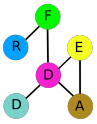


Project FREDDA

Deliverable 1

Report on Advances on Formalization

Summary

Various research collaborated works have been initiated during the first 18 months of the project FREDDA regarding the Task 1 Formalization. The main ideas of these works are presented in the following sections and corresponding published articles are given in Appendix.

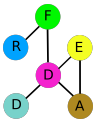
1 Modeling distributed algorithms with identifiers and shared memory

Participants : LSV and IRIF

Many interesting distributed algorithms, including those that solve consensus problems, require unique identifiers to distinguish processes from one another. Consensus is one of the most important problems in fault-tolerant computing.

In [1], we designed a model of distributed algorithms in which each process is assigned a unique identifier. Our model is round-based and uses synchronous communication. However, it should be noted that, under certain circumstances, it can be executed reliably in asynchronous environments. In each round, a process can exchange messages with its direct neighbors in the underlying network. More precisely, it can "see" the current configurations of the neighboring processes including its own, compare the identifiers contained therein, and store them in local registers. Though, apart from these comparisons, no calculations can be performed involving identifiers, our model captures many natural distributed algorithms such as leader election and the computation of a spanning tree. Our paper also deals with the "Subtask 1.2 Logical specification": We design a specification language that allows us to easily specify useful properties, such as the existence of a 3-coloring or of a Hamiltonian cycle in the underlying network. The logic includes comparisons of identifiers and is, therefore, more powerful than many known graph logics like MSO, which are mostly based on a finite alphabet. Our main result is the equivalence of our model of distributed algorithms and our specification language. In particular, we can convert any logical specification into an equivalent algorithm, which can then be considered as correct per design. At the heart of this translation is a sort of consensus algorithm that constructs a spanning tree on the distributed network whose root is the process with the smallest identifier. Although this work involved only experts in the field of formal methods, we have benefited a lot from discussions with the project member Matthias Függer, who is an expert in distributed algorithms and consensus protocols.

In another ongoing work, we consider a model that is capable of capturing renaming protocols, a particular class of consensus protocols. The aim of renaming algorithms is to convert a possibly very large namespace into a smaller



interval. Our model, called distributed memory automata, is inspired by existing algorithms. It distinguishes between local registers and a global memory that can be asynchronously overwritten with local register values. Each process can take a snapshot of the global memory and compare the values it contains with its own local value. We now have a good understanding of this model and obtained first results regarding the complexity of control-state reachability. Next, we plan to study other fundamental problems such as termination and correctness (a process gets a unique value and all values are in a certain interval).

2 Formal model for composing simple distributed algorithms

Participants : LaBRI and IRIF

Differently from consensus which aims at bringing processes to select a common value, the goal in symmetry-breaking tasks is for processes to pick values in a predetermined range, perhaps depending on the number of participants, in such a way that not all processes select the same value. Renaming, which asks each process to pick a distinct integer in some bounded range, is a proeminent representant of this class of tasks.

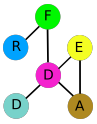
In this on-going work, we are interested in automatically ascertaining the correctness (liveness and safety) of shared-memory protocols for symmetry-breaking tasks. This is challenging because protocols are designed for an arbitrary number of participants and although processes have no input, they are endowed with a unique id which is taken from an unbounded domain. We address these difficulties by focusing on some classes of protocols, in which each register is accessed according to some specific patterns. Those restrictions still leave the room for interesting protocols. Indeed, many classical renaming protocols found in the literature fit into this framework. We are then able to show whether or not a given protocol satisfies a given symmetry-breaking task is decidable by reduction to reachability in Petri nets. We are currently investigating the complexity of the verification procedure. A master student (A. Rappetti) is taking part to these investigations. Future work includes studying robustness of symmetry breaking protocol. That is, trying to assert how changes in the communication model (for example, a fraction of the registers are malfunctionning) or the code (a fraction of the processes deviate from their program) impact on the symmetry-breaking task solved by the protocol.

3 Analysis of the Heard Of Model

Participants : LaBRI

The Heard-Of model [2] abstracts away many details of message passing systems. Computations in this model occur in round. In a round, each process broadcast a message according to its protocol, and receives a subset of the messages broadcast in the same round. Messages that are not received are lost. A communication predicate specifies which subset of messages may be received.

The consensus problem is a fundamental problem in distributed computing. It is a heart of many fault-tolerant distributed services. In this problem, every process has an initial proposal and the goal is for the processes to agree on one



of the proposals. Many algorithms for various execution models can be found of the literature, tolerating various execution modes (including several failures models and various synchrony assumptions). Recent algorithms tend to be very involved, as they try to reach a common decision as fast as permitted by the underlying network conditions. Due to their practical relevance, it is important it ascertain their correctness.

The part of the project took a step direction, by considering a subset of the consensus protocols for the HO model. The restrictions impose on the protocols are light and somewhat natural, in the sense that is captures the vast majority of existing consensus algorithm. Preliminaries results so far include undecidability results, and for consensus a table of all possible programs that solve this tasks (Verifying if a given HO protocol solve consensus is thus just a syntactic check.). An article is in preparation to be submitted this year.

4 Synthesis with respect to execution contexts

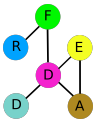
Participants : IRIF

Even though most distributed algorithms for problems like leader election, consensus, set agreement, or renaming, are not very long, their behavior is difficult to understand due to the numerous possible interleavings and their correctness proofs are extremely intricate. Furthermore these proofs strongly depend on the specific assumptions made on the *execution context* which specifies the way the different processes are scheduled and when it is required for a process to terminate. In the case of distributed algorithms with shared registers, interesting execution contexts are for instance the *wait-free* model which requires that each process terminates after a finite number of its own steps, no matter what the other processes are doing or the *obstruction-free* model where every process that eventually executes in isolation has to terminate. It is not an easy task to describe formally such execution context and the difference between contexts can be crucial when searching for a corresponding distributed algorithm. As a matter of fact, there is no wait-free distributed algorithm to solve consensus, even with only two processes, but there exist algorithms in the obstruction-free case.

In [3], we first define a simple model to describe distributed algorithms for a finite number of processes communicating thanks to shared registers. We then show that the correctness of these algorithms can be specified by a formula of the linear time temporal logic LTL and more interestingly we show that classical execution contexts can also be specified in LTL. We then provide a way to synthesize automatically distributed algorithms from a specification. Following SAT-based model-checking approach, we have furthermore implemented our method in a prototype which relies on the SMT-solver Z3 and for some specific cases synthesizes non-trivial algorithms. Of course the complexity is high and we can at present only generate algorithms for two processes but they are interesting by themselves and meet their specification w.r.t. several execution contexts.

References

[1] Benedikt Bollig, Patricia Bouyer, and Fabian Reiter. Identifiers in registers - describing network algorithms with logic. In *FOSSACS'19*, volume 11425



of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2019.

[2] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[3] Carole Delporte-Gallet, Hugues Fauconnier, Yan Jurski, François Laroussinie, and Arnaud Sangnier. Towards synthesis of distributed algorithms with smt solvers. In *NETYS'19*, Lecture Notes in Computer Science, 2019. To appear.

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 PSPACE in terms of partial fixpoint logic (a proper superclass of the logic we consider here).

1 Introduction

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

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

Of particular interest to us is a result due to Abiteboul, Vianu [1], and Vardi [18], which states that on structures equipped with a total order relation, the properties decidable in PSPACE coincide with those definable in *partial fixpoint logic*. The latter is an extension of first-order logic with an operator that allows us to inductively define new relations of arbitrary arity. Basically, this means that new relations can occur as free (second-order) variables in the logical formulas that define them. Those variables are initially interpreted as empty relations and then iteratively updated, using the defining formulas as update 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 synchronously execute the same finite-state machine and communicate with each other by broadcasting messages to their neighbors. Motivated by this result, several variants of distributed automata were investigated by Kuusisto and Reiter in [14], [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 immediately yield a formal model for distributed algorithms in arbitrary networks. In particular, it is not clear what would be the canonical way for nodes to communicate with a non-constant number of peers, if we require that they all follow the same, finitely representable set of rules.

The model we propose here, dubbed *distributed register automata*, is an attempt at a solution. As in [2,3], nodes proceed in synchronous rounds and have a fixed number of registers, which they can compare and update without having

access to numeric values. The new key ingredient that allows us to formalize communication between nodes of unbounded degree is a local computing device we call *transition maker*. This is a special kind of register machine that the nodes can use to scan the states and register values of their entire neighborhood in a sequential manner. In every round, each node runs the transition maker to update its own local configuration (i.e., its state and register valuation) based on a snapshot of the local configurations of its neighbors in the previous round. A way of interpreting this is that the nodes communicate by broadcasting their local configurations as messages to their neighbors. Although the resulting model of computation is by no means universal, it allows formalizing algorithms for a wide range of problems, such as constructing a spanning tree (see Example 5) or testing whether a graph is Hamiltonian (see Example 6).

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

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

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

After giving some preliminary definitions in 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 $\mathbb{N} = \{0, 1, 2, \dots\}$, and the set of integers by $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$. The cardinality of any set S is written as $|S|$ and the power set as 2^S.

In analogy to the commonly used notation for real intervals, we define the notation $[m : n] := \{i \in \mathbb{Z} \mid m \leq i \leq n\}$ for any $m, n \in \mathbb{Z}$ such that $m \leq n$.

To indicate that an endpoint is excluded, we replace the corresponding square bracket with a parenthesis, e.g., $(m:n] \coloneqq [m:n] \setminus \{m\}$. Furthermore, if we omit the first endpoint, it defaults to 0. This gives us shorthand notations such as $[n] \coloneqq [0:n]$ and $[n) \coloneqq [0:n) = [0:n-1]$.

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

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

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

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

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

3 Distributed register automata

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

Let us now formalize the notion of transduction. For any two signatures $\Sigma^{in} = (I, R^{in})$ and $\Sigma^{out} = (O, R^{out})$, a Σ^{in}-Σ^{out}-*transduction* is a *partial*

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

Next, we introduce our formal model of distributed algorithms.

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

Automaton A computes a transduction $T_A \in \mathbb{T}(\Sigma^{in}, \Sigma^{out})$. To do so, it runs in a sequence of synchronous rounds on the input configuration's underlying graph $G = (V, E)$. After each round, the automaton's global configuration is a (Q, R)-configuration $C = (G, \mathfrak{q}, \mathfrak{r})$, i.e., the underlying graph is always G. As mentioned before, for a node $v \in V$, we interpret $\mathfrak{q}(v) \in Q$ as the current state of v and $\mathfrak{r}(v) \in V^R$ as the current register valuation of v. Abusing notation, we let $C(v) \coloneqq (\mathfrak{q}(v), \mathfrak{r}(v))$ 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 $C = (G, \mathfrak{q}, \mathfrak{r}) \in \mathbb{C}(\Sigma^{in})$, the automaton's *initial configuration* is $C' = (G, \iota \circ \mathfrak{q}, \mathfrak{r}')$, where for all $v \in V$, we have $\mathfrak{r}'(v)(r) = \mathfrak{r}(v)(r)$ if $r \in R^{in}$, and $\mathfrak{r}'(v)(r) = v$ if $r \in R \setminus R^{in}$. This means that every node v is initialized to state $\iota(\mathfrak{q}(v))$, and v's initial register valuation $\mathfrak{r}'(v)$ assigns v's own identifier (provided by G) to all non-input registers while keeping the given values assigned by $\mathfrak{r}(v)$ to the input registers.

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

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

that maps from nonempty sequences of local configurations to local configurations. This allows the automaton A to transition from a given configuration C to the next configuration C' as follows. For every node $u \in V$ of degree d, we consider the list $v_1, \dots v_d$ of u's neighbors sorted in ascending (identifier) order, i.e., $v_i < v_{i+1}$ for $i \in [1 : d)$. (See Figure 1 for an example, where u corresponds to node 17.) If u is already in a halting state, i.e., if $C(u) = (q, \rho) \in H \times V^R$, then its local configuration does not change anymore, i.e., $C'(u) = C(u)$. Otherwise, we define $C'(u) = [\![\Delta]\!]\big(C(u), C(v_1), \dots, C(v_d)\big)$, which we may write more suggestively as

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

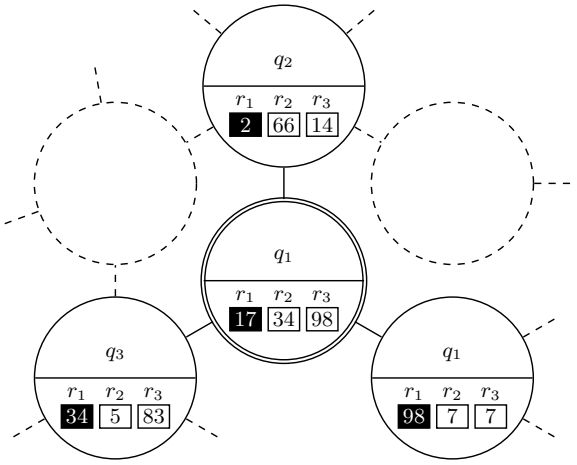


Fig. 1. Part of a configuration, as seen by a single node. Assuming the identifiers of the nodes are the values represented in black boxes (i.e., those stored in register r_1), the automaton at node 17 will update its own local configuration $(q_1, \{r_1, r_2, r_3 \mapsto 17, 34, 98\})$ by running the 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 Δ to scan a snapshot of its neighbors' local configurations. As the system is synchronous, this update procedure is performed simultaneously by all nodes.

