A Classification of Weak Asynchronous Models of Distributed Computing

Javier Esparza
Technical University of Munich, Germany
esparza@in.tum.de

Fabian Reiter
LIGM, Université Gustave Eiffel, Marne-la-Vallée, France
fabian.reiter@gmail.com

Abstract

We conduct a systematic study of asynchronous models of distributed computing consisting of identical finite-state devices that cooperate in a network to decide if the network satisfies a given graph-theoretical property. Models discussed in the literature differ in the detection capabilities of the agents residing at the nodes of the network (detecting the set of states of their neighbors, or counting the number of neighbors in each state), the notion of acceptance (acceptance by halting in a particular configuration, or by stable consensus), the notion of step (synchronous move, interleaving, or arbitrary timing), and the fairness assumptions (non-starving, or stochastic-like). We study the expressive power of the combinations of these features, and show that the initially twenty possible combinations fit into seven equivalence classes. The classification is the consequence of several equi-expressivity results with a clear interpretation. In particular, we show that acceptance by halting configuration only has non-trivial expressive power if it is combined with counting, and that synchronous and interleaving models have the same power as those in which an arbitrary set of nodes can move at the same time. We also identify simple graph properties that distinguish the expressive power of the seven classes.

2012 ACM Subject Classification Theory of computation → Automata extensions; Theory of computation → Concurrency; Theory of computation → Distributed computing models

Keywords and phrases Asynchrony, Concurrency theory, Weak models of distributed computing

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2020.10

Related Version A full version with appendix is available at `https://arxiv.org/abs/2007.03291v1`.

Funding This work was supported by the ERC project PaVeS (Advanced Grant 787367).

Acknowledgements The authors thank Ahmed Bouajjani for many interesting discussions, and several anonymous reviewers for their helpful feedback.

1 Introduction

Distributed computing is increasingly interested in the study of networks of natural or artificial devices, like molecules, cells, microorganisms, or nano-robots. These devices have very limited computational and communication capabilities, and are indistinguishable. In particular, a device cannot recognize whether its current communication partner is the same as a past one. This stands in stark contrast to the devices of standard computer networks, which has motivated researchers to question the suitability of traditional distributed computing models for the study of these networks, and to propose new ones. Examples include population protocols [3, 1], chemical reaction networks [13], networked finite state machines [6], the weak models of distributed computing of [8], and the beeping model [5]. A survey discussing many of them, and more, can be found in [11].

All these models share several common features, introduced to capture the limitations of the devices [6]: the network can have an arbitrary topology; all nodes of the network have a finite number of states, independent of the size of the network or its topology; all nodes run the same protocol; and state changes only depend on the states of a bounded number of neighbors, again independent of the size of the network.

Unfortunately, despite this very substantial common ground, the models still differ in many aspects, which makes it hard to compare results across them, or decide which features are essential for a particular result. A study of the models allows one to identify four specific junctions at which they choose different paths:

- *Detection.* In some models, agents can only detect the *existence* of neighbors in a certain state [8]. In others, they can *count* their number, up to a fixed threshold [6, 8]. For example, in biological models, cells communicate by emitting special kinds of proteins, and detecting them; in some models the cells may detect the presence of the protein when its concentration exceeds a given threshold, while in others they are able to detect different concentration levels.
- *Acceptance.* Some models compute by *stable consensus*, which requires all nodes to eventually agree on the outcome of the computation (but the nodes do not need to *know* that consensus has been reached) [3, 1, 13], while others require the nodes to reach a consensus in a halting configuration [8]. Acceptance by stable consensus is computationally powerful, since it permits the algorithm designer to concentrate on ensuring that every bad input is eventually rejected; declaring all non-rejecting states accepting ensures that every good input is eventually accepted.
- *Selection.* In some models, at each step a scheduler chooses an arbitrary set of nodes to make a step [6, 12], while in others it is exactly one, or exactly one pair of neighboring nodes [3, 1, 13]. We call the latter *exclusive* or *interleaving* models. Intuitively, interleaving models are useful when it can be assumed that process steps are much faster than the time interval between them, while the former policy does not need this assumption. In addition, they help the algorithm designer, who can assume that agents act in mutual exclusion. (Examples where this is useful can be found in the proofs of Propositions 16 and 20.) Another common option for selection is the *synchronous* execution model [8], where all nodes are selected in each step. Again this can be helpful for designing algorithms, but it is incompatible with exclusive selection.
- *Fairness.* Some models use fairness assumptions designed to model or approximate stochastic behavior [3, 1, 13], while others choose minimal notions, like "all nodes make a step infinitely often", which only assume the absence of crash faults (see, e.g., [7, 9]). Stochastic-like assumptions are reasonable for biological or chemical models, but can be too strong for networks of artificial nodes, which may follow non-random execution policies. Stochastic models may be able to solve problems that cannot be solved with weaker fairness assumptions.

The goal of this paper is to explore the space of models spanned by the above parameters, and compare their computational power within a specific framework. For this we use *distributed automata*, a generic formalism for the description of finite-state distributed algorithms. Such an automaton consists of a set of rules that tell the nodes of a graph how to change their state depending on the states of their neighbors. Intuitively, the automaton describes an algorithm that allows the nodes of an input graph to decide, in a distributed way, whether the graph satisfies a given property. The computational power of a class of distributed automata is then given by the class of graph languages recognized by the automata in the class, or, in other words, by the graph properties that the class of automata can decide.

We start with twenty classes of distributed automata, and show that with respect to their computational power, they fall into seven different classes. This reduction is a consequence of two results presented in this paper: (1) acceptance by halting configuration only has non-trivial expressive power if it is combined with counting; (2) both interleaving and synchronous selection have the same power as liberal selection where arbitrarily many nodes can move at the same time (and therefore, one can design an automaton in an interleaving or synchronous model, which is less error prone, and then translate it to a liberal model). Some of the simulations we design to prove the results are of independent interest. In particular, we give explicit constructions showing how to simulate interleaving models by non-interleaving ones.

The paper is organized as follows. Section 2 introduces distributed automata and their variants. Sections 3 to 5 show that the variants collapse to at most the seven equivalence classes mentioned above. Section 6 contains separation results showing that the seven classes are different. Finally, Section 7 presents further results on their expressive power. Proofs missing or only sketched can be found in the appendix of the arXiv version.

2 A taxonomy of distributed automata

Given sets X, Y, we denote by 2^X the power set of X, and by X^Y the set of functions $Y \to X$. We define $[m:n] := \{i \in \mathbb{Z} \mid m \leq i \leq n\}$ and $[n] := [0:n]$, for any $m, n \in \mathbb{Z}$ such that $m \leq n$. Angle brackets indicate excluded endpoints, e.g., $\langle m:n] := [m-1:n]$ and $[n\rangle := [0:n-1]$.

