Undecidability of the Emptiness Problem for Weak Models of Distributed Computing

Flavio T. Principato
Technical University of Munich, Germany

Javier Esparza
Technical University of Munich, Germany

Philipp Czerner
Technical University of Munich, Germany

Abstract

Esparza and Reiter have recently conducted a systematic comparative study of weak asynchronous models of distributed computing, in which a network of identical finite-state machines acts cooperatively to decide properties of the network's graph. They introduced a distributed automata framework encompassing many different models, and proved that w.r.t. their expressive power (the graph properties they can decide) distributed automata collapse into seven equivalence classes. In this contribution, we turn our attention to the formal verification problem: Given a distributed automaton, does it decide a given graph property? We consider a fundamental instance of this question – the *emptiness problem*: Given a distributed automaton, does it accept any graph at all? Our main result is negative: the emptiness problem is undecidable for six of the seven equivalence classes, and trivially decidable for the remaining class.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Undecidability, Emptiness Problem, distributed Automata

Digital Object Identifier 10.4230/LIPIcs.SAND.2025.5

Related Version *Full version with appendices available at*: `https://arxiv.org/abs/2504.07339`

1 Introduction

The concepts of distributed computing can be used to model interactions between a variety of natural or artificial agents, like molecules, cells, microorganisms, or nanorobots. Typical features of these models are that agents do not have identities, and each agent has very limited computational power and restricted communication capabilities. Weak models of distributed computing are hence the appropriate paradigm in this context – in contrast to traditional distributed computing models used to study computer networks. Examples of such models include population protocols [3, 4], chemical reaction networks [13], networked finite-state machines [7], the weak models of distributed computing of [10], and the beeping model [1, 5]. Many of these and other models are discussed in comparative surveys [9, 12].

All these weak models share the following features [7]: the communication network can have arbitrary topology; all agents run the same protocol; each agent has a finite number of states, independent of the network size or topology; state transitions only depend on the agent's state and the states of a bounded number of neighbours; nodes do not know their neighbours, in the sense of [2]. Nonetheless, they differ in several other aspects. Esparza and Reiter classified them in [8] according to four parameters:

- *Detection.* In some models, agents can only detect the *existence* of neighbours in a given state (`d`), while in others they can *count* their number up to a fixed threshold (`D`).

- *Acceptance.* A given input is accepted or rejected by all agents reaching a respective state. Some models require agents to *halt* once they reach an accepting or rejecting state (`a`), while others only require *stable consensus* (`A`), i.e., agents can keep changing their decision, as long as they eventually agree on acceptance or rejection.
- *Selection.* In each step of the execution of a model, a certain selection of agents acts. In *synchronous* models (`$`), all agents are selected in every step. Models with *liberal* selection (`s`) select an arbitrary subset of agents in each step. Finally, models with *exclusive* selection (`S`) select exactly one agent at a time.
- *Fairness.* Different assumptions about the occurrence of the permitted selections can be made. Some models only ensure that every agent will be selected infinitely many times. We call this *weak fairness* (`f`). Others use stochastic-like selection, which guarantees that any finite sequence of permitted selections will occur infinitely often. We call this *strong fairness* (`F`).

In [8], Esparza and Reiter studied the expressive power of all possible combinations of these parameters. For this, they introduced a generic model, called *distributed automata*, which can be equipped with any parameter combination – for example, one can study the class of `dA$f`-distributed automata. The behaviour of a distributed automaton is described by a finite-state machine, inputs are labelled graphs and the output is boolean (acceptance/rejection). Intuitively, each node of the input graph becomes an agent running a copy of the finite-state machine where the initial state depends on the node's label, and its transitions depend on the agent's own and the agent's neighbours' states. At each step of the execution a subset of the nodes is selected by a scheduler, and each selected agent executes a transition of the finite-state machine, thereby moving into a new state. Note that this adds a component of nondeterminism, such that multiple runs on the same input graph could a priori yield different results. This ambiguity is resolved by requiring the *consistency condition*, which essentially states that for a given input graph the distributed automaton's output must always be the same. The set of graphs accepted by a given distributed automaton forms a graph language; we say the automaton decides the graph language. A model's expressive power is determined by the class of graph languages it can decide.

The main result of [8] is that all the possible combinations of parameters collapse into seven equivalence classes w.r.t. their expressive power. Notably, all proofs of equivalence are constructive. In particular, every class of distributed automata was shown to be equivalent to some class with liberal selection (`s`). For this reason, we can often omit the selection parameter when referring to an equivalence class $xyz \in \{\mathtt{d},\mathtt{D}\}\{\mathtt{a},\mathtt{A}\}\{\mathtt{f},\mathtt{F}\}$, implicitly meaning $xy\mathtt{s}z$. Furthermore, the classes $\mathtt{da}z$ with $z \in \{\mathtt{f},\mathtt{F}\}$ were shown to both have trivial expressive power, i.e., every $\mathtt{da}z$-distributed automaton either accepts all labelled graphs or none. The seven distinct equivalence classes assemble in a hierarchy of expressive power shown in Figure 1. Finally, we mention that Czerner *et al.* [6] characterized which labelling properties each class can decide, i.e., properties depending only on the labelling and not on the structure of the graph.

Distributed automata are designed to decide properties of graphs, e.g., whether a given graph is cyclic or whether at least one of its nodes is labelled as red. So, we are interested in the verification problem: Does the graph language $L(A)$ of a given distributed automaton A satisfy a given property, i.e., does $L(A) \subseteq \mathcal{C}$ hold, where $\mathcal{C}$ is the class of graphs with the desired property? This is equivalent to asking if $L(A) \cap \overline{\mathcal{C}} = \emptyset$ holds, where $\overline{\mathcal{C}}$ denotes the complement of $\mathcal{C}$. Since distributed automata are closed under intersection (as we show in this paper), verification reduces to the *emptiness problem* whenever $\overline{\mathcal{C}}$ is decidable by distributed automata.

