An Automata-Theoretic Approach to the Verification of Distributed Algorithms[☆]

C. Aiswarya[a], Benedikt Bollig[b], Paul Gastin[c]

[a] *Chennai Mathematical Institute, India*
[b] *CNRS, ENS Paris-Saclay, LSV, France*
[c] *ENS Paris-Saclay, CNRS, LSV, France*

Abstract

We introduce an automata-theoretic method for the verification of distributed algorithms running on ring networks. In a distributed algorithm, an arbitrary number of processes cooperate to achieve a common goal (e.g., elect a leader). Processes have unique identifiers (pids) from an infinite, totally ordered domain. An algorithm proceeds in synchronous rounds, each round allowing a process to perform a bounded sequence of actions such as send or receive a pid, store it in some register, and compare register contents wrt. the associated total order. An algorithm is supposed to be correct independently of the number of processes. To specify correctness properties, we introduce a logic that can reason about processes and pids. Referring to leader election, it may say that, at the end of an execution, each process stores the maximum pid in some dedicated register.

We show that the verification problem of distributed algorithms can be reduced to satisfiability of a formula from propositional dynamic logic with loop and converse (LCPDL), interpreted over grids over a finite alphabet. This translation is independent of any restriction imposed on the algorithm. However, since the verification problem (and satisfiability for LCPDL) is undecidable, we propose an underapproximation technique, which bounds the number of rounds. This is an appealing approach, as the number of rounds needed by a distributed algorithm to conclude is often exponentially smaller than the number of processes. Using our reduction to LCPDL, we provide an automata-theoretic solution, reducing model checking to emptiness for alternating two-way automata on words. Overall, we show that round-bounded verification of distributed algorithms over rings is PSPACE-complete, provided the number of rounds is given in unary.

Keywords:
2000 MSC: 68Q60, 68Q85

[☆]Supported by LIA INFORMEL.

Email addresses: `aiswarya@cmi.ac.in` (C. Aiswarya), `bollig@lsv.fr` (Benedikt Bollig), `gastin@lsv.fr` (Paul Gastin)

1. Introduction

Distributed algorithms are a classic discipline of computer science and continue to be an active field of research [33, 9, 23]. A distributed algorithm employs several processes, which perform one and the same program to achieve a common goal. It is required to be correct independently of the number of processes. A prominent example is the class of leader-election algorithms, whose task is to determine a unique leader process and to announce it to all other processes. Those algorithms are often studied for ring architectures. One practical motivation comes from local-area networks that are based on a token-ring protocol. Moreover, rings generally allow one to nicely illustrate the main conceptual ideas of an algorithm.

However, it is well-known that there is no (deterministic) distributed algorithm over rings that elects a leader under the assumption of anonymous processes. Therefore, classical algorithms, such as Franklin's algorithm [25] or the Dolev-Klawe-Rodeh/Peterson algorithm [15, 35], assume that every process is equipped with a unique process identifier (pid) from an infinite, totally ordered domain. In this paper, we consider such distributed algorithms, which work on ring architectures and can access unique pids as well as the associated total order.

Distributed algorithms are difficult to analyze. Correctness proofs are often intricate and use subtle inductive arguments. Therefore, it is worthwhile to consider automatic verification methods such as model checking [14]. Besides a formal model of an algorithm, this requires a generic specification language that is feasible from an algorithmic point of view but expressive enough to formulate correctness properties. In this paper, we propose a language that can reason about processes, states, and pids. In particular, it will allow us to formalize when a leader-election algorithm is correct: *At the end of an execution, every process stores, in register r, the maximum pid among all processes.* Our language is inspired by Data-XPath, which can reason about trees over infinite alphabets [10, 11, 20].

However, formal verification of distributed algorithms cumulates various difficulties that already arise, separately, in more standard verification: First, the number of processes is unknown, which amounts to parameterized verification [18]; second, processes manipulate data from an infinite domain [11, 20]. In each case, even simple verification questions are undecidable, and so is in the combination of both.

In various other contexts, a successful approach to retrieving decidability has been a form of *bounded model checking*. The idea is to consider correctness up to some parameter, which restricts the set of runs of the algorithm in a non-trivial way. In multi-threaded recursive programs, for example, one may restrict the number of control switches between different threads [36]. Actually, this idea seems even more natural in the context of distributed algorithms, which usually proceed in *rounds*. In each round, a process may emit some messages (here: pids) to its neighbors, and then receive messages from its neighbors. Pids can be stored in registers, and a process can check the relation between stored pids before it moves to a new state and is ready for a new round. It turns out that the number of rounds is often exponentially smaller than the number of processes (cf. the above-mentioned leader-election algorithms). Thus, roughly speaking, a small number of rounds allows us to verify correctness of an algorithm for a large number of processes.

The key idea of our method is to interpret an execution of a distributed algorithm symbolically as a grid-like structure over a finite alphabet. The finite alphabet is constituted by the transitions that occur in the algorithm and possibly contain tests of pids wrt. equality or the associated total order. To determine feasibility of a symbolic execution (i.e., *is there a ring that satisfies all the guards employed?*), we use propositional dynamic logic with loop and converse (LCPDL) [26]. Basically, we translate a given distributed algorithm into a formula that detects cyclic (i.e., contradictory) smaller-than tests. Its models are precisely the feasible symbolic executions. A specification is translated into LCPDL as well so that verification amounts to checking satisfiability of a single formula. Note that both translations do not impose any restriction on the algorithm or the formula. In other words, verification of distributed algorithms is reduced to satisfiability in a logic over a finite alphabet, which is interesting on its own. Decidability is then obtained by bounding the number of rounds so that satisfiability can be reduced to a non-emptiness problem for alternating two-way automata over words. We then obtain a PSPACE procedure for round-bounded model checking. Note that the bound can be adjusted gradually without changing the already computed LCPDL formula.

Related Work. Our work is situated in the field of *parameterized verification*, which aims at showing correctness of a system independently of the number of processes. Parameterized verification has mainly focused on systems without data, i.e., over a finite message alphabet (see [2] for a comprehensive bibliography). For example, [17, 16] consider communication through token passing in ring-based architectures, which was later extended to more general architectures in [7, 8]. The paper [37] also addresses some kind of parameterized-verification problem in an automata-theoretic/logical framework, but the goal is to reason about robots that explore an unknown and unbounded environment. A symbolic encoding of an infinite number of configurations, though very different from ours, has been used in [3]. Another branching-time temporal logic that, like LCPDL, can explicitly talk about cycles is presented in [24].

Actually, some of the above-mentioned papers rely on a generic approach that reduces model checking to verifying finite-state systems up to a certain number of processes, which is referred to as a *cut-off* [17, 16, 7, 39, 8]. Note that, due to general undecidability, cut-offs do not exist in our setting, though it would be interesting to find restrictions that come with cut-offs while preserving the combination of a parameterized process topology and unbounded data. Other parameterized systems can be modeled as Petri nets or well-structured transition systems, which enjoy positive decidability results [1, 21]. In the context of data words, well-structured transition systems are at the heart of satisfiability checking for a variety of temporal logics [19].

As far as distributed algorithms are concerned, considerable effort has been devoted to the verification of fault-tolerant algorithms, which have to cope with obstacles such as lost or corrupted messages (e.g., [22, 13, 30]). After all, there have been only very few generic approaches to model checking distributed algorithms. In [29], several possible reasons for this are identified, among them the presence of unbounded data types and an unbounded number of processes, which we have to treat simultaneously in our framework.

The theory of words and trees over infinite alphabets (aka data words/trees) provides an elegant formal framework for database-related notions such as XML documents [11], or for the analysis of programs with data structures such as lists and arrays [5, 6]. Notably, streaming transducers [5] also work over an infinite, totally ordered domain. The difference to our work is that we model distributed algorithms and provide a logical specification language. The latter borrows concepts from [10, 11, 20], whose logics are designed to reason about XML documents. A fragment of MSO logic over *ordered* data trees was studied in [40]. The paper [12] also pursued a symbolic model-checking approach to systems involving data. But the model was purely sequential and pids could only be compared for equality. The ordering on the data domain actually has a subtle impact on the choice of the specification language.

Outline. In Section, 2, we present our model of a distributed algorithm. Section 3 demonstrates how valid computations of a given algorithm can be represented as a formula from LCPDL over a finite alphabet. Essentially, we show that a symbolic representation of a concrete run is enough to keep track of pids and to validate its feasability. Section 4 introduces a specification language to express correctness criteria. We first present a logic that allows us to talk about (dis)equality of process identifiers. We then extend that logic to also talk about a linear order on pids. Finally, we explain how our logic can be translated to LCPDL. In Section 5, we exploit the results from Sections 3 and 4 and show how to solve the round-bounded model-checking problem in polynomial space. We conclude in Section 6.

This paper is a revised and extended version of [4].

2. Distributed Algorithms

Let $\mathbb{N} = \{0, 1, 2, \dots\}$ denote the set of natural numbers and $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$ denote the set of integers. For $n \in \mathbb{N}$, let $(\mathbb{Z}_n, \oplus, 0)$ be the cyclic group, where $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$ and $\oplus$ is addition modulo n. The set of finite words over an alphabet A is denoted by A^*, and the set of nonempty finite words by A^+.

Syntax of Distributed Algorithms. We consider distributed algorithms that run on arbitrary ring architectures. A ring architecture of size n is described by the cyclic group $(\mathbb{Z}_n, \oplus, 0)$. Every process has a unique right neighbor (referred to by **right**) and a unique left neighbor (referred to by **left**). The right (resp. left) neighbor of process $i \in \mathbb{Z}_n$ is process $i \oplus 1$ (resp. $i \oplus n - 1$). In a concrete ring, each process has a unique identifier (pid). So a ring R is an injective map $R\colon \mathbb{Z}_n \to \mathbb{N}$ that we will often describe as the tuple $R = (R(0), \dots, R(n-1))$. A distributed algorithm will not be able to distinguish between cyclic permutations of rings, for example, between $R = (4, 1, 5, 2)$ and $R' = (5, 2, 4, 1)$.

We denote by $\mathsf{pids}(R) = R(\mathbb{Z}_n)$ the set of pids of processes participating in the ring. We also define the collection of processes that are located *between* two distinguished processes $i, j \in \mathbb{Z}_n$ (i.e., those processes that we see when we start from i and go to the right until we reach j). Formally, we let $\mathit{Between}(i,j) \overset{\text{def}}{=} \{i+1,\dots,j-1\}$ if $i < j$, and $\mathit{Between}(i,j) \overset{\text{def}}{=} \{i+1,\dots,n-1,0,\dots,j-1\}$ if $j \le i$.

One given distributed algorithm can be run on *any* ring. It is given by a single program $\mathcal{D}$, but each process will actually run a copy of $\mathcal{D}$. It is convenient to think of $\mathcal{D}$ as a (finite) automaton with registers. Processes proceed in synchronous rounds. In one round, every process executes one transition of its program. In addition to the change of state, it may optionally perform the following phases within a transition: (i) send a message, possibly incuding pids, to its neighbors, (ii) receive a message from its neighbors and store the included pids in registers, (iii) compare register contents with one another, (iv) update its registers.

We will now give the formal syntax of distributed algorithms. To this end, we fix a countably infinite supply $\mathcal{N}$ of *names*. A name may refer to a *message type*, a *register*, or a *proposition* (i.e., a state property in the algorithm such as "leader"). Thus, $\mathcal{N}$ is partitioned into $\mathcal{N} = \mathcal{M} \uplus \mathcal{R} \uplus \mathcal{P}$. We assume that there is a special register $\mathsf{id} \in \mathcal{R}$, which will never be updated so that a process can, at any time, access its own pid in terms of id. In addition, every message type $a \in \mathcal{M}$ comes with an arity $k \in \mathbb{N}$, which determines the number of pids that can be sent along with a. Messages with arity k are collected in $\mathcal{M}^{(k)}$ so that $\mathcal{M} = \bigcup_{k\in\mathbb{N}} \mathcal{M}^{(k)}$. We assume all those sets to be infinite.

Definition 1. A *distributed algorithm* $\mathcal{D} = (S, \mathsf{init}, \Delta, \mu)$ consists of a nonempty finite set S of *(local) states*, an *initial state* $\mathsf{init} \in S$, a nonempty *finite* set Δ of *transitions*, and a mapping $\mu\colon S \to 2^{\mathcal{P}}$ assigning to each state s the *finite* set of propositions that hold in s. A transition is of the form

$$\langle s\colon \mathit{send}\,;\mathit{rec}\,;\mathit{guard}\,;\mathit{update}\,;\mathbf{goto}\ s'\rangle$$

where $s, s' \in S$ and the components *send*, *rec*, *guard*, and *update* are built according to the grammar given in Figure 1. There, $k, k' \in \mathbb{N}$, $a \in \mathcal{M}^{(k)}$, $a' \in \mathcal{M}^{(k')}$ and r, r', $r_1,\dots,r_k$, $r'_1,\dots,r'_{k'}$ range over $\mathcal{R}$. In addition, we require that

(1) in a *rec* statement of the form $\mathbf{left}?a(r_1,\dots,r_k)\,;\mathbf{right}?a'(r'_1,\dots,r'_{k'})$ (resp. $\mathbf{left}?a(r_1,\dots,r_k)$ or $\mathbf{right}?a'(r'_1,\dots,r'_{k'})$), the registers $r_1,\dots,r_k,r'_1,\dots,r'_{k'}$ (resp. $r_1,\dots,r_k$ or $r'_1,\dots,r'_{k'}$) are pairwise distinct (in particular, the order of the two receive actions does not matter),

(2) in an *update* statement, every register occurs at most once as a left-hand side, and

(3) id occurs neither in a *rec* statement nor as a left-hand side of an *update* statement.

In the following, occurrences of "**skip** ;" are omitted; this will not affect the semantics. $\lhd$

By $\mathit{Names}(\mathcal{D}) \subseteq \mathcal{N}$, we denote the (finite) set of messages, registers, and propositions that occur in $\mathcal{D}$, with one exception: we always assume $\mathsf{id} \in \mathit{Names}(\mathcal{D})$.

In an execution of an algorithm, registers will be instantiated with pids. We will refer to $a(p_1,\dots,p_k)$, with $a \in \mathcal{M}^{(k)}$ and $p_1,\dots,p_k \in \mathbb{N}$, as a *message*, while a is its message type.

$$
\begin{array}{lcl}
\mathit{send} & ::= & (\mathbf{skip} \mid \mathbf{fwd\text{-}left} \mid \mathbf{left}!a(r_1,\ldots,r_k))\,;(\mathbf{skip} \mid \mathbf{fwd\text{-}right} \mid \mathbf{right}!a'(r'_1,\ldots,r'_{k'})) \\
\mathit{rec} & ::= & (\mathbf{skip} \mid \mathbf{left}?a(r_1,\ldots,r_k))\,;(\mathbf{skip} \mid \mathbf{right}?a'(r'_1,\ldots,r'_{k'})) \\
\mathit{guard} & ::= & \mathbf{skip} \mid r < r' \mid r = r' \mid \mathit{guard}\,;\mathit{guard} \\
\mathit{update} & ::= & \mathbf{skip} \mid r := r' \mid \mathit{update}\,;\mathit{update}
\end{array}
$$

Figure 1: Transition labels

However, when convenient, we may also refer to $a(r_1, \ldots, r_k)$, with $r_1, \ldots, r_k$ registers, as a message.

To give an example of a transition, consider

$$t = \langle s\colon \mathbf{left}!a(r_1, r_2)\,; \mathbf{right}!b(r')\,; \mathbf{right}?a(r'_1, r'_2)\,; r_1 < r'_1\,; r := r'_1\,; \mathbf{goto}\ s' \rangle\,.$$

A process can execute t if it is in state s. It then sends a along with the contents of registers r_1 and r_2 to its left neighbor, as well as b and the contents of r' to its right neighbor. If, afterwards, it receives a with attached pids (p_1, p_2) from its right neighbor, it stores p_1 and p_2 in r'_1 and r'_2, respectively. If p_1 is strictly greater than what has been stored in r_1, it sets r to p_1 and goes to state s'. Otherwise, the transition is not applicable. In the *send* statement, we could also employ a forward command like **fwd-left**, which will forward a message that is received from the right, to the left. Note that a message can be forwarded, in one and the same synchronous round, across several processes provided they all execute **fwd-left**. Forwards can be understood as a kind of *acceleration*. They will allow us to save a certain number of rounds in favor of a more efficient verification algorithm. Note that each forward-step corresponds to one step in an asynchronous system like in [15].

Note that a guard $r \leq r'$ can be simulated in terms of guards $r < r'$ and $r = r'$, using several transitions. We separate $<$ and $=$ for convenience. They are actually quite different in nature, as we will see later in the proof of our main results.

In the semantics, we will suppose that all updates of a transition happen simultaneously, i.e., after executing $r := r'\,; r' := r$, the values previously stored in r and r' will be swapped (and do not necessarily coincide). Moreover, the order of two sends and the order of two receives within a transition do not matter, even though we assume sends happen before receives. This allows us to associate with a transition the set of its instructions and guards as well as the local state of the process. To this end, from a transition $t = \langle s\colon \mathbf{left}!a(r_1, r_2)\,; \mathbf{right}!b(r')\,; \mathbf{right}?a(r'_1, r'_2)\,; r_1 < r'_1\,; r := r'_1\,; \mathbf{goto}\ s' \rangle$, we extract a set of *propositions* which we denote by $\mathsf{PropSetOf}(t)$:

$$\mathsf{PropSetOf}(t) \overset{\text{def}}{=} \{s\,,\ \mathbf{left}!a(r_1, r_2)\,,\ \mathbf{right}!b(r')\,,\ \mathbf{right}?a(r'_1, r'_2)\,,\ r_1 < r'_1\,,\ r := r'_1\,,\ \mathbf{goto}\ s'\}$$

We call its elements propositions, because in our logic we will later reason about them just like about propositions from $\mathcal{P}$.

states: $\mathit{active}_0, \mathit{active}_1$	$t_1 = \langle \mathit{active}_0\colon \textbf{right}!a(r)\,;\textbf{left}?a(r')\,;\textbf{goto}\ \mathit{active}_1\rangle$
$\mathit{passive}, \mathit{found}$	$t_2 = \langle \mathit{active}_1\colon \textbf{right}!a(r')\,;\textbf{left}?a(r'')\,;r'' < r'\,;r < r'\,;r := r'\,;\textbf{goto}\ \mathit{active}_0\rangle$
initial state: active_0	$t_3 = \langle \mathit{active}_1\colon \underline{\hspace{4cm}}\,;r' < r\,;\textbf{goto}\ \mathit{passive}\rangle$
registers: id, r, r', r''	$t_4 = \langle \mathit{active}_1\colon \underline{\hspace{4cm}}\,;r' < r''\,;\textbf{goto}\ \mathit{passive}\rangle$
	$t_5 = \langle \mathit{active}_1\colon \underline{\hspace{4cm}}\,;r = r'\,;\textbf{goto}\ \mathit{found}\rangle$
	$t_6 = \langle \mathit{passive}\colon \textbf{fwd-right}\,;\textbf{left}?a(r)\,;\textbf{goto}\ \mathit{passive}\rangle$

Figure 2: Dolev-Klawe-Rodeh/Peterson leader-election algorithm $\mathcal{D}_{\mathsf{LE}}$

Before defining the semantics of a distributed algorithm formally, we will look at an example. Note that, at the beginning of an execution of an algorithm, every register (from the *finite* set of registers that the algorithm uses) contains the pid of the respective process.