Let Λ be a finite set. A *(Λ-labeled, undirected) graph* is a triple $G = (V, E, \lambda)$, where V is a finite nonempty set of *nodes*, E is a set of undirected *edges* of the form $e = \{u, v\} \subseteq V$ such that $u \neq v$, and $\lambda\colon V \to \Lambda$ is a *labeling*. Isomorphic graphs are considered to be equal. **Convention**: Throughout the paper, all graphs have at least two nodes and are connected.

2.1 Distributed automata

Distributed automata take a graph as input, and either accept or reject it. To define them we first introduce distributed machines.

Distributed machines. Let Λ be a finite set of symbols and let $\beta \in \mathbb{N}_+$. A *(distributed) machine* with *input alphabet* Λ and *counting bound* β is a tuple $M = (Q, \delta_0, \delta, Y, N)$, where Q is a finite set of *states*, $\delta_0\colon \Lambda \to Q$ is an *initialization function*, $\delta\colon Q \times [\beta]^Q \to Q$ is a *transition function*, and $Y, N \subseteq Q$ are two sets of *accepting* and *rejecting* states, respectively. The function δ updates the state of a node v based on the number of neighbors v has in each state, but it can only detect if v has $0, 1, \ldots, (\beta - 1)$, or at least β neighbors in a given state.

Selections, schedules, configurations, runs, and acceptance. A *selection* of a Λ-labeled graph $G = (V, E, \lambda)$ is a set $S \subseteq V$, and a *schedule* of G is an infinite sequence of selections $\sigma = (S_0, S_1, S_2, \ldots) \in (2^V)^\omega$. Intuitively, the selection S_t is the set of nodes activated by the scheduler at time t.

Let $M = (Q, \delta_0, \delta, Y, N)$ be a distributed machine with input alphabet Λ. A *configuration* of M on G is a mapping $C\colon V \to Q$. Given a configuration C and a node $v \in V$, we let $N_v^C\colon Q \to [\beta]$ denote the function that assigns to each state q the number of neighbors of v that are in state q up to threshold β, i.e., $\min\{\beta, \operatorname{card}(\{u \mid \{u, v\} \in E \land C(u) = q\})\}$. We call N_v^C the *β-bounded multiset* of states of v's neighbors.

For any selection S, we define the *successor configuration* of C via S to be the configuration $succ_\delta(C,S)$ that one obtains from C if all nodes in S evaluate the transition function δ simultaneously while the remaining nodes keep their current state. Formally, for all $v \in V$,

$$succ_\delta(C,S)(v) = \begin{cases} C(v) & \text{if } v \notin S \\ \delta\big(C(v), N_v^C\big) & \text{if } v \in S. \end{cases}$$

This brings us directly to the notion of a run. Given a schedule $\sigma = (S_0, S_1, S_2, \ldots)$, the *run* of M on G scheduled by σ is the infinite sequence $\rho = (C_0, C_1, C_2, \dots)$ of configurations that are defined inductively as follows, where $\circ$ denotes function composition, and $t \in \mathbb{N}$:

$$C_0 = \delta_0 \circ \lambda \qquad \text{and} \qquad C_{t+1} = succ_\delta(C_t, S_t).$$

A configuration C is *accepting* if $C(v) \in Y$ for every $v \in V$, and *rejecting* if $C(v) \in N$ for every $v \in V$. A run $\rho = (C_0, C_1, C_2, \dots)$ of M on G is *accepting* if there is a time $t \in \mathbb{N}$ such that $C_{t'}$ is accepting for every $t' \geq t$. In other words, a run is accepting if from some time on it only visits accepting configurations. Similarly, ρ is *rejecting* if eventually all visited configurations are rejecting. Following [3], we call this *acceptance by stable consensus.*

Distributed automata. Not every schedule of a distributed machine models an execution; for example, schedules in which a node is never activated are usually considered illegal. We assume that distributed machines are controlled by a scheduler that ensures that the machine executes a legal run. Formally, a *scheduler* is a pair $\Sigma = (s, f)$, where s is a *selection constraint* that assigns to every graph $G = (V, E, \lambda)$ a set $s(G) \subseteq 2^V$ of *permitted selections* such that every node $v \in V$ occurs in at least one selection $S \in s(G)$, and f is a *fairness constraint* that assigns to every graph G a set $f(G) \subseteq s(G)^\omega$ of *fair schedules* of G. We call the runs with schedules in $f(G)$ *fair runs* (with respect to Σ).

A *distributed automaton* is a pair $A = (M, \Sigma)$, where M is a machine and Σ is a scheduler satisfying the *consistency condition*: for every graph G, either all fair runs of M on G are accepting, or all fair runs of M on G are rejecting. Intuitively, the machine is “immune” to the scheduler because its answer is independent of the scheduler’s choices. This formalizes the standard notion of “asynchronous distributed algorithm”. Notice that the consistency condition is a very strong *semantic* requirement. Although we will not do so in this paper, one can prove that it is undecidable whether a given pair (M, Σ) satisfies it.

A accepts G if every fair run of A on G is accepting, and *rejects G* otherwise. The language $L(A)$ recognized by A is the set of graphs it accepts. Two automata are equivalent if they recognize the same language.

2.2 Classifying distributed automata

We classify automata according to four criteria: detection capabilities, acceptance condition, selection constraint, and fairness constraint. The first two criteria concern the distributed machine, and the other two the scheduler. For each criterion, we investigate some of the major options that have been considered in the literature.

Detection. In some models, agents can only detect the existence of neighbors in a certain state. This corresponds to *non-counting machines*, i.e., machines with counting bound $\beta = 1$. Other models can detect the number of neighbors up to a higher bound [8].

Acceptance. As mentioned above, distributed machines accept by stable consensus. This is the acceptance condition of population protocols and chemical reaction networks [3, 1, 13]. Other models consider a notion of acceptance where each node explicitly decides to accept or reject [8]. This notion is captured by halting automata. A machine M is *halting* if its transition function does not allow the nodes to leave accepting or rejecting states, i.e., if $\delta(q, P) = q$ for every $q \in Y \cup N$ and every β-bounded multiset $P \in [\beta]^Q$. In halting machines, each node knows whether the input graph will be accepted the moment it enters an accepting or rejecting state. Indeed, by the consistency condition, in every fair run, eventually either all nodes occupy accepting states, or all nodes occupy rejecting states. Since nodes can never leave an accepting state once they enter it, each node that enters such a state knows that all other nodes will eventually do likewise. The same applies to rejecting states.

