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. 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.

1 Introduction

Distributed algorithms are a classic discipline of computer science and continue to be an active field of research [13, 19]. 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 [14] or the Dolev-Klawe-Rodeh algorithm [9], 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 verification methods such as model checking [8]. Besides a formal model of an algorithm, this requires a generic specification language that is feasible from an algorithmic point of

* Supported by LIA InForMel.

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, 5, 12].

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 [11]; second, processes manipulate data from an infinite domain [5, 12]. In each case, even simple verification questions are undecidable, and so is 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 [20]. 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 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 [15]. 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 verification of fault-tolerant algorithms, which have to cope with faults such as lost or corrupted messages (e.g., [7, 17]). After all, there have been only very few generic approaches to model checking distributed algorithms. In [16], 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, 10].

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 [5], or for the analysis of programs with data structures such as lists and arrays [1, 2]. Notably, streaming transducers [1] 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. Recall that the latter borrows concepts from [4, 5, 12], whose logic is designed to reason about XML documents. A fragment of MSO logic over *ordered* data trees was studied in [21]. The paper [6] 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 introduces the specification language to express correctness criteria. In Section 4, we show how to solve the round-bounded model-checking problem in polynomial space. We conclude in Section 5. Some proof details are omitted but can be found in the appendix.

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 process $i \in [n]$. 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 $i > 2$, 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 the change of state, a process may optionally perform the following phases within a transition: (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:

$$\mathit{send} ::= \mathbf{skip} \mid \mathbf{fwd} \mid \mathbf{left}!r \mid \mathbf{right}!r \mid \mathbf{left}!r \,; \mathbf{right}!r'$$

$$\mathit{rec} ::= \mathbf{skip} \mid \mathbf{left}?r \mid \mathbf{right}?r \mid \mathbf{left}?r \,; \mathbf{right}?r'$$

$$\mathit{guard} ::= \mathbf{skip} \mid r < r' \mid r = r' \mid \mathit{guard} \,; \mathit{guard}$$

$$\mathit{update} ::= \mathbf{skip} \mid r := r' \mid \mathit{update} \,; \mathit{update}$$

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

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

states: $\mathit{active}_0, \mathit{active}_1$	$t_1 = \langle \mathit{active}_0\colon \mathbf{right}!r\,;\mathbf{left}?r'\,;\mathbf{goto}\ \mathit{active}_1\rangle$
passive, *found*	$t_2 = \langle \mathit{active}_1\colon \mathbf{right}!r'\,;\mathbf{left}?r''\,;r'' < r'\,;r < r'\,;r := r'\,;\mathbf{goto}\ \mathit{active}_0\rangle$
initial state: active_0	$t_3 = \langle \mathit{active}_1\colon \underline{\qquad\qquad\qquad}\,;r' < r\,;\mathbf{goto}\ \mathit{passive}\rangle$
registers: id, r, r', r''	$t_4 = \langle \mathit{active}_1\colon \underline{\qquad\qquad\qquad}\,;r' < r''\,;\mathbf{goto}\ \mathit{passive}\rangle$
	$t_5 = \langle \mathit{active}_1\colon \underline{\qquad\qquad\qquad}\,;r = r'\,;\mathbf{goto}\ \mathit{found}\rangle$
	$t_6 = \langle \mathit{passive}\colon \mathbf{fwd}\,;\mathbf{left}?r\,;\mathbf{goto}\ \mathit{passive}\rangle$

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

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; this does not affect the semantics. $\lhd$

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.

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 $\mathit{id} \in \mathit{Reg}$ that is never updated, i.e., no transition contains a command of the form $\mathbf{left}?\mathit{id}$, $\mathbf{right}?\mathit{id}$, or $\mathit{id} := r$. 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 [14]. 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., $r_1 < \mathit{id}$ and $r_2 < \mathit{id}$ 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). 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. $\lhd$

Example 3 (Dolev-Klawe-Rodeh Leader-Election Algorithm)**.** The Dolev-Klawe-Rodeh leader-election algorithm [9] 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 $\mathcal{D}_{\mathsf{Franklin}}$. $\lhd$

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

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

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":

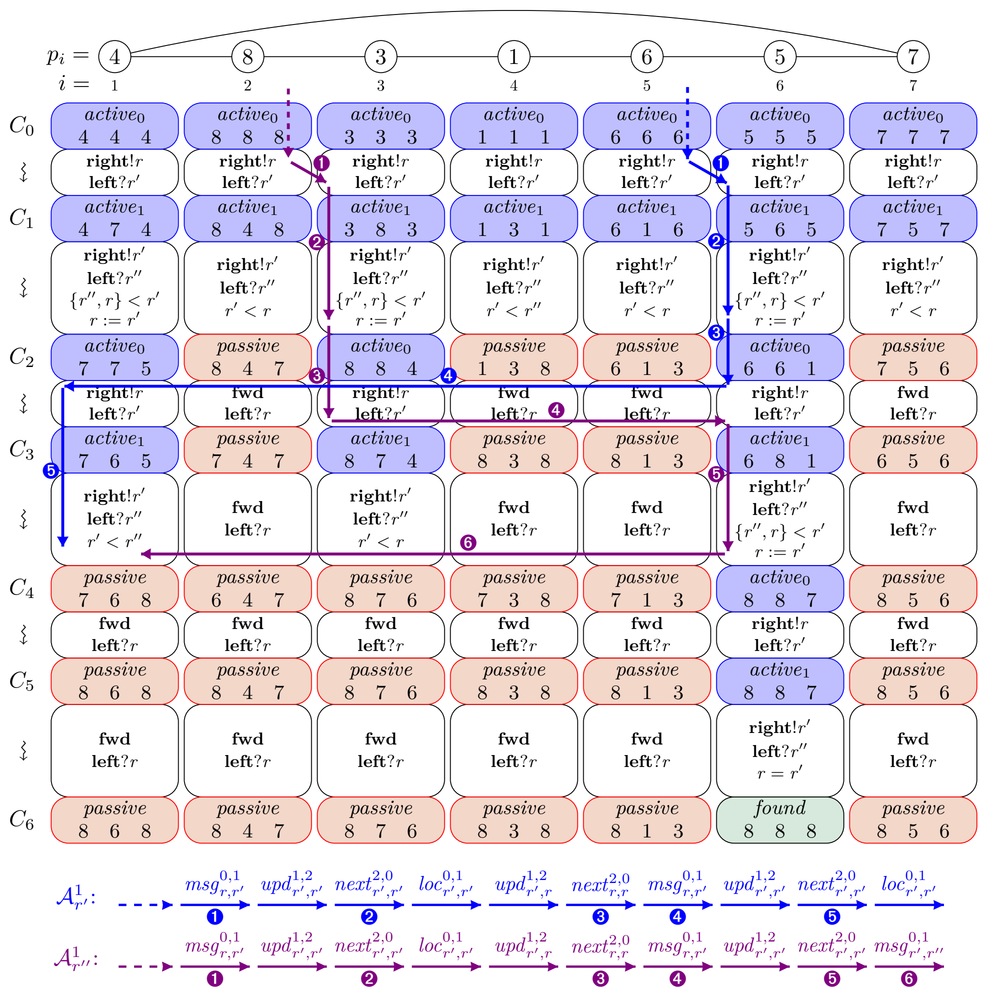
$$\mathbf{left}!r \in t_i \;\wedge\; \mathbf{right}?r' \in t_j \;\wedge\; \mathbf{fwd} \in t_k \text{ for all } k \in \mathit{Between}(j,i).$$

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:



$p_i =$ 4 8 3 1 6 5 7
$i =$ 1 2 3 4 5 6 7
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!r left?r' (all processes)
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!r' left?r'' $\{r'', r\} < r'$ $r := r'$ | right!r' left?r'' $r' < r$ | right!r' left?r'' $\{r'', r\} < r'$ $r := r'$ | right!r' left?r'' $r' < r''$ | right!r' left?r'' $r' < r$ | right!r' left?r'' $\{r'', r\} < r'$ $r := r'$ | right!r' left?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!r left?r' | fwd left?r | right!r left?r' | fwd left?r | fwd left?r | right!r left?r' | fwd left?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!r' left?r'' $r' < r''$ | fwd left?r | right!r' left?r'' $r' < r$ | fwd left?r | fwd left?r | right!r' left?r'' $\{r'', r\} < r'$ $r := r'$ | fwd left?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 left?r | fwd left?r | fwd left?r | fwd left?r | fwd left?r | right!r left?r' | fwd left?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 left?r | fwd left?r | fwd left?r | fwd left?r | fwd left?r | right!r' left?r'' $r = r'$ | fwd left?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
$\mathcal{A}^1_{r'}$: $msg^{0,1}_{r,r'}$ ❶ $upd^{1,2}_{r',r'}$ $next^{2,0}_{r',r'}$ ❷ $loc^{0,1}_{r',r'}$ $upd^{1,2}_{r',r}$ $next^{2,0}_{r,r}$ ❸ $msg^{0,1}_{r,r'}$ ❹ $upd^{1,2}_{r',r'}$ $next^{2,0}_{r',r'}$ ❺ $loc^{0,1}_{r',r'}$
$\mathcal{A}^1_{r''}$: $msg^{0,1}_{r,r'}$ ❶ $upd^{1,2}_{r',r'}$ $next^{2,0}_{r',r'}$ ❷ $loc^{0,1}_{r',r'}$ $upd^{1,2}_{r',r}$ $next^{2,0}_{r,r}$ ❸ $msg^{0,1}_{r,r'}$ ❹ $upd^{1,2}_{r',r'}$ $next^{2,0}_{r',r'}$ ❺ $msg^{0,1}_{r',r''}$ ❻