Example 1 (Dolev-Klawe-Rodeh/Peterson Leader-Election Algorithm). We consider the leader-election algorithm by Dolev-Klawe-Rodeh/Peterson [15, 35], which determines a leader in a ring. In fact, it is an adaptation of Franklin's algorithm [25] to cope with unidirectional rings. That is, a process can only, say, send to the right and receive from the left. The algorithm, denoted $\mathcal{D}_{\mathsf{LE}}$, is given in Figure 2. Here, we assume that a state $s \in \{\mathit{active}_0, \mathit{active}_1, \mathit{passive}, \mathit{found}\}$ is also the only proposition that holds in s, i.e., $\mu(s) = \{s\}$. The rough idea of the algorithm is that every process determines whether its pid is a local maximum among its (active) neighbors. If so, it continues to compare its pid with active neighbors that are further away. Otherwise, it becomes passive and will henceforth forward any pid it receives.

However, as we assume the ring to be unidirectional, the local maximum among the processes $i-2, i-1, i$ is actually determined by i (rather than $i-1$). Therefore, each process i will execute two transitions, namely t_1 and t_2, and store the pids sent by $i-2$ and $i-1$ in r'' and r', respectively. After two rounds, since r still contains the pid of i itself, i can test if $i-1$ is a local maximum among $i-2, i-1, i$ using the guards in transition t_2. If both guards are satisfied, i stores the pid sent by $i-1$ in r. It henceforth "represents" process $i-1$, which is still in the race, and goes to state active_0. Otherwise, it enters *passive*. The algorithm is correct in the following sense: At the end of an accepting run (each process ends in *passive* or *found*), (i) there is exactly one process that terminates in *found* (but not necessarily the one with the highest pid), and (ii) all processes store the maximal pid in register r. The algorithm terminates after at most $2\lfloor \log_2 n \rfloor + 2$ rounds. $\triangleleft$

Semantics of Distributed Algorithms. Now, we give the formal semantics of a distributed algorithm $\mathcal{D} = (S, \mathsf{init}, \Delta, \mu)$. Let $\textsc{Regs}_{\mathcal{D}} = \mathit{Names}(\mathcal{D}) \cap \mathcal{R}$ denote the set of registers that occur in $\mathcal{D}$ (which includes id). Recall that $\mathcal{D}$ can be run on any ring $R\colon \mathbb{Z}_n \to \mathbb{N}$, for any $n \in \mathbb{N}$. An (R-)configuration of $\mathcal{D}$ is a pair (s, ρ) with $s\colon \mathbb{Z}_n \to S$ and $\rho\colon \mathbb{Z}_n \to \textsc{Regs}_{\mathcal{D}} \to \mathsf{pids}(R)$. We often write $s = (s_0, \ldots, s_{n-1})$ and $\rho = (\rho_0, \ldots, \rho_{n-1})$ where $s_i = s(i)$ is the current state of process i and $\rho_i = \rho(i)\colon \textsc{Regs}_{\mathcal{D}} \to \mathsf{pids}(R)$ maps each register to a pid. Note that $\rho_i(\mathsf{id})$ will not change throughout a run. The configuration is called *initial* if, for every process $i \in \mathbb{Z}_n$, we have $s_i = \mathsf{init}$ and $\rho_i(r) = R(i)$ for all $r \in \textsc{Regs}_{\mathcal{D}}$. Note that there is a unique initial R-configuration.

In one synchronous round, the algorithm moves from one configuration to another one. This is described by a relation

$$C \overset{t}{\rightsquigarrow} C'$$

where $C = (s, \rho)$ and $C' = (s', \rho')$ are R-configurations and $t \colon \mathbb{Z}_n \to \Delta$. Again, we often write $t = (t_0, \ldots, t_{n-1})$ where $t_i = t(i)$ is the transition executed by process i. To determine when $C \overset{t}{\rightsquigarrow} C'$ holds, we first define some auxiliary relations.

Let $i, j \in \mathbb{Z}_n$ be processes. For $a \in \mathcal{M}^{(k)}$, we write $i \overset{a}{\rightarrowtail} j$ if a (together with k process identifiers) is sent by i to the right and received by j (from the left). Similarly, for registers $r, r' \in \textsc{Regs}_{\mathcal{D}}$, we write $r@i \rightarrowtail r'@j$ if the contents of r is sent to the right from i to j, where it is stored in r'. Formally, we let $i \overset{a}{\rightarrowtail} j$ (respectively $r@i \rightarrowtail r'@j$) if

$$\textbf{right}!b(r_1, \ldots, r_k) \in \mathsf{PropSetOf}(t_i) \;\wedge\; \textbf{left}?b(r'_1, \ldots, r'_k) \in \mathsf{PropSetOf}(t_j) \;\wedge\; \textbf{fwd-right} \in \mathsf{PropSetOf}(t_{i'}) \text{ for all } i' \in \mathit{Between}(i, j)$$

such that $b = a$ (respectively $r_\ell = r$ and $r'_\ell = r'$ for some ℓ). Note that, due to the **fwd-right** command, $i \overset{a}{\rightarrowtail} j$ and $r@i \rightarrowtail r'@j$ may hold for several j and r'. That is, a process may receive a message and forward it at the same time, as for instance in transition t_6 of algorithm $\mathcal{D}_{\mathsf{LE}}$ (see Figure 2).

The meaning of $j \overset{a}{\leftarrowtail} i$ and $r'@j \leftarrowtail r@i$ is analogous, we just replace "right" by "left":

$$\textbf{left}!b(r_1, \ldots, r_k) \in \mathsf{PropSetOf}(t_i) \;\wedge\; \textbf{right}?b(r'_1, \ldots, r'_k) \in \mathsf{PropSetOf}(t_j) \;\wedge\; \textbf{fwd-left} \in \mathsf{PropSetOf}(t_{i'}) \text{ for all } i' \in \mathit{Between}(j, i)$$

such that $b = a$ (respectively $r_\ell = r$ and $r'_\ell = r'$ for some ℓ).

The guards in the transitions of t are checked against "intermediate" register assignments $\hat{\rho} \colon \mathbb{Z}_n \to \textsc{Regs}_{\mathcal{D}} \to \mathsf{pids}(R)$, which are defined as follows:

$$\hat{\rho}_j(r') = \begin{cases} \rho_i(r) & \text{if } r@i \rightarrowtail r'@j \text{ or } r'@j \leftarrowtail r@i \\ \rho_j(r') & \text{if, for all } r, i, \text{ neither } r@i \rightarrowtail r'@j \text{ nor } r'@j \leftarrowtail r@i \end{cases}$$

Note that this is well-defined, due to condition (1) in Definition 1.

Now, there is a synchronous round from C to C' with transitions t, denoted $C \overset{t}{\rightsquigarrow} C'$ if, for all $j \in \mathbb{Z}_n$, $k \in \mathbb{N}$, $a \in \mathcal{M}^{(k)}$, and $r_1, \ldots, r_k, r, r' \in \textsc{Regs}_{\mathcal{D}}$, the following hold:

(R$_1$) $s_j \in \mathsf{PropSetOf}(t_j)$ and $(\textbf{goto}\ s'_j) \in \mathsf{PropSetOf}(t_j)$,

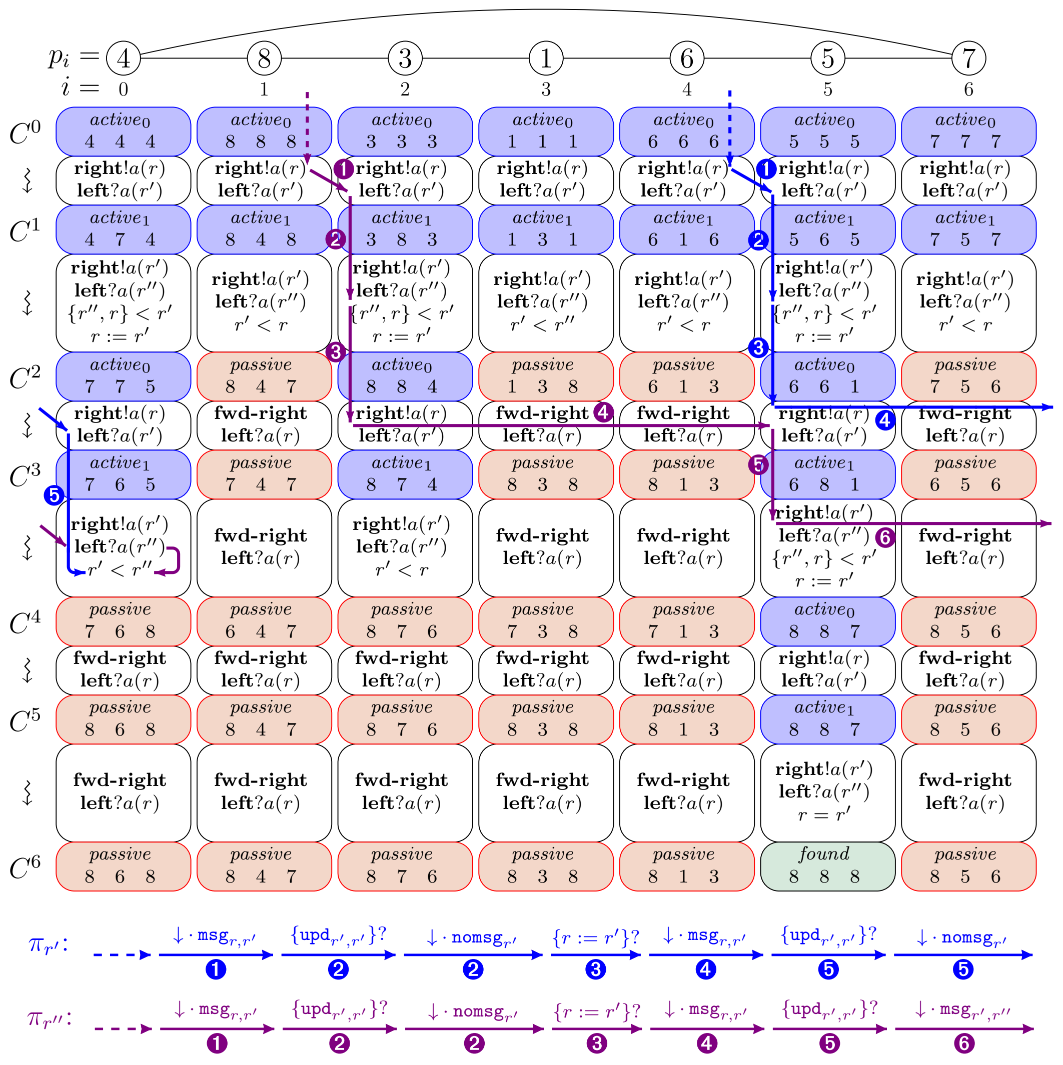
(R$_2$) if $\textbf{left}?a(r_1, \ldots, r_k) \in \mathsf{PropSetOf}(t_j)$, then there is $i \in \mathbb{Z}_n$ such that $i \overset{a}{\rightarrowtail} j$,

(R$_3$) if $\textbf{right}?a(r_1, \ldots, r_k) \in \mathsf{PropSetOf}(t_j)$, then there is $i \in \mathbb{Z}_n$ such that $j \overset{a}{\leftarrowtail} i$,

(R$_4$) $\hat{\rho}_j(r) = \hat{\rho}_j(r')$ if $(r = r') \in \mathsf{PropSetOf}(t_j)$,

(R$_5$) $\hat{\rho}_j(r) < \hat{\rho}_j(r')$ if $(r < r') \in \mathsf{PropSetOf}(t_j)$,

(R$_6$) $\rho'_j(r) = \begin{cases} \hat{\rho}_j(r') & \text{if } (r := r') \in \mathsf{PropSetOf}(t_j) \\ \hat{\rho}_j(r) & \text{if } t_j \text{ does not contain an update of the form } r := r'' \end{cases}$



$p_i =$ 4 8 3 1 6 5 7
$i =$ 0 1 2 3 4 5 6
C^0: active$_0$ 4 4 4 | active$_0$ 8 8 8 | active$_0$ 3 3 3 | active$_0$ 1 1 1 | active$_0$ 6 6 6 | active$_0$ 5 5 5 | active$_0$ 7 7 7
right!$a(r)$ left?$a(r')$ (all columns)
C^1: active$_1$ 4 7 4 | active$_1$ 8 4 8 | active$_1$ 3 8 3 | active$_1$ 1 3 1 | active$_1$ 6 1 6 | active$_1$ 5 6 5 | active$_1$ 7 5 7
right!$a(r')$ left?$a(r'')$ $\{r'', r\} < r'$ $r := r'$ | right!$a(r')$ left?$a(r'')$ $r' < r$ | right!$a(r')$ left?$a(r'')$ $\{r'', r\} < r'$ $r := r'$ | right!$a(r')$ left?$a(r'')$ $r' < r''$ | right!$a(r')$ left?$a(r'')$ $r' < r$ | right!$a(r')$ left?$a(r'')$ $\{r'', r\} < r'$ $r := r'$ | right!$a(r')$ left?$a(r'')$ $r' < r$
C^2: active$_0$ 7 7 5 | passive 8 4 7 | active$_0$ 8 8 4 | passive 1 3 8 | passive 6 1 3 | active$_0$ 6 6 1 | passive 7 5 6
right!$a(r)$ left?$a(r')$ | fwd-right left?$a(r)$ | right!$a(r)$ left?$a(r')$ | fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | right!$a(r)$ left?$a(r')$ | fwd-right left?$a(r)$
C^3: active$_1$ 7 6 5 | passive 7 4 7 | active$_1$ 8 7 4 | passive 8 3 8 | passive 8 1 3 | active$_1$ 6 8 1 | passive 6 5 6
right!$a(r')$ left?$a(r'')$ $r' < r''$ | fwd-right left?$a(r)$ | right!$a(r')$ left?$a(r'')$ $r' < r$ | fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | right!$a(r')$ left?$a(r'')$ $\{r'', r\} < r'$ $r := r'$ | fwd-right left?$a(r)$
C^4: passive 7 6 8 | passive 6 4 7 | passive 8 7 6 | passive 7 3 8 | passive 7 1 3 | active$_0$ 8 8 7 | passive 8 5 6
fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | right!$a(r)$ left?$a(r')$ | fwd-right left?$a(r)$
C^5: passive 8 6 8 | passive 8 4 7 | passive 8 7 6 | passive 8 3 8 | passive 8 1 3 | active$_1$ 8 8 7 | passive 8 5 6
fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | fwd-right left?$a(r)$ | right!$a(r')$ left?$a(r'')$ $r = r'$ | fwd-right left?$a(r)$
C^6: passive 8 6 8 | passive 8 4 7 | passive 8 7 6 | passive 8 3 8 | passive 8 1 3 | found 8 8 8 | passive 8 5 6
$\pi_{r'}$: $\downarrow \cdot \mathtt{msg}_{r,r'}$ 1 | $\{\mathtt{upd}_{r',r'}\}?$ 2 | $\downarrow \cdot \mathtt{nomsg}_{r'}$ 2 | $\{r := r'\}?$ 3 | $\downarrow \cdot \mathtt{msg}_{r,r'}$ 4 | $\{\mathtt{upd}_{r',r'}\}?$ 5 | $\downarrow \cdot \mathtt{nomsg}_{r'}$ 5
$\pi_{r''}$: $\downarrow \cdot \mathtt{msg}_{r,r'}$ 1 | $\{\mathtt{upd}_{r',r'}\}?$ 2 | $\downarrow \cdot \mathtt{nomsg}_{r'}$ 2 | $\{r := r'\}?$ 3 | $\downarrow \cdot \mathtt{msg}_{r,r'}$ 4 | $\{\mathtt{upd}_{r',r'}\}?$ 5 | $\downarrow \cdot \mathtt{msg}_{r',r''}$ 6

Figure 3: Run of Dolev-Klawe-Rodeh algorithm and runs of path automata

Here, (R_6) is well-defined thanks to condition (2) in Definition 1.

An (R-)*run* of $\mathcal{D}$ is a sequence $\chi = C^0 \overset{t^1}{\rightsquigarrow} C^1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C^k$ where $k \geq 1$, C^0 is the initial R-configuration, and $t^j : \mathbb{Z}_n \to \Delta$ for all $1 \leq j \leq k$. We call k the *length* of χ. Note that χ uniquely determines the underlying ring R.

Remark 1. *A receive command has to be matched by a corresponing send, while a message that has been sent may not necessarily be received. However, adopting alternative semantics, such as non-blocking receives, would not change any of the forthcoming results.* $\lhd$

Example 2. A run of $\mathcal{D}_{\mathsf{LE}}$ from Example 1 on the ring $R = (4, 8, 3, 1, 6, 5, 7)$ is depicted in Figure 3 (for the moment, we may ignore the blue and violet lines). A colored row forms a configuration. The three pids in a cell refer to registers r, r', r'', respectively (we ignore id). Moreover, a non-colored row forms, together with the states above and below, a transition tuple. When looking at the step from C^3 to C^4, we have, for example, $r'@2 \rightarrowtail r@3$ and $r'@2 \rightarrowtail r''@5$. Moreover, $r'@5 \rightarrowtail r@6$ and $r'@5 \rightarrowtail r''@0$ (recall that we are in a ring). Note that the run conforms to the correctness property formulated in Example 1. In particular, in the final configuration, all processes store the maximum pid in register r. $\lhd$

Example 3 (Odd-even transposition sort [27]). We will now consider a distributed sorting algorithm, called odd-even transposition sort, which sorts the elements of an array by parallel compare-and-swaps. We adapt the algorithm to rings in which the process with the highest pid is assumed to be the leader. At the end of the algorithm, the ring will store the pids in a sorted order starting from the leader.

Towards this, every process is equipped with a dedicated register, say r, which can store some pid value (not necessarily its own). Initially, this register contains its own pid. We assume that the ring has a leader, and further that the leader is the process with the highest pid. Once the distributed sorting is finished, the pid stored in the register r of every process (other than the leader) will be higher than the pid stored in register r of its left neighbor. This can be achieved by the odd-even transposition sort, in two phases. In the first phase, the processes will determine whether they are at an odd distance (parity 1) or an even distance (parity 0) from the leader. Every process will send its guessed parity b to the right, and ensures that it receives the dual parity $1 - b$ from its left. The leader will deterministically send *even* to the right, and discards what it receives from the left.

