An Automata-Theoretic Approach to the Verification of Distributed Algorithms*

C. Aiswarya[1], Benedikt Bollig[2], and Paul Gastin[2]

1 **Uppsala University**
aiswarya.cyriac@it.uu.se
2 **LSV, ENS Cachan, CNRS, Inria**
{bollig,gastin}@lsv.ens-cachan.fr

Abstract

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. Since the verification of distributed algorithms 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. 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.

1998 ACM Subject Classification C.2.4 Distributed Systems; F.3.1 Specifying and Verifying and Reasoning about Programs

1 Introduction

Distributed algorithms are a classic discipline of computer science and continue to be an active field of research [17]. 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. Prominent examples are 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 [12] or the Dolev-Klawe-Rodeh algorithm [7], 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 intrinsically hard to analyze. Correctness proofs are often intricate and use subtle inductive arguments. Therefore, it is worthwhile to consider automatic

* Supported by LIA InForMel.

verification methods such as model checking. 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 [4, 11].

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 [10]; second, processes manipulate data from an infinite domain [4, 11]. In each case, even simple verification questions are undecidable, and so is the combination of both.

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. This is 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. The number of rounds is often exponentially smaller than the number of processes. Thus, 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 a (round-bounded) execution of a distributed algorithm symbolically as a word-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) over words [13]. 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. The latter can be reduced to an emptiness problem for alternating two-way automata over words so that we obtain a PSPACE procedure for round-bounded model checking.

Related Work: Considerable effort has been devoted to the formal verification of fault-tolerant algorithms, which have to cope with faults such as lost or corrupted messages (e.g., [6, 15]). After all, there have been only very few generic approaches to model checking distributed algorithms. In [14], 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. Parameterized model checking of ring-based systems where communication is subject to a token policy and the message alphabet is finite has been studied in [3, 8, 9]. In [8], cutoff results are obtained for LTL\X specifications when a bound is placed on the number of times a token may change values.

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 [4], or for the analysis of programs with data structures such as lists [2]. The difference to our work is that we model distributed algorithms and provide a logical specification language which borrows concepts from [4, 11]. The paper [5] pursued a symbolic model-checking approach to *sequential* systems involving data, but pids could only be compared for equality. The ordering on the data domain has a subtle impact on the choice of the specification language.

Full proofs can be found in the full version of the paper [1].

2 Distributed Algorithms

By $\mathbb{N} = \{0, 1, 2, \ldots\}$, we denote the set of natural numbers. For $n \in \mathbb{N}$, we set $[n] = \{1, \ldots, n\}$ and $[n]_0 = \{0, 1, \ldots, n\}$. 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 consists of a finite number of processes, each having a unique process identifier (pid). Every process has a unique left neighbor (referred to by **left**) and a unique right neighbor (referred to by **right**). Formally, a *ring* is a tuple $\mathcal{R} = (n : p_1, \ldots, p_n)$, given by its size $n \geq 1$ and the pids $p_i \in \mathbb{N}$ assigned to processes $i \in [n]$. We require that pids are unique, i.e., $p_i \neq p_j$ whenever $i \neq j$. For a process $i < n$, process $i + 1$ is the right neighbor of i. Moreover, 1 is the right neighbor of n. Analogously, if $i > 1$, then $i - 1$ is the left neighbor of i. Moreover, n is the left neighbor of 1. Thus, processes 1 and n must not be considered as the "first" or "last" process. Actually, a distributed algorithm will not be able to distinguish between, for example, $(4 : 4, 1, 5, 2)$ and $(4 : 5, 2, 4, 1)$.

One given distributed algorithm can be run on *any* ring. It is given by a single program $\mathcal{D}$, and each process runs a copy of $\mathcal{D}$. It is convenient to think of $\mathcal{D}$ as a (finite) automaton. Processes proceed in synchronous rounds. In one round, *every* process executes one transition of its program. In addition to changing its state, each process may optionally perform the following phases within a round: (i) send some pids to its neighbors, (ii) receive pids from its neighbors and store them in registers, (iii) compare register contents with one another, (iv) update its registers. For example, consider the transition $t = \langle s\colon \mathbf{left}!r\,;\mathbf{right}!r'\,;\mathbf{right}?r'\,;r < r'\,;r := r'\,;\mathbf{goto}\ s'\rangle$. A process can execute t if it is in state s. It then sends the contents of register r to its left neighbor and the contents of r' to its right neighbor. If, afterwards, it receives a pid p from its right neighbor, it stores p in r'. If p is greater than what has been stored in r, it sets r to p and goes to state s'. Otherwise, the transition is not applicable. The first phase can, alternatively, be filled with a special command **fwd**. Then, a process will just forward any pid it receives. Note that a message can be forwarded, in one and the same round, across several processes executing **fwd**.

▶ **Definition 1.** A *distributed algorithm* $\mathcal{D} = (S, s_0, \mathit{Reg}, \Delta)$ consists of a nonempty finite set S of *(local) states*, an *initial state* $s_0 \in S$, a nonempty finite set *Reg* of *registers*, and a nonempty finite set Δ of *transitions*. 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 as follows:

$$
\begin{aligned}
\mathit{send} &::= \mathbf{skip} \mid \mathbf{fwd} \mid \mathbf{left}!r \mid \mathbf{right}!r \mid \mathbf{left}!r\,;\mathbf{right}!r' \\
\mathit{rec} &::= \mathbf{skip} \mid \mathbf{left}?r \mid \mathbf{right}?r \mid \mathbf{left}?r\,;\mathbf{right}?r' \\
\mathit{guard} &::= \mathbf{skip} \mid r < r' \mid r = r' \mid \mathit{guard}\,;\mathit{guard} \\
\mathit{update} &::= \mathbf{skip} \mid r := r' \mid \mathit{update}\,;\mathit{update}
\end{aligned}
$$

with r and r' ranging over *Reg*. We require that (1) in a *rec* statement of the form $\mathbf{left}?r\,;\mathbf{right}?r'$, we have $r \neq r'$ (actually, the order of the two receive actions does not matter), and (2) in an *update* statement, every register occurs at most once as a left-hand side. In the following, occurrences of "**skip**;" are omitted. ◀

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 result.

states: *active*, *passive*, *found*	$t_1 = \langle active\colon \mathbf{left}!id\,;\mathbf{right}!id\,;\mathbf{left}?r_1\,;\mathbf{right}?r_2\,;r_1 < id\,;r_2 < id\,;\mathbf{goto}\ active\rangle$
initial state: *active*	$t_2 = \langle active\colon$ ______ $;id < r_1\,;\mathbf{goto}\ passive\rangle$
registers: id, r, r_1, r_2	$t_3 = \langle active\colon$ ______ $;id < r_2\,;\mathbf{goto}\ passive\rangle$
	$t_4 = \langle active\colon$ ______ $;id = r_1\,;r := id\,;\mathbf{goto}\ found\rangle$
	$t_5 = \langle passive\colon \mathbf{fwd}\,;\mathbf{left}?r\,;\mathbf{goto}\ passive\rangle$