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

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

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

An ($\mathcal{R}$-)*run* of $\mathcal{D}$ is a sequence $\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C_k$ 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 4. 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 5. 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. $\lhd$

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 $\mathcal{D} = (S, s_0, \mathit{Reg}, \Delta)$, 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 $\forall_{\mathit{rings}} \forall_{\mathit{runs}} \forall_{\mathsf{m}} \varphi$, 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, 5, 12]. 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 6. The logic DataPDL($\mathcal{D}$) is given by the following grammar:

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

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

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 $\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C_k$ 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 $(i,j) \in \mathit{Pos}(\chi)$ where $\mathit{Pos}(\chi) = [n] \times [k]_0$. 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 $[\![\pi]\!]_{\chi,m} \subseteq \mathit{Pos}(\chi) \times \mathit{Pos}(\chi)$, 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: $[\![\{\varphi\}?]\!]_{\chi,m} = \{(x,x) \mid x \in \mathit{Pos}(\chi) \text{ such that } \chi, m, x \models \varphi\}$, and $[\![\epsilon]\!]_{\chi,m} = \{(x,x) \mid x \in \mathit{Pos}(\chi)\}$. Using $\rightarrow$, we move to the right neighbor of a process: $[\![\rightarrow]\!]_{\chi,m} = \{((i,j),(i+1,j)) \mid i \in [n-1] \text{ and } j \in [k]_0\} \cup \{((n,j),(1,j)) \mid j \in [k]_0\}$. We define $[\![\leftarrow]\!]_{\chi,m}$ accordingly. Moreover, $[\![\downarrow]\!]_{\chi,m} = \{((i,j),(i,j+1)) \mid i \in [n] \text{ and } j \in [k-1]_0\}$, 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 $\forall_{\mathit{rings}} \forall_{\mathit{runs}} \forall_{\mathsf{m}} \varphi$, written $\mathcal{D} \models \forall_{\mathit{rings}} \forall_{\mathit{runs}} \forall_{\mathsf{m}} \varphi$, 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. all processes m.

Next, we define a restricted logic, $\text{DataPDL}^{\ominus}(\mathcal{D})$, 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 $x \in \mathit{Pos}(\chi)$, there is at most one $x' \in \mathit{Pos}(\chi)$ 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 7. A $\text{DataPDL}(\mathcal{D})$ formula is contained in $\text{DataPDL}^{\ominus}(\mathcal{D})$ 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. $\lhd$

Example 8. Let us *formalize*, in $\text{DataPDL}^{\ominus}(\mathcal{D})$, 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:

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

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 $\text{DataPDL}^{\ominus}$ formula. Consider the $\text{DataPDL}^{\ominus}$ formula

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

It says that, at the end (i.e., in the last configuration) of each accepting run, expressed by $[\downarrow^*]\big((\varphi_{\mathsf{last}} \wedge \varphi_{\mathsf{acc}}) \Rightarrow \ldots\big)$, 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. $\lhd$

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 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 [20]. 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. Examples 2 and 3). 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 proof, a reduction from the intersection-emptiness problem for a list of finite automata, can be found in the appendix. 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 [18], 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) [15], which in turn can be reduced to an emptiness check of alternating two-way automata (A2As) over words [23]. 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 A2A that recognizes 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 emptiness 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 $\Phi \in \mathrm{DataPDL}^{\ominus}(\mathcal{D})$. We will reduce model checking to the satisfiability problem for LCPDL [15]. 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 (or picture) 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 any of its disjuncts 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 [15] by a reduction to the emptiness problem for A2As over words [23]. Our approach is, thus, automata theoretic in spirit, though the power of alternation is used differently than in [22], which translates LTL formulas into automata.

Next, we present the logic LCPDL over symbolic runs. Then, in separate subsections, 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., $T[i] = T[i,0] \ldots T[i,k] \in \Delta^+$. Let Δ^{++} denote the set of all tables.

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

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

where $t \in \Delta$, $s \in S$, $r, r' \in \mathit{Reg}$, $d \in \{\epsilon, \rightarrow, \downarrow\}$, and $\mathcal{A}$ is a *path automaton*: a non-deterministic finite automaton whose transitions are labeled with path formulas π. Again, ψ is called a

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

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

Figure 4 Path formulas to trace back transmission of pids

local formula. We use common abbreviations to include disjunction, implication, *true*, and *false*, and we let $\pi^+ = \pi \cdot \pi^*$, $[\pi]\psi = \neg\langle\pi\rangle\neg\psi$, $\langle\pi\rangle = \langle\pi\rangle true$, $\leftarrow\, = \,\rightarrow^{-1}$, and $\uparrow\, = \,\downarrow^{-1}$.

The semantics of LCPDL is very similar to that of DataPDL. A local formula ψ is interpreted over a table $T = (n, k, \lambda)$ and a position $x \in Pos(T)$. When it is satisfied, we write $T, x \models \psi$. Moreover, a path formula π determines a binary relation $[\![\pi]\!]_T \subseteq Pos(T) \times Pos(T)$, 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 $\hookleftarrow\, = \,\hookrightarrow^{-1}$. Actually, 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 $[\![\mathcal{A}]\!]_T = \{(x, y) \mid \text{there is } \pi_1 \ldots \pi_\ell \in L(\mathcal{A}) \text{ with } (x, y) \in [\![\pi_1 \cdot \ldots \cdot \pi_\ell]\!]_T\}$ where $L(\mathcal{A})$ contains a *sequence* $\pi_1 \ldots \pi_\ell$ of path formulas if $\mathcal{A}$ admits a path $q_0 \xrightarrow{\pi_1} q_1 \xrightarrow{\pi_2} \ldots \xrightarrow{\pi_\ell} q_\ell$ 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 $k \leq b$.

Theorem 11 (essentially [15])**.** *The following problem is PSPACE-complete: Given a distributed algorithm $\mathcal{D}$, a formula $\psi \in \mathrm{LCPDL}(\mathcal{D})$, and $b \geq 1$ (encoded in unary), do we have $L_b(\psi) = \emptyset$? (The input $\mathcal{D}$ is only needed to determine the signature of the logic.)*

From Distributed Algorithms to LCPDL. Wlog., 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.

Let $\mathcal{R} = (n : p_1, \ldots, p_n)$ be a ring and $\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C_k$ 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 $T_\chi[i] = \mathsf{t}\, t^1_i \ldots t^k_i$. The purpose of the dummy transition 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 $\mathrm{r} \in \mathit{Reg}$, 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 $\iota \cup (\{0, 1, 2\} \times \mathit{Reg})$. For all $r \in \mathit{Reg}$, 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:

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

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':

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

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

Finally, we can easily write an LCPDL formula ψ_{col} that checks whether every column $T[i] \in \Delta^+$ (ignoring 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* $L(\psi_{\mathcal{D}}) = \{T_\chi \mid \chi$ *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* $T \in \{T_\chi \mid \chi$ *is a run of* $\mathcal{D}\}$ *and* φ *be a local* DataPDL$^\ominus(\mathcal{D})$ *formula. We have* $T, (1, 0) \models \widetilde{\varphi} \iff \big(\chi, 1, (1, 0) \models \varphi$ *for all runs* χ *of* $\mathcal{D}$ *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,* $\Phi = \forall_{\mathit{rings}}\forall_{\mathit{runs}}\forall_{\mathsf{m}}\varphi \in$ 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)* $\mathcal{D} \models_b \Phi \iff L_b(\psi_{\mathcal{D}} \wedge \neg\widetilde{\varphi}) = \emptyset$.

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

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

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

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

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

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

$\widetilde{\pi}$ 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

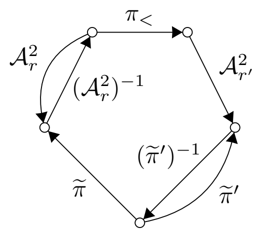


Figure 6 $\widetilde{\langle\pi\rangle r < \langle\pi'\rangle r'}$

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.