The second phase is the actual sorting phase where processes alternate between their neighbors to perform a compare-and-swap. A process which guessed parity 0 will first send its stored pid to the right, receive a pid from the right, and store the minimum of these two in register r. Next, it will send its stored pid to the left, receive a pid from the left, and store the maximum of these two in register r. Symmetrically, a process which guessed parity 1 will send its stored pid to the left, receive a pid from the left, and store the maximum of these two in its register r. Next, it will send its stored pid to the right, receive a pid from the right, and store the minimum of these two in register r. The leader will only compare and swap with its right neighbor. It tricks its left neighbor to not swap by always sending its own pid (which is the highest pid in the ring) to the left.

Simultaneously to the compare-and-swap rounds, a token is sent to the right neighbor by the leader, and moves one step to the right with every round. Once the token reaches back the leader, the algorithm terminates. Correctness of this distributed sorting algorithm, i.e., the fact that the pids are sorted when the algorithm terminates, is not obvious. $\lhd$

3. From Distributed Algorithms to Propositional Dynamic Logic

In this section, we provide a symbolic abstraction of runs of distributed algorithms. A symbolic abstraction ignores pids while keeping all the other information such as states and transitions.

We will provide a translation of distributed algorithms into propositional dynamic logic with loop and converse (LCPDL) [26] over *finitely labeled cylinders*. The models of an LCPDL formula will be precisely the symbolic runs that give rise to some concrete runs.

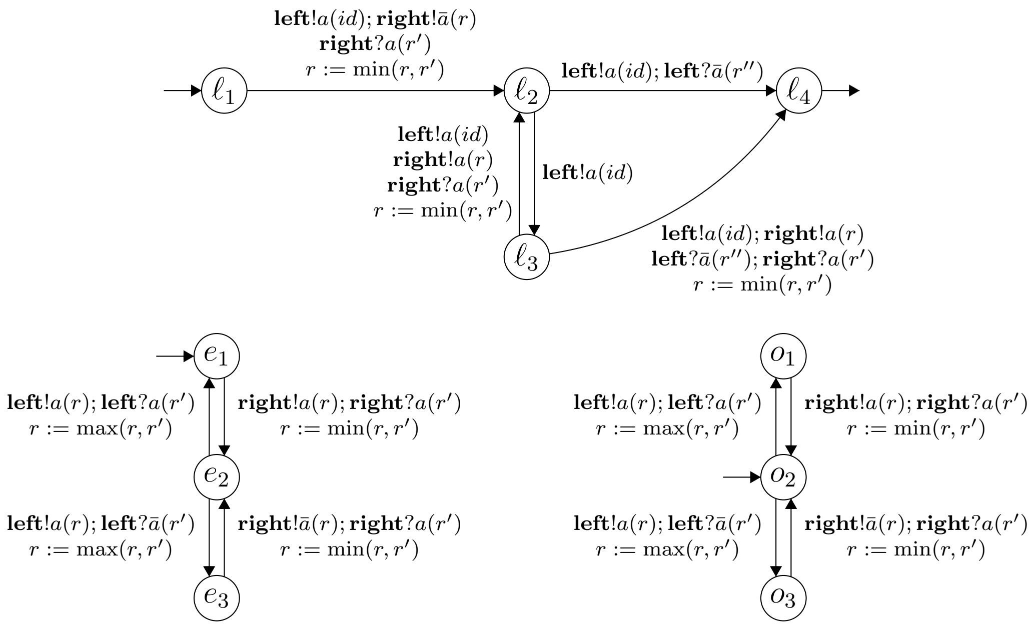

$\mathbf{left}!a(id); \mathbf{right}!\bar{a}(r)$
$\mathbf{right}?a(r')$
$r := \min(r, r')$
ℓ_1
ℓ_2
$\mathbf{left}!a(id); \mathbf{left}?\bar{a}(r'')$
ℓ_4
$\mathbf{left}!a(id)$
$\mathbf{right}!a(r)$
$\mathbf{right}?a(r')$
$r := \min(r, r')$
$\mathbf{left}!a(id)$
ℓ_3
$\mathbf{left}!a(id); \mathbf{right}!a(r)$
$\mathbf{left}?\bar{a}(r''); \mathbf{right}?a(r')$
$r := \min(r, r')$
e_1
$\mathbf{left}!a(r); \mathbf{left}?a(r')$
$r := \max(r, r')$
$\mathbf{right}!a(r); \mathbf{right}?a(r')$
$r := \min(r, r')$
e_2
$\mathbf{left}!a(r); \mathbf{left}?\bar{a}(r')$
$r := \max(r, r')$
$\mathbf{right}!\bar{a}(r); \mathbf{right}?a(r')$
$r := \min(r, r')$
e_3
o_1
$\mathbf{left}!a(r); \mathbf{left}?a(r')$
$r := \max(r, r')$
$\mathbf{right}!a(r); \mathbf{right}?a(r')$
$r := \min(r, r')$
o_2
$\mathbf{left}!a(r); \mathbf{left}?\bar{a}(r')$
$r := \max(r, r')$
$\mathbf{right}!\bar{a}(r); \mathbf{right}?a(r')$
$r := \min(r, r')$
o_3

Figure 4: Distributed sorting algorithm for a ring of size at least 2. In the automata above, messages $\bar{a}$ carry the token, whereas messages a are regular. Moreover, $r := \min(r, r')$ actually stands for two transitions, one with guard $r' < r$ and update $r := r'$ and the other with guard $r \leq r'$ and no update. The situation is symmetric for $r := \max(r, r')$. The top automaton is for the leader. The bottom left (resp. right) automaton is for a process whose distance from the leader is even (resp. odd).

We will first define labeled cylinders and propositional dynamic logic with loop and converse over them.

3.1. PDL with Loop and Converse (LCPDL)

Let Props be a finite set of propositions. A valuation of the propositions is a function $\gamma\colon \textsc{Props} \to \{0,1\}$. For $k \in \mathbb{N}$, we set $[k] = \{0, 1, \ldots, k\}$. An $(n \times k)$-*cylinder* is a mapping $\zeta\colon \mathbb{Z}_n \times [k] \to 2^{\textsc{Props}}$. The set of positions (or coordinates) of an $(n \times k)$-cylinder is $Pos(\zeta) \overset{\text{def}}{=} \mathbb{Z}_n \times [k]$. Thus a cylinder has n columns and $k+1$ rows. Further, the columns are cyclically arranged such that every column has a neighboring column on its right (and its left). A cylinder ζ associates a valuation of the propositions $\gamma = \zeta(x)$ to every position $x = (i, j)$ of the cylinder. Let $\mathbb{Cyl}(\textsc{Props})$ denote the set of all $(n \times k)$-cylinders over Props, for all possible n and k.

Propositional dynamic logic has modalities that allows one to "walk" in a cylinder just like a *tree-walking automaton* walks in a tree. Therefore, apart from *local formulas* which check that a property holds at the *current* position of a cylinder, there will be *path formulas.* The latter allow us to go to the right-/down- neighboring coordinate. We can even describe paths of unbounded length using regular expressions over the basic steps and their converses. There is also a special loop construct that requires a path to describe a cycle in the cylinder (i.e., it starts and ends in one and the same coordinate).

$\zeta \models \mathsf{E}\,\psi$	if	$\zeta, x \models \psi$ for some $x \in Pos(\zeta)$
$\zeta, x \models \ddagger$	if	$x = (0, j)$ for some j
$\zeta, x \models p$	if	$p \in \zeta(x)$
$\zeta, x \models \langle\pi\rangle\psi$	if	$\zeta, x, y \models \pi$ and $\zeta, y \models \psi$ for some $y \in Pos(\zeta)$
$\zeta, x \models \mathsf{loop}(\pi)$	if	$\zeta, x, x \models \pi$
$\zeta, x, y \models \{\psi\}?$	if	$\zeta, x \models \psi$ and $x = y$
$\zeta, x, y \models \rightarrow$	if	$x = (i, j)$ and $y = (i \oplus 1, j)$ for some i, j
$\zeta, x, y \models \downarrow$	if	$x = (i, j)$ and $y = (i, j + 1)$ for some i, j
$\zeta, x, y \models \pi + \pi'$	if	$\zeta, x, y \models \pi$ or $\zeta, x, y \models \pi'$
$\zeta, x, y \models \pi \cdot \pi'$	if	$\zeta, x, z \models \pi$ and $\zeta, z, y \models \pi'$ for some $z \in Pos(\zeta)$
$\zeta, x, y \models \pi^*$	if	there are positions $z_0, \ldots, z_\ell \in Pos(\zeta)$ ($\ell \geq 0$) such that $x = z_0$, $y = z_\ell$ and $\zeta, z_i, z_{i+1} \models \pi$ for all $0 \leq i < \ell$
$\zeta, x, y \models \pi^{-1}$	if	$\zeta, y, x \models \pi$
$\zeta, x, y \models \mathcal{A}$	if	$\zeta, x, y \models \pi_1 \cdot \pi_2 \cdots \pi_\ell$ and there is an accepting path in $\mathcal{A}$: $q_0 \xrightarrow{\pi_1} q_1 \xrightarrow{\pi_2} \cdots \xrightarrow{\pi_\ell} q_\ell$ from its initial state q_0 to some final state q_ℓ.

Table 1: Semantics of LCPDL. $\zeta \in \mathbb{C}\mathrm{yl}(\textsc{Props})$ is an $(n \times k)$-cylinder, $x, y \in Pos(\zeta)$ are positions.

The syntax of formulas in LCPDL(Props) is given as follows:

$$\begin{aligned}
\Psi, \Psi' &::= \mathsf{E}\,\psi \mid \neg\Psi \mid \Psi \wedge \Psi' \\
\psi, \psi' &::= \ddagger \mid p \mid \neg\psi \mid \psi \wedge \psi' \mid \langle\pi\rangle\psi \mid \mathsf{loop}(\pi) \\
\pi, \pi' &::= \{\psi\}? \mid \rightarrow \mid \downarrow \mid \pi + \pi' \mid \pi \cdot \pi' \mid \pi^* \mid \pi^{-1} \mid \mathcal{A}
\end{aligned}$$

where $p \in \textsc{Props}$ and $\mathcal{A}$ is a *path automaton*[1]: a non-deterministic finite automaton whose transitions are labeled with formulas π. We call ψ a *local formula*, π a *path formula*, and Ψ a *sentence*. We include a special marker $\ddagger$ in local formulas which refers to process 0. This marker will not be used in this section but is necessary for specifications (see next section).

A sentence $\Psi \in$ LCPDL(Props) is interpreted over a cylinder ζ over Props. We write $\zeta \models \Psi$ if Ψ is satisfied in ζ. A local formula ψ is interpreted over a position $x \in Pos(\zeta)$. When it is satisfied, we write $\zeta, x \models \psi$. A path formula π depends on two positions $x, y \in Pos(\zeta)$. We write $\zeta, x, y \models \pi$ when we can walk from x to y using a path matching π. The semantics of the boolean operators is as expected, and that of other modalities is defined in Table 1. Finally, a sentence Ψ defines the language $L(\Psi) = \{\zeta \in \mathbb{C}\mathrm{yl}(\textsc{Props}) \mid \zeta \models \Psi\}$.

We use common abbreviations to include disjunction, implication, *true*, and *false*. For path formulas, we let $\leftarrow\, = \,\rightarrow^{-1}$ (go left), $\uparrow\, = \,\downarrow^{-1}$ (go up), $\epsilon = \{\mathit{true}\}?$ (stay on the

[1]We explicitly add path automata for complexity reasons. Indeed, we could replace an automaton by an equivalent rational expression, but the latter could be exponentially larger.

current position), $\pi^+ = \pi \cdot \pi^*$, $[\pi]\psi = \neg\langle\pi\rangle\neg\psi$, and $\langle\pi\rangle = \langle\pi\rangle \mathit{true}$. For sentences, we let $\mathsf{A}\,\psi = \neg\,\mathsf{E}\neg\psi$.

3.2. Translating Distributed Algorithms into LCPDL

We will now describe runs of distributed algorithms using LCPDL. This looks mysterious because we do not have any provision for reasoning about data in LCPDL. On the other hand, runs of distributed algorithm progress by comparing pids and updating registers. Thus, this section demonstrates how to get rid of unbounded data and still reason about the data-dependent decisions taken along the run using LCPDL. This translation depends crucially on the loop modality of LCPDL.

A *symbolic run* of a distributed algorithm on a ring of size n is an $(n \times k)$-cylinder, for some $k \in \mathbb{N}$ that corresponds to the number of rounds. The label of position (i, j) will carry information about which transition was taken by process i in round j. We will have an LCPDL formula that describes such symbolic runs of an algorithm. We will require additional formulas to track the flow of pids between the processes through messages.

Recall that $\mathsf{PropSetOf}(t)$ denotes the set of propositions that occur in a transition t. Let $\mathcal{D} = (S, \mathsf{init}, \Delta, \mu)$ be a distributed algorithm. The set of propositions induced by $\mathcal{D}$ is $\textsc{Props}_{\mathcal{D}} = \bigcup_{s \in S} \mu(s)$. As the LCPDL formula associated with $\mathcal{D}$ will talk about transitions, too, we also set $\textsc{Props}^{\mathrm{tr}}_{\mathcal{D}} = \textsc{Props}_{\mathcal{D}} \cup \bigcup_{t \in \Delta} \mathsf{PropSetOf}(t)$.

For the remainder of this section, we fix a distributed algorithm $\mathcal{D} = (S, \mathsf{init}, \Delta, \mu)$. Wlog., we assume that Δ contains a dummy transition $\mathsf{dt} = \langle \mathsf{s}\text{: } \mathbf{goto}\ \mathsf{init} \rangle$ where s is distinct from init, does not carry any proposition, and does not occur in any other transition.

Let $R\colon \mathbb{Z}_n \to \mathbb{N}$ be a ring and $\chi = C^0 \overset{t^1}{\rightsquigarrow} C^1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C^k$ be an R-run of $\mathcal{D}$, where $t^j\colon \mathbb{Z}_n \to \Delta$ for all $1 \le j \le k$. We simply write t^j_i for $t^j(i)$. From χ, we extract the *symbolic run* $\zeta_\chi \in \mathbb{C}\mathsf{yl}(\textsc{Props}^{\mathrm{tr}}_{\mathcal{D}})$ given by

$$\zeta_\chi(i, j) = \mu(s^j_i) \cup \mathsf{PropSetOf}(t^j_i)$$

for all $(i, j) \in \mathit{Pos}(\zeta_\chi)$, where s^j_i is the (unique) state such that the proposition $\mathbf{goto}\ s^j_i$ is present in $\mathsf{PropSetOf}(t^j_i)$ and $t^0_i = \mathsf{dt}$ is the dummy transition whose purpose is simply to match the number of configurations in a run. For instance, Figure 3 represents a run χ. By keeping only the transition-rows (including, however, states and the associated propositions) and adding on top the row of dummy transitions, we obtain the associated symbolic (7×6)-cylinder ζ_χ. Every distributed algorithm $\mathcal{D}$ gives rise to a set of symbolic runs (cylinders): $\mathbb{C}\mathsf{yl}_{\mathcal{D}} = \{\zeta_\chi \mid \chi \text{ is a run of } \mathcal{D} \text{ on some ring}\}$.

Theorem 1. *From a distributed algorithm $\mathcal{D}$, we can construct in polynomial time a formula $\Psi_{\mathcal{D}} \in \mathrm{LCPDL}(\textsc{Props}^{\mathrm{tr}}_{\mathcal{D}})$ such that*

$$L(\Psi_{\mathcal{D}}) = \mathbb{C}\mathsf{yl}_{\mathcal{D}}\,.$$

In particular, $\Psi_{\mathcal{D}}$ will verify that the table defines a symbolic abstract run (Conditions R_1–R_3), registers in equality guards can be traced back to the same origin (to ensure R_4), and there are no cyclic dependencies that arise from $<$-guards (to ensure R_5). In that case, the symbolic run is consistent and corresponds to a "real" run of $\mathcal{D}$.

We will now explain how to check in LCPDL that a cylinder defines a symbolic run. For each transition $t = \langle \ldots \ \mathbf{goto}\ s\rangle \in \Delta$, it is easy to write a local formula ψ_t checking that the label of the current position is precisely $\mu(s) \cup \mathsf{PropSetOf}(t)$. Then, the local formula $\Psi^1_{\mathsf{run}} = \mathsf{A} \bigvee_{t\in\Delta} \psi_t$ states that all positions of the cylinder are labeled with transitions from Δ. We can also check that row 0 is labeled with the dummy transition and that at least one "real" transition is taken: $\Psi^2_{\mathsf{run}} = \mathsf{A}(\neg\langle\uparrow\rangle \implies (\psi_{\mathsf{dt}} \wedge \langle\downarrow\rangle))$. Next, we make sure that the target state of a transition matches the source state of the transition in the next round, i.e., Condition (R_1): $\Psi^3_{\mathsf{run}} = \mathsf{A}(\neg \bigvee_{s,s'\in S \mid s\neq s'} (\mathbf{goto}\ s) \wedge \langle\downarrow\rangle s')$. In order to check Conditions (R_2–R_3) we introduce path formulas corresponding to the relations $i \overset{a}{\rightarrowtail} j$ and $j \overset{a}{\leftarrowtail} i$ defined in Section 2. For $a \in \mathcal{M}^{(k)}$, we define

$$\overset{a}{\rightarrowtail} = \{\mathbf{right}!a\}? \cdot (\rightarrow \cdot \{\mathbf{fwd\text{-}right}\}?)^* \cdot \rightarrow \cdot \{\mathbf{left}?a\}?$$
$$\overset{a}{\leftarrowtail} = \{\mathbf{left}!a\}? \cdot (\leftarrow \cdot \{\mathbf{fwd\text{-}left}\}?)^* \cdot \leftarrow \cdot \{\mathbf{right}?a\}?$$

with $\mathbf{right}!a = \bigvee \mathbf{right}!a(r_1,\ldots,r_k)$ where the disjunction ranges over all $(r_1,\ldots,r_k)$ such that $\mathbf{right}!a(r_1,\ldots,r_k) \in \textsc{Props}^{\mathrm{tr}}_{\mathcal{D}}$, and similarly for $\mathbf{left}?a$, $\mathbf{left}!a$ and $\mathbf{right}?a$. Clearly, we have $\zeta,(i,\ell),(j,\ell) \models \overset{a}{\rightarrowtail}$ iff during round ℓ message a is sent to the right by the ith process and received from the left by the jth process. Then, Condition (R_2) can be checked with $\Psi^4_{\mathsf{run}} = \mathsf{A}(\mathbf{left}?a \implies \langle \overset{a}{\rightarrowtail}^{-1}\rangle)$. Condition ($\mathsf{R}_3$) can be checked with a similar formula Ψ^5_{run}. Finally, we let $\Psi_{\mathsf{run}} = \Psi^1_{\mathsf{run}} \wedge \Psi^2_{\mathsf{run}} \wedge \Psi^3_{\mathsf{run}} \wedge \Psi^4_{\mathsf{run}} \wedge \Psi^5_{\mathsf{run}}$.