Figure 1 Franklin's leader-election algorithm $\mathcal{D}_{\mathsf{Franklin}}$

states: $active_0, active_1$, *passive*, *found*	$t_1 = \langle active_0\colon \mathbf{right}!r\,;\mathbf{left}?r'\,;\mathbf{goto}\ active_1\rangle$
initial state: $active_0$	$t_2 = \langle active_1\colon \mathbf{right}!r'\,;\mathbf{left}?r''\,;r'' < r'\,;r < r'\,;r := r'\,;\mathbf{goto}\ active_0\rangle$
registers: id, r, r', r''	$t_3 = \langle active_1\colon$ ______ $;r' < r\,;\mathbf{goto}\ passive\rangle$
	$t_4 = \langle active_1\colon$ ______ $;r' < r''\,;\mathbf{goto}\ passive\rangle$
	$t_5 = \langle active_1\colon$ ______ $;r = r'\,;\mathbf{goto}\ found\rangle$
	$t_6 = \langle passive\colon \mathbf{fwd}\,;\mathbf{left}?r\,;\mathbf{goto}\ passive\rangle$

Figure 2 Dolev-Klawe-Rodeh leader-election algorithm $\mathcal{D}_{\mathsf{DKR}}$

At the beginning of an execution of an algorithm, every register contains the pid of the respective process. We also assume, wlog., that there is a special register $id \in Reg$ that is never updated, i.e., no transition contains a command of the form $\mathbf{left}?id$, $\mathbf{right}?id$, or $id := r$. A process can thus, at any time, access its own pid in terms of id.

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). As, moreover, the order of two sends and the order of two receives within a transition do not matter, this will allow us to identify a transition with the set of states, commands (apart from **skip**), and guards that it contains. For example, $t = \langle s\colon \mathbf{left}!r\,;\mathbf{right}!r'\,;\mathbf{right}?r'\,;r < r'\,;r := r'\,;\mathbf{goto}\ s'\rangle$ is considered as the set $t = \{s\,,\ \mathbf{left}!r\,,\ \mathbf{right}!r'\,,\ \mathbf{right}?r'\,,\ r < r'\,,\ r := r'\,,\ \mathbf{goto}\ s'\}$.

Before defining the semantics of a distributed algorithm, we will look at two examples.

▶ **Example 2** (Franklin's Leader-Election Algorithm)**.** Consider Franklin's algorithm $\mathcal{D}_{\mathsf{Franklin}}$ to determine a leader in a ring [12]. It is given in Figure 1. The goal is to assign leadership to the process with the highest pid. To do so, every process sends its own pid to both neighbors, receives the pids of its left and right neighbor, and stores them in registers r_1 and r_2, respectively (transitions $t_1, \ldots, t_4$). If a process is a local maximum, i.e., $r_1 < id$ and $r_2 < id$ hold, it is still in the race for leadership and stays in state *active*. Otherwise, it has to take t_2 or t_3 and goes into state *passive*. In *passive*, a process will just forward any pid it receives and store the message coming from the left in r (transition t_5). Notice that, within the same round, a message may be forwarded through (and stored by) several consecutive passive processes, until it reaches an active one. When an active process receives its own pid (transition t_4), it knows it is the only remaining active process. It copies its own pid into r, which henceforth refers to the leader. We may say that a run is accepting (or terminating) when all processes terminate in *passive* or *found*. Then, at the end of any accepting run, (i) there is exactly one process i_0 that terminates in *found*, (ii) all processes store the pid of i_0 in register r, and the pid of i_0 is the maximum of all pids in the ring. Since, in every round, at least half of the active processes become passive, the algorithm terminates after at most $\lfloor \log_2 n \rfloor + 1$ rounds where n is the number of processes. ◀

▶ **Example 3** (Dolev-Klawe-Rodeh Leader-Election Algorithm)**.** The Dolev-Klawe-Rodeh leader-election algorithm [7] is an adaptation of Franklin's algorithm to cope with unidirectional

rings, where a process can only, say, send to the right and receive from the left. The algorithm, denoted $\mathcal{D}_{\mathsf{DKR}}$, is given in Figure 2. The idea is that the local maximum among the processes $i-2, i-1, i$ is 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*, which has the same task as in Franklin's algorithm. 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. Note that the correctness of $\mathcal{D}_{\mathsf{DKR}}$ is less clear than that of $\mathcal{D}_{\mathsf{Franklin}}$. ◀

Semantics of Distributed Algorithms. Now, we give the formal semantics of a distributed algorithm $\mathcal{D} = (S, s_0, \mathit{Reg}, \Delta)$. Recall that $\mathcal{D}$ can be run on any ring $\mathcal{R} = (n : p_1, \ldots, p_n)$. An ($\mathcal{R}$-)configuration of $\mathcal{D}$ is a tuple $(s_1, \ldots, s_n, \rho_1, \ldots, \rho_n)$ where s_i is the current state of process i and $\rho_i : \mathit{Reg} \to \{p_1, \ldots, p_n\}$ maps each register to a pid. The configuration is called *initial* if, for all processes $i \in [n]$, we have $s_i = s_0$ and $\rho_i(r) = p_i$ for all $r \in \mathit{Reg}$. Note that there is a unique initial $\mathcal{R}$-configuration.

In one round, the algorithm moves from one configuration to another one. This is described by a relation $C \overset{t}{\rightsquigarrow} C'$ where $C = (s_1, \ldots, s_n, \rho_1, \ldots, \rho_n)$ and $C' = (s'_1, \ldots, s'_n, \rho'_1, \ldots, \rho'_n)$ are $\mathcal{R}$-configurations and $t = (t_1, \ldots, t_n) \in \Delta^n$ is a tuple of transitions where t_i is executed by process i. To determine when $C \overset{t}{\rightsquigarrow} C'$ holds, we first define two auxiliary relations. For registers $r, r' \in \mathit{Reg}$ and processes $i, j \in [n]$, 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'. Thus, we require that

$$\textbf{right}!r \in t_i \ \wedge\ \textbf{left}?r' \in t_j \ \wedge\ \textbf{fwd} \in t_k \text{ for all } k \in \mathit{Between}(i, j)$$

where $\mathit{Between}(i, j)$ means $\{i+1, \ldots, j-1\}$ if $i < j$ or $\{1, \ldots, j-1, i+1, \ldots, n\}$ if $j \leq i$. Note that, due to the **fwd** command, $r@i \rightarrowtail r'@j$ may hold for several r' and j. The meaning of $r'@j \leftarrowtail r@i$ is analogous, we just replace "right direction" by "left direction":

$$\textbf{left}!r \in t_i \ \wedge\ \textbf{right}?r' \in t_j \ \wedge\ \textbf{fwd} \in t_k \text{ for all } k \in \mathit{Between}(j, i).$$

The guards in the transitions $t_1, \ldots, t_n$ are checked against "intermediate" register assignments $\hat{\rho}_1, \ldots, \hat{\rho}_n : \mathit{Reg} \to \{p_1, \ldots, p_n\}$, 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, we write $C \overset{t}{\rightsquigarrow} C'$ if, for all $j \in [n]$ and $r, r' \in \mathit{Reg}$, the following hold:

1. $s_j \in t_j$ and $(\textbf{goto } s'_j) \in t_j$,
2. $\hat{\rho}_j(r) < \hat{\rho}_j(r')$ if $(r < r') \in t_j$,
3. $\hat{\rho}_j(r) = \hat{\rho}_j(r')$ if $(r = r') \in t_j$,
4. $\rho'_j(r) = \begin{cases} \hat{\rho}_j(r') & \text{if } (r := r') \in t_j \\ \hat{\rho}_j(r) & \text{if } t_j \text{ does not contain an update of the form } r := r'' \end{cases}$

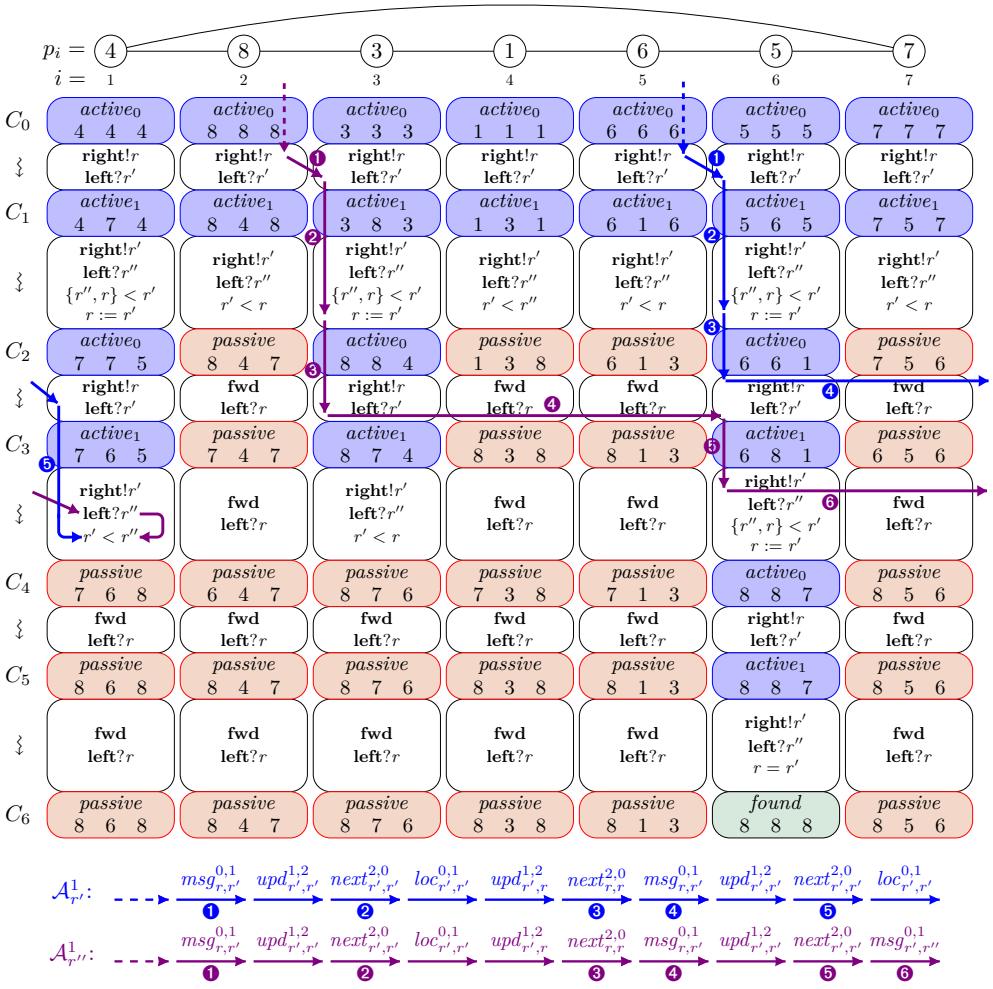

Figure 3 Run of Dolev-Klawe-Rodeh algorithm and runs of path automata

Again, 4. is well-defined thanks to condition (2) in Definition 1.

An ($\mathcal{R}$-)*run* of $\mathcal{D}$ is a sequence $\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \cdots \overset{t^k}{\rightsquigarrow} C_k$ where $k \geq 1$, C_0 is the initial $\mathcal{R}$-configuration, and $t^j = (t^j_1, \ldots, t^j_n) \in \Delta^n$ for all $j \in [k]$. We call k the *length* of χ. Note that χ uniquely determines the underlying ring $\mathcal{R}$.

▶ Remark. A receive command is always non-blocking even if there is no corresponding send. As an alternative semantics, one could require that it can only be executed if there has been a matching send, or vice versa. One could even include tags from a finite alphabet that can be sent along with pids. All this will not change any of the forthcoming results. ◀

▶ **Example 4.** A run of $\mathcal{D}_{\mathsf{DKR}}$ from Example 3 on the ring $\mathcal{R} = (7 : 4, 8, 3, 1, 6, 5, 7)$ 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, $r'@3 \rightarrowtail r@4$ and $r'@3 \rightarrowtail r''@6$. Moreover, $r'@6 \rightarrowtail r@7$ and $r'@6 \rightarrowtail r''@1$ (recall that we are in a ring). Note that the run conforms to the correctness property formulated in Example 3. In particular, in the final configuration, all processes store the maximum pid in register r. ◀

3 The Specification Language

In Examples 2 and 3, we informally stated the correctness criterion for the presented algorithms (e.g., "at the end, all processes store the maximal pid in register r"). Now, we introduce a *formal* language to specify correctness properties. It is defined wrt. a given distributed algorithm $\mathcal{D} = (S, s_0, Reg, \Delta)$, which we fix for the rest of this section.

Typically, one requires that a distributed algorithm is correct no matter what the underlying ring is. Since we will bound the number of rounds, we moreover study a form of partial correctness. Accordingly, a property is of the form $\forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\varphi$, which has to be read as "for all rings, all runs, and all processes m, we have φ". The marking m is used to avoid to "get lost" in a ring when writing the property φ. This is like placing a pebble in the ring that can be retrieved at any time. Actually, φ allows us to "navigate" back and forth ($\uparrow$ and $\downarrow$) in a run, i.e., from one configuration to the previous or next one (similar to a temporal logic with past operators). By means of $\leftarrow$ and $\rightarrow$, we may also navigate horizontally within a configuration, i.e., from one process to a neighboring one.

Essentially, a sequence of configurations is interpreted as a cylinder (cf. Figure 3) that can be explored using regular expressions π over $\{\epsilon, \leftarrow, \rightarrow, \uparrow, \downarrow\}$ (where ϵ means "stay"). At a given position/coordinate of the cylinder, we can check *local (or positional)* properties like the state taken by a process, or whether we are on the marked process m. Such a property can be combined with a regular expression π: The formula $[\pi]\varphi$ says that φ holds at every position that is reachable through a π-path (a path matching π). Dually, $\langle\pi\rangle\varphi$ holds if there is a π-path to some position where φ is satisfied. The most interesting construct in our logic is $\langle\pi\rangle r \bowtie \langle\pi'\rangle r'$, where $\bowtie \in \{=, \neq, <, \leq\}$, which has been used for reasoning about XML documents [4, 11]. 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 pid stored in register r at y and the pid stored in r' at y' satisfy the relation $\bowtie$.