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

2 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.

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

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

6 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.

7 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.

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

9 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.

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

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

12 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.

13 W. Fokkink. *Distributed Algorithms: An Intuitive Approach.* MIT Press, 2013.

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

15 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.

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

17 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.

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

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

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

21 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.

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

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

A Proof of Theorem 9

The following remark will be exploited in the proof of Theorem 9 and for the lower-bound proof of Theorem 10.

Remark 15. Note that the only way to communicate information from one process to another is by exchanging and comparing pids. However, we can simulate the exchange of messages from a *finite* alphabet $B = \{b_1, \ldots, b_k\}$ that can be compared for equality.

Assume a ring $\mathcal{R} = (n : p_1, \ldots, p_n)$. A possible protocol for simulation can employ a leader election algorithm first. Afterwards, the leader identifies k distinct pids (say the k closest pids on its left), and transmits them to all other processes who keep them in dedicated registers $\hat{r}_1, \ldots, \hat{r}_k$. After this initialization phase, the actual simulation can take place with the convention that message b_j is identified by the pid in $\hat{r}_j$ (of any process). In order for the simulation to work, we have to require that $n \geq k$.

The drawback of the above protocol is that the initialization phase requires $\log(n)$ rounds. Below we describe another protocol where the initialization can be achieved in k rounds.

Assume a ring $\mathcal{R} = (n : p_1, \ldots, p_n)$ and that $n \geq k$. Each process has $k+1$ dedicated registers $\hat{r}_0, \ldots, \hat{r}_k$. After the initialization (described below), for each process i, register $\hat{r}_j$ holds p_{i-j} (modulo n). Thus $\hat{r}_j$ of process i holds the same value as $\hat{r}_{j+1}$ of process $i+1$.

Conventions. To send message b_j to left, a process simply sends the contents of $\hat{r}_j$. On the other hand, to send message b_j to right, it sends the contents of $\hat{r}_{j-1}$. When a process receives a message from the left, it compares it with registers $\hat{r}_1, \ldots, \hat{r}_k$, and if it matches $\hat{r}_j$ then the message is interpreted as b_j. On receiving from right, on contrary, it is compared to $\hat{r}_0, \ldots \hat{r}_{k-1}$, and if it matches $\hat{r}_j$ then the message is interpreted as b_{j+1}.

Initialization. It uses $k+1$ control states $s_0, \ldots, s_k$. At s_0, all registers have self pid. This fills in the correct value for $\hat{r}_0$. In round j, a process moves from s_{j-1} to s_j, sending $\hat{r}_{j-1}$ to the right and receiving in $\hat{r}_j$ from the left.

Notice that this simulation cannot be used to forward a message to another process using **fwd**-commands in between. However, the lower bound proofs presented below do not rely on **fwd**-commands. $\lhd$

Proof of Theorem 9. We give a reduction from the halting problem of Turing machines. It is equivalent to checking whether a given Turing machine TM can never reach a specific target state (call it HALT) on any (some) input. Let S_{TM} be the set of control states of a Turing machine. Let B_{TM} be the tape alphabet of the Turing Machine. Wlog., we assume that the TM starts on the empty tape. From the empty tape, it may simulate an arbitrary input using non-determinism. We also assume that, on reaching the state HALT, it writes HALT in the current cell. Thus HALT $\in S_{\mathsf{TM}}$ and HALT $\in B_{\mathsf{TM}}$. We describe the distributed algorithm $\mathcal{D}_{\mathsf{TM}}$.

Intuitively, the number of processes in the ring gives an upper bound to the space needed by the Turing machine. Every process will correspond to a cell in the Turing machine's work 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 processes indicate the corresponding cell contents. It also indicates whether the head is currently present at the respective cell. Thus the local states are pairs of the form $(\mathsf{sym}, \mathsf{head})$ where $\mathsf{sym} \in B_{\mathsf{TM}}$ indicates the content of a tape cell, and head is a boolean value denoting the presence of the head of the Turing machine at the current cell. Initially, only the leader process has the head bit set *true*. In the simulation, only the process with $\mathsf{head} = \textit{true}$ can send messages, and once it emits a message, the head bit is turned *false*. The process that receives the message turns the head bit *true*. The message alphabet (cf. Remark 15) is S_{TM} which denotes the target control state upon simulating one transition of the Turing machine. The control state of the TM is stored in a designated register r_{state}.

We describe the construction in detail now. There are two preliminary phases to facilitate the actual simulation. In phase 1, the processes agree upon the message alphabet S_{TM} as described in Remark 15. This phase requires $|S_{\mathsf{TM}}| + 1$ registers and local states. Recall that the ring must have size bigger than $|S_{\mathsf{TM}}|$ for simulating the encoding described in Remark 15. Otherwise, the distributed algorithm will be blocked in this phase. However, our reduction would still work because of two reasons. First, our specification will be true for rings smaller than this threshold. This is, in a sense, reducing the model-checking problem with $\forall_{rings}\forall_{runs}\forall_{\mathsf{m}}$ prefix to another model-checking problem where the prefix is rephrased to "All rings of size bigger than ℓ" (here, $\ell = |S_{\mathsf{TM}}|$). Second, the run which uses only a small amount of tape can be simulated on a big tape. (It maintains the unnecessary cells on the right with the empty tape symbol always. In our simulation these processes will be in the state $(\$, \mathit{false})$.) Notice that, the number of processes in the ring is only an upper bound of (rather than exact) space needed by the Turing machine.

Phase 2 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 phase 2 is completed, the configuration of the ring proceeds to represent the initial configuration of TM. For this, all processes other than the leader will move to the state $(\$, \mathit{false})$, i.e., representing the empty tape cell and indicating the absence of the head. The leader process will move to the state $(\$, \mathit{true})$. On taking this transition, the register r_{state} of all the processes are set to hold 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, true) to (b, false) which also (i) ensures (by a guard) that r_{state} contains the encoding of s, (ii) ensures (by a guard) that it is not the leftmost cell ($r_{\text{leader}} \neq \mathit{id}$), and (iii) sends the encoding of s' to the left. For this transition to take place, there are complementary transitions at the receive end which go from $(\text{-}, \mathit{false})$ to $(\text{-}, \mathit{true})$ upon receiving a value from a neighbor (left or right) to its register r_{state}. 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 $\mathsf{head} = \mathit{true}$. Notice also that the leader process does not send to left. Also, there are no forwarding states.

There is actually one subtlety here that arises from the fact that receptions are non-blocking. We have to make sure that a process is aware whether a "real" message was received or not. To do so, we introduce a register $r_{\perp}$, containing a special message $\perp$. Note that the first preliminary phase must indeed be executed for an extended message alphabet that also includes the special symbol $\perp$. For incoming messages, a process will use a special register r_{in}, which initially contains $\perp$. After executing a receive action, a process will check whether $r_{\mathsf{in}} \neq r_{\perp}$, which makes sure that a message has indeed arrived. The subsequent update will then execute $r_{\mathsf{in}} := r_{\perp}$ to reset r_{in}.

Finally, the specification φ_{TM} checks that there is no process in the state $(\text{HALT}, \mathit{true})$. Thus, if the model-checking problem answers negatively, then there is a ring and a run which encodes a valid Turing machine computation on a tape of size bigger than S_{TM} (which also simulates any smaller size tape) and still reaches the HALT state:

$$\varphi_{\mathsf{TM}} = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}[\downarrow^*]\neg(\textsc{halt}, \mathit{true})$$

This concludes the proof of Theorem 9. □

B Proof of Lower bound of Theorem 10

Proof. 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 $\mathcal{A}_i = (Q_i, \Delta_i, \mathsf{init}_i, \mathsf{F}_i)$, whether $\bigcap_i L(\mathcal{A}_i) = \emptyset$? This problem is known to be PSPACE-complete.

We will need only unidirectional rings for our reduction. We construct the distributed algorithm $\mathcal{D}$ as follows.

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. A preliminary phase sets the message alphabet as per Remark 15. At round i after the preliminary phase, all the processes try to simulate a transition of automaton $\mathcal{A}_i$ on the respective position. We give the details below.

In a preliminary phase, the distributed algorithm establishes the finite message alphabet $B = \bigcup_i Q_i$. This requires $|B| + 1$ states, registers, and rounds. In case the ring is smaller than $|B|$, the distributed algorithm will be blocked in this phase. However, our reduction would still work because of two reasons. First, our specification will be true for rings smaller than this threshold. Second, if a word is accepted by all the automata $\mathcal{A}_i$, then acceptance of that word can be simulated on arbitrarily large rings. This will become clear below when we give the actual construction.

