Parameterized Verification of Communicating Automata under Context Bounds

Benedikt Bollig[1], Paul Gastin[1], and Jana Schubert[2]

[1] LSV, ENS Cachan & CNRS
{bollig,gastin}@lsv.ens-cachan.fr
[2] Fakultät für Informatik, TU Dresden
jana.schubert@tu-dresden.de

Abstract. We study the verification problem for parameterized communicating automata (PCA), in which processes synchronize via message passing. A given PCA can be run on any topology of bounded degree (such as pipelines, rings, or ranked trees), and communication may take place between any two processes that are adjacent in the topology. Parameterized verification asks if there is a topology from a given topology class that allows for an accepting run of the given PCA. In general, this problem is undecidable even for synchronous communication and simple pipeline topologies. We therefore consider context-bounded verification, which restricts the behavior of each single process. For several variants of context bounds, we show that parameterized verification over pipelines, rings, and ranked trees is decidable. More precisely, it is PSPACE-complete for pipelines and rings, and EXPTIME-complete for ranked trees. Our approach is automata-theoretic. We build a finite (tree, respectively) automaton that identifies those topologies that allow for an accepting run of the given PCA. The verification problem then reduces to checking nonemptiness of that automaton.

1 Introduction

Communicating automata (CA) are a fundamental and well-studied model of parallel systems [7]. They consist of finite-state machines that exchange messages over channels determined by a fixed and known communication topology. CA are known to be Turing equivalent so that even basic problems of formal verification such as reachability are undecidable. Therefore, modifications and restrictions have been considered which bring back decidability. Reachability is decidable, for example, when the analysis is restricted to executions with a fixed maximum number of pending messages, or when channels are lossy [2].

In some contexts such as ad-hoc networks, multi-core programming, or communication-protocol verification, assuming a fixed and known communication topology is not appropriate. Lately, there has been a lot of (ongoing) research in the area of *parameterized* verification [1, 3, 9, 13, 14], which aims to validate a given system independently of the number of processes and the communication topology. A lot of different models of such systems have been proposed (cf. [12]

for a recent survey). In this paper, we investigate the reachability problem for parameterized communicating automata (PCAs). A PCA is a collection of finite automata that can be plugged into *any* communication topology of bounded degree. PCAs have recently been introduced to initiate a logical study of parameterized systems [5]. Their verification problem has not been considered. Roughly, it can be stated as follows: Given a PCA $\mathcal{A}$ and a regular set $\mathfrak{T}$ of pipeline, ring, or tree topologies, is there a topology $\mathcal{T} \in \mathfrak{T}$ such that $\mathcal{A}$ has an accepting run on $\mathcal{T}$? Here, "regular" means given by some finite automaton (for pipelines and rings) or tree automaton (for tree topologies), which is part of the input. Note that there is also a universal variant of that problem, and our decision procedures will take care of that case as well.

We actually consider a restriction of PCAs with rendez-vous synchronization, albeit distinguishing between send and receive events. This considerably simplifies the presentation, but the overall approach can be extended to systems with asynchronous bounded channels. Note that rendez-vous communication can also be seen as an underapproximation of the latter.

While bounding the channel capacity or imposing rendez-vous communication bring back decidability of reachability for CA with fixed communication topology, this is no longer true in the case of PCAs. For various other (undecidable) models of concurrent systems, decidability is achieved by introducing a context (or "phase") bound, limiting the part of the model simulating synchronization or communication of concurrent processes [6,15,16,18,19]. We adopt the general approach, but introduce new natural definitions of contexts that are suitable for our setting. An *interface-context* restricts communication of a process to one neighbor in the topology (e.g., the left neighbor in the pipeline). Another context type separates send from receive events while restricting reception to one interface. Imposing such bounds is justified, as many distributed algorithms use a bounded number of contexts, such as certain leader-election protocols, P2P protocols, etc.

We show that context-bounded parameterized verification is decidable: it is PSPACE-complete for pipelines and rings, and EXPTIME-complete for ranked trees. Our decidability proof is automata-theoretic and uniform. We transform a given PCA $\mathcal{A}$, in several steps, into a topology acceptor (a finite automaton or a tree automaton) that recognizes the set of pipeline and, respectively, tree topologies allowing for an accepting run of $\mathcal{A}$. For rings, an additional adjustment is needed, which rules out cyclic behaviors that the topology acceptor is not able to detect on its own.

Related Work. Parameterized verification can be classified into verification of multithreaded programs running on a single core, and protocol verification. Context-bounded verification for systems consisting of an unbounded number of threads has already been considered [4, 17]. In [4], a model with process creation is presented, in which a context switch is observed whenever an active thread is interrupted and resumed. In [17], an unbounded number of threads are scheduled in several rounds. In both cases, the context bound does not impose a bound on the number of threads. However, every thread will be resumed and

become active a bounded number of times. For protocol verification, which is based on the concept of independent (finite-state) processes communicating over a network-like structure, this does not seem to be suitable. For example, take four processes, $P_1, \ldots, P_4$. Suppose P_1 synchronizes unboundedly often with P_2, and P_3 synchronizes unboundedly often with P_4. In particular, no communication takes place between $\{P_1, P_2\}$ and $\{P_3, P_4\}$. Due to the absence of a global scheduler, there should be no bound on the number of switches between P_2 and P_3 (or P_1 and P_3, etc.). This issue is particularly important when a system is compared to a partial global specification that is not necessarily closed under permutation of independent events. Our *local* context definition does not impose any a priori bound on the number of switches between *independent* processes.

A versatile framework for parameterized verification, capturing rendez-vous communication in pipelines, rings, and trees, is presented in [1]. The verification problem is phrased in terms of minimal bad configurations, which does not necessitate context bounds. Motivated by ad-hoc networks, [9] considers systems modeled by finite automata that communicate in a broadcast or unicast manner. In the case of unicast communication, the recipient is chosen nondeterministically from the set of neighbors, which is incomparable with the unicast communication of PCAs. Direction-aware token-passing systems [3, 10, 11] can be modeled in our framework as far as bounded-degree structures such as rings are concerned. To the best of our knowledge, neither context bounds nor the PCA model have been considered yet for protocol verification.