Selection. A scheduler $\Sigma = (s, f)$ is *synchronous* on $G = (V, E, \lambda)$ if $s(G) = \{V\}$. Intuitively, at every step all nodes make a move. Σ is *exclusive* or *interleaving-based* on G if $s(G) = \{\{v\} \mid v \in V\}$. Intuitively, at every step exactly one node makes a move, i.e., nodes execute steps in mutual exclusion. Finally, Σ is *liberal* on G if $s(G) = 2^V$. Intuitively, at every step an arbitrary subset of nodes makes a move. A scheduler is called *synchronous* if it is synchronous on every graph. Exclusive and liberal schedulers are defined analogously.

Fairness. A schedule $\sigma = (S_0, S_1, \ldots)$ of a graph G is *weakly fair* if for every node v of G, there exist infinitely many indices t such that $v \in S_t$. In other words, a schedule is weakly fair if every node is active infinitely often. A scheduler $\Sigma = (s, f)$ is *weakly fair* if $f(G)$ contains precisely the weakly-fair schedules of $s(G)^\omega$ for every graph G. This is the weakest fairness constraint one can impose on distributed automata; it only excludes runs in which a node crashes, and does not participate in the computation anymore.

With respect to a given selection constraint s, a schedule $\sigma = (S_0, S_1, \ldots) \in s(G)^\omega$ of a graph G is *strongly fair* if for every finite sequence $(T_0, \ldots, T_n) \in s(G)^*$ there exist infinitely many indices t such that $(S_t, S_{t+1}, \ldots, S_{t+n}) = (T_0, T_1, \ldots, T_n)$. Intuitively, strong fairness requires that every possible finite sequence of selections is scheduled infinitely often. If every node is selected independently with positive probability, stochastic schedules are almost surely strongly fair. A scheduler $\Sigma = (s, f)$ is *strongly fair* if for every graph G, the set $f(G)$ contains precisely the strongly-fair schedules of $s(G)^\omega$.

▶ Remark 1. Whether a schedule σ of a graph $G = (V, E, \lambda)$ is strongly fair or not depends on $s(G)$. For example, if $s(G) = \{V\}$, then the synchronous schedule V^ω is strongly fair, but if $s(G) = 2^V$, then it is not.

Our notion of strong fairness implies an apparently stronger one, used frequently in the literature, stating that in a strongly fair run, a sequence of configurations that is enabled infinitely often must occur infinitely often:

▶ **Lemma 2.** *Let A be a strongly fair automaton and $(D_0, \ldots, D_n)$ be a sequence of configurations of A such that D_{i+1} is the successor configuration of D_i via some selection S_i permitted by A, for $i \in [0:n\rangle$. For any fair run $\rho = (C_0, C_1, \ldots)$ of A, if $C_i = D_0$ for infinitely many indices $i \in \mathbb{N}$, then $(C_j, \ldots, C_{j+n}) = (D_0, \ldots, D_n)$ for infinitely many indices $j \in \mathbb{N}$.*

The classification above yields 24 classes of automata (four classes of machines and six classes of schedulers). To assign mnemonics to them, we use lowercase letters for the most restrictive machine variants (i.e., non-counting and halting), and the same letters in uppercase for the other variants. With schedulers we proceed the other way round, assigning lowercase letters to the most liberal variants (i.e., liberal selection and weak fairness). Intuitively, due

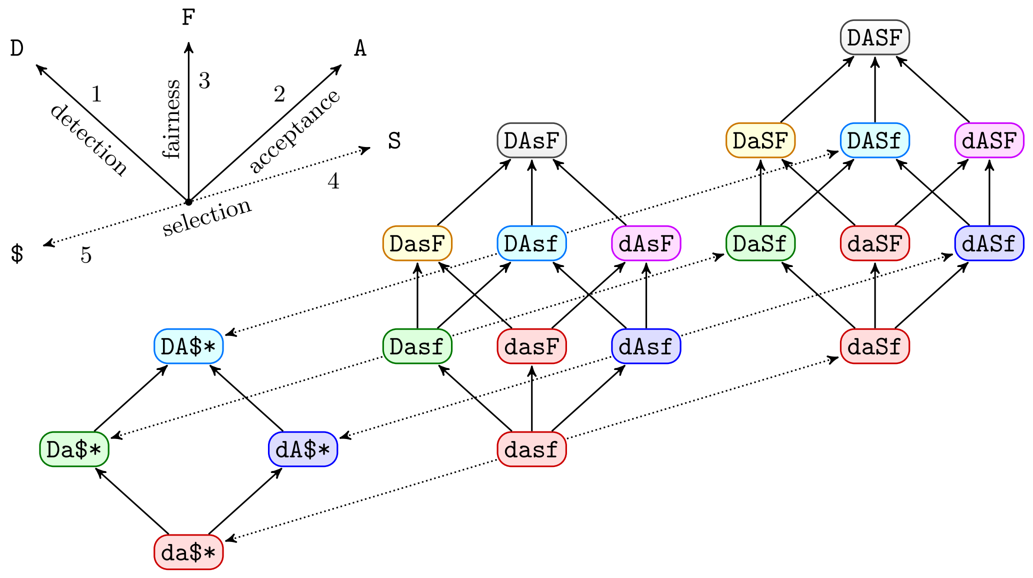

Figure 1 Initial classification of the models according to the class of graph languages they recognize. Arrows indicate inclusion between classes of languages. The diagram can be thought of as lying in four-dimensional space, where each dimension represents one of our four parameters. The vectors of the "coordinate system" are labeled with the statement number of Lemma 3 that proves the inclusions in the corresponding direction. In the coming sections, classes are shown to be equal if and only if they have the same color, reducing the 20 classes to 7, as shown in Figure 4. This means in particular that we completely eliminate the dimension of selection (shown in dotted lines), leaving us with only three dimensions.

to the consistency condition, the more liberal a scheduler, the harder it is for an automaton to recognize a graph language, because more runs have to yield the same result. So, loosely speaking, we expect the expressive power to increase with the number of uppercase letters.

Detection	*Acceptance*	*Selection*	*Fairness*
d: non-counting	a: halting	s: liberal	f: weak
D: counting	A: stable consensus	S: exclusive	F: strong
		$: synchronous	

We denote each class of automata by a string $wxyz \in \{\mathtt{d},\mathtt{D}\} \times \{\mathtt{a},\mathtt{A}\} \times \{\mathtt{s},\mathtt{S},\mathtt{\$}\} \times \{\mathtt{f},\mathtt{F}\}$. The class of languages recognized by $wxyz$-automata is denoted $\mathcal{G}(wxyz)$. The following lemma states all relations between language classes that follow directly from the definitions. Statement 1 abbreviates "$\mathcal{G}(\mathtt{d}xyz) \subseteq \mathcal{G}(\mathtt{D}xyz)$ for all $x \in \{\mathtt{a},\mathtt{A}\}$, $y \in \{\mathtt{s},\mathtt{S},\mathtt{\$}\}$, $z \in \{\mathtt{f},\mathtt{F}\}$". We use the same convention in Statements 2 to 5, and throughout the paper. That is, any statement with four-letter strings containing the wildcard symbol * must be expanded into the list of all statements that can be obtained by replacing identically positioned occurrences of * with the same letter.