The register used for sending the value of a state s to the right is denoted $\mathsf{EncOf}(s)$. On receiving a value from the left, let $\mathsf{DecOf}(s)$ be the register against which it is compared to ensure that the received value corresponds to state s.

After the preliminary phase, a process non-deterministically moves to a local state from the set $(\Sigma \cup \{\$\}) \times [1]$. 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 i from $[k]$, indicating that it is currently simulating $\mathcal{A}_i$. For each $a \in \Sigma$ and $i \leq k$, we have a transition of the form

$$\langle (a, i)\colon \mathbf{right}!\mathsf{EncOf}(s')\,;\mathbf{left}?r\,; r = \mathsf{DecOf}(s)\,; \mathbf{goto}\ (a, i+1)\rangle$$

if $(s, a, s') \in \Delta_i$. Further we have

$$\langle (\$, i)\colon \mathbf{right}!\mathsf{EncOf}(\mathsf{init})\,;\mathbf{left}?r\,; r = \mathsf{DecOf}(f)\,; \mathbf{goto}\ (\$, i+1)\rangle$$

if $f \in \mathsf{F}_i$. Notice that the symbol associated to a process does not change in any of these transitions.

Thus, the number of rounds needed by the distributed algorithm is $b = |B| + m + 1$, which is polynomial in the size of the input to intersection emptiness problem of finite state automata. The size of the distributed algorithm $\mathcal{D}$ is also polynomial.

Finally, the DataPDL$^{\ominus}(\mathcal{D})$ formula states that a state of the form $(\$, k+1)$ cannot be reached:

$$\varphi_m = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}[\downarrow^*]\neg(\$, k+1)$$

Notice that, if the bounded model checking answers no, then there are a ring, a run, and a marked process m such that m eventually reaches the state $(\$, k+1)$. This means that, on all states $(\$, i)$, 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 $(\$, i)$. Note that m' can be 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 non-empty, 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 stated in Theorem 10. ◻

C Proof of Theorem 11

We can restrict to pictures of height $k = b$ (rather than $k \leq b$), since checking satisfiability for every height separately does not change the complexity. We reduce the problem to words, for which LCPDL satisfiability is known to be PSPACE-complete [15] (since formulas from LCPDL have bounded intersection width). A picture $T = (n, k, \lambda)$ is considered as the word $T[1] \cdot \ldots \cdot T[n] \in \Delta^+$. 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 $\rightarrow^{k+1}$. An additional difficulty stems from the fact that we allow automata as path expressions, but it is straightforward to integrate them into the construction of an alternating two-way automaton from [15].

D Proof of Lemma 12

Let us first introduce some notation. Let $\mathcal{T}_\mathcal{D} = \{T_\chi \mid \chi \text{ is a run of } \mathcal{D}\}$. For a table $T \in \Delta^{++}$, let $\mathit{Runs}(T) = \{\chi \mid \chi \text{ is run of } \mathcal{D} \text{ such that } T_\chi = T\}$.

A *pseudo* ($\mathcal{R}$-)*run* of $\mathcal{D}$ is like an ($\mathcal{R}$-)run $\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C_k$, but conditions 1.-3. are not checked. That is, target and source states are not necessarily matching, and =- and <-guards are ignored. Thus, every run is a pseudo run, but not vice versa. We define T_χ and $\mathit{Pos}(\chi)$ 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 $C_j = (s_1^j, \ldots, s_n^j, \rho_1^j, \ldots, \rho_n^j)$) and $(i,j) \in \mathit{Pos}(\chi)$, we set $\chi_i^j = \rho_i^j$ (abusing notation). Moreover, for $j \geq 1$, $\hat{\chi}_i^j = \hat{\rho}_i^j$ defines the corresponding j-th intermediate register assignment, which was defined in Section 2 to obtain the mapping ρ_i^j. Finally, we set $\hat{\chi}_i^0 = \chi_i^0$.

To prove Lemma 12, we will need two further lemmas:

Lemma 16. *For all pseudo runs χ of $\mathcal{D}$, coordinates $(i,j), (i',j') \in \mathit{Pos}(\chi)$, and registers $r \in \mathit{Reg}$, the following hold:*

(a) $((i,j),(i',j')) \in [\![\mathcal{A}_r^1]\!]_{T_\chi} \iff \big(\chi_i^0(id) = \hat{\chi}_{i'}^{j'}(r) \;\wedge\; j = 0\big)$

(b) $((i,j),(i',j')) \in [\![\mathcal{A}_r^2]\!]_{T_\chi} \iff \big(\chi_i^0(id) = \chi_{i'}^{j'}(r) \;\wedge\; j = 0\big)$

Proof. Let the pseudo $\mathcal{R}$-run in question be given by $\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C_k$ where $t^j = (t_1^j, \ldots, t_n^j) \in \Delta^n$.

To be able to perform an induction, we show a more general statement that captures both (a) and (b). To this aim, we define the automaton $\mathcal{A}_r^0$ in the expected manner, i.e., where the only final state is $(0, r)$. We will show, for all $h \in \{0, 1, 2\}$,

$$((i,j),(i',j')) \in [\![\mathcal{A}_r^h]\!]_{T_\chi} \iff \big(\chi_i^0(id) = \chi_{i'}^{j'}[h](r) \;\wedge\; j = 0\big)\,. \tag{1}$$

Here, $\chi_{i'}^{j'}[2]$ refers to $\chi_{i'}^{j'}$ and $\chi_{i'}^{j'}[1]$ refers to $\hat{\chi}_{i'}^{j'}$ (recall that $\hat{\chi}_{i'}^0 = \chi_{i'}^0$). For $j' \geq 1$, we let $\chi_{i'}^{j'}[0](r)$ refer to the value of r at position (i', j') before reception. Finally, we set $\chi_{i'}^0[0] = \chi_{i'}^0[1] (= \chi_{i'}^0)$.

Before we come to the actual proof of (1), we define the relation

$$\longrightarrow_\chi \subseteq \mathit{Conf} \times \{\mathit{loc}, \mathit{upd}, \mathit{next}, \mathit{msg}\} \times \mathit{Conf}$$

where $\mathit{Conf} = \mathit{Pos}(\chi) \times \{0,1,2\} \times \mathit{Reg}$. The idea is that $\longrightarrow_\chi$ captures the flow of pids in χ. We let $\longrightarrow_\chi$ be the least relation satisfying the following:

- $(i,j,r,0) \xrightarrow{loc}_\chi (i,j,r,1)$ if there are no r', i' such that $r'@i' \rightarrowtail r@i$ (in step $C_{j-1} \overset{t^j}{\rightsquigarrow} C_j$)
- $(i,j,r,1) \xrightarrow{upd}_\chi (i,j,r',2)$ if $r \neq r'$ and $(r' := r) \in t_i^j$, or $r = r'$ and $(r := r'') \notin t_i^j$ for all $r'' \neq r$
- $(i,j,r,2) \xrightarrow{next}_\chi (i,j+1,r,0)$
- $(i,j,r,0) \xrightarrow{msg}_\chi (i',j,r',1)$ if $r@i \rightarrowtail r'@i'$ or $r'@i' \leftarrowtail r@i$ (in step $C_{j-1} \overset{t^j}{\rightsquigarrow} C_j$)

Note that $(i,j,r,h) \xrightarrow{\theta}_\chi (i',j',r',h')$ immediately implies $\chi_i^j[h](r) = \chi_{i'}^{j'}[h'](r')$. We will show that, moreover, we have

$$(i,j,r,h) \xrightarrow{\theta}_\chi (i',j',r',h') \iff ((i,j),(i',j')) \in [\![\theta_{r,r'}^{h,h'}]\!]_{T_\chi} \tag{2}$$

To prove this, we distinguish four cases:

- Suppose $\theta = \mathit{loc}$. Then, we can assume $h = 0$ and $h' = 1$. We have

$$\begin{aligned} & (i,j,r,0) \xrightarrow{\theta}_\chi (i',j',r',1) \\ \iff\; & r = r' \;\wedge\; (i,j) = (i',j') \;\wedge\; \neg\exists \bar{r}, \bar{i} \text{ such that } \bar{r}@\bar{i} \rightarrowtail r@i \text{ (in step } C_{j-1} \overset{t^j}{\rightsquigarrow} C_j) \\ \iff\; & r = r' \;\wedge\; ((i,j),(i',j')) \in [\![\{\textstyle\bigwedge_{\bar{r}\in Reg} \neg\langle (msg_{\bar{r},r}^{0,1})^{-1}\rangle\}?]\!]_{T_\chi} \\ \iff\; & ((i,j),(i',j')) \in [\![loc_{r,r'}^{0,1}]\!]_{T_\chi} \end{aligned}$$