Lemma 1. *Let ζ be a cylinder. Then, $\zeta \models \Psi_{\mathsf{run}}$ iff ζ describes a symbolic abstract run satisfying* (R_1–R_3).

Now, we have to check that guards are satisfied, i.e., Conditions (R_4–R_5). This is the most difficult part. Every register contains the identifier of some process taking part in the run. The idea is to write, for each register r, a path formula π_r that describes the transmission of pids in a symbolic run, up to a point where they are checked in some guard. A path satisfying π_r walks from some initial position $(i,0)$ to some position (i',j') iff the identity of process i is contained in register r of process i' after the message exchanges of round j', i.e., when guards are checked in round j'. The path formula π_r will only focus on transmission of register values through updates and message exchanges. It does not check whether guards are satisfied along the way.

In Section 3.3 below, we will explain how the path formulas π_r are constructed. But first we describe how they are used to check that guards are satisfied. For equality guards, using the expressive power of loop and converse, we define the LCPDL formula

$$\Psi_{=} \;=\; \mathsf{A} \bigwedge\nolimits_{r,r'\in\textsc{Regs}_{\mathcal{D}}} \Big(r = r' \implies \mathsf{loop}(\pi_r^{-1} \cdot \pi_{r'})\Big)\,.$$

It says that, whenever an equality check $r = r'$ occurs in the symbolic run, then the pids stored in r and r' have a common origin. The next path formula connects the first coordinate of a process i with the first coordinate of another process i' if some guards force the pid of i to be smaller than that of i':

$$\pi_< \;=\; \Big(\textstyle\sum_{r,r' \in \textsc{Regs}_\mathcal{D}} \pi_r \cdot \{r < r'\}? \cdot \pi_{r'}^{-1}\Big)^+ .$$

Note that, here, we use the (strict) transitive closure. Consistency of $<$-guards now reduces to saying that there is no $\pi_<$-loop:

$$\Psi_< \;=\; \mathsf{A}(\neg\langle\uparrow\rangle \implies \neg\mathsf{loop}(\pi_<))\,.$$

Finally, the formula announced in Theorem 1 is defined as

$$\Psi_\mathcal{D} \;=\; \Psi_{\mathsf{run}} \wedge \Psi_= \wedge \Psi_<\,.$$

In the next subsection, we will define the path formulas π_r and show that they conform to the semantics given above. Then, in Section 3.4, we prove that the formula $\Psi_\mathcal{D}$ defined above satisfies the statement of Theorem 1.

3.3. The path formulas π_r

First, we introduce a local formula $\mathtt{upd}_{r,r'}$ which states that the current transition updates register r' with the contents of register r.

$$\mathtt{upd}_{r,r'} = \begin{cases} (r' := r) & \text{if } r' \neq r \\ (r := r) \vee \neg \bigvee_{r''}(r := r'') & \text{if } r' = r\,. \end{cases}$$

Note that, when $r' = r$, the formula also takes care of the absence of an update command for the register.

Next, we define a path formula $\mathtt{rightmsg}_{r,r'}$. It describes paths across several processes which relate the sending of r to the right and a corresponding receive in r' from the left, requiring that messages are forwarded by all intermediate processes. This corresponds to the relation $r@i \rightarrowtail r'@i'$ defined in Section 2. Similarly, we define a path formula $\mathtt{leftmsg}_{r,r'}$ corresponding to the relation $r'@i' \leftarrowtail r@i$. We also define $\mathtt{nomsg}_r$ which states that no received messages update register r. Formally, these formulas are defined by

$$\begin{aligned}
\mathtt{rightmsg}_{r,r'} &= \textstyle\sum_{(\alpha,\alpha')\in \mathtt{Mg}(r,r')} \quad \{\mathbf{right}!\alpha\}? \cdot (\rightarrow \cdot \{\mathbf{fwd\text{-}right}\}?)^* \cdot \rightarrow \cdot \{\mathbf{left}?\alpha'\}? \\
\mathtt{leftmsg}_{r,r'} &= \textstyle\sum_{(\alpha,\alpha')\in \mathtt{Mg}(r,r')} \quad \{\mathbf{left}!\alpha\}? \cdot (\leftarrow \cdot \{\mathbf{fwd\text{-}left}\}?)^* \cdot \leftarrow \cdot \{\mathbf{right}?\alpha'\}? \\
\mathtt{nomsg}_r &= \{\neg \textstyle\bigvee_{\alpha \in \mathtt{Rc}(r)} \mathbf{left}?\alpha \vee \mathbf{right}?\alpha\}?
\end{aligned}$$

where $\mathtt{Mg}(r,r')$ is the finite set of pairs (α,α') of messages occurring in $\mathcal{D}$ and which may transmit r to r': $\alpha = a(r_1,\ldots,r_k)$, $\alpha' = a(r'_1,\ldots,r'_k)$ with $r_\ell = r$ and $r'_\ell = r'$ for some $1 \leq \ell \leq k$ and $\mathbf{right}!\alpha, \mathbf{left}?\alpha' \in \textsc{Props}^{\mathrm{tr}}_\mathcal{D}$ or $\mathbf{left}!\alpha, \mathbf{right}?\alpha' \in \textsc{Props}^{\mathrm{tr}}_\mathcal{D}$. Also, $\mathtt{Rc}(r)$ is the set of receive messages α of $\mathcal{D}$ in which register r occurs: $\alpha = a(r_1,\ldots,r_k)$ with $r = r_\ell$ for some $1 \leq \ell \leq k$ and $\mathbf{left}?\alpha \in \textsc{Props}^{\mathrm{tr}}_\mathcal{D}$ or $\mathbf{right}?\alpha \in \textsc{Props}^{\mathrm{tr}}_\mathcal{D}$.

We can easily check that $\zeta, (i,j), (i',j') \models \texttt{rightmsg}_{r,r'}$ (resp. $\texttt{leftmsg}_{r,r'}$) iff $j' = j$ (same round) and $r@i \rightarrowtail r'@i'$ (resp. $r'@i' \leftarrowtail r@i$). Then, register updates through messages are described by the path formula

$$\texttt{msg}_{r,r'} = \begin{cases} \texttt{rightmsg}_{r,r'} + \texttt{leftmsg}_{r,r'} & \text{if } r' \neq r \\ \texttt{rightmsg}_{r,r} + \texttt{leftmsg}_{r,r} + \texttt{nomsg}_{r} & \text{if } r' = r\,. \end{cases}$$

It follows that $\zeta, (i,j), (i',j') \models \texttt{msg}_{r,r'}$ iff $j' = j$ (same round) and register r' of process i' *after the exchange of messages of the round* contains the value that was in register r of process i *before* the round.

Now we can define the main path formula:

$$\texttt{round}_{r,r'} = \textstyle\sum_{r'' \in \textsc{Regs}_{\mathcal{D}}} \texttt{msg}_{r,r''} \cdot \{\texttt{upd}_{r'',r'}\}?$$

We deduce that $\zeta, (i,j), (i',j') \models \texttt{round}_{r,r'}$ iff $j' = j$ (same round) and register r' of process i' *after* round j contains the value of register r of process i *before* round j.

Now, remember that our goal was to write, for each register r, a path formula π_r which walks from some initial position $(i,0)$ to some position (i',j') iff the identity of process i is contained in register r of process i' when guards are checked in round j'. We are actually almost done. Basically, π_r should describe a path of the form

$$\downarrow \cdot\, \texttt{round}_{r_1,r_2} \cdot \downarrow \cdot\, \texttt{round}_{r_2,r_3} \cdots \downarrow \cdot\, \texttt{round}_{r_{j'-1},r_{j'}} \cdot \downarrow \cdot\, \texttt{msg}_{r_{j'},r}\,.$$

This is a kind of iteration, but we have to ensure a correct flow through the intermediary registers. More precisely, the expression $(\sum_{r_1,r_2} \downarrow \cdot \text{round}_{r_1,r_2})^*$ is not correct for the iteration since the 'output' register of one iteration may be different from the 'input' register of the next iteration. This is where using a path automaton $\mathcal{A}$ will be convenient.[2] The state of $\mathcal{A}$ remembers the register between two iterations. For instance, Figure 5 gives the path automaton $\mathcal{A}$ when we only have two registers: $\textsc{Regs}_{\mathcal{D}} = \{r_1, r_2\}$.

Formally, the set of states of $\mathcal{A}$ is $\{\iota\} \uplus \textsc{Regs}_{\mathcal{D}}$ and its initial state is ι. We have two types of transitions. For all $r, r' \in \textsc{Regs}_{\mathcal{D}}$:

- $(\iota, \{\neg\langle\uparrow\rangle\}?, r)$: transitions from the initial state check that the path starts at the top row and non-deterministically chooses some register r, which is initialized with the pid of the current process.
- $(r, \downarrow \cdot\, \texttt{round}_{r,r'}, r')$: these transitions go to the next round and check that the pid contained in register r is transmitted to register r' during that round.

[2]Indeed we can write a rational expression which is equivalent to the path automaton, but the size of this rational expression would be exponential in the number of registers, whereas the size of $\mathcal{A}$ is only polynomial in the number of registers.

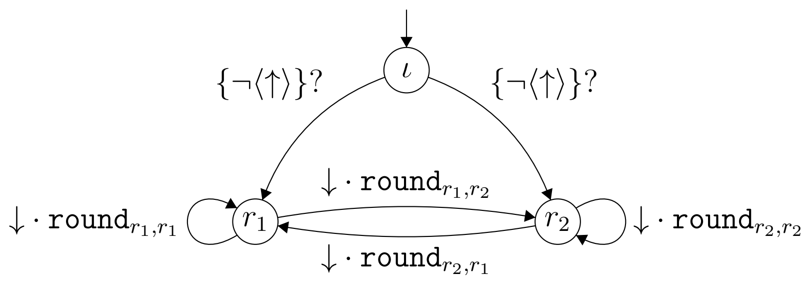

$\{\neg\langle\uparrow\rangle\}?$
ι
$\{\neg\langle\uparrow\rangle\}?$
$\downarrow\cdot\,\texttt{round}_{r_1,r_2}$
$\downarrow\cdot\,\texttt{round}_{r_1,r_1}$
r_1
r_2
$\downarrow\cdot\,\texttt{round}_{r_2,r_2}$
$\downarrow\cdot\,\texttt{round}_{r_2,r_1}$

Figure 5: Path automaton $\mathcal{A}$ when $\textsc{Regs}_{\mathcal{D}} = \{r_1, r_2\}$.

We denote by $\mathcal{A}_r$ the path automaton $\mathcal{A}$ with single final state r. We show below (Lemma 2) that an accepting run of $\mathcal{A}_r$ describes a path expression which connects some initial position $(i, 0)$ to some position (i', j') iff the pid of process i is contained in register r of process i' after round j'. Figure 3 describes (partial) paths (illustrated by the blue and violet lines, respectively) satisfying $\pi_{r'}$ and $\pi_{r''}$ which allow us to identify the origin of r' and r'' when applying the guard $r' < r''$.

Finally, the path formula π_r is simply given by

$$\pi_r = \textstyle\sum_{r' \in \textsc{Regs}_{\mathcal{D}}} \mathcal{A}_{r'} \cdot \downarrow \cdot\, \texttt{msg}_{r',r}\,.$$

To formally state and prove Lemma 2, we introduce some notation. A *pseudo* (*R*-)*run* of $\mathcal{D}$ is like an (R-)run $\chi = C^0 \overset{t^1}{\rightsquigarrow} C^1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C^k$, but conditions ($\mathsf{R}_4$–$\mathsf{R}_5$) are not checked. That is, $=$-guards and $<$-guards are ignored. Thus, every run is a pseudo run, but not vice versa. We define ζ_χ in exactly the same way as for runs.

Given a (pseudo) run $\chi = C^0 \overset{t^1}{\rightsquigarrow} C^1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C^k$ of $\mathcal{D}$ (where $C^j = (s^j, \rho^j)$) and $(i, j) \in Pos(\zeta_\chi)$, we set $\chi_i^j = \rho^j(i)$. Moreover, for $j \geq 1$, $\overline{\chi}_i^j = \hat{\rho}^{j-1}(i)$ defines the corresponding j-th intermediate register assignment, which was defined in Section 2.

Lemma 2. *For all pseudo runs χ of $\mathcal{D}$, positions $(i, j), (i', j') \in Pos(\zeta_\chi)$, and registers $r \in \textsc{Regs}_{\mathcal{D}}$, the following hold (recall that* id *is the register holding the pid of the respective process):*

$$\zeta_\chi, (i, j), (i', j') \models \mathcal{A}_r \iff \big(\chi_{i'}^{j'}(r) = \chi_i^0(\mathsf{id}) \wedge j = 0\big) \tag{1}$$
$$\zeta_\chi, (i, j), (i', j') \models \pi_r \iff \big(\overline{\chi}_{i'}^{j'}(r) = \chi_i^0(\mathsf{id}) \wedge j = 0 \wedge j' > 0\big) \tag{2}$$

Proof. Let the pseudo R-run in question be given by $\chi = C^0 \overset{t^1}{\rightsquigarrow} C^1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C^k$.

From the definitions of the formulas $\texttt{upd}_{r,r'}$ and $\texttt{msg}_{r,r'}$, it is easy to see that, for all positions $(i, j), (i', j') \in Pos(\zeta_\chi)$ with $j \geq 1$ and registers $r, r' \in \textsc{Regs}_{\mathcal{D}}$, the following hold:

$$\zeta_\chi, (i, j), (i', j') \models \texttt{msg}_{r,r'} \iff \big(\overline{\chi}_{i'}^{j'}(r') = \chi_i^{j-1}(r) \wedge j' = j\big) \tag{3}$$
$$\zeta_\chi, (i, j), (i', j') \models \texttt{upd}_{r,r'} \iff \big(\chi_{i'}^{j'}(r') = \overline{\chi}_i^{j}(r) \wedge j' = j \wedge i' = i\big)$$

We deduce that

$$\zeta_\chi, (i,j), (i',j') \models \mathtt{round}_{r,r'} \iff \left(\chi_{i'}^{j'}(r') = \chi_i^{j-1}(r) \wedge j' = j\right)$$

Now, $\zeta_\chi, (i,j) \models \neg\langle\uparrow\rangle$ iff $j = 0$ and $\chi_i^0(r) = \chi_i^0(\mathsf{id})$ for all $r \in \textsc{Regs}_\mathcal{D}$. We deduce that (1) holds. Using (3) we obtain (2). □

3.4. Proof of Theorem 1

Recall that $\mathbb{C}\mathrm{yl}_\mathcal{D} = \{\zeta_\chi \mid \chi \text{ is a run of } \mathcal{D}\}$. For a cylinder $\zeta \in \mathbb{C}\mathrm{yl}(\textsc{Props}_\mathcal{D}^{\mathrm{tr}})$, let $\mathsf{Runs}(\zeta) = \{\chi \mid \chi \text{ is run of } \mathcal{D} \text{ such that } \zeta_\chi = \zeta\}$. Note that $\mathsf{Runs}(\zeta) \neq \emptyset$ for all $\zeta \in \mathbb{C}\mathrm{yl}_\mathcal{D}$.

Lemma 3. *For all $\zeta \in \mathbb{C}\mathrm{yl}_\mathcal{D}$ and $(i,0), (i',0) \in Pos(\zeta)$, we have*

$$\zeta, (i,0), (i',0) \models \pi_< \iff \forall\chi \in \mathsf{Runs}(\zeta) : \chi_i^0(\mathsf{id}) < \chi_{i'}^0(\mathsf{id})\,.$$

Proof. Assume that $\zeta \in \mathbb{C}\mathrm{yl}_\mathcal{D}$ is an $(n \times k)$-cylinder. Note that run conditions (R_1–R_3) are satisfied, since $\zeta \in \mathbb{C}\mathrm{yl}_\mathcal{D}$. There are two directions to show.

($\Rightarrow$): Suppose $\zeta, (i,0), (i',0) \models \pi_<$. Then, there are $\ell > 0$ and $i = i_0, \ldots, i_\ell = i'$ such that

$$\zeta, (i_{m-1},0), (i_m,0) \models \sum_{r,r' \in \textsc{Regs}_\mathcal{D}} \pi_r \cdot \{r < r'\}? \cdot \pi_{r'}^{-1}$$

for all $m \in \{1,\ldots,\ell\}$. Let $\chi \in \mathsf{Runs}(\zeta)$. By Lemma 2, we have $\chi_{i_{m-1}}^0(\mathsf{id}) < \chi_{i_m}^0(\mathsf{id})$ for all $m \in \{1,\ldots,\ell\}$. We deduce $\chi_i^0(\mathsf{id}) < \chi_{i'}^0(\mathsf{id})$.

($\Leftarrow$): We denote the processes in question by u and u'. Suppose that $\zeta, (u,0), (u',0) \not\models \pi_<$. We are going to show that there is $\chi \in \mathsf{Runs}(\zeta)$ such that $\chi_u^0(\mathsf{id}) \geq \chi_{u'}^0(\mathsf{id})$. Let $\prec = \{(i,i') \mid \zeta, (i,0), (i',0) \models \pi_<\}$. In particular, $u \not\prec u'$. By direction ($\Rightarrow$) and using $\mathsf{Runs}(\zeta) \neq \emptyset$, we have that $\prec$ is a *strict* partial order.

Let R be any ring such that (i) $R(u) \geq R(u')$ and (ii) for all $i, i' \in \mathbb{Z}_n$, $i \prec i'$ implies $R(i) < R(i')$. Since $\prec$ is a strict partial order and $u \not\prec u'$, such a ring must exist. Now, note that there is a unique *pseudo* R-run

$$\chi = C^0 \overset{t^1}{\rightsquigarrow} C^1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C^k$$

such that $\zeta_\chi = \zeta$. We will show that χ is indeed also an R-run, which concludes the proof.

Let $(i,j) \in Pos(\zeta)$ with $j \geq 1$ and $r, r' \in \textsc{Regs}_\mathcal{D}$ such that $(r < r') \in \mathsf{PropSetOf}(t_i^j)$. We have to show that $\overline{\chi}_i^j(r) < \overline{\chi}_i^j(r')$. By Lemma 2, there are $\ell, \ell' \in \mathbb{Z}_n$ such that

- $\chi_\ell^0(\mathsf{id}) = \overline{\chi}_i^j(r)$ and $\chi_{\ell'}^0(\mathsf{id}) = \overline{\chi}_i^j(r')$, and
- $\zeta_\chi, (\ell,0), (i,j) \models \pi_r$ and $\zeta_\chi, (\ell',0), (i,j) \models \pi_{r'}$.