▶ **Lemma 3.** ***1.*** $\mathcal{G}(\mathtt{d***}) \subseteq \mathcal{G}(\mathtt{D***})$, ***2.*** $\mathcal{G}(\mathtt{*a**}) \subseteq \mathcal{G}(\mathtt{*A**})$, ***3.*** $\mathcal{G}(\mathtt{***f}) \subseteq \mathcal{G}(\mathtt{***F})$, ***4.*** $\mathcal{G}(\mathtt{**sf}) \subseteq \mathcal{G}(\mathtt{**Sf})$, ***5.*** $\mathcal{G}(\mathtt{**sf}) \subseteq \mathcal{G}(\mathtt{**\$f})$, ***6.*** $\mathcal{G}(\mathtt{**\$F}) \subseteq \mathcal{G}(\mathtt{**\$f})$.

Lemma 3 leads to the diagram in Figure 1, showing 20 automata classes (we have $\mathcal{G}(\mathtt{**\$f}) = \mathcal{G}(\mathtt{**\$F})$ by Statements 3 and 6). An arrow between two classes means that every graph language recognized by the source class is also recognized by the target class.

The reader probably finds Figure 1 very complicated. We also do, and this was the motivation for the present paper. How many of these classes are really different? In the next sections we show that classes with the same color have the same expressivity, and thus that the diagram of Figure 1 collapses to the one of Figure 4, which contains only seven classes.

3 The weakest classes have no expressiveness

We prove that `das*`-automata have no expressive power, and the results in Sections 4 and 5 will generalize this to `da**`-automata. Intuitively, if agents cannot count their neighbors, and must reach a halting configuration, then they cannot distinguish any two graphs. Formally, a graph property is *trivial* if either every graph satisfies it, or no graph satisfies it. We have:

▶ **Theorem 4.** *Every* `das*`*-automaton recognizes a trivial graph property.*

Proof sketch. By Statement 3 of Lemma 3, it suffices to prove the claim for `dasF`-automata. So let A be a `dasF`-automaton, and let G and H be two graphs (connected and with at least two nodes by convention). Assume that A accepts G but rejects H. By the consistency condition, all fair runs of A on G accept, and all fair runs on H reject. Now let ρ^G and ρ^H be any such runs, and let $t \in \mathbb{N}$ be a time at which all nodes in ρ^G and ρ^H have halted. We define a new graph K that consists of t copies $\{G_i\}_{i\in[1:t]}$ and $\{H_i\}_{i\in[1:t]}$ of G and H, with additional edges defined as follows. For each node w^X of the original graph $X \in \{G, H\}$, we denote its copy in X_i by w_i^X, where $i \in [1:t]$. Let u^G and v^G be two *adjacent* nodes of G, and u^H and v^H be two *adjacent* nodes of H. We add the connecting edges $\{u_i^X, v_{i+1}^X\}$ for all $i \in [1:t\rangle$ and $X \in \{G, H\}$, as well as the edge $\{u_t^G, u_t^H\}$. This is illustrated in Figure 2.

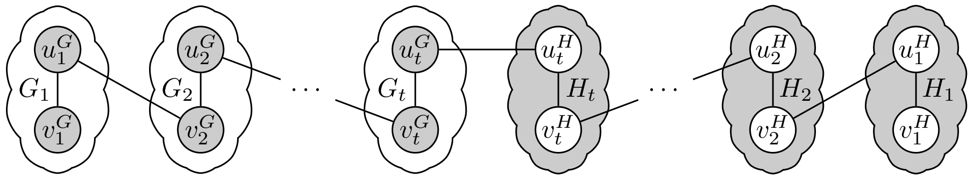

Figure 2 Graph K used in the proof of Theorem 4.

We show that there is a fair run ρ of A on K that neither accepts nor rejects. It follows that A does not satisfy the consistency condition, contradicting the hypothesis. Since A is a non-counting automaton, initially every node w_i^X except for u_t^G and u_t^H "sees" the same neighborhood as the corresponding node w^X in the original graph X. Only the two nodes u_t^G and u_t^H may have a different neighborhoods than u^G and u^H, and this might affect their behavior starting at time 1. Their different behavior can be propagated to other nodes in subsequent rounds, but it takes time before it reaches every node. We exploit this to construct ρ in such a way that some nodes of K (those of G_1) reach an accepting state, while others (those of H_1) reach a rejecting state. Since A is a halting automaton, these nodes will never change their state again, and so the run is neither accepting nor rejecting. ◀

4 Synchronicity can always be simulated

We show that every class with synchronous selection is equivalent to the corresponding class with liberal selection. Albeit non-trivial, this is easy to prove by a standard technique of distributed computing known as *alpha synchronizer*. (The term was introduced in [4], but a similar idea appeared earlier in cellular automata theory [10].) Given a machine

$M = (Q, \delta_0, \delta, Y, N)$, we define a machine $\tilde{M} = (\tilde{Q}, \tilde{\delta_0}, \tilde{\delta}, \tilde{Y}, \tilde{N})$ such that for every graph G, the unique synchronous run of M on G accepts (rejects) iff every weakly fair run ρ of $\tilde{M}$ on G accepts (rejects). The gadget achieving this is called a "synchronizer", because it ensures that the nodes of G behave "as in the synchronous case", even when selection is liberal.

The set of states of $\tilde{M}$ is $\tilde{Q} := Q \times Q \times \{0, 1, 2\}$. Given $(q, q', i) \in \tilde{Q}$, we call q the *past M-state*, q' the *current M-state*, and i the *phase*. The initialization function is given by $\tilde{\delta}_0(a) := (\delta_0(a), \delta_0(a), 0)$. In order to define the transition function $\tilde{\delta}$, let v be a node in state (q, q', i). If v is selected by the scheduler, its next state is determined as follows:

- If at least one neighbor of v is in phase $(i - 1) \bmod 3$, then v does not change state. Intuitively, if some neighbor is still one phase behind, then v waits for it to "catch up".
- If every neighbor of v is in phase i or $(i+1) \bmod 3$, then v moves to $(q', q'', (i+1) \bmod 3)$, where q'' is defined as follows. Let N_v be the set of neighbors of v, and for each $u \in N_v$, let (q_u, q'_u, i_u) be the state of u. Further, let $q''_u := q'_u$ if $i_u = i$, and $q''_u := q_u$ if $i_u = (i + 1) \bmod 3$, and let $\mathcal{M}$ be the multiset over Q containing for each $u \in N_v$ a copy of the state q''_u. (Loosely speaking, $\mathcal{M}$ contains the current M-states of the neighbors of v that are in the same phase as v, and the past M-states of the neighbors that are one phase ahead, i.e., the states they had when they were in the same phase as v). Let $\mathcal{M}_\beta$ be given by $\mathcal{M}_\beta(q) = \min\{\beta, \mathcal{M}_\beta(q)\}$. We define $q'' := \delta(q, \mathcal{M}_\beta)$; loosely speaking, v moves to the state it would move to in M if all its neighbors were in the same phase.