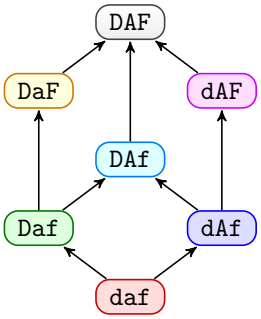

Figure 1 The hierarchy of expressive power of classes of distributed automata as proven in [8] – The arrows indicate strictly increasing expressive power.

We show that the emptiness problem is undecidable for six of the seven classes, and trivially decidable for the class of `daf`-distributed automata[1]. We reduce from the halting problem for Turing machines on blank tape. For each class xyz, we define a suitable family of labelled graphs to represent a finite tape section. Then we construct an xyz-distributed automaton that solves two tasks: it decides whether the graph belongs to the suitable family and, if so, simulates the execution of a given Turing machine until either the machine halts, in which case the automaton accepts, or the head exceeds the finite tape section, in which case the automaton rejects. So, we need families of labelled graphs that can be used to represent arbitrarily long finite tape sections, and are decidable by xyz-distributed automata. This is especially difficult for the classes of the form `Da`z, which have extremely limited expressive power. Identifying a suitable family of labelled graphs for this case is the main contribution of our paper.

Related work. Kuusisto and Reiter have studied the emptiness problem for a specific class of automata on directed graphs working synchronously [11]. This work predates [8], which considers a large variety of models on undirected graphs[2].

Structure of the paper. Section 2 contains preliminaries. Section 3 identifies suitable families of graphs for the more capable classes `DA`z and `dA`z, and prepares the ground for Section 4, which introduces a suitable family for `Da`z. Section 5 presents conclusions. Formal descriptions of the constructions and detailed proofs are given in the appendices, which can be found in the full version.

2 Preliminaries

For sets A and B, $\mathcal{P}(A)$ denotes the powerset of A and B^A denotes the set of all functions from A to B. Given a partial function $f\colon A \to B$, we write $f(a) = \perp$ to denote that f is not defined for $a \in A$. We use the convention $0 \in \mathbb{N}$ and define $[n] \coloneqq \{0, ..., n\}$ for $n \in \mathbb{N}$ and $[-1] \coloneqq \emptyset$.

[1] Since a `daf`-distributed automaton either accepts all graphs or no graph, emptiness can be decided by checking whether the automaton accepts, e.g., the graph with one node and no edges, which is easy.

[2] Unfortunately, [11] and [8] use the name distributed automata with different meanings: [11] for a specific automata model, and [8] as a generic name for all the automata classes in their classification.

Words. Given a word $w \in \Sigma^*$ over an alphabet Σ, we let $|w|$ denote the length of w and $w_{i\dots j}$ the substring of w from the i^{th} to the j^{th} symbol for $i, j \in [|w| - 1]$, with $w_{i\dots j} \coloneqq \varepsilon$ for $j < i$ and $w_i \coloneqq w_{i\dots i}$. Note that we use 0-indexing. For $a \in \Sigma$, a^ω denotes the infinite repetition of a.

Graphs. We implicitly assume graphs to be non-empty, finite, undirected, unweighted, and connected. A labelled graph over a set of labels Λ is a triple $G = (V, E, \lambda)$, where V and E are sets of nodes and edges, and $\lambda\colon V \to \Lambda$ is a labelling function that assigns a label to each node $v \in V$. For $U \subseteq V, v \in V$, the function $\operatorname{dist}(U, v)$ denotes the length of the shortest path from any node in U to v.

Turing machines. A Turing machine (TM) is a 7-tuple $T = (Q, q_0, F, \Gamma, \Sigma, \square, \delta)$ with a finite set of states Q, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, a tape alphabet Γ, an input alphabet $\Sigma \subseteq \Gamma$, the blank symbol $\square \in \Gamma \setminus \Sigma$, and a (partial) transition function $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ with $\delta(f, \gamma) = \bot$ for all $f \in F, \gamma \in \Gamma$. In our model, the TM tape is only unbounded to the right. The TM head starts on the leftmost cell and moves right ($+1$) or left (-1) in each transition.

2.1 Distributed Machines

A *distributed machine* operating on a *labelling alphabet* Λ with counting bound $\beta \in \mathbb{N} \setminus \{0\}$ is a 5-tuple $M = (Q, \delta_0, \delta, Y, N)$ with a finite set of *states* Q, an *initialization function* $\delta_0\colon \Lambda \to Q$, a *transition function* $\delta\colon Q \times [\beta]^Q \to Q$, and disjoint sets $Y, N \subseteq Q$ of *accepting* and *rejecting states*, respectively.

A distributed machine M runs on a labelled graph $G = (V, E, \lambda)$. Intuitively, at each node $v \in V$ an *agent* starts in the state $\delta_0(\lambda(v))$. The machine proceeds in steps, where in each step a set S of nodes is selected. Each agent $v \in S$ transitions to a new state according to the function δ, which depends on v's and v's neighbours' current states.

We proceed to formalize this intuition. A *configuration* C of M on G is a function $C\colon V \to Q$ that assigns a *current state* to every agent of G. For every configuration C and agent v, we define the function $\mathcal{N}_v^C\colon Q \to [\beta], q \mapsto \min\{n_q(v), \beta\}$, where $n_q(v) \in \mathbb{N}$ is the number of agents w neighbouring v with $C(w) = q$. To simplify notation, we define for $Q = R \times S \times T$ that $\mathcal{N}_v^C(r, *, t) = \sum_{s \in S} \mathcal{N}_v^C(r, s, t)$. Given two configurations C, D and a *selection* $S \subseteq V$, we let $C \xrightarrow{S}_M D$ denote that $D(v) = \delta\big(C(v), \mathcal{N}_v^C\big)$ for all $v \in S$ and $D(v) = C(v)$ otherwise, that is, the machine transitions from C to D by performing state transitions according to δ for exactly the agents selected by S. We write $C \to_M D$ if there exists some selection $S \subseteq V$ such that $C \xrightarrow{S}_M D$.