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

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

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

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

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

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

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

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

Example 5 (Spanning tree). We present a simple automaton $A = (Q, R, \iota, \Delta, H, o)$ with input signature $\Sigma^{in} = (\mathbb{1}, \emptyset)$ and output signature $\Sigma^{out} = (\mathbb{1}, \{parent, root\})$ 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 $C = (G, \mathfrak{q}, \mathfrak{r})$, every node will store the identifier of its tree parent in register *parent* and the identifier of the root (i.e., the smallest identifier) in register *root*. Thus, as a side effect, A also solves the leader election problem by electing the root as the leader.

The automaton operates in three phases, which are represented by the set of states $Q = \{1, 2, 3\}$. A node terminates as soon as it reaches the third phase, i.e., we set $H = \{3\}$. Accordingly, the (trivial) input and output functions are $\iota\colon \varepsilon \mapsto 1$ and $o\colon 3 \mapsto \varepsilon$. In addition to the output registers, each node has an auxiliary register *self* that will always store its own identifier. Thus, we choose

Algorithm 1 Transition maker of the automaton from Example 5

```
if ∃ neighbor nb (nb.root < my.root):                                   } Rule 1
    my.state ← 1;   my.parent ← nb.self;   my.root ← nb.root            }

else if my.state = 1                                                    }
        ∧ ∀ neighbor nb [nb.root = my.root ∧                            } Rule 2
                         (nb.parent ≠ my.self ∨ nb.state = 2)]:         }
    my.state ← 2                                                        }

else if (my.state = 2 ∧ my.root = my.self) ∨ (my.parent.state = 3):     } Rule 3
    my.state ← 3                                                        }

else do nothing
```

$R = \{self, parent, root\}$. For the sake of simplicity, we describe the transition maker Δ in Algorithm 1 using pseudocode rules. However, it should be clear that these rules could be relatively easily implemented according to Definition 3.

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

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

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

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

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

Building on the automaton from Example 5, we now give an example of a graph property that can be decided in our model of distributed computing. The

following automaton should be compared to the logical formula presented later in Example 8, which is much more compact and much easier to specify.

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

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

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

There is only one exception to the above rules: if a node that is adjacent to the root cannot find any *unvisited* neighbor, it chooses the root as its *successor*. This way, the constructed path becomes a cycle. In order to check whether that cycle is Hamiltonian, the root now broadcast a *request* down the spanning tree. If the *request* reaches an *unvisited* node, that node replies by sending the message *bad* towards the root. On the other hand, every *visited* leaf replies with the message *good*. While *bad* is always forwarded up to the root, *good* is only forwarded by nodes that receive this message from all of their children. If the root receives only *good*, then it knows that the current cycle is Hamiltonian and it switches to the halting state YES. The information is then broadcast through the entire graph, so that all nodes eventually accept. Otherwise, the root sends the *backtrack* signal to its predecessor, and the search for a Hamiltonian cycle continues. In case

there is none (in particular, if there is not even an arbitrary cycle), the root will eventually receive the *backtrack* signal from its greatest neighbor, which indicates that all possibilities have been exhausted. If this happens, the root switches to the halting state NO, and all other nodes eventually do the same. □

4 Functional fixpoint logic

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

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

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

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

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

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

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

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

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

$$\mathrm{pfp}\begin{bmatrix} f_1\colon\ \varphi_1(f_1,\dots,f_\ell,\mathrm{IN},\mathrm{OUT}) \\ \vdots \\ f_\ell\colon\ \varphi_\ell(f_1,\dots,f_\ell,\mathrm{IN},\mathrm{OUT}) \end{bmatrix}\psi\,, \tag{$*$}$$

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

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

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

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

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

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

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

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

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

That is, for $i \in (\ell]$,

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

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

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

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

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

Syntactic sugar

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

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

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

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

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

5 Translating between automata and logic

Having introduced both automata and logic, we can proceed to explain the first part of Theorem 1 (stated in 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 φ_A of functional fixpoint logic that defines P. For each state $q \in Q$, our formula uses a set variable X_q to represent the set of nodes of the input graph that are in state q. Also, for each register $r \in R$, it uses a function variable f_r to represent the function that maps each node u to the node v whose identifier is stored in u's register r. By means of a partial fixpoint operator, we enforce that on any I-labeled graph $(G, \mathfrak{q})$, the final interpretations of $(X_q)_{q \in Q}$ and $(f_r)_{r \in R}$ represent the halting configuration reached by A on $(G, \mathfrak{q})$. The main formula is simply

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

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

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

Let us now consider the opposite direction and sketch how to go from functional fixpoint logic to distributed register automata (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 φ will be evaluated by a dedicated automaton A_φ, and several such automata can then be combined to build an automaton for a composite formula. For this purpose, it is convenient to design *centralized* automata, which operate on a given spanning tree (as computed in Example 5) and are coordinated by the root in a fairly sequential manner. In A_φ, each free node variable x of φ is represented by a corresponding input register x whose value at the root is the current interpretation x^σ of x. Similarly, to represent a function variable f, every node v has a register f storing $f^\sigma(v)$. The nodes also possess some auxiliary registers whose purpose will be explained below. In the end, for any formula φ (potentially with free variables), we will have an automaton A_φ computing a transduction $T_{A_\varphi} \colon \mathbb{C}(I, \{\mathit{parent}, \mathit{root}\} \cup \mathrm{free}(\varphi)) \to \mathbb{C}(\{\text{YES}, \text{NO}\}, \emptyset)$, where *parent* and *root* are supposed to constitute a spanning tree. The computation is triggered by the root, which means that the other nodes are waiting for a signal to wake up. Essentially, the nodes involved in the evaluation of φ collect some information, send it towards the root, and go back to sleep. The root then returns YES or NO, depending on whether or not φ holds in the input graph under the variable assignment provided by the input registers. Centralizing A_φ in that way makes it very convenient (albeit not efficient) to evaluate composite formulas. For example, in $A_{\varphi\vee\psi}$, the root will first run A_φ, and then A_ψ in case A_φ returns NO.

The evaluation of atomic formulas is straightforward. So let us focus on the most interesting case, namely when $\varphi = \mathrm{pfp}[f_i \colon \varphi_i]_{i\in(\ell)}\,\psi$. The root's program is outlined in Algorithm 2. Line 1 initializes a counter that ranges from 0 to $n^{\ell n} - 1$, where n is the number of nodes in the input graph. This counter is distributed in the sense that every node has some dedicated registers that together store the current counter value. Every execution of A_{inc} will increment the counter by 1, or return NO if its maximum value has been exceeded. Now, in each iteration of the loop starting at Line 2, all registers f_i and f_i^{new} are updated in such a way that they represent the current and next stage, respectively, of the partial fixpoint induction. For the former, it suffices that every node copies, for all i, the contents of f_i^{new} to f_i (Line 3). To update f_i^{new}, Line 4 calls a subroutine $\mathit{update}(f_i^{\mathrm{new}})$

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

```
1  init(A_inc)
2  repeat
3      @every node do for i ∈ [1:ℓ] do f_i ← f_i^new
4      for i ∈ [1:ℓ] do update(f_i^new)
5      if @every node (∀i ∈ [1:ℓ] : f_i^new = f_i) then goto 8
6  until execute(A_inc) returns NO      /* until global counter at maximum */
7  @every node do for i ∈ [1:ℓ] do f_i ← self
8  execute(A_ψ)
```

whose effect is that $f_i^{\mathrm{new}} = F_{\varphi_i}((f_i)_{i\in(\ell]})$ for all i, where $F_{\varphi_i}\colon (V^V)^\ell \to V^V$ 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^{new} and f_i. The corresponding truth value is propagated back to the root, where *false* is given preference over *true*. If the result is *true*, we exit the loop and proceed with calling A_ψ to evaluate ψ (Line 8). Otherwise, we try to increment the global counter by executing A_{inc} (Line 6). If the latter returns NO, the fixpoint computation is aborted because we know that it has reached a cycle. In accordance with the partial fixpoint semantics, all nodes then write their own identifier to every register f_i (Line 7) before ψ is evaluated (Line 8). □

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/beatcs/index.php/beatcs/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 Lempiäinen, 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 Lempiäinen, 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

```
if ∃ neighbor nb (nb.root < my.root):                                   } Rule 1
    my.state ← 1;   my.parent ← nb.self;   my.root ← nb.root            }

else if my.state = 1                                                    }
        ∧ ∀ neighbor nb [nb.root = my.root ∧                            } Rule 2
                         (nb.parent ≠ my.self ∨ nb.state = 2)]:         }
    my.state ← 2                                                        }

else if (my.state = 2 ∧ my.root = my.self) ∨ (my.parent.state = 3):     } Rule 3
    my.state ← 3                                                        }

else do nothing
```

A Spanning-tree automaton

A.1 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 *root* (which, at the beginning of the run, equals the identifier of the respective node). The *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.

A.2 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 $v.root = 0$ and $v.parent$ 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 $v.root = 0$, 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,

$$\text{at time } t\text{:}\quad \big(u.state = 2 \ \wedge \ u.root = u\big)\,. \tag{a}$$

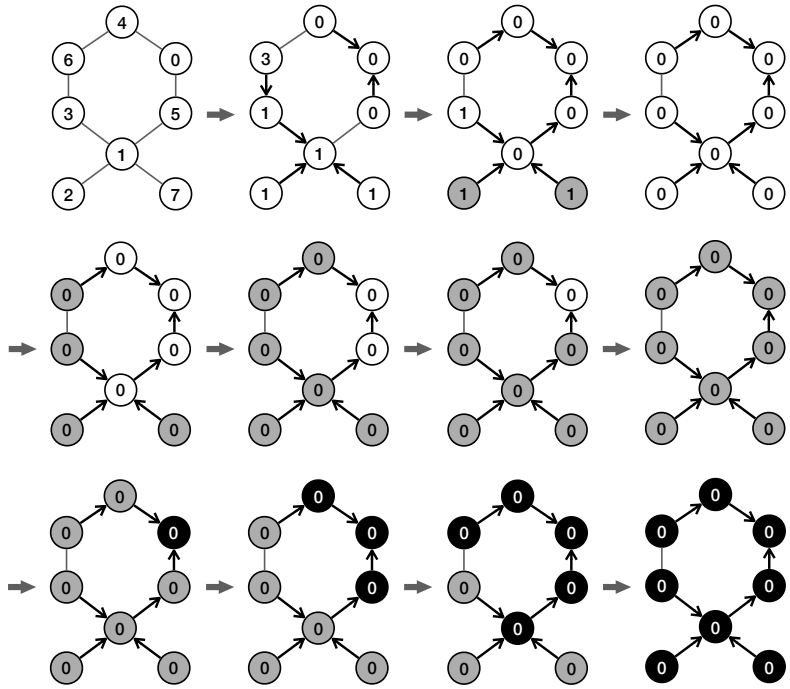


Fig. 2. A run of the spanning-tree automaton from Example 5

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 $v \in U_{i+1}$,

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

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

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

$$\text{at time } t-i\text{:} \quad \forall v \in U_i\text{:} \ \big(v.state = 2 \ \wedge \ v.root = u\big)\,. \tag{P_i}$$

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 $v.state = 2$ and $v.root = u$. Suppose that v reaches such a local configuration for the first time at time $t' \le t-i$. 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 $v'.root = u$. 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 $\hat{t} \in [t'-1 : t-(i+1)]$. 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}. □

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 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 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 $\mathcal{S}$ 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 $\mathcal{S}$. Naturally, the semantics is that "t 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:

$$\mathit{succ}[s,t] \coloneqq \big(s < t \wedge \neg\exists z(s < z \wedge z < t)\big) \vee \forall z(t \leq z \wedge z \leq s)$$

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

Now, let us consider a formula of the form $\mathrm{pfp}\big[(f_i\colon \varphi_i)_{i\in(\ell]}, (X\colon \vartheta)\big]\,\psi$, where $X \in \mathcal{S}$, $f_1, \dots, f_\ell \in \mathcal{F}$, and $\varphi_1, \dots, \varphi_\ell, \vartheta, \psi$ are formulas. We assume that the

variable OUT does not occur freely in ϑ. On an intuitive level, since X is a set variable, the set membership defined by ϑ is with respect to a single special variable, namely IN. A node v lies in the corresponding set precisely if ϑ is satisfied by interpreting 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 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 $\mathrm{pfp}\big[(f_i : \varphi'_i)_{i\in(\ell]}, (f_X : \vartheta')\big]\, \psi'$. To transform $(\varphi_i)_{i\in(\ell]}$ and ψ into their new forms $(\varphi'_i)_{i\in(\ell]}$ and ψ', 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, $\psi' \coloneqq \psi\big[t \in X \mapsto f_X(t) \neq t\big]$. For the subformula ϑ', 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

$$\vartheta' \coloneqq \vartheta\big[t \in X \mapsto f_X(t) \neq t\big] \wedge \mathit{succ}[\mathrm{IN}, \mathrm{OUT}].$$

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 id_V, which represents the empty set.