The latter implies $\zeta_\chi, (\ell,0), (\ell',0) \models \pi_r \cdot \{r < r'\}? \cdot \pi_{r'}^{-1}$. In particular, $\zeta_\chi, (\ell,0), (\ell',0) \models \pi_<$. We deduce $\ell \prec \ell'$. This implies $\chi_\ell^0(\mathsf{id}) < \chi_{\ell'}^0(\mathsf{id})$. We conclude that $\overline{\chi}_i^j(r) < \overline{\chi}_i^j(r')$.

Finally, let $(i,j) \in Pos(\zeta)$ with $j \geq 1$ and $r, r' \in \textsc{Regs}_\mathcal{D}$ such that $(r = r') \in \mathsf{PropSetOf}(t_i^j)$. Since $\mathsf{Runs}(\zeta) \neq \emptyset$, there is a run that validates guard $r = r'$ at coordinate (i,j). By Lemma 2, this is actually true for all pseudo runs of ζ. We deduce that $\overline{\chi}_i^j(r) = \overline{\chi}_i^j(r')$. □

Proof (of Theorem 1). We have to show $L(\Psi_{\mathcal{D}}) = \mathbb{C}\mathsf{yl}_{\mathcal{D}} = \{\zeta_\chi \mid \chi \text{ is a run of } \mathcal{D}\}$, where $\Psi_{\mathcal{D}}$ is given as $\Psi_{\mathsf{run}} \wedge \Psi_{=} \wedge \Psi_{<}$.

($\subseteq$): Let ζ be an $(n \times k)$-cylinder such that $\zeta \models \Psi_{\mathcal{D}}$. We will show $\zeta \in \mathbb{C}\mathsf{yl}_{\mathcal{D}}$ by constructing a run χ of $\mathcal{D}$ such that $\zeta_\chi = \zeta$.

As in the proof of Lemma 3, let $\prec = \{(i, i') \in Pos(\zeta) \mid \zeta, (i,0), (i',0) \models \pi_<\}$. As $\zeta \models \Psi_<$ we get $\zeta, (i,0) \models \neg\mathsf{loop}(\pi_<)$ for all $i \in \mathbb{Z}_n$. We deduce that $\prec$ is a strict partial order. Choose any ring R such that, for all $i, i' \in \mathbb{Z}_n$, $i \prec i'$ implies $R(i) < R(i')$. There is a unique pseudo R-run

$$\chi = C^0 \overset{t^1}{\rightsquigarrow} C^1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C^k$$

of $\mathcal{D}$ such that $\zeta_\chi = \zeta$. Let $1 \leq j \leq k$. We have to show $C^{j-1} \overset{t^j}{\rightsquigarrow} C^j$ where, this time, all run conditions (R_1–R_6) are checked. Condition (R_6) of the definition of $\rightsquigarrow$ is satisfied thanks to the definition of a pseudo run. Conditions (R_1–R_3) are ensured by $\zeta \models \Psi_{\mathsf{run}}$ and Lemma 1. Let $i \in \mathbb{Z}_n$ and suppose $(r = r') \in \mathsf{PropSetOf}(t_i^j)$. We have $\zeta, (i,j) \models \mathsf{loop}(\pi_r^{-1} \cdot \pi_{r'})$. By Lemma 2, we have $\overline{\chi}_i^j(r) = \overline{\chi}_i^j(r')$. Finally, suppose $(r < r') \in \mathsf{PropSetOf}(t_i^j)$. We proceed like in the reverse direction of the proof of Lemma 3 to show that $\overline{\chi}_i^j(r) < \overline{\chi}_i^j(r')$.

Altogether, it follows that χ is a run.

($\supseteq$): Let ζ be an $(n \times k)$-cylinder such that $\zeta \not\models \Psi_{\mathcal{D}}$. To show $\zeta \notin \mathbb{C}\mathsf{yl}_{\mathcal{D}}$, we distinguish three (non-disjoint) cases.

- Suppose $\zeta \not\models \Psi_{\mathsf{run}}$. By Lemma 1, this implies $\zeta \notin \mathbb{C}\mathsf{yl}_{\mathcal{D}}$.
- Suppose $\zeta \not\models \Psi_=$. Recall that $\Psi_= = \mathsf{A} \bigwedge_{r,r' \in \textsc{Regs}_{\mathcal{D}}} \Big(r = r' \implies \mathsf{loop}(\pi_r^{-1} \cdot \pi_{r'}) \Big)$. Thus, there are a coordinate $(i,j) \in Pos(\zeta)$ and registers $r_1, r_2 \in \textsc{Regs}_{\mathcal{D}}$ such that we have $(r_1 = r_2) \in \mathsf{PropSetOf}(t_i^j)$ (hence $j \geq 1$) and $\zeta, (i,j) \not\models \mathsf{loop}(\pi_{r_1}^{-1} \cdot \pi_{r_2})$. Towards a contradiction, suppose there is $\chi \in \mathsf{Runs}(\zeta)$. By Lemma 2, there are (unique) $i_1, i_2 \in \mathbb{Z}_n$ such that $\chi_{i_1}^0(\mathsf{id}) = \overline{\chi}_i^j(r_1)$ and $\chi_{i_2}^0(\mathsf{id}) = \overline{\chi}_i^j(r_2)$, as well as $\zeta, (i_1, 0), (i,j) \models \pi_{r_1}$ and $\zeta, (i_2, 0), (i,j) \models \pi_{r_2}$. Since $\zeta, (i,j) \not\models \mathsf{loop}(\pi_{r_1}^{-1} \cdot \pi_{r_2})$, we have that $i_1 \neq i_2$. We deduce $\overline{\chi}_i^j(r_1) \neq \overline{\chi}_i^j(r_2)$, which contradicts $\chi \in \mathsf{Runs}(\zeta)$. Altogether, we obtain $\zeta \notin \mathbb{C}\mathsf{yl}_{\mathcal{D}}$.
- Suppose $\zeta \not\models \Psi_<$. Then, there is $i \in \mathbb{Z}_n$ such that $\zeta, (i,0) \models \mathsf{loop}(\pi_<)$. By Lemma 3, we have $\chi_i^0(\mathsf{id}) < \chi_i^0(\mathsf{id})$ for all runs $\chi \in \mathsf{Runs}(\zeta)$. Thus, $\mathsf{Runs}(\zeta) = \emptyset$ and, therefore, $\zeta \notin \mathbb{C}\mathsf{yl}_{\mathcal{D}}$. $\square$

Remark 1. Note that, in the translation from distributed algorithms to LCPDL, the set of propositions include the propositions that arise from the transitions and also $\mathcal{P}$. The latter, however, are only used in the formulas to form the link with the specification. More specifically, the specification is independent of the details of the algorithm such as the states, or the registers it uses or the number of transitions it has. The specification in DataPDL only talks about a pre-agreed set of propositions from $\mathcal{P}$, and a pre-agreed set of registers. The actual algorithm may use registers. The LCPDL formulas to which both the algorithm and the specification is translated, will include the propositions that arise from both sides. The mapping μ is employed to link the symbolic runs labeled by transition-generated propositions to the propositions from the specification.

4. The Specification Language

In Example 1, we informally stated a correctness criterion for the presented algorithm (e.g., "at the end, there is one leader and all processes store the pid of the leader in register r"). Now, we introduce a *formal* language to specify correctness properties.

While LCPDL provides good expressive power over abstract runs, it is not sufficient for specifying requirements of distributed algorithms. The specification must be able to reason about the process identities and make comparisons between them. For instance, it may need to say that the process with the maximum pid is elected as the leader.

Toward this purpose, we will first extend cylinders to data cylinders, where each position carries some register valuation also in addition to the valuation of the propositions. Then we extend PDL to reason about the values stored in registers, thus obtaining DataPDL.

However, it poses a problem with expressing some properties – like, there exist at least two nodes satisfying a particular proposition in the same row. We may "get lost" by moving cyclically to the right indefinitely without realizing whether all the nodes in the current row have been visited.

As a solution to this problem we provide a special proposition $\ddagger$ to act as a reference point in the ring. We let the proposition $\ddagger$ be true at, and only at, the process 0 (process $0 \in \mathbb{Z}_n$ may have arbitrary pid assigned to it). Notice that this is available only to the specification; the distributed algorithm is oblivious to which process was process 0. We will now see the formal definitions of data cylinders and DataPDL over them.

4.1. Data cylinders and DataPDL

Let $\textsc{Props}$ be a finite set of propositions and let $\textsc{Regs}$ be a finite set of registers. An $(n \times k)$-*data-cylinder* over $\textsc{Props}$ and $\textsc{Regs}$ is a mapping $\xi\colon \mathbb{Z}_n \times [k] \to 2^{\textsc{Props}} \times \mathbb{N}^{\textsc{Regs}}$. The set of positions (or coordinates) of an $(n \times k)$-cylinder ξ is $Pos(\xi) \overset{\text{def}}{=} \mathbb{Z}_n \times [k]$. That is, ξ labels a position x of a cylinder with both a valuation of the propositions $\gamma\colon \textsc{Props} \to \{0,1\}$, and a valuation of the registers $\nu\colon \textsc{Regs} \to \mathbb{N}$. Suppose $\xi(x) = (\gamma, \nu)$. Abusing notations we write $p \in \xi(x)$ for $\gamma(p) = 1$, and $\xi(x)(r)$ for $\nu(r)$.

Let $\mathbb{DCyl}(\textsc{Props}, \textsc{Regs})$ denote the set of all $(n \times k)$-data-cylinders over $\textsc{Props}$ and $\textsc{Regs}$, for all possible n and k.

We will now present a data-aware navigational logic DataPDL that walks through the positions of a data cylinder using path formulas π as in the case of LCPDL. At a given position of the data cylinders, we can check *local* (or *positional*) properties like a proposition from $\textsc{Props}$, or whether we are on the marked column $\ddagger$. Such a property can be combined with a regular path expression π like in the case of LCPDL.

The most interesting construct in our logic is $\langle\pi\rangle r \bowtie \langle\pi'\rangle r'$, where $\bowtie \in \{=, \neq\}$, which has been used for reasoning about XML documents [10, 11, 20]. It says that, from the current position, there are a π-path and a π'-path that lead to positions y and y', respectively, such that the value stored in register r at y and the value stored in r' at y' satisfy the relation $\bowtie$. Later in the section, we will also extend the logic with register comparisons $<$ or $\leq$.

We will now introduce a (first) logic to reason about data cylinders. It is parametrized by finite sets $\textsc{Props}$ and $\textsc{Regs}$.

Definition 2. The logic DataPDL(Props, Regs) is given by the following grammar:

$$\begin{aligned}\Phi, \Phi' &::= \mathsf{E}\,\varphi \mid \neg\Phi \mid \Phi \wedge \Phi' \\ \varphi, \varphi' &::= \ddagger \mid p \mid \neg\varphi \mid \varphi \wedge \varphi' \mid [\pi]\varphi \mid \langle\pi\rangle r \bowtie \langle\pi'\rangle r' \\ \pi, \pi' &::= \{\varphi\}? \mid \rightarrow \mid \downarrow \mid \pi + \pi' \mid \pi \cdot \pi' \mid \pi^* \mid \pi^{-1}\end{aligned}$$

where $p \in$ Props, $r, r' \in$ Regs, and $\bowtie \in \{=, \neq\}$. ◁

We call Φ a sentence, φ a local formula, and π a path formula. A sentence is interpreted on a data cylinder; a local formula is interpreted on a position; and a path formula on a pair of positions of a data cylinder. Let ξ be a $(n \times k)$-data cylinder and x, y be positions of ξ. As in the case of LCPDL, we write $\xi \models \Phi$, $\xi, x \models \varphi$, and $\xi, x, y \models \pi$ to indicate whether the model satisfies the formula. The semantics is as in the case of LCPDL (cf. Table 1) for the operators that also appear in LCPDL syntax. Further we have

- $\xi, x \models p$ if $p \in \xi(x)$.
- $\xi, x \models [\pi]\varphi$ if, for all y such that $\xi, x, y \models \pi$, we have $\xi, y \models \varphi$
- $\xi, x \models \langle\pi\rangle r \bowtie \langle\pi'\rangle r'$ (where $\bowtie \in \{=, \neq\}$) if there exist x_1, x_2 such that $\xi, x, x_1 \models \pi$ and $\xi, x, x_2 \models \pi'$ and $\xi(x_1)(r) \bowtie \xi(x_2)(r')$.

Once again, we use common abbreviations as in the case of LCPDL. Notice that the logic DataPDL does not have the loop modality of the logic LCPDL.

Data cylinders of distributed algorithms Let $\mathcal{D} = (S, \mathsf{init}, \Delta, \mu)$ be a distributed algorithm. Recall that $\text{Props}_{\mathcal{D}} = \bigcup_{s \in S} \mu(s)$ and that $\text{Regs}_{\mathcal{D}}$ is the set of registers that occur in $\mathcal{D}$, including id. Let $R\colon \mathbb{Z}_n \to \mathbb{N}$ be a ring of size n. Consider an R-run of $\mathcal{D}$

$$\chi = C^0 \overset{t^1}{\rightsquigarrow} C^1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C^k$$

where $C^j = (s^j, \rho^j)$. This run gives rise to the $(n \times k)$-data cylinder ξ_χ over $\text{Props}_{\mathcal{D}}$ and $\text{Regs}_{\mathcal{D}}$ defined by $\xi_\chi(i, j) = (\mu(s^j(i)), \rho^j(i))$. Every distributed algorithm $\mathcal{D}$ gives rise to a set of data cylinders: $\mathbb{DC}\mathsf{y}\mathbb{l}_{\mathcal{D}} = \{\xi_\chi \mid \chi \text{ is a run of } \mathcal{D} \text{ on some ring}\}$.

Given a distributed algorithm $\mathcal{D}$ and a sentence $\Phi \in \text{DataPDL}(\text{Props}_{\mathcal{D}}, \text{Regs}_{\mathcal{D}})$, we write $\mathcal{D} \models \Phi$ if $\xi \models \Phi$ for every $\xi \in \mathbb{DC}\mathsf{y}\mathbb{l}_{\mathcal{D}}$.

Note that our logic allows one to simulate temporal operators (in terms of $\downarrow$) as well as quantification over processes (in terms of $\rightarrow$) as used in the *indexed temporal logics* from [17]. For example, $\neg[\downarrow^*]\neg\varphi$ means that the "current" process eventually satisfies φ (which corresponds to the future modality in LTL), and $[\rightarrow^*]\varphi$ says that every process satisfies φ.

4.2. The model checking problem

Given a distributed algorithm and a specification written in DataPDL, the model checking problem asks whether all runs of the distributed algorithm satisfy the specification:

Input:	A distributed algorithm $\mathcal{D}$, a sentence $\Phi \in \text{DataPDL}(\text{Props}_{\mathcal{D}}, \text{Regs}_{\mathcal{D}})$
Question:	Is it the case that $\mathcal{D} \models \Phi$?

Example 4. Let us *formalize*, in DataPDL, the correctness criteria for $\mathcal{D}_{\mathsf{LE}}$ that we stated informally in Example 1. Consider the following local formulas (again, propositions are chosen to be the states themselves):

$$\varphi_{\mathsf{acc}} = [\rightarrow^*](\mathit{passive} \vee \mathit{found})$$
$$\varphi_{\mathsf{found}} = \langle \pi_{\mathsf{found}} \rightarrow (\{\neg \mathit{found}\}?\rightarrow)^* \rangle \ddagger \qquad \varphi_{r=r} = \neg\big(\langle \epsilon \rangle r \neq \langle \rightarrow^* \rangle r\big)$$

where $\pi_{\mathsf{found}} = (\{\neg \mathit{found}\}?\rightarrow)^*\{\mathit{found}\}?$. Consider the DataPDL formulas

$$\Phi_1 = \mathsf{A}\,\big((\ddagger \wedge \varphi_{\mathsf{acc}}) \implies (\varphi_{\mathsf{found}} \wedge \varphi_{r=r})\big)$$

These formulas say that, after reaching accepting states, expressed by $(\ddagger \wedge \varphi_{\mathsf{acc}}) \implies \cdots$, we have that

(i) there is exactly one process that ends in state *found* (guaranteed by φ_{found}),

(ii) all processes store the same pid in r ($\varphi_{r=r}$),

Thus, $\mathcal{D}_{\mathsf{LE}} \models \Phi_1$. $\triangleleft$

It is unsurprising that model checking distributed algorithms against DataPDL is undecidable when the number of rounds is *unbounded*:

Theorem 2. *The following problem is undecidable: Given a distributed algorithm $\mathcal{D}$ and $\Phi \in \mathrm{DataPDL}(\mathrm{P{\scriptsize ROPS}}_{\mathcal{D}}, \mathrm{R{\scriptsize EGS}}_{\mathcal{D}})$, do we have $\mathcal{D} \models \Phi$? (Actually, undecidability even holds for formulas Φ that express simple state-reachability properties and do not use any guards on pids.)*

Proof. We give a reduction from the non-halting problem of Turing machines starting from an empty tape. Let S_{TM} be the set of control states of TM, and let B_{TM} be its tape alphabet. We assume that if the Turing machine halts, it halts in a special state HALT. We also assume that HALT is a proposition, i.e., $\mathrm{HALT} \in S_{\mathsf{TM}} \cap \mathcal{P}$. We describe the distributed algorithm $\mathcal{D}_{\mathsf{TM}}$.

Intuitively, the number of processes in the ring gives an upper bound on the space needed by the Turing machine. Every process will correspond to a cell in the Turing machine's tape. Since there is no specific starting process for a ring, we run a leader election algorithm first, and the leader will act as the leftmost cell of the tape. The i-th process to the right of the leader acts as the i-th tape cell. The local state of a process indicates the corresponding cell contents. It also indicates whether the head is currently present at the respective cell, and in that case remembers the current state of TM. Thus, the local states are pairs of the form $(\mathsf{sym}, \mathsf{head})$. Here, $\mathsf{sym} \in B_{\mathsf{TM}}$ indicates the content of a tape cell. Moreover, $\mathsf{head} \in S_{\mathsf{TM}} \cup \{\bot\}$ indicates the current state of TM if the head is present in the current cell, and it is $\bot$ otherwise. We assume that proposition HALT is true precisely in those local states whose second component is HALT. The messages (message types) are the states from S_{TM}, each having arity 0. A message will denote the target control state upon simulating one transition of the Turing machine. Initially, only the leader process has $\mathsf{head} \neq \perp$. In the simulation, only the process with $\mathsf{head} \neq \perp$ can send messages, and once it emits a message, the head tag is set to $\perp$. The process that receives a message $s \in S_{\mathsf{TM}}$ would turn the head tag to s.

We describe the construction in detail now. First, there is a preliminary phase which simulates a leader-election protocol, say, the Dolev-Klawe-Rodeh algorithm. The pid of the leader is stored in all processes in a special register r_{leader}. Recall that the leader process will act as the leftmost cell of the tape. A process can always check whether it is the leftmost by comparing the value of r_{leader} to the register id. This check will be used in guards later in transitions involving moving the head of TM to the left.