2.2 Distributed Automata

In principle, a distributed automaton consists of a distributed machine and a scheduler specifying which sequences of selections are allowed. All schedulers guarantee the minimal assumption that every node is selected infinitely often.

A *schedule* is a sequence of selections $\sigma = (S_i)_{i \in \mathbb{N}}$ such that for every $v \in V$ there exist infinitely many $i \in \mathbb{N}$ with $v \in S_i$, i.e., every agent is selected infinitely often. A sequence of configurations $\rho = (C_i)_{i \in \mathbb{N}}$ with $C_0 = \delta_0 \circ \lambda$ and $C_i \to_M C_{i+1}$ for all $i \in \mathbb{N}$ is called a *run* of M on G. We say that ρ is *induced* by σ iff $C_i \xrightarrow{S_i}_M C_{i+1}$ for all $i \in \mathbb{N}$. A configuration C is called *accepting* iff $C(V) \subseteq Y$ and *rejecting* iff $C(V) \subseteq N$. A run $\rho = (C_i)_{i \in \mathbb{N}}$ is called *accepting* (*rejecting*) iff there exists an index $I \in \mathbb{N}$ such that C_i is accepting (rejecting) for all $i \geq I$.

A *scheduler* is a pair $\Sigma = (s, f)$. The *selection constraint* s maps any graph $G = (V, E, \lambda)$ to a subset of $\mathcal{P}(V)$ of *permitted* selections, such that $\bigcup_{S \in s(G)} S = V$, i.e., every agent is in at least one permitted selection. The *fairness constraint* f maps G to a subset of $s(G)^{\mathbb{N}}$ of *fair* schedules. For a distributed machine M and a scheduler Σ, a run of M on a labelled graph G is called *fair* iff it is induced by a fair schedule.

A pair $A = (M, \Sigma)$ consisting of a distributed machine M and a scheduler Σ satisfies the *consistency condition* on a class of graphs $\mathcal{C}$ iff for every given graph $G \in \mathcal{C}$, either all fair runs of M on G are accepting or all are rejecting. Intuitively, whether M accepts $G \in \mathcal{C}$ does not depend on which fair run we consider. In this case, we call A a *$\mathcal{C}$-distributed automaton* and say that A *accepts* G or A *rejects* G, respectively. $L(A) \subseteq \mathcal{C}$ denotes the language of Λ-labelled graphs accepted by A. A *distributed automaton* is a $\mathcal{G}$-distributed automaton, where $\mathcal{G}$ denotes the set of all graphs.

2.3 Classes of Distributed Automata

We formalize the parameters *detection*, *acceptance*, *selection* and *fairness* mentioned in the introduction. The first two concern the distributed machine $M = (Q, \delta_0, \delta, Y, N)$, while the other two concern the scheduler $\Sigma = (s, f)$. Let $G = (V, E, \lambda)$ be any Λ-labelled graph.

Detection. A distributed machine with counting bound $\beta = 1$ has *existence detection* (`d`), while a machine with $\beta > 1$ has *counting detection* (`D`).

Acceptance. A distributed machine with *halting acceptance* (`a`) fulfils $\delta(q, \mathcal{N}) = q$ for all $q \in Y \cup N, \mathcal{N} \in [\beta]^Q$. Otherwise, it accepts by *stable consensus* (`A`).

Selection. A *synchronous* (`$`) scheduler fulfils $s(G) = \{V\}$, i.e., the only permitted selection is all of V. The other extreme, a *liberal* (`s`) scheduler, is characterized by $s(G) = \mathcal{P}(V)$, i.e., any selection is permitted. Lastly, the scheduler is *exclusive* (`S`) iff $s(G) = \{\{v\}\}_{v \in V}$, i.e., in every step, exactly one agent is selected.

Fairness. For a *weakly fair* (`f`) scheduler, $f(G)$ contains all schedules in $s(G)^{\mathbb{N}}$. (Recall that a schedule, by definition, selects every agent infinitely often.) A schedule $(S_i)_{i \in \mathbb{N}}$ is strongly fair iff for every finite sequence of permitted selections $(T_i)_{i \in [n]} \in s(G)^*$, there are infinitely many $j \in \mathbb{N}$, such that $(S_{j+i})_{i \in [n]} = (T_i)_{i \in [n]}$. For a *strongly fair* (`F`) scheduler, $f(G) \subseteq s(G)^{\mathbb{N}}$ contains exactly the strongly fair schedules.

The expressive power of all possible combinations of these parameters is analysed in [8]. As mentioned in the introduction, after merging classes with the same expressive power one obtains the hierarchy shown in Figure 1. For the purpose of studying the emptiness problem, we are interested in the six classes with non-trivial expressive power, as the problem is trivially decidable for the `daf` class.

Recall that xyz implicitly means $xy\mathtt{s}z$. However, [8] also proves the following result:

▶ **Proposition 1** ([8, Lemma 3 (5) and Theorem 5]). *For all $xy \in \{\mathtt{d}, \mathtt{D}\}\{\mathtt{a}, \mathtt{A}\}$, the classes $xy\mathtt{f}$ and $xy\mathtt{\$f}$ have the same expressive power. Moreover, given a distributed automaton in one of the classes, one can effectively construct an equivalent automaton in the other class.*

Therefore, in order to prove the undecidability of the emptiness problem for `DAf`, `dAf`, and `Daf`, it suffices to prove it for `DA$f`, `dA$f`, and `Da$f`. We will make use of this.

