

Model-checking parametric lock-sharing systems against regular constraints

Corto Mascle

Université de Bordeaux, France

Anca Muscholl

Université de Bordeaux, France

Igor Walukiewicz

Université de Bordeaux, CNRS, France

Abstract

In parametric lock-sharing systems processes can spawn new processes to run in parallel, and can create new locks. The behavior of every process is given by a pushdown automaton. We consider infinite behaviors of such systems under strong process fairness condition. A result of a potentially infinite execution of a system is a limit configuration, that is a potentially infinite tree. The verification problem is to determine if a given system has a limit configuration satisfying a given regular property. This formulation of the problem encompasses verification of reachability as well as of many liveness properties. We show that this verification problem, while undecidable in general, is decidable for nested lock usage.

We show EXPTIME-completeness of the verification problem. The main source of complexity is the number of parameters in the spawn operation. If the number of parameters is bounded, our algorithm works in PTIME for properties expressed by parity automata with a fixed number of ranks.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Parametric systems Locks Model-checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.24

1 Introduction

Locks are a widely used concurrency primitive. They appear in classical programming languages such as Java, as well as in recent ones such as Rust. The principle of creating shared objects and protecting them by mutexes (like the “synchronized” paradigm in Java) requires *dynamic lock creation*. The challenge is to be able to analyze programs with dynamic creation of threads *and* locks.

Our system model is based on Dynamic Pushdown Networks (DPNs) as introduced in [7], where processes are pushdown systems that can spawn new processes. The DPN model was extended in [20] by adding synchronization through a fixed number of locks. Here we take a step further and allow dynamic lock creation: when spawning a new process, the parent process can pass some of its locks, and new locks can be created for the new thread. This way we model recursive programs with creation of threads and locks. We call such systems *dynamic lock-sharing systems* (DLSS).

The focus in both [7] and [20] is computing the Pre^* of a regular set of configurations, and they achieve this by extending suitably the saturation technique from [6]. Here we consider not only reachability but also infinite behaviors of DLSS under fairness conditions. For this we propose a different approach than saturation from [7, 20] as saturation is not suited to cope with liveness properties.

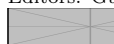
We show that verifying regular properties of DLSS is decidable if every process follows *nested lock usage*. This means that locally every process acquires and releases the locks according to the stack discipline. Nested locking is assumed in most papers on parametric



© Corto Mascle, Anca Muscholl and Igor Walukiewicz;
licensed under Creative Commons License CC-BY 4.0

34th International Conference on Concurrency Theory (CONCUR 2023).

Editors: Guillermo A. Pérez and Jean-François Raskin; Article No. 24; pp. 24:1–24:28



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

verification of systems with synchronization over locks. It is also considered as good programming practice, sometimes even enforced syntactically, as in Java through synchronized blocks.

Without any restriction on lock usage we show that our problem is undecidable, even for finite state processes and reachability properties that refer to a single process. Note that our model does not have global variables. It is well-known that reachability is undecidable already for two pushdown processes with one lock and one global variable.

Outline of the paper. Our starting point is to use trees to represent configurations of DLSS. This representation was introduced in [20]. The advantage is that it does not require to talk about interleavings of local runs of processes. Instead it represents every local run as a left branch in a tree and the spawn operations as branching to the right. At each computation step one or two nodes are added below a leaf of the current configuration. Thus, the result of a run of DLSS is an infinite tree that we call a *limit configuration*. Our first observation is that it is easy to read out from a limit configuration of a run if the run is strongly process-fair (Proposition 3).

An important step is to characterize those trees that are limit configurations of runs of a given *finite state* DLSS, namely where every process is a finite state system. This is done in Lemma 11. To deal with lock creation this lemma refers to the existence of some global acyclic relation on locks. We show that this global relation can be recovered from local orderings in every node of the configuration tree (Lemma 12). Finally, we show that there is a nondeterministic Büchi tree automaton verifying all the conditions of Lemmas 11 and 12. This is the desired tree automaton recognizing limit configurations of process-fair runs. Our verification problem is solved by checking if there is a tree satisfying the specification and accepted by this automaton. This way we obtain the upper bound from Theorem 7. Surprisingly the size of the Büchi automaton is linear in the size of DLSS, and exponential only in the *arity* of the DLSS, which is the maximal number of locks a process can access. For example, in the dining philosophers setting (cf. Figure 1) the arity is 3, as every philosopher has access only to its left and right forks, implemented as locks; and there is one more fork to close the cycle.

The extension of our construction to pushdown processes requires one more idea to get an optimal complexity. In this case, ensuring that the limit tree represents a computation requires using pushdown automata. Hence, the Büchi tree automaton as described in the previous paragraph becomes a pushdown Büchi automaton on trees. The emptiness of pushdown Büchi tree automata is EXPTIME-complete, which is an issue as the automaton constructed is already exponential in the size of the input. However, we observe that the automata we obtain are right-resetting, since new threads are spawned with empty pushdown. Intuitively, the pushdown is needed only on left paths of the configuration tree to check correctness of local runs. A right-resetting automaton resets its stack each time it goes to the right child. We show that the emptiness of right-resetting parity pushdown tree automata can be checked in PTIME if the biggest rank in the parity condition is fixed (if it is not fixed then the problem is at least as complex as solving parity games). This gives the upper bound from Theorem 8.

Finally, we obtain the matching lower bound by proving EXPTIME-hardness of checking if a process of the DLSS has an infinite run (Proposition 22). This holds even for finite state processes. We also show that even for finite state processes the DLSS verification problem is undecidable if we allow arbitrary usage of locks (Theorem 5).

Related work. Parametrized verification has remained an active research area for almost three decades [1, 5, 13]. It has brought a steady stream of works on parametric systems with

locks. As already mentioned, the first directly relevant paper is [7] introducing Dynamic Pushdown Networks (DPNs). These consist of pushdown processes with spawn but no locks. The main idea is to represent a configuration as a sequence of process identifiers, each identifier followed by a stack content. Computing Pre^* of a regular set of configurations is decidable by extending the saturation technique from [6].

An important step is made in [20] where the authors introduce a tree representation of configurations. This is essentially the same representation as we use here. They extend DPNs by a fixed set of locks, and show how to adapt the saturation technique to compute Pre^* in this case. Their result is an EXPTIME decision procedure for verifying reachability of a regular set of configurations. This work has been extended to incorporate join operations [12], or priorities on processes [9]. Our work extends [20] in two directions: it adds lock creation, and considers liveness properties. It is not clear how one could extend saturation methods to deal with liveness properties.

The saturation method has been adapted to DPNs with lock creation in the recent thesis [17]. The approach relies on hyperedge replacement grammars, and gives decidability without complexity bounds. Our liveness conditions can express this kind of reachability conditions.

Actually, the first related paper to deal with lock creation is probably [25]. The authors consider a model of higher-order programs with spawn, joins, and lock creation. Apart from nested locking, a new restriction of scope safety is imposed. Under these conditions, reachability of pairs of states is shown to be decidable.

The works above have been followed by implementations [9, 18, 25]. In particular [9] reports on verification of several substantial size programs and detecting an error in xvisor [8].

In all the works above nested locking is assumed. In [16] the interest of nested locking is underlined by showing that reachability with two pushdown processes using locks is undecidable in general, but it is decidable for nested locking. There are only few related works without this assumption. The work [15] generalizes nested locking to bounded lock-chain condition, and shows decidability of reachability for two pushdown processes. In [19] the authors consider contextual locking where arbitrary locking may occur as long as it does not cross procedure boundaries. This condition is incomparable with nested locking.

Finally, we comment on shared state and global variables. These are not present in the above models because reachability for two pushdown processes with one lock and one global variable is already undecidable. There is an active line of study of multi-pushdown systems where shared state is modeled as global control. In this model decidability is recovered by imposing restrictions on stack usage such as bounded context switching and variations thereof [2, 22–24]. Observe that these are restrictions on global runs, and not on local runs of processes, as we consider here. Another approach to recover decidability is to have shared state but no locks [10, 11, 14, 21]. Finally, there is a very interesting model of threaded pools [3, 4], without locks, where verification is decidable once again assuming bounded context switching. But the complexity of this model is as high as Petri net coverability [4].

Structure of the paper. The next section presents the main definitions and results. The main proof for finite state processes is outlined in Sections 3 and 4. Section 5 describes the extension to pushdown processes. All missing proofs are included in the appendix.

2 Definitions and results

A dynamic lock-sharing system is a set of processes, each process has access to a set of locks and can spawn other processes. A spawned process can inherit some locks of the spawning

$$\begin{aligned}
p_{init} &: \text{spawn}(first, \text{new}, \text{new}) \\
first(x_l, x_r) &: \text{spawn}(phil, x_l, x_r); \text{spawn}(next, x_r, \text{new}, x_l) \\
next(x_l, x_r, x_{lfirst}) &: \text{or} \begin{cases} \text{spawn}(phil, x_l, x_{lfirst}) \\ \text{spawn}(phil, x_l, x_r); \text{spawn}(next, x_r, \text{new}, x_{lfirst}) \end{cases} \\
phil(x_l, x_r) &: \text{repeat-forever or} \begin{cases} \text{get}_{x_l}; \text{get}_{x_r}; \text{eat}; \text{rel}_{x_r}; \text{rel}_{x_l} \\ \text{get}_{x_r}; \text{get}_{x_l}; \text{eat}; \text{rel}_{x_l}; \text{rel}_{x_r} \end{cases}
\end{aligned}$$

■ **Figure 1** Dining philosophers: process *first* starts the first philosopher and an iterator process *next* starts successive philosophers. The forks, modeled as locks, are passed via variables x_l and x_r . The third variable x_{lfirst} of *next* is the left fork of the first philosopher used also by the last philosopher. The system is nested as *phil* takes and releases forks in the stack order. The arity of the system is 3.

process and can also create new locks. All processes run in parallel. A run of the system must be fair, meaning that if a process can move infinitely many times then it eventually does.

More formally, we start with a finite set of process identifiers *Proc*. Each process identifier $p \in Proc$ has an arity $ar(p) \in \mathbb{N}$ telling how many locks the process uses. The process can refer to these locks through the variables $Var(p) = \{x_1^p, \dots, x_{ar(p)}^p\}$. At each step a process can do one of the following operations:

$$\begin{aligned}
Op(p) = & \{\text{nop}\} \cup \{\text{get}_x, \text{rel}_x \mid x \in Var(p)\} \\
& \cup \{\text{spawn}(q, \sigma) \mid q \in Proc, \sigma : Var(q) \rightarrow (Var(p) \cup \{\text{new}\})\}
\end{aligned}$$

Operation **nop** does nothing. Operation **get_x** acquires the lock designated by x , while **rel_x** releases it. Operation **spawn(q, σ)** spawns an instance of process q where every variable of q designates a lock determined by the substitution σ ; this can be a lock of the spawning process or a new lock, if $\sigma(x^q) = \text{new}$. We require that the mapping σ is *injective* on $Var(p)$. This is important for the definition of nested stack usage.

□ A *dynamic lock-sharing system* (DLSS for short) is a tuple

$$\mathcal{S} = (Proc, ar, (\mathcal{A}_p)_{p \in Proc}, p_{init}, Locks)$$

where *Proc*, and *ar* are as described above. For every process p , \mathcal{A}_p is a transition system describing the behavior of p . Process $p_{init} \in Proc$ is the initial process. Finally, *Locks* is an infinite pool of locks.

Each transition system \mathcal{A}_p is a tuple $(S_p, \Sigma_p, \delta_p, op_p, init_p)$ with S_p a finite set of states, $init_p$ the initial state, Σ_p a finite alphabet, $\delta_p : S_p \times \Sigma_p \rightarrow S_p$ a *partial* transition function, and $op_p : \Sigma_p \rightarrow Op(p)$ an assignment of an operation to each action. We require that the Σ_p are pairwise disjoint, and define $\Sigma = \bigcup_{p \in Proc} \Sigma_p$. We write $op(b)$ instead of $op_p(b)$ for $b \in \Sigma_p$, as b determines the process p .

For simplicity, we require that p_{init} is of arity 0. In particular, process p_{init} has no **get** or **rel** operations.

An example in Figure 1 presents a DLSS modeling an arbitrary number of dining philosophers. The system can generate a ring of arbitrarily many philosophers, but can also generate infinitely many philosophers without ever closing the ring.

A configuration of \mathcal{S} is a tree representing the runs of all active processes. The leftmost branch represents the run of the initial process p_{init} , the left branches of nodes to the right of the leftmost branch represent runs of processes spawned by p_{init} etc. So a leaf of a configuration represents the current situation of a process that is started at the first ancestor above the leaf that is a right child. A node of a configuration is associated with a process, and tells in what state the process is, which locks are available to it, and which of them it holds.

