arXiv:2210.07914v1 [cs.FL] 14 Oct 2022

Model-checking lock-sharing systems against regular constraints

Corto Mascle

LaBRI, Université de Bordeaux
corto.mascle@labri.fr
https://corto-mascle.github.io/

Abstract. We study the verification of distributed systems where processes are finite automata with access to a shared pool of locks. We consider objectives that are boolean combinations of local regular constraints. We show that the problem, PSPACE-complete in general, falls in NP with the right assumptions on the system. We use restrictions on the number of locks a process can access and the order in which locks can be released. We provide tight complexity bounds, as well as a subcase of interest that can be solved in PTIME.

Keywords: Distributed systems · Locks · Model-checking

1 Introduction

Concurrent programs often prove more challenging to verify than sequential ones, as the state space explodes easily, unless processes follow very closely what the others are doing or have completely decorrelated executions. Verification of such programs can be traced back to the work of Taylor [12], and has been the subject of a variety of approaches, which reflect the numerous possible modelisations of distributed systems. Looking for an error trace is typically PSPACE-hard when processes are finite-state systems, i.e., the cost of exploring an exponential number of configurations. The reason is that most models of concurrent programs, be it with rendez-vous, message passing, or shared variables, can encode the problem of deciding whether a set of deterministic finite automata have a common accepted word. This is the case for instance for the classical model of Zielonka automata [13].

We study lock-sharing systems (LSS for short), a simple model for concurrent programs using mutexes. Processes have access to a pool of locks. Each process is represented by an automaton whose transitions acquire and release locks. Locks restrict the behaviours of the system, as a process cannot take a lock already held by another process. Similar systems were considered by Gupta, Kahlon and Ivancić in [9], with only two processes, each being a pushdown system. They proved that the verification of regular constraints relating local runs was undecidable, and provided a fine-grained analysis of the decidable cases in that paper and later ones [7,8]. They also showed that detecting deadlocks is decidable

under some restrictions. This exact approach contrasts with other ones, such as in [1] or [11], which tackle more general systems but use approximations of the set of possible runs. Chapter 18 of [3] gives an overview of those works.

We consider the verification problem for the model studied in [6]. That paper focused of synthesizing local strategies to avoid global deadlocks. Here we consider a much larger family of properties: boolean combinations of local regular properties. Unlike [6] we do not discuss the synthesis problem, but the model-checking problem.

In this work we present an analysis of restrictions on lock-sharing systems that suffice in order to obtain more tractable complexities than PSPACE. We mainly focus on two restrictions, 2LSS and nested LSS. The first one demands that each process only accesses two different locks, the second one that each process takes and releases locks as if they were stored in a stack: they can only release the lock taken latest. Several works already showed the interest of the nested restriction to obtain tractable verification problems, see for instance [2,9]. The contribution of [2] consists in an NP algorithm (and an implementation) for detecting deadlocks (more specifically, configurations where some subset of processes is blocked as they all try to acquire locks held by other processes of that subset) in concurrent programs. They use a syntax for programs that can be translated to what we call sound nested exclusive LSS in this paper. As for the systems with two locks per process, they can already exhibit a variety of behaviours. Dijkstra's famous dining philosophers problem matches this constraint. These restrictions have a common point: local runs can be summarised in short descriptions, called *patterns*. Patterns contain enough information to determine whether local runs can be interleaved to form a global run. Some form of patterns for finite runs of nested systems, called acquisition history, was already considered in [9], but was only focused on systems with two processes and with no considerations of complexity. In [6] patterns are defined on finite runs and used to compute local strategies to prevent deadlocks in LSS. We show that we can extend the techniques to handle much larger classes of specifications, in the framework of verification.

In order to do this, we extend the notion of patterns to infinite runs and provide necessary and sufficient conditions on patterns to represent runs that can be interleaved into a (fair) global run. This allows us to verify the system against local specifications by first guessing for each process a pattern, checking compatibility of these patterns and then checking individually in each process the existence of a bad run with the corresponding pattern. Thus we avoid exploring the product of all processes.

This approach yields NP algorithms for the verification of (boolean combinations of) local specifications for 2LSS and nested LSS. With an additional constraint, called exclusiveness, we even obtain a PTIME algorithm for some specific objective called process deadlock, which requires one given process to be forever unable to advance after some point in the run.

We provide matching lower bounds for these results. In general our problem is PSPACE-complete, even with a bounded number of locks per process. It is NP-

complete in the nested case, even with exclusiveness, for some weak objectives (the hardness proof solves a question left open by the authors in [2]), and a bounded number of locks per process. As for 2LSS, the problem is NP-complete as well. Furthermore, those lower bounds make little use of the specification, proving that the complexities are in some sense inherent to the systems.