3 Simulating Turing Machines with Distributed Automata

The emptiness problem for xyz-distributed automata, where $xyz \in \{\mathtt{d},\mathtt{D}\}\{\mathtt{a},\mathtt{A}\}\{\mathtt{f},\mathtt{F}\}$, consists of deciding whether a given xyz-distributed automaton A satisfies $L(A) = \emptyset$. We prove undecidability of the emptiness problem for $xy \neq \mathtt{da}$ by reduction from the halting problem for Turing machines on blank tape. Formally, we show:

▶ **Theorem 2** (Undecidability of the Emptiness Problem for Distributed Automata). *Let $xyz \in \{\mathtt{d},\mathtt{D}\}\{\mathtt{a},\mathtt{A}\}\{\mathtt{f},\mathtt{F}\}$ with $xy \neq \mathtt{da}$. Given a Turing machine T, one can effectively construct an xyz-distributed automaton A^T, such that $L(A^T) \neq \emptyset$ iff T halts on blank tape. The emptiness problem for xyz-distributed automata is thus undecidable.*

The proof of the theorem uses the same idea for every class xyz, which we proceed to describe. First, we observe that distributed automata are closed under intersection:

▶ **Lemma 3** (Closure under Intersection). *Let $xywz \in \{\mathtt{d},\mathtt{D}\}\{\mathtt{a},\mathtt{A}\}\{\mathtt{\$},\mathtt{s},\mathtt{S}\}\{\mathtt{f},\mathtt{F}\}$. Given two $xywz$-distributed automata A_1, A_2 operating on the same labelling alphabet Λ, we can effectively construct an $xywz$-distributed automaton A, such that $L(A) = L(A_1) \cap L(A_2)$. Moreover, this result remains valid even when A_2 is only an $L(A_1)$-distributed automaton.*

Note that we should not omit the selection parameter $w \in \{\mathtt{\$},\mathtt{s},\mathtt{S}\}$ here, as the results of [8] (which justified omitting the selection parameter elsewhere) do not necessarily hold for $L(A_1)$-distributed automata.

Proof sketch. Detailed proof in Appendix A of the full version. We outline the proof of the first claim. The construction is very similar to the product construction known from DFAs. Intuitively, A executes A_1, A_2 in parallel, transitioning between states in $Q_1 \times Q_2$ according to $\delta((q_1,q_2),\mathcal{N})) = (\delta_1(q_1,\mathcal{N}),\delta_2(q_2,\mathcal{N}))$. Further, $(q_1,q_2) \in Y$ iff $q_1 \in Y_1$ and $q_2 \in Y_2$, and $(q_1,q_2) \in N$ iff $q_1 \in N_1$ or $q_2 \in N_2$. ◁

Given a Turing machine T, the proof of Theorem 2 consists of exhibiting a family of labelled graphs $\mathcal{L} = \{G_n\}_{n\geq 1}$ together with a distributed automaton A^L and an $\mathcal{L}$-distributed automaton A^H satisfying (1) A^L is an xyz-distributed automaton with $L(A^L) = \mathcal{L}$, and (2) A^H, for a graph $G_n \in \mathcal{L}$, accepts iff T halts after visiting at most n tape cells, and rejects otherwise; intuitively, a run of A^H on G_n simulates the execution of T as long as the TM head never exceeds the first n tape cells.

Notice that this implies the existence of the distributed automaton A^T of Theorem 2. For this, let A^T be an automaton deciding the language $L(A^L) \cap L(A^H)$, which exists by Lemma 3. We show that $L(A^T) \neq \emptyset$ iff T halts: If $L(A^T) \neq \emptyset$, let $G \in L(A^L) \cap L(A^H)$. By (1), $G = G_n$ for some $n \geq 1$. By (2), T halts visiting at most n tape cells. In particular, T halts. Conversely, if T halts, then it does so after a finite number of steps, visiting a finite number n of tape cells. So by (1) and (2) both A^L and A^H accept G_n, yielding $G_n \in L(A^L) \cap L(A^H)$. In particular, $L(A^T) \neq \emptyset$.

We will construct A^L and A^H for the classes $xy\mathtt{\$f}$ with $xy \in \{\mathtt{DA},\mathtt{dA},\mathtt{Da}\}$. By Proposition 1, this implies that we can effectively construct an equivalent automaton for each class $xy\mathtt{f}$, and by the hierarchy of expressive power this can be lifted to the respective $xy\mathtt{F}$ class as well. This is thus sufficient to prove Theorem 2 for all classes[3].

[3] In fact, it would even suffice to only construct A^L and A^H for the classes $xy\mathtt{\$f}$ with $xy \in \{\mathtt{dA},\mathtt{Da}\}$, omitting $xy = \mathtt{DA}$. Including the construction for $xy = \mathtt{DA}$ was a didactic choice, as it lays the groundwork and helps motivate the other two constructions.

The rest of the section is structured as follows. We first consider the class DA$f, for which Subsection 3.1 defines a suitable family $\mathcal{L}$ and the automaton A^L, and Subsection 3.2 defines the automaton A^H. In Subsection 3.3, we find a slightly more complex family and automaton for the dA$f class.

3.1 Representing the Tape

The simplest possible representation of a finite TM tape section is a linear graph where each node represents a tape cell and the labelling gives rise to a notion of positive/right and negative/left directions. A labelling achieving this is numbering the nodes modulo 3, starting with 0; an agent with numbering i can identify its left and right neighbour as the unique neighbour with numbering $(i-1) \bmod 3$ and $(i+1) \bmod 3$, respectively. We call this family of graphs *numbered linear graphs* (NLGs) and denote it by NLG. Figure 2 shows an example for an NLG. The *origin node* of an NLG is the unique node that has numbering 0 and no left neighbour, i.e., no neighbour numbered 2.

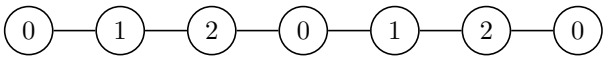

Figure 2 A numbered linear graph of length 7.

▶ **Lemma 4** (Decidability of Numbered Linear Graphs). *We can effectively construct a* DA$f*-distributed automaton that decides* NLG.

Proof sketch. Detailed proof in Appendix A of the full version. We sketch the relevant construction. The labelling alphabet is $\Lambda^L = \mathbb{Z}_3$ and the agents have states in $Q^L = (\Lambda^L \times \{0,1\}) \cup \{\bot\}$. The first component of the state is the numbering, which is taken directly from the labelling and stays static, and the second component is the agent's current guess whether the graph has an origin node. $\bot$ is an error state that, once it occurs, propagates through the graph, that is, if any neighbour of an agent v is in state $\bot$, v transitions to $\bot$ as well. The only accepting states are those with guess 1; all other states, most notably $\bot$, are rejecting.