Let $\tilde{\rho}$ be any weakly-fair run of $\tilde{M}$ on a graph G. Fix a node v of G, and extract from $\tilde{\rho}$ the sequence $q'_{10}q'_{11}q'_{12}\,q'_{20}q'_{21}q'_{22} \ldots q'_{i0}q'_{i1}q'_{i2} \ldots$, where q'_{ij} denotes the current M-state of v immediately after entering phase j for the i-th time. Now, let ρ be the unique synchronous run of M on G, and let $q'_0q'_1q'_2 \ldots$ be the sequence obtained by projecting ρ onto the states of v. It is easy to see that these two sequences coincide. By the definition of stable acceptance, $\tilde{\rho}$ accepts iff ρ accepts, and rejects iff ρ rejects. Using this construction, we obtain:

▶ **Theorem 5.** *For every* `**$*`*-automaton there is an equivalent* `**s*`*-automaton.*

5 Exclusivity does not increase expressiveness

In this section, we obtain the rather surprising result that the computational power of a class of automata does not increase if we restrict its schedulers to interleaving ones (which guarantees that agents act in mutual exclusion with all other agents).

5.1 Exclusivity under strong fairness

We start by considering strongly fair models, i.e., we compare a class of the form `**sF` with the corresponding class `**SF`. On an intuitive level, their equivalence might be less surprising than the subsequent result presented in Section 5.2 because strong fairness provides a way to break symmetry, which can be exploited to simulate exclusivity. Nevertheless, neither class trivially subsumes the other, so we have to prove inclusions in both directions.

▶ **Theorem 6.** *For every* `**sF`*-automaton there is an equivalent* `**SF`*-automaton.*

Proof sketch. Given a `**sF`-automaton A, we construct a `**SF`-automaton B such that for all input graphs G, every strongly fair run of B on G simulates a strongly fair run of A on G. The difficulty lies in the fact that A and B do not share the same notion of strong fairness because they have different selection constraints. While A's liberal scheduler guarantees that arbitrary sequences of selections will occur infinitely often, B's exclusive scheduler can select only one node at a time. To simulate A's behavior with B, we adapt the synchronizer from Section 4. Just like there, nodes keep track of their previous and current state in A, as well as the current phase number modulo 3. However, instead of updating their state in every phase, they only do so if an additional *activity flag* is set. Thus, we can simulate an arbitrary selection S by raising the flags of exactly those nodes that lie in S. The outcome of a phase simulated in this way will be the same as if all the nodes in S made a transition simultaneously. The main issue is how to set the activity flags in each phase in such a way that every finite sequence $(S_1, \dots, S_n)$ of selections is guaranteed to occur infinitely often. We show that this is possible, exploiting the fact that B's scheduler is strongly fair. ◀

▶ **Theorem 7.** *For every* `**SF`*-automaton there is an equivalent* `**sF`*-automaton.*

Proof sketch. First, we note that the only way exclusivity could possibly be useful is to break symmetry between adjacent nodes. This is because for an independent set (i.e., a set of pairwise non-adjacent nodes), the order of activation is irrelevant: whether the scheduler activates them all at once or one by one in some arbitrary order, the outcome will always be the same. Consequently, to simulate a run with exclusivity, it suffices to simulate a run where no two adjacent nodes are active at the same time. We provide a simple protocol that makes use of the strong fairness constraint (in an environment with liberal selection) to ensure that if a node wants to execute a transition, then it will eventually be able to do so while all its neighbors remain passive. ◀

5.2 Exclusivity under weak fairness

We now show that even in the absence of strong fairness, the restriction to interleaving schedulers does not increase expressive power. At first sight, this may be quite surprising because exclusivity inherently breaks symmetry, whereas an automaton with liberal selection and weak fairness can always be assumed to run synchronously and thus be incapable of breaking symmetry. In fact, it is easy to come up with examples of automata that exploit exclusivity to ensure termination.

▶ **Proposition 8.** *For every* `**sf`*-automaton, there exists a* `**Sf`*-automaton that recognizes the same graph language but makes use of exclusive selection to ensure termination. If run synchronously, it never terminates (and hence it is not a valid* `**sf`*-automaton).*

However, although the automata described in Proposition 8 make use of exclusivity, they do not really benefit from it; they only recognize languages that can also be recognized by liberal automata. As we will see in Theorem 11, this observation can be generalized to arbitrary `**Sf`-automata. Intuitively, since exclusivity does not add any expressive power, it can in a certain sense be simulated without needing to break symmetry.

The proof of Theorem 11 is based on the notion of Kronecker cover. The *Kronecker cover* (also known as *bipartite double cover*) of a graph $G = (V, E, \lambda)$ is the bipartite graph $G' = (V', E', \lambda')$ where $V' = V \times \{0, 1\}$, $E' = \bigcup_{\{u,v\} \in E} \{\, \{(u, 0), (v, 1)\}, \{(u, 1), (v, 0)\} \,\}$, and $\lambda'((v, i)) = \lambda(v)$ for all $(v, i) \in V'$. An example is provided in Figure 3.

The Kronecker cover in Figure 3 is connected because the nodes in $\{u, v, w\} \times \{0, 1\}$ form a cycle. The following lemma generalizes this observation.

▶ **Lemma 9.** *The Kronecker cover of a connected graph G is connected if and only if G contains a cycle of odd length, (i.e., if and only if G is non-bipartite).*

If a Kronecker cover is connected, then it constitutes a legal input for a distributed automaton. The next key lemma shows that, in this case, a weakly fair automaton cannot even distinguish between a graph and its Kronecker cover.

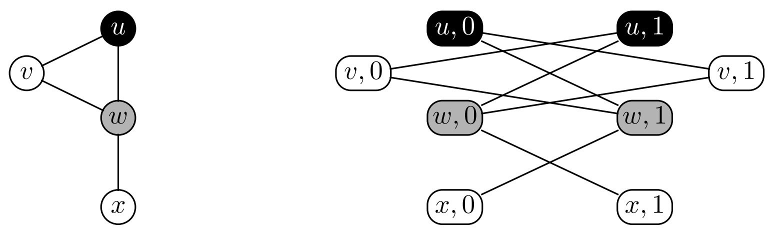

Figure 3 A graph (on the left) and its Kronecker cover (on the right).