- Suppose $\theta = \mathit{upd}$. We can assume $h = 1$ and $h' = 2$. We distinguish two subcases.
  1. Suppose $r \neq r'$. Then, we have

$$\begin{aligned} & (i,j,r,1) \xrightarrow{\theta}_\chi (i',j',r',2) \\ \iff\; & (i,j) = (i',j') \;\wedge\; (r' := r) \in t_{i'}^{j'} \\ \iff\; & ((i,j),(i',j')) \in [\![\{r' := r\}?]\!]_{T_\chi} \\ \iff\; & ((i,j),(i',j')) \in [\![update_{r,r'}^{1,2}]\!]_{T_\chi} \end{aligned}$$

  2. Suppose $r = r'$. Then,

$$\begin{aligned} & (i,j,r,1) \xrightarrow{\theta}_\chi (i',j',r',2) \\ \iff\; & (i,j) = (i',j') \;\wedge\; (r := \bar{r}) \notin t_{i'}^{j'} \text{ for all } \bar{r} \neq r \\ \iff\; & ((i,j),(i',j')) \in [\![\{\textstyle\bigwedge_{\bar{r}\neq r} \neg(r := \bar{r})\}?]\!]_{T_\chi} \\ \iff\; & ((i,j),(i',j')) \in [\![update_{r,r'}^{1,2}]\!]_{T_\chi} \end{aligned}$$

- Suppose $\theta = \mathit{next}$. We can assume $h = 2$ and $h' = 0$. We have

$$\begin{aligned} & (i,j,r,2) \xrightarrow{\theta}_{\chi} (i',j',r',0) \\ \iff\ & r = r' \ \wedge\ i = i' \ \wedge\ j' = j+1 \\ \iff\ & r = r' \ \wedge\ ((i,j),(i',j')) \in [\![\downarrow]\!]_{T_\chi} \\ \iff\ & ((i,j),(i',j')) \in [\![\mathit{next}^{2,1}_{r,r'}]\!]_{T_\chi} \end{aligned}$$

We are now ready to prove (1).

($\Rightarrow$): First note that $((i,j),(i',j')) \in [\![\mathcal{A}^h_r]\!]_{T_\chi}$ always implies $j = 0$, since the automaton has to read $\{\neg\langle\uparrow\rangle\}?$ before it can accept at all (its initial state ι is not a final state).

Consider an (accepting) execution

$$\iota \xrightarrow{\pi_1} (r_1,h_1) \xrightarrow{\pi_2} \ldots \xrightarrow{\pi_\ell} (r_\ell,h_\ell) = (h,r)$$

of $\mathcal{A}^h_r$, with $\ell \geq 1$, $\pi_1 = \{\neg\langle\uparrow\rangle\}?$, and $\pi_l = (\theta_l)^{h_{l-1},h_l}_{r_{l-1},r_l}$ for all $l \in \{2,\ldots,\ell\}$, connecting $(u,0)$ with (i,j). That is, $((u,0),(i,j)) \in [\![\pi_1 \cdot \ldots \cdot \pi_\ell]\!]_{T_\chi}$. We have to show $\chi^0_u(\mathit{id}) = \chi^j_i[h](r)$.

There are positions $(u,0) = (i_0,j_0),(i_1,j_1),\ldots,(i_\ell,j_\ell) = (i,j) \in \mathit{Pos}(\chi)$ such that $((i_{l-1},j_{l-1}),(i_l,j_l)) \in [\![\pi_l]\!]_{T_\chi}$ for all $l \in [\ell]$. By (2), we obtain

$$(i_1,j_1,r_1,h_1) \xrightarrow{\theta_2}_{\chi} (i_2,j_2,r_2,h_2) \xrightarrow{\theta_3}_{\chi} \ldots \xrightarrow{\theta_\ell}_{\chi} (i_\ell,j_\ell,r_\ell,h_\ell)\,.$$

This implies $\chi^{j_1}_{i_1}[h_1](r_1) = \chi^{j_\ell}_{i_\ell}[h_\ell](r_\ell)$, which equals $\chi^j_i[h](r)$. Since $\pi_1 = \{\neg\langle\uparrow\rangle\}?$, we also have $(u,0) = (i_1,j_1)$ and, therefore, $\chi^0_u(\mathit{id}) = \chi^{j_1}_{i_1}[h_1](r_1)$. We conclude $\chi^0_u(\mathit{id}) = \chi^j_i[h](r)$.

($\Leftarrow$): Suppose $\chi^0_u(\mathit{id}) = \chi^j_i[h](r)$. We will show that $((u,0),(i,j)) \in [\![\mathcal{A}^h_r]\!]_{T_\chi}$.

By the semantics of $\mathcal{D}$, pid $\chi^0_u(\mathit{id})$ has to be transmitted along transitions or messages. Thus, there are $\ell \geq 1$, positions $(i_1,j_1),\ldots,(i_\ell,j_\ell) = (i,j) \in \mathit{Pos}(\chi)$, registers $r_1,\ldots,r_\ell = r$, stages $0 = h_1,\ldots,h_\ell = h \in \{0,1,2\}$, and $\theta_2,\ldots,\theta_\ell \in \{\mathit{loc},\mathit{upd},\mathit{next},\mathit{msg}\}$ such that

- $((u,0),(i_1,j_1)) \in [\![\neg\langle\uparrow\rangle]\!]_{T_\chi}$ (therefore, $(u,0) = (i_1,j_1)$), and
- $(i_{l-1},j_{l-1},r_{l-1},h_{l-1}) \xrightarrow{\theta_\ell}_{\chi} (i_l,j_l,r_l,h_l)$ for all $l \in \{2,\ldots,\ell\}$.

By (2), we have

$$((i_{l-1},j_{l-1}),(i_l,j_l)) \in [\![(\theta_l)^{h_{l-1},h_l}_{r_{l-1},r_l}]\!]_{T_\chi}$$

for all $l \in \{2,\ldots,\ell\}$. We deduce

$$((u,0),(i,j)) = ((u,0),(i_l,j_l)) \in [\![\mathcal{A}^{h_\ell}_{r_\ell}]\!]_{T_\chi} = [\![\mathcal{A}^h_r]\!]_{T_\chi}\,.$$

This concludes the proof of Lemma 16. ☐

Lemma 17. *For all $T = (n,k,\lambda) \in \mathcal{T}_{\mathcal{D}}$ and $i,i' \in [n]$, we have*

$$((i,0),(i',0)) \in [\![\pi_<]\!]_T \iff \forall \chi \in \mathit{Runs}(T) : \chi^0_i(\mathit{id}) < \chi^0_{i'}(\mathit{id})\,.$$

Proof. There are two directions to show.

($\Rightarrow$): Suppose $((i,0),(i',0)) \in [\![\pi_<]\!]_T$. Then, there are $\ell \geq 1$ and $i = i_0,\ldots,i_\ell = i'$ such that

$$((i_{l-1},0),(i_l,0)) \in [\![\ \sum_{r,r' \in \mathit{Reg}} \mathcal{A}^1_r \cdot \{r < r'\}? \cdot (\mathcal{A}^1_{r'})^{-1}\ ]\!]_T$$

for all $l \in [\ell]$. Let $\chi \in \mathit{Runs}(T)$. By Lemma 16, we have $\chi^0_{i_{l-1}}(id) < \chi^0_{i_l}(id)$ for all $l \in [\ell]$. We deduce $\chi^0_i(id) < \chi^0_{i'}(id)$.

($\Leftarrow$): We denote the processes in question by u and u'. Suppose that $((u,0),(u',0)) \notin [\![\pi_<]\!]_T$. We are going to show that there is $\chi \in \mathit{Runs}(T)$ such that $\chi^0_u(id) \geq \chi^0_{u'}(id)$. Let $\prec = \{(i,i') \mid ((i,0),(i',0)) \in [\![\pi_<]\!]_T\}$. In particular, $u \not\prec u'$. By direction ($\Rightarrow$), we have that $\prec$ is a (strict) partial order.

Let $\mathcal{R} = (n : p_1, \ldots, p_n)$ be any ring such that (i) $p_u \geq p_{u'}$ and (ii) for all $i, i' \in [n]$, $i \prec i'$ implies $p_i < p_{i'}$. 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* $\mathcal{R}$-run

$$\chi = C_0 \overset{t^1}{\rightsquigarrow} C_1 \overset{t^2}{\rightsquigarrow} \ldots \overset{t^k}{\rightsquigarrow} C_k$$

(where $t^j = (t^j_1, \ldots, t^j_n) \in \Delta^n$) such that $T_\chi = T$. We will show that χ is indeed also an $\mathcal{R}$-run, which concludes the proof.