Outline. Section 2 recapitulates basic notions such as words and finite (tree) automata. In Section 3, we introduce topologies, PCAs, and several context-bounded verification problems. Section 4 presents our main results and illustrates the crucial proof ideas. Missing details can be found in the full version of the paper: `http://hal.archives-ouvertes.fr/hal-00984421/`

2 Preliminaries

For $n \in \mathbb{N}$, we set $[n] := \{1, \ldots, n\}$. Let $\mathbb{A}$ be an alphabet, i.e., a nonempty finite set. The set of finite words over $\mathbb{A}$ is denoted by $\mathbb{A}^*$, which includes the empty word ε. The concatenation of words $w_1, w_2 \in \mathbb{A}^*$ is denoted by $w_1 \cdot w_2$ or $w_1.w_2$. Given an index set I and a tuple $a = (a_i)_{i \in I} \in \mathbb{A}^I$, we write $a|_i$ to denote a_i.

A *finite automaton* over $\mathbb{A}$ is a tuple $\mathcal{B} = (S, \Longrightarrow, \iota, F)$ where S is the finite set of states, $\iota \in S$ is the initial state, $F \subseteq S$ is the set of final states, and $\Longrightarrow \subseteq S \times \mathbb{A} \times S$ is the transition relation. We write $s \overset{a}{\Longrightarrow} s'$ instead of $(s, a, s') \in \Longrightarrow$. A run of $\mathcal{B}$ on a word $w = a_1 \ldots a_n \in \mathbb{A}^*$ is a sequence $s_0 s_1 \ldots s_n \in S^*$ of states such that $s_0 = \iota$ and $s_{i-1} \overset{a_i}{\Longrightarrow} s_i$ for all $i \in [n]$. The run is accepting if $s_n \in F$. Finally, the language of $\mathcal{B}$ is defined as $L(\mathcal{B}) := \{w \in \mathbb{A}^* \mid \text{there is an accepting run of } \mathcal{B} \text{ on } w\}$.

For trees, we fix a (maximal) *rank* $r \in \mathbb{N}$ with $r \geq 2$. An *r-tree* over $\mathbb{A}$ is a pair (V, π) where V is a nonempty finite prefix-closed subset of $\{1, \ldots, r\}^*$, and $\pi : V \to \mathbb{A}$ is a labeling function. The set V is the set of nodes of the tree, and

ε is its root. For $u \in V$ and $l \in [r]$ with $u.l \in V$, we say that $u.l$ is the l-th child of u. An *r-tree automaton* over $\mathbb{A}$ is a tuple $\mathcal{B} = (S, \Delta, F)$ where S is the finite set of states, $F \subseteq S$ is the set of final states, and $\Delta \subseteq S \times \mathbb{A} \times (S \uplus \{\bot\})^r$ is the transition relation. A run of $\mathcal{B}$ on an r-tree (V, π) is a mapping $\rho : V \to S$ such that, for all $u \in V$, $(\rho(u), \pi(u), (s_l)_{l \in [r]}) \in \Delta$ where $s_l = \rho(u.l)$ if $u.l \in V$, and $s_l = \bot$ if $u.l \notin V$. The run is accepting if $\rho(\varepsilon) \in F$. By $L(\mathcal{B})$, we denote the set of r-trees accepted by $\mathcal{B}$.

3 Parameterized Communicating Automata

In this section, we introduce our model of a communicating system that can be run on arbitrary topologies of bounded degree.

Topologies. A topology is a graph, whose nodes are connected via interfaces. The idea is that each node runs a finite-state process (of type $p, q, \ldots$). Some topologies are depicted in Figures 1–3. In Figure 1, for example, nodes are arranged in a pipeline, which allows a process to communicate with a left and a right neighbor (if they exist). When a node u emits a message m via its interface right, then m can be received by the neighbor on the right of u, using interface left. Let $\mathcal{N} = \{a, b, c, \ldots\}$ and $\mathcal{P} = \{p, q, \ldots\}$ be nonempty finite sets of *interface names* (or, simply, interfaces) and *process types*, respectively.

Definition 1. *A* topology *over $\mathcal{N}$ and $\mathcal{P}$ is a tuple $\mathcal{T} = (V, \nu, \pi)$ where V is the nonempty finite set of* nodes *(or* processes*), $\pi : V \to \mathcal{P}$ associates with every node a process type, and $\nu : V \times \mathcal{N} \rightharpoonup V$ is a partial mapping. Intuitively, $\nu(u, a) = v$ means that the interface a of u points to v. We suppose that, for all $u \in V$, there is at last one $a \in \mathcal{N}$ such that $\nu(u, a)$ is defined. Moreover, we require that $\nu(u, a) = v$ implies*

- *$u \neq v$ (there are no self-loops),*
- *$\nu(v, b) = u$ for some $b \in \mathcal{N}$ (adjacent processes are mutually connected), and*
- *$\nu(u, a') = v'$ implies $[a = a'$ iff $v = v']$, for all $a' \in \mathcal{N}$ and $v' \in V$ (an interface points to at most one process, and two distinct interfaces point to distinct processes).*

We write $u \overset{a\ b}{\longmapsto} v$ if $\nu(u, a) = v$ and $\nu(v, b) = u$, and we write $u \longmapsto v$ if $u \overset{a\ b}{\longmapsto} v$ for some $a, b \in \mathcal{N}$. This paper will focus on three topology classes:

Pipelines. A *pipeline* over a nonempty finite set $\mathcal{P}$ of process types is a topology over $\mathcal{N} = \{\mathsf{left}, \mathsf{right}\}$ and $\mathcal{P}$. It is of the form $\mathcal{T} = (\{1, \ldots, n\}, \nu, \pi)$, with $n \geq 2$, such that $\nu(i, \mathsf{right}) = i + 1$ and $\nu(i + 1, \mathsf{left}) = i$ for all $i \in [n - 1]$, and $\nu(1, \mathsf{left})$ and $\nu(n, \mathsf{right})$ are both undefined. A finite automaton $\mathcal{B}$ over $\mathcal{P}$ can be seen as a pipeline recognizer. Indeed, a pipeline is uniquely given by the sequence $\pi(1) \ldots \pi(n) \in \mathcal{P}^*$. So, we let $L_{pipe}(\mathcal{B})$ denote the set of pipelines $(\{1, \ldots, n\}, \nu, \pi)$ over $\mathcal{P}$ such that $\pi(1) \ldots \pi(n) \in L(\mathcal{B})$. Instead of $\mathcal{B}$, we may use a classical regular expression. An example pipeline is depicted in Figure 1. It is uniquely given by the word $pqpq$.

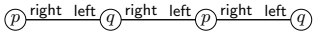

Fig. 1. Pipeline

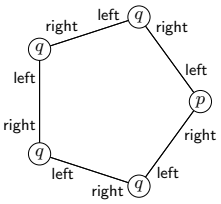

Fig. 2. Ring

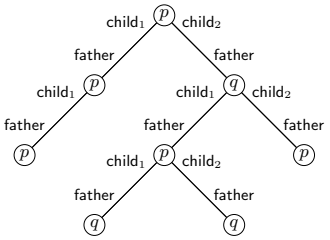

Fig. 3. Tree

Rings. A *ring* over $\mathcal{P}$ is a topology over $\mathcal{N} = \{\mathsf{left}, \mathsf{right}\}$ and $\mathcal{P}$ of the form $\mathcal{T} = (\{1, \ldots, n\}, \nu, \pi)$, with $n \geq 3$, where $\nu(i, \mathsf{right}) = (i \bmod n) + 1$ and $\nu((i \bmod n) + 1, \mathsf{left}) = i$ for all $i \in [n]$. Similarly to pipelines, a finite automaton $\mathcal{B}$ over $\mathcal{P}$ can be used as a *ring* recognizer: we let $L_{ring}(\mathcal{B})$ denote the set of rings $(\{1, \ldots, n\}, \nu, \pi)$ over $\mathcal{P}$ such that there is $i \in [n]$ satisfying $\pi(i) \ldots \pi(n)\pi(1) \ldots \pi(i-1) \in L(\mathcal{B})$. This takes into account that, a priori, rings do not have an "initial" node. Figure 2 depicts a ring with five nodes.

Trees. For $r \geq 2$, an *r-tree topology* over $\mathcal{P}$ is a topology $\mathcal{T} = (V, \nu, \pi)$ over $\{\mathsf{father}, \mathsf{child}_1, \ldots, \mathsf{child}_r\}$ and $\mathcal{P}$ such that (V, π) is an r-tree over $\mathcal{P}$, $\nu(\varepsilon, \mathsf{father})$ is undefined, and for all $u \in V$ and $l \in [r]$, we have (1) $u.l \in V$ implies $\nu(u, \mathsf{child}_l) = u.l$ and $\nu(u.l, \mathsf{father}) = u$, and (2) $u.l \notin V$ implies that $\nu(u, \mathsf{child}_l)$ is undefined. An r-tree automaton $\mathcal{B}$ over $\mathcal{P}$ can be seen as a recognizer for tree topologies: we write $L_{tree}(\mathcal{B})$ for the set of r-tree topologies (V, ν, π) such that $(V, \pi) \in L(\mathcal{B})$. A sample 2-tree topology is depicted in Figure 3.

The Automata Model. Next, we introduce our system model. As suggested above, a parameterized communicating automaton is a collection of finite-state processes whose actions refer to an interface. Unless stated otherwise, we assume that $\mathcal{N}$ is a *fixed* nonempty finite set of interface names.

Definition 2. *A* parameterized communicating automaton (PCA) *over $\mathcal{N}$ is a tuple $\mathcal{A} = (\mathcal{P}, Msg, (\mathcal{A}_p)_{p \in \mathcal{P}})$ where*

- *$\mathcal{P}$ is a nonempty finite set of process types,*
- *Msg is a nonempty finite set of* messages*, and*
- *$\mathcal{A}_p$ is a finite automaton over $\Sigma_{\mathcal{A}} := \{a!m\,,\ a?m \mid a \in \mathcal{N} \text{ and } m \in Msg\}$, for every $p \in \mathcal{P}$.*

We call the elements of $\Sigma_{\mathcal{A}}$ actions*.*

A *pipeline PCA* or *ring PCA* is a PCA over $\{\mathsf{left}, \mathsf{right}\}$. Moreover, for $r \geq 2$, an *r-tree PCA* is a PCA over $\{\mathsf{father}, \mathsf{child}_1, \ldots, \mathsf{child}_r\}$.

The idea is the following: When $\mathcal{A}$ is run on a topology (V, ν, π) with adjacent processes $u \overset{a\ \ b}{\longmapsto} v$, then u runs a copy of $\mathcal{A}_{\pi(u)}$ and can emit a message m through interface a by executing $a!m$. Process v receives the message if it is ready to execute $b?m$. We assume that communication is by rendez-vous, i.e., messages are received instantaneously.

For convenience, we write Σ instead of $\Sigma_{\mathcal{A}}$. Sometimes, we will even mention Σ without any reference to $\mathcal{A}$. However, notice that the alphabet depends on a PCA (more precisely, on $\mathcal{N}$ and a set of messages). Let $\Sigma_! := \{a!m \mid a \in \mathcal{N} \text{ and } m \in Msg\}$ and let $\Sigma_?$ be defined accordingly. These sets are further refined to $\Sigma_{a!}$ and $\Sigma_{a?}$, containing only those actions that refer to interface $a \in \mathcal{N}$.