▶ **Lemma 10.** *For every* `***f`*-automaton A with input alphabet Λ and every non-bipartite Λ-labeled graph G, A accepts G if and only if it accepts the Kronecker cover of G.*

We can now prove the main technical result of this section:

▶ **Theorem 11.** *For every* `**Sf`*-automaton there is an equivalent* `**sf`*-automaton.*

Proof sketch. Given a `**Sf`-automaton A, we construct an equivalent `**$f`-automaton B (i.e., a synchronous automaton). This is sufficient to prove the claim, because we know from Theorem 5 that B can always be simulated by a `**sf`-automaton using a synchronizer.

Let G be an input graph for A. If we were guaranteed that the labels of G define a proper vertex coloring (i.e., edges connect nodes of different colors), then the task would be straightforward. Indeed, since each color of a proper coloring represents an independent set, B could simply operate in cyclically repeating phases, each one activating precisely the nodes of one of the colors. As explained in the proof of Theorem 7, such a run is equivalent to a run of an exclusive scheduler that activates the nodes of each independent set one by one (in some arbitrary order).

This approach can be adapted to bipartite graphs because a bipartite graph has exactly two possible 2-colorings. However, computing one of the two 2-colorings would require to break symmetry, which a `**$f`-automaton cannot do. So instead, the states of automaton B have two components, one corresponding to each coloring, and nodes update both components when they are activated.

Using these ideas, we construct B in such a way that it recognizes the same bipartite graphs as A. Then we use Lemmas 9 and 10 to prove that $L(A) = L(B)$. Indeed, if G is not bipartite, then by Lemma 9, its Kronecker cover G' is connected and therefore constitutes a legal input for a distributed automaton. By Lemma 10, B accepts G if and only if it accepts G'. Since Kronecker covers are bipartite by definition, we know from the above discussion that B accepts G' if and only if A accepts G'. Finally, again by Lemma 10, A accepts G' if and only if it accepts G. From this chain of equivalences, we can conclude that G is accepted by B if and only if it is accepted by A. ◀

6 Separations

In Sections 3, 4 and 5 we have shown that the classes of graph languages in Figure 1 collapse to *at most* the seven classes shown on the left of Figure 4. In this section we show that the seven classes are all different. For this we examine four graph languages, and determine which classes are expressive enough to recognize them:

- $\mathcal{B}$: The language of graphs with set of labels $\{black, white\}$ having at least one black node.
- $\mathcal{S}$: The language of star graphs, i.e., the set of all connected, unlabeled graphs in which one node (the *center*) has degree at least 2, and all others (the *leaves*) have degree 1.

- $\mathcal{C}_3$: The language containing one single graph, namely the cycle C_3 with three nodes labeled by 0, 1, and 2, respectively.
- $\mathcal{S}_{even}$: The language of even stars, i.e., the graphs of $\mathcal{S}$ with an even number of leaves.

The results are summarized on the right of Figure 4.

$$\begin{tikzcd}
& \texttt{DAsF} & \\
\texttt{DasF} \arrow[ur] & \texttt{DAsf} \arrow[u] & \texttt{dAsF} \arrow[ul] \\
\texttt{Dasf} \arrow[u] \arrow[ur] & & \texttt{dAsf} \arrow[u] \arrow[ul] \\
& \texttt{dasf} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Class	$\mathcal{B}$	$\mathcal{S}$	$\mathcal{C}_3$	$\mathcal{S}_{even}$
`DAsF`	✓	✓	✓	✓
`DasF`	✗	✓	✗	✓
`DAsf`	✓	✓	✗	✗
`dAsF`	✓	✓	✓	✗
`Dasf`	✗	✓	✗	✗
`dAsf`	✓	✗	✗	✗
`dasf`	✗	✗	✗	✗

Figure 4 On the left, quotient of the classification of Figure 1. On the right, four graph languages, and the automata models capable of recognizing them.

Recognizing properties of labeled graphs: the language $\mathcal{B}$

The main difference between the two types of acceptance is that halting automata cannot recognize properties that require nodes to wait an unlimited amount of time for some information that may never arrive, while even the simplest class of automata accepting by stable consensus can recognize some of those properties, such as $\mathcal{B}$.

▶ **Proposition 12.** *$\mathcal{B}$ is recognizable by a* `dAsf`*-automaton, but not by any* `*a**`*-automaton.*

Proof sketch. The `dAsf`-automaton has two states, called *black* and *white*. The initial state of a node is given by its label. Black nodes remain always black, and white nodes with a black neighbor become black. Since graphs are connected by assumption, if a graph contains some black node then eventually all nodes are black, otherwise all nodes stay white.

For the second part, one can show that `DasF`-automata cannot distinguish between an entirely white cycle and a sufficiently long path graph whose nodes are all white except for two black nodes at the endpoints. (The argument is similar to the proof of Theorem 4.) ◀

Recognizing properties of unlabeled graphs: the language $\mathcal{S}$

We show in Proposition 13 that `dAsf`-automata cannot recognize any non-trivial property of *unlabeled* graphs (which we identify with the labeled graphs whose nodes all carry the same label). That is, while `dAsf`-automata can recognize properties of the labeling of a graph, they cannot recognize any non-trivial property of its *structure*. Then we show in Proposition 14 that the strong fairness of `dAsF`-automata allows them to recognize $\mathcal{S}$.

▶ **Proposition 13.** `dAsf`*-automata can only recognize trivial properties of* unlabeled *graphs. In particular, $\mathcal{S}$ is not recognizable by a* `dAsf`*-automaton.*

Proof. Let A be a `dAsf`-automaton, and let $\rho = (C_0, C_1, \ldots)$ be the synchronous run of A on an unlabeled graph $G = (V, E)$, i.e., the run scheduled by V^ω. We show that A either accepts all unlabeled graphs, or rejects all unlabeled graphs. Since V^ω is a weakly fair schedule, ρ is a fair run, and so by the consistency condition A accepts G iff ρ is accepting. Since G is unlabeled, in C_0 every node of G is in the same state q_0, which is independent of G. Moreover, since ρ is synchronous and A is non-counting, in each configuration C_i every node of G is in the same state q_i, which is also independent of G. So the states visited by ρ are independent of G, and so A either accepts all unlabeled graphs, or rejects all unlabeled graphs. ◀

▶ **Proposition 14.** *$\mathcal{S}$ is recognizable by a* `dAsF`*-automaton and by a* `Dasf`*-automaton.*

Proof sketch. We give a `dAsF`-automaton that recognizes $\mathcal{S}$. The states of the automaton are pairs (d, c), where $d \in \{\mathit{leaf}, \mathit{center}, \mathit{unknown}, \mathit{neither}\}$ is the *estimate* of v, and $c \in \{0, 1\}$ is its *color*. Every time a node is selected it flips its color. When a node with estimate *unknown* sees two neighbors with different colors, it switches to *center*, and if from then on it sees a neighbor with estimate *center*, it moves to *neither*. Strong fairness is crucial for correctness: by Lemma 2, it ensures that a node that is not a leaf will eventually be selected in a configuration in which at least two of its neighbors have different colors.