More formally, a *configuration* is a, possibly infinite, tree $\tau \subseteq \{0, 1\}^*$, with each node ν labeled by a tuple (p, s, a, L, H) where $p \in Proc$ is the process executing in ν , $s \in \Sigma_p$ the state of p , $a \in \Sigma_p$ the action p executed at ν , or $\perp \notin \Sigma$ if ν is a root, $L : Var(p) \rightarrow Locks$ an assignment of locks to variables of p , and $H \subseteq L(Var(p))$ the set of locks p holds before executing a . We use $p(\nu)$, $s(\nu)$, $a(\nu)$, $L(\nu)$ and $H(\nu)$ to address the components of the label of ν . For ease of notation we will write $Var(\nu)$ instead of $Var(p(\nu))$.

We write $H(\tau)$ for the set of locks *ultimately held* by some process in τ , that is, $H(\tau) = \{\ell : \text{for some } \nu, \ell \in H(\nu') \text{ for all } \nu' \text{ on the leftmost path from } \nu\}$. If τ is finite this is the same as to say that $H(\tau)$ is the union of $H(\nu)$ over all leaves ν of τ .

The initial configuration is the tree τ_{init} consisting only of the root ε labeled by $(p_{init}, init_p, \perp, \emptyset, \emptyset)$. Suppose that ν is a leaf of τ labeled by (p, s, b, L, H) , and there is a transition $s \xrightarrow{a} s'$ for some s' in \mathcal{A}_p . A transition between two configurations $\tau \xrightarrow{\nu, a} \tau'$ is defined by the following rules.

- If $op(a) = \text{spawn}(q, \sigma)$ then τ' is obtained from τ by adding two children $\nu 0, \nu 1$ of ν . The label of the left child $\nu 0$ is (p, s', a, L, H) . The label of the right child $\nu 1$ is $(q, init_q, \perp, L', \emptyset)$ where $L'(x^q) = L(\sigma(x^q))$ if $\sigma(x^q) \neq \text{new}$ and $L'(x^q) = \ell_{\nu, x^q}$ is a fresh lock, otherwise.
- Otherwise, τ' is obtained from τ by adding a left child $\nu 0$ to ν . The label of $\nu 0$ must be of the form (p, s', a, L, H') subject to the following constraints:
 - If $op(a) = \text{nop}$ then $H' = H$,
 - If $op(a) = \text{get}_x$ and $L(x) \notin H(\tau)$ then $H' = H \cup \{L(x)\}$,
 - If $op(a) = \text{rel}_x$ and $L(x) \in H$ then $H' = H \setminus \{L(x)\}$.

Note that we do not allow a process to acquire a lock it already holds, or release a lock it does not have. We call this property *soundness*.

A *run* is a (finite or infinite) sequence of configurations $\tau_0 \xrightarrow{\nu_1, a_1} \tau_1 \xrightarrow{\nu_2, a_2} \dots$. As the trees in a run are growing we can define the *limit configuration* of that run as its last configuration if it is finite, and as the limit of its configurations if it is infinite.

► **Remark 1.** Note that in a run, at every moment distinct variables of a process are associated with distinct locks: $L(\nu_i)(x) \neq L(\nu_i)(y)$ for all $x, y \in Var(\nu)$ with $x \neq y$.

► **Remark 2.** The labels L and H can be computed out of the other three labels in the tree just following the transition rules. We could have defined configurations as trees with only three labels (p, s, a) , but we preferred to include L and H for readability. Yet, later we will work with tree automata recognizing configurations and there it will be important that the labels come from a finite set.

A configuration τ is *fair* if for no leaf ν there is a transition $\tau \xrightarrow{\nu, a} \tau'$ for some a and τ' . We show that this compact definition of fairness captures strong process fairness of runs. Recall that a run is *strongly process-fair* if whenever from some position in the run a process is enabled infinitely often then it moves after this position.

► **Proposition 3.** Consider a run $\tau_0 \xrightarrow{\nu_1, a_1} \tau_1 \xrightarrow{\nu_2, a_2} \dots$ and its limit configuration τ . The run is strongly process-fair if and only if τ is fair.

Objectives. Instead of using some specific temporal logic we stick to a most general specification formalism and use regular tree properties for specifications. A *regular objective* is given by a nondeterministic tree automaton \mathcal{B} over $\Sigma \cup \{\perp\}$, which defines a language of accepted *limit configurations*. The trees we work with can have nodes of rank 0, 1, or 2. So we suppose that the alphabet is partitioned into Σ_0 , Σ_1 and Σ_2 . The nondeterministic transition function reflects this with $\delta(q, a) \subseteq \{\top\}$ if $a \in \Sigma_0$, $\delta(q, a) \subseteq Q$ if $a \in \Sigma_1$, and $\delta(q, a) \subseteq Q \times Q$ if $a \in \Sigma_2$. A run of the automaton on a tree t is a labeling of t with states respecting δ . In particular if ν is a leaf of t then $\top \in \delta(q, a)$, where q is the state and a is the letter in ν . A run is accepting if for every infinite path the sequence of states on this path is in the accepting set of the automaton. We will work with accepting sets given by parity conditions. We say that a *configuration* τ satisfies \mathcal{B} when \mathcal{B} accepts the tree obtained from τ by restricting only to action labels.

Regular objectives can express many interesting properties. For example, “for every instance of process p its run is in a regular language \mathcal{C} ”. Or more complicated “there is an instance of p with a run in a regular language \mathcal{C}_1 and all the instances of p have runs in the language \mathcal{C}_2 ”. Of course, it is also possible to talk about boolean combinations of such properties for different processes. Observe that the resulting automaton \mathcal{B} for these kinds of properties can be a parity automaton with ranks 1, 2, 3 (properties of sequences can be expressed by Büchi automata, and rank 3 is used to implement existential quantification on process instances).

Regular objectives can express deadlock properties. Since we only consider *process-fair runs*, a finite branch in a *limit configuration* indicates that a process is blocked forever after some point. Hence, we can express properties such as “there is an instance of p that is blocked forever after a finite run in a regular language \mathcal{C} ”. We can also express that all branches are finite, which is equivalent to a global deadlock since we are considering only *process-fair runs*.

Reachability properties are also expressible with regular objectives. We can check simultaneous reachability of several states in different branches, for instance “there is a reachable configuration in which some process p reaches s while some process p' reaches s' ”. There are ways to do it directly, but the shortest argument is through a small modification of the DLSS. We can simply add transitions to stop processes non-deterministically in desired states: adding new *nop* transitions from s and s' to new deadlock states. Using ideas from [19] we can also check reachability of a regular set of configurations.

Going back to our dining philosophers example from Figure 1, we can see also other types of properties we would like to express. For example, we would like to say that there are finitely many philosophers in the system. This can be done simply by saying that there are not infinitely many spawns in the limit configuration. (In this example it is equivalent to saying that there is no branch turning infinitely often to the right.) Then we can verify a property like “if there are finitely many processes in the system and some philosopher eats infinitely often then all philosophers eat infinitely often”. This property holds under process-fairness, as philosophers release both their forks after eating.

► **Definition 4 (DLSS verification problem).** *Given a DLSS \mathcal{S} and a regular objective \mathcal{B} decide if there is a process-fair run of \mathcal{S} whose limit configuration τ satisfies \mathcal{B} .*

Without any further restrictions we show that our problem is undecidable:

► **Theorem 5.** *The DLSS verification problem is undecidable. The result holds even if the DLSS is finite-state and every process uses at most 4 locks.*

This result is obtained by creating an unbounded chain of processes simulating a Turing machine. Each process memorizes the content of a position on the tape, and communicates with its neighbours by interleaving lock acquisitions. The trick for processes to exchange information by interleaving lock acquisitions was already used in [16], and requires a non-nested usage of locks.

The situation improves significantly if we assume nested usage of locks.

► **Definition 6.** A process \mathcal{A}_p is **nested** if it takes and releases locks according to a stack discipline, i.e., for all $x, y \in \text{Var}(p)$, for all paths $s_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} s_n$ in \mathcal{A}_p , with $op(a_1) = \text{get}_x$, $op(a_n) = \text{rel}_x$, $op(a_m) \neq \text{rel}_x$ for all $m < n$: if $op(a_i) = \text{get}_y$ for some $i < n$ then there exists $i < k < n$ such that $op(a_k) = \text{rel}_y$. A DLSS is nested if all its processes are nested.

We can state the first main result of the paper. Its proof is outlined in the next two sections.

► **Theorem 7.** The DLSS verification problem for nested DLSS is EXPTIME-complete. It is in PTIME when the number of priorities in the specification automaton, and the maximal arity of processes are fixed.

We can extend this result to DLSS where transition systems of each process are given by a pushdown automaton (see definitions in Section 5). The complexity remains the same as for finite state processes.

► **Theorem 8.** The DLSS verification problem for nested pushdown DLSS is EXPTIME-complete. It is in PTIME when the number of priorities in the specification automaton, and the maximal arity of processes is fixed.

3 Characterizing limit configurations

A configuration is a labeled tree. We give a characterization of such trees that are limit configurations of a process-fair run of a given DLSS. In the following section we will show that the set of limit configurations of a given DLSS is a regular tree language, which will imply the decidability of our verification problem.

► **Definition 9.** Given a configuration τ with nodes ν, ν' and variables $x \in \text{Var}(\nu)$, $x' \in \text{Var}(\nu')$, we write $x \sim x'$ if $L(\nu)(x) = L(\nu')(x')$, so if x and x' are mapped to the same lock. The **scope** of a lock ℓ is the set $\{\nu : \ell \in L(\nu)(\text{Var}(\nu))\}$.

► **Remark 10.** It is easy to see that in any configuration, the scope of a lock is a subtree.

We say that a node ν is labeled by an **unmatched get** if it is labeled by some get_x and there is no rel_x operation in the leftmost path starting from ν . Recall that $H(\tau)$ is the set of locks ℓ for which there is some node ν with an **unmatched get** _{x} and $L(\nu)(x) = \ell$.

We define a relation \prec_H on $H(\tau)$ by letting $\ell \prec_H \ell'$ if there exist two nodes ν, ν' such that ν is an ancestor of ν' , ν is labeled with an **unmatched get** of ℓ , and ν' is labeled with a **get** of ℓ' .

After these preparations we can state a central lemma giving a structural characterization of limit configurations of process-fair runs.

► **Lemma 11.** A tree τ is the limit configuration of a process-fair run of a nested DLSS S if and only if

F1 The node labels in τ match the local transitions of S .

- F2** For every leaf ν every possible transition from $s(\nu)$ has operation get_x for some x with $L(\nu)(x) \in H(\tau)$.
- F3** For every lock $\ell \in H(\tau)$ there are finitely many nodes with operations on ℓ , and there is a unique node labeled with an unmatched get of ℓ .
- F4** The relation \prec_H is acyclic.
- F5** The relation \prec_H has no infinite descending chain.

Before presenting the proof of the previous lemma note that the main difficulty is the fact that some locks can be taken and never released. If $H(\tau) = \emptyset$ then from τ we can easily construct a run with limit configuration τ by exploiting the nested lock usage. This is because any local run can be executed from a configuration where all locks are available.

Proof. We start with the left-to-right implication. Suppose that we have a process-fair run $\tau_0 \xrightarrow{\nu_1, a_1} \tau_1 \xrightarrow{\nu_2, a_2} \dots$ with limit configuration τ .

With every lock $\ell \in H(\tau)$ we associate the maximal position $m = m_\ell$ such that $op(a_m) = \text{get}_x$ and $L(\nu_m)(x) = \ell$, so the position m_ℓ where ℓ is acquired for the last time (and never released after).

It remains to check the conditions of the lemma. The first one holds by definition of a run. The second condition is due to process fairness and soundness, since a process can always execute transitions other than acquiring a lock, and locks not in $H(\tau)$ are free infinitely often. All actions involving $\ell \in H(\tau)$ must happen before position m_ℓ , hence there are finitely many of them. Moreover, a lock cannot be acquired and never released more than once. This shows condition F3. Conditions F4 and F5 are both satisfied because if $\ell \prec_H \ell'$ then $m_\ell < m_{\ell'}$. Thus \prec_H is acyclic and it cannot have infinite descending chains.

For the right-to-left implication, let τ satisfy all conditions of the lemma. In order to construct a run from τ we first build a total order $<$ on $H(\tau)$ that extends \prec_H and has no infinite descending chain. Let ℓ'_0, ℓ'_1, \dots be some arbitrary enumeration of $H(\tau)$ (which exists as τ is countable, thus so is $H(\tau)$). For all i let $\downarrow \ell'_i = \{\ell' \in H(\tau) \mid \ell' \prec_H^+ \ell'_i\}$. As τ satisfies condition F3, the set of nodes that are ancestors of a node with an operation on ℓ'_i is finite. Since additionally by condition F5 there are no infinite descending chains for \prec_H , the set $\downarrow \ell'_i$ is finite as well (by König's lemma). As \prec_H is acyclic by condition F4, we can choose some strict total order $<_i$ on $\downarrow \ell'_i$ that extends \prec_H . We define for all $\ell \in H(\tau)$ the index $m_\ell = \min\{i \in \mathbb{N} \mid \ell \in \downarrow \ell'_i\}$. Finally, we set $\ell < \ell'$ if either $m_\ell < m_{\ell'}$ or if $m_\ell = m_{\ell'}$ and $\ell <_{m_\ell} \ell'$. By definition $<$ is a strict total order on $H(\tau)$ with no infinite descending chains. Moreover it is easy to see that if $\ell \prec_H \ell'$ then $\ell < \ell'$. This is the case because $\ell \prec_H \ell'$ and $\ell' \prec_H^+ \ell_i$ implies $\ell \prec_H^+ \ell_i$, so $m_\ell \leq m_{\ell'}$.