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 pfp operators.

Consider a formula of the form $\exists f\, \varphi$, where f is a function variable and φ is a formula. Obviously, the semantics is that "there exists a function f such that φ 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 φ, 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 φ. 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 φ does not contain any free occurrences of IN and OUT.[1] Hence, we can rewrite $\exists f\, \varphi$ as the partial fixpoint formula $\mathrm{pfp}[f : \psi]\, \varphi$, where

$$\psi \coloneqq \bigwedge \begin{bmatrix} [\varphi \vee \exists x, y(x < \text{IN} \wedge f(x) < y)] & \Rightarrow & \text{OUT} = f(\text{IN}) \\ \neg[\varphi \vee \exists x, y(x < \text{IN} \wedge f(x) < y)] & \Rightarrow & \mathit{succ}[f(\text{IN}), \text{OUT}] \end{bmatrix}.$$

The subformula ψ 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 φ, 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 $\mathit{succ}[s, t]$ 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 $\mathrm{pfp}[f : \psi]\, \varphi$ on a structure $(G, \mathfrak{q}), \sigma$, there are two possibilities. Either there exists no interpretation of f that satisfies φ, in which case the partial fixpoint induction does not reach a fixpoint. This means that f defaults to id_V and $\mathrm{pfp}[f : \psi]\, \varphi$ evaluates to false. Or φ can be satisfied, in which case the induction reaches a satisfying fixpoint. This fixpoint is chosen as the interpretation of f, and $\mathrm{pfp}[f : \psi]\, \varphi$ evaluates to true. In both cases, the result is the same as when evaluating $\exists f\, \varphi$ on $(G, \mathfrak{q}), \sigma$.

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 $A = (Q, R, \iota, \Delta, H, o)$ with transition maker $\Delta = (\tilde{Q}, \tilde{R}, \tilde{\iota}, \tilde{\delta}, \tilde{o})$ and assume that A decides a graph property P over label set I. In the remainder of this section, we construct a formula φ_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 $\exists x, y(x = \text{IN} \wedge y = \text{OUT} \wedge \exists f\, \varphi[\text{IN}, \text{OUT} \mapsto x, y])$, where φ'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 $(G, \mathfrak{q})$, the interpretations of $(X_q)_{q\in Q}$ and $(f_r)_{r\in R}$ represent the halting configuration reached by A on $(G, \mathfrak{q})$. Hence, the final formula is simply

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

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

$$\varphi_q \coloneqq \Big(\bigvee_{a\in I \colon \iota(a)=q} \langle a \rangle \textsc{in} \wedge \bigwedge_{p\in Q} \textsc{in} \notin X_p \Big) \vee \varphi_q^{\Delta} \vee \underbrace{\big(\textsc{in} \in X_q\big)}_{\text{only if } q\in H}.$$

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 OUT. The first disjunct of φ_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 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 φ_q^{Δ} defined below to ensure that if node IN is currently active, i.e., in a non-halting state, then IN switches to state q when executing the transition maker Δ. Finally, in case q is a halting state, the third disjunct of φ_q states that 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 φ_q^{Δ}, we use another partial fixpoint induction, which simulates the behavior of the transition maker Δ between two consecutive synchronous rounds. This second induction is thus nested within the main induction of φ_A. Similarly to before, we introduce a set variable $Y_{\tilde{q}}$ for each inner state $\tilde{q}$ and a function variable $g_{\tilde{r}}$ for each inner register $\tilde{r}$. 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 $g_\uparrow$ to represent the transition maker's reading head at each node and a set variable Y_{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 $(Y_{\tilde{q}})_{\tilde{q}\in\tilde{Q}}$ and $(g_{\tilde{r}})_{\tilde{r}\in\tilde{R}}$ represent the final inner configurations reached by the nodes. Relying on that, we define

$$\varphi_q^\Delta \coloneqq \Big(\bigvee_{p\in Q\setminus H} \textsc{in} \in X_p\Big) \wedge \operatorname{pfp}\begin{bmatrix}(Y_{\tilde{q}}\colon \psi_{\tilde{q}})_{\tilde{q}\in\tilde{Q}}\\ (g_{\tilde{r}}\colon \psi_{\tilde{r}})_{\tilde{r}\in\tilde{R}}\\ g_\uparrow\colon \psi_\uparrow\\ Y_{\text{end}}\colon \psi_{\text{end}}\end{bmatrix} \bigvee_{\substack{\tilde{q}\in\tilde{Q}\\ \exists\tilde{\beta}\colon \tilde{o}(\tilde{q})=(q,\tilde{\beta})}} \textsc{in} \in Y_{\tilde{q}}\,.$$

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

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

$$\varphi_r \coloneqq \varphi_r^\Delta \vee \Big(\bigvee_{p\in H} \textsc{in} \in X_p \wedge \textsc{out} = f_r(\textsc{in})\Big).$$

Here, the subformula φ_r^Δ states that IN is still in a non-halting state and writes the identifier of OUT to its register r after running the transition maker. The second disjunct of φ_r covers the case where 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, φ_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 φ_r^Δ is very similar to that of φ_q^Δ. In particular, it uses the same nested induction to simulate Δ:

$$\varphi_r^\Delta \coloneqq \Big(\bigvee_{p\in Q\setminus H} \textsc{in} \in X_p\Big) \wedge \operatorname{pfp}\begin{bmatrix}(Y_{\tilde{q}}\colon \psi_{\tilde{q}})_{\tilde{q}\in\tilde{Q}}\\ (g_{\tilde{r}}\colon \psi_{\tilde{r}})_{\tilde{r}\in\tilde{R}}\\ g_\uparrow\colon \psi_\uparrow\\ Y_{\text{end}}\colon \psi_{\text{end}}\end{bmatrix} \bigvee_{\substack{\tilde{q}\in\tilde{Q},\tilde{r}\in\tilde{R}\\ \exists q,\tilde{\beta}\colon \tilde{o}(\tilde{q})=(q,\tilde{\beta})\,\wedge\,\tilde{\beta}(r)=\tilde{r}}} \big(\textsc{in} \in Y_{\tilde{q}} \wedge \textsc{out} = g_{\tilde{r}}(\textsc{in})\big)$$

The only difference to φ_q^Δ is the big disjunction within the scope of the pfp operator. It stipulates that IN ends up in some inner state $\tilde{q}$ that causes the transition maker to update IN's register r to the identifier of OUT. For this to be true, the identifier of OUT must be stored in the inner register $\tilde{r}$ 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 $g_\uparrow$ and Y_{end} to keep track of the transition maker's scanning process. In each stage of the induction, $g_\uparrow$ 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 $g_\uparrow$ is initialized to the identity function at stage 0 of the nested induction. The function is then updated with the following formula:

$$\psi_\uparrow \coloneqq \textsc{in} \nleftrightarrow \textsc{out} \;\wedge\; \bigvee \begin{bmatrix} g_\uparrow(\textsc{in}) = \textsc{in} \;\wedge\; \forall x \big(\textsc{in} \nleftrightarrow x \;\Rightarrow\; \textsc{out} \leq x\big) \\ g_\uparrow(\textsc{in}) \neq \textsc{in} \;\wedge\; g_\uparrow(\textsc{in}) < \textsc{out} \;\wedge\; \forall x \big(\textsc{in} \nleftrightarrow x \wedge g_\uparrow(\textsc{in}) < x \;\Rightarrow\; \textsc{out} \leq x\big) \\ g_\uparrow(\textsc{in}) \neq \textsc{in} \;\wedge\; g_\uparrow(\textsc{in}) = \textsc{out} \;\wedge\; \forall x \big(\textsc{in} \nleftrightarrow x \;\Rightarrow\; x \leq \textsc{out}\big) \end{bmatrix}$$

Here, the first conjunct ensures that the reading head of $\textsc{in}$ can be moved only to a neighbor of $\textsc{in}$, 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 $\textsc{in}$ 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 $\psi_\uparrow$ 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_{end}, which will be interpreted as the set of all nodes that have finished scanning their neighborhood. If $\textsc{in}$'s reading head reaches the last neighbor in stage i, then $\textsc{in}$ is added to Y_{end} in stage $i+1$:

$$\psi_{\text{end}} \coloneqq g_\uparrow(\textsc{in}) \neq \textsc{in} \;\wedge\; \forall x \big(\textsc{in} \nleftrightarrow x \;\Rightarrow\; x \leq g_\uparrow(\textsc{in})\big)$$

Having formalized how the transition maker Δ scans its input, we now make use of the variables $g_\uparrow$ and Y_{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

$$\psi_{\tilde{q}} \coloneqq \Big(\textsc{in} \in Y_{\text{end}} \;\wedge\; \textsc{in} \in Y_{\tilde{q}}\Big) \;\vee\; \Big(\textsc{in} \notin Y_{\text{end}} \;\wedge\; \bigvee_{\substack{\tilde{p} \in \tilde{Q},\, p \in Q,\, \prec \in 2^{(\tilde{R} \cup R)^2} \\ \exists \tilde{\alpha} \colon \tilde{\delta}(\tilde{p}, p, \prec) = (\tilde{q}, \tilde{\alpha})}} \vartheta^{\text{curr}}_{(\tilde{p}, p, \prec)}\Big).$$

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

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

$$\begin{aligned}\vartheta^{\text{curr}}_{(\tilde{p},p,\prec)} \coloneqq\ & \vartheta^{\text{curr}}_{\tilde{p}} \ \wedge\ g_\uparrow(\text{IN}) \in X_p \ \wedge \\ & \bigwedge_{\tilde{s}\in\tilde{R},\, s\in R\colon \tilde{s}\prec s} g_{\tilde{s}}(\text{IN}) < f_s(g_\uparrow(\text{IN})) \ \wedge \ \bigwedge_{\tilde{s}\in\tilde{R},\, s\in R\colon s\prec\tilde{s}} f_s(g_\uparrow(\text{IN})) < g_{\tilde{s}}(\text{IN}) \ \wedge \\ & \bigwedge_{\tilde{s},\tilde{s}'\in\tilde{R}\colon \tilde{s}\prec\tilde{s}'} g_{\tilde{s}}(\text{IN}) < g_{\tilde{s}'}(\text{IN}) \quad \wedge \ \bigwedge_{s,s'\in R\colon s\prec s'} f_s(g_\uparrow(\text{IN})) < f_{s'}(g_\uparrow(\text{IN}))\end{aligned}$$

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

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

$$\vartheta^{\text{curr}}_{\tilde{p}} \coloneqq \begin{cases} \text{IN} \in Y_{\tilde{p}} \ \vee \bigwedge_{\tilde{q}\in\tilde{Q}} \text{IN} \notin Y_{\tilde{q}} & \text{if } \tilde{p} = \tilde{\iota}, \\ \text{IN} \in Y_{\tilde{p}} & \text{otherwise.} \end{cases}$$