Now we give a `Dasf`-automaton with $\beta = 2$ that recognizes $\mathcal{S}$. Since $\beta = 2$, a node can determine for each state q if it has 0, 1, or at least 2 neighbors in q. The automaton's states are $\{\mathit{init}, \mathit{leaf}, \mathit{non\text{-}leaf}, \mathit{accept}, \mathit{reject}\}$. Initially all nodes are in state *init*. The nodes update their estimates depending on the number of neighbors (0, 1, or at least 2) in each state. ◀

Symmetry breaking: the language $\mathcal{C}_3$

We show that the language $\mathcal{C}_3$ requires both acceptance by stable consensus and strong fairness to be recognizable. Intuitively, both of them are required to distinguish C_3 from arbitrarily long cycles that repeat the labeling of C_3 cyclically.

▶ **Proposition 15.** *$\mathcal{C}_3$ is recognizable by a* `dAsF`*-automaton, but neither by* `DA*f`*-automata nor by* `Da*F`*-automata.*

Proof sketch. Our `dAsF`-automaton for $\mathcal{C}_3$ checks two conditions: first, that the input graph is a cycle with cyclic labeling $0{-}1{-}2$, and second, that it contains exactly one node labeled by 2 (which implies that the cycle has length 3). For both conditions, we use a similar trick as in Proposition 14, relying on acceptance by stable consensus and strong fairness to eventually break symmetry between otherwise indistinguishable nodes. To verify the second condition, each node labeled by 2 successively sends signals in both directions through the cycle, and checks that those signals always come back from the expected direction.

For the second part of the claim, we show that `DA*f`- and `Da*F`-automata cannot distinguish C_3 from C_6, the hexagon whose nodes are labeled by $0{-}1{-}2{-}0{-}1{-}2$ (and back to 0). To do so, given a fair run ρ_3 of such an automaton on C_3, we construct a fair run ρ_6 on C_6 that "duplicates" the behavior of ρ_3. In the case of `Da*F`-automata, this duplication is performed only until ρ_3 has reached a halting configuration (because otherwise ρ_6 would violate the strong fairness constraint). ◀

Counting neighbors modulo a number: the language $\mathcal{S}_{even}$

Since counting automata can only count up to a threshold β, no node can directly observe that it has an even number of neighbors. This makes the language $\mathcal{S}_{even}$ rather difficult to recognize. We now show that the combination of counting and strong fairness can do the job. The proof also provides a good example where exclusivity helps to design an algorithm.

▶ **Proposition 16.** *$\mathcal{S}_{even}$ is recognizable by a* `DasF`*-automaton.*

Proof sketch. In Proposition 14 we have exhibited a `Dasf`-automaton A recognizing $\mathcal{S}$. We now give a `DaSF`-automaton B that uses counting, exclusivity, and strong fairness to further decide if the number of leaves is even. Loosely speaking, B first executes A; if A rejects, then B rejects, because the graph is not even a star. If A accepts, then B enters a new phase during which it counts the number of leaves modulo 2. By Theorem 7, B is equivalent to a `DasF`-automaton.

We can assume that when A accepts, all nodes are labeled with either *leaf* or *center* (the unique non-leaf). We give an informal description of B. Leaves can be in states *visible*, *invisible*, *dead*, *even*, or *odd*. While leaves have not been counted by the center, they alternate between the states *visible* and *invisible*. The center only increments its modulo-2 counter if exactly one leaf is *visible*. After a leaf is counted, it moves to *dead*. When all leaves become dead, i.e., when they have all been counted, the center decides whether to accept or reject; the leaves read the decision from the counter, and move to *even* or *odd* accordingly. ◀

The next two results show that recognizing $\mathcal{S}_{even}$ needs both counting and strong fairness.

▶ **Proposition 17.** *$\mathcal{S}_{even}$ is not recognizable by* `DA*f`*-automata.*

Proof. We show that for every `DA*f`-automaton A there exist stars G and G' such that exactly one of G and G' belongs to $\mathcal{S}_{even}$, but A either accepts both of them or rejects both of them. Let $\beta \geq 1$ be A's counting bound, and let G and G' be the stars with $\beta + 1$ and $\beta + 2$ leaves, respectively. Now consider the synchronous runs ρ and ρ' of A on G and G'. By symmetry, and since the number of leaves exceeds β in both G and G', at every time $t \in \mathbb{N}$, the center is in the same state in ρ and ρ', and likewise all leaves are in the same state. So the sequences of states visited by the center and the leaves are the same in both ρ and ρ', and therefore ρ is accepting iff ρ' is accepting. ◀

▶ **Proposition 18.** *$\mathcal{S}_{even}$ is not recognizable by* `dA*F`*-automata.*

Proof sketch. Given a `dA*F`-automaton A, the proof identifies an even number n, depending on A, such that if A accepts the star with n leaves, then it cannot reject the star with $n + 1$ leaves. The proof is involved, and can be found in the appendix of the arXiv version. ◀

7 Expressive power

As a first application of our results, we investigate the expressivity of our models for graph languages that depend only on the labeling function of a graph, and not on its topology.

Given a Λ-labeled graph $G = (V, E, \lambda)$, where $\Lambda = \{\ell_1, \dots, \ell_k\}$, let $\#_G \colon \Lambda \to \mathbb{N}$ be the mapping that assigns to each label ℓ the number $\#_G(\ell)$ of nodes of V such that $\lambda(v) = \ell$. A language is *Presburger-definable* if there is a formula $\varphi(x_1, \dots, x_k)$ of Presburger arithmetic such that a Λ-labeled graph G belongs to the language if and only if $\varphi(\#_G(\ell_1), \dots, \#_G(\ell_k))$ holds. An example of such a language is $\mathcal{B}$, the set of graphs that contain a black node.

We show that `DAsF`-automata recognize all Presburger languages, but none of the other six classes do. The negative part of the result follows easily from the table in Figure 4.