Let $(i,j) \in \mathit{Pos}(T)$ and $r, r' \in \mathit{Reg}$ such that $(r < r') \in t^j_i$. We have to show that $\hat{\chi}^j_i(r) < \hat{\chi}^j_i(r')$. By Lemma 16, there are $o, o' \in [n]$ such that

- $\chi^0_o(id) = \hat{\chi}^j_i(r)$ and $\chi^0_{o'}(id) = \hat{\chi}^j_i(r')$, and
- $((o,0),(i,j)) \in [\![\mathcal{A}^1_r]\!]_{T_\chi}$ and $((o',0),(i,j)) \in [\![\mathcal{A}^1_{r'}]\!]_{T_\chi}$.

The latter implies

$$((o,0),(o',0)) \in [\![\mathcal{A}^1_r \cdot \{r < r'\}? \cdot (\mathcal{A}^1_{r'})^{-1}]\!]_{T_\chi} .$$

In particular, $((o,0),(o',0)) \in [\![\pi_<]\!]_{T_\chi}$. We deduce $o \prec o'$. This implies $\chi^0_o(id) < \chi^0_{o'}(id)$. We conclude that $\chi^j_i(r) < \chi^j_i(r')$.

Finally, let $(i,j) \in \mathit{Pos}(T)$ and $r, r' \in \mathit{Reg}$ such that $(r = r') \in t^j_i$. Since $\mathit{Runs}(T) \neq \emptyset$, there is a run that validates guard $r = r'$ at coordinate (i,j). By Lemma 16, this is actually true for all pseudo runs of T. We deduce $\chi^j_i(r) = \chi^j_i(r')$.

Note that run condition 1. is satisfied, since $T \in \mathcal{T}_\mathcal{D}$. This concludes the proof. ◻

We will now proceed to the proof of Lemma 12.

Proof of Lemma 12. Recall that we have to show $L(\psi_\mathcal{D}) = \mathcal{T}_\mathcal{D}$, where $\psi_\mathcal{D} = \psi_= \wedge \psi_< \wedge \psi_{\mathsf{col}}$.

($\subseteq$): Let $T = (n,k,\lambda) \in L(\psi_\mathcal{D})$. We will show $T \in \mathcal{T}_\mathcal{D}$ by constructing a run χ of $\mathcal{D}$ such that $T_\chi = T$.

Again, let $\prec = \{(i,i') \mid ((i,0),(i',0)) \in [\![\pi_<]\!]_T\}$. As $T,(1,0) \models \psi_< = \neg\langle\rightarrow^*\rangle\mathsf{loop}(\pi_<)$, we have that $\prec$ is a strict partial order. Choose any ring $\mathcal{R} = (n : p_1, \ldots, p_n)$ such that, for all $i, i' \in [n]$, $i \prec i'$ implies $p_i < p_{i'}$. There is a unique pseudo $\mathcal{R}$-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 $T_\chi = T$. Let $j \in [k]$. We have to show $C_{j-1} \overset{t^j}{\rightsquigarrow} C_j$ where, this time, all run conditions are checked. Condition 4. of the definition of $\rightsquigarrow$ is satisfied thanks to the definition of a pseudo run. Condition 1. is ensured by $T \in L(\psi_{\mathsf{col}})$. Let $i \in [n]$ and suppose $(r = r') \in t^j_i$. We have $T,(i,j) \models \mathsf{loop}((\mathcal{A}^1_r)^{-1} \cdot \mathcal{A}^1_{r'})$. By Lemma 16, we have $\hat{\chi}^j_i(r) = \hat{\chi}^j_i(r')$. Finally, suppose $(r < r') \in t^j_i$. We proceed like in the reverse direction of the proof of Lemma 17 to show that $\hat{\chi}^j_i(r) < \hat{\chi}^j_i(r')$.

Altogether, it follows that χ is a run.

($\supseteq$): Let $T = (n,k,\lambda) \in \Delta^{++}$ such that $T \notin L(\psi_\mathcal{D})$. To show $T \notin \mathcal{T}_\mathcal{D}$, we distinguish three (non-disjoint) cases.

- Suppose $T \notin L(\psi_{\mathsf{col}})$. Obviously, this implies $T \notin \mathcal{T}_{\mathcal{D}}$.
- Suppose $T \notin (\psi_{=})$. Recall that

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

  Thus, there are a coordinate $(i,j) \in [n] \times [k]_0$ and registers $r_1, r_2 \in Reg$ such that we have $(r_1 = r_2) \in T[i,j]$ and $T,(i,j) \not\models \mathsf{loop}((\mathcal{A}^1_{r_1})^{-1} \cdot \mathcal{A}^1_{r_2})$. Towards a contradiction, suppose there is $\chi \in Runs(T)$. By Lemma 16, there are (unique) $i_1, i_2 \in [n]$ such that $\chi^0_{i_1}(id) = \hat{\chi}^j_i(r_1)$ and $\chi^0_{i_2}(id) = \hat{\chi}^j_i(r_2)$, as well as $((i_1,0),(i,j)) \in [\![\mathcal{A}^1_{r_1}]\!]_T$ and $(i_2,0),(i,j)) \in [\![\mathcal{A}^1_{r_2}]\!]_T$. Since $T,(i,j) \not\models \mathsf{loop}((\mathcal{A}^1_{r_1})^{-1} \cdot \mathcal{A}^1_{r_2})$, we have that $i_1 \neq i_2$. We deduce $\hat{\chi}^j_i(r_1) \neq \hat{\chi}^j_i(r_2)$, which contradicts $(r_1 = r_2) \in T[i,j]$. Altogether, we obtain $T \notin \mathcal{T}_{\mathcal{D}}$.
- Suppose $T \notin L(\psi_<)$ where $\psi_< \;=\; \neg\langle\rightarrow^*\rangle\mathsf{loop}(\pi_<)$. Then, there is $i \in [n]$ such that $T,(i,0) \models \mathsf{loop}(\pi_<)$. By Lemma 17, we have $\chi^0_i(id) < \chi^0_i(id)$ for all runs $\chi \in Runs(T)$. Thus, $Runs(T) = \emptyset$ and, therefore, $T \notin \mathcal{T}_{\mathcal{D}}$.

This concludes the proof of Lemma 12. □

E Proof of Lemma 13

We show a more general statement. First, call a local DataPDL$^\ominus$ formula φ *good* if it does not contain any guard of the form $<$ or $\leq$. Recall that we set $\mathcal{T}_{\mathcal{D}} = \{T_\chi \mid \chi \text{ is a run of } \mathcal{D}\}$ and, for a table $T \in \Delta^{++}$, $Runs(T) = \{\chi \mid \chi \text{ is run of } \mathcal{D} \text{ such that } T_\chi = T\}$.

We will simultaneously show the following statements:

- For all local DataPDL$^\ominus(\mathcal{D})$ formulas φ:

  (a) for all $T \in \mathcal{T}_{\mathcal{D}}$ and $(i,j) \in Pos(T)$,

  $$T,(i,j) \models \widetilde{\varphi} \iff \chi, 1, (i,j) \models \varphi \text{ for all } \chi \in Runs(T).$$

- For all good local DataPDL$^\ominus(\mathcal{D})$ formulas φ:

  (b) for all runs χ of $\mathcal{D}$ and all $(i,j) \in Pos(\chi)$,

  $$T_\chi,(i,j) \models \widetilde{\varphi} \iff \chi, 1, (i,j) \models \varphi.$$

- For all DataPDL$^\ominus(\mathcal{D})$ path formulas π:

  (c) for all runs χ of $\mathcal{D}$, we have $[\![\tilde{\pi}]\!]_{T_\chi} = [\![\pi]\!]_{\chi,1}$.

We first consider local formulas. We proceed by induction on the structure of φ. Note that (b) is a stronger statement: when we show that (b) holds for a formula, then (a) holds for that formula, too.

- Suppose $\varphi = \mathsf{m}$. It is enough to show (b). Recall that $\widetilde{\mathsf{m}} = \neg\langle\leftarrow\rangle$. We have $T_\chi,(i,j) \models \neg\langle\leftarrow\rangle \iff i = 1 \iff \chi, 1, (i,j) \models \mathsf{m}$.

- Suppose $\varphi = s \in S$. Again, it is enough to show (b). Recall that $\widetilde{s} = \textbf{goto}\ s$. By the definition of runs, the semantics of DataPDL$^\ominus$, and T_χ, we have that $T_\chi,(i,j) \models \textbf{goto}\ s \iff \chi, 1, (i,j) \models s$.

- Consider $\neg\varphi$. Then, φ is a good formula. Recall that $\widetilde{\neg\varphi} = \neg\widetilde{\varphi}$. We have $T_\chi, (i,j) \models \neg\widetilde{\varphi} \iff T_\chi, (i,j) \not\models \widetilde{\varphi} \iff$ (by I.H.(b)) $\chi, 1, (i,j) \not\models \varphi \iff \chi, 1, (i,j) \models \neg\varphi$.