Semantics of PCAs. Let $\mathcal{A} = (\mathcal{P}, Msg, (\mathcal{A}_p)_{p\in\mathcal{P}})$ be a PCA over $\mathcal{N}$, with $\mathcal{A}_p = (S_p, \Longrightarrow_p, \iota_p, F_p)$ for all $p \in \mathcal{P}$. The PCA $\mathcal{A}$ can be run on any topology $\mathcal{T} = (V, \nu, \pi)$ over $\mathcal{N}$ and $\mathcal{P}$. Its semantics wrt. $\mathcal{T}$ is a finite automaton $[\![\mathcal{A}]\!]^{\mathcal{T}} = (S, \Longrightarrow, \iota, F)$ over $\Sigma^{\mathcal{T}} \subseteq (\Sigma \cup \{\varepsilon\})^V$. The alphabet $\Sigma^{\mathcal{T}}$ contains, for all $v \overset{a\ b}{\longmapsto} v'$ and $m \in Msg$, the tuple $\langle v, m, v'\rangle := (\sigma_u)_{u\in V}$ where $\sigma_v = a!m$, $\sigma_{v'} = b?m$, and $\sigma_u = \varepsilon$ for all $u \in V \setminus \{v, v'\}$. For $W = \gamma_1 \dots \gamma_n \in (\Sigma^{\mathcal{T}})^*$ and $u \in V$, we define the projection of W to u as $W|_u := (\gamma_1|_u) \cdot \ldots \cdot (\gamma_n|_u) \in \Sigma^*$.

Given a process $u \in V$, we write $\mathcal{A}_u, S_u, \Longrightarrow_u, \iota_u, F_u$ as abbreviations for $\mathcal{A}_{\pi(u)}, S_{\pi(u)}, \Longrightarrow_{\pi(u)}, \iota_{\pi(u)}, F_{\pi(u)}$, respectively. The set of states of $[\![\mathcal{A}]\!]^{\mathcal{T}}$ is $S = \prod_{u\in V} S_u$, keeping track of the local state of every process in the topology. Accordingly, the initial state is $\iota = (\iota_u)_{u\in V}$, and the set of final states is $F = \prod_{u\in V} F_u$. The transition relation $\Longrightarrow \subseteq S \times \Sigma^{\mathcal{T}} \times S$ is defined as follows. Let $\mathrm{s} = (s_u)_{u\in V} \in S$, $\mathrm{s}' = (s'_u)_{u\in V} \in S$, and $\sigma = (\sigma_u)_{u\in V} \in \Sigma^{\mathcal{T}}$. Then, $\mathrm{s} \overset{\sigma}{\Longrightarrow} \mathrm{s}'$ if, for all $u \in V$, we have that $\sigma_u \neq \varepsilon$ implies $s_u \overset{\sigma_u}{\Longrightarrow}_u s'_u$, and $\sigma_u = \varepsilon$ implies $s_u = s'_u$. The language of $\mathcal{A}$ wrt. $\mathcal{T}$ is defined as $L(\mathcal{A}, \mathcal{T}) := L([\![\mathcal{A}]\!]^{\mathcal{T}})$.

Example 1. We consider a simplified version of the IEEE 802.5 token-ring protocol, in which a binary token (carrying a value in $\{0, 1\}$) circulates in a ring. At any time of an execution, there is exactly one process that has the token. When a process executes an action of the form $\mathsf{right}!m$, it sets the token value to $m \in \{0, 1\}$ and passes it to its right neighbor. The latter executes $\mathsf{left}?m$ to receive the token. Since we discard actions of the form $\mathsf{left}!m$ and $\mathsf{right}?m$, we actually deal with a unidirectional ring.

In our protocol, a process of type p emits a message, which will circulate on the given ring until it is received. The fact that the message is currently in transit is indicated by token value 1 (the concrete message contents is abstracted away). Processes of type q will just pass on the token without changing its value. When the token reaches a process of type $\bar{p}$, the message is received. The receiving process sets the token to 0 and passes it to its right neighbor. From there, it is again forwarded by processes of type q until it reaches the "initial" process, which thus gets the confirmation that its message has been received.

Our protocol is modeled by the ring PCA $\mathcal{A} = (\mathcal{P}, Msg, (\mathcal{A}_p, \mathcal{A}_{\bar{p}}, \mathcal{A}_q))$, over the set of interfaces $\mathcal{N} = \{\mathsf{left}, \mathsf{right}\}$, where $\mathcal{P} = \{p, \bar{p}, q\}$, $Msg = \{0, 1\}$, and the local languages are given as follows:

- $L(\mathcal{A}_p) = \{(\mathsf{right}!1)(\mathsf{left}?0)\}$
- $L(\mathcal{A}_{\bar{p}}) = \{(\mathsf{left}?1)(\mathsf{right}!0)\}$
- $L(\mathcal{A}_q) = \{(\mathsf{left}?1)(\mathsf{right}!1), (\mathsf{left}?0)(\mathsf{right}!0)\}$

Note that $L(\mathcal{A}, \mathcal{T}) = \emptyset$ for all $\mathcal{T} \in L_{ring}(q^*)$. Even though two successive processes qq match locally, in the sense that the letter $\mathsf{right}!m$ in the execution of

Pipeline-Nonemptiness(t)	
I:	pipeline PCA $\mathcal{A} = (\mathcal{P}, Msg, (\mathcal{A}_p)_{p\in\mathcal{P}})$ $k \geq 1$; finite automaton $\mathcal{B}$ over $\mathcal{P}$
Q:	$L_{(k,t)}(\mathcal{A}, \mathcal{T}) \neq \emptyset$ for some $\mathcal{T} \in L_{pipe}(\mathcal{B})$?