Using the order $<$ on $H(\tau)$ we construct a process-fair run $\tau_0 \xrightarrow{+} \tau_1 \xrightarrow{+} \dots$ with τ as limit configuration. During the construction we maintain the following invariant for every i :

There exists $k_i \in \mathbb{N}$ such that all operations on locks ℓ_j with $j < k_i$ are already executed in τ_i (there is no operation on these locks in $\tau \setminus \tau_i$). Moreover, all other locks are free after executing τ_i : $H_i := H(\tau_i) = \{\ell_0, \dots, \ell_{k_i-1}\}$.

For $i = 0$ the invariant is clearly satisfied as all locks are free ($k_0 = 0$).

For $i > 0$ we assume that there is a run $\tau_0 \xrightarrow{+} \tau_i$ and τ_i satisfies the invariant. Thus, all locks ℓ_j with $j < k_i$ are ultimately held and all other locks are free in τ_i .

We say that a leaf ν of τ_i is *available* if one of the following holds:

1. either there is a descendant $\nu' \neq \nu$ on the leftmost path from ν in τ with $H(\nu') = H(\nu)$ in τ ,
2. or the left child ν' of ν in τ is labeled with an unmatched get of ℓ_{k_i} , and there is no further operation on ℓ_{k_i} in $\tau \setminus \tau_i$.

In particular, leaves of τ cannot be available. The strategy is to find the smallest available node ν in BFS order, and execute the actions on the left path from ν to ν' . The execution is possible as on this path there are no actions using locks from H_i and all other locks are free. Let τ_{i+1} denote the configuration thus obtained from τ_i . The invariant is satisfied after this execution, with $H_{i+1} = H_i$ in the first case above, resp. $H_{i+1} = H_i \cup \{\ell_{k_i}\}$ in the second case.

It remains to show that if a node is a leaf in τ_i for all i after some point, then it is a leaf in τ . This shows, in particular, that there always exists some available node.

Suppose that ν and i_0 are such that ν is a leaf of τ_i for all $i \geq i_0$. If ν becomes available at some point then it stays available in all future configurations, and there are finitely many nodes before ν in the BFS order. Thus ν cannot be available in some τ_i , as otherwise it would eventually be taken. Note that by the invariant (and soundness), no leaf of τ_i has the left child labeled by some `rel` operation. Moreover, every leaf ν of τ_i with left child ν' in τ labeled by `nop`, `spawn()`, or by some matched `get`, is available (the latter because we consider nested DLSS). Hence, the left child of ν must be labeled with an `unmatched get` of some $\ell \in H(\tau)$. Thus there is some `unmatched get` on a lock of $H(\tau)$ that is never executed.

Let m be the minimal index in the enumeration of $H(\tau)$ such that an `unmatched get` of ℓ_m in τ is never executed. By minimality of m , there exists i_1 such that $m = k_i$ for all $i \geq i_1$. After i_1 , all operations on locks $\ell < \ell_m$ have been executed. Thus, as $<$ extends $<_H$, all `unmatched get` operations that have some descendant in τ with operation on ℓ_m , have been executed. By the previous argument, the nodes with left child not labeled with an `unmatched get` cannot stay leaves forever. Hence, all nodes whose left child has some operation on ℓ_m eventually become leaves. The ones with matched `get` or other operations are then available and eventually executed.

Hence, after some point the only remaining operations on ℓ_m are `unmatched get`. Furthermore by the condition F3 of the lemma there is exactly one. As a result, when it is reached and all other operations on ℓ_m have been executed, it becomes available, and is thus eventually executed, contradicting the definition of m .

This proves that the limit of the run we have constructed is τ . Observe finally that the run is process-fair because of condition F2 of the lemma. \blacktriangleleft

The next lemma is an important step in the proof as it simplifies condition F4 of Lemma 11. This condition talks about the existence of a global order on some locks. The next lemma replaces this order with local orders in each of the nodes. These orders can be guessed by a finite automaton.

► **Lemma 12.** *Suppose that τ satisfies the first three conditions of Lemma 11. The relation $<_H$ is acyclic if and only if there is a family of strict total orders $<_\nu$ over a subset of variables from $\text{Var}(\nu)$ such that:*

F4.1 *x is ordered by $<_\nu$ if and only if $L(\nu)(x) \in H(\tau)$.*

F4.2 *if $x <_\nu x'$, ν' is a child of ν , and $y, y' \in \text{Var}(\nu')$ are such that $x \sim y$ and $x' \sim y'$ then $y <_{\nu'} y'$.*

F4.3 *if $x, x' \in \text{Var}(\nu)$ and $L(\nu)(x) <_H L(\nu)(x')$ then $x <_\nu x'$.*

4 Recognizing limit configurations

Recall that a configuration is a possibly infinite tree with five labels p, s, a, L, H . As we have mentioned in Remark 2, configurations need actually only three labels p, s, a . The other two can be calculated from the tree. Hence, configurations are labeled trees with node

24:10 Model-checking parametric lock-sharing systems against regular constraints

labels coming from a finite alphabet. Our goal in this section is to define a tree automaton recognizing limit configurations of process-fair runs of a given DLSS.

Our plan is to check the conditions (F1-5) of Lemma 11. Actually we will check (F1-3,5) and the conditions of Lemma 12 that are equivalent to F4 of Lemma 11.

□ We first observe that since our processes are finite state it is immediate to construct a nondeterministic tree automaton \mathcal{B}_1 verifying condition F1. This automaton just verifies local constraints between the labeling of a node and the labelings of its children. The constraints talk only about the labels p, s, a . The automaton does not need any acceptance condition, every run is accepting. We will say τ is *process-consistent* if it is accepted by \mathcal{B}_1 .

Checking condition F2 is more complicated because it refers to a set $H(\tau)$ of locks that are ultimately held by some process. Our approach will be to define four types of predicates and color the nodes of τ with these predicates. From a correct coloring of τ it will be easy to read out $H(\tau)$. Then we will show that the correct coloring can be characterized by conditions verifiable by finite tree automata. The coloring will be also instrumental in checking the remaining conditions F3, F4, F5.

For a configuration τ , a node ν and a variable $x \in \text{Var}(\nu)$ we define four predicates.

- $\nu \models \text{keeps}(x)$ if at ν process $p(\nu)$ holds the lock $\ell = L(\nu)(x)$ and never releases it: $\ell \in H(\nu')$ for every left descendant ν' of ν .
- $\nu \models \text{ev-keeps}(x)$ if $\nu \not\models \text{keeps}(x)$ and there is a descendant ν' of ν and a variable $x' \in \text{Var}(\nu')$ with $x \sim x'$ and $\nu' \models \text{keeps}(x')$.
- $\nu \models \text{avoids}(x)$ if neither $p(\nu)$ nor any descendant takes $\ell = L(\nu)(x)$, namely $\ell \notin H(\nu')$ for every descendant ν' of ν (including ν).
- $\nu \models \text{ev-avoids}(x)$ if $\nu \not\models \text{avoids}(x)$ and on every path from ν there is ν' such that $\nu' \models \text{avoids}(x)$.

Observe a different quantification used in *ev-keeps* and *ev-avoids*. In the first case we require one ν' to exist, in the second we want that such a ν' exists on every path.

The next lemma shows how we can use the coloring to determine $H(\tau)$.

► **Lemma 13.** *Let τ be a process-consistent configuration. A lock $\ell \in H(\tau)$ if and only if there is a node ν of τ and a variable $x \in \text{Var}(\nu)$ such that $\nu \models \text{keeps}(x)$ and $L(\nu)(x) = \ell$.*

Proof. Follows from the definitions, since $\nu \models \text{keeps}(x)$ if and only if $\ell \in H(\nu')$ for every left descendant ν' of ν . ◀

□ The above conditions define a *semantically correct* coloring of nodes of a configuration τ by sets of predicates

$$\mathcal{C}(\nu) = \{P(x) : x \in \text{Var}(\nu), \nu \models P(x)\}$$

where $P(x)$ is one of *keeps*(x), *ev-keeps*(x), *avoids*(x), *ev-avoids*(x). Observe that the four predicates are mutually exclusive, but it may be also the case that none of them holds. We say that a variable $x \in \text{Var}(\nu)$ is *uncolored* in ν if $\mathcal{C}(\nu)$ contains no predicate on x .

We now describe consistency conditions on a coloring of configurations guaranteeing that a coloring is semantically correct.

Before moving forward we introduce one piece of notation. A node that is a right child, namely a node of a form $\nu 1$ is due to $\text{spawn}(q, \sigma)$ operation. More precisely $op(\nu 0) = \text{spawn}(q, \sigma)$. We refer to this σ as $\sigma(\nu 1)$.

□ A coloring of a configuration τ is *branch-consistent* if for every node ν of τ and every variable $x \in \text{Var}(\nu)$ the following conditions are satisfied.

- If ν has one successor $\nu 0$ then $\nu 0$ inherits the colors from ν except for two cases depending on $op(\nu 0)$, i.e, the operation used to obtain $\nu 0$:
 - If $ev\text{-keeps}(x)$ is in $\mathcal{C}(\nu)$ and the operation is \mathbf{get}_x then $\mathcal{C}(\nu 0)$ must have either $ev\text{-keeps}(x)$ or $keeps(x)$.
 - If $ev\text{-avoids}(x)$ is in $\mathcal{C}(\nu)$ and the operation is \mathbf{rel}_x then $\mathcal{C}(\nu 0)$ must have either $ev\text{-avoids}(x)$ or $avoids(x)$.
- If ν has two successors $\nu 0, \nu 1$, and there is no y with $\sigma(\nu 1)(y) = x$ then $\nu 0$ inherits x color from ν and there is no constraint due to x on colors in $\nu 1$.
- If ν has two successors and $x = \sigma(\nu 1)(y)$ for some $y \in Var(\nu 1)$ then
 - If $keeps(x)$ in $\mathcal{C}(\nu)$ then $keeps(x)$ in $\mathcal{C}(\nu 0)$ and $avoids(y)$ in $\mathcal{C}(\nu 1)$.
 - If $avoids(x)$ in $\mathcal{C}(\nu)$ then $avoids(x)$ in $\mathcal{C}(\nu 0)$ and $avoids(y)$ in $\mathcal{C}(\nu 1)$.
 - If $ev\text{-keeps}(x)$ in $\mathcal{C}(\nu)$ then either
 - * $ev\text{-keeps}(x)$ in $\mathcal{C}(\nu 0)$ and either $avoids(y)$ or $ev\text{-avoids}(y)$ in $\nu 1$, or
 - * $ev\text{-keeps}(y)$ in $\mathcal{C}(\nu 1)$ and either $avoids(x)$ or $ev\text{-avoids}(x)$ in $\nu 0$.
 - If $ev\text{-avoids}(x)$ is ν then $ev\text{-avoids}(x)$ in $\mathcal{C}(\nu 0)$ and $ev\text{-avoids}(y)$ in $\mathcal{C}(\nu 1)$.

□ Next we describe when a coloring is *eventuality-consistent*. An *ev-trace* is a sequence of pairs $(\nu_1, x_1), (\nu_2, x_2), \dots$ where :

- ν_1, ν_2, \dots is a path in τ ,
- $x_i \in Var(\nu_i)$; moreover $x_{i+1} = x_i$ if ν_{i+1} is the left successor of ν_i , and $\sigma(\nu_{i+1})(x_{i+1}) = x_i$ if ν_{i+1} is the right successor of ν_i .
- $ev\text{-keeps}(x_i)$ or $ev\text{-avoids}(x_i)$ is in $\mathcal{C}(\nu_i)$.

Observe that it follows that it cannot be the case that we have $ev\text{-keeps}(x_i)$ and $ev\text{-avoids}(x_{i+1})$ or vice versa. A coloring is *eventuality-consistent* if every *ev-trace* in the coloring of a configuration is finite.

□ Finally, a coloring is *recurrence-consistent* if for every ν and uncolored $x \in Var(\nu)$ the lock $\ell = L(\nu)(x)$ is taken and released infinitely often below ν .

□ A coloring is *syntactically correct* if it is branch-consistent, eventuality-consistent, and recurrence-consistent. We show that syntactically correct colorings characterize semantically correct colorings. The two implications are stated separately as the statements are slightly different.