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

$$\psi_{\tilde{r}} \coloneqq \big(\textsc{in} \in Y_{\text{end}} \wedge \textsc{out} = g_{\tilde{r}}(\textsc{in})\big) \vee \Bigg[\textsc{in} \notin Y_{\text{end}} \wedge \Bigg(\bigvee_{\substack{\tilde{p}\in\tilde{Q},\, p\in Q,\, \prec\in 2^{(\tilde{R}\cup R)^2},\, \tilde{s}\in\tilde{R} \\ \exists \tilde{q},\tilde{\alpha}\colon \tilde{\delta}(\tilde{p},p,\prec)=(\tilde{q},\tilde{\alpha}) \wedge \tilde{\alpha}(\tilde{r})=\tilde{s}}} \big[\vartheta^{\text{curr}}_{(\tilde{p},p,\prec)} \wedge \textsc{out} = g_{\tilde{s}}(\textsc{in})\big] \vee \bigvee_{\substack{\tilde{p}\in\tilde{Q},\, p\in Q,\, \prec\in 2^{(\tilde{R}\cup R)^2},\, s\in R \\ \exists \tilde{q},\tilde{\alpha}\colon \tilde{\delta}(\tilde{p},p,\prec)=(\tilde{q},\tilde{\alpha}) \wedge \tilde{\alpha}(\tilde{r})=s}} \big[\vartheta^{\text{curr}}_{(\tilde{p},p,\prec)} \wedge \textsc{out} = f_s(g_\uparrow(\textsc{in}))\big]\Bigg)\Bigg],$$

whose basic structure is very similar to that of $\psi_{\tilde{q}}$. The first line just states that the transition maker Δ retains the current value of $\tilde{r}$ if it has reached the end of its input. The second line covers the case where Δ has to update IN's inner register $\tilde{r}$ to a new value OUT, based on what it sees from the current inner configuration of IN and the current local configuration of $g_\uparrow(\textsc{in})$. When Δ 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 IN, or it is obtained from some register s of $g_\uparrow(\textsc{in})$. 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 $R_{tree} = \{parent, root\}$ and suppose $C \in \mathbb{C}(Q, R)$ is a configuration such that $R_{tree} \subseteq R$. 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^{in} and output registers R^{out} such that $R_{tree} \subseteq (R^{in} \setminus R^{out})$ 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 modifed 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 $S \times \{main, call\} \times Q_1$ 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 $C = ((V, E), \mathfrak{q}, \mathfrak{r}) \in \mathbb{C}(Q, R)$, with root v_0, such that the set R contains variables from $\mathcal{N} \cup \mathcal{F}$. Then, C defines a variable assignment σ as follows: For a node variable $x \in R$, we let $x^\sigma = \mathfrak{r}(v_0)(x)$. 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^σ of the function variable f by letting $f^\sigma(v) = \mathfrak{r}(v)(f)$ 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*

$$T_{A_t} \colon \mathbb{C}(\mathbb{1}, R_{tree} \cup \mathrm{free}(t)) \to \mathbb{C}(\mathbb{1}, \{t\})$$

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

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

Given a spanning-tree input configuration $C = ((V, E), \mathfrak{q}, \mathfrak{r})$, the automaton A_t will use auxiliary registers t_i to compute and store the intermediary values $t_i^\sigma = f_i^\sigma(\dots f_1^\sigma(x^\sigma)\dots)$. The register $t = t_\ell$ is the only output register. We compute the values for $t_1, \dots, t_\ell$ successively. As an invariant, we maintain that, after computing t_i^σ, the root stores t_i^σ in its register t_i.

We reserve an auxiliary register *self* for storing the own identifier of a node. To compute t_{i+1}^σ, we proceed as follows. Starting from the root, the value of t_i^σ 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}. □

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*

$$T_{A_{r{+}{+}}} \colon \mathbb{C}(\mathbb{1}, R_{tree} \cup \{r\}) \to \mathbb{C}(\mathbb{1}, \{r\})$$

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

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 $w + 1$. □

We are now ready to transform any formula into an equivalent automaton.

Proposition 15. *Let φ be a formula of functional fixpoint logic over some finite set of labels I. There is a centralized automaton A_φ computing a transduction*

$$T_{A_\varphi} : \mathbb{C}(I, R_{tree} \cup \mathrm{free}(\varphi)) \to \mathbb{C}(\{\textsc{yes}, \textsc{no}\}, \emptyset)$$

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

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

Case $\varphi = \langle a \rangle\, t$. Given a spanning-tree input configuration $C = ((V, E), \mathfrak{q}, \mathfrak{r})$ encoding a variable assignment σ, the automaton A_φ first applies, as a subroutine, A_t from Proposition 13. The latter eventually outputs a configuration in $\mathbb{C}(\mathbb{1}, \{t\})$ such that the value of register t at the root is t^σ. Similarly to the construction of Proposition 13, the root then broadcasts t^σ to all other nodes along the spanning tree, down to the leaves. During a subsequent back-propagation phase, node t^σ 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^σ and t^σ, respectively. The root then compares both values and outputs the corresponding truth value.

Case $\varphi = s \leftrightarrow t$. Like in the previous case, the root first launches A_s and then A_t so that it eventually stores s^σ and t^σ in registers s and t, respectively. Both are then propagated along the spanning tree. During a subsequent back-propagation, node s^σ checks whether t^σ 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_ψ 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 $\varphi = \varphi_1 \vee \varphi_2$. The automaton A_φ first applies A_{φ_1} as a subroutine. If the root outputs YES, then it stops. Otherwise, A_{φ_2} is launched.

Case $\varphi = \exists x\, \psi$. Automaton A_φ 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_ψ. 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_ψ again, and so on. If no increment is possible anymore, A_φ outputs NO.

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

```
1  init(A_inc)
2  repeat
3      @every node do for i ∈ [1:ℓ] do f_i ← f_i^new
4      for i ∈ [1:ℓ] do update(f_i^new)
5      if @every node (∀i ∈ [1:ℓ] : f_i^new = f_i) then goto 8
6  until execute(A_inc) returns NO      /* until global counter at maximum */
7  @every node do for i ∈ [1:ℓ] do f_i ← self
8  execute(A_ψ)
```

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

```
1′ for IN ∈ V do
2′     OUT^✓ ← IN; found ← false
3′     for OUT ∈ V do
4′         if execute(A_φi) returns YES      /* φ_i[IN, OUT] satisfied */
5′             then if found = false
6′                     then OUT^✓ ← OUT; found ← true
7′                     else OUT^✓ ← IN; goto 8′
8′     @IN do f_i^new ← root.OUT^✓
```

Case $\varphi = \mathrm{pfp}[f_i\colon \varphi_i]_{i\in(\ell]}\,\psi$. First of all, recall that $\mathrm{free}(\varphi)$ is the union of all sets $\mathrm{free}(\varphi_i) \setminus \{f_1, \ldots, f_\ell, \mathrm{IN}, \mathrm{OUT}\}$ and $\mathrm{free}(\psi) \setminus \{f_1, \ldots, f_\ell\}$. We are interested in a transduction $T_{A_\varphi}\colon \mathbb{C}(I, R_{tree} \cup \mathrm{free}(\varphi)) \to \mathbb{C}(\{\mathrm{YES}, \mathrm{NO}\}, \emptyset)$. To implement the partial fixpoint computation, we introduce auxiliary registers $f_1, \ldots, f_\ell$ (for the current interpretation) and $f_1^{\mathrm{new}}, \ldots, f_\ell^{\mathrm{new}}$ (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 ℓ-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_{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_φ 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^{new} are updated in such a way that they represent the current and next stage, respectively, of the partial fixpoint induction. For the former, it suffices that every node copies, for all i, the contents of f_i^{new} to f_i (Line 3). To update f_i^{new}, Line 4 calls the subroutine provided by Algorithm 3 (which will be explained in the

next paragraph). As a result of this algorithm, we have $f_i^{\text{new}} = F_{\varphi_i}((f_i)_{i\in(\ell]})$ for all i, where $F_{\varphi_i}\colon (V^V)^\ell \to V^V$ 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^{new} and f_i. The corresponding truth value is propagated back to the root, where *false* is given preference over *true*. If the result is *true*, we exit the loop and proceed with calling A_ψ to evaluate ψ (Line 8). Otherwise, we try to increment the global counter by executing A_{inc} (Line 6). If the latter returns YES, i.e., incrementation was possible, then another iteration takes place. However, if A_{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 ψ (Line 8).

Let us now describe the subprocedure *update*(f_i^{new}) given by Algorithm 3. Using $A_{\text{IN}^{++}}$ 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_{φ_i} to evaluate φ_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 $\text{OUT}^\checkmark$ (Line $6'$). Moreover, it sets the flag *found* to *true*, which allows it to check whether the node henceforth stored in $\text{OUT}^\checkmark$ is the only one to make φ_i true for the given IN. If it is not (i.e., the test in Line $5'$ eventually fails), we set $\text{OUT}^\checkmark = \text{IN}$. Finally, the node whose identifier corresponds to IN sets its register f_i^{new} to the computed value (Line $8'$). To implement Line $8'$, the root will send $\text{OUT}^\checkmark$, along the spanning tree, to node IN, which stores it in its register f_i^{new}.

A distributed counter. We now sketch how to implement A_{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 ℓ registers $r_0, \dots, r_{\ell-1}$, each storing a number between 0 and $n-1$. Thus, a register valuation function $\mathfrak{r}\colon V \to V^{\{r_0,\dots,r_{\ell-1}\}}$ can be seen as a ℓn-digit number written in base n, where the i-th least significant digit is stored in register $r_{(i \bmod \ell)}$ of the $\lfloor i/\ell \rfloor$-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 ℓ most significant digits are stored in the root.

Formally, A_{inc} computes a transduction

$$T_{A_{\text{inc}}}\colon \mathbb{C}(\mathbb{1}, R_{tree} \cup \{max, r_0, \dots, r_{\ell-1}\}) \to \mathbb{C}(\{\text{YES}, \text{NO}\}, \{r_0, \dots, r_{\ell-1}\}).$$

It expects as input is a spanning-tree configuration $C = (G, \mathfrak{q}, \mathfrak{r})$ that encodes some number $m \in [0 : n^{\ell n} - 1]$ in the registers $r_0, \dots, r_{\ell-1}$ (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_{inc}

is such that the new values of $r_0, \ldots, r_{\ell-1}$ represent the number $m+1$ and the root outputs YES, indicating that the incrementation was successful. Otherwise, all registers are set to zero and the root outputs NO.

In order to generate the first valid input configuration for A_{inc}, the command $\texttt{init}(A_{\text{inc}})$ in Algorithm 2 (Line 1) sets all registers r_i to 0 (thereby initializing the counter to zero) and all registers *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_{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 *max*. If not, it sets all of them to zero by copying the value 0 from register *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 *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 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 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. □

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

Proof (Proof of Proposition 10). Let φ be a formula of functional fixpoint logic without free variables. The automaton deciding the graph property defined by φ

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_φ from Proposition 15, with set of input registers R_{tree}. Finally, the root informs the other nodes, along the spanning tree, about the outcome, i.e., YES or NO. □

Towards synthesis of distributed algorithms with SMT solvers*

Carole Delporte-Gallet, Hugues Fauconnier, Yan Jurski, François Laroussinie, and Arnaud Sangnier

IRIF, Univ Paris Diderot, CNRS, France

Abstract. We consider the problem of synthesizing distributed algorithms working on a specific execution context. We show it is possible to use the linear time temporal logic in order to both specify the correctness of algorithms and their execution contexts. We then provide a method allowing to reduce the synthesis problem of finite state algorithms to some model-checking problems. We finally apply our technique to automatically generate algorithms for consensus and epsilon-agreement in the case of two processes using the SMT solver Z3.

Introduction

On the difficulty to design correct distributed algorithms. When designing distributed algorithms, researchers have to deal with two main problems. First, it is not always possible to find an algorithm which solves a specific task. For instance, it is known that there is no algorithm for distributed consensus in the full general case where processes are subject to failure and communication is asynchronous[6]. Second, they have to prove that their algorithms are correct, which can sometimes be very tedious due to the number of possible executions to consider. Moreover distributed algorithms are often designed by assuming a certain number of hypothesis which are sometimes difficult to properly formalize.

Even though most distributed algorithms for problems like leader election, consensus, set agreement, or renaming, are not very long, their behavior is difficult to understand due to the numerous possible interleavings and their correctness proofs are extremely intricate. Furthermore these proofs strongly depend on the specific assumptions made on the *execution context* which specifies the way the different processes are scheduled and when it is required for a process to terminate. In the case of distributed algorithms with shared registers, interesting execution contexts are for instance the *wait-free* model which requires that each process terminates after a finite number of its own steps, no matter what the other processes are doing [8] or the *obstruction-free* model where every process that eventually executes in isolation has to terminate [9]. It is not an easy task to describe formally such execution context and the difference between contexts can be crucial when searching for a corresponding distributed algorithm. As a matter of fact, there is no wait-free distributed algorithm to solve consensus [10], even with only two processes, but there exist algorithms in the obstruction-free case.

* Supported by ANR FREDDA (ANR-17-CE40-0013).

Proving correctness vs synthesis. When one has designed a distributed algorithm for a specific execution context, it remains to prove that it behaves correctly. The most common way consists in providing a 'manual' proof hoping that it covers all the possible cases. The drawback of this method is that manual proofs are subject to bugs and they are sometimes long and difficult to check. It is often the case that the algorithms and their specification are described at a high-level point of view which may introduce some ambiguities in the expected behaviors. Another approach consists in using automatic or partly automatic techniques based on formal methods. For instance, the tool TLA+ [3] provides a language to write proofs of correctness which can be checked automatically thanks to a proof system. This approach is much safer, however finding the correct proof arguments so that the proof system terminates might be hard. For finite state distributed algorithms, another way is to rely on model-checking [2, 14]. Here, a model for the algorithm together with a formula specifying its correctness, expressed for example in temporal logics like LTL or CTL [5], are given, and checking whether the model satisfies the specification is then automatic. This is the approach of the tool SPIN [11] which has allowed to verify many algorithms.