Ring-Nonemptiness(t)	
I:	ring PCA $\mathcal{A} = (\mathcal{P}, Msg, (\mathcal{A}_p)_{p\in\mathcal{P}})$ $k \geq 1$; finite automaton $\mathcal{B}$ over $\mathcal{P}$
Q:	$L_{(k,t)}(\mathcal{A}, \mathcal{T}) \neq \emptyset$ for some $\mathcal{T} \in L_{ring}(\mathcal{B})$?

Tree$_r$-Nonemptiness(t)	
I:	r-tree PCA $\mathcal{A} = (\mathcal{P}, Msg, (\mathcal{A}_p)_{p\in\mathcal{P}})$ $k \geq 1$; r-tree automaton $\mathcal{B}$ over $\mathcal{P}$
Q:	$L_{(k,t)}(\mathcal{A}, \mathcal{T}) \neq \emptyset$ for some $\mathcal{T} \in L_{tree}(\mathcal{B})$?

	s⊕r1	intf
pipelines	PSPACE-c	PSPACE-c
rings	PSPACE-c	PSPACE-c
trees	EXPTIME-c	EXPTIME-c

Table 1. Context-bounded nonemptiness problems and summary of results

the first q matches the letter left?m in the second occurrence of q, closing a sequence q^n towards a ring is not possible due to the causal dependencies that are created. The receive that remains open on the first q is always scheduled before the remaining open send in the last q. Thus, matching both will create a cyclic dependency and not lead to a valid run of $\mathcal{A}$. We actually have, for all rings $\mathcal{T}$ over $\mathcal{P}$, that $L(\mathcal{A}, \mathcal{T}) \neq \emptyset$ iff $\mathcal{T} \in L_{ring}((pq^*\bar{p}q^*)^*)$. Detecting cyclic dependencies will be one challenge when we tackle the verification problem for rings.

As we aim at modeling a token-ring protocol, we shall only consider rings that contain exactly one process of type p (only one process can have the token). In our decision problems, the input will contain a finite (tree, respectively) automaton that may serve as a corresponding filter. ◊

Note that reachability in token-ring protocols is undecidable when the token is binary [11]. Our approach to get decidability is orthogonal to that from [3,11]. Though the latter assume that a process knows whether it has the token or not, the token itself is unary and does not carry extra information. In our setting, simulating a unary token corresponds to letting *Msg* be a singleton set. In this paper, we do not restrict the amount of (finite) information that a token can carry (i.e., *Msg* can be an arbitrary nonempty finite set), but the local process behavior. This allows us to verify protocols like in Example 1.

Context-Bounded Parameterized Nonemptiness. Next, we define several natural variants of contexts, which restrict the behavior of each process of a PCA. A word $w \in \Sigma^*$ is called an

- (s⊕r)-context if $w \in \Sigma_!^* \cup \Sigma_?^*$,
- (s1+r1)-context if $w \in (\Sigma_{a!} \cup \Sigma_{b?})^*$ for some $a, b \in \mathcal{N}$,
- (s⊕r1)-context if $w \in \Sigma_!^* \cup \Sigma_{a?}^*$ for some $a \in \mathcal{N}$, and
- intf-context if $w \in (\Sigma_{a!} \cup \Sigma_{a?})^*$ for some $a \in \mathcal{N}$.

The case s1⊕r ($w \in \Sigma_{a!}^* \cup \Sigma_?^*$ for some $a \in \mathcal{N}$) is symmetric to s⊕r1, and we only consider the latter. All results hold verbatim when we replace s⊕r1 with s1⊕r.

Let $k \geq 1$ and $t \in \{\text{s⊕r}, \text{s1+r1}, \text{s⊕r1}, \text{intf}\}$ be a context type. We say that $w \in \Sigma^*$ is (k,t)-bounded if there are $w_1, \ldots, w_k \in \Sigma^*$ such that $w = w_1 \cdot \ldots \cdot w_k$

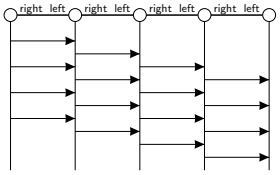

Fig. 4. Undecidability for s1+r1

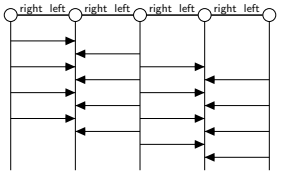

Fig. 5. Undecidability for s⊕r

and w_i is a t-context, for all $i \in [k]$. The set of all (k,t)-bounded words (over a fixed Σ) is denoted by $\mathbb{W}_{(k,t)}$. For a PCA $\mathcal{A} = (\mathcal{P}, Msg, (\mathcal{A}_p)_{p\in\mathcal{P}})$ and a topology $\mathcal{T} = (V, \nu, \pi)$, we define $L_{(k,t)}(\mathcal{A}, \mathcal{T}) := \{W \in L(\mathcal{A}, \mathcal{T}) \mid W|_u \in \mathbb{W}_{(k,t)}$ for all $u \in V\}$. Note that $\mathbb{W}_{(k,t)}$ is a regular word language that is recognized by a finite automaton $\mathcal{B}_{(k,t)}$ whose number of states is linear in k and at most quadratic in $|\mathcal{N}|$ (but linear for the decidable cases of t). Let $\mathcal{A}'$ be the PCA $(\mathcal{P}, Msg, (\mathcal{A}_p \times \mathcal{B}_{(k,t)})_{p\in\mathcal{P}})$ where $\mathcal{A}_p \times \mathcal{B}_{(k,t)}$ is the classical product of two finite automata. It is easy to see that $L_{(k,t)}(\mathcal{A}, \mathcal{T}) = L(\mathcal{A}', \mathcal{T})$. This means that the context-bound restriction can be built into the PCA.