We will now introduce our logic in full generality. Later, we will restrict the use of $<$- and $\leq$-guards to obtain positive results.

▶ **Definition 5.** The logic DataPDL($\mathcal{D}$) is given by the following grammar:

$$\begin{aligned}
\Phi &::= \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\varphi \\
\varphi, \varphi' &::= \mathsf{m} \mid s \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \varphi \Rightarrow \varphi' \mid [\pi]\varphi \mid \langle\pi\rangle r \bowtie \langle\pi'\rangle r' \\
\pi, \pi' &::= \{\varphi\}? \mid d \mid \pi + \pi' \mid \pi \cdot \pi' \mid \pi^*
\end{aligned}$$

where $s \in S$, $r, r' \in Reg$, $\bowtie \in \{=, \neq, <, \leq\}$, and $d \in \{\epsilon, \leftarrow, \rightarrow, \uparrow, \downarrow\}$. ◀

We call φ a *local formula*, and π a *path formula*. We use common abbreviations such as $\mathit{false} = \mathsf{m} \wedge \neg\mathsf{m}$, $\langle\pi\rangle\varphi = \neg[\pi]\neg\varphi$, and $\varphi \vee \varphi' = \neg(\neg\varphi \wedge \neg\varphi')$, and we may write $\pi\pi'$ instead of $\pi \cdot \pi'$. Implication $\Rightarrow$ is included explicitly in view of the restriction defined below.

Next, we define the semantics. Consider a run $\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \cdots \overset{t^k}{\rightsquigarrow} C_k$ of $\mathcal{D}$ where $C_j = (s_1^j, \ldots, s_n^j, \rho_1^j, \ldots, \rho_n^j)$, i.e., n is the number of processes in the underlying ring. A local formula φ is interpreted over χ wrt. a marked process $m \in [n]$ and a position $(i, j) \in Pos(\chi)$ where $Pos(\chi) = [n] \times [k]_0$. Let us define when $\chi, m, (i, j) \models \varphi$ holds. The operators $\neg$, $\wedge$, and $\Rightarrow$ are as usual. Moreover, $\chi, m, (i, j) \models \mathsf{m}$ if $i = m$, and $\chi, m, (i, j) \models s$ if $s_i^j = s$.

The other local formulas use path formulas. The semantics of a path formula π is given in terms of a binary relation $[\![\pi]\!]_{\chi,m} \subseteq Pos(\chi) \times Pos(\chi)$, which we define below. First, we set:

- $\chi, m, (i, j) \models [\pi]\varphi$ if $\forall (i', j')$ such that $((i, j), (i', j')) \in [\![\pi]\!]_{\chi,m}$, we have $\chi, m, (i', j') \models \varphi$

- $\chi, m, (i,j) \models \langle\pi\rangle r \bowtie \langle\pi'\rangle r'$ (where $\bowtie \in \{=, \neq, <, \leq\}$) if $\exists (i_1,j_1),(i_2,j_2)$ such that $((i,j),(i_1,j_1)) \in [\![\pi]\!]_{\chi,m}$ and $((i,j),(i_2,j_2)) \in [\![\pi']\!]_{\chi,m}$ and $\rho_{i_1}^{j_1}(r) \bowtie \rho_{i_2}^{j_2}(r')$

It remains to define $[\![\pi]\!]_{\chi,m}$ for a path formula π. First, a local test and a stay ϵ do not "move" at all: $[\![\{\varphi\}?]\!]_{\chi,m} = \{(x,x) \mid x \in Pos(\chi)$ such that $\chi, m, x \models \varphi\}$, and $[\![\epsilon]\!]_{\chi,m} = \{(x,x) \mid x \in Pos(\chi)\}$. Using $\rightarrow$, we move to the right neighbor of a process: $[\![\rightarrow]\!]_{\chi,m} = \{((i,j),(i+1,j)) \mid i \in [n-1]$ and $j \in [k]_0\} \cup \{((n,j),(1,j)) \mid j \in [k]_0\}$. We define $[\![\leftarrow]\!]_{\chi,m}$ accordingly. Moreover, $[\![\downarrow]\!]_{\chi,m} = \{((i,j),(i,j+1)) \mid i \in [n]$ and $j \in [k-1]_0\}$, and similarly for $[\![\uparrow]\!]_{\chi,m}$. The regular constructs, $+$, $\cdot$, and $*$ are as expected and refer to the union, relation composition, and star over binary relations.

Finally, $\mathcal{D}$ satisfies the DataPDL formula $\forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\varphi$, written $\mathcal{D} \models \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\varphi$, if, for all rings $\mathcal{R} = (n : \ldots)$, all $\mathcal{R}$-runs χ, and all processes $m \in [n]$, we have $\chi, m, (m,0) \models \varphi$. Thus, φ is evaluated at the first configuration, wrt. process m which can be chosen arbitrarily.

Next, we define a restricted logic, DataPDL$^{\ominus}(\mathcal{D})$, for which we later present our main result. We say that a path formula π is *unambiguous* if, from a given position, it defines at most one reference point. Formally, for all rings $\mathcal{R} = (n : \ldots)$, $\mathcal{R}$-runs χ of $\mathcal{D}$, processes $m \in [n]$, and positions $x \in Pos(\chi)$, there is at most one $x' \in Pos(\chi)$ such that $(x,x') \in [\![\pi]\!]_{\chi,m}$. For example, ϵ, $\downarrow$, $\rightarrow$, and $\rightarrow^*\{\mathsf{m}\}?$ are unambiguous, while $\rightarrow^*$ and $\leftarrow + \rightarrow$ are not unambiguous.

▶ **Definition 6.** A DataPDL($\mathcal{D}$) formula is contained in DataPDL$^{\ominus}(\mathcal{D})$ if every subformula $\varphi = \langle\pi\rangle r \bowtie \langle\pi'\rangle r'$ with $\bowtie \in \{<, \leq\}$ is such that π and π' are *unambiguous*. Moreover, φ must *not* occur (i) in the scope of a negation, (ii) on the left-hand side of an implication $\_ \Rightarrow \_$, or (iii) within a test $\{\_\}?$. Note that guards using $=$ and $\neq$ are still unrestricted. ◀