Once the preliminary phase is completed, the configuration of the ring proceeds to represent the initial configuration of TM. For this, the leader sends a message which is forwarded and received by all processes, so that all processes other than the leader move to state $(\$, \perp)$, i.e., representing the empty tape cell and indicating the absence of the head. The leader process will move to the state $(\$, s_{\text{in}})$ where s_{in} is the initial state of the Turing machine.

The simulation of the Turing machine works as follows. Consider a transition of the Turing machine which checks that the current state is s and the current cell contains a, updates the cell content to b, moves the head to the left and updates the control state to s'. The distributed algorithm will have a transition which moves from local state (a, s) to $(b, \perp)$ which also (i) ensures (by a guard) that it is not the leftmost cell ($r_{\text{leader}} \neq \mathsf{id}$), and (ii) sends s' to the left. For this transition to take place, there are complementary transitions at the receive end which go from $(\text{-}, \perp)$ to $(\text{-}, s')$ upon receiving an s'-message from a neighbor (left or right). In fact, such a receive transition is enabled for all processes in all the states. Other transitions of the Turing machine are also implemented similarly. Notice that message transmissions are performed by a process only if $\mathsf{head} \neq \perp$. Notice also that the leader process neither sends to the left nor receives from the left. Also, there are no forwarding states.

Finally, the specification $\varphi_{\mathsf{TM}} = \neg\, \mathsf{E}\, \textsc{halt}$ checks that no process visits a state whose second component is $\textsc{halt}$. This concludes the proof of Theorem 2. $\square$

4.3. Model checking can be reduced to satisfiability of LCPDL

In this subsection, we will show that the model-checking problem can be reduced to the satisfiability checking of LCPDL. The models of a DataPDL formula are data-cylinders, while that of LCPDL are finitely labeled cylinders. We will get rid of the register comparisons in the logic, and this depends crucially on the loop construct of LCPDL. Further the translation from DataPDL to LCPDL also depends on the distributed algorithm. In fact, it relies on the runs of the distributed algorithm to track the flow of data values. Thanks to this reduction, it follows from Theorem 2 that LCPDL satisfiability checking is also undecidable.

Data abstraction For a given distributed algorithm $\mathcal{D}$, when we refer to DataPDL, we actually mean DataPDL($\textsc{Props}_{\mathcal{D}}, \textsc{Regs}_{\mathcal{D}}$). Recall that the (extended) vocabulary $\textsc{Props}^{\text{tr}}_{\mathcal{D}}$ induced by $\mathcal{D}$ is $\textsc{Props}^{\text{tr}}_{\mathcal{D}} = \textsc{Props}_{\mathcal{D}} \cup \bigcup_{t \in \Delta} \mathsf{PropSetOf}(t)$. As in Section 3, we now explain how to abstract away the data. We show how to inductively translate every DataPDL formula Φ, φ, or π into a corresponding LCPDL($\textsc{Props}^{\mathrm{tr}}_{\mathcal{D}}$) formula $\widetilde{\Phi}$, $\widetilde{\varphi}$, or $\widetilde{\pi}$. This translation depends on the distributed algorithm $\mathcal{D}$.

The translation of sentences is trivial, we simply set $\widetilde{\mathsf{E}\,\Phi} = \mathsf{E}\,\widetilde{\Phi}$, $\widetilde{\neg\Phi} = \neg\widetilde{\Phi}$, and $\widetilde{\Phi_1 \wedge \Phi_2} = \widetilde{\Phi_1} \wedge \widetilde{\Phi_2}$. The translation $\widetilde{\pi}$ of a path formula π is simply obtained by replacing tests $\{\varphi\}?$ by $\{\widetilde{\varphi}\}?$. For atomic propositions of local formulas, we set $\widetilde{\ddagger} = \ddagger$ and $\widetilde{p} = p$. Boolean connectives are unchanged, i.e., $\widetilde{\neg\varphi} = \neg\widetilde{\varphi}$ and $\widetilde{\varphi_1 \wedge \varphi_2} = \widetilde{\varphi_1} \wedge \widetilde{\varphi_2}$. We also simply set $\widetilde{[\pi]\varphi} = [\widetilde{\pi}]\widetilde{\varphi}$. The interesting case is for data comparisons. Consider $\widetilde{\langle\pi\rangle r \neq \langle\pi'\rangle r'}$ (the equality case is simpler). To "prove" $\langle\pi\rangle r \neq \langle\pi'\rangle r'$ at a given position in a symbolic run, we require that there are a $\widetilde{\pi}$-path and a $\widetilde{\pi}'$-path to coordinates x and x', respectively, whose registers r and r' have different values. To guarantee the latter, the pids stored in r and r' have to originate from distinct processes. Again, using converse, this can be expressed with a loop as below:

$$\widetilde{\langle\pi\rangle r = \langle\pi'\rangle r'} = \mathsf{loop}(\widetilde{\pi} \cdot (\mathcal{A}_r)^{-1} \cdot \mathcal{A}_{r'} \cdot (\widetilde{\pi'})^{-1})$$

$$\widetilde{\langle\pi\rangle r \neq \langle\pi'\rangle r'} = \mathsf{loop}(\widetilde{\pi} \cdot (\mathcal{A}_r)^{-1} \cdot ((\{\neg\ddagger\}?{\leftarrow})^{+} + (\{\neg\ddagger\}?{\rightarrow})^{+} + \{\ddagger\}?{\rightarrow}^{+}\{\neg\ddagger\}?) \cdot \mathcal{A}_{r'} \cdot (\widetilde{\pi'})^{-1})$$

Note that, $\mathcal{A}_r$ and $\mathcal{A}_{r'}$ (cf. page 17) refer to register contents *after* updates, which reflects the fact that DataPDL speaks about *configurations*.

Lemma 4. *Let χ be a run of $\mathcal{D}$. Recall that the symbolic run associated to χ is ζ_χ and the associated data-cylinder is ξ_χ. Note that $Pos(\zeta_\chi) = Pos(\xi_\chi)$, which we denote by $Pos(\chi)$. For all* DataPDL *sentences Φ, local formulas φ, path formulas π, and all $x, y \in Pos(\chi)$, we have*

$$\zeta_\chi \models \widetilde{\Phi} \iff \xi_\chi \models \Phi \tag{4}$$
$$\zeta_\chi, x \models \widetilde{\varphi} \iff \xi_\chi, x \models \varphi \tag{5}$$
$$\zeta_\chi, x, y \models \widetilde{\pi} \iff \xi_\chi, x, y \models \pi. \tag{6}$$

Proof. The proof is by structural induction on the formulas. Notice that (4) follows directly from (5). The atomic cases when $\varphi = \ddagger$, and $\pi = \rightarrow$ or $\pi = \downarrow$ are trivial.

- Suppose $\varphi = p \in \mathcal{P}$. By the definition of ξ_χ and ζ_χ, for all $x \in Pos(\chi)$, we have $\zeta_\chi, x \models \widetilde{p} = p \iff \xi_\chi, x \models p$.
- The inductive steps for local formulas are obvious for negation, conjunction, and $[\pi]\varphi$.
- The inductive step for path formulas are also obvious for tests $\{\varphi\}?$ and for all regular operations π^{-1}, $\pi + \pi'$, $\pi \cdot \pi'$, and π^*.
- Suppose $\varphi = \big(\langle\pi_1\rangle r_1 \neq \langle\pi_2\rangle r_2\big)$. This is the most interesting case. For all $x = (i,j) \in Pos(\chi)$ we have

$$\zeta_\chi, x \models \widetilde{\langle\pi_1\rangle r_1 \neq \langle\pi_2\rangle r_2}$$

$\iff$ $\zeta_\chi, x \models \mathsf{loop}(\widetilde{\pi}_1 \cdot (\mathcal{A}_{r_1})^{-1} \cdot ((\{\neg\ddagger\}?{\leftarrow})^+ + (\{\neg\ddagger\}?{\rightarrow})^+ + \{\ddagger\}?{\rightarrow}^+\{\neg\ddagger\}?) \cdot \mathcal{A}_{r_2} \cdot (\widetilde{\pi}_2)^{-1})$

$\iff$ there are coordinates $x_1, x_2, (i_1, 0), (i_2, 0) \in Pos(\chi)$ such that:

1. $\zeta_\chi, x, x_1 \models \widetilde{\pi}_1$ and $\zeta_\chi, x, x_2 \models \widetilde{\pi}_2$
2. $\zeta_\chi, (i_1, 0), x_1 \models \mathcal{A}_{r_1}$ and $\zeta_\chi, (i_2, 0), x_2 \models \mathcal{A}_{r_2}$
3. $\zeta_\chi, (i_1, 0), (i_2, 0) \models (\{\neg\ddagger\}?{\leftarrow})^+ + (\{\neg\ddagger\}?{\rightarrow})^+ + \{\ddagger\}?{\rightarrow}^+\{\neg\ddagger\}?$

$\iff$ (by (6) and Lemma 2)
there are coordinates $x_1, x_2, (i_1, 0), (i_2, 0) \in Pos(\chi)$ such that:

1. $\xi_\chi, x, x_1 \models \pi_1$ and $\xi_\chi, x, x_2 \models \pi_2$
2. $\xi_\chi(i_1, 0)(\mathsf{id}) = \xi_\chi(x_1)(r_1)$ and $\xi_\chi(i_2, 0)(\mathsf{id}) = \xi_\chi(x_2)(r_2)$
3. $i_1 \neq i_2$

$\iff$ $\xi_\chi, x \models \langle\pi_1\rangle r_1 \neq \langle\pi_2\rangle r_2$

- The case $\varphi = \big(\langle\pi_1\rangle r_1 = \langle\pi_2\rangle r_2\big)$ is almost identical. We just have to adapt 3. accordingly.

This concludes the proof of Lemma 4. $\square$

We say that two runs χ, χ' of $\mathcal{D}$ are *similar*, denoted $\chi \sim \chi'$ if they induce the same symbolic run: $\zeta_\chi = \zeta_{\chi'}$. This is an equivalence relation. Basically, two similar runs differ by the pid assignments of the processes. We immediately deduce from Lemma 4 the following closure properties.

Corollary 1. *Let χ, χ' be two* similar *runs of $\mathcal{D}$. Then, for all* DataPDL *sentences Φ, local formulas φ, path formulas π, and all $x, y \in Pos(\chi)$, we have*

$$\xi_\chi \models \Phi \iff \xi_{\chi'} \models \Phi$$
$$\xi_\chi, x \models \varphi \iff \xi_{\chi'}, x \models \varphi$$
$$\xi_\chi, x, y \models \pi \iff \xi_{\chi'}, x, y \models \pi$$

Completing the reduction In Section 3, we showed how to translate a distributed algorithm into LCPDL. In the previous subsection, we showed how to translate a DataPDL specification into LCPDL. We will now combine these two results and reduce model checking of a distributed algorithm to (non-)satisfiability of a single LCPDL formula. More precisely, we reduce model checking to non-satisfiability of the conjunction of two LCPDL formulas of polynomial size: the formula representing the algorithm, and the *negation* of the formula representing the specification.

Lemma 5. *Let $\mathcal{D}$ be a distributed algorithm and let $\Phi \in$* DataPDL(Props$_\mathcal{D}$, Regs$_\mathcal{D}$) *be a sentence. We have*

$$\mathcal{D} \models \Phi \iff L(\Psi_\mathcal{D} \wedge \neg\widetilde{\Phi}) = \emptyset\,.$$

Proof. ($\Rightarrow$): Suppose $\mathcal{D} \models \Phi$. Let $\zeta \in L(\Psi_{\mathcal{D}})$. By Theorem 1, there is a run χ of $\mathcal{D}$ such that $\zeta_\chi = \zeta$. Moreover, since $\mathcal{D} \models \Phi$, for all runs χ of $\mathcal{D}$, we have $\xi_\chi \models \Phi$. By Lemma 4, we have, for all runs χ of $\mathcal{D}$, $\zeta_\chi \models \widetilde{\Phi}$. We conclude $L(\Psi_{\mathcal{D}} \wedge \neg\widetilde{\Phi}) = \emptyset$.

($\Leftarrow$): Suppose $\mathcal{D} \not\models \Phi$. Then, there are a ring R, and an R-run χ of $\mathcal{D}$ such that $\xi_\chi \not\models \Phi$. By Lemma 4, $\zeta_\chi \not\models \widetilde{\Phi}$ and, therefore, $\zeta_\chi \models \neg\widetilde{\Phi}$. Due to Theorem 1, we also have $\zeta_\chi \models \Psi_{\mathcal{D}}$. We conclude $L(\Psi_{\mathcal{D}} \wedge \neg\widetilde{\Phi}) \neq \emptyset$. $\square$

4.4. Extension with inequalities

Now, we will show that the results above all hold for an extended version of the logic. More precisely, we consider an extension of DataPDL with modalities for checking inequality between registers. We define the model-checking problem as before, but the specification can be from the extended logic. We show that the model checking problem against the richer specification can also be reduced to satisfiability of LCPDL.

In some cases, the specification might need to compare data values that are stored in registers wrt. the total order. For instance, the specification of the leader-election algorithm may require that the process with the highest pid is elected. Moreover, a specification for a sorting algorithm clearly should compare data values wrt. $<$. Thus, we will extend the comparisons $\langle\pi\rangle r \bowtie \langle\pi'\rangle r'$ by allowing $\bowtie \in \{<, \leq\}$. But we have to be a bit careful since the closure properties of Corollary 1 do not hold in general with this extension. Indeed, let χ, χ' be two similar runs (i.e., $\zeta_\chi = \zeta_{\chi'}$). We may have $\xi_\chi(0,0)(r) < \xi_\chi(1,0)(r)$ but $\xi_{\chi'}(0,0)(r) > \xi_{\chi'}(1,0)(r)$ depending on the pid assignment in the ring. Therefore, the formula $\langle\varepsilon\rangle r < \langle\rightarrow\rangle r$ is not invariant under similar runs. Our extension of DataPDL is defined as follows.

Definition 3. The logic DataPDL$^\oplus$(Props, Regs), which is parameterized by finite sets Props and Regs, is given by the following grammar:

$$\begin{aligned}
\Phi, \Phi' &::= \mathsf{A}\,\phi \mid \Phi \wedge \Phi' \\
\phi, \phi' &::= \varphi \mid \phi \wedge \phi' \mid \varphi \vee \phi \mid [\pi]\phi \mid \langle\eta\rangle r \bowtie' \langle\eta'\rangle r' \\
\varphi, \varphi' &::= \ddagger \mid p \mid \neg\varphi \mid \varphi \wedge \varphi' \mid [\pi]\varphi \mid \langle\pi\rangle r \bowtie \langle\pi'\rangle r' \\
\pi, \pi' &::= \{\varphi\}? \mid \rightarrow \mid \downarrow \mid \pi + \pi' \mid \pi \cdot \pi' \mid \pi^* \mid \pi^{-1} \\
\eta, \eta' &::= \{\varphi\}? \mid \leftarrow \mid \rightarrow \mid \downarrow \mid \uparrow \mid \eta \cdot \eta' \mid \mathsf{F}^\eta_\varphi
\end{aligned}$$

where $p \in$ Props, $r, r' \in$ Regs, $\bowtie \in \{=, \neq\}$, and $\bowtie' \in \{<, \leq\}$. The macro F^η_φ stands for $(\{\neg\varphi\}? \cdot \eta)^* \cdot \{\varphi\}?$. It moves to the *very next* position where φ holds using steps defined by the path formula η. $\lhd$

In the following, when Props and Regs are clear from the context or not important, we just write DataPDL$^\oplus$.

Notice that we have no negations or disjunctions at the level of sentences Φ. Also, we have no negations on local formulas ϕ. So we include disjunction, but one of its arguments should be a local formula from DataPDL. In fact, there is a subtle point concerning disjunction. In the following, let us write $r < r'$ instead of $\langle\epsilon\rangle r < \langle\epsilon\rangle r'$. Satisfaction of a formula $r < r'$ can only be guaranteed in a symbolic execution if the flow of pids provides *evidence* that $r < r'$ really holds. More concretely, the (hypothetic) formula $(r < r') \vee (r = r') \vee (r' < r)$ is a tautology, but it may not be possible to prove any of its disjuncts on the basis of a symbolic run.

Also, the path formulas η, η' used for data inequalities are syntactically restricted to *unambiguous* path formulas. It can be easily checked that, for all rings R, R-runs χ of $\mathcal{D}$, and positions $x \in \mathit{Pos}(\chi)$, there is at most one $x' \in \mathit{Pos}(\chi)$ such that $\xi_\chi, x, x' \models \eta$. Notice that union ($+$) and arbitrary stars (*) are not allowed since they may introduce ambiguous path formulas such as $\leftarrow + \rightarrow$ and $\rightarrow^*$.

Example 5. Notice that the path formula $\pi_{\mathsf{found}} = \mathsf{F}^{\rightarrow}_{\mathit{found}}$ of Example 4 is unambiguous. Hence, the local formula $\phi_{\mathsf{max}} = [\rightarrow^*]\big(\langle\epsilon\rangle\mathsf{id} \leq \langle\pi_{\mathsf{found}}\rangle r\big)$ is in DataPDL$^\oplus$. So we can strengthen the specification of a leader election algorithm by requiring that the process with the highest pid is elected:

$$\Phi_1' = \mathsf{A}\left((\ddagger \wedge \varphi_{\mathsf{acc}}) \implies (\varphi_{\mathsf{found}} \wedge \varphi_{r=r} \wedge \phi_{\mathsf{max}})\right)$$

The next example formulates the correctness constraint for a distributed sorting algorithm (cf. Example 3). We would like to say that, when the algorithm terminates (leader is in state/satisfies proposition ℓ_4), the pids stored in registers r are strictly totally ordered. Then,

$$\Phi_2 = \mathsf{A}\left(\ell_4 \implies [\rightarrow^*\{\neg\ell_4\}?](\langle\leftarrow\rangle r < \langle\epsilon\rangle r)\right)$$

makes sure that, whenever process j is not terminating in ℓ_4, its left neighbor i stores a smaller pid in r than j does. Note that Φ_1' and Φ_2 are DataPDL$^\oplus$ formulas. ◁

Our goal now is to show that Lemma 5 can be lifted to DataPDL$^\oplus$ as well. Towards this, we lift the model-checking problem and the data abstraction to the new setting:

Input:	A distributed algorithm $\mathcal{D}$, a sentence $\Phi \in \text{DataPDL}^\oplus(\textsc{Props}_\mathcal{D}, \textsc{Regs}_\mathcal{D})$
Question:	Is it the case that $\mathcal{D} \models \Phi$?