Applying the definitions to the PCA $\mathcal{A}$ from Example 1, we have $L(\mathcal{A}, \mathcal{T}) = L_{(2,\mathsf{s}\oplus\mathsf{r}1)}(\mathcal{A}, \mathcal{T}) = L_{(2,\mathsf{intf})}(\mathcal{A}, \mathcal{T})$ for all topologies over $\{\mathsf{left}, \mathsf{right}\}$ and $\{p, \bar{p}, q\}$.

Note that many distributed algorithms use a bounded number of contexts (or even a bounded number of actions) per process. Prominent examples are some leader-election protocols and P2P protocols. Even when the number of contexts is unbounded, there is often an exponential trade-off between the number of contexts and the (larger) number of processes (e.g., for leader election). Thus, context-bounded verification may sometimes be more appropriate than cut-off techniques, which bound the number of processes.

For $t \in \{\mathsf{s}\oplus\mathsf{r}, \mathsf{s}1{+}\mathsf{r}1, \mathsf{s}\oplus\mathsf{r}1, \mathsf{intf}\}$, we consider the problems listed in Table 1. Note that the context bound k is part of the input. We assume that k is encoded in unary. Table 1 also contains a summary of the positive results of the paper. For some context types, however, all problems are undecidable.

Theorem 1. *All problems listed in Table 1 are undecidable for $t \in \{\mathsf{s}\oplus\mathsf{r}, \mathsf{s}1{+}\mathsf{r}1\}$, even when we restrict to one context for each process.*

Proof (sketch). Figures 4 and 5 demonstrate how to generate grid-like structures of arbitrary height i and width j, using only one context on each single process. Figure 4, for example, visualizes an execution of the form

$$(\langle 1, m_{(1,1)}, 2\rangle\langle 2, m_{(1,2)}, 3\rangle \dots \langle j, m_{(1,j)}, j+1\rangle) \dots (\langle 1, m_{(i,1)}, 2\rangle\langle 2, m_{(i,2)}, 3\rangle \dots \langle j, m_{(i,j)}, j+1\rangle)\,.$$

The idea is now to simulate a Turing machine, using the (unbounded) vertical dimension to encode its tape, which changes along the (unbounded) horizontal line. More precisely, the leftmost process generates a sequence of messages $(m_{(1,1)}, \dots, m_{(i,1)})$ that corresponds to the initial configuration with arbitrarily many cells. Each further process may locally change that configuration while passing it to its right neighbor, and so on. In the case of $\mathsf{s}\oplus\mathsf{r}$, the transfer of

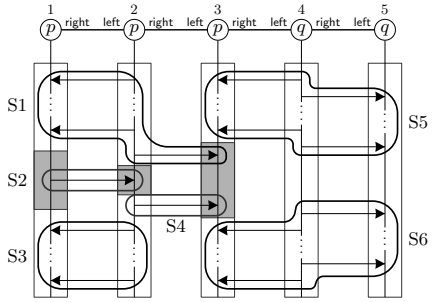

Fig. 6. Cell transitions wrt. s⊕r1

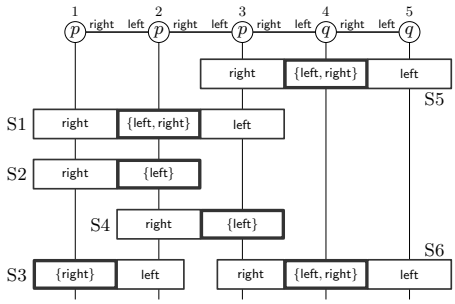

Fig. 7. Run of finite automaton

a configuration is sometimes accomplished by a receive context. Obviously, the encoding also works for rings and for trees. □

4 Context-Bounded Parameterized Verification

We now present our main results: decidability of all our context-bounded parameterized verification problems, as far as context types s⊕r1 and intf are concerned.

Theorem 2. *For all* $t \in \{\mathsf{s}\oplus\mathsf{r1}, \mathsf{intf}\}$*, the following hold:*

- Pipeline-Nonemptiness(t) *is* PSPACE*-complete,*
- Ring-Nonemptiness(t) *is* PSPACE*-complete, and*
- Tree$_r$-Nonemptiness(t) *is* EXPTIME*-complete, for all* $r \geq 2$*.*

In the remainder of this section, we develop the main proof ideas.

General Proof Idea for Upper Bounds. We illustrate the proof by means of pipelines and context type s⊕r1, which is slightly more difficult than the case of intf. Given a pipeline PCA $\mathcal{A} = (\mathcal{P}, Msg, (\mathcal{A}_p)_{p\in\mathcal{P}})$ and $k \geq 1$, we will construct a finite automaton $\mathcal{B}_\mathcal{A}$ that recognizes exactly those pipelines $\mathcal{T}$ such that $L_{(k,\mathsf{s}\oplus\mathsf{r1})}(\mathcal{A}, \mathcal{T}) \neq \emptyset$. While reading a pipeline (i.e., a word over $\mathcal{P}$), the finite automaton will guess an accepting run of $\mathcal{A}$. When every local language $L(\mathcal{A}_p)$ is finite, this can be done as follows: A state of $\mathcal{B}_\mathcal{A}$ is a string from some local language. When reading p, the automaton guesses an element of $L(\mathcal{A}_p)$ and checks if its projection to left-actions matches the current state. A state is final if it does not communicate through interface right. However, though we can restrict to $(k, \mathsf{s}\oplus\mathsf{r1})$-bounded words, the local language $L(\mathcal{A}_p)$ of a process type p is in general infinite so that the naive construction is not applicable.

The trick is to find a bounded abstraction of the infinitely many (local) runs. This is illustrated in Figure 6, which depicts a $(3, \mathsf{s}\oplus\mathsf{r1})$-bounded execution (in fact, a set of "order"-equivalent executions). Processes 1, 2, and 3 use three contexts, while 4 and 5 can do with a single one. The dotted areas on a process line suggest that we actually consider an arbitrary number of actions. Our aim is

to aggregate these unboundedly many actions in a bounded number of summaries Si so that a finite automaton can read the pipeline (i.e., the word $pppqq$) from left to right, while verifying that the summaries can be glued together towards an accepting run of the given PCA.