▶ **Example 7.** Let us *formalize*, in DataPDL$^{\ominus}(\mathcal{D})$, the correctness criteria for $\mathcal{D}_{\mathsf{Franklin}}$ and $\mathcal{D}_{\mathsf{DKR}}$ that we stated informally in Examples 2 and 3. Consider the following local formulas:

$$\varphi_{\mathsf{last}} = [\downarrow]\mathit{false} \qquad \varphi_{\mathsf{max}} = [\rightarrow^*]\big(\langle\epsilon\rangle id \leq \langle\pi_{\mathsf{found}}\rangle r\big)$$
$$\varphi_{\mathsf{acc}} = [\rightarrow^*](\mathit{passive} \vee \mathit{found}) \qquad \varphi_{r=id} = \langle\pi_{\mathsf{found}}\rangle\big(\langle\epsilon\rangle r = \langle\epsilon\rangle id\big)$$
$$\varphi_{\mathsf{found}} = \langle\pi_{\mathsf{found}}\rightarrow(\{\neg\mathit{found}\}?\rightarrow)^*\rangle\mathsf{m} \qquad \varphi_{r=r} = \neg\big(\langle\epsilon\rangle r \neq \langle\rightarrow^*\rangle r\big)$$

where $\pi_{\mathsf{found}} = (\{\neg\mathit{found}\}?\rightarrow)^*\{\mathit{found}\}?$. Note that π_{found} is unambiguous: while going to the right, it always stops at the *nearest* process that is in state *found*. Thus, φ_{max} is indeed a local DataPDL$^{\ominus}$ formula. Consider the DataPDL$^{\ominus}$ formula

$$\Phi_1 = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}[\downarrow^*]\big((\varphi_{\mathsf{last}} \wedge \varphi_{\mathsf{acc}}) \Rightarrow (\varphi_{\mathsf{found}} \wedge \varphi_{\mathsf{max}} \wedge \varphi_{r=r} \wedge \varphi_{r=id})\big)\,.$$

It says that, at the end (i.e., in the last configuration) of each accepting run, expressed by $[\downarrow^*]\big((\varphi_{\mathsf{last}} \wedge \varphi_{\mathsf{acc}}) \Rightarrow \cdots\big)$, we have that (i) there is exactly one process i_0 that ends in state *found* (guaranteed by φ_{found}), (ii)register r of i_0 contains the maximum over all pids (φ_{max}), (iii)register r of i_0 contains the pid of i_0 itself ($\varphi_{r=id}$), and (iv) all processes store the same pid in r ($\varphi_{r=r}$). Thus, $\mathcal{D}_{\mathsf{Franklin}} \models \Phi_1$. On the other hand, we have $\mathcal{D}_{\mathsf{DKR}} \not\models \Phi_1$, because in $\mathcal{D}_{\mathsf{DKR}}$ the process that ends in *found* is not necessarily the process with the maximum pid. However, we still have $\mathcal{D}_{\mathsf{DKR}} \models \Phi_2$ where

$$\Phi_2 = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}[\downarrow^*]\big((\varphi_{\mathsf{last}} \wedge \varphi_{\mathsf{acc}}) \Rightarrow (\varphi_{\mathsf{found}} \wedge \varphi_{\mathsf{max}} \wedge \varphi_{r=r})\big)\,.$$

The next example formulates the correctness constraint for a distributed sorting algorithm. We would like to say that, at the end of an accepting run, the pids stored in registers r are strictly totally ordered. Suppose φ_{acc} represents an acceptance condition and φ_{least} says that

there is exactly one process that terminates in some dedicated state *least*, similarly to φ_{found} above. Then,

$$\Phi_3 = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}[\downarrow^*]\big((\varphi_{\mathsf{last}} \wedge \varphi_{\mathsf{acc}}) \Rightarrow (\varphi_{\mathsf{least}} \wedge [\rightarrow^*\{\neg least\}?](\langle\leftarrow\rangle r < \langle\epsilon\rangle r))\big)$$

makes sure that, whenever process j is not terminating in *least*, its left neighbor i stores a smaller pid in r than j does.

Note that Φ_1, Φ_2, and Φ_3 are indeed DataPDL$^{\ominus}$ formulas. ◀

▶ **Example 8.** We give a couple of examples to illustrate how the logic can be used to reason about temporal properties. Consider the specifications: (a) Every process remains active until it becomes found or passive forever; (b) The value of the register r on any process is monotonously non-decreasing. These can be expressed by the DataPDL$^{\ominus}$ formulas below:

$$\Phi_a = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\langle(\{active\}?\downarrow)^*\rangle(found \vee [\downarrow^*](passive))\,.$$

$$\Phi_b = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}[\downarrow^*]([\downarrow]false \vee (\langle\epsilon\rangle r \leq \langle\downarrow\rangle r))\,.$$ ◀

Unsurprisingly, model checking distributed algorithms against DataPDL$^{\ominus}$ is undecidable:

▶ **Theorem 9.** *The following problem is undecidable: Given a distributed algorithm $\mathcal{D}$ and $\Phi \in$ DataPDL$^{\ominus}(\mathcal{D})$, do we have $\mathcal{D} \models \Phi$? (Actually, this even holds for formulas Φ that express simple state-reachability properties and do not use any guards on pids.)*

4 Round-Bounded Model Checking

In situations where model checking is undecidable, a fruitful approach has been to underapproximate the behavior of a system. The idea is to introduce a parameter that measures a characteristic of a run. One then imposes a bound on this parameter and explores all behaviors up to that bound. In numerous distributed algorithms (cf. Examples 2 and 3), the number b of rounds needed to conclude is exponentially smaller than the number of processes, Recall that, in a single round, a message may be forwarded through an arbitrarily long sequence of processes. Therefore, b seems to be a promising parameter for bounded model checking of distributed algorithms.

For a distributed algorithm $\mathcal{D}$, a formula $\Phi = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\varphi \in$ DataPDL$(\mathcal{D})$, and $b \geq 1$, we write $\mathcal{D} \models_b \Phi$ if, for all rings $\mathcal{R} = (n : \ldots)$, all $\mathcal{R}$-runs χ of length $k \leq b$, and all processes $m \in [n]$, we have $\chi, m, (m, 0) \models \varphi$. We now present our main result:

▶ **Theorem 10.** *The following problem is* PSPACE*-complete: Given a distributed algorithm $\mathcal{D}$, $\Phi \in$ DataPDL$^{\ominus}(\mathcal{D})$, and a natural number $b \geq 1$ (encoded in unary), do we have $\mathcal{D} \models_b \Phi$?*

The lower-bound can be obtained by a reduction from the intersection-emptiness problem for a list of finite automata. 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 [16], 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, but 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 a satisfiability check in the well-studied propositional dynamic logic with loop and converse (LCPDL) [13] on tables of bounded height. In Theorem 11 we show that this satifiability problem can be solved in PSPACE by a reduction to an emptiness check of alternating two-way automata (A2As) [21] over words. The "naïve" approaches, on the other hand, do not seem to give rise to viable algorithms. Finally, our approach is uniform in the following sense: We will construct, in polynomial time, an LCPDL formula that describes precisely the symbolic abstractions of runs (over arbitrary rings) that violate (or satisfy) a given formula. Our construction is *independent* of the parameter b. The satisfiability check then requires a bound on the number of rounds (or on the number of processes), which can be adjusted gradually without changing the automaton.