► **Lemma 14.** *If τ is a limit configuration and \mathcal{C} is a semantically correct coloring of τ then \mathcal{C} is syntactically correct.*

For the other direction, we prove a more general statement without assuming that τ is a limit configuration. This is important as ultimately we will use the consistency properties to test if τ is a limit configuration.

► **Lemma 15.** *If τ is a configuration and \mathcal{C} a syntactically correct coloring of τ , then \mathcal{C} is semantically correct.*

Having a correct coloring will help us to verify all conditions of Lemma 11. Condition F2 refers to $L(\nu)(x) \in H(\tau)$. We need another labeling to be able to express this.

□ A *syntactic H-labeling* of τ assigns to every node ν a subset $H^s(\nu) \subseteq Var(\nu)$. We require the following properties:

- For the root ε we have $H^s(\varepsilon) = \emptyset$.
- If $\nu 0$ exists: $x \in H^s(\nu 0)$ if and only if $x \in H^s(\nu)$.
- If $\nu 1$ exists: $y \in H^s(\nu 1)$ if and only if either $\sigma(\nu 1)(y) = \mathbf{new}$ and $\nu 1 \models ev\text{-keeps}(y)$, or $\sigma(\nu 1)(y) = x$ and $\nu \models ev\text{-keeps}(x)$.

It is clear that every configuration tree has a unique H^s labelling.

24:12 Model-checking parametric lock-sharing systems against regular constraints

► **Lemma 16.** *Let τ be a process-consistent configuration with syntactically correct coloring. For every node ν and variable $x \in \text{Var}(\nu)$ we have: $L(\nu)(x) \in H(\tau)$ if and only if $x \in H^s(\nu)$.*

Thanks to Lemma 16 we obtain

► **Lemma 17.** *Let τ be a process-consistent configuration with a syntactically correct coloring. Condition F2 of Lemma 11 holds for τ if and only if for every leaf ν of τ , every possible transition from $s(\nu)$ has some get_x operation with $x \in H^s(\nu)$.*

► **Lemma 18.** *Let τ be a process-consistent configuration with a syntactically correct coloring. Then condition F3 of Lemma 11 holds for τ .*

It remains to deal with conditions F4 and F5 of Lemma 11. Condition F4 is more difficult to check as it requires to find an acyclic relation with some properties. Fortunately Lemma 12 gives an equivalent condition talking about a family of local orders $<_\nu$ for every node ν of a configuration. An automaton can easily guess such a family of orders. We show that it can also check the required properties.

□ A *consistent order labeling* assigns to every node ν of τ a total order $<_\nu$ on some subset of $\text{Var}(\nu)$. The assignment must satisfy the following conditions for every node ν :

1. x is ordered by $<_\nu$ if and only if $x \in H^s(\nu)$,
2. if $x <_\nu x'$ and $x, x' \in \text{Var}(\nu 0)$ then $x <_{\nu 0} x'$,
3. if $x <_\nu x'$, $\nu 1$ exists, and $\sigma(\nu 1)(y) = x$, $\sigma(\nu 1)(y') = x'$ then $y <_{\nu 1} y'$,
4. if $\nu \models \text{keeps}(x)$ and $y <_\nu x$ then $\nu \models \text{keeps}(y)$ or $\nu \models \text{avoids}(y)$.

► **Lemma 19.** *Let τ be a process-consistent configuration with a syntactically correct coloring. A family of local orders $<_\nu$ is a consistent order labeling of τ if and only if it satisfies the conditions of Lemma 12.*

□ We consider now condition F5. We say that a consistent order labeling of τ admits an *infinite descending chain* if there exist a sequence of nodes ν_1, ν_2, \dots and variables $(x_i)_i, (y_i)_i$ such that for every $i > 0$: (i) ν_i is an ancestor of ν_{i+1} , (ii) $y_i \sim x_{i+1}$, and (iii) $y_i <_{\nu_i} x_i$.

► **Lemma 20.** *Let τ be a process-consistent configuration with a syntactically correct coloring. If \prec_H has no infinite descending chain then there is a consistent order labeling of τ with no infinite descending chain. If \prec_H has an infinite descending chain then every consistent order labeling of τ admits an infinite descending chain.*

The next proposition summarizes the development of this section stating that all the relevant properties can be checked by a Büchi tree automaton.

► **Proposition 21.** *For a given DLSS, there is a non-deterministic Büchi tree automaton $\hat{\mathcal{B}}$ accepting exactly the limit configurations of process-fair runs of DLSS. The size of $\hat{\mathcal{B}}$ is linear in the size of the DLSS and exponential in the maximal arity of the DLSS.*

We will show that the previous proposition yields an EXPTIME algorithm. We match it with an EXPTIME lower bound to obtain completeness.

► **Proposition 22.** *The DLSS verification problem for nested DLSS and Büchi objective is EXPTIME-hard. The result holds even if the Büchi objective refers to a single process.*

The hardness proof involves a reduction from the problem of determining whether the intersection of the languages of k deterministic tree automata over binary trees is empty. To achieve this, we create a DLSS that simulates all the tree automata concurrently. Each node of the tree in the intersection is simulated by a process, which encodes a state for each automaton through the locks it holds. So each process creates two children with whom it shares locks. The children are able to access the states of the parent by the following technique: Suppose processes p and q share locks 0 and 1, and p acquires one lock and retains it indefinitely. In this scenario, q can guess the lock chosen by p and try to acquire the other lock. If q guesses incorrectly, the system deadlocks. However, if the guess is correct, the execution continues, and q knows about the lock held by p .

Now we have all ingredients for the proof of Theorem 7:

Proof of Theorem 7. The lower bound follows from Proposition 22.

For the upper bound we use the Büchi tree automaton $\hat{\mathcal{B}}$ recognizing limit configurations of the DLSS (Proposition 21).

We build the product of $\hat{\mathcal{B}}$ with the regular objective automaton \mathcal{A} , which is a parity tree automaton. From $\hat{\mathcal{B}} \times \mathcal{A}$ we can obtain with a bit more work an equivalent parity tree automaton \mathcal{C} with the same number of priorities, plus one. For this we modify the rank function in order to only store in the state the maximal priority seen between two consecutive occurrences of Büchi accepting states, and make the maximal priority visible at the next Büchi state. When the state of the $\hat{\mathcal{B}}$ component is not a Büchi state, the priority is odd and lower than all the ones of \mathcal{A} .

By Proposition 21, \mathcal{C} is non-empty if and only if there exists a limit configuration of the system that satisfies the regular objective \mathcal{A} . Moreover, we know that $\hat{\mathcal{B}}$ has size linear in the size of the DLSS and exponential only in the maximal arity of processes. So \mathcal{C} has size that is exponential w.r.t. the DLSS and the objective, and polynomial size if the maximal arity is fixed.

Finally, non-emptiness of \mathcal{C} amounts to solve a parity game of the same size as \mathcal{C} : player Automaton chooses transitions of \mathcal{C} , and player Pathfinder chooses the direction (left/right child). To sum up, we obtain a parity game of exponential size, so solving the game takes exponential time since the number of priorities is polynomial. If both the number of priorities and the maximal arity are fixed, the game can be solved in polynomial time. ◀

5 Pushdown systems with locks

Till now every process has been a finite state system. Here we consider the case when processes can be pushdown automata. The definition of a *pushdown DLSS* is the same as before but now each automaton \mathcal{A}_p is a deterministic pushdown automaton.

We will reduce our verification problem to the emptiness test of a nondeterministic pushdown automata on infinite trees. These automata will have parity acceptance conditions. While in general testing emptiness of such automata is EXPTIME-complete, we will notice that the automata we construct have a special form allowing to test emptiness in PTIME for a fixed number of ranks in the parity condition.

□ We start by defining *pushdown tree automata*. We work with a ranked alphabet $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, so a letter determines whether a node has zero, one or two children. Our automaton will be quite standard but for an additional stack instruction. Apart standard `pop` and `push(a)`, we have a `reset` instruction that empties the stack. A pushdown tree automaton is a tuple $(Q, \Sigma, \Gamma, q^0, \perp, \delta, \Omega)$, where Q is a finite set of states, Σ an input alphabet, Γ a stack alphabet, $q^0 \in Q$ an initial state, $\perp \in \Gamma$ a bottom stack symbol, and

24:14 Model-checking parametric lock-sharing systems against regular constraints

$\Omega : Q \rightarrow \{1, \dots, d\}$ a parity condition. Finally, δ is a partial transition function taking as the arguments the current state q , the current input letter a , and the current stack symbol γ . The form of transitions in δ depends on the rank of the letter a :

- For $a \in \Sigma_0$, we have $\delta(q, a, \gamma) = \top$ for a special symbol \top . This means that the automaton accepts in a leaf of the tree if δ is defined.
- For $a \in \Sigma_1$, we have $\delta(q, a, \gamma) = (q', \text{instr})$ where instr is one of the stack instructions.
- For $a \in \Sigma_2$, we have $\delta(q, a, \gamma) = ((q_l, \text{instr}_l), (q_r, \text{instr}_r))$, so now we have two states, going to the left and right, respectively, and two separate stack instructions.

A run of such an automaton on a Σ -labeled tree is an assignment of configurations to nodes of the tree; each configuration has the form (q, w) where $q \in Q$ is a state and $w \in \Gamma^+$ is a sequence of stack symbols representing the stack (top symbol being the leftmost). The root is labeled with (q^0, \perp) . The labelling of children must depend on the labeling of the parent according to the transition function δ . In particular, if a leaf of the tree is labeled a and has assigned a configuration (q, w) then $\delta(q, a, \gamma)$ must be defined, where γ is the leftmost symbol of w . A run is accepting if for every infinite path the sequence of assigned states satisfies the max parity condition given by Ω : the maximum of ranks of states seen on the path must be even.

□ We say that a pushdown tree automaton is *right-resetting* if for every transition $\delta(q, a, \gamma) = ((q_l, \text{instr}_l), (q_r, \text{instr}_r))$ we have that instr_r is **reset**.

► **Proposition 23.** *For a fixed d , the emptiness problem for right-resetting pushdown tree automata with a parity condition over ranks $\{1, \dots, d\}$ can be solved in PTIME.*

Proof. We consider the representative case of $d = 3$. Suppose we are given a right-resetting pushdown tree automaton $\mathcal{A} = (Q, \Sigma, \Gamma, q^0, \perp, \delta, \Omega)$.

The first step is to construct a pushdown word automaton $\mathcal{A}^l(G_1, G_2, G_3)$ depending on three sets of states $G_1, G_2, G_3 \subseteq Q$. The idea is that \mathcal{A}^l simulates the run of \mathcal{A} on the leftmost branch of a tree. When \mathcal{A} has a transition going both to the left and to the right then \mathcal{A}^l goes to the left and checks if the state going to the right is in an appropriate G_i . This means that \mathcal{A}^l works over the alphabet Σ^l that is the same as Σ but all letters from Σ_2 have rank 1 instead of 2. The states of $\mathcal{A}^l(G_1, G_2, G_3)$ are $Q \times \{1, 2, 3\}$ with the second component storing the maximal rank of a state seen so far on the run. The transitions of $\mathcal{A}^l(G_1, G_2, G_3)$ are defined according to the above description. We make precise only the case for a transition of \mathcal{A} of the form $\delta(q, a, \gamma) = ((q_l, \text{instr}_l), (q_r, \text{instr}_r))$. In this case, \mathcal{A}^l has a transition $\delta^l((q, i), a, \gamma) = ((q_l, \max(i, \Omega(q_l))), \text{instr}_l)$ if $q_r \in G_{\max(i, \Omega(q_r))}$. Observe that instr_r is necessarily **reset** as \mathcal{A} is right-resetting.

The next step is to observe that for given sets G_1, G_2, G_3 we can calculate in PTIME the set of states from which $\mathcal{A}^l(G_1, G_2, G_3)$ has an accepting run.

The last step is to compute the following fixpoint expression in the lattice of subsets of Q :

$$W = \text{LFP}X_3. \text{GFP}X_2. \text{LFP}X_1. P(X_1, X_2, X_3) \quad \text{where}$$

$$P(X_1, X_2, X_3) = \{q : \mathcal{A}^l(X_1, X_2, X_3) \text{ has an accepting run from } q\} .$$

Observe that $P : \mathcal{P}(Q)^3 \rightarrow \mathcal{P}(Q)$ is a monotone function over the lattice of subsets of Q . Computing W requires at most $|Q|^3$ computations of P for different triples of sets of states.

We claim that \mathcal{A} has an accepting run from a state q , if and only if, $q \in W$. The argument is presented in the appendix. ◀

Proof of Theorem 8. The lower bound follows already from Theorem 7.