Overview In Section 2 we recall some definitions and give some intuition about the global framework. Then in Section 3 we generalise the notion of patterns that was used in [6] (Definition 14), after which we present the results that we are able to obtain through this technique: In Section 4 we discuss a particular specification, for which the problem can be solved in PTIME for exclusive systems, and provide an NP lower bound when we do not assume exclusiveness. In Section 5 we prove the PSPACE-completeness of the general problem and contrast it with its NP-completeness in the 2LSS case. Finally, in Section 6, we prove that the verification of nested systems is NP-complete, with a very robust lower bound, that survives exclusiveness, weak objectives, and even a bounded number of locks.

2 Definitions

First we recall the definition of a lock-sharing system

Definition 1 (Lock-sharing system). *Let Proc be a finite set of processes.*

A lock-sharing system (LSS for short) $\mathcal{S} = ((\mathcal{A}_p)_{p \in Proc}, T, op)$ is given by a family of transition systems, a set T of locks, and a function op described below.

Each transition system $\mathcal{A}_p$ is given as a tuple $(S_p, \Sigma_p, \delta_p, init_p)$ with S_p a finite set of states, $init_p$ the initial state, Σ_p a finite alphabet and $\delta_p : S_p \times \Sigma_p \to S_p$ a ***partial*** *function. We require that the Σ_p are pairwise disjoint, and define $\Sigma = \bigcup_{p \in Proc} \Sigma_p$.*

Consider a set of operations $Op(T) = \{\mathtt{get}_t, \mathtt{rel}_t, nop \mid t \in T\}$. The function $op : \Sigma \to Op(T)$ associates with each letter of Σ an operation on locks. For all $p \in Proc$ we define $T_p = \{t \in T \mid \exists a \in \Sigma_p, op(a) = \mathtt{get}_t\}$ the set of locks p may acquire.

A 2LSS is an LSS where every T_p has two elements.

Remark 1. In [6], the transition functions δ_p output a pair (s, op) with a state and an operation. Here we will assume without loss of generality that the operation of a transition is determined by its action; we can use $\Sigma \times Op(T)$ as our alphabet instead of just Σ and thus explicitly describe the sequence of operations in the actions.

We fix an LSS $\mathcal{S} = ((\mathcal{A}_p)_{p \in Proc}, T, op)$ for the rest of this section.

A *local configuration* of process p is a state from S_p together with the locks p currently owns: $(s, B) \in S_p \times 2^{T_p}$. The initial configuration of p is $(init_p, \emptyset)$, namely the initial state with no locks. A transition between configurations $(s, B) \xrightarrow{a} (s', B')$ exists when $\delta_p(s, a) = s'$ and one of the following holds:

– $op(a) = nop$ and $B = B'$;
– $op(a) = \texttt{get}_t$, $t \notin B$ and $B' = B \cup \{t\}$;
– $op(a) = \texttt{rel}_t$, $t \in B$, and $B' = B \setminus \{t\}$.

A *local run* $a_1 a_2 \cdots$ of $\mathcal{A}_p$ is defined as a finite or infinite sequence over Σ_p such that there exists a sequence of local configurations $(init_p, \emptyset) = (s_0, B_0) \xrightarrow{a_1}_p (s_1, B_1) \xrightarrow{a_2}_p \cdots$ (we will specify explicitly when we talk about local runs that do not start in the initial configuration).

We say that a finite local run $w_p = a_1 \cdots a_n$ is *neutral* if for all $1 \leq i \leq n$ such that $op(a_i) = \texttt{get}_t$ for some $t \in T$, there exists $j > i$ such that $op(a_j) = \texttt{rel}_t$. Equivalently, the configuration obtained after executing w_p is in $S_p \times \{\emptyset\}$.

A *global configuration* is a tuple of local configurations $C = (s_p, B_p)_{p \in Proc}$ provided the sets B_p are pairwise disjoint: $B_p \cap B_q = \emptyset$ for $p \neq q$. This is because **a lock can be taken by at most one process at a time**. The initial configuration is the tuple of initial configurations of all processes.

Runs of such systems are *asynchronous*, with transitions between two consecutive configurations done by a single process: $C \xrightarrow{(p,a)} C'$ if $(s_p, B_p) \xrightarrow{a}_p (s'_p, B'_p)$ and $(s_q, B_q) = (s'_q, B'_q)$ for every $q \neq p$. A global run is a sequence of transitions between global configurations. Since our systems are deterministic we usually identify a global run with the sequence of transition labels. A global run w *determines a local run* of each process: $w|_p$ is the subsequence of p's actions in w. We also say that $w|_p$ is the projection of w on p.