- Suppose $\varphi = (\varphi_1 \wedge \varphi_2)$.

(a) We have

$$\begin{aligned}
& T, (i,j) \models \widetilde{\varphi_1} \wedge \widetilde{\varphi_2} \\
\iff\ & T, (i,j) \models \widetilde{\varphi_1} \text{ and } T, (i,j) \models \widetilde{\varphi_2} \\
\overset{\text{I.H.(a)}}{\iff}\ & \big(\chi, 1, (i,j) \models \varphi_1 \text{ for all } \chi \in \mathit{Runs}(T)\big) \text{ and} \\
& \big(\chi, 1, (i,j) \models \varphi_2 \text{ for all } \chi \in \mathit{Runs}(T)\big) \\
\iff\ & \chi, 1, (i,j) \models \varphi_1 \wedge \varphi_2 \text{ for all } \chi \in \mathit{Runs}(T)
\end{aligned}$$

(b) Suppose φ_1 and φ_2 are good. We have

$$\begin{aligned}
& T_\chi, (i,j) \models \widetilde{\varphi_1} \wedge \widetilde{\varphi_2} \\
\iff\ & T_\chi, (i,j) \models \widetilde{\varphi_1} \text{ and } T_\chi, (i,j) \models \widetilde{\varphi_2} \\
\overset{\text{I.H.(b)}}{\iff}\ & \chi, 1, (i,j) \models \varphi_1 \text{ and } \chi, 1, (i,j) \models \varphi_2 \\
\iff\ & \chi, 1, (i,j) \models \varphi_1 \wedge \varphi_2
\end{aligned}$$

- Consider $\varphi = (\varphi_1 \Rightarrow \varphi_2)$. Then, φ_1 is good.

(a) There are two directions to show:

($\Rightarrow$): We have

$$\begin{aligned}
& T, (i,j) \models \widetilde{\varphi_1} \Rightarrow \widetilde{\varphi_2} \\
\implies\ & T, (i,j) \not\models \widetilde{\varphi_1} \text{ or } T, (i,j) \models \widetilde{\varphi_2} \\
\overset{\text{I.H.(b),(a)}}{\implies}\ & \big(\chi, 1, (i,j) \not\models \varphi_1 \text{ for all } \chi \in \mathit{Runs}(T)\big) \text{ or} \\
& \big(\chi, 1, (i,j) \models \varphi_2 \text{ for all } \chi \in \mathit{Runs}(T)\big) \\
\implies\ & \chi, 1, (i,j) \models \varphi_1 \Rightarrow \varphi_2 \text{ for all } \chi \in \mathit{Runs}(T)
\end{aligned}$$

($\Leftarrow$): We have

$$\begin{aligned}
& T, (i,j) \not\models \widetilde{\varphi_1} \Rightarrow \widetilde{\varphi_2} \\
\implies\ & T, (i,j) \models \widetilde{\varphi_1} \text{ and } T, (i,j) \not\models \widetilde{\varphi_2} \\
\overset{\text{I.H.(b),(a)}}{\implies}\ & \big(\chi, 1, (i,j) \models \varphi_1 \text{ for all } \chi \in \mathit{Runs}(T)\big) \text{ and} \\
& \big(\chi, 1, (i,j) \not\models \varphi_2 \text{ for some } \chi \in \mathit{Runs}(T)\big) \\
\implies\ & \chi, 1, (i,j) \not\models \varphi_1 \Rightarrow \varphi_2 \text{ for some } \chi \in \mathit{Runs}(T)
\end{aligned}$$

(b) Here, we require that both φ_1 and φ_2 are good. Then,

$$\begin{aligned}
& T_\chi, (i,j) \models \widetilde{\varphi_1} \Rightarrow \widetilde{\varphi_2} \\
\iff\ & T_\chi, (i,j) \not\models \widetilde{\varphi_1} \text{ or } T_\chi, (i,j) \models \widetilde{\varphi_2} \\
\overset{\text{I.H.(b)}}{\iff}\ & \chi, 1, (i,j) \not\models \varphi_1 \text{ or } \chi, 1, (i,j) \models \varphi_2 \\
\iff\ & \chi, 1, (i,j) \models \varphi_1 \Rightarrow \varphi_2
\end{aligned}$$

- Consider formula $[\pi]\varphi$. Let $x = (i,j)$. For a set $A \subseteq \mathit{Pos}(T) \times \mathit{Pos}(T)$, let $A(x) = \{x' \in \mathit{Pos}(T) \mid (x, x') \in A\}$.

(a) We have

$$
\begin{aligned}
& T, x \models [\widetilde{\pi}]\widetilde{\varphi} \\
\iff\quad & \forall x' \in [\![\widetilde{\pi}]\!]_T(x) : T, x' \models \widetilde{\varphi} \\
\overset{\text{I.H.(a)}}{\iff}\quad & \forall x' \in [\![\widetilde{\pi}]\!]_T(x) : \forall \chi \in \mathit{Runs}(T) : \chi, 1, x' \models \varphi \\
\iff\quad & \forall \chi \in \mathit{Runs}(T) : \forall x' \in [\![\widetilde{\pi}]\!]_{T_\chi}(x) : \chi, 1, x' \models \varphi \\
\overset{\text{I.H.(c)}}{\iff}\quad & \forall \chi \in \mathit{Runs}(T) : \forall x' \in [\![\pi]\!]_{\chi,1}(x) : \chi, 1, x' \models \varphi \\
\iff\quad & \forall \chi \in \mathit{Runs}(T) : \chi, 1, x \models [\pi]\varphi
\end{aligned}
$$

(b) Suppose φ is good. We have

$$
\begin{aligned}
& T_\chi, x \models [\widetilde{\pi}]\widetilde{\varphi} \\
\iff\quad & \forall x' \in [\![\widetilde{\pi}]\!]_{T_\chi}(x) : T_\chi, x' \models \widetilde{\varphi} \\
\overset{\text{I.H.(b)}}{\iff}\quad & \forall x' \in [\![\widetilde{\pi}]\!]_{T_\chi}(x) : \chi, 1, x' \models \varphi \\
\overset{\text{I.H.(c)}}{\iff}\quad & \forall x' \in [\![\pi]\!]_{\chi,1}(x) : \chi, 1, x' \models \varphi \\
\iff\quad & \chi, 1, x \models [\pi]\varphi
\end{aligned}
$$

- Suppose $\varphi = \langle\pi_1\rangle r_1 \leq \langle\pi_2\rangle r_2$. Then, π_1 and π_2 are both unambiguous. By I.H.(c), $\widetilde{\pi_1}$ and $\widetilde{\pi_2}$ are unambiguous (wrt. symbolic runs). We show (a):

$$
\begin{aligned}
& T, (i,j) \models \widetilde{\langle\pi_1\rangle r_1 \leq \langle\pi_2\rangle r_2} \\
\iff\quad & T, (i,j) \models \mathsf{loop}(\widetilde{\pi_1} \cdot (\mathcal{A}^2_{r_1})^{-1} \cdot (\pi_< + \epsilon) \cdot \mathcal{A}^2_{r_2} \cdot (\widetilde{\pi_2})^{-1})
\end{aligned}
$$

$\iff$ there are coordinates $(i_1, j_1), (i_2, j_2), (i'_1, 0), (i'_2, 0) \in \mathit{Pos}(T)$ such that:

1. $((i,j),(i_1,j_1)) \in [\![\widetilde{\pi_1}]\!]_T$ and $((i,j),(i_2,j_2)) \in [\![\widetilde{\pi_2}]\!]_T$
2. $((i'_1,0),(i_1,j_1)) \in [\![\mathcal{A}^2_{r_1}]\!]_T$ and $((i'_2,0),(i_2,j_2)) \in [\![\mathcal{A}^2_{r_2}]\!]_T$
3. $((i'_1,0),(i'_2,0)) \in [\![\pi_<]\!]_T$ or $i'_1 = i'_2$

$\overset{(*)}{\iff}$ there exist coordinates $(i_1, j_1), (i_2, j_2), (i'_1, 0), (i'_2, 0) \in \mathit{Pos}(T)$ such that:

1. $\forall \chi \in \mathit{Runs}(T) : ((i,j),(i_1,j_1)) \in [\![\pi_1]\!]_{\chi,1}$ and $((i,j),(i_2,j_2)) \in [\![\pi_2]\!]_{\chi,1}$
2. $\forall \chi \in \mathit{Runs}(T) : \chi^0_{i'_1}(\mathit{id}) = \chi^{j_1}_{i_1}(r_1)$ and $\chi^0_{i'_2}(\mathit{id}) = \chi^{j_2}_{i_2}(r_2)$
3. $\big(\forall \chi \in \mathit{Runs}(T) : \chi^0_{i'_1}(\mathit{id}) < \chi^0_{i'_2}(\mathit{id})\big)$ or $i'_1 = i'_2$