If an agent v has the same numbering as one of its neighbours, it transitions to the error state $\bot$ immediately; the same happens if v detects two of its neighbours having the same numbering as each other. A graph with faulty numbering or nodes of degree greater than 2 will therefore always produce an error.

Lastly, we have to distinguish numbered linear graphs from circular graphs with correct numbering. This is where the guess component comes in: If an origin node exists, it is the first to change its guess to 1, which then propagates through the graph to accept. If no origin node exists, no agent will be the first to guess 1 and thus all agents will stay in rejecting states indefinitely. ◁

3.2 Simulating the Head

Given an NLG of length $n \geq 1$ to represent a finite TM tape section, where each agent represents a tape cell, we can simulate the behaviour of the TM head by augmenting the states. Every agent carries a symbol of the tape alphabet in an additional component. Further, exactly one agent indicates that the TM head is currently on its corresponding tape cell and saves the TM state. When the TM head performs a transition, it changes the agent's tape symbol and the TM state and indicates in which direction it will move next. The agent in the respective direction detects this and takes over as TM head. This clearly mimics the behaviour of an actual Turing machine on a tape of finite length n.

Recall that in our model, the TM tape is only unbounded to the right and the TM head starts on the leftmost cell. So, the simulation starts with all agents carrying a blank symbol and the TM head on the unique origin node's tape cell. The TM head state will move through the NLG until the TM halts, producing an accepting state, or the TM runs out of space in the finite TM tape section, producing a rejecting state. The accepting (rejecting) state then propagates through the graph to accept (reject).

For any given Turing machine T, we can construct an equivalent Turing machine T_∞ that either halts or visits every tape cell; in particular, it can never happen that the TM gets stuck in a loop. By simulating T_∞ rather than T itself, we can be sure that the simulation always either accepts by halting or rejects by running out of space.

Carrying all of this out formally in Appendix A of the full version yields the following result.

▶ **Lemma 5** (Simulating the Head). *Given a Turing machine T, we can effectively construct an* NLG*-distributed automaton A^H in the class* `da$f` *such that:*

- *If T does not halt on blank tape, then A^H rejects all numbered linear graphs.*
- *If T halts on blank tape, then there exists a threshold $n_0 \in \mathbb{N}$, such that A^H accepts all numbered linear graphs of length $n \geq n_0$ and rejects all numbered linear graphs of length $n < n_0$.*

The threshold n_0 will turn out to be the finite number of tape cells that T_∞ visits before it halts. We construct A^H in the trivial class `da$f`, as this enables us to lift the construction to any of the non-trivial classes of distributed automata by the hierarchy of expressive power[4]. By doing so for the class `DA$f`, we now have all the puzzle pieces to execute the proof of Theorem 2 for `DA$f`-distributed automata, as outlined at the beginning of Section 3. The details can be found in Appendix A of the full version.

3.3 Weakly Representing the Tape

It is easy to see that `dA$f`-automata cannot decide NLG. A formal proof can be given by adapting [6, Proposition D.1], but we sketch the argument.

Take an NLG of length at least 3 and duplicate its origin node v_0 into two nodes v_{01} and v_{02}, which become identical left neighbours of the second node v_1. The new graph is not a linear graph and should be rejected. Let A be a `dA$f`-distributed automaton. The agents at v_{01} and v_{02} start in the same state and both have v_1 as their only neighbour. As A selects all agents in every step (`$`), v_{01} and v_{02} will always have the same state as each other. Since agents of A cannot count (`d`), v_1 will thus never be able to detect that it has two left neighbours.

We exhibit a new family of suitable graphs for `dA$f`-automata, namely *numbered quasi-linear graphs* (NQLGs). Intuitively, an NQLG is obtained similarly to the counterexample above: take an NLG and replicate each node an arbitrary number of times where every replica of a node has at least one predecessor (successor) iff the original node had a predecessor (successor). Figure 3 shows an example of an NQLG with two replicas of the first node, three of the second node, etc. The formal definition takes a bit of a different angle on the same family of graphs:

[4] Note that Lemma 5 does not constitute a contradiction to `da$f`-distributed automata having trivial expressive power, as A^H is only an NLG-distributed automaton, not a distributed automaton.

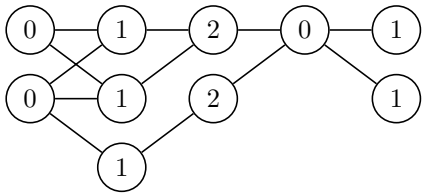

Figure 3 A numbered quasi-linear graph of length 5.

▶ **Definition 6** (Numbered Quasi-Linear Graphs). Let $G = (V, E, \lambda)$ with $\lambda \colon V \to \mathbb{Z}_3$. Further, let $O = \{v \in V \mid \lambda(v) = 0, \lambda(w) \neq 2 \text{ for all neighbours } w \text{ of } v\}$. We call G a *numbered quasi-linear graph* of length $n \in \mathbb{N} \setminus \{0\}$ iff O is non-empty and
(QL1) $\forall v \in V \colon \lambda(v) = \operatorname{dist}(O, v) \bmod 3$,
(QL2) $\forall \{v, w\} \in E \colon \lambda(v) \neq \lambda(w)$, and
(QL3) for all $v \in V \colon \big(\exists \{v, w\} \in E \colon \operatorname{dist}(O, w) = \operatorname{dist}(O, v) + 1\big) \iff \operatorname{dist}(O, v) < n - 1$.
The set O is called the *origin set*; the elements of O are called *origin nodes*. We denote the family of numbered quasi-linear graphs by NQLG.

Observe that every NLG is an NQLG, i.e., NLG $\subseteq$ NQLG. Notice also that (QL3) requires every node with a distance to the origin strictly smaller than $n - 1$ to have a successor and prohibits nodes with distance $n - 1$ to do so. This enables us to unambiguously assign a length to an NQLG.