Proof Outline for Upper Bound of Theorem 10. Let $\mathcal{D}$ be the given distributed algorithm and $\Phi \in \text{DataPDL}^{\ominus}(\mathcal{D})$. We will reduce model checking to the satisfiability problem for LCPDL [13]. While DataPDL$^{\ominus}$ is interpreted over runs, containing pids from an infinite alphabet, the new logic will reason about symbolic abstractions over a *finite* alphabet. A symbolic abstraction of a run only keeps the transitions and discards pids. Thus, it can be seen as a table whose entries are transitions (cf. Figure 3).

First, we translate $\mathcal{D}$ into an LCPDL formula. Essentially, it checks that guards are not used in a contradictory way. To compare $\mathcal{D}$ with Φ, the latter is translated into an LCPDL formula, too. However, there is a subtle point here. For simplicity, 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 $r < r'$ or $r' < r$ on the basis of a symbolic run. This is the reason why DataPDL$^{\ominus}$ restricts $<$- and $\leq$-tests. It is then indeed enough to reason about symbolic runs (cf. Lemma 13 below). We leave open whether one can deal with full DataPDL.

Overall, we reduce model checking to 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. Satisfiability of LCPDL over symbolic runs (of bounded height) can be checked in PSPACE as stated in Theorem 11. Our approach is, thus, automata theoretic in spirit, though the power of alternation is used differently than in [20], which translates LTL formulas into automata.

Next, we present the logic LCPDL over symbolic runs. Then, we translate $\mathcal{D}$ as well as its DataPDL$^{\ominus}$ specification into LCPDL. For the remainder of this section, we fix a distributed algorithm $\mathcal{D} = (S, s_0, \mathit{Reg}, \Delta)$.

PDL with Loop and Converse (LCPDL). As mentioned before, a symbolic abstraction of a run of $\mathcal{D}$ is a table, whose entries are transitions from the finite alphabet Δ. A *table* is a triple $T = (n, k, \lambda)$ where $n, k \geq 1$ and $\lambda : \mathit{Pos}(T) \to \Delta$ labels each position/coordinate from $\mathit{Pos}(T) = [n] \times [k]_0$ with a transition. Thus, we may consider that T has n columns and $k+1$ rows. In the following, we will write $T[i,j]$ for $\lambda(i,j)$, and $T[i]$ for the i-th column of T, i.e., $T[i] = T[i,0] \cdots T[i,k] \in \Delta^+$. Let Δ^{++} denote the set of all tables.

Formulas $\psi \in \text{LCPDL}(\mathcal{D})$ are interpreted over tables. Their syntax is given as follows:

$$\begin{aligned}
\psi, \psi' \;&::=\; t \mid s \mid \textbf{goto}\ s \mid \textbf{fwd} \mid \textbf{left}!r \mid \textbf{right}!r \mid \textbf{left}?r \mid \textbf{right}?r \mid r < r' \mid r = r' \mid r := r' \mid \\
&\quad\; \neg\psi \mid \psi \wedge \psi' \mid \langle\pi\rangle\psi \mid \mathsf{loop}(\pi) \\
\pi, \pi' \;&::=\; \{\psi\}? \mid d \mid \pi + \pi' \mid \pi \cdot \pi' \mid \pi^* \mid \pi^{-1} \mid \mathcal{A}
\end{aligned}$$

where $t \in \Delta$, $s \in S$, $r, r' \in \mathit{Reg}$, $d \in \{\epsilon, \rightarrow, \downarrow\}$, and $\mathcal{A}$ is a *path automaton*[1]: a non-deterministic finite automaton whose transitions are labeled with path formulas π. Again, ψ is called a *local formula*. We use common abbreviations for disjunction, implication, *true*, and *false*, and we let $\pi^+ = \pi \cdot \pi^*$, $[\pi]\psi = \neg\langle\pi\rangle\neg\psi$, $\langle\pi\rangle = \langle\pi\rangle \mathit{true}$, $\leftarrow = \rightarrow^{-1}$, and $\uparrow = \downarrow^{-1}$.

The semantics of LCPDL is very similar to that of DataPDL. A local formula ψ is interpreted over a table $T \in \Delta^{++}$ and a position $x \in \mathit{Pos}(T)$. When it is satisfied, we write $T, x \models \psi$. Moreover, a path formula π determines a binary relation $[\![\pi]\!]_T \subseteq \mathit{Pos}(T) \times \mathit{Pos}(T)$, relating those positions that are connected by a path matching π.

We consider only the most important cases: We have $T, (i,j) \models t$ if $T[i,j] = t$. For a state, command, guard, or update γ, let $T, (i,j) \models \gamma$ if $\gamma \in T[i,j]$. Loop and converse are as expected: $T, x \models \mathsf{loop}(\pi)$ if $(x,x) \in [\![\pi]\!]_T$, and $[\![\pi^{-1}]\!]_T = \{(y,x) \mid (x,y) \in [\![\pi]\!]_T\}$. The semantics of $\rightarrow$ (and $\leftarrow$) is slightly different than in DataPDL, since we are not allowed to go beyond the last and first column. Thus, $[\![\rightarrow]\!]_T = \{((i,j),(i+1,j)) \mid i \in [n-1] \text{ and } j \in [k]_0\}$. However, we can simulate the "roundabout" of a ring and set $\hookrightarrow = \rightarrow + \{\neg\langle\rightarrow\rangle\}?\leftarrow^*\{\neg\langle\leftarrow\rangle\}?$ as well as $\hookleftarrow = \hookrightarrow^{-1}$. By symmetry, the first column of a table will play the role of a marked process in a ring (later, m will be translated to $\neg\langle\leftarrow\rangle$).

Finally, the semantics of path automata is given by $[\![\mathcal{A}]\!]_T = \{(x,y) \mid \text{there is } \pi_1 \cdots \pi_\ell \in L(\mathcal{A}) \text{ with } (x,y) \in [\![\pi_1 \cdots \pi_\ell]\!]_T\}$ where $L(\mathcal{A})$ contains a *sequence* $\pi_1 \cdots \pi_\ell$ of path formulas if $\mathcal{A}$ admits a path $q_0 \xrightarrow{\pi_1} q_1 \xrightarrow{\pi_2} \cdots \xrightarrow{\pi_\ell} q_\ell$ from its initial state q_0 to a final state q_ℓ.

A formula $\psi \in \mathrm{LCPDL}(\mathcal{D})$ defines the language $L(\psi) = \{T \in \Delta^{++} \mid T, (1,0) \models \psi\}$. For $b \geq 1$, we denote by $L_b(\psi)$ the set of tables $(n,k,\lambda) \in L(\psi)$ such that $k \leq b$. The bounded height satisfiability problem for LCPDL asks the following: Given a distributed algorithm $\mathcal{D}$, a formula $\psi \in \mathrm{LCPDL}(\mathcal{D})$, and $b \geq 1$ (encoded in unary), do we have $L_b(\psi) = \emptyset$? Note that the input $\mathcal{D}$ is only needed to determine the signature of the logic.