We extend the data abstraction to DataPDL$^\oplus$ and augment the translation defined in Section 4.3 as follows:

$$\begin{aligned}
\widetilde{\varphi \vee \phi} &= \widetilde{\varphi} \vee \widetilde{\phi} \\
\widetilde{\langle\eta\rangle r < \langle\eta'\rangle r'} &= \mathsf{loop}(\widetilde{\eta} \cdot (\mathcal{A}_r)^{-1} \cdot \pi_< \cdot \mathcal{A}_{r'} \cdot (\widetilde{\eta'})^{-1}) \\
\widetilde{\langle\eta\rangle r \leq \langle\eta'\rangle r'} &= \mathsf{loop}(\widetilde{\eta} \cdot (\mathcal{A}_r)^{-1} \cdot (\pi_< + \epsilon) \cdot \mathcal{A}_{r'} \cdot (\widetilde{\eta'})^{-1})
\end{aligned}$$

Lemma 6. *For all runs χ of $\mathcal{D}$, all* DataPDL$^\oplus$ *sentences Φ, local formulas ϕ, path formulas π, and all $x, y \in \mathit{Pos}(\chi)$, we have*

$$\zeta_\chi \models \widetilde{\Phi} \iff \forall \chi' \sim \chi : \xi_{\chi'} \models \Phi \tag{7}$$

$$\zeta_\chi, x \models \widetilde{\phi} \iff \forall \chi' \sim \chi : \xi_{\chi'}, x \models \phi \tag{8}$$

$$\zeta_\chi, x, y \models \widetilde{\pi} \iff \forall \chi' \sim \chi : \xi_{\chi'}, x, y \models \pi\,. \tag{9}$$

Proof. We fix a run χ of $\mathcal{D}$. The proof is by structural induction. First, we deduce from Lemma 4 and Corollary 1 that property (8) holds for local formulas $\phi = \varphi \in \text{DataPDL}$. We deduce also that (9) holds for all path formulas π in DataPDL. This applies in particular to unambiguous path formulas $\pi = \eta$. Notice also that (7) follows easily from (8).

We consider now the new cases of DataPDL$^\oplus$.

- Consider the formula $\varphi \vee \phi$ with φ in DataPDL and ϕ in DataPDL$^\oplus$. Let $x \in \mathit{Pos}(\chi)$.

$$\begin{aligned}\zeta_\chi, x \models \widetilde{\varphi \vee \phi} = \widetilde{\varphi} \vee \widetilde{\phi} &\iff \zeta_\chi, x \models \widetilde{\varphi} \text{ or } \zeta_\chi, x \models \widetilde{\phi} \\ &\overset{(5,8)}{\iff} (\xi_\chi, x \models \varphi) \text{ or } (\xi_{\chi'}, x \models \phi \text{ for all runs } \chi' \sim \chi) \\ &\overset{\text{Cor. 1}}{\iff} \xi_{\chi'}, x \models \varphi \vee \phi \text{ for all runs } \chi' \sim \chi\,.\end{aligned}$$

  Notice that we use the closure property of Corollary 1 for the last equivalence. One direction is easy. Conversely, assume that $\xi_{\chi'}, x \models \varphi \vee \phi$ for all runs $\chi' \sim \chi$. Then, either $\xi_\chi, x \models \varphi$ and we are done. Or, by Corollary 1, we know that $\xi_{\chi'}, x \not\models \varphi$ for all $\chi' \sim \chi$. We deduce that $\xi_{\chi'}, x \models \phi$ for all runs $\chi' \sim \chi$.

- Suppose $\phi = \phi_1 \wedge \phi_2$. For all $x \in \mathit{Pos}(\chi)$, we have

$$\begin{aligned}\zeta_\chi, x \models \widetilde{\phi_1 \wedge \phi_2} = \widetilde{\phi_1} \wedge \widetilde{\phi_2} &\iff \zeta_\chi, x \models \widetilde{\phi_1} \text{ and } \zeta_\chi, x \models \widetilde{\phi_2} \\ &\overset{(8)}{\iff} (\forall \chi' \sim \chi : \xi_{\chi'}, x \models \phi_1) \text{ and } (\forall \chi' \sim \chi : \xi_{\chi'}, x \models \phi_2) \\ &\iff \forall \chi' \sim \chi : \xi_{\chi'}, x \models \phi_1 \wedge \phi_2\,.\end{aligned}$$

- Consider the formula $[\pi]\phi$. For all $x \in \mathit{Pos}(\chi)$, we have

$$\begin{aligned}\zeta_\chi, x \models \widetilde{[\pi]\phi} = [\widetilde{\pi}]\widetilde{\phi} &\iff \forall y \in \mathit{Pos}(\chi) : \quad \zeta_\chi, x, y \models \widetilde{\pi} \implies \zeta_\chi, y \models \widetilde{\phi} \\ &\overset{(8)}{\iff} \forall y \in \mathit{Pos}(\chi),\ \forall \chi' \sim \chi : \quad \zeta_\chi, x, y \models \widetilde{\pi} \implies \xi_{\chi'}, y \models \phi \\ &\overset{(9)}{\iff} \forall \chi' \sim \chi,\ \forall y \in \mathit{Pos}(\chi) : \quad \xi_{\chi'}, x, y \models \pi \implies \xi_{\chi'}, y \models \phi \\ &\iff \forall \chi' \sim \chi : \quad \xi_{\chi'}, x \models [\pi]\phi\end{aligned}$$

- Suppose $\phi = \langle \eta_1 \rangle r_1 \leq \langle \eta_2 \rangle r_2$. Let $x \in \mathit{Pos}(\chi)$. We have

$$\zeta_\chi, x \models \widetilde{\langle \eta_1 \rangle r_1 \leq \langle \eta_2 \rangle r_2} = \mathsf{loop}(\widetilde{\eta_1} \cdot (\mathcal{A}_{r_1})^{-1} \cdot (\pi_< + \epsilon) \cdot \mathcal{A}_{r_2} \cdot (\widetilde{\eta_2})^{-1})$$

  $\iff$ $\exists x_1, x_2, (i_1, 0), (i_2, 0) \in \mathit{Pos}(\chi)$ such that:
  1. $\zeta_\chi, x, x_1 \models \widetilde{\eta_1}$ and $\zeta_\chi, x, x_2 \in \widetilde{\eta_2}$
  2. $\zeta_\chi, (i_1, 0), x_1 \models \mathcal{A}_{r_1}$ and $\zeta_\chi, (i_2, 0), x_2 \models \mathcal{A}_{r_2}$
  3. $i_1 = i_2$ or $\zeta_\chi, (i_1, 0), (i_2, 0) \models \pi_<$

  $\iff$ (by (9), Lemma 2, and Lemma 3)
  $\exists x_1, x_2, (i_1, 0), (i_2, 0) \in \mathit{Pos}(\chi)$ such that:
  1. $\forall \chi' \sim \chi : \xi_{\chi'}, x, x_1 \models \eta_1$ and $\xi_{\chi'}, x, x_2 \models \eta_2$
  2. $\forall \chi' \sim \chi : \xi_{\chi'}(i_1, 0)(\mathsf{id}) = \xi_{\chi'}(x_1)(r_1)$ and $\xi_{\chi'}(i_2, 0)(\mathsf{id}) = \xi_{\chi'}(x_2)(r_2)$
  3. $i_1 = i_2$ or $\forall \chi' \sim \chi : \xi_{\chi'}(i_1, 0)(\mathsf{id}) < \xi_{\chi'}(i_2, 0)(\mathsf{id})$

$\iff$ $\exists x_1, x_2, (i_1, 0), (i_2, 0) \in \mathit{Pos}(\chi)$ such that $\forall \chi' \sim \chi$:

1. $\xi_{\chi'}, x, x_1 \models \eta_1$ and $\xi_{\chi'}, x, x_2 \models \eta_2$
2. $\xi_{\chi'}(i_1, 0)(\mathsf{id}) = \xi_{\chi'}(x_1)(r_1)$ and $\xi_{\chi'}(i_2, 0)(\mathsf{id}) = \xi_{\chi'}(x_2)(r_2)$
3. $\xi_{\chi'}(i_1, 0)(\mathsf{id}) \leq \xi_{\chi'}(i_2, 0)(\mathsf{id})$

$\overset{(*)}{\iff}$ $\forall \chi' \sim \chi : \exists x_1, x_2, (i_1, 0), (i_2, 0) \in \mathit{Pos}(\chi)$ such that:

1. $\xi_{\chi'}, x, x_1 \models \eta_1$ and $\xi_{\chi'}, x, x_2 \models \eta_2$
2. $\xi_{\chi'}(i_1, 0)(\mathsf{id}) = \xi_{\chi'}(x_1)(r_1)$ and $\xi_{\chi'}(i_2, 0)(\mathsf{id}) = \xi_{\chi'}(x_2)(r_2)$
3. $\xi_{\chi'}(i_1, 0)(\mathsf{id}) \leq \xi_{\chi'}(i_2, 0)(\mathsf{id})$

$\iff$ $\forall \chi' \sim \chi : \xi_{\chi'}, x \models \langle \eta_1 \rangle r_1 \leq \langle \eta_2 \rangle r_2$

$(*)$ For the right to left direction, we may exchange the quantifications $\forall\exists$ to $\exists\forall$ since the coordinates x_1, x_2 are uniquely determined by the run χ, the starting position x and the unambiguous paths η_1, η_2. In turn, the indices i_1, i_2 are also uniquely determined by the run χ, the registers r_1, r_2, and the positions x_1, x_2.

- The case $\varphi = \langle \eta_1 \rangle r_1 < \langle \eta_2 \rangle r_2$ is simpler than the previous one. We just have to adapt 3. accordingly.

This concludes the proof of Lemma 6. $\square$

We are now ready to state the variant of Lemma 5 for DataPDL$^{\oplus}$:

Lemma 7. *Let $\mathcal{D}$ be a distributed algorithm and let $\Phi \in \text{DataPDL}^{\oplus}(\text{Props}_{\mathcal{D}}, \text{Regs}_{\mathcal{D}})$ be a sentence. We have*

$$\mathcal{D} \models \Phi \iff L(\Psi_{\mathcal{D}} \wedge \neg\widetilde{\Phi}) = \emptyset\,.$$

Proof. ($\Rightarrow$): Suppose $\mathcal{D} \models \Phi$. Let $\zeta \in L(\Psi_{\mathcal{D}})$. By Theorem 1, there is a run χ of $\mathcal{D}$ such that $\zeta_\chi = \zeta$. Moreover, since $\mathcal{D} \models \Phi$, for all runs χ of $\mathcal{D}$, we have $\xi_\chi \models \Phi$. By Lemma 6, we have, for all runs χ of $\mathcal{D}$, $\zeta_\chi \models \widetilde{\Phi}$. We conclude $L(\Psi_{\mathcal{D}} \wedge \neg\widetilde{\Phi}) = \emptyset$.

($\Leftarrow$): Suppose $\mathcal{D} \not\models \Phi$. Then, there are a ring R and an R-run χ of $\mathcal{D}$ such that $\xi_\chi \not\models \Phi$. By Lemma 6, $\zeta_\chi \not\models \widetilde{\Phi}$ and, therefore, $\zeta_\chi \models \neg\widetilde{\Phi}$. Due to Theorem 1, we also have $\zeta_\chi \models \Psi_{\mathcal{D}}$. We conclude $L(\Psi_{\mathcal{D}} \wedge \neg\widetilde{\Phi}) \neq \emptyset$. $\square$

5. (Round-Bounded) Model Checking

In the realm of multithreaded concurrent programs, where model checking is undecidable in general, a fruitful approach has been to underapproximate the behavior of a system [36]. The idea is to introduce a parameter that measures a characteristic of a run such as the number of thread switches it performs. One then imposes a bound on this parameter and explores all behaviors up to that bound. In numerous distributed algorithms, the number

B of rounds needed to conclude is exponentially smaller than the number of processes (cf. Example 1). Therefore, B seems to be a promising parameter for bounded model checking of distributed algorithms.

For a distributed algorithm $\mathcal{D}$, a sentence $\Phi \in \text{DataPDL}^{\oplus}(\text{Props}_{\mathcal{D}}, \text{Regs}_{\mathcal{D}})$, and $B \geq 1$, we write $\mathcal{D} \models_B \Phi$ if, for all rings R, and all R-runs χ of length $k \leq B$, we have $\xi_\chi \models \Phi$.

Theorem 3. *The following problem is PSPACE-complete: Given a distributed algorithm $\mathcal{D}$, a sentence $\Phi \in \text{DataPDL}^{\oplus}(\text{Props}_{\mathcal{D}}, \text{Regs}_{\mathcal{D}})$, and a natural number $B \geq 1$ (encoded in unary), do we have $\mathcal{D} \models_B \Phi$?*

Proof (of lower bound). To prove the lower bound, we give a polynomial reduction from the intersection-emptiness problem of finite-state automata. That is, given k finite-state automata $\mathcal{A}_1, \ldots, \mathcal{A}_k$ over a finite alphabet Σ, where $\mathcal{A}_i = (Q_i, \Delta_i, \mathsf{init}_i, \mathsf{F}_i)$, to decide whether $\bigcap_i L(\mathcal{A}_i) = \emptyset$. This problem is known to be PSPACE-complete [31].

We will need only unidirectional rings for our reduction. The number of processes in the ring corresponds to the length of a candidate word accepted by all the automata $\mathcal{A}_i$. Each process thus corresponds to a position in the word. The local state of the process remembers the letter from Σ at the respective position. The message contents will be the states of the automata. At round i, all the processes try to simulate a transition of automaton $\mathcal{A}_i$ on the respective position. Thus the local states will be $(\Sigma \cup \{\$\}) \times [k]$ and the set of messages (of arity 0) will be $\bigcup_i Q_i$.

A process non-deterministically moves to a local state from the set $(\Sigma \cup \{\$\}) \times \{0\}$. The special symbol \$ marks that a candidate word may start at the right of this process and end at the left of this process. The local state may also remember an index $0 < i < k$, indicating that the next round will simulate $\mathcal{A}_{i+1}$. For each $a \in \Sigma$ and $0 < i \leq k$, if $(s, a, s') \in \Delta_i$ we have a transition of the form $\langle (a, i-1)\colon \textbf{right}!s' \,; \textbf{left}?s \,; \textbf{goto}\ (a, i) \rangle$. Further, if $f_i \in \mathsf{F}_i$ we have $\langle (\$, i-1)\colon \textbf{right}!\mathsf{init}_i \,; \textbf{left}?f_i \,; \textbf{goto}\ (\$, i) \rangle$. Notice that the symbol from Σ associated to a process does not change in any of these transitions.

The size of the distributed algorithm $\mathcal{D}$ is polynomial in the size of the input to the intersection-emptiness problem. The number of rounds needed is only k.

Finally, the DataPDL formula states that the state $(\$, k)$ cannot be reached:

$$\Phi = \mathsf{A}\,\neg(\$, k)$$

Notice that, if the bounded model checking answers no, then there are a ring and a run such that some process m eventually reaches the state $(\$, k)$. This means that, on all states $(\$, i-1)$, m has received a state $f_i \in \mathsf{F}_i$. Let m' be the first process on the left of m which has a state of the form $(\$, i-1)$. Note that m' can be the same as m. The word represented by the states of the processes between m' and m is in $\bigcap_i L(\mathcal{A}_i)$. Note that, even if this is the empty word (that is, m' is the left neighbor of m), it must be in the intersection since $\mathsf{init}_i \in \mathsf{F}_i$ for every automaton $\mathcal{A}_i$. On the other hand, if the intersection is nonempty, there is a run that violates the specification.

Thus, the bounded model checking of $\mathcal{D}$ answers yes if, and only if, the intersection of the $L(\mathcal{A}_i)$ is empty. This proves the PSPACE lower bound. $\square$

Currently, we do not know what happens when the number B is encoded in binary. Before we prove the upper bound, let us discuss the result in more detail. We will first compare it with "naïve" approaches to solve related questions. Consider the problem to determine whether a distributed algorithm satisfies its specification for all rings up to size n and all runs up to length B. This problem is in coNP: We guess a ring (i.e., essentially, a permutation of pids) and a run, and we check, using [32], whether the run does *not* satisfy the formula. Next, suppose only B is given and the question is whether, for all rings up to size 2^B and all runs up to length B, the property holds. Then, the above procedure gives us a coNEXPTIME algorithm.

Thus, our result is interesting complexity-wise. Moreover it offers some other advantages. First, it actually checks correctness (up to round number B) for *all* rings. This is essential when verifying distributed *protocols* against safety properties. Second, it reduces to satisfiability checking of LCPDL formulas [26], which in turn can be reduced to non-emptiness of alternating two-way automata (A2As) over words [41, 34, 38]. The "naïve" approaches, on the other hand, do not seem to give rise to viable algorithms.

Proof (of Upper Bound of Theorem 3). We show the upper bound by a series of polynomial reductions. First, we will reduce (cf. Lemma 8) round-bounded model checking to satisfiability checking of LCPDL over height-bounded cylinders. We will then reduce (cf. Lemma 9) the latter problem to satisfiability checking of LCPDL over words, which can in turn be reduced (cf. Lemmas 11 and 10) to emptiness checking of alternating two-way automata over words, which is in PSPACE.

For an LCPDL sentence Ψ and $B \geq 1$, we denote by $L_B(\Psi)$ the set of (n,k)-cylinders $\zeta \in L(\Psi)$ such that $k \leq B$ and n is arbitrary.

Lemma 8. *Let $\mathcal{D}$ be a distributed algorithm, $\Phi \in \mathrm{DataPDL}^{\oplus}(\textsc{Props}_{\mathcal{D}}, \textsc{Regs}_{\mathcal{D}})$ be a sentence, and $B \geq 1$. We have*

$$\mathcal{D} \models_B \Phi \iff L_B(\Psi_{\mathcal{D}} \wedge \neg\widetilde{\Phi}) = \emptyset\,.$$

The size of $\Psi_{\mathcal{D}}$ is only polynomial in the size of $\mathcal{D}$ and that of $\neg\widetilde{\Phi}$ is only $\mathcal{O}(|\Phi| \times |\mathcal{D}|)$. Further, the formulas $\Psi_{\mathcal{D}}$ and $\neg\widetilde{\Phi}$ can be constructed in polynomial time.

Proof. The proof proceeds in exactly the same way as for Lemma 7, restricting the height of a cylinder and length of a run by the given bound B. □

Now, we will show that satisfiability checking of LCPDL over height-bounded cylinders can be reduced, in polynomial time, to satisfiability checking of LCPDL over words. In fact, a sentence from LCPDL(Props) can be interpreted over nonempty words $w \in (2^{\textsc{Props}})^+$ or $w \in \textsc{Props}^+$ in a straightforward manner. A local formula is interpreted wrt. a position of w. Moreover, $\rightarrow$ moves to the next word position. However, $\rightarrow$ cannot be applied on the last position of w, and $\leftarrow$ cannot be applied on its first position. Finally, $\downarrow$ is meaningless (i.e., it cannot be applied at all).

An $(n \times k)$-cylinder is called *B-bounded* if $k \leq B$.