In the rest of the section, we show that (1) dA$f-distributed automata can decide NQLG (the counterpart to Lemma 4), and (2) the TM head simulation A^H of Lemma 5 works on NQLGs as well. We start with part (1):

▶ **Lemma 7** (Decidability of Numbered Quasi-Linear Graphs). *We can effectively construct a* dA$f*-distributed automaton that decides* NQLG.

Proof sketch. Detailed proof in Appendix A of the full version. We sketch the necessary construction. It builds upon the one in Lemma 4, using the same labelling alphabet Λ^L and states in $Q^L = (\Lambda^L \times \{0, 1, 2\}) \cup \{\bot\}$. The first component of the state and the $\bot$ state work exactly as in Lemma 4. The second component indicates one of three stages: 0 – the initial stage, 1 – origin set detected, or 2 – origin set and end of graph detected. The only accepting states are those in stage 2, all other states are rejecting.

If the given graph is an NQLG, first, all agents in the origin set O transition to stage 1 at the same time. In the d^{th} subsequent step, all agents at distance d from the origin set O transition to stage 1 simultaneously, until finally the agents at distance $n - 1$ reach stage 1, which are exactly the agents that do not have a successor. These agents at the end of the graph then continue to stage 2. Again, in the d^{th} step after that, all nodes at distance $n - 1 - d$ reach stage 2 simultaneously, ultimately accepting the graph.

If, however, the graph is not an NQLG, the automaton can always detect this. As for circular graphs in Lemma 4, if O is empty, all agents will passively reject by never leaving stage 0. If (QL2) is violated, this will be detected and rejected immediately. A violation of (QL1) breaks simultaneity during the stage 1 transitions, while a violation of (QL3) breaks simultaneity during the stage 2 transitions. In the full proof, we show that agents that can detect these faults exist in any non-NQLG. ◁

For part (2), we show that for any dA$f-distributed automaton A, running on an NLG $G \in$ NLG and running on some NQLG $\tilde{G} \in$ NQLG of the same length are equivalent. In particular, this holds if we choose A to be equivalent to A^H from Lemma 5, which is possible

by the hierarchy of expressive power. Consider the runs of A on G and $\tilde{G}$. As in the intuitive description of NQLGs, $\tilde{G}$ can be seen as arising from G by replicating its nodes. Since agents of A cannot count (d), they can only distinguish between no predecessor (successor) and at least one predecessor (successor). It follows that, in the runs of A on G and $\tilde{G}$, an agent at a node v in G and all of its replicas in $\tilde{G}$ visit exactly the same sequence of states. The next proposition formalizes this. The proof can be found in Appendix A of the full version.

▶ **Proposition 8** (Correspondence of NQLGs and NLGs). *Let $\tilde{G} = (\tilde{V}, \tilde{E}, \tilde{\lambda}) \in$ NQLG be a numbered quasi-line graph with origin set O and length $n \in \mathbb{N} \setminus \{0\}$, and let $G = (V = \{v_0, \dots, v_{n-1}\}, E, \lambda) \in$ NLG be the numbered line graph of the same length n. Further, let M be a dA$f-distributed machine, and let $\tilde{\rho} = (\tilde{C}_i)_{i\in\mathbb{N}}$ and $\rho = (C_i)_{i\in\mathbb{N}}$ be the (unique) fair runs of M on $\tilde{G}$ and G, respectively. Then, for every $i \in \mathbb{N}$ and every $\tilde{v} \in \tilde{V}$, we have $\tilde{C}_i(\tilde{v}) = C_i(v_{\mathrm{dist}(O,\tilde{v})})$.*

4 Snowball Fight!

We turn our attention to the class Da$f. Like dA$f-automata, these cannot decide NLG either, but the reason is more fundamental: Distributed automata with halting acceptance (a) cannot distinguish between a circular graph with correct numbering modulo 3 (we will call this a *numbered circular graph* (NCG)), and a sufficiently long NLG. This can be proven analogously to [8, Proposition 12], as we will now sketch.

Let $A = (M, \Sigma)$ be a distributed automaton with halting acceptance (a) that rejects an NCG Z. By the consistency condition, any fair run of M on Z is rejecting. Let $\rho = (C_i)_{i\in\mathbb{N}}$ be such a fair run, induced by a fair schedule σ, and $I \in \mathbb{N}$ such that C_I is rejecting. We construct an NLG Z' by deleting an edge from Z between a pair of nodes with numbering 2 and 0, and then concatenate $2I + 1$ copies of Z' to obtain the NLG L. Consider the run of M on L induced by a schedule σ' that replicates the first I selections of σ on all copies of Z' (note that this can easily be turned into a fair schedule as we only fix finitely many selections). It is easy to see that, for an agent at distance k from the two endpoints of L, at least the first k transitions are exactly those that the agent at the corresponding original node in Z undergoes. The agents of L in the middle copy of Z' will therefore reach rejecting states after at most I transitions and halt due to halting acceptance (a). By the consistency condition, A thus has to reject the NLG L too.

This shows that an automaton with halting acceptance (a) that accepts all NLGs necessarily also accepts NCGs. However, NCGs lack an origin node, which is essential for our TM simulation as that is where the TM head starts. Without an origin node, there is no simulated TM head, breaking the simulation[5].

To solve this problem, we observe that for our purpose of representing arbitrarily long TM tape sections, we do not need to accept all NLGs; rather, it suffices to accept infinitely many NLGs. This guarantees that for every TM that halts (after finitely many steps), we accept an NLG that is long enough to simulate this run correctly. We will thus construct a Da$f-automaton that accepts infinitely many, but not all, NLGs. For this purpose, we introduce a new family of graphs, which can be seen as a subfamily of NLGs. In the case of NLGs and

[5] Requiring that initially exactly one node has a special label, say h, modelling the head does not work either. Again, analogously to [8, Proposition 12], one can show that an automaton with halting acceptance (a) cannot accept all graphs with exactly one h-labelled node and reject all graphs with none.