For the upper bound we reuse the Büchi tree automaton $\hat{\mathcal{B}}$ from Proposition 21. This time $\hat{\mathcal{B}}$ is a pushdown tree automaton, however it is right-resetting because processes are spawned with empty stack. We follow the lines of the proof of Theorem 7, building the product of $\hat{\mathcal{B}}$ with the regular objective automaton \mathcal{A} , and constructing an equivalent parity, right-resetting pushdown tree automaton \mathcal{C} . Proposition 23 concludes the proof. \blacktriangleleft

6 Conclusions

We have considered verification of parametric lock sharing systems where processes can spawn other processes and create new locks. Representing configurations as trees, and the notion of the limit configuration, are instrumental in our approach. We believe that we have made stimulating observations about this representation. It is very easy to express fairness as a property of a limit configuration. Many interesting properties, including liveness, can be formulated very naturally as properties of limit trees (cf. page 6). Moreover, there are structural conditions characterizing when a tree is a limit configuration of a run of a given system (Lemma 12).

We expect that the parameters in Theorem 8 will be usually quite small. As the dining philosophers example suggests, for many systems the maximal arity should be quite small (cf. Figure 1). Indeed, the maximal arity of the system corresponds to the tree width of the graph where process instances are nodes and edges represent sharing a lock. The maximal priority will be often 3. In our opinion, most interesting properties would have the form “there is a left path such that” or “all left paths are such that”, and these properties need only automata with three priorities. So in this case our verification algorithm is in PTIME.

Our handling of pushdown processes is different from the literature. Most of our development is done for finite state processes, while the transition to pushdown process is handled through right-resetting concept. Proposition 23 implies that in our context pushdown processes are essentially as easy to handle as finite processes.

As further work it would be interesting to see if it is possible to extend our approach to treat join operation [12]. An important question is how to extend the model with some shared state and still retain decidability for the pushdown case.

References

- 1 Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 685–725. Springer, 2018. doi:10.1007/978-3-319-10575-8_21.
- 2 S. Akshay, Paul Gastin, Shankara Narayanan Krishna, and Sparsa Roychowdhury. Revisiting underapproximate reachability for multipushdown systems. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 387–404. Springer, 2020. doi:10.1007/978-3-030-45190-5_21.
- 3 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetsche. The complexity of bounded context switching with dynamic thread creation. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPICs*, pages 111:1–111:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ICALP.2020.111.

- 4 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetsche. Context-bounded verification of thread pools. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi:10.1145/3498678.
- 5 Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
- 6 Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR'97: Concurrency Theory, 8th International Conference, Warsaw*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. doi:10.1007/3-540-63141-0\10.
- 7 Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 473–487. Springer, 2005. doi:10.1007/11539452\36.
- 8 Xvisor commit message fixing issue:. URL: <https://github.com/xvisor/xvisor/commit/e5dd8291b5e3f0c552b9aacc73ef2f000ae14c09>.
- 9 Marcio Diaz and Tayssir Touili. Dealing with priorities and locks for concurrent programs. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2017. doi:10.1007/978-3-319-68167-2\15.
- 10 Javier Esparza, Pierre Ganty, and Rupak Majumdar. Parameterized verification of asynchronous shared-memory systems. *J. ACM*, 63(1):10:1–10:48, 2016. doi:10.1145/2842603.
- 11 Marie Fortin, Anca Muscholl, and Igor Walukiewicz. Model-checking linear-time properties of parametrized asynchronous shared-memory pushdown systems. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 155–175. Springer, 2017. doi:10.1007/978-3-319-63390-9\9.
- 12 Thomas Martin Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2011. doi:10.1007/978-3-642-18275-4\15.
- 13 S. A. German and P. A. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
- 14 Matthew Hague. Parameterised pushdown systems with non-atomic writes. In Supratik Chakraborty and Amit Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*, volume 13 of *LIPICs*, pages 457–468. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPICs.FSTTCS.2011.457.
- 15 Vineet Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 27–36, 2009. doi:10.1109/LICS.2009.45.
- 16 Vineet Kahlon, Franjo Ivancić, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV'05*, page 505–518, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11513988_49.
- 17 Sebastian Kenter. *Lock-sensitive reachability analysis for parallel recursive programs with dynamic creation of threads and locks: a graph-based approach*. PhD thesis, University of Münster, Germany, 2022. URL: <https://nbn-resolving.org/urn:nbn:de:hbz:6-21089543742>.

- 18 Peter Lammich. *Lock sensitive analysis of parallel programs*. PhD thesis, University of Münster, 2011. URL: <https://nbn-resolving.org/urn:nbn:de:hbz:6-43459441169>.
- 19 Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Contextual locking for dynamic pushdown networks. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 477–498. Springer, 2013. doi:10.1007/978-3-642-38856-9_25.
- 20 Peter Lammich, Markus Müller-Olm, and Alexander Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 525–539. Springer, 2009. doi:10.1007/978-3-642-02658-4_39.
- 21 Anca Muscholl, Helmut Seidl, and Igor Walukiewicz. Reachability for dynamic parametric processes. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2017. doi:10.1007/978-3-319-52234-0_23.
- 22 Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005. doi:10.1007/978-3-540-31980-1_7.
- 23 Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 161–170. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.9.
- 24 Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. Reachability of scope-bounded multistack pushdown systems. *Inf. Comput.*, 275:104588, 2020. doi:10.1016/j.ic.2020.104588.
- 25 Kazuhide Yasukata, Takeshi Tsukada, and Naoki Kobayashi. Verification of higher-order concurrent programs with dynamic resource creation. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 335–353, 2016. doi:10.1007/978-3-319-47958-3_18.

A Fairness

► **Proposition 3.** *Consider a run $\tau_0 \xrightarrow{\nu_1, a_1} \tau_1 \xrightarrow{\nu_2, a_2} \dots$ and its limit configuration τ . The run is strongly process-fair if and only if τ is fair.*

Proof. Consider the left-to-right implication. Suppose towards a contradiction that the run is not process-fair. So there is a transition $\tau \xrightarrow{\nu, a} \tau'$ for some leaf ν . Let p be the process moving in ν , and let τ_i be the first configuration where ν appears in τ . We show that p moves after this configuration, contradicting the fact that ν is a leaf.

If $op(a)$ is not a **get** operation then (ν, a) is enabled in every configuration τ_j for $j > i$. By strong fairness p must move after i .

A more interesting case is when $op(a)$ is **get_x** for some x . Let $\ell = L(\nu)(x)$ be the lock taken by the transition. As (ν, a) is enabled in τ , we have that $\ell \notin H(\tau)$. We show that this implies that process p is enabled infinitely often after position i . By soundness, as ν, a is enabled, p cannot hold ℓ : so $\ell \notin H(\nu)$. If $\ell \in H(\tau_i)$ and ℓ is never released afterwards then there is a node ν' in τ_i (and thus in τ) such that for every left descendant of ν' we have $\ell \in H(\nu')$. But this is impossible since we have assumed $\ell \notin H(\tau)$. Hence, either ℓ is free in τ_i or it is free in some later configuration τ_{i_1} such that $\tau_{i_1} \xrightarrow{\nu', b} \tau_{i_1+1}$ and $op(b) = \mathbf{get}_y$ with $L(\nu')(y) = \ell$. So, (ν, a) is enabled in τ_{i_1} . If ℓ is never taken after i_1 then (ν, a) is enabled always after i_1 , and we get a move of p by strong fairness as before. If ℓ is taken after i_1 then by the same argument as above there must be also a position i_2 when ℓ is released. So, (ν, a) is enabled in τ_{i_2} . This argument shows that (ν, a) must be enabled infinitely often after i , so by strong fairness there must be a move by p after i .

Consider now the right-to-left implication. Suppose that τ is process-fair, and the process p is enabled infinitely often after position i . By contradiction, assume that p does not move after position i and let ν be the last node of p 's local run. If the action a of p that is enabled infinitely often is not a **get** then a is enabled in every τ_j with $j \geq i$, and $\tau \xrightarrow{\nu, a} \tau'$, contradicting process-fairness. Else, $op(a)$ is **get_x** with $L(\nu)(x) = \ell$. Since a is enabled infinitely often, $\ell \notin H(\tau)$. Again we have $\tau \xrightarrow{\nu, a} \tau'$, contradicting process-fairness. ◀

B Characterizing limit configurations

► **Lemma 12.** *Suppose that τ satisfies the first three conditions of Lemma 11. The relation \prec_H is acyclic if and only if there is a family of strict total orders $<_\nu$ over a subset of variables from $Var(\nu)$ such that:*

F4.1 x is ordered by $<_\nu$ if and only if $L(\nu)(x) \in H(\tau)$.

F4.2 if $x <_\nu x'$, ν' is a child of ν , and $y, y' \in Var(\nu')$ are such that $x \sim y$ and $x' \sim y'$ then $y <_{\nu'} y'$.

F4.3 if $x, x' \in Var(\nu)$ and $L(\nu)(x) \prec_H L(\nu)(x')$ then $x <_\nu x'$.

Proof. For the left-to-right direction we fix a strict total order $<$ on $H(\tau)$ that is compatible with \prec_H (for instance the strict order $<$ defined in the proof of Lemma 11). Then we order the variables $x \in Var(\nu)$ with $L(\nu)(x) \in H(\tau)$ according to $<$. The three conditions of the lemma then follow directly.

For the converse we define \prec on $H(\tau)$ by $\ell \prec \ell'$ if for some node ν with variables $x \neq x'$ such that $L(\nu)(x) = \ell$ and $L(\nu)(x') = \ell'$ we have $x <_\nu x'$.

We start by showing that \prec is acyclic. Assume by contradiction that $\ell_0 \prec \ell_1 \prec \dots \prec \ell_k \prec \ell_0$ is a cycle of minimal length, so the locks $\ell_i \in H(\tau)$ are all distinct. We use indices modulo $k + 1$, so $k + 1 \equiv 0$. Note that $k > 1$ because of condition F4.2.

By assumption, the scopes of ℓ_i and ℓ_{i+1} intersect, for every i . Since scopes are subtrees of τ (Remark 10) this means that two scopes that intersect have roots that are ordered by the ancestor relation in τ .

Assume first that $k > 2$. Let i be such that the depth of the root of the scope of ℓ_i is maximal. So the roots of the scopes of ℓ_{i-1} and ℓ_{i+1} are ancestors of the root ν of the scope of ℓ_i . In the scope of ℓ_i there exist nodes that belong to the scope of ℓ_{i-1} and of ℓ_{i+1} , respectively. This means that ν is in the scope of ℓ_{i-1} , ℓ_i and ℓ_{i+1} . So the scopes of ℓ_{i-1} and ℓ_{i+1} intersect, and we have either $\ell_{i-1} <_\nu \ell_{i+1}$ or $\ell_{i+1} <_\nu \ell_{i-1}$. Thus we get from the definition of $<$ either $\ell_{i-1} < \ell_{i+1}$ or $\ell_{i+1} < \ell_{i-1}$. In both cases the cycle $\ell_0 < \ell_1 \cdots < \ell_k < \ell_0$ is not minimal, a contradiction.

It remains to consider the case $\ell_0 < \ell_1 < \ell_2 < \ell_0$. With a similar argument as before there exists a node ν which is in the scope of all of ℓ_0, ℓ_1, ℓ_2 , so this node gives a total order on these locks and there cannot exist a cycle.

We now show that $<_H$ is acyclic as well.

Like before, suppose there exists a cycle of distinct nodes $\ell_0 <_H \ell_1 <_H \cdots <_H \ell_k <_H \ell_{k+1} = \ell_0$ with $k > 0$. We consider such a cycle of minimal size. Hence every ℓ_i is comparable with ℓ_{i-1}, ℓ_{i+1} and incomparable with all the other ℓ_j (as otherwise we would obtain a shorter cycle).

Given $\ell, \ell' \in H(\tau)$ such that $\ell <_H \ell'$, let ν be the node with an **unmatched get** of ℓ . By condition F3, this node is unique. By the definition $<_H$ this node has some descendant ν' with an operation on ℓ' . There are two possibilities, one is that the scopes of ℓ, ℓ' intersect, in which case by condition F4.3 we have $\ell < \ell'$. The other possibility is that the two subtrees do not intersect, in which case the root of $\theta(\ell')$ is strictly below the **unmatched get** of ℓ .

As $<$ is acyclic, there exists some i such that the scopes of ℓ_{i-1} and ℓ_i are disjoint, hence all nodes of the scope of ℓ_i are below the **unmatched get** of ℓ_{i-1} . In particular the **unmatched get** of ℓ_{i-1} is an ancestor of the **unmatched get** of ℓ_i . As a result, by the definition of $<_H$, $\ell_{i-1} <_H \ell_{i+1}$.

If $k \geq 2$ then the above argument shows that the cycle was not minimal, yielding a contradiction.

If $k = 1$ then we have a contradiction as well, as either the scopes intersect, so we cannot have both $\ell_0 < \ell_1$ and $\ell_1 < \ell_0$. Or they do not intersect, but then there is a node ν in the intersection with either $\ell_0 <_\nu \ell_1$ or $\ell_1 <_\nu \ell_0$, but not both.