$\iff$ there exist coordinates $(i_1, j_1), (i_2, j_2), (i'_1, 0), (i'_2, 0) \in \mathit{Pos}(T)$ such that:

1. $\forall \chi \in \mathit{Runs}(T) : ((i,j),(i_1,j_1)) \in [\![\pi_1]\!]_{\chi,1}$ and $((i,j),(i_2,j_2)) \in [\![\pi_2]\!]_{\chi,1}$
2. $\forall \chi \in \mathit{Runs}(T) : \chi^0_{i'_1}(\mathit{id}) = \chi^{j_1}_{i_1}(r_1)$ and $\chi^0_{i'_2}(\mathit{id}) = \chi^{j_2}_{i_2}(r_2)$
3. $\forall \chi \in \mathit{Runs}(T) : \chi^0_{i'_1}(\mathit{id}) < \chi^0_{i'_2}(\mathit{id})$ or $\chi^0_{i'_1}(\mathit{id}) = \chi^0_{i'_2}(\mathit{id})$

$\overset{(**)}{\iff}$ for all $\chi \in \mathit{Runs}(T)$, there are $(i_1, j_1), (i_2, j_2), (i'_1, 0), (i'_2, 0) \in \mathit{Pos}(T)$ such that:

1. $((i,j),(i_1,j_1)) \in [\![\pi_1]\!]_{\chi,1}$ and $((i,j),(i_2,j_2)) \in [\![\pi_2]\!]_{\chi,1}$
2. $\chi^0_{i'_1}(\mathit{id}) = \chi^{j_1}_{i_1}(r_1)$ and $\chi^0_{i'_2}(\mathit{id}) = \chi^{j_2}_{i_2}(r_2)$
3. $\chi^0_{i'_1}(\mathit{id}) \leq \chi^0_{i'_2}(\mathit{id})$

$\iff \quad \forall \chi \in \mathit{Runs}(T) : \chi, 1, (i,j) \models \langle \pi_1 \rangle r_1 \leq \langle \pi_2 \rangle r_2$

$(*)$ by I.H.(c), and Lemmas 16 and 17

$(**)$ by I.H.(c), Lemmas 16 and 17, and the fact that π_1 and π_2 are unambiguous, the coordinates are uniquely determined by T, (i,j), and φ

- The case $\varphi = \langle \pi_1 \rangle r_1 < \langle \pi_2 \rangle r_2$ is simpler than the previous one. We just have to adapt 3. accordingly.

- Consider the case $\varphi = \big(\langle \pi_1 \rangle r_1 \neq \langle \pi_2 \rangle r_2\big)$. We show (b):

$T_\chi, (i,j) \models \widetilde{\langle \pi_1 \rangle r_1 \neq \langle \pi_2 \rangle r_2}$

$\iff \quad T_\chi, (i,j) \models \mathsf{loop}(\widetilde{\pi_1} \cdot (\mathcal{A}^2_{r_1})^{-1} \cdot (\leftarrow^+ + \rightarrow^+) \cdot \mathcal{A}^2_{r_2} \cdot (\widetilde{\pi_2})^{-1})$

$\iff$ there are coordinates $(i_1, j_1), (i_2, j_2), (i'_1, 0), (i'_2, 0) \in \mathit{Pos}(\chi)$ such that:

1. $((i,j),(i_1,j_1)) \in \llbracket \widetilde{\pi_1} \rrbracket_{T_\chi}$ and $((i,j),(i_2,j_2)) \in \llbracket \widetilde{\pi_2} \rrbracket_{T_\chi}$

2. $((i'_1,0),(i_1,j_1)) \in \llbracket \mathcal{A}^2_{r_1} \rrbracket_{T_\chi}$ and $((i'_2,0),(i_2,j_2)) \in \llbracket \mathcal{A}^2_{r_2} \rrbracket_{T_\chi}$

3. $i'_1 \neq i'_2$

$\iff$ (by I.H.(c) and Lemma 16)
there are coordinates $(i_1, j_1), (i_2, j_2), (i'_1, 0), (i'_2, 0) \in \mathit{Pos}(\chi)$ such that:

1. $((i,j),(i_1,j_1)) \in \llbracket \pi_1 \rrbracket_{\chi,1}$ and $((i,j),(i_2,j_2)) \in \llbracket \pi_2 \rrbracket_{\chi,1}$

2. $\chi^0_{i'_1}(id) = \chi^{j_1}_{i_1}(r_1)$ and $\chi^0_{i'_2}(id) = \chi^{j_2}_{i_2}(r_2)$

3. $i'_1 \neq i'_2$

$\iff \quad \chi, 1, (i,j) \models \langle \pi_1 \rangle r_1 \neq \langle \pi_2 \rangle r_2$

- The case $\varphi = \big(\langle \pi_1 \rangle r_1 = \langle \pi_2 \rangle r_2\big)$ is almost identical. In 3., we just replace $\neq$ by $=$.

- Consider the path formula $\pi = \{\varphi\}?$. Note that φ is good. We show (c):

$$\begin{aligned}
& \llbracket \widetilde{\{\varphi\}?} \rrbracket_{T_\chi} = \llbracket \{\widetilde{\varphi}\}? \rrbracket_{T_\chi} \\
= \quad & \{(x,x) \mid x \in \mathit{Pos}(\chi) : T_\chi, x \models \widetilde{\varphi}\} \\
\overset{\text{I.H.(b)}}{=} \quad & \{(x,x) \mid x \in \mathit{Pos}(\chi) : \chi, 1, x \models \varphi\} \\
= \quad & \llbracket \{\varphi\}? \rrbracket_{\chi,1}
\end{aligned}$$

- Consider $\pi = \rightarrow$. Suppose the coordinate set of χ is $[n] \times [k]_0$. We show (c):

$$\begin{aligned}
& \llbracket \widetilde{\rightarrow} \rrbracket_{T_\chi} = \llbracket \rightarrow + \{\neg\langle \rightarrow \rangle\}?\leftarrow^*\{\neg\langle \leftarrow \rangle\}? \rrbracket_{T_\chi} \\
= \quad & \{((i,j),(i+1,j)) \mid (i,j) \in [n-1] \times [k]_0\} \cup \{((n,j),(1,j)) \mid j \in [k]_0\} \\
= \quad & \llbracket \rightarrow \rrbracket_{\chi,1}
\end{aligned}$$

- The regular operations as well as $\uparrow$ and $\downarrow$ are obvious, and the case $\leftarrow$ is symmetric to $\rightarrow$.

F Proof of Lemma 14

Let us prove (a).

($\Rightarrow$): Suppose $\mathcal{D} \models \Phi = \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\varphi$. Let $T \in L(\psi_{\mathcal{D}})$. By Lemma 12, there is a run χ of $\mathcal{D}$ such that $T_\chi = T$. Moreover, since $\mathcal{D} \models \Phi$, all runs χ of $\mathcal{D}$ satisfy $\chi, 1, (1, 0) \models \varphi$. This applies, in particular, to all runs χ such that $T_\chi = T$. By Lemma 13, we have $T, (1, 0) \models \widetilde{\varphi}$. We conclude $L(\psi_{\mathcal{D}} \wedge \neg\widetilde{\varphi}) = \emptyset$.

($\Leftarrow$): Suppose $\mathcal{D} \not\models \forall_{rings}\forall_{runs}\forall_{\mathsf{m}}\varphi$. Then, there are a ring $\mathcal{R} = (n : \ldots)$, an $\mathcal{R}$-run χ of $\mathcal{D}$, and a process $m \in [n]$ such that $\chi, m, (m, 0) \not\models \varphi$. Since φ cannot distinguish isomorphic rings, we can shift $\mathcal{R}$ until m "arrives" on position 1. Thus, there are $\mathcal{R}' = (n : \ldots)$ and an $\mathcal{R}'$-run χ' of $\mathcal{D}$ such that $\chi', 1, (1, 0) \not\models \varphi$. By Lemma 13, $T_{\chi'}, (1, 0) \not\models \widetilde{\varphi}$ and, therefore, $T_{\chi'}, (1, 0) \models \neg\widetilde{\varphi}$. Due to Lemma 12, we also have $T_{\chi'}, (1, 0) \models \psi_{\mathcal{D}}$. we conclude $L(\psi_{\mathcal{D}} \wedge \neg\widetilde{\varphi}) \neq \emptyset$.

Part (b) is shown in exactly the same way, restricting the height of a table and length of a run by the given bound b.