▶ **Proposition 19.** *There exist Presburger-definable languages that are not recognizable by* `d***`*-,* `*a**`*-, or* `***f`*-automata.*

Proof. By Proposition 12, `*a**`-automata cannot recognize the language $\mathcal{B}$, which is Presburger-definable. Furthermore, by Propositions 14, 17 and 18, `dA*F`- and `DA*f`-automata can recognize the language $\mathcal{S}$ of star graphs but not the language $\mathcal{S}_{even}$ of stars with an even number of leaves. This implies that dA*F- and DA*f-automata cannot recognize the Presburger-definable language of graphs with an odd number of nodes, because the intersection of this language with $\mathcal{S}$ is equal to $\mathcal{S}_{even}$, and languages recognizable by distributed automata are closed under intersection (by a standard product construction). ◀

For the positive part, we proceed in three steps: First, following [1] and Section 5 of [3], we introduce *graph population protocols*, a graph variant of the well-known population protocol model introduced in [2, 3]. Then we recall a result of [3] showing that graph population protocols recognize all Presburger-definable languages. Finally, we show that every graph population protocol can be simulated by a DAsF-automaton.

Our definition of graph population protocols is equivalent to that of [1, 3], but reuses the notation of Section 2 as far as possible. A *graph population protocol* $\mathit{\Pi} = (Q, \delta_0, \delta, Y, N)$ is defined like a DASF-automaton with machine $M = \mathit{\Pi}$, except for the following differences:

- The transition function is of the form $\delta\colon Q^2 \to Q^2$.
- A selection of a graph $G = (V, E, \lambda)$ is an ordered pair $S = (u, v) \in V^2$ of *adjacent* nodes (instead of a singleton $\{u\} \subseteq V$), and the selection constraint on G is $\{(u, v) \mid \{u, v\} \in E\}$.
- $C_t(v)$ is defined inductively as follows, for $t \in \mathbb{N}$ and $v \in V$:

$$C_0(v) = \delta_0(\lambda(v)) \quad \text{and} \quad C_{t+1}(v) = \begin{cases} \delta\big(C_t(v), C_t(u)\big)_{\text{fst}} & \text{if } S_t = (v, u) \text{ for some } u, \\ \delta\big(C_t(u), C_t(v)\big)_{\text{snd}} & \text{if } S_t = (u, v) \text{ for some } u, \\ C_t(v) & \text{otherwise,} \end{cases}$$

  where P_{fst} and P_{snd} denote the first and second component of a pair P.

So, intuitively, the scheduler selects two adjacent nodes, which update their states according to δ. The definitions of all other relevant notions remain the same. This holds in particular for acceptance by stable consensus and strong fairness (which are baked into the model), and the consistency condition. Standard population protocols correspond to graph population protocols on complete graphs, where every pair of distinct nodes is connected by an edge.

It is shown in [3] that standard population protocols recognize all Presburger-definable languages. Further, Theorem 7 of [3] shows that every language recognized by population protocols is also recognized by graph population protocols. Loosely speaking, given a population protocol, one constructs the protocol on graphs in which, when an edge of the graph is selected, either the two nodes connected by it interact as in the population protocol, or they swap their states. By strong fairness, the states of the nodes can "move around the graph", and any pair of states eventually interacts infinitely often. The choice between interacting or swapping is nondeterministic, but it can be simulated by deterministic transitions (see [3]). Therefore, in order to show that DA*F-automata recognize all Presburger-definable languages, it suffices to simulate graph population protocols with distributed automata. As in the proof of Proposition 16, we make use of exclusivity to simplify the construction.

▶ **Proposition 20.** *For every graph population protocol there is an equivalent* DA*F-*automaton.*

Proof sketch. We present a simulation that runs a population protocol on a distributed automaton. To this end, the automaton has to simulate a scheduler that selects ordered pairs of adjacent nodes instead of arbitrary sets of nodes. For any pair (u, v) that is selected to perform a transition, let us call u the *initiator* and v the *responder* of the transition. By Theorem 7, we may assume that the automaton's scheduler selects a single node in each step.

The main idea is as follows: When a node u is selected and sees that it can become the initiator of a transition, it declares its intention to do so by raising the flag "?". Then u waits until some neighbor v is selected and raises the flag "!", which signals that v wants to become the responder of a transition. If this happens, the next time u is selected, it computes its new state according to the state of v and the transition function of the population protocol, but also keeps its old state in memory so that v can still see it. After that, v also updates its state, and finally u deletes its old state, which completes the transition. Throughout this protocol, the nodes verify that they have exactly one partner during each transition. If this condition is violated, they raise the error flag "$\perp$" and abort their current transition. ◀

▶ **Corollary 21.** `DA*F`*-automata recognize all Presburger-definable languages.*

8 Conclusions

We have conducted an extensive comparative analysis of the expressive power of weak asynchronous models of distributed computing. Our analysis has reduced the initial "jungle" of twenty different models to only seven. This reduction in complexity is achieved by Theorems 4, 5, 6, 7, and 11, all of which have a clear and intuitive interpretation.

We have also shown that the seven classes are distinct, and have identified inclusions and non-inclusions between them. However, two inclusions remain open: Are `Dasf` or `DAsf` included in `dAsF`? Intuitively, this asks if strong fairness and acceptance by stable consensus can be used to simulate counting. We can provide a positive answer for graphs of bounded degree (a limitation common in practice), because in this case even `dA*F` and `DAsF` coincide.

▶ **Proposition 22.** *For every* `DA*F`*-automaton A and every $k \in \mathbb{N}$ there is a* `dA*F`*-automaton B equivalent to A on graphs of maximum degree k.*

However, for arbitrary graphs we conjecture that neither `Dasf` nor `DAsf` are included in `dAsF`.

Finally, we have made a first step towards characterizing the graph languages recognizable by the different classes, by transferring a characterization for population protocols.

As a last note, observe that our results hold for *decision* problems on *undirected* graphs that can be solved by consensus in the framework of distributed automata. Several of our constructions (e.g., those in Theorems 5 and 7) rely on bidirectional communication, which is not guaranteed on directed graphs. Furthermore, exclusive selection leads to higher computational power for non-decision problems. For instance, it can be used to solve the vertex coloring problem on graphs of bounded degree (by a standard greedy algorithm), which, for symmetry reasons, is impossible in a model with synchronous selection.

References

1 Dana Angluin, James Aspnes, Melody Chan, Michael J Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In *International Conference on Distributed Computing in Sensor Systems*, pages 63–74. Springer, 2005.

2 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC*, pages 290–299. ACM, 2004.

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

4 Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985. `doi:10.1145/4221.4227`.

5 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *DISC*, volume 6343 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010.

6 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *PODC*, pages 137–146. ACM, 2013.

7 Nissim Francez. *Fairness.* Texts and Monographs in Computer Science. Springer, 1986.

8 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.

9 Daniel Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *ICALP*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 1981.

10 Katsuhiko Nakamura. Synchronous to asynchronous transformation of polyautomata. *J. Comput. Syst. Sci.*, 23(1):22–37, 1981. doi:10.1016/0022-0000(81)90003-9.

11 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.

12 Fabian Reiter. Asynchronous distributed automata: A characterization of the modal mu-fragment. In *ICALP*, volume 80 of *LIPIcs*, pages 100:1–100:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

13 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, 2008.