As a result, the relation $<_H$ is acyclic. ◀

C Recognizing limit configurations

► **Lemma 14.** *If τ is a limit configuration and \mathcal{C} is a semantically correct coloring of τ then \mathcal{C} is syntactically correct.*

Proof. Suppose \mathcal{C} is a correct coloring of τ . Clearly τ is process-consistent. Branch consistency follows from Lemma 11 (condition F3). Indeed, all the clauses for *keeps*(x) and *ev-keeps*(x) hold because of the third condition of this lemma. The clauses for *avoids*(x) and *ev-avoids*(x) follow directly from the semantics. Directly from definition of the correct coloring it follows that it is also eventuality-consistent. It is slightly more difficult to verify that it is recurrence-consistent.

To verify recurrence consistency of \mathcal{C} consider an arbitrary node ν of τ and an uncolored variable $x \in \text{Var}(\nu)$. We find an infinite sequence:

$$(\nu, x) = (\nu_0, x_0), (\nu_1, x_1), \dots$$

such that

- $x_i \in \text{Var}(\nu_i)$ and x_i is uncolored in ν_i ,
- ν_0, ν_1, \dots is a path,
- $x_i \sim x_{i+1}$.

Let us see why it is possible. Since every leaf satisfies either $\nu \models \text{keeps}(x)$ or $\nu \models \text{avoids}(x)$, node ν is not a leaf. If $\nu 0$ satisfies $\text{keeps}(x)$ or $\text{ev-keeps}(x)$ then so does ν . If $\nu 0$ satisfies $\text{avoids}(x)$ or $\text{ev-avoids}(x)$ and is the unique successor then so does ν . Hence, if $\nu 0$ is the unique successor of ν then x cannot be colored in $\nu 0$. If $\nu 1$ exists, but there is no y with $x = \sigma(\nu 1)(y)$ then the same verification shows that x cannot be colored in $\nu 0$. Finally, if $x = \sigma(\nu 1)(y)$ and y is colored in $\nu 1$ then x must be colored too. Hence, y is not colored in $\nu 1$ in this last case. This shows how to find (ν_1, x_1) . Repeating this argument we obtain the desired sequence.

To terminate we show why the existence of the above sequence implies the recurrence condition. First note that $x_i \sim x_j$ for all $i, j \geq 0$. Let $\ell = L(\nu)(x)$. We observe that since ν_i does not satisfy $\text{avoids}(x_i)$ then there must be an operation on ℓ below ν_i , and since it does not satisfy $\text{ev-keeps}(x)$ it must be a release. So we have found an infinite path such that in the subtree of every node of this path there is a release operation. This means that there are infinitely many get and release operations on ℓ in the tree below ν ◀

► **Lemma 15.** *If τ is a configuration and \mathcal{C} a syntactically correct coloring of τ , then \mathcal{C} is semantically correct.*

Proof. Process consistency guarantees that locally labels follow the transition relations. Branch consistency on $\text{keeps}(x)$ and $\text{avoids}(x)$ labels guarantees that if ν is labeled by one of these predicates then the predicate holds in ν . To get the same property for $\text{ev-keeps}(x)$ and $\text{ev-avoids}(x)$ we need the eventuality-consistent condition.

Finally, if x is uncolored at ν then the recurrence-consistent condition implies that x satisfies none of the four predicates. ◀

► **Lemma 16.** *Let τ be a process-consistent configuration with syntactically correct coloring. For every node ν and variable $x \in \text{Var}(\nu)$ we have: $L(\nu)(x) \in H(\tau)$ if and only if $x \in H^s(\nu)$.*

Proof. Suppose $\ell = L(\nu)(x)$ and $\ell \in H(\tau)$. Take the node ν' that is closest to the root and has $\ell = L(\nu')(x')$ for some x' . We have $\nu' \models \text{ev-keeps}(x')$ and ν' is a right child (it cannot be the root as $\text{Var}(\varepsilon) = \emptyset$). Hence, $x' \in H^s(\nu')$. By induction on the length of the path from ν' to ν we show that $x \in H^s(\nu)$.

For the other direction, if $\nu \models \text{ev-keeps}(x)$ then $L(\nu)(x) \in H(\tau)$. It is also easy to see that membership in $H(\tau)$ is preserved by all the rules. ◀

► **Lemma 17.** *Let τ be a process-consistent configuration with a syntactically correct coloring. Condition F2 of Lemma 11 holds for τ if and only if for every leaf ν of τ , every possible transition from $s(\nu)$ has some get_x operation with $x \in H^s(\nu)$.*

Proof. By Lemma 16. ◀

► **Lemma 18.** *Let τ be a process-consistent configuration with a syntactically correct coloring. Then condition F3 of Lemma 11 holds for τ .*

Proof. Consider $\ell \in H(\tau)$. By Lemma 13 there is a node ν and $x \in \text{Var}(\nu)$ with $\nu \models \text{keeps}(x)$, $\ell = L(\nu)(x)$. Let ν' be the root of the scope of ℓ . We have $\nu' \models \text{ev-keeps}(x')$ for $x' \in \text{Var}(\nu')$ with $x' \sim x$. By consistency conditions on the coloring:

- for every node ν'' on the path from ν' to ν we have $\nu'' \models \text{ev-keeps}(x'')$ for $x'' \sim x$, and for every right child ν''' of ν'' we have $\nu''' \models \text{ev-avoids}(x''')$ for $x''' \sim x$.

Observe that $\nu''' \models \text{ev-avoids}(x''')$ guarantees that there are only finitely many operations on ℓ below ν''' , and that there is no unmatched get of ℓ below ν''' . Since there are no operations on ℓ below ν , we are done. \blacktriangleleft

► **Lemma 19.** *Let τ be a process-consistent configuration with a syntactically correct coloring. A family of local orders $<_\nu$ is a consistent order labeling of τ if and only if it satisfies the conditions of Lemma 12.*

Proof. Let us take a family of orders $<_\nu$ satisfying conditions F4.1, F4.2, F4.3 of Lemma 12. We show that it is a consistent order labeling of τ . By Lemma 16 the first condition is satisfied. The next two conditions follow from condition F4.2. The fourth condition requires some verification. Consider ν as in that condition, so with $y <_\nu x$ and $\nu \models \text{keeps}(x)$. It follows that there is some ancestor ν' of ν , together with some $x' \sim x$, $x' \in \text{Var}(\nu')$, such that the action at ν' is an unmatched $\text{get}_{x'}$ of the lock $\ell = L(\nu')(x') = L(\nu)(x)$. If there were some operation on $\ell' = L(\nu)(y)$ below or at ν then $\ell \prec_H \ell'$, implying $x <_\nu y$ by F4.3. Thus there is no operation on ℓ' below or at ν , meaning that $\nu \models \text{keeps}(y)$ or $\nu \models \text{avoids}(y)$.

For the other direction, take a consistent order labeling $<_\nu$. We show that it satisfies the conditions F4.1, F4.2, F4.3 of Lemma 12. From the first condition on $<_\nu$ and Lemma 16 we see that $<_\nu$ orders only variables associated with locks from $H(\tau)$; this gives us F4.1. Condition F4.2 follows directly from the second and third property of consistent order labeling.

It remains to show F4.3. For this take a node ν and two locks $\ell = L(\nu)(x)$ and $\ell' = L(\nu)(y)$ for some $x, y \in H^s(\nu)$. Suppose $\ell \prec_H \ell'$. This means that there is an unmatched get of ℓ , say in a node ν' , and an operation on ℓ' at some node ν'' below ν' .

We show below that we can find some node ν_1 in the scope of both ℓ and ℓ' , and such that $\nu_1 \models \text{keeps}(x_1)$ and $\nu_1 \not\models \text{keeps}(y_1)$ and $\nu_1 \not\models \text{avoids}(y_1)$, with $x \sim x_1$ and $y \sim y_1$. This will show that we cannot have $y_1 <_{\nu_1} x_1$, so it must hold that $x_1 <_{\nu_1} y_1$, thus also $x <_\nu y$ by local consistency.

- If either ν, ν' are incomparable, or ν is an ancestor of ν' , or $\nu = \nu'$, then ν' and $\nu'0$ are in the scope of both ℓ and ℓ' (note that $\nu'0$ is an ancestor of ν'' , or they can be equal). We choose $\nu_1 = \nu'0$.
- If $\nu' \neq \nu$ is an ancestor of ν , but ν and ν'' are either incomparable, or ν is an ancestor of ν'' , then we chose ν_1 as the least common ancestor of ν'' and ν . Note that ν_1 is below or equal to $\nu'0$, and belongs to the scope of both ℓ and ℓ' .
- If ν'' is an ancestor of ν then ν'' is in the scope of both ℓ and ℓ' , so we chose ν_1 to be ν'' .

► **Lemma 20.** *Let τ be a process-consistent configuration with a syntactically correct coloring. If \prec_H has no infinite descending chain then there is a consistent order labeling of τ with no infinite descending chain. If \prec_H has an infinite descending chain then every consistent order labeling of τ admits an infinite descending chain.*

Proof. The first statement is easy: take the well-founded strict order on locks $<$ defined in the proof of Lemma 11, and for each node ν take as $<_\nu$ the order given by $<$ on $L(\nu)(\text{Var}(\nu))$. The well-foundedness of $<$ implies that there is no infinite descending chain in the order labeling.

For the second part, assume \prec_H has an infinite descending chain and let $(\prec_\nu)_{\nu \in \tau}$ be a consistent order labeling of τ . Let $\ell_0 \succ_H \ell_1 \succ_H \dots$ be an infinite descending chain for \prec_H .

For all $i \geq 1$ let μ_i be a node with an unmatched **get** of ℓ_i and with a descendant with a **get** of ℓ_{i-1} . Let c_i be the root of the scope of ℓ_i in τ . As μ_{i+1} is an ancestor of a node where ℓ_i appears, it is comparable with c_i (two nodes are *comparable* if one is an ancestor of the other). As c_{i+1} is an ancestor of μ_{i+1} , it is comparable with c_i .

Claim 1: For all $a \leq b$, there exists $i \in \{a, \dots, b\}$ such that c_i is an ancestor of all $(c_k)_{a \leq k \leq b}$.

Proof. We proceed by induction on $b - a$. If $b - a = 0$ this is clear. If $b - a > 0$, by induction hypothesis there exists $i \in \{a, \dots, b - 1\}$ such that c_i is an ancestor of all $(c_j)_{a \leq j \leq b-1}$. As c_b is comparable with c_{b-1} , which is a descendant of c_i , c_b is comparable with c_i . If c_b is a descendant of c_i , then c_i is an ancestor of all $(c_k)_{a \leq k \leq b}$. If c_b is an ancestor of c_i , then c_b is an ancestor of all $(c_k)_{a \leq k \leq b}$. \blacktriangleleft

Consider the subtree of τ formed by all c_i and their ancestors. It is an infinite, but finitely-branching tree, thus it has an infinite branch by König's lemma. We first argue that there must be infinitely many c_i on that branch. Let $a \in \mathbb{N}$, let ν_a be the lowest ancestor of c_a on the branch. Let ν' be lower on the branch than ν_a , then ν' has some descendant c_b . Note that ν_a is the lowest common ancestor of c_a and c_b . We can assume $a < b$ (the case $b < a$ is symmetric), then by Claim 1 there is some $i \in \{a, \dots, b\}$ such that c_i is an ancestor of all $(c_k)_{a \leq k \leq b}$, and in particular of c_a and c_b . Further, as ν_a is the lowest common ancestor of c_a and c_b , c_i is an ancestor of ν_a and is thus on the branch. As a result, for all $a \in \mathbb{N}$ we can find $i \geq a$ such that c_i is on the branch.

We pick a sequence of c_i as follows: we start with the highest c_{i_0} on the branch, and then define $c_{i_{j+1}}$ as the highest c_i on the branch with $i > i_j$, for all j . By definition for all j we have that no c_i with $i > i_j$ is a strict ancestor of $c_{i_{j+1}}$.

As a consequence of Claim 1, there exists $i \in \{i_j + 1, \dots, i_{j+1}\}$ such that c_i is an ancestor of all $(c_k)_{i_j+1 \leq k \leq i_{j+1}}$. As noted above, as $i > i_j$ we cannot have c_i as a strict ancestor of $c_{i_{j+1}}$, hence $i = i_{j+1}$. As a result, $c_{i_{j+1}}$ is an ancestor of all $(c_k)_{i_j < k < i_{j+1}}$.

For the remaining of the proof we fix a consistent order labeling $(\prec_\nu)_\nu$ for τ .

Claim 2: For all $a < b$, if node ν is in the scope of both ℓ_a and ℓ_b , and if ν is ancestor of all $(c_k)_{a < k < b}$, then $x \succ_\nu y$, with x, y such that $L(\nu)(x) = \ell_a$ and $L(\nu)(y) = \ell_b$.