Lemma 9. *Let* Props *be a finite set. For every sentence* $\Psi \in \mathrm{LCPDL}(\textsc{Props})$ *over cylinders over* Props *and every* $B \geq 1$*, we can construct a sentence* $\Psi_B \in \mathrm{LCPDL}(\textsc{Props} \uplus \{\#\})$ *such that* Ψ *is satisfiable over* B*-bounded cylinders iff* Ψ_B *is satisfiable over words in* $(2^{\textsc{Props} \uplus \{\#\}})^+$*. Further,* $|\Psi_B|$ *is bounded by a polynomial in* $|\Psi|$*.*

Proof. Let $\zeta \colon \mathbb{Z}_n \times [k] \to 2^{\textsc{Props}}$ be an $(n \times k)$-cylinder for some $k \leq B$. We encode ζ as the word $w_\zeta \in (2^{\textsc{Props} \uplus \{\#\}})^+$ given as:

$$\#\zeta(0,0)\zeta(0,1)\ldots\zeta(0,k)\#\zeta(1,0)\zeta(1,1)\ldots\zeta(1,k)\#\ldots\#\zeta(n-1,0)\zeta(n-1,1)\ldots\zeta(n-1,k)\#$$

Here, # acts as a delimiter for columns, which never appears along with another proposition (note that we simply write # instead of {#}). Thus, the columns are written horizontally rather than vertically. We write down the columns successively, starting from column 0.

We translate an LCPDL formula over cylinders of height k into an LCPDL formula over their word encodings. Going down corresponds to going right without reaching the end-marker #: $\downarrow$ is translated to $\rightarrow\{\neg\#\}?$. Moreover, going to the right in the cylinder is same as going $k+2$ steps to the right in the word, provided we are not at column $n-1$ (for right). At the column $n-1$, we go right as much as possible (say m steps), and take the remaining $k+2-m$ steps from the beginning of the word. Thus $\rightarrow$ gets translated to

$$\rightarrow^{k+2} \; + \; +_{m:0<m<k+2} \rightarrow^m \cdot \{\neg\langle\rightarrow\rangle\}? \cdot \leftarrow^* \cdot \{\neg\langle\leftarrow\rangle\}? \cdot \rightarrow^{k+2-m}$$

The formula $\ddagger$ is true only in column 0 which corresponds to the first block between #. Hence $\ddagger$ gets translated to $\langle(\{\neg\#\}? \cdot \leftarrow)^* \cdot \{\# \wedge \neg\langle\leftarrow\rangle\}?\rangle$. We add a formula φ_k checking that between two consecutive # there are $k+1$ positions (which are not labeled #). That is, φ_k checks that each column has length $k+1$.

Let Ψ^k be the translation we obtained as above for the sentence Ψ when restricted to cylinders of height k. The required Ψ_B is $\bigvee_{k \leq B} \Psi^k$. $\square$

Now, we have a polynomial sized formula in $\mathrm{LCPDL}(\textsc{Props} \uplus \{\#\})$ over words. However, the models of this formula are words over the alphabet $2^{\textsc{Props} \uplus \{\#\}}$. For the PSPACE upper bound that we are aiming at, it is convenient to get rid of this exponential blow-up in the alphabet.

Lemma 10. *Let* Props *be a finite set. For every sentence* $\Psi \in \mathrm{LCPDL}(\textsc{Props})$ *over words in* $(2^{\textsc{Props}})^+$*, we can construct a sentence* $\widehat{\Psi} \in \mathrm{LCPDL}(\{0,1,2\})$ *over words in* $\{0,1,2\}^+$ *such that* Ψ *is satisfiable iff* $\widehat{\Psi}$ *is satisfiable. Further,* $|\widehat{\Psi}| = \mathcal{O}(|\Psi| \times |\textsc{Props}|)$*.*

Proof. Let $\textsc{Props} = \{p_1, p_2, \ldots, p_m\}$. That is, $|\textsc{Props}| = m$, and $p_1, p_2, \ldots, p_m$ is an enumeration of the propositions. Let $\sigma \subseteq 2^{\textsc{Props}}$ be an assignment of the propositions. We encode σ by a bit vector $\widehat{\sigma} \in \{0,1\}^m$ of length m. The ith bit of $\widehat{\sigma}$ is 1 iff $p_i \in \sigma$. We use the letter 2 as a delimiter to separate the bit-vectors in the encoding.

Let $w = \sigma_1\sigma_2\ldots\sigma_n \in (2^{\textsc{Props}})^+$. We encode w as the word

$$\widehat{w} = 2 \cdot \widehat{\sigma_1} \cdot 2 \cdot \widehat{\sigma_2} \cdot 2 \cdots 2 \cdot \widehat{\sigma_n} \;\in \{0,1,2\}^+$$

of length $|w| \times (1+m)$.

In the translation from LCPDL(PROPS) we "relativize" to the positions labeled 2. In $\widehat{\Psi}$, we reason about letters rather than propositions, but the semantics is exactly as if $0, 1$ and 2 were propositions.

We can write an LCPDL($\{0,1,2\}$) sentence to ensure that a word is contained in $(\{2\} \cdot \{0,1\}^m)^+$. Further, we ensure that translations of node formulas are evaluated at positions labeled 2. $\mathsf{E}\,\psi$ gets translated to $\mathsf{E}(2 \wedge \widehat{\psi})$, where $\widehat{\psi}$ is the translation of ψ. Further, the atomic formula p_i gets translated to $\langle \rightarrow^i \rangle 1$. The atomic path formula $\rightarrow$ gets translated to $\rightarrow^{m+1}$. The translations for the other formulas/path expressions are as expected. □

Now we will show that we can construct an alternating two-way (word) automaton of polynomial size corresponding to an LCPDL formula over words such that the former is nonempty iff the latter is satisfiable.

Lemma 11. *Given a sentence $\Psi \in$ LCPDL(Γ) over words in Γ^+, we can construct, in polynomial time, an alternating two-way automaton $\mathcal{A}_\Psi$ such that $L(\mathcal{A}_\Psi) = L(\Psi)$ (where $L(\Psi)$ denotes the set of words over Γ^+ that are a model of Ψ). Further, the number of states of $\mathcal{A}_\Psi$ is only polynomial in $|\Psi|$, and the number of transitions is polynomial in $|\Psi|$ and $|\Gamma|$.*

PROOF (SKETCH). The construction of the alternating two-way automaton is rather standard. Essentially, the states of the alternating two-way automaton are subformulas of Ψ. The transition function δ of the automaton maps a state and an input letter to a positive boolean combination of $\texttt{States} \times \{-1, 0, 1\}$, where -1 means moving to the left on the word, 0 means staying at the current position of the word, and 1 means moving to the right on the word. The initial state is the sentence Ψ and the accepting states are the formula *true*.

For a sentence $\mathsf{E}\,\psi$, the automaton non-deterministically moves forward or decides to launch the sub-automaton for ψ. For the other formulas, we proceed inductively. If the state (i.e., the current subformula) is a letter a, the transitions ensure that the current letter is a: $\delta(a,a) = \mathit{true}$ and $\delta(a,b) = \mathit{false}$, if $b \neq a$. For conjuction, we have $\delta(\psi_1 \wedge \psi_2, a) = (\psi_1, 0) \wedge (\psi_2, 0)$. For negation $\neg\psi$, we take the complement of the alternating automaton for ψ. Notice that the complementing an alternating two-way automaton just amounts to taking the dual of the transitions: conjunction replaced by disjunction and vice-versa, and *true* replaced with *false* and vice-versa (in fact, it is slightly more complicated than that when path expressions come into play, cf. below). Thus, it does not cause any blow-up in the number of states or size of transitions.

We will now explain the path expressions, before dealing with $\langle \pi \rangle \psi$ and $\mathsf{loop}(\pi)$. A path expression is a rational expression over $\leftarrow$ and $\rightarrow$, and it may have test formulas $\{\psi\}?$ in addition. By following the standard algorithms for rational expression to automata translations, for each path expression π, we get a polynomial-sized automaton $\mathcal{A}_\pi = (S_\pi, \delta_\pi, \mathsf{init}_\pi, F_\pi)$ whose transitions are labeled with $\leftarrow$, $\rightarrow$, or $\{\psi\}?$ subformulas. The alternating two-way automaton simulates a path expression by tracing a successful run of the automaton $\mathcal{A}_\pi$. Thus, the states of the alternating two-way automaton also include the states of the automaton $\mathcal{A}_\pi$. We may assume that the states of $\mathcal{A}_\pi$ and $\mathcal{A}_{\pi'}$ are disjoint if $\pi \neq \pi'$. In our alternating two-way automaton, we have the following transitions: $\delta(s,a)$ contains $(s',-1)$ as a disjunct if $(s,\leftarrow,s') \in \delta_\pi$. Similarly $\delta(s,a)$ contains $(s',1)$ as a disjunct if $(s,\rightarrow,s') \in \delta_\pi$. Further, $\delta(s,a)$ contains $(\psi,0) \wedge (s',0)$ as a disjunct if $(s,\{\psi\}?,s') \in \mathcal{A}_\pi$. Finally, $\delta(s,a)$ contains *true* as a disjunct if s is a final state of $\mathcal{A}_\pi$.

Now, we can give the transition for $\langle\pi\rangle \mathit{true}$. Note that this case is not restrictive since $\langle\pi\rangle\psi \equiv \langle\pi \cdot \{\psi\}?\rangle \mathit{true}$. We have $\delta(\langle\pi\rangle \mathit{true}, a) = (\mathsf{init}_\pi, 0)$ where init_π is the initial state of the automaton $\mathcal{A}_\pi$.

Next, we look at $\mathsf{loop}(\pi)$. Again, we consider the polynomial-sized automaton $\mathcal{A}_\pi$. Following the construction of [26], we will track two states of the automaton $\mathcal{A}_\pi$ — one from the beginning and one from the end. Intuitively, if there is a loop on π, then $\pi = \pi_1 \cdot \pi_2$ such that we reach the same node by π_1 and π_2^{-1}, and moreover, the real moves in the word ($\leftarrow$ and $\rightarrow$) can be taken synchronously. Thus, $\delta(\mathsf{loop}(\pi), a) = \bigvee_{f \in F_\pi}((\mathsf{init}_\pi, f), 0)$. Further, $\delta((s,t),a)$ contains $(\psi,0) \wedge (s',t,0)$ as a disjunct if $(s,\{\psi\}?,s') \in \mathcal{A}_\pi$. Symmetrically, $\delta((s,t),a)$ contains $(\psi,0) \wedge (s,t',0)$ as a disjunct if $(t',\{\psi\}?,t) \in \mathcal{A}_\pi$. Further, $\delta((s,t),a)$ contains the disjunct $(s',t',1)$ if $(s,\rightarrow,s') \in \mathcal{A}_\pi$ and $(t',\leftarrow,t) \in \mathcal{A}_\pi$. Symmetrically, $\delta((s,t),a)$ contains the disjunct $(s',t',-1)$ if $(s,\leftarrow,s') \in \mathcal{A}_\pi$ and $(t',\rightarrow,t) \in \mathcal{A}_\pi$. Finally, $\delta((s,s),a)$ contains *true* as a disjunct.

The above outlined construction suffices if we had to deal with only finite runs. Notice that, even though the language of such an alternating two automaton consists of finite words, it may have infinite runs since it is two-way. Hence, we have to resort to stronger acceptance conditions, such as parity, for the sake of infinite runs. However, even in this case, complementation does not cause any blow-up. $\square$

Fact 1. *[38, 34] Non-emptiness of alternating two-way word automata is in PSPACE.*

From Lemma 10, Lemma 11, and Fact 1 we get:

Theorem 4. *The following problem is PSPACE-complete:*
Input: *A finite set* $\textsc{Props}$ *and a sentence* $\Psi \in \mathrm{LCPDL}(\textsc{Props})$
Question: *Is there a word from* $(2^{\textsc{Props}})^+$ *that is a model of* Ψ *?*

Hardness follows from the PSPACE hardness of satisfiability for LTL and the fact that LTL formulas can be linearly encoded in PDL. Using Lemma 9 and Theorem 4 we get:

Theorem 5. *The following problem is PSPACE-complete:*
Input: *A finite set* $\textsc{Props}$*, a sentence* $\Psi \in \mathrm{LCPDL}(\textsc{Props})$*, and* $B \geq 1$ *encoded in unary*
Question: *Is there a* B*-bounded cylinder over* $\textsc{Props}$ *that is a model of* Φ *?*

6. Conclusion

In this paper, we provided a conceptually new approach to the verification of distributed algorithms.

Actually, we made some assumptions that simplify the presentation, but are not crucial to the approach and results. For example, we assumed that an algorithm is synchronous, i.e., there is a global clock that, at every clock tick, triggers a round, in which every process participates. This can be relaxed to handle communication via (bounded) channels. Though the restriction to the class of rings is crucial for the complexity of our algorithm, the logical framework we developed is largely independent of concrete (ring) architectures. Essentially, we could choose any class of architectures for which LCPDL is decidable.

We leave open whether round-bounded model checking can deal with a more general logic that does not restrict $<$-tests.

Another ambitious goal would be to find automata-theoretic proof techniques that would allow us to relax the *strict* bound on the number of rounds (e.g., in terms of a restriction that varies wrt. the number of processes).

Finally, it would be worthwhile to explore in how far our approach is practical and allows one to verify little protocols such as the leader election algorithm given in the paper. Unfortunately, there do not seem to be tools that could deal with LCPDL or alternating automata on words. One may, however, employ tools for MSO logic though its complexity is a priori much worse [28].

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proceedings of LICS'96*, pages 313–321. IEEE Computer Society Press, 1996.

[2] P. A. Abdulla and G. Delzanno. Parameterized verification. *Int. J. Softw. Tools Technol. Transf.*, 18(5):469–473, October 2016.

[3] P. A. Abdulla, F. Haziza, and L. Holík. Block me if you can! - context-sensitive parameterized verification. In *Proceedings of the 21st International Symposium on Static Analysis (SAS'14)*, volume 8723 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2014.

[4] C. Aiswarya, B. Bollig, and P. Gastin. An automata-theoretic approach to the verification of distributed algorithms. In *CONCUR'15*, volume 42 of *Leibniz International Proceedings in Informatics*, pages 340–353. Leibniz-Zentrum für Informatik, 2015.

[5] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL'11*, pages 599–610. ACM, 2011.

[6] R. Alur, P. Černý, and S. Weinstein. Algorithmic analysis of array-accessing programs. *ACM Trans. Comput. Logic*, 13(3):27:1–27:29, August 2012.

[7] B. Aminof, S. Jacobs, A. Khalimov, and S. Rubin. Parameterized model checking of token-passing systems. In *VMCAI'14*, volume 8318 of *LNCS*, pages 262–281, 2014.

[8] B. Aminof and S. Rubin. Model checking parameterised multi-token systems via the composition method. In *Proceedings of the 8th International Joint Conference on Automated Reasoning (IJCAR'16)*, volume 9706 of *Lecture Notes in Computer Science*, pages 499–515. Springer, 2016.

[9] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, 2004.

[10] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008.

[11] M. Bojanczyk, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM*, 56(3), 2009.

[12] B. Bollig, A. Cyriac, P. Gastin, and K. Narayan Kumar. Model checking languages of data words. In *FoSSaCS'12*, volume 7213 of *LNCS*, pages 391–405. Springer, 2012.

[13] M. Chaouch-Saad, B. Charron-Bost, and S. Merz. A reduction theorem for the verification of round-based distributed algorithms. In *Proceedings of RP'09*, volume 5797 of *LNCS*, pages 93–106. Springer, 2009.

[14] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT Press, 2001.

[15] D. Dolev, M. M. Klawe, and M. Rodeh. An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. *J. Algorithms*, 3(3):245–260, 1982.

[16] E. A. Emerson and V. Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL'04*, volume 3210 of *LNCS*, pages 325–339, 2004.

[17] E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *Int. J. Found. Comput. Sci.*, 14(4):527–550, 2003.

[18] J. Esparza. Keeping a crowd safe: On the complexity of parameterized verification. In *STACS'14*, volume 25 of *LIPIcs*, pages 1–10, 2014.

[19] D. Figueira. *Reasoning on words and trees with data.* PhD thesis, École normale supérieure de Cachan, France, 2010.

[20] D. Figueira and L. Segoufin. Bottom-up automata on data trees and vertical XPath. In *STACS'11*, volume 9 of *LIPIcs*, pages 93–104, 2011.

[21] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[22] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *Proceedings of TACAS'08*, volume 4963 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2008.

[23] W. Fokkink. *Distributed Algorithms: An Intuitive Approach.* MIT Press, 2013.

[24] G. Fontaine, F. Mogavero, A. Murano, G. Perelli, and L. Sorrentino. Cycle detection in computation tree logic. In *Proceedings of the 7th International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2016)*, volume 226 of *EPTCS*, pages 164–177, 2016.

[25] R. Franklin. On an improved algorithm for decentralized extrema finding in circular configurations of processors. *Commun. ACM*, 25(5):336–337, 1982.

[26] S. Göller, M. Lohrey, and C. Lutz. PDL with intersection and converse: satisfiability and infinite-state model checking. *J. Symb. Log.*, 74(1):279–314, 2009.

[27] N. Habermann. Parallel neighbor-sort (or the glory of the induction principle). Technical Report Paper 2087, Carnegie Mellon University - Computer Science Departement, 1972.

[28] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[29] I. Konnov, H. Veith, and J. Widder. Who is afraid of model checking distributed algorithms?, 2012.

[30] I. Konnov, H. Veith, and J. Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In *Proceedings of CONCUR'14*, volume 8704 of *LNCS*, pages 125–140. Springer, 2014.

[31] D. Kozen. Lower bounds for natural proof systems. In *FOCS'77*, pages 254–266. IEEE Computer Society, 1977.

[32] M. Lange. Model checking propositional dynamic logic with all extras. *J. Applied Logic*, 4(1):39–49, 2006.

[33] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

[34] R. Mennicke. Propositional dynamic logic with converse and repeat for message-passing systems. *Logical Methods in Computer Science*, 9(2:12):1–35, 2013.

[35] G. L. Peterson. An o(n log n) unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.*, 4(4):758–762, 1982.

[36] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS'05*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[37] S. Rubin, F. Zuleger, A. Murano, and B. Aminof. Verification of asynchronous mobile-robots in partially-known environments. In *Proceedings of the 18th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA'15)*, volume 9387 of *Lecture Notes in Computer Science*, pages 185–200. Springer, 2015.

[38] O. Serre. Parity games played on transition graphs of one-counter processes. In *FOSSACS'06*, volume

3921, pages 337–351. Springer, 2006.

[39] A. Siirtola and J. Kortelainen. Multi-parameterised compositional verification of safety properties. *Inf. Comput.*, 244:23–48, 2015.

[40] T. Tan. Extending two-variable logic on data trees with order on data values and its automata. *ACM Trans. Comput. Log.*, 15(1):8, 2014.

[41] M. Y. Vardi. Reasoning about the past with two-way automata. In *ICALP'98*, LNCS, pages 628–641. Springer, 1998.