▶ **Theorem 11.** *The bounded height satisfiability problem for* LCPDL *is* PSPACE*-complete.*

Proof sketch. We can restrict to tables of height $k = b$ (rather than $k \leq b$), since checking satisfiability for every height separately does not change the space complexity. We reduce the problem to words: A table $T = (n,k,\lambda)$ is considered as the word $T[1] \cdots T[n] \in \Delta^+$. Thus, the columns are written horizontally rather than vertically. When translating an LCPDL formula over tables into an LCPDL formula over words, going to the left or right involves some modulo counting: $\leftarrow$ is translated to $\leftarrow^{k+1}$, and $\rightarrow$ is translated to $\rightarrow^{k+1}$. We then follow the construction of [13] to obtain, in polynomial time, an alternating two-way automaton (A2A) of polynomial size corresponding to the LCPDL formula (since formulas from LCPDL have bounded intersection width). Though [13] uses an exponential sized alphabet (subsets of propositions), our alphabet is the (linear-sized) set of transitions Δ, ensuring that the transition relation has only polynomial size. We allow automata as path expressions, but it is straightforward to integrate them into the construction of the A2A. Finally, satisfiability checking amounts to emptiness checking of the A2A. Emptiness checking of A2A over words can be done in PSPACE (cf. [18, 19]). ◀

From Distributed Algorithms to LCPDL. Without loss of generality, we assume that Δ contains $\mathsf{t} = \langle \mathsf{s}\colon \mathbf{skip}\,;\mathbf{skip}\,;\mathbf{skip}\,;\mathbf{skip}\,;\mathbf{goto}\ s_0\rangle$ where $\mathsf{s} \neq s_0$ does not occur in any other transition.

[1] We use automata in addition to regular expressions since using states makes it easier to describe a language. It is also important for the complexity since an automaton may be exponentially smaller than a regular expression.

$$loc^{0,1}_{r,r'} = \begin{cases} \{\bigwedge_{\bar{r}\in Reg} \neg\langle (msg^{0,1}_{\bar{r},r})^{-1}\rangle\}? & \text{if } r = r' \\ \{false\}? & \text{if } r \neq r' \end{cases} \qquad upd^{1,2}_{r,r'} = \begin{cases} \{\bigwedge_{\bar{r}\neq r} \neg(r := \bar{r})\}? & \text{if } r = r' \\ \{r' := r\}? & \text{if } r \neq r' \end{cases}$$

$$msg^{0,1}_{r,r'} = \begin{pmatrix} \{\mathbf{right}!r\}? \cdot (\hookrightarrow \cdot \{\mathbf{fwd}\}?)^* \cdot \hookrightarrow \cdot \{\mathbf{left}?r'\}? \\ + \ \{\mathbf{left}!r\}? \cdot (\hookleftarrow \cdot \{\mathbf{fwd}\}?)^* \cdot \hookleftarrow \cdot \{\mathbf{right}?r'\}? \end{pmatrix} \qquad next^{2,0}_{r,r'} = \begin{cases} \downarrow & \text{if } r = r' \\ \{false\}? & \text{if } r \neq r' \end{cases}$$

Figure 4 Path formulas to trace back transmission of pids

Let $\mathcal{R} = (n : p_1, \ldots, p_n)$ be a ring and $\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \cdots \overset{t^k}{\rightsquigarrow} C_k$ be an $\mathcal{R}$-run of $\mathcal{D}$, where $t^j = (t^j_1, \ldots, t^j_n) \in \Delta^n$ for all $j \in [k]$. From χ, we extract the *symbolic run* $T_\chi = (n, k, \lambda) \in \Delta^{++}$ given by its columns $T_\chi[i] = \mathtt{t}\, t^1_i \cdots t^k_i$. The purpose of the dummy transition $\mathtt{t}$ at the beginning of a column is to match the number of configurations in a run.

We will construct, in polynomial time, a formula $\psi_\mathcal{D} \in \mathrm{LCPDL}(\mathcal{D})$ such that $L(\psi_\mathcal{D}) = \{T_\chi \mid \chi \text{ is a run of } \mathcal{D}\}$. In particular, $\psi_\mathcal{D}$ will verify that (i) there are no cyclic dependencies that arise from $<$-guards, and (ii) registers in equality guards can be traced back to the same origin. In that case, the symbolic run is consistent and corresponds to a "real" run of $\mathcal{D}$.

The main ingredients of $\psi_\mathcal{D}$ are some path formulas that describe the transmission of pids in a symbolic run. They are depicted in Figure 4. For $\theta \in \{loc, msg, upd, next\}$ and $h \in \{0,1,2\}$, the meaning of $(x, y) \in [\![\theta^{h,h'}_{r,r'}]\!]_T$ is that the pid stored in r at *stage* h of position/transition x has been propagated to register r' at stage h' of y. Here, $h = 0$ means "after sending", $h = 1$ "after receiving", and $h = 2$ "after register update". The interpretation of "propagated" depends on θ. Formula $loc^{0,1}_{r,r'}$ says that the value of register r is not affected by reception. Similarly, $upd^{1,2}_{r,r'}$ takes care of updates. Formula $next^{2,0}_{r,r'}$ allows us to switch to the next transition of a process, preserving the value of $r(= r')$. The most interesting case is $msg^{0,1}_{r,r'}$, which describes paths across several processes. It relates the sending of r and a corresponding receive in r', which requires that all intermediate transitions are forward transitions. All path formulas are illustrated in Figure 3.

Since pids can be transmitted along several transitions and messages, the formulas $\theta^{h,h'}_{r,r'}$ will be composed by path automata. For $\mathrm{h} \in \{1,2\}$ and $\mathrm{r} \in Reg$, we define a path automaton $\mathcal{A}^{\mathrm{h}}_{\mathrm{r}}$ that, in T_χ, connects some positions $(i, 0)$ and (i', j') iff, in χ, register r stores p_i at stage h of position (i', j'). Its set of states is $\iota \cup (\{0,1,2\} \times Reg)$. For all $r \in Reg$, there is a transition from the initial state ι to $(0, r)$ with transition label $\{\neg\langle\uparrow\rangle\}?$. Thus, the automaton starts at the top row and non-deterministically chooses some register r. From state (h, r), it can read any transition label $\theta^{h,h'}_{r,r'}$ and move to (h', r'). The only final state is (h, r). Figure 3 describes (partial) runs of $\mathcal{A}^1_{r'}$ and $\mathcal{A}^1_{r''}$, which allow us to identify the origin of r' and r'' when applying the guard $r' < r''$.