NQLGs in Sections 3.1 and 3.3, we first defined the families, and then constructed automata recognizing them. In this section, we proceed differently: we construct the automaton first, thereby implicitly defining a family of graphs as the automaton's accepted language.

Intuitively, the automaton lets the agents engage in a *snowball fight*! For this, we augment the labelling of all agents with a direction, positive or negative, modelling the direction the agent is facing at the beginning of the fight. Additionally, every other agent initially holds a snowball. The labelling alphabet hence is $\Lambda^L = \mathbb{Z}_3 \times \{-1, +1\} \times \{0, 1\}$: the first component is the numbering, the second is the direction, and the third indicates whether the agent starts with a snowball. A *snowball fight NLG* (SFNLG) is a Λ^L-labelled that becomes an NLG after projecting all labels onto their first component; SFNCGs are defined analogously. We denote the family of SFNLGs by SFNLG. For example, the third graph of Figure 5 is an SFNLG of length 7. The agents carry out a snowball fight following a fixed set of rules. Agents can throw snowballs, and catch or get hit by snowballs thrown at them. Throughout the course of the fight, the number of snowballs will decrease. The graph is rejected if an agent gets hit, and is accepted if this does not happen until eventually no snowballs are left. We will show that the former happens for all SFNCGs, while the latter happens in infinitely many SFNLGs. More details follow below. The formal description can be found in Appendix B of the full version.

▶ **Construction 9** (Snowball Fight! (sketch))**.** We construct a Da$f-distributed automaton with the labelling alphabet Λ^L as described above. The set of states comprises Λ^L and some auxiliary states: the accepting state ☑, the rejecting error state ⊥, and a state ☐ to indicate the intention to accept.

We give an informal description of the behaviour of the automaton. First, the automaton uses counting detection (D) as in Lemma 4 to check that all nodes have at most degree 2 and the numbering is correct, producing a ⊥ state if the check fails. This already guarantees that the automaton can only accept SFNLGs or SFNCGs. Further, each agent ensures that either itself or both of its neighbours are starting with a snowball. Then, the automaton performs the snowball fight, following four simple rules (and some special rules for the agents at the endpoints of an SFNLG, which we will state separately):

(0) If an agent v is holding a snowball, v throws the snowball in the direction it is facing.

(1) If exactly one snowball is thrown at an agent v and v faces towards it, then v catches the snowball and turns around.

(2) If exactly one snowball is thrown at an agent v and v faces away from it, then v gets hit and transitions to the ⊥ state.

(3) If two snowballs are thrown at an agent v, then v catches both of them, merges them into one, and turns around.

The possible scenarios that can arise from these rules are illustrated in Figure 4. Lastly, the agents at the endpoints of SFNLGs can throw a snowball into the void (as they only have one neighbour). If this happens at the origin node, it indicates its intention to accept by a ☐ state; otherwise, this produces a ⊥ state. Intuitively, the ☐ state means that the agent wants to accept (and thus halt) but cannot do so yet, as it first has to ensure that no other agent has halted in the rejecting ⊥ state. For this, the ☐ state propagates until it either reaches the other end of the SFNLG and turns into an accepting ☑ state, or it is intercepted by a rejecting ⊥ state. Both the ☑ and the ⊥ state propagate unconditionally, ultimately causing acceptance or rejection, respectively.

We show that Construction 9 only accepts SFNLGs, and accepts infinitely many SFNLGs, which is sufficient for our purpose. We split this up into three lemmas:

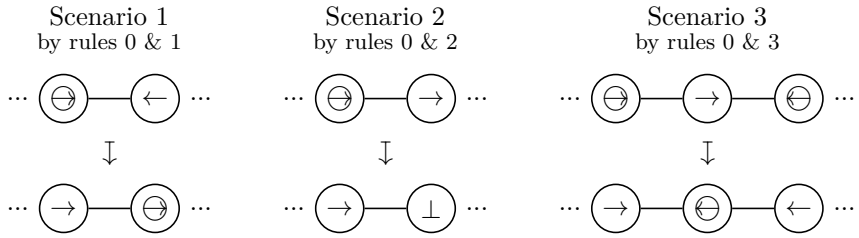

Figure 4 The three scenarios arising from rules 0 to 3 – The arrow in each node indicates the direction the agent is facing. A circle indicates that the agent is holding a snowball. Note that scenario 2 assumes that we are not in scenario 3, otherwise rule 2 would not apply.

(1) If at least one rejecting $\perp$ state occurs, the graph is rejected.
(2) If no $\perp$ state occurs, the graph is in SFNLG and it is accepted.
(3) There are infinitely many SFNLGs that get accepted.
These lemmas are proved as Lemmas 21, 22 and 24, respectively, in Appendix B of the full version. We sketch the proof ideas for the first two and present the relevant graph construction for the third.

Proof sketch of (1). We have to show that after an error state occurred, no accepting ☑ state can arise any more. Notice that for a ☑ state to be produced, the ☐ state has to propagate from the origin node to the other end of the graph. Further notice that after an agent takes on the ☐ state, it cannot be the first to produce an $\perp$ state, it can only adopt a $\perp$ state from neighbouring agents. So, the first time that an error occurs has happen at an agent that has not yet indicated its intention to accept by a ☐ state. In that case however, the $\perp$ state would intercept the propagation of the ☐ state, preventing it from reaching the other end of the graph to produce an accepting ☑ state. ◁

Proof sketch of (2). For no error to occur the numbering has to be correct, so the given graph can only be an SFNLG or SFNCG. Intuition suggests that on a finite graph, the snowballs should eventually meet and get merged, ultimately decreasing their number to one. This is indeed correct. In an SFNCG, this last snowball inevitably hits an agent, producing an error by rule 2 (at the latest after doing a whole round of the circular graph and turning all agents to face the same direction). As we assumed no errors to occur, the given graph has to be in SFNLG. The last snowball eventually reaches the origin node, producing a ☐ state, which propagates through the graph unhindered and produces an accepting ☑ state. ◁

We conclude that Construction 9 accepts or rejects every graph and is therefore indeed a distributed automaton. Moreover, any accepted graph has to be in SFNLG. Lastly, we construct an infinite (non-exhaustive) family of accepted SFNLGs.