Proof. Let x, y be such that $L(\nu)(x) = \ell_a$ and $L(\nu)(y) = \ell_b$. We proceed by induction on $b - a$.

If $b = a + 1$ then $\ell_a \succ_H \ell_b$. Since we assume that $(\prec_\nu)_\nu$ is a consistent order labeling, by Lemma 19 and F4.3 of Lemma 12 we have $x \succ_\nu y$ as claimed.

If $b - a \geq 2$, by Claim 1, there exists $i \in \{a + 1, \dots, b - 1\}$ such that c_i is an ancestor of all $(c_k)_{a < k < b}$. In particular, c_i is an ancestor of c_{a+1} , itself an ancestor of μ_{a+1} , itself an ancestor of a node ν' with a **get** of ℓ_a . Recall that ν itself is an ancestor of c_i , by assumption. As the scope of a lock is a subtree, ℓ_a appears in all nodes between ν and ν' , thus in particular in c_i .

Moreover, μ_b is an ancestor of some node with a **get** of ℓ_{b-1} , which is a descendant of c_{b-1} , thus of c_i , hence μ_b and c_i are comparable. If μ_b is an ancestor of c_i , then as ν' is a descendant of c_i , ν' is also a descendant of μ_b , hence $\ell_a \succ_H \ell_b$. As a result, $x \succ_\nu y$ as the consistent order labeling satisfies F4.3 (Lemma 19). If μ_b is a descendant of c_i then as the scope of a lock is a subtree, ℓ_b appears in all nodes between ν and μ_b , thus in particular in c_i . We set $x', y', z' \in \text{Var}(c_i)$ such that $L(c_i)(x') = \ell_a$, $L(c_i)(y') = \ell_b$ and $L(c_i)(z') = \ell_i$ and by induction hypothesis we have $x \succ_{c_i} z$ and $z \succ_{c_i} y$ thus $x \succ_{c_i} y$ as \succ_{c_i} is total. Finally, as we have a consistent order labeling, $x \succ_\nu y$ holds as well. \blacktriangleleft

Recall that $c_{i_{j+1}}$ is an ancestor of all $(c_k)_{i_j < k < i_{j+1}}$. In particular, $c_{i_{j+1}}$ is an ancestor of c_{i_j+1} , thus of μ_{i_j+1} , itself an ancestor of some node ν' with a `get` of ℓ_{i_j} . Thus ℓ_{i_j} appears in $c_{i_{j+1}}$, because $c_{i_{j+1}}$ is between c_{i_j} and ν' .

Let x_j, y_j be such that $L(c_{i_{j+1}})(x_j) = \ell_{i_j}$ and $L(c_{i_{j+1}})(y_j) = \ell_{i_{j+1}}$. By Claim 2, we have $x_j >_\nu y_j$. The sequences $(c_{i_{j+1}})_{j>0}$, $(x_j)_{j>0}$ and $(y_j)_{j>0}$ thus form an infinite descending chain, proving the lemma. ◀

► **Proposition 21.** *For a given DLSS, there is a non-deterministic Büchi tree automaton $\widehat{\mathcal{B}}$ accepting exactly the limit configurations of process-fair runs of DLSS. The size of $\widehat{\mathcal{B}}$ is linear in the size of the DLSS and exponential in the maximal arity of the DLSS.*

Proof. Given a tree τ labeled with p, a, s the automaton $\widehat{\mathcal{B}}$ guesses a coloring \mathcal{C} , labeling H^s and an ordering labeling \mathcal{O} . It then checks if \mathcal{C} , H^s and \mathcal{O} satisfy all the consistency conditions. This automaton is a product of the following automata:

- \mathcal{B}_1 recognizing process-consistent trees,
- $\mathcal{B}_{\mathcal{C}}$ checking if the coloring is syntactically correct,
- \mathcal{B}_H checking if H^s is a syntactic H -labeling,
- \mathcal{B}_2 checking the conditions of Lemma 17,
- $\mathcal{B}_{\mathcal{O}}$ checking if \mathcal{O} is a consistent order labeling.
- \mathcal{B}_5 checking the absence of infinite descending chains (Lemma 20).

Apart from $\mathcal{B}_{\mathcal{C}}$ and \mathcal{B}_5 the other automata only check relations between a node and its children and some additional conditions local to a node. So they are automata with trivial acceptance conditions. Automaton $\mathcal{B}_{\mathcal{C}}$ needs a Büchi condition to check that \mathcal{C} is eventuality-consistent and recurrence-consistent. The number of labels is polynomial in the size of DLSS and exponential in the maximal arity as we have sets of predicates and orderings on variables as labels. Automaton \mathcal{B}_5 can be obtained by first constructing an automaton for its complement: one can easily define a non-deterministic Büchi automaton guessing a branch and following a sequence of variables along that branch witnessing an infinite decreasing sequence of locks. As it only needs to remember a pointer to one of the variables of a node, its number of states is the maximal arity of the DLSS. Thus we can complement it to get a non-deterministic Büchi automaton checking the absence of such sequence, of size exponential in the maximal arity of the DLSS, and polynomial in the alphabet (itself exponential in the arity and polynomial in the DLSS).

We need to check that τ is a fair limit configuration if and only if it is accepted by $\widehat{\mathcal{B}}$.

If τ is a limit configuration then it is process consistent, so it is accepted by \mathcal{B}_1 . Guessing \mathcal{C} to be semantically correct coloring ensures that $\mathcal{B}_{\mathcal{C}}$ accepts τ with this coloring (Lemma 14). As we have observed, given the coloring there is unique syntactic H -labeling, so \mathcal{B}_H can accept it. By Lemma 11, configuration τ satisfies properties F1-5. So τ is accepted by \mathcal{B}_2 . Finally, by Lemma 12, τ satisfies properties F4.1, F4.2, F4.3, so τ is accepted by $\mathcal{B}_{\mathcal{O}}$ thanks to Lemma 19. By Lemma 20, the automaton \mathcal{B}_5 accepts τ as well.

For the other direction suppose τ is accepted by $\widehat{\mathcal{B}}$. Thanks to Lemma 11 it is sufficient to check properties F1-5. Property F1 is verified by automaton \mathcal{B}_1 . Thanks to $\mathcal{B}_{\mathcal{C}}$ we know that the guessed coloring is syntactically correct. Then \mathcal{B}_2 ensures that τ satisfies F2 thanks to Lemma 17. Lemma 18 ensures that τ satisfies F3. Finally, automaton $\mathcal{B}_{\mathcal{O}}$ checks that the guessed orderings are a consistent order labeling. Hence, Lemma 19 guarantees that τ satisfies the conditions of Lemma 12 giving us F4. Finally, by Lemma 20 automaton \mathcal{B}_5 verifies condition F5. ◀

D Pushdown systems

► **Proposition 23.** *For a fixed d , the emptiness problem for right-resetting pushdown tree automata with a parity condition over ranks $\{1, \dots, d\}$ can be solved in PTIME.*

Proof. We consider the representative case of $d = 3$. Suppose we are given a right-resetting pushdown tree automaton $\mathcal{A} = (Q, \Sigma, \Gamma, q^0, \perp, \delta, \Omega)$.

The first step is to construct a pushdown word automaton $\mathcal{A}^l(G_1, G_2, G_3)$ depending on three sets of states $G_1, G_2, G_3 \subseteq Q$. The idea is that \mathcal{A}^l simulates the run of \mathcal{A} on the leftmost branch of a tree. When \mathcal{A} has a transition going both to the left and to the right then \mathcal{A}^l goes to the left and checks if the state going to the right is in an appropriate G_e . This means that \mathcal{A}^l works over the alphabet Σ^l that is the same as Σ but all letters from Σ_2 have rank 1 instead of 2. The states of $\mathcal{A}^l(G_1, G_2, G_3)$ are $Q \times \{1, 2, 3\}$ with the second component storing the maximal rank of a state seen so far on the run. The transitions of $\mathcal{A}^l(G_1, G_2, G_3)$ are defined according to the above description. We make precise only the case for a transition of \mathcal{A} of the form $\delta(q, a, \gamma) = ((q_l, \mathbf{instr}_l), (q_r, \mathbf{instr}_r))$. In this case, \mathcal{A}^l has a transition $\delta^l((q, e), a, \gamma) = ((q_l, \max(e, \Omega(q_l))), \mathbf{instr}_l)$ if $q_r \in G_{\max(e, \Omega(q_r))}$. Observe that \mathbf{instr}_r is necessarily **reset** as \mathcal{A} is right-resetting.

The next step is to observe that for given sets G_1, G_2, G_3 we can calculate in PTIME the set of states from which $\mathcal{A}^l(G_1, G_2, G_3)$ has an accepting run.

The last step is to compute the following fixpoint expression in the lattice of subsets of Q :

$$W = \text{LFP}X_3. \text{GFP}X_2. \text{LFP}X_1. P(X_1, X_2, X_3) \quad \text{where}$$

$$P(X_1, X_2, X_3) = \{q : \mathcal{A}^l(X_1, X_2, X_3) \text{ has an accepting run from } q\} .$$

Observe that $P : \mathcal{P}(Q)^3 \rightarrow \mathcal{P}(Q)$ is a monotone function over the lattice of subsets of Q . Computing W requires at most $|Q|^3$ computations of P for different triples of sets of states.

We claim that \mathcal{A} has an accepting run from a state q , if and only if, $q \in W$.

Let us look at the right-to-left direction of the claim. For this we recall how the least fixpoint is calculated. Consider any monotone function $R(X)$ over $\mathcal{P}(Q)$, and its least fixpoint $R^\omega = \text{LFP}X. R(X)$. This fixpoint can be computed by a sequence of approximations:

$$R^0 = \emptyset \quad R^{i+1} = R(R^i)$$

The sequence of R^i is increasing and $R^\omega = R^i$ for some $i \leq |Q|$.

Now we come back to our set W . Observe that $W = \text{LFP}X_3. R$ where $R(X) = \text{GFP}X_2. \text{LFP}X_1. P(X_1, X_2, X)$. As in the previous paragraph we can define

$$W^0 = \emptyset \quad \text{and} \quad W^{i+1} = \text{GFP}X_2. \text{LFP}X_1. P(X_1, X_2, W^i) .$$

So, if $q \in W$ then $q \in W^i$ for some i . Now observe that $W^i = \text{LFP}X_1. P(X_1, W^i, W^{i-1})$, since W^i is a fixpoint of $\text{GFP}X_2$. By similar reasoning we define

$$W^{i,0} = \emptyset \quad \text{and} \quad W^{i,j+1} = P(W^{i,j}, W^i, W^{i-1}) .$$

Now, $q \in W^i$ implies $q \in W^{i,j}$ for some j . We write $\text{sig}(q)$ for the lexicographically smallest (i, j) such that $q \in W^{i,j}$.

We examine what $\text{sig}(q) = (i, j)$ means. By definition $q \in P(W^{i,j-1}, W^i, W^{i-1})$, so there is an accepting run of $\mathcal{A}^l(W^{i,j-1}, W^i, W^{i,j-1})$ from q . Looking at the run of \mathcal{A} that \mathcal{A}^l simulates we can see that whenever this run branches to the right with some q' and e is the maximal rank on the run till this branching then

- if $e = 1$ then $q' \in W^{i,j-1}$,
- if $e = 2$ then $q' \in W^{i,k}$ for some k ,
- if $e = 3$ then $q' \in W^{i-1,k}$ for some k .

With this observation we can construct an accepting run of \mathcal{A} from every state in W . If $\text{sig}(q) = (i, j)$ then consider an accepting run of \mathcal{A}^l on the left path given by $P(W^{i,j-1}, W^i, W^{i-1})$. For every state branching to the right from this left path we recursively apply the same procedure. By construction, every path that is eventually a left path is accepting. A path branching right infinitely often is also accepting by the previous paragraph since signatures cannot go below 0. More precisely, the path cannot see 3 infinitely often because the first component of the signature decreases. If it sees 1 infinitely often then it needs to see also 2 infinitely often, because of the second component that decreases.