As process 4 alternates between sending to 3 and sending to 5, its summaries have to include the behavior of processes 3 and 5. A summary is then given by a *cell transition* of the form $\mathrm{s} \xrightarrow{pqq} \mathrm{s}'$. Here, *cell* refers to pqq, which represents an isomorphism type of a pipeline of length three. Moreover, $\mathrm{s}, \mathrm{s}' \in S_p \times S_q \times S_q$ denote how states evolve in that particular fragment within a bigger pipeline, for example when executing all actions gathered in S5. Cells have bounded size so that the set of cell transitions can be effectively computed and represented.

Now, the behavior of process 4 can only be captured when we use at least two cell transitions (for S5 and S6). The reason is that receives of process 3 from 4 are interrupted by receives from process 2. Similarly, the receive context in the middle of process 3 will belong to two different summaries, as it is interrupted by a context switch on process 2. The splitting is not unique, as we could have merged S3 and S4. However, the total number of splits can be bounded: a send (receive) context is split whenever the complementary receives (sends, respectively) belong to distinct contexts. Thus, it is divided into at most $k \cdot |\mathcal{N}|$ summaries. Using this, one can show that any $(k, \mathsf{s}{\oplus}\mathsf{r1})$-bounded execution of a PCA is captured by a sequence of cell transitions such that each process is involved at most $k \cdot (|\mathcal{N}|^2 + 2|\mathcal{N}| + 1)$ times. This gives us a bounded abstraction of a priori unbounded behaviors so that we can build a finite automaton that guesses such an abstraction and, simultaneously, checks if it corresponds to an accepting run of the PCA. A run of the finite automaton is depicted in Figure 7 (where we omit local states). On a process, we only keep "blocks" indicating both the interfaces that are employed and whether we deal with a sending phase (*set* of interfaces) or a receiving phase (*single* interface).

Note that the size of $\mathcal{B}_\mathcal{A}$ is exponential in k. However, nonemptiness can be checked "on-the-fly", which takes only polynomial space. The construction works similarly for trees; we then come up with a tree automaton, which gives us an EXPTIME procedure. However, the idea is not directly applicable to rings. Consider the PCA from Example 1. Figure 8 illustrates a possible run of the finite automaton $\mathcal{B}_\mathcal{A}$ over $qqqq$. Since the final state and the state taken after reading the first position match locally, we are tempted to say that $\mathcal{B}_\mathcal{A}$ should accept the *ring* $\mathcal{T}$ induced by $qqqq$. However, we have $L(\mathcal{A}, \mathcal{T}) = \emptyset$. The trick is now to retrieve cyclic dependencies that violate the run conditions of PCAs. In the example, we have to record that the gray-shaded left-block (which arose from a receive action) is scheduled *before* the gray-shaded {right}-block (which arose from a send action). Those blocks cannot be matched, i.e., the run of the finite automaton $\mathcal{B}_\mathcal{A}$ does not reflect a run of $\mathcal{A}$. We will, therefore, enrich the previous construction to obtain a decision procedure for rings.

Dependence Graphs. The idea is to add *dependence graphs*, which keep track of the causal dependencies between cell transitions. They arise naturally when we combine the behavior of two processes in terms of states of $\mathcal{B}_\mathcal{A}$. For pro-

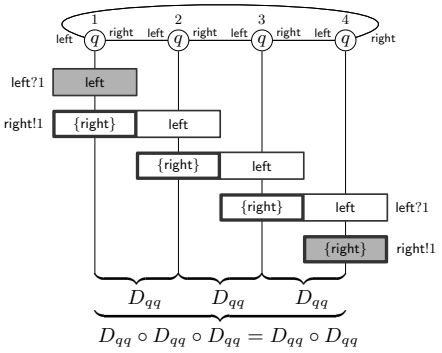

Fig. 8. Finite automaton on a ring

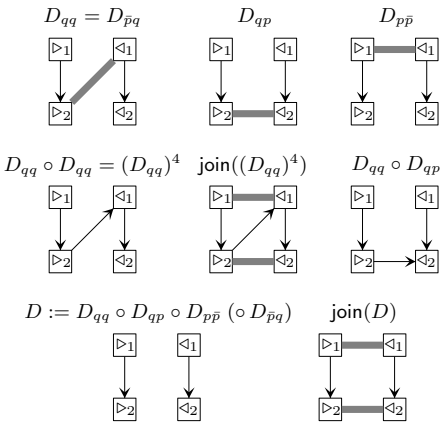

Fig. 9. Dependence graphs

cesses 1 and 2 in Figure 8, we obtain the dependence graph D_{qq} depicted in Figure 9. There are two kinds of constraints, an undirected (i.e., symmetric) one representing synchronizations (the thick gray lines), and a directed one for strict causality (depicted by arrows $\rightarrow$). In D_{qq}, for example, the nodes $\triangleright_1$ and $\triangleright_2$ on the left represent the two strictly causally ordered blocks of the first process, while the nodes $\triangleleft_1$ and $\triangleleft_2$ on the right represent the two blocks of the second process. Moreover, $\triangleright_2$ and $\triangleleft_1$ are synchronized, i.e., they happen instantaneously.

The effect of appending a further process of type q can be computed as a composition $D_{qq} \circ D_{qq}$, which we obtain as follows:

1. Merge every node $\triangleleft_i$ of the first graph with the corresponding node $\triangleright_i$ of the second graph.
2. A path containing at least one $\rightarrow$-constraint and synchronization constraints in either direction becomes a new $\rightarrow$-constraint.
3. The new synchronization constraints are given by the transitive closure of the (union of the) old ones.
4. Remove the merge nodes.

Note that, in the figure, we represent the composition by a minimal set of constraints.