Construction for (3). We sketch the iterative construction of the direction and snowball labelling (see Figure 5): We start with a single agent facing left and holding a snowball. Clearly, the last (and only) snowball will get thrown into the void to the left. To construct the $(n+1)^{\text{th}}$ iteration of the construction, we take two copies of the n^{th} construction, mirror one of them and connect both to a new right-facing agent such that the last two snowballs, one from each copy of the n^{th} construction, meet and merge at this new middle agent. From there, the one remaining snowball will be passed to the left until it gets thrown into the void to the left.

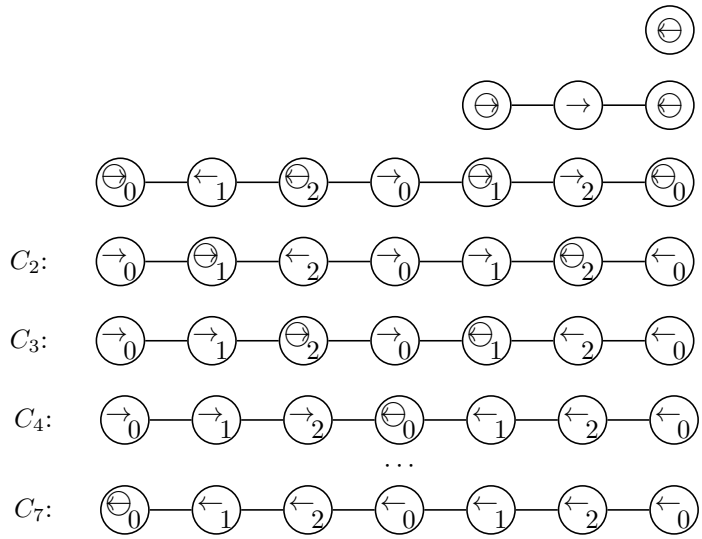

Figure 5 Iterative construction of the labelling of an accepted SFNLG of length 7 and its accepting run – The first three rows show the construction of the SFNLG; the other rows show how the snowball fight is performed without errors.

By numbering the nodes such that the leftmost node is the origin node, this ultimately causes acceptance for any iteration of the construction. The n^{th} construction has length $2^n - 1$, yielding infinitely many accepted SFNLGs. ◁

5 Conclusion

We have initiated the study of verification problems for the classes of distributed automata introduced in [8]. We have shown that the emptiness problem, a fundamental verification problem, is undecidable or trivially decidable for all classes of [8]. Since distributed automata are closed under intersection (Lemma 3), this means that the safety problem – given an automaton and a class $\mathcal{D}$ of "dangerous" graphs, does the automaton accept some graph of $\mathcal{D}$? – is undecidable whenever $\mathcal{D}$ is recognized by some automaton.

Our undecidability proofs simulate the execution of a Turing machine on blank tape by means of a distributed automaton running on families of labelled graphs (NLG, NQLG, and a subfamily of SFNLG) having both a specific structure and a specific labelling. The proofs break down if we restrict ourselves to automata that only allow unlabelled graphs as inputs, so they can only decide purely structural properties of the input graph, or to automata that only decide labelling properties. The decision power of the latter was studied by Czerner *et al.* in [6]. In future work, we plan to study these two special cases. We conjecture that the emptiness problem for automata deciding structural properties remains undecidable, but becomes decidable for some of the classes of [6].

References

1 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. *Distributed Comput.*, 26(4):195–208, 2013. doi:10.1007/S00446-012-0175-7.

2 Dana Angluin. Local and global properties in networks of processors (extended abstract). In Raymond E. Miller, Seymour Ginsburg, Walter A. Burkhard, and Richard J. Lipton, editors,

Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA, pages 82–93. ACM, 1980. doi:10.1145/800141.804655.

3 Dana Angluin, James Aspnes, Melody Chan, Michael J. Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In Viktor K. Prasanna, S. Sitharama Iyengar, Paul G. Spirakis, and Matt Welsh, editors, *Distributed Computing in Sensor Systems, First IEEE International Conference, DCOSS 2005, Marina del Rey, CA, USA, June 30 - July 1, 2005, Proceedings*, volume 3560 of *Lecture Notes in Computer Science*, pages 63–74. Springer, 2005. doi:10.1007/11502593_8.

4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006. doi:10.1007/S00446-005-0138-3.

5 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010. doi:10.1007/978-3-642-15763-9_15.

6 Philipp Czerner, Roland Guttenberg, Martin Helfrich, and Javier Esparza. Decision power of weak asynchronous models of distributed computing. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 115–125. ACM, 2021. doi:10.1145/3465084.3467918.

7 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In Panagiota Fatourou and Gadi Taubenfeld, editors, *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 137–146. ACM, 2013. doi:10.1145/2484239.2484244.

8 Javier Esparza and Fabian Reiter. A classification of weak asynchronous models of distributed computing. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CONCUR.2020.10.

9 Ofer Feinerman and Amos Korman. Theoretical distributed computing meets biology: A review. In Chittaranjan Hota and Pradip K. Srimani, editors, *Distributed Computing and Internet Technology, 9th International Conference, ICDCIT 2013, Bhubaneswar, India, February 5-8, 2013. Proceedings*, volume 7753 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013. doi:10.1007/978-3-642-36071-8_1.

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. *Distributed Comput.*, 28(1):31–53, 2015. doi:10.1007/S00446-013-0202-3.

11 Antti Kuusisto and Fabian Reiter. Emptiness problems for distributed automata. *Inf. Comput.*, 272:104503, 2020. doi:10.1016/J.IC.2019.104503.

12 Saket Navlakha and Ziv Bar-Joseph. Distributed information processing in biological and computational systems. *Commun. ACM*, 58(1):94–102, 2015. doi:10.1145/2678280.

13 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Nat. Comput.*, 7(4):615–633, 2008. doi:10.1007/S11047-008-9067-Y.