Let us now look at the left-to-right direction. Take an accepting run of \mathcal{A} from q^0 . We construct something that we call a skeleton tree of this run. As the nodes of the skeleton tree we take the root and all the nodes that are a right child; so these are the nodes of the tree of the form $(0^*1)^*$. The skeleton has an edge $\nu \xrightarrow{e} \nu 0^k 1$ if e is the maximal rank of a state of \mathcal{A} on the path from ν to $\nu 0^k 1$. Observe that a node can have infinitely many children. As we have started with an accepting run, every path in this skeleton tree satisfies the parity condition. In particular, for every node, on every path from this node there is a finite number of 3 edges. Thus, to every node ν we can assign an ordinal $\theta^3(\nu)$ such that if $\nu \xrightarrow{e} \nu'$ then $\theta^3(\nu') \leq \theta^3(\nu)$ and the inequality is strict if $e = 3$. It is also the case that on every path from ν there is a finite number of 1 edges before some 2 or 3 edge. This allows to define $\theta^1(\nu)$ with the property that if $\nu \xrightarrow{1} \nu'$ then $\theta^1(\nu') < \theta^1(\nu)$. Now we can show that for every node ν of the skeleton tree, if q is the state assigned to ν then $q \in W^{\theta^3(\nu), \theta^1(\nu)}$ for $W^{i,j}$ as defined in the computation of W (putting $W^{\theta^3} = W$ for every $\theta^3 > |Q|$, and $W^{\theta^3, \theta^1} = W^{\theta^3}$ for every $\theta^1 > |Q|$). The proof is by induction on the lexicographic order on $(\theta^3(\nu), \theta^1(\nu))$. ◀

E Lower bounds

In this section we show the two remaining lower bounds, namely EXPTIME-hardness for nested DLSS and undecidability for arbitrary ones.

► **Proposition 22.** *The DLSS verification problem for nested DLSS and Büchi objective is EXPTIME-hard. The result holds even if the Büchi objective refers to a single process.*

Proof. We show that the difficulty of the problem stems from the systems and not the specification, by proving that checking if some copy of a process has an infinite run is already EXPTIME-hard.

We provide a reduction from the emptiness problem for the intersection of top-down tree automata (over finite trees). Let $\mathcal{A}_1, \dots, \mathcal{A}_k$ be finite tree automata over a ranked alphabet A , with $\mathcal{A}_i = (S_i, \delta_i, s_{0,i}, F_i)$. We can assume that $A = \{a, b, c\}$ with a, b of arity 2 and c of arity 0, and that all automata only recognize trees with root labelled by a . We are going to construct a DLSS that simulates their computations simultaneously on the same tree T , by using locks to memorize their states.

The idea is to have a new process copy for each node of T . Each such copy uses the variables x_i^s and y_i^s for all $1 \leq i \leq k$ and $s \in S_i$, as well as the variables z_1, z_2 and t . Variable x_i^s is supposed to encode the information about the state of the parent node in the run of \mathcal{A}_i , while y_i^s will encode the state of \mathcal{A}_i at the current node.

We use processes p_0, q, ch, tk , plus processes $p_a^0, p_a^1, p_b^0, p_b^1$ and p_c . The processes are sketched in Figure 2. Process p_0 is the initial one. Process q is the root of T , and after

spawning its children it takes and releases the lock ℓ_t associated with variable t indefinitely. Lock ℓ_t will be shared with all processes, except for tk . The specification is that q should keep running forever. Equivalently, lock ℓ_t should be free infinitely often.

Whenever process ch is spawned, the purpose is to check if some lock is ultimately held. The first **get** in process ch corresponds to this test. If the lock associated with variable s is not taken, by **process-fairness** ch eventually takes it, and then it also takes ℓ_t (forever), preventing q to have an infinite run. As for tk , it simply takes the lock that it is given as argument.

The processes representing nodes of T are described next. Each node of T is represented by a copy of process p_a^i or p_b^i , depending on the letter a or b its parent is labelled with, and on whether it is a left or right child of its parent ($i = 0$ and $i = 1$, respectively). The root is represented by process q .

Process p_a^1 proceeds as follows: it chooses for each of its children a letter and spawns the associated processes, with each variable x_i^s of the child mapped to its own variable y_i^s , and all y_i^s of the child mapped to **new** (see actions sp in Figure 2). If the node represented by p_a^1 is a leaf, then p_a^1 spans a unique child p_c . For each $1 \leq i \leq k$, process p_a^1 then guesses the state s of its parent in \mathcal{A}_i and spawns a process ch in charge of checking the guess, so whether the lock associated with x_i^s is taken (see actions β in Figure 2). Finally, p_a^1 spawns a copy of process tk in charge of taking the lock associated with $y_i^{s'}$, where $s' = \delta_i(s, a, 1)$ is the state of the current node (see actions γ in Figure 2).

Processes p_a^0, p_b^0, p_b^1 are defined similarly. Process p_c simply guesses a final state s of its parent in each \mathcal{A}_i and spawns a copy of ch to verify the guesses.

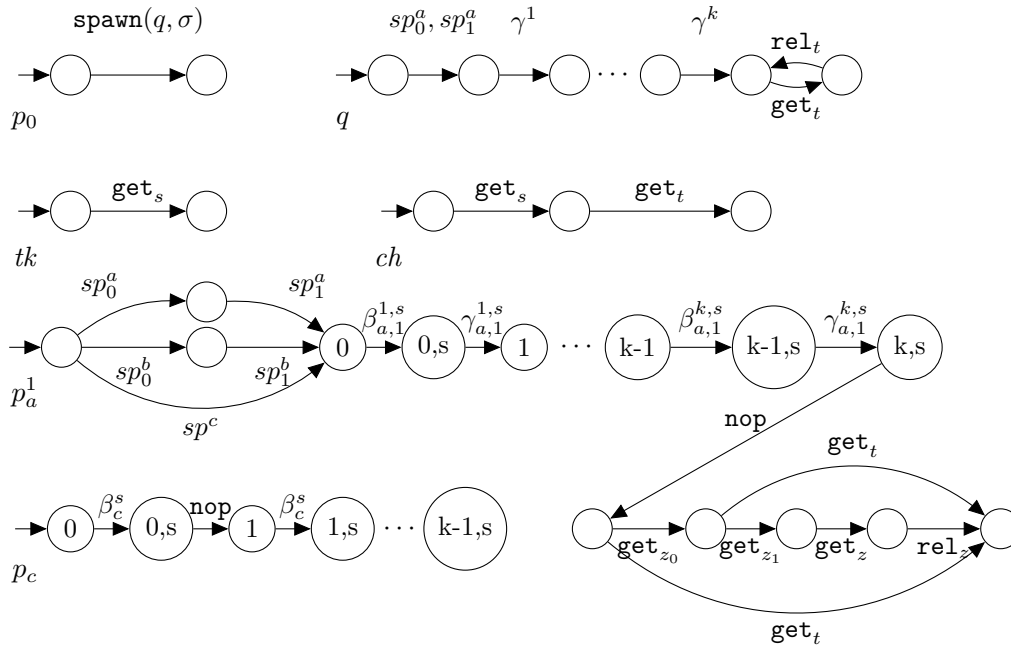
Process q is also in charge of representing the root, which we assumed to be labelled by a , hence all it has to do is spawn two children p_a^0 and p_a^1 with the variable assignments described below, and spawn copies of tk to take the locks associated with the variables $x_i^{s_0, i}$ (actions γ^i).

The problem is that we might end up producing an infinity of processes, representing a computation of the automata \mathcal{A}_i over an infinite tree. To avoid that, we use variables z_0, z_1 and z . Each copy of p_a^i and p_b^i , after doing all its other operations, takes z_0 and z_1 forever, and then takes and releases z . When spawning other processes representing nodes, each of p_a^i and p_b^i maps the z of the spawned process to its z_i (depending on whether it is its first or second spawned process), and the z_0, z_1 to **new**. We also add edges taking t forever that will eventually be executed, due to **process-fairness**, if z is taken forever before this process can take and release it. Hence all such processes must first acquire the locks of z_0, z_1 and then use the lock of z . This imposes that this part of the run is executed in the current process after it is executed in both children. This is only possible if the tree T is finite.

To sum up, if the \mathcal{A}_i all accept a finite tree T then we can construct a **process-fair** run of this DLSS by spawning all the processes representing nodes of T , then having all processes tk execute their **get**. There is no conflict as no two copies of tk take the same lock. All copies of process ch are then stuck in their first state. We then execute the actions on z, z_0, z_1 of each p_a^i, p_b^i in a bottom-up fashion, so that we can execute them all. Finally, we run q forever by having it take and release the lock ℓ_t indefinitely.

Conversely, if this DLSS has a **process-fair** run where q runs forever, then we can construct a tree T over a, b, c by taking q as root and defining the children of a node as the processes p_a^i, p_b^i spawned by the corresponding process. A node whose children are p_a^i is labelled a , one whose children are p_b^i is labelled b , and the leaves are labelled c .

We know that T is finite thanks to the previous argument involving the z, z_0, z_1 . We can associate with each node of T a state of each \mathcal{A}_i , inferred from the set of locks held so



■ **Figure 2** DLSS in Proposition 22

that we obtain runs of the automata \mathcal{A}_i over T (transitions are respected as otherwise some ch process would get a lock that is not taken by any tk and thus would eventually take ℓ_t forever). This run is furthermore accepting by definition of processes p_c .

The DLSS we constructed is clearly nested, which shows the claim. \blacktriangleleft

► **Theorem 5.** *The DLSS verification problem is undecidable. The result holds even if the DLSS is finite-state and every process uses at most 4 locks.*

Proof. The proof idea is to simulate an accepting run of TM M using n cells by spawning a chain of n processes, P_0, \dots, P_{n-1} . We assume that M accepts when the head is leftmost.

The initial process P_0 uses three locks, called a, b, c_1 , and acquires a, b before spawning P_1 . Process P_1 uses locks a, b, c_1 , plus a fresh lock c_2 . It acquires c_1 before spawning P_2 . More generally, process P_k ($1 \leq k < n-1$) uses locks a, b, c_k, c_{k+1} , and it acquires c_k before spawning the next process P_{k+1} . The last process P_{n-1} uses only three locks, a, b, c_{n-1} .

A configuration $(p, k, A_0 \dots A_{n-1})$ of the TM corresponds to each P_j storing A_j , with process P_k storing in addition state p . A TM step to the right, from cell k to $k+1$, needs to communicate the next state q .

In the following we denote the process that currently owns locks a and b , as “sender”. The notation S^+, S^- used below indicates that the sender tries to send the state to the right or left neighbour, respectively. Similarly, R^+, R^- indicates that a “receiver” is ready to receive from the right or the left neighbour, respectively.

Sending q from P_k to P_{k+1} is implemented by P_k using the following sequences of actions:

$$\begin{aligned}
 S_a^+ &= \text{rel}_a \text{get}_{c_{k+1}} \text{rel}_b \text{get}_a \text{rel}_{c_{k+1}} \text{get}_b \\
 S_b^+ &= \text{rel}_b \text{get}_{c_{k+1}} \text{rel}_a \text{get}_b \text{rel}_{c_{k+1}} \text{get}_a
 \end{aligned}$$

Process P_{k+1} (“receiver”) uses matching sequences:

$$\begin{aligned} R_a^- &= \text{get}_a \text{rel}_{c_{k+1}} \text{get}_b \text{rel}_a \text{get}_{c_{k+1}} \text{rel}_b \\ R_b^- &= \text{get}_b \text{rel}_{c_{k+1}} \text{get}_a \text{rel}_b \text{get}_{c_{k+1}} \text{rel}_a \end{aligned}$$

Suppose now that the sender P_k wants to send state q to receiver P_{k+1} . This will be done by P_k by trying to execute the sequence $(S_a^+)^q S_b^+$. Every process P_j with $j > k$ is ready to execute either R_a^- or R_b^- . Symmetrically, every process P_j with $j < k$ is ready to execute either R_a^+ or R_b^+ .

We show next that the DLSS deadlocks if P_k, P_{k+1} do not execute $(S_a^+)^q S_b^+$ and $(R_a^-)^q R_b^-$, resp., in lockstep manner:

Claim. Assume that P_k owns $\{a, b\}$, every $P_j, j < k$, owns c_{j+1} , and every $P_j, j > k$, owns c_j . Moreover, P_k wants to send a to P_{j+1} . Then either P_k, P_{k+1} execute S_a^+ and R_a^- , resp., in lockstep manner, or all processes deadlock.

Proof of claim. Process P_k is the only process who can start, since all other processes wait for acquiring either a or b .

After releasing a , process P_k needs c_{k+1} . It can only proceed and take c_{k+1} if P_{k+1} starts executing R_a^- , taking a and releasing c_{k+1} . Then P_k releases b , and waits to get back a . If b is taken by another process than the receiver, say $P_j, j \neq k+1$, then P_j will release its lock $c \neq c_{j+1}$, and c is now the only available lock. Lock a will never become available because P_{j+1} will not release it, so all processes deadlock.

Assume that P_{j+1} takes b , and releases a . If a is taken by another process than the sender, say $P_j, j \neq k$, then P_j will release its lock $c \neq c_{j+1}$, and c is now the only available lock. Lock a will never become available because P_{j+1} does not release b , so all processes deadlock.

Assume that P_j takes a back. Then it releases c_{j+1} , which can be taken only by P_{j+1} , who releases also b . If b is taken by another process than the sender, say $P_j, j \neq k$, then P_j will release its lock $c \neq c_{j+1}$, and c is now the only available lock. Lock b will never become available because P_j does not release a anymore. Once again, all processes deadlock. ◀

We conclude the proof by noting that P_0 reaches a final state of M if and only if M accepts. ◀