These methods are useful when they succeed in showing that a distributed algorithm is correct, but when it appears that the algorithm does not respect its specification, then a new algorithm has to be conceived and the tedious work begins again. One way to solve this issue is to design distributed algorithms which are correct by construction. In other words, one provides a specification and then an automatic tool synthesizes an algorithm for this specification. Synthesis has been successfully applied to various kinds of systems, in particular to design reactive systems which have to take decisions according to their environment: in such cases, the synthesis problem consists in finding a winning strategy in a two player games (see for instance [7]). In a context of distributed algorithms, some recent works have developed some synthesis techniques in order to obtain automatically some thresholds bounds for fault-tolerant distributed algorithms [12]. The advantage of such methods is that the synthesis algorithm can be used to produce many distributed algorithms and there is no need to prove that they are correct, the correctness being ensured (automatically) by construction.

Our contributions. In this work, we first define a simple model to describe distributed algorithms for a finite number of processes communicating thanks to shared registers. We then show that the correctness of these algorithms can be specified by a formula of the linear time temporal logic LTL[13, 15] and more interestingly we show that classical execution contexts can also be specified in LTL. We then provide a way to synthesize automatically distributed algorithms from a specification. Following SAT-based model-checking approach [1], we have furthermore implemented our method in a prototype which relies on the SMT-solver Z3 [4] and for some specific cases synthesizes non-trivial algorithms. Of course the complexity is high and we can at present only generate algorithms for two processes but they are interesting by themselves and meet their specification w.r.t. several execution contexts.

1 Distributed algorithms and specification language

1.1 Distributed algorithms with shared memory

We begin by defining a model to represent distributed algorithms using shared memory. In our model, each process is equipped with an atomic register that it is the only one to write but that can be read by all the others processes (*single writer-multiple readers registers*).

The processes manipulate a data set $\mathcal{D}$ including a set of input values $\mathcal{D}_{\mathcal{I}} \subseteq \mathcal{D}$, a set of output values $\mathcal{D}_{\mathcal{O}} \subseteq \mathcal{D}$ and a special value $\perp \in \mathcal{D} \setminus (\mathcal{D}_{\mathcal{I}} \cup \mathcal{D}_{\mathcal{O}})$ used to characterize a register that has not yet been written. The actions performed by the processes are of three types, they can either write a data in their register, read the register of another process or decide a value. For a finite number of processes n, we denote by $Act(\mathcal{D}, n) = \{\mathbf{wr}(d), \mathbf{re}(k), \mathbf{dec}(o) \mid d \in \mathcal{D} \setminus \{\perp\}, k \in [1, n], o \in \mathcal{D}_{\mathcal{O}}\}$ where $\mathbf{wr}(d)$ stands for "write the value d to the register", $\mathbf{re}(k)$ for "read the register of process k", and $\mathbf{dec}(o)$ for "output (or decide) the value o".

The action performed by a process at a specific instant depends on the values it has read in the registers of the other processes, we hence suppose that each process stores a local copy of the shared registers that it modifies when it performs a read or a write. Furthermore, in some cases, a process might perform different actions with the same local copy of the registers, because for instance it has stored some information on what has happened previously. This is the reason why we equip each process with a local memory as well. A process looking at its copy of the registers and at its memory value decides to perform an unique action on its local view and to update its memory. According to this, we define the code executed by a process in a distributed algorithm as follows.

Definition 1 (Process algorithm). *A* process algorithm P *for an environment of n processes over the data set $\mathcal{D}$ is a tuple (M, δ) where:*

1. *M is a finite set corresponding to the local memory values of the process;*
2. *$\delta : \mathcal{D}_{\mathcal{I}} \cup (\mathcal{D}^n \times M) \mapsto Act(\mathcal{D}, n) \times M$ is the action function which determines the next action to be performed and the update of the local memory, such that if $\delta(s) = (\mathbf{dec}(o), m')$ then $s = (\mathbf{V}, m) \in \mathcal{D}^n \times M$ and $m = m'$.*

A pair $(a, m) \in Act(\mathcal{D}, n) \times M$ is called a *move*. The last condition ensures that a process first move cannot be to decide a value (this is only to ease some definitions) and when a process has decided then it cannot do anything else and its decision remains the same. Note that the first move to be performed by the process from an input value i in $\mathcal{D}_{\mathcal{I}}$ is given by $\delta(i)$.

A process state s for a process algorithm P is either an initial value in $\mathcal{D}_{\mathcal{I}}$ or a pair $(\mathbf{V}, m) \in \mathcal{D}^n \times M$ where the first component corresponds to the local view of the processes and m is the memory value. Let $\mathcal{S}_P \subseteq \mathcal{D}_{\mathcal{I}} \cup (\mathcal{D}^n \times M)$ the states associated to P. An initial state belongs to $\mathcal{D}_{\mathcal{I}}$. We now define the behavior of a process when it has access to a shared memory $\mathbf{R} \in \mathcal{D}^n$ and its identifier in the system is $i \in [1, n]$. For this we define a transition relation $\xrightarrow{i} \subseteq (\mathcal{S}_P \times \mathcal{D}^n) \times (Act(\mathcal{D}, n) \times M) \times (\mathcal{S}_P \times \mathcal{D}^n)$ such that $(s, \mathbf{R}) \xrightarrow{i,(a,m')} (s', \mathbf{R}')$

iff for all $j \in [1,n]$ if $i \neq j$ then $\mathbf{R}[j] = \mathbf{R}'[j]$, and we are in one of the the following cases:

1. if $a = \mathbf{wr}(d)$ then $\mathbf{R}'[i] = d$ and $s' = (\mathbf{V}', m')$ such that $\mathbf{V}'[i] = d$ and, for all $j \in [1,n] \setminus \{i\}$, if $s = (\mathbf{V}, m)$ (i.e. $s \notin \mathcal{D}_{\mathcal{I}}$) then $\mathbf{V}'[j] = \mathbf{V}[j]$ and otherwise $\mathbf{V}'[j] = \perp$ i.e. the write action updates the corresponding shared register as well as the local view.
2. if $a = \mathbf{re}(k)$ then $\mathbf{R}' = \mathbf{R}$, and $s' = (\mathbf{V}', m')$ (i.e. $s \notin \mathcal{D}_{\mathcal{I}}$) with $\mathbf{V}'[k] = \mathbf{R}[k]$ and, for all $j \in [1,n] \setminus \{k\}$, if $s = (\mathbf{V}, m)$ then $\mathbf{V}'[j] = \mathbf{V}[j]$ and otherwise $\mathbf{V}'[j] = \perp$, i.e. the read action copies the value of the shared register of process k in the local view.
3. if $a = \mathbf{dec}(o)$ then $\mathbf{R}' = \mathbf{R}$ and $s' = s$, i.e. the decide action does not change the local state of any process, neither the shared registers.

The transition relation $\xrightarrow{i}_P \subseteq (\mathcal{S}_P \times \mathcal{D}^n) \times (\mathcal{S}_P \times \mathcal{D}^n)$ associated to the process algorithm P is defined by: $(s, \mathbf{R}) \xrightarrow{i}_P (s', \mathbf{R}')$ iff $(s, \mathbf{R}) \xrightarrow{i,\delta(s)} (s', \mathbf{R}')$. Different process algorithms can then be combined to form a distributed algorithm.

Definition 2 (Distributed algorithm). *A n processes* distributed algorithm *A over the data set $\mathcal{D}$ is given by $P_1 \otimes P_2 \otimes \ldots \otimes P_n$ where P_i is a process algorithm for an environment of n processes over the data set $\mathcal{D}$ for all $i \in [1,n]$.*

We now define the behavior of such a n processes distributed algorithm $P_1 \otimes P_2 \otimes \ldots \otimes P_n$. We call a *configuration* of A a pair of vectors $C = (\mathbf{S}, \mathbf{R})$ where $\mathbf{S}$ is a n dimensional vector such that $\mathbf{S}[i] \in \mathcal{S}_{P_i}$ represents the state for process i and $\mathbf{R} \in \mathcal{D}^n$ represents the values of the shared registers. We use $\mathcal{C}_A$ to represent the set of configurations of A. The initial configuration for the vector of input values $\mathbf{In} \in \mathcal{D}_{\mathcal{I}}^n$ is then simply $(\mathbf{In}, \mathbf{R})$ with $\mathbf{R}[i] = \perp$ for all $i \in [1,n]$. Given a process identifier $i \in [1,n]$ and a pair (a,m) where $a \in Act(\mathcal{D}, n)$ and m is a memory value for process i, we define the transition relations $\xrightarrow{i,(a,m)}$ over configurations as $(\mathbf{S}, \mathbf{R}) \xrightarrow{i,(a,m)} (\mathbf{S}', \mathbf{R}')$ iff we have $(\mathbf{S}[i], \mathbf{R}) \xrightarrow{i,(a,m)} (\mathbf{S}'[i], \mathbf{R}')$ and for every $j \neq i$: $\mathbf{S}'[j] = \mathbf{S}[j]$. The execution step $\overset{i}{\Rightarrow}_A$ of process i for the distributed algorithm A is defined by $(\mathbf{S}, \mathbf{R}) \overset{i}{\Rightarrow}_A (\mathbf{S}', \mathbf{R}')$ iff $(\mathbf{S}[i], \mathbf{R}) \xrightarrow{i}_{P_i} (\mathbf{S}'[i], \mathbf{R}')$, note that in that case we have $(\mathbf{S}, \mathbf{R}) \xrightarrow{i,\delta_i(S[i])} (\mathbf{S}', \mathbf{R}')$ if δ_i is the action function of P_i.

1.2 Example

Algorithm 1 provides a classical representation of a tentative distributed algorithm to solve consensus with two processes. Each process starts with an input value V and the consensus goal is that both processes eventually decide the same value which must be one of the initial values. It is well known that there is no wait-free algorithm to solve consensus [6, 8] hence this algorithm will not work for any set of executions, in particular one could check that if the two processes start with a different input value and if they are executed in a round-robin manner (i.e. process 1 does one step and then process 2 does one and so on) then none of the

Algorithm 1 Consensus algorithm for process i with i $\in \{1, 2\}$

Require: V: the input value of process i

```
 1: while true do
 2:     r[i]:=V
 3:     tmp:=r[3-i]
 4:     if tmp=V or tmp = ⊥ then
 5:         Decide(V)
 6:         Exit()
 7:     else
 8:         V:=tmp
 9:     end if
10: end while
```

process will ever decide and they will exchange their value for ever. We shall see however later that under some restrictions on the set of considered executions this algorithm solves consensus.

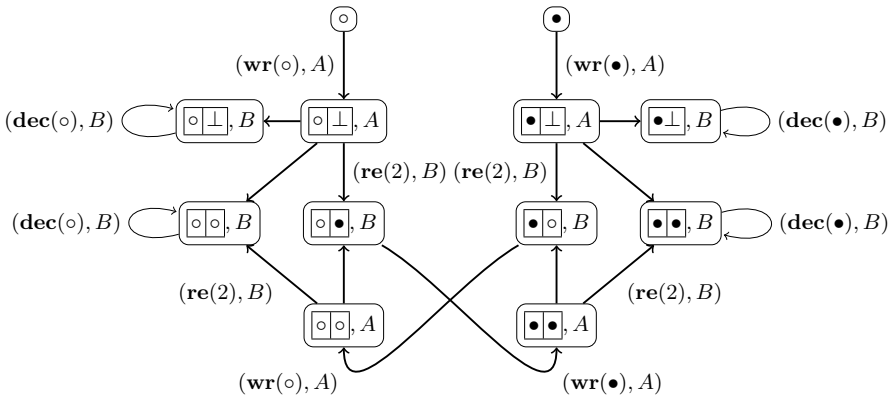


Fig. 1. View of a process algorithm P for a process with identifier 1

Figure 1 gives a visual description of the process algorithm corresponding to the Algorithm 1 supposing that the corresponding process has identifier 1. In this graph, each nodes represents a process state, the memory is the set $\{A, B\}$ and the data belongs to $\{\circ, \bullet\}$. From each node, we have some edges labeled with the action to perform according to the process state. The first action consists in writing the input data in the register, which leads to a state where the local view contains the data in the first register and $\perp$ in the local copy of the second register and the local memory cell is A. Afterwards, the process reads the second register and on Figure 1, we represent all the possible data that could be in this register (i.e either $\circ$, $\bullet$ or $\perp$) in the local view and the memory cell evolves to B.

Hence, the elements A and B of the memory set are used to represent the local state of the algorithm: when the local memory is A it means that the last action performed by the process was the write action corresponding to the Line 2 of Algorithm 1 and when its value is B, it means that the Algorithm has performed the read action corresponding to the Line 3. We only need these two values for the memory, because in our setting after having read the memory, the read value is stored in the local copy of the register and according to it, the algorithm either decides or goes back to Line 2. Note that when we leave one of the state at the bottom of the figure by reading the second register, we take into account that $\perp$ cannot be present in this register, since at this stage this register has necessarily been written.

