a fixpoint so that, according to the partial fixpoint semantics, each node writes its own identifier to every register $f_{i}$ (Line 7) before the automaton evaluates $\psi$ (Line 8).

Let us now describe the subprocedure update($f_{i}^{\text{new}}$) given by Algorithm 3. Using $A_{\text{inc}}$ and $A_{\text{out}}$ provided by Proposition 14, the root will gradually increment its register IN and, in a nested fashion, OUT so as to let IN and OUT range over $V$ (lines 1’ and 3’). After each increment, it will launch $A_{\varphi_{i}}$ to evaluate $\varphi_{i}$ with the current interpretations of IN and OUT (line 4’). If the result is YES, then the root transfers the contents of OUT to OUT’ (Line 6’). Moreover, it sets the flag found to true, which allows it to check whether the node henceforth stored in OUT’ is the only one to make $\varphi_{i}$ true for the given IN. If it is not (i.e., the test in Line 5’ eventually fails), we set OUT’ = IN. Finally, the node whose identifier corresponds to IN sets its register $f_{i}^{\text{new}}$ to the computed value (Line 8’). To implement Line 8’, the root will send OUT’, along the spanning tree, to node IN, which stores it in its register $f_{i}^{\text{new}}$.

A distributed counter. We now sketch how to implement $A_{\text{inc}}$. On an underlying graph $G = (V,E)$ of size $|V| = n$, this automaton can be used to count in a distributed manner from 0 to $n^{\ell n} - 1$. The basic idea is somewhat similar to the construction presented in Section B.2, as we will identify a register valuation function with a number written in base $n$. More precisely, each of the $n$ nodes will have $\ell$ registers $r_{0}, \ldots, r_{\ell - 1}$, each storing a number between 0 and $n - 1$. Thus, a register valuation function $\tau: V \rightarrow V^{\{r_{0}, \ldots, r_{\ell - 1}\}}$ can be seen as an $\ell n$-digit number written in base $n$, where the $i$-th least significant digit is stored in register $r_{[i \mod \ell]}$ of the $[i/\ell]$-th node, with respect to some total order $\sqsubseteq$ on $V$.

In the following, given a spanning tree of $G$, we will choose $\sqsubseteq$ to be the order in which the nodes are visited in the post-order traversal of the tree (where the children of each node are visited in ascending identifier order). This way, the $\ell$ most significant digits are stored in the root.

Formally, $A_{\text{inc}}$ computes a transduction

$$T_{A_{\text{inc}}}: \mathbb{C}(1, R_{\text{tree}} \cup \{\text{max}, r_{0}, \ldots, r_{\ell - 1}\}) \rightarrow \mathbb{C}(\{\text{YES, NO}\}, \{r_{0}, \ldots, r_{\ell - 1}\}).$$

It expects as input a spanning-tree configuration $C = (G, q, \tau)$ that encodes some number $m \in \{0 : n^{\ell n} - 1\}$ in the registers $r_{0}, \ldots, r_{\ell - 1}$ (as described above). In addition, the largest identifier of $G$ (i.e., $n - 1$) must be stored in each node’s register max. If $m < n^{\ell n} - 1$, then the output configuration $C'$ produced by $A_{\text{inc}}$...
is such that the new values of \( r_0, \ldots, r_{\ell-1} \) represent the number \( m + 1 \) and the root outputs \( \text{YES} \), indicating that the incrementation was successful. Otherwise, all registers are set to zero and the root outputs \( \text{NO} \).

In order to generate the first valid input configuration for \( A_{\text{inc}} \), the command \text{init}(A_{\text{inc}}) \) in Algorithm 2 (Line 1) sets all registers \( r_i \) to 0 (thereby initializing the counter to zero) and all registers \( \text{max} \) to \( n - 1 \). While 0 is already known by each node (since this is the root’s identifier), \( n - 1 \) can be determined by a simple subroutine that is very similar to the automaton described in Proposition 14: every node \( v \) sends to its parent in the spanning tree the greatest value among the node identifiers in the subtree rooted at \( v \), which ensures that the largest value received by the root is \( n - 1 \).

After that, every execution of \( A_{\text{inc}} \) performs incrementation by one as follows. First, the root sends the command “increment” in the direction of the leaf \( u \) that stores the least significant digits. More precisely, it sends the command to its smallest child (i.e., the one with the smallest identifier), and every node that receives the command forwards it to its own smallest child.

Upon receiving the command “increment”, leaf \( u \) checks whether at least one of its registers \( r_0, \ldots, r_{\ell-1} \) contains a value smaller than \( \text{max} \). If not, it sets all of them to zero by copying the value 0 from register \( \text{root} \). Then, \( u \) sends the command “increment” back to its parent, which will forward the “carry digit” to the next node in the order \( \sqsubseteq \). On the other hand, if there exists a smallest index \( i \) such that \( r_i \) contains a value smaller than \( \text{max} \), then \( u \) sets to zero only the registers \( r_0, \ldots, r_{i-1} \) and increments \( r_i \) by executing a subroutine that we will describe below. As soon as that subroutine has terminated, \( u \) sends an acknowledgment to the root, which then instructs all other nodes to terminate before switching itself to the halting state \( \text{YES} \).

More generally, whenever a node \( v \) receives the command “increment”, \( v \) tries to forward it to the smallest child that has not yet received it. (In particular, a command received from one child will be forwarded to the next smallest child.) If this is not possible, either because \( v \) is a leaf or because all of its children have already sent back the command, then \( v \) performs an incrementation itself. To do so, it proceeds in exactly the same way as described above for node \( u \).

Since the root stores the most significant digit in its register \( r_{\ell-1} \), it is able to detect the integer overflow that occurs if the input value \( m \) is equal to \( n^{\ell n} - 1 \). In that case, instead of sending “increment” to its parent, it first instructs all other nodes to terminate and then switches to the halting state \( \text{NO} \).

It only remains to explain the subroutine that allows a node \( v \) to increment its register \( r_i \) by one. To do so, \( v \) will send a request to the root including the value of register \( r_i \). The root stores the latter in some auxiliary register \( s \), launches \( A_{s++} \) from Proposition 14, and then sends the result back to \( v \), which writes it into \( r_i \).

This completes the induction and thereby the proof of Proposition 15. \( \square \)

We now have the main building block to prove the result of this section:

**Proof (Proof of Proposition 10).** Let \( \varphi \) be a formula of functional fixpoint logic without free variables. The automaton deciding the graph property defined by \( \varphi \)
proceeds as follows: It first constructs a spanning tree on the given input graph by executing a variant of the automaton from Example 5 (where the root must be the last node to enter a halting state). Then, it launches $A_\varphi$ from Proposition 15, with set of input registers $R_{\text{tree}}$. Finally, the root informs the other nodes, along the spanning tree, about the outcome, i.e., YES or NO. \qed