Now, consistency of equality guards can indeed be verified by an LCPDL formula. 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. This can be conveniently expressed in terms of loop and converse. Note that guards are checked at stage $h = 1$ of the corresponding transition:

$$\psi_= \ = \ [(\rightarrow + \downarrow)^*] \bigwedge_{r,r'\in Reg} \Big(r = r' \ \Rightarrow \ \mathsf{loop}((\mathcal{A}^1_r)^{-1} \cdot \mathcal{A}^1_{r'}) \Big) .$$

The next path formula connects the first coordinate of a process i with the first coordinate of another process i' if some guard forces the pid of i to be smaller than that of i':

$$\pi_< \ = \Big(\sum_{r,r'\in Reg} \mathcal{A}^1_r \cdot \{r < r'\}? \cdot (\mathcal{A}^1_{r'})^{-1} \Big)^+ .$$

$\widetilde{\mathsf{m}} = \neg\langle\leftarrow\rangle \qquad \widetilde{s} = \mathbf{goto}\ s$ for all $s \in S$

$\widetilde{\neg\varphi} = \neg\widetilde{\varphi} \quad \widetilde{\varphi_1 \wedge \varphi_2} = \widetilde{\varphi_1} \wedge \widetilde{\varphi_2} \quad \widetilde{\varphi_1 \Rightarrow \varphi_2} = \widetilde{\varphi_1} \Rightarrow \widetilde{\varphi_2} \quad \widetilde{[\pi]\varphi} = [\widetilde{\pi}]\widetilde{\varphi}$

$\widetilde{\langle\pi\rangle r < \langle\pi'\rangle r'} = \mathsf{loop}(\widetilde{\pi} \cdot (\mathcal{A}^2_r)^{-1} \cdot \pi_< \cdot \mathcal{A}^2_{r'} \cdot (\widetilde{\pi}')^{-1})$

$\widetilde{\langle\pi\rangle r \leq \langle\pi'\rangle r'} = \mathsf{loop}(\widetilde{\pi} \cdot (\mathcal{A}^2_r)^{-1} \cdot (\pi_< + \epsilon) \cdot \mathcal{A}^2_{r'} \cdot (\widetilde{\pi}')^{-1})$

$\widetilde{\langle\pi\rangle r = \langle\pi'\rangle r'} = \mathsf{loop}(\widetilde{\pi} \cdot (\mathcal{A}^2_r)^{-1} \cdot \mathcal{A}^2_{r'} \cdot (\widetilde{\pi}')^{-1})$

$\widetilde{\langle\pi\rangle r \neq \langle\pi'\rangle r'} = \mathsf{loop}(\widetilde{\pi} \cdot (\mathcal{A}^2_r)^{-1} \cdot (\leftarrow^+ + \rightarrow^+) \cdot \mathcal{A}^2_{r'} \cdot (\widetilde{\pi}')^{-1})$

$\widetilde{\pi}$ is inductively obtained from π by replacing tests $\{\varphi\}?$ by $\{\widetilde{\varphi}\}?$, $\rightarrow$ by $\hookrightarrow$, and $\leftarrow$ by $\hookleftarrow$

Figure 5 From DataPDL$^\ominus$ to LCPDL

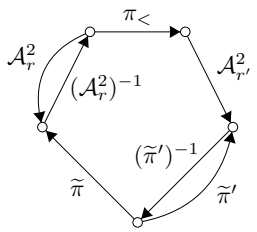

Figure 6 $\widetilde{\langle\pi\rangle r < \langle\pi'\rangle r'}$

Note that, here, we use the (strict) transitive closure. Consistency of $<$-guards now reduces to saying that there is no $\pi_<$-loop: $\psi_< \ = \ \neg\langle\rightarrow^*\rangle\mathsf{loop}(\pi_<)$.

Finally, we can easily write an LCPDL formula ψ_{col} that checks whether every column $T[i] \in \Delta^+$ (ignoring $\mathtt{t}$) is a valid transition sequence of $\mathcal{D}$. Finally, let $\psi_{\mathcal{D}} = \psi_= \wedge \psi_< \wedge \psi_{\mathsf{col}}$.

▶ **Lemma 12.** *We have* $L(\psi_{\mathcal{D}}) = \{T_\chi \mid \chi \text{ is a run of } \mathcal{D}\}$.

From DataPDL$^\ominus$ to LCPDL. Next, we inductively translate every local DataPDL$^\ominus(\mathcal{D})$ formula φ into an LCPDL$(\mathcal{D})$ formula $\widetilde{\varphi}$. The translation is given in Figure 5. As mentioned before, the first column in a table plays the role of a marked process so that $\widetilde{\mathsf{m}} = \neg\langle\leftarrow\rangle$. The standard formulas are translated as expected. Now, consider $\widetilde{\langle\pi\rangle r < \langle\pi'\rangle r'}$ (the remaining cases are similar). To "prove" $\langle\pi\rangle r < \langle\pi'\rangle r'$ 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' satisfy $r < r'$. To guarantee the latter, the pids stored in r and r' have to go back to coordinates that are connected by a $\pi_<$-path. Again, using converse, this can be expressed as a loop (cf. Figure 6). Note that, hereby, $\mathcal{A}^2_r$ and $\mathcal{A}^2_{r'}$ refer to stage $h = 2$, which reflects the fact that DataPDL speaks about *configurations* (determined after updates).

▶ **Lemma 13.** *Let* $T \in \{T_\chi \mid \chi \text{ is a run of } \mathcal{D}\}$ *and* φ *be a local* DataPDL$^\ominus(\mathcal{D})$ *formula. We have* $T, (1,0) \models \widetilde{\varphi} \iff \big(\chi, 1, (1,0) \models \varphi \text{ for all runs } \chi \text{ of } \mathcal{D} \text{ such that } T_\chi = T\big)$.

Using Lemmas 12 and 13, we can now prove Lemma 14 below. Together with Theorem 11, the upper bound of Theorem 10 follows.

▶ **Lemma 14.** *Let* $\mathcal{D}$ *be a distributed algorithm,* $\Phi = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\varphi \in$ DataPDL$^\ominus(\mathcal{D})$*, and* $b \geq 1$*. We have (a)* $\mathcal{D} \models \Phi \iff L(\psi_{\mathcal{D}} \wedge \neg\widetilde{\varphi}) = \emptyset$*, and (b)* $\mathcal{D} \models_b \Phi \iff L_b(\psi_{\mathcal{D}} \wedge \neg\widetilde{\varphi}) = \emptyset$.

5 Conclusion

In this paper, we provided a conceptually new approach to the verification of distributed algorithms that is robust against small changes of the model.

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. Second, messages are pids, but they could contain message contents from a finite alphabet as well. 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, for instance trees.

We leave open whether round-bounded model checking can deal with full DataPDL, or with properties of the form $\forall_{rings}\exists_{run}\forall_{\mathsf{m}}\varphi$, which are branching-time in spirit.

References

1 C. Aiswarya, B. Bollig and P. Gastin. An Automata-Theoretic Approach to the Verification of Distributed Algorithms. *CoRR*, abs/1504.06534. 2015.

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

3 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.

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

5 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.

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

7 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.

8 E. A. Emerson and V. Kahlon. Parameterized Model Checking of Ring-Based Message Passing Systems. In *CSL'05*, volume 3210 of *LNCS*, pages 325-339. Springer, 2004.

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

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

11 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.

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

13 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.

14 I. Konnov, H. Veith, and J. Widder. Who is afraid of model checking distributed algorithms? In *CAV'12 Workshop (EC)^2*. 2012.

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

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

17 N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

18 R. Mennicke. Propositional Dynamic Logic with Converse and Repeat for Message-Passing Systems. *LMCS* 9(2:12) 2013.

19 O. Serre. Parity Games Played on Transition Graphs of One-Counter Processes. In *FOSSACS'06*, LNCS, pages 337–351. Springer, 2006.

20 M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

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