2 Using *LTL* to reason on distributed algorithms

2.1 Kripke structures and *LTL*

We specify distributed algorithms with the Linear time Temporal Logic (LTL). We recall here some basic definitions concerning this logic and how its formulae are evaluated over Kripke structures labeled with atomic propositions from a set AP.

Definition 3 (Kripke structure). *A* Kripke structure $\mathcal{K}$ *is a* 4*-tuple* (Q, E, ℓ, q_{init}) *where* Q *is a countable set of states,* $q_{init} \in Q$ *is the initial state,* $E \subseteq Q^2$ *is a total* [1] *relation and* $\ell \colon Q \to 2^{\mathsf{AP}}$ *is a labelling function.*

A path (or an execution) in $\mathcal{K}$ from a state q is an infinite sequence $q_0 q_1 q_2 \cdots$ such that $q_0 = q$ and $(q_i, q_{i+1}) \in E$ for any i. We use $\mathsf{Path}_{\mathcal{K}}(q)$ to denote the set of paths from q. Given a path ρ and $i \in \mathbb{N}$, we write ρ^i for the path $q_i q_{i+1} q_{i+2} \cdots$ (the i-th suffix of ρ) and $\rho(i)$ for the i-th state q_i.

In order to specify properties over the execution of a Kripke structure, we use the Linear time Temporal Logic (LTL) whose syntax is given by the following grammar $\phi, \psi ::= p \mid \neg\phi \mid \phi \vee \psi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \psi$ where p ranges over AP. We use standard abbreviations: $\top$, $\perp$, $\vee$, $\Rightarrow \ldots$ as well as the classical temporal modalities $\mathbf{F}\phi \stackrel{\text{def}}{=} \top \mathbf{U} \phi$ and $\mathbf{G}\phi \stackrel{\text{def}}{=} \neg \mathbf{F} \neg \phi$. Given a path ρ of a Kripke structure $\mathcal{K} = (Q, E, \ell, q_{\text{init}})$, the satisfaction relation $\models$ for LTL is defined inductively by:

$$
\begin{aligned}
\rho &\models p \text{ iff } p \in \ell(\rho(0)) \\
\rho &\models \neg\phi \text{ iff } \rho \not\models \phi \\
\rho &\models \phi \vee \psi \text{ iff } \rho \models \phi \text{ or } \rho \models \psi \\
\rho &\models \mathbf{X}\phi \text{ iff } \rho^1 \models \phi \\
\rho &\models \phi \mathbf{U} \psi \text{ iff } \exists i \geq 0.\ \rho^i \models \psi \text{ and } \forall 0 \leq j < i.\ \rho^j \models \phi
\end{aligned}
$$

We then write $\mathcal{K} \models \phi$ iff $\rho \models \phi$ for any $\rho \in \mathsf{Path}_{\mathcal{K}}(q_{\text{init}})$. Since we quantify over all the paths, we speak of universal model-checking.

[1] *I.e.*, for all $q \in Q$, there exists $q' \in Q$ s.t. $(q, q') \in E$.

2.2 Specifying distributed algorithms

We will now see how to use LTL formulae for specifying the correctness of distributed algorithms under specific execution contexts. We consider distributed algorithms for n processes working over a data set $\mathcal{D}$. The set of atomic propositions that we will use in this context will then be : $\mathsf{AP}^n_{\mathcal{D}} = \{\mathsf{active}_i, \mathsf{D}_i\}_{1\leq i\leq n} \cup \{\mathsf{In}^d_i\}_{1\leq i\leq n, d\in\mathcal{D}_\mathcal{I}} \cup \{\mathsf{Out}^d_i\}_{1\leq i\leq n, d\in\mathcal{D}_\mathcal{O}}$ where active_i represents the fact that process i has been the last one to execute an action, D_i that process i has decided, In^d_i that the initial value of process i is d and Out^d_i that the output value of process i is d. Note that we always have: $\mathsf{D}_i \Leftrightarrow \bigvee_d \mathsf{Out}^d_i$.

We shall now see how we associate a Kripke structure labeled with these propositions with a distributed algorithm. Let $A = P_1 \otimes P_2 \otimes \ldots \otimes P_n$ be a n process distributed algorithm over the data set $\mathcal{D}$. The states of the Kripke structures contain configurations of A together with information on which was the last process to perform an action as well as the output value for each process (set to $\perp$ if the process did not output any value yet). Formally, we define $\mathcal{K}_A = (Q_A, E_A, \ell_A, q^A_{\text{init}})$ with:

- $Q_A = \{q^A_{\text{init}}\} \cup (\mathcal{C}_A \times [0,n] \times (\mathcal{D}_\mathcal{O} \cup \{\perp\})^n)$, the first component is a configuration of A, the second is the identifier of the last process which has performed an action (it is set to 0 at the beginning), the third contains the output value;
- E_A is such that:
  - $\big(q^A_{\text{init}}, ((\mathbf{In}, \perp), 0, \perp)\big) \in E$ for all initial configurations $(\mathbf{In}, \perp)$ of A (here $\perp$ stands for the unique vector in $\{\perp\}^n$), i.e. the initial configurations are the one accessible from the initial state q_{init} after one step,
  - $\big(((\mathbf{S},\mathbf{R}), i, \mathbf{O}), ((\mathbf{S}',\mathbf{R}'), j, \mathbf{O}')\big) \in E_A$ iff $(\mathbf{S},\mathbf{R}) \stackrel{j}{\Rightarrow}_A (\mathbf{S}',\mathbf{R}')$ and if the action performed by process j (from $\mathbf{S}[j]$ to $\mathbf{S}'[j]$) is $\mathbf{dec}(o)$ then $\mathbf{O}'[j] = o$ and $\mathbf{O}'[k] = \mathbf{O}[k]$ for all $k \in [1,n] \setminus \{j\}$, otherwise $\mathbf{O} = \mathbf{O}'$.
- the labelling function ℓ_A is such that:
  - $\ell_A(q^A_{\text{init}}) = \emptyset$,
  - $\mathsf{active}_i \in \ell_A((\mathbf{S},\mathbf{R}), i, \mathbf{O})$ and $\mathsf{active}_j \notin \ell((\mathbf{S},\mathbf{R}), i, \mathbf{O})$ if $j \neq i$, i.e the last process which has performed an action is i,
  - $\mathsf{In}^d_j \in \ell_A((\mathbf{S},\mathbf{R}), i, \mathbf{O})$ iff $\mathbf{S}[j] \in \mathcal{D}_\mathcal{I}$ and $d = \mathbf{S}[j]$, i.e. process j is still in its initial configuration with its initial value d,
  - $\mathsf{D}_j \in \ell_A((\mathbf{S},\mathbf{R}), i, \mathbf{O})$ iff $\mathbf{O}[j] \neq \perp$, i.e. process j has output its final value;
  - Out^d_j iff $\mathbf{O}[j] = d$, i.e. the value output by process j is d.

For a LTL formula ϕ over $\mathsf{AP}^n_{\mathcal{D}}$, we say that the distributed algorithm A satisfies ϕ, denoted by $A \models \phi$, iff $\mathcal{K}_A \models \phi$.

The LTL formulae over $\{\mathsf{In}^d_i\}_{1\leq i\leq n, d\in\mathcal{D}_\mathcal{I}} \cup \{\mathsf{Out}^d_i\}_{1\leq i\leq n, d\in\mathcal{D}_\mathcal{O}}$ will be typically used to state some correctness properties about the link between input and output values. The strength of our specification language is that it allows to specify execution contexts thanks to the atomic propositions in $\{\mathsf{active}_i, \mathsf{D}_i\}_{1\leq i\leq n}$.

Even if this is not the main goal of this research work, we know that given a n processes distributed algorithm A over a finite data set $\mathcal{D}$ and a LTL formula Φ over $\mathsf{AP}^n_{\mathcal{D}}$, one can automatically verify whether $A \models \Phi$ and this can be done in polynomial space. Indeed model-checking an LTL formula Φ over a Kripke

structure can be achieved in polynomial space [15]: the classical way consists in using a Büchi automaton corresponding to the negation of the formula Φ (which can be of exponential size in the size of the formula) and then checking for intersection emptiness *on the fly* (the automaton is not built but traveled). The same technique can be applied here to verify $A \models \Phi$ without building explicitly $\mathcal{K}_A$. Therefore we have the following result which is a direct consequence of [15]:

Proposition 1. *Given a n processes distributed algorithm A over a finite data set $\mathcal{D}$ and a LTL formula Φ over $\mathsf{AP}^n_{\mathcal{D}}$, verifying whether $A \models \Phi$ is in* PSPACE.

2.3 Examples

Specification for consensus algorithms. We recall that the consensus problem for n processes can be stated as follows: each process is equipped with an initial value and then all the processes that decide must decide the same value (*agreement*) and this value must be one of the initial one (*validity*). We do not introduce for the moment any constraints on which process has to propose an output, this will come later. We assume that the consensus algorithms work over a data set $\mathcal{D}$ with $\mathcal{D}_{\mathcal{I}} = \mathcal{D}_{\mathcal{O}}$, i.e. the set of input values and of output values are equal. The agreement can be specified by the following formula:

$$\Phi^c_{\text{agree}} \stackrel{\text{def}}{=} \mathbf{G} \bigwedge_{1 \leq i \neq j \leq n} \Big((\mathsf{D}_i \wedge \mathsf{D}_j) \Rightarrow (\bigwedge_{d \in \mathcal{D}_{\mathcal{O}}} \mathsf{Out}^d_i \Leftrightarrow \mathsf{Out}^d_j) \Big)$$

We state here that if two processes have decided a value, then this value is the same. For what concerns the validity, it can be expressed by:

$$\Phi^c_{\text{valid}} \stackrel{\text{def}}{=} \mathbf{X} \bigwedge_{1 \leq i \leq n} \bigwedge_{d \in \mathcal{D}_{\mathcal{I}}} \Big(\Big(\mathbf{F}\, \mathsf{Out}^d_i \Big) \Rightarrow \Big(\bigvee_{1 \leq j \leq n} \mathsf{In}^d_j \Big) \Big)$$

In this case, the formula simply states that if eventually a value is output, then this value was the initial value of one the processes. Note that this formula begins with the temporal operator $\mathbf{X}$ because in the considered Kripke structure the initial configurations are reachable after one step from q_{init}.

We are now ready to provide specifications for the execution context, i.e. the formulae which tell when processes have to decide. First we consider a *wait-free* execution context, each process produces an output value after a finite number of its own steps, independently of the steps of the other processes [8]. This can be described by the *LTL* formula:

$$\Phi_{\text{wf}} \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \leq n} \Big((\mathbf{G}\, \mathbf{F}\, \mathsf{active}_i) \Rightarrow (\mathbf{F}\, \mathsf{D}_i) \Big)$$

This formula states that for each process, if it is regularly (infinitely often) active, then at some point (i.e. after a finite number of steps) it must decide. Consequently if a distributed algorithm A is such that $A \models \Phi^c_{\text{agree}} \wedge \Phi^c_{\text{valid}} \wedge \Phi_{\text{wf}}$, then A is a wait-free distributed algorithm for consensus. However we know that even for two

processes such an algorithm does not exist [6, 8]. But, when considering other execution contexts, it possible to have an algorithm for consensus.

An another interesting execution context is the *obstruction-free* context. Here, every process that eventually executes in isolation has to produce an output value [9]. This can be ensured by the following LTL formula which exactly matches the informal definition.

$$\Phi_{\text{of}} \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \leq n} ((\mathbf{F}\ \mathbf{G}\ \mathsf{active}_i) \Rightarrow (\mathbf{F}\ \mathsf{D}_i))$$

The distributed algorithm $A^c_{of} = P_1 \otimes P_2$, where P_1 is the process algorithm described by Figure 1 and P_2 is the symmetric of P_1 obtained by replacing the action $\mathbf{re}(2)$ actions by $\mathbf{re}(1)$, is such that $A^c_{of} \models \Phi^c_{\text{agree}} \wedge \Phi^c_{\text{valid}} \wedge \Phi_{\text{of}}$.

Finally, another interesting context is the one corresponding to a round-robin scheduling policy. This context is given by the LTL formula, which basically states that if the n processes behave in a round-robin fashion, i.e. there are active one after another, then they all have to decide.

$$\Phi_{\text{rr}} \stackrel{\text{def}}{=} \Big[\mathbf{G}\Big(\bigwedge_{1 \leq i \leq n} (\mathsf{active}_i \Rightarrow \mathbf{X}\ \mathsf{active}_{(1+i\%n)})\Big)\Big] \Rightarrow \Big[\bigwedge_{1 \leq i \leq n} \Big(\mathbf{F}\ D_i\Big)\Big]$$

For the previously mentioned algorithm, we have $A^c_{of} \not\models \Phi_{\text{rr}}$, in fact as said in Section 1.2, if the processes are scheduled in a round-robin fashion and if their input values are different, then they will exchange their value forever and never decide. Note that we could easily define some Φ^k_{rr} formula to specify a round-robin policy where every process performs exactly k successive moves (instead of 1).