Now, "closing" the pipeline $qqqq$ towards a ring corresponds to joining the left and right hand side of $(D_{qq})^4 = (D_{qq})^2$. Technically, we add synchronization constraints between $\triangleright_i$ and $\triangleleft_i$. The result is depicted as $\mathsf{join}((D_{qq})^4)$ in Figure 9. However, the join contains a cycle using at least one constraint of type $\rightarrow$ (recall that synchronization edges can be taken in either direction), which has to be interpreted as a violation of the run condition of PCAs.

Consider, on the other hand, the "pipeline" $qqp\bar{p}$. It induces the graph $D_{qq} \circ D_{qp} \circ D_{p\bar{p}}$ (depicted at the bottom left of the figure), which resolves any dependency between the leftmost and the rightmost process. To check whether

the pipeline can be closed towards a ring, we apply the join operation to $D_{qq} \circ D_{qp} \circ D_{p\bar{p}} \circ D_{\bar{p}q}$. The result is depicted at the bottom right of Figure 9. The join is harmless, since it does not create any cycle containing at least one $\rightarrow$-edge. Thus, the ring given by $qqp\bar{p}$ allows for an accepting run of the given PCA.

Note that, in the ring case (and for context type $\mathsf{s}{\oplus}\mathsf{r1}$), summaries are defined in a slightly different way to make sure that dependencies are reflected correctly. A summary then either involves only two processes, or it has at least two alternations between sending to the left and sending to the right. This guarantees that the induced synchronization constraints in dependence graphs are indeed symmetric. The new definition of summaries results in a linear blow up of the number of blocks on each process.

Lower Bounds. To illustrate the lower-bound proofs, we consider trees. For $t \in \{\mathsf{s}{\oplus}\mathsf{r1}, \mathsf{intf}\}$ and $r \geq 2$, EXPTIME-hardness of TREE$_r$-NONEMPTINESS(t) is established by a reduction from the intersection problem for binary-tree automata, which is EXPTIME-complete [20] (similarly, the lower bounds for pipelines and rings use the intersection problem for finite automata). Without loss of generality, we assume here that (1) tree automata accept only trees where the root and every internal node have exactly two children and (2) the node labeling tells us whether we deal with the root, a leaf, or an internal node. Given $k \geq 1$ and binary-tree automata $\mathcal{B}_1, \ldots, \mathcal{B}_k$, we can construct, in polynomial time, a PCA $\mathcal{A}$ such that, for all 2-tree topologies $\mathcal{T}$, we have $L_{(2k,\mathsf{s}\oplus\mathsf{r1})}(\mathcal{A}, \mathcal{T}) \neq \emptyset$ iff $L_{(3k,\mathsf{intf})}(\mathcal{A}, \mathcal{T}) \neq \emptyset$ iff $\mathcal{T} \in L_{tree}(\mathcal{B}_1) \cap \ldots \cap L_{tree}(\mathcal{B}_k)$. The idea is that each process u with two children chooses transitions $\delta_1, \ldots, \delta_k$ of $\mathcal{B}_1, \ldots, \mathcal{B}_k$, respectively, that are applied at u. These transitions are sent to the children $u.1$ and $u.2$ of u. When $u.1$ (or $u.2$) receives a transition δ_i, it immediately sends a corresponding transition δ_i' to its own children. This is why the PCA works with $2k$ and $3k$ contexts.

5 Conclusion

We showed that verification of PCAs running on pipelines, rings, and trees is decidable under certain context bounds. Using automata complementation, we also obtain decidability of the *universal* variants of our verification problem: Do *all* topologies accepted by a finite (tree) automaton allow for an accepting run of the given PCA?

It would be worthwhile to study if there are other natural, maybe more general classes of graphs that come with a decidable context-bounded nonemptiness problem. Moreover, one may consider model checking against temporal logics, and automata models that run over topologies of unbounded degree such as star topologies and unranked trees. These models may include registers so that a process can remember some of its neighbors [8].

Acknowledgment. We thank the anonymous reviewers for comments that helped to improve the presentation of the paper.

References

1. P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *VMCAI'13*, volume 7737 of *LNCS*, pages 476–495. Springer, 2013.
2. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *LICS'93*, pages 160–170, 1993.
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. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Log. Methods Comput. Sci.*, 7(4), 2011.
5. B. Bollig. Logic for communicating automata with parameterized topology. In *CSL-LICS'14*. ACM, 2014.
6. A. Bouajjani and M. Emmi. Bounded phase analysis of message-passing programs. In *TACAS'12*, volume 7214 of *LNCS*, pages 451–465. Springer, 2012.
7. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2), 1983.
8. G. Delzanno, A. Sangnier, and R. Traverso. Parameterized verification of broadcast networks of register automata. In *RP'13*, volume 8169 of *LNCS*, pages 109–121. Springer, 2013.
9. G. Delzanno, A. Sangnier, and G. Zavattaro. On the power of cliques in the parameterized verification of ad hoc networks. In *FoSSaCS'11*, volume 6604 of *LNCS*, pages 441–455. Springer, 2011.
10. E. A. Emerson and V. Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL'04*, volume 3210 of *LNCS*, pages 325–339, 2004.
11. E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *Int. J. Found. Comput. Sci.*, 14(4):527–550, 2003.
12. J. Esparza. Keeping a crowd safe: On the complexity of parameterized verification. In *STACS'14*, volume 25 of *LIPIcs*, pages 1–10, 2014.
13. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS'99*, pages 352–359. IEEE Computer Society Press, 1999.
14. J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. In *CAV'13*, volume 8044 of *LNCS*, pages 124–140, 2013.
15. A. Heußner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. *Log. Methods Comput. Sci.*, 8(3:23):1–20, 2012.
16. S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS'08*, volume 4963 of *LNCS*, pages 299–314, 2008.
17. S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV'10*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.
18. P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL'11*, pages 283–294. ACM, 2011.
19. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS'05*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
20. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.