Specification for ε-agreement algorithms. We assume that the data set $\mathcal{D}$ is such that $\mathcal{D}_{\mathcal{I}}$ and $\mathcal{D}_{\mathcal{O}}$ are finite subset of $\mathbb{Q}$. We now present a variant of the ε-agreement. As for consensus, each process receives an initial value and the output values must respect the following criteria: (1) they should be between the smallest input value and the greatest one (*validity*) and (2) the outputs values all stand in an interval whose width is less or equal to ε (*agreemeent*). For instance, if we take $\mathcal{D}_{\mathcal{I}} = \{0, 1\}$ and $\mathcal{D}_{\mathcal{O}} = \{0, \frac{1}{2}, 1\}$, then if the two processes have input 0 and 1 respectively, the sets of accepted output values for $\frac{1}{2}$-agreement is $\{\{0\}, \{1\}, \{\frac{1}{2}\}, \{0, \frac{1}{2}\}, \{\frac{1}{2}, 1\}\}$. In this case, we can rewrite the formula for validity and agreement as follows:

$$\Phi^{\varepsilon}_{\text{valid}} \stackrel{\text{def}}{=} \mathbf{X}\Big[\bigvee_{d_m \leq d_M \in \mathcal{D}_{\mathcal{I}}} \Big[(\bigvee_{1 \leq i \leq n} \mathsf{In}_i^{d_m}) \wedge (\bigvee_{1 \leq i \leq n} \mathsf{In}_i^{d_M}) \wedge \mathbf{G}\Big[\Big(\bigwedge_{d < d_m \in \mathcal{D}_{\mathcal{O}}} \bigwedge_{1 \leq i \leq n} \neg\, \mathsf{Out}_i^d\Big) \wedge \Big(\bigwedge_{d > d_M \in \mathcal{D}_{\mathcal{O}}} \bigwedge_{1 \leq i \leq n} \neg\, \mathsf{Out}_i^d\Big)\Big]\Big]\Big]$$

And:

$$\Phi^{\varepsilon}_{\text{agree}} \stackrel{\text{def}}{=} \mathbf{G} \bigwedge_{1 \leq i \neq j \leq n} \Big((\mathsf{D}_i \wedge \mathsf{D}_j) \Rightarrow (\bigvee_{d,d' \in \mathcal{D}_{\mathcal{O}}\, s.t. |d'-d| \leq \varepsilon} \mathsf{Out}_i^d \wedge \mathsf{Out}_j^{d'})\Big)$$

For what concerns the specification of the execution context, we can take the same formulae Φ_{wf}, Φ_{of} and Φ_{rr} introduced previously for the consensus.

3 Synthesis

3.1 Problem

We wish to provide a methodology to synthesize automatically a distributed algorithm satisfying a specification given by a LTL formula. In this matter, we fix the number of processes n, the considered data set (which contains input and output values) $\mathcal{D}$ and the set of memory values M for each process. A process algorithm P is said to use memory M iff $P = (M, \delta)$. A distributed algoritm $A = P_1 \otimes ... \otimes P_n$ uses memory M if for $i \in [1, n]$, the process P_i uses memory M. The **synthesis problem** can then be stated as follows:

Inputs: A number n of processes, a data set $\mathcal{D}$, a set of memory values M and a LTL formula Φ over $\mathsf{AP}^n_{\mathcal{D}}$

Output: Is there a n processes distributed algorithm A over $\mathcal{D}$ which uses memory M and such that $A \models \Phi$?

We propose a method to solve this decidability problem and in case of positive answer we are able to generate as well the corresponding distributed algorithm.

3.2 A set of universal Kripke structures for the synthesis problem

We show here how the synthesis problem boils down to find a specific Kripke structure which satisfies a specific LTL formula. In the sequel, we fix the parameters of our synthesis problem: a number n of processes, a data set $\mathcal{D}$, a set of memory values M and a LTL formula Φ_{spec} over $\mathsf{AP}^n_{\mathcal{D}}$. We build a Kripke structure $\mathcal{K}_{n,\mathcal{D},M}$ similar to the Kripke structure $\mathcal{K}_A$ associated to a distributed algorithm A but where the transition relation allows all the possible behaviors (all the possible move for every process in any configuration).

First, note that each process algorithm P for an environment of n processes over the data set $\mathcal{D}$ which uses memory M has the same set of process states $\mathcal{S}_P$. We denote $\mathcal{S} = \mathcal{D}_{\mathcal{I}} \cup (\mathcal{D}^n \times M)$ this set. Similarly each n processes distributed algorithm A over $\mathcal{D}$ which uses memory M has the same set of configurations $\mathcal{C}_A$ that we will denote simply $\mathcal{C}$. We recall that these configurations are of the form $(\mathbf{S}, \mathbf{R})$ with $S \in \mathcal{S}^n$ is a vector of n processes states and $\mathbf{R} \in \mathcal{D}^n$.

The Kripke structure $\mathcal{K}_{n,\mathcal{D},M}$ uses the set of atomic propositions $\mathsf{AP}^n_{\mathcal{D}} \cup \mathsf{AP}_{\mathcal{C},\mathcal{O}}$ where $\mathsf{AP}_{\mathcal{C},\mathcal{O}} = \{\mathsf{P}_{C,\mathbf{O}} \mid C \in \mathcal{C}, \mathbf{O} \in (\mathcal{D}_{\mathcal{O}} \cup \{\bot\})^n\}$ contains one atomic proposition for every pair made by a configuration C and vector of output values $\mathbf{O}$. Its states will be the same as $\mathcal{K}_A$ but for every possible actions there will be an outgoing edge. Formally, we have $\mathcal{K}_{n,\mathcal{D},M} = (Q, E, \ell, q_{\text{init}})$ with:

- $Q = \{q_{\text{init}}\} \cup (\mathcal{C} \times [0, n] \times (\mathcal{D}_{\mathcal{O}} \cup \{\bot\})^n)$ (as for $\mathcal{K}_A$)
- E is such that:

- $\big(q_{\text{init}}, ((\mathbf{In}, \bot), 0, \bot)\big) \in E$ for all initial configurations $(\mathbf{In}, \bot)$ in $\mathcal{D}_{\mathcal{I}}^n \times \{\bot\}^n$), (as for $\mathcal{K}_A$),
- $\big(((\mathbf{S}, \mathbf{R}), i, \mathbf{O}), ((\mathbf{S}', \mathbf{R}'), j, \mathbf{O}')\big) \in E$ iff $(\mathbf{S}, \mathbf{R}) \xrightarrow{j,(a,m)} (\mathbf{S}', \mathbf{R}')$ for some $(a, m) \in Act(\mathcal{D}, n) \times M$. And:
  - if $a = \mathbf{dec}(o)$ then $\mathbf{S}[j] = (\mathbf{V}, m)$ for $\mathbf{V} \in \mathcal{D}^n$ and $\mathbf{O}'[j] = o$ and $\mathbf{O}'[k] = \mathbf{O}[k]$ for all $k \in [1, n] \setminus \{j\}$, otherwise $\mathbf{O} = \mathbf{O}'$ (the memory cells does not change once the decision is fixed),
  - if $\mathbf{O}[j] \neq \bot$, then $a = \mathbf{dec}(\mathbf{O}[j])$ (the decision cannot change, no other action can be performed).

– the labelling function ℓ is defined the same way as in $\mathcal{K}_A$ for the atomic propositions in $\mathsf{AP}_{\mathcal{D}}^n$ and $\mathsf{P}_{C,O} \in \ell((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$ iff $C = (\mathbf{S}, \mathbf{R})$ and $O = \mathbf{O}$.

Hence the relation E simulates all the possible moves from any configuration $(\mathbf{S}, \mathbf{R})$ and the Kripke structure $\mathcal{K}_{n,\mathcal{D},M}$ contains all possible executions of any n processes algorithms over $\mathcal{D}$ using memory M.

Defining an algorithm consists in selecting exactly one action for each process in every configuration. Here we do this by adding to the structure extra atomic propositions $\mathsf{P}^i_{(a,m)}$ with $1 \leq i \leq n$ and $(a, m) \in Act(\mathcal{D}, n) \times M$ which specifies for each configuration what should be the next move of process i. We denote by $\mathsf{AP}^n_{Act,M}$ this set of new atomic propositions. An *algorithm labelling* for $\mathcal{K}_{n,\mathcal{D},M}$ is then simply a function $\ell' : Q \mapsto 2^{\mathsf{AP}^n_{Act,M}}$. We denote by $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ the Kripke structure obtained by adding to $\mathcal{K}_{n,\mathcal{D},M}$ the extra labelling provided by ℓ'. When defining such an algorithm labelling, we need to be careful that it corresponds effectively to a distributed algorithm: our processes are deterministic (only one action is allowed for P_i in some configuration) and a process has to choose the same action when its local view is identical. Such an algorithm labelling ℓ' is said to be *consistent* iff the following conditions are respected:

1. $\ell'(q_{\text{init}}) = \emptyset$,
2. for all $((\mathbf{S}, \mathbf{R}), i, \mathbf{O}) \in Q$, for all $j \in [1, n]$ there exists a unique $\mathsf{P}^j_{(a,m)} \in \ell'((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$, each process has exactly one move in each configuration,
3. for all $((\mathbf{S}, \mathbf{R}), i, \mathbf{O}), ((\mathbf{S}', \mathbf{R}'), j, \mathbf{O}') \in Q$, if $\mathbf{S}[k] = \mathbf{S}'[k]$ and if $\mathsf{P}^k_{(a,m)} \in \ell'((\mathbf{S}, \mathbf{R}), i, \mathbf{O})$ then $\mathsf{P}^k_{(a,m)} \in \ell'((\mathbf{S}', \mathbf{R})', j, \mathbf{O}')$, i.e. in all configuration with the same state of process k, the moves of process k must be the same.

A consistent algorithm labelling ℓ' induces then a distributed algorithm $A^{\ell'} = P_1 \otimes \ldots \otimes P_n$ where for all $j \in [1, n]$, we have $P_i = (M, \delta_i)$ and $\delta_i(s) = (a, m)$ iff for all configurations $((\mathbf{S}, \mathbf{R}), j, \mathbf{O}) \in Q$ such that $\mathbf{S}[i] = s$, we have $\mathsf{P}^i_{(a,m)} \in \ell'(((\mathbf{S}, \mathbf{R}), j, \mathbf{O}))$. Conditions 1. to 3. ensure that this definition is well-founded.

To check by the analysis of the Kripke structure $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ whether the algorithm $A^{\ell'}$ induced by a consistent algorithm labelling satisfies the specification Φ_{spec}, we have to find a way to extract from $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ the execution corresponding to $A^{\ell'}$. This can be achieved by the following LTL formula:

$$\Phi_{\text{out}} \stackrel{\text{def}}{=} \mathbf{X}\,\mathbf{G}\Big[\bigvee_{C \in \mathcal{C}} \bigvee_{\mathbf{O} \in \mathcal{O}} \bigvee_{i,a,m} \Big(\mathsf{P}^i_{(a,m)} \wedge \mathsf{P}_{C,\mathbf{O}} \wedge \mathbf{X}(\mathsf{active}_i \Rightarrow \mathsf{P}_{\text{Next}(C,\mathbf{O},i,a,m)})\Big)\Big]$$

where $\mathrm{Next}(C, \mathbf{O}, i, a, m)$ is the (unique) extended configuration $(C', \mathbf{O}')$ such that $C \xrightarrow{i,(a,m)} C'$ and $\mathbf{O}[j] = \mathbf{O}'[j]$ for all $j \neq i$ and $\mathbf{O}'[i] = o$ if $a = \mathbf{dec}(o)$ otherwise $\mathbf{O}'[i] = \bot$. We can then combine Φ_{out} with the correctness specification Φ_{spec} to check in $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ whether the executions of $A^{\ell'}$ (which are the executions of $\mathcal{K}_{A^{\ell'}}$) satisfy Φ_{spec}.

Proposition 2. *Given a consistent algorithm labelling ℓ' and its induced distributed algorithm $A^{\ell'}$,*

$$A^{\ell'} \models \Phi_{spec} \quad \textit{iff} \quad \mathcal{K}^{\ell'}_{n,\mathcal{D},M} \models \Phi_{out} \Rightarrow \Phi_{spec}$$

Sketch of proof. To prove this it is enough to see that the control states of $\mathcal{K}_{A^{\ell'}}$ and of $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ are the same and that any infinite sequence of such states ρ beginning in q_{init} is an execution in $\mathcal{K}_{A^{\ell'}}$ iff it is an execution in $\mathcal{K}^{\ell'}_{n,\mathcal{D},M}$ verifying Φ_{out}. □

Consequently, to solve the synthesis problem it is enough to find a consistent algorithm labelling ℓ' such that $\mathcal{K}^{\ell'}_{n,\mathcal{D},M} \models \Phi_{\mathrm{out}} \Rightarrow \Phi_{\mathrm{spec}}$. Note that as explained before this produces exactly the correct algorithm $A^{\ell'}$. We have hence a decision procedure for the synthesis problem: it reduces to some instances of model-checking problem for LTL formulae.

4 Experiments

We have implemented a prototype to automatically synthesize algorithms for consensus and ε-agreement problems. For this we use the SMT solver Z3[4]: it is now classical to use SAT solver for model-checking [1] and it was natural to consider this approach especially because we need to add an existential quantification over the atomic propositions encoding the moves of the processes [2]. Our prototype is however a bit different from the theoretical framework explained in Section 3 and we explain here the main ideas behing its implementation.

First, the implementation does not consider general (quantified) LTL formulas but encodes directly the considered problem (consensus or ε-agreement) for a set of parameters provided by the user into a Z3-program, and the result provided by the SMT solver Z3 is then *automatically* analysed in order to get algorithms for processes.

We now sketch the main aspects of the reduction to Z3. The code starts by existentially quantifying over the action functions for each process: an action function δ^p for a process p is encoded as an integer value δ^p_s for every process state s which gives the next action to performed. In Z3, such a δ^p_s is a bitvector (whose size is $\log_2(|Act(\mathcal{D}, n) \times M| + 1)$). It remains to encode the different properties we want to ensure (depending on the considered problem). Here are several examples:

[2] We do not describe here the reduction: it uses standard techniques for encoding LTL formulae to SAT instance.

– To deal with the formula Φ^c_{agree} for the consensus, we use a set of Boolean constants (one for every global configuration C). Their truth value can be easily defined as true when all processes in C have terminated and decided the same value, or as false when at least two processes have decided different values in C. For the other cases, we add constraints stating that the value associated with C equals true when for every successor (here a successor is any configuration reachable after an action (a, m) of some process p such that this action (a, m) corresponds to the value δ^p_s where s is the state of p in C). It remains to add a last constraint: for every initial configuration C_0, the constant associated with Φ^c_{agree} has to be true. Note that this definition is based on the fact that the property is an invariant: we want to ensure that no reachable configuration violates a local property.
– Encoding the formula Φ^c_{valid} follows the same approach: we use a boolean value for every configuration C and for every input data d, and define their truth value in such a way that it is true iff the value d cannot be decided in the next configurations. If some process has already decided d in C, the constant equals to false. If all processes have decided and no one choose d, it is true. Otherwise a positive value requires that for every successor C', the constants are also true. Finally we add constraints specifying for every initial configuration C_0 the values d that cannot be chosen by requiring that their corresponding values are true.
– The obstruction free context Φ_{of} is encoded as follows: we need two sets of constants for every process p. The first set contains one integer value (encoded as a bitvector in Z3) for every configuration and it is defined in order to be the number of moves that process p has to perform (alone) to decide (and terminate). This distance is bounded by the number of states nb_{state} of process p (and we use the value nb_{state} to represent the non-termination of the process). In addition, we consider a set of boolean values (one for every configuration) which are defined in order to equal to true iff for every reachable configuration from C, the computed distance is strictly less than nb_{loc}.
– Encoding the wait-free context uses the same idea. We have to verify that from every reachable configuration, every process will terminate (for this we use the fact that when a process decides a value, it does not perform action anymore, and then other processes progress). Note that in this case, the bound on the distance is the number of *global* configurations.

In addition to this encoding, we can also use standard techniques of bounded model-checking by fixing a smaller bound for the computation of the distances described above. When this is done, the program may provide an algorithm, or answer that an algorithm **with this bound** does not exist (it remains to try with a greater bound). This heuristic is crucial to synthesize algorithms in many cases (the computation of distances is quite expensive since it is connected to the number of states or configurations).

The parameters of our prototype are then: (1) the number of processes: n, (2) the range of initial values and the range of possible values in registers, (3)

the size of the processess memory, (4) the types of scheduling policy (wait free, obstruction free, round-robin, or a combination of them), and (5) the value of ε for the ε-agreement problem. Finally one can ask for symmetric programs (each process has the same action function) and in the following we only consider symmetric solutions.

State explosion problem. As explained in previous sections, we are faced with a huge complexity. For example, with 2 processes, two possible initial values and a memory size equals to 2, there are more than 450 configurations for the distributed algorithms. If we consider 3 processes, 2 initial values et a memory size equals to 3, we get more than 240 *thousands* configurations ! This gap explains why our prototype only provides algorithms for 2 processes. Note that even for the case $n = 2$, the complete encoding of the problem may use several thousands of variables in the Z3 code, and the SMT solver succeeds in providing a result. Of course, the implementation of our prototype in its current form is quite naive and some efficiency improvements are possible.

Moreover note that our prototype is often more efficient for finding algorithms when they exist than for proving that no algorithm within the resource fixed by the parameters[3] exists. First it is often easier to find a valuation than verifying that no valuation exists, and secondly we can use heuristics to accelerate the procedure (for example by bounding the length of computations: in this case, if a valuation is found, we can stop, otherwise we have to try again with different settings). This fact can be seen as a variant of a well-known phenomenon in sat-based model-checking: it is usually very efficient to find a bug (that is an execution satisfying or not a formula), but it is not the case to prove full verification.

Consensus. For 2 components, 2 initial and final values, a memory of size 2 and the obstruction free policy, we get the algorithm of Section 1.2 (Figure 1) except that the processes use their register to write the value they *do not* plan to decide (it is clearly symmetric to the previous algorithm). Note that the size of memory is important: there is no algorithm with memory of size 1: indeed we need to distinguish the configuration $(0, 0)$ (the proper register equals to 0 and the last read value of the register of other process is 0) when it is reached after a Read (both process agree on the value to decide) and when it is reached after a Write(0) performed by the process to update its register in order to agree with the other process. This absence of algorithm with a memory of size 1 corresponds to an UNSAT result for the program: the formula Φ_{synth} with these parameters is not satisfiable. When we tried to look for algorithms for wait-free case, we found no solution with our program: indeed we know that there is no such algorithms !

More interestingly we can ask for a program correct w.r.t. several execution contexts. For example, we can ask for program correct w.r.t. obstruction free, round-robin for one step and also round-robin for two steps. The program generates [4] the algorithm depicted in Figure 2 (we follow the same presentation

[3] Note that we cannot prove that no algorithm exists, but only that no algorithm *with this memory bound* exists if the corresponding SAT instance has no solution.

[4] It takes few seconds to produce the algorithm on a standard laptop.

as in Section 1 for the algorithm and since we have only two processes, we use $(\mathbf{re}, -)$ instead of $(\mathbf{re}(1), -)$: a read operation always deals with the other process).

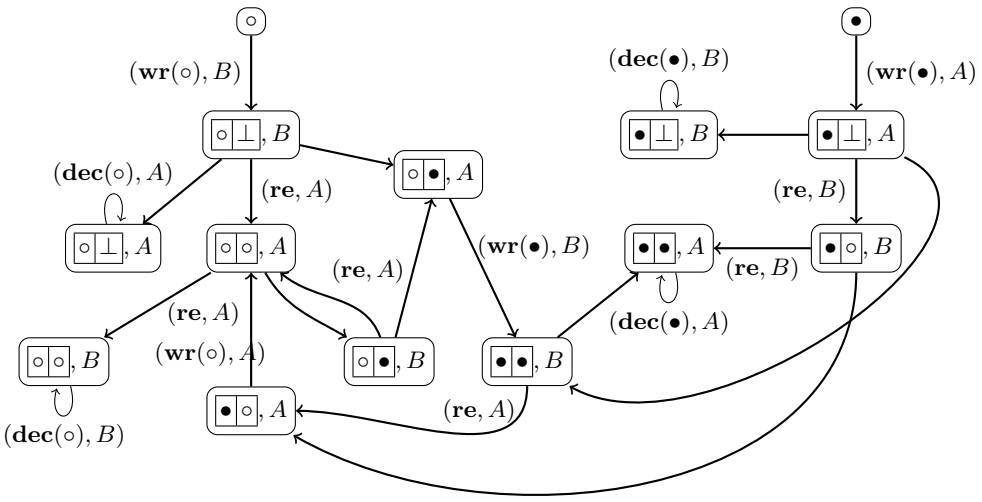


Fig. 2. View of a process algorithm P for consensus, w.r.t. to obstruction free and round-robin 1 and 2.

ε-agreement. For this problem, we have to fix ε. In Figure 3, we present an algorithm for $\frac{1}{5}$-agreement for 2 processes, with initial values $\{0, 1\}$ and memory 3. The set of possible decision values is $\{0, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1\}$. Note that this algorithm works for the wait-free execution context, and therefore also for round-robin (for any step) and for obstruction free. Here the memory size equals to 3: this is illustrated by the fact that the configuration $(0, 0)$ (the register's value is 0 and the last read value from the other process is 0) appears in three nodes.

5 Conclusion

We have shown here that in theory it is possible to solve the synthesis problem for distributed algorithm as soon as we fix the set of data that can be written in the registers and the memory needed by each process in the algorithm. However even if this problem is decidable, our method has to face two different problems: first, it does not scale and second, when the answer to the synthesis problem is negative, we cannot conclude that there is no algorithm at all. In the future, we will study more intensively whether for some specific cases we can decide

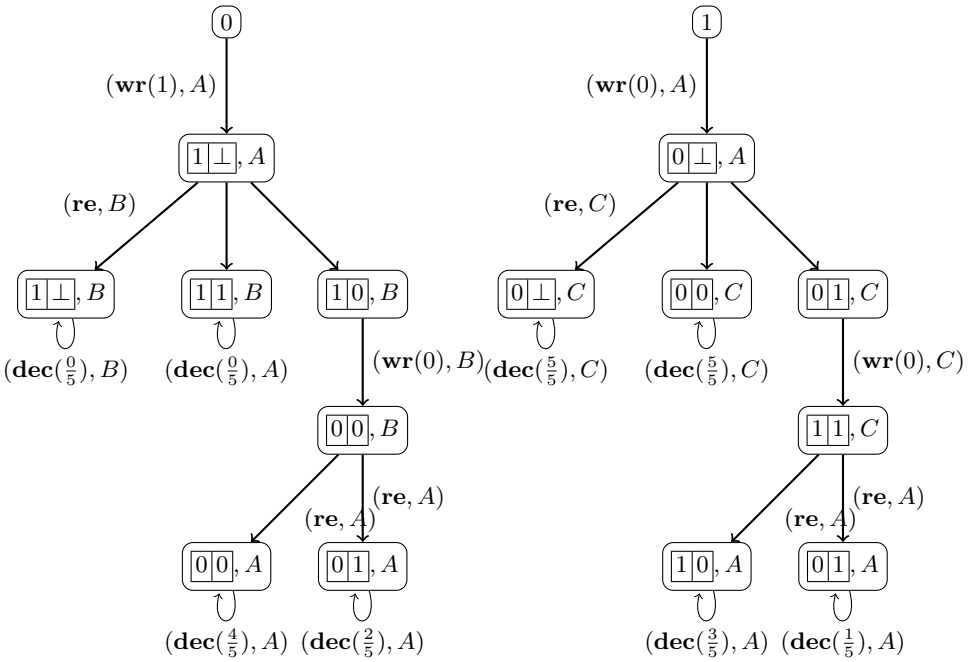


Fig. 3. View of a process algorithm P for $\frac{1}{5}$-agreement, w.r.t. wait free scheduling.

the existence of a distributed algorithm satisfying a given specification without fixing any restrictions on the exchanged data or on the size of the algorithms. We believe that for some specific distributed problems, this is in fact feasible.

References

1. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
2. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. C. Kozen, editor, *LOP'81*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1982.
3. D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA + proofs. In *FM'12*, volume 7436 of *LNCS*, pages 147–154. Springer, 2012.
4. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
5. E. A. Emerson. Temporal and modal logic. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier, 1990.
6. M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
7. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *LNCS*. Springer, 2002.
8. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
9. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS'03*, pages 522–529, 2003.
10. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
11. G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
12. M. Lazic, I. Konnov, J. Widder, and R. Bloem. Synthesis of distributed algorithms with parameterized threshold guards. In *OPODIS'17*, volume 95 of *LIPIcs*, pages 32:1–32:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
13. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Comp. Soc. Press, Oct.-Nov. 1977.
14. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *SOP'82*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, Apr. 1982.
15. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS'86*, pages 332–344. IEEE Computer Society, 1986.