In what follows we will assume that each process keeps track in its state of the set of locks it owns. Note that this assumption does not compromise the complexity results provided there is a bound on the number of locks a process can access: the number of states is then multiplied by a constant factor.

Definition 2. *A process of an LSS is* sound *if its transition system $\mathcal{A}_p$ keeps track of the set of locks it has in its states. Formally, let $\mathcal{A}_p = (S_p, \delta_p, init_p)$, p is sound if there exists a function* $\mathrm{owns}_p : S_p \to 2^{T_p}$ *such that:*

– *for all local runs $w = a_1 a_2 \cdots a_n$ ending in a state s, we have $(init_p, \emptyset) \xrightarrow{a_1} \cdots \xrightarrow{a_n} (s, \mathrm{owns}_p(s))$.*
– *for all states $s \in S_p$, there is no outgoing transition of s that acquires a lock in $\mathrm{owns}_p(s)$ or releases a lock that is not in $\mathrm{owns}_p(s)$.*

An LSS is sound if all its processes are.

Note that this property can be easily checked on a given LSS: it suffices to set $owns(init_p)$ to $\emptyset$, apply a DFS to compute candidates for $owns(s)$ for all states, and then check consistency of $owns$ with respect to each transition.

We want to be able to define deadlocks in terms of languages of runs. To this end, we have to restrict our attention to process-fair runs, in which every process is either blocked after some point or executes an action infinitely many times. This is often called strong fairness in the literature. This way if a process stops doing anything after some point in a run, it means it is blocked.

Definition 3. *A run w is called process-fair if for all $p \in Proc$, either w contains infinitely many actions of Σ_p, or there is a point after which no action of p can ever be executed at any moment in the run.*

We say that a process-fair run yields a global deadlock if it is finite, i.e., at some point there are no actions that can be executed in any of the processes, and the system cannot advance any more. Note that a process-fair run is finite if and only if it yields a global deadlock.

We say that a process-fair run yields a partial deadlock if its projection on one of the Σ_p is finite, i.e., after some point one of the processes is never able to execute any action.

In all that follows we will have to work with finite and infinite words simultaneously as LSS executions may be finite or infinite. We will use a dummy letter $\square$, and finite runs will be padded with an infinite suffix $\square^\omega$ so that we can express objectives as languages of infinite words.

From now on we will write $u^{\square}$ for the padded version of a word u, i.e.,

$$u^{\square} = \begin{cases} u & \text{if } u \text{ is infinite} \\ u\square^\omega & \text{if } u \text{ is finite.} \end{cases}$$

We will now define the set of properties we want to verify. This class of objectives is inspired by Emerson-Lei automata, introduced in [4], which we will use for several proofs of upper bounds. Note that we will use non-deterministic Emerson-Lei automata, while our objectives are expressed using deterministic automata.

Definition 4. *An Emerson-Lei automaton (ELA for short) is a tuple $\mathcal{A} = (S, \Sigma, \Delta, init, \varphi)$ with S a finite set of states, Σ a finite alphabet, $\Delta : S \times \Sigma \times S$ a transition function, $init \in S$ the initial state and φ a boolean formula over variables $\{inf_s \mid s \in S\}$.*

Such an automaton recognises a language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$. An infinite word w is accepted if there is a run of w in $\mathcal{A}$ such that φ is satisfied by the valuation evaluating inf_s to $\top$ if and only if s appears infinitely often in the run.

Our objectives are defined in a similar fashion, but with one automaton per process and a single formula expressing a condition on which states (among the ones of all automata) are seen infinitely often.

Definition 5. *A regular objective is a pair $((\mathcal{B}_p)_{p\in Proc}, \varphi)$ such that each $\mathcal{B}_p$ is a deterministic automaton with a set of states $S_{\mathcal{B}_p}$ over the alphabet $\Sigma_p \cup \{\square\}$, and φ is a boolean formula over the set of variables $\{inf_{p,s} \mid p \in Proc, s \in S_{\mathcal{B}_p}\}$.*

Let w be a ***process-fair*** *run, and for each p let w_p be its projection on Σ_p. We say that w satisfies a regular objective $((\mathcal{B}_p)_{p\in Proc}, \varphi)$ if φ is satisfied by the valuation evaluating $inf_{p,s}$ to $\top$ if and only if the unique run of $\mathcal{L}(\mathcal{B}_p)$ on $w_p^{\square}$ goes through s infinitely many times.*

We argue that these specifications are quite expressive and at the same time allow us to stay in reasonably low complexity classes.

Regular objectives are expressive. They can express properties such as reachability (with local or global configurations) or safety, as well as properties related to deadlocks, such as partial deadlock or global deadlock: As we focus on process-fair runs, a local projection of a run is finite if and only if the corresponding process is blocked at some point and has no available action for the rest of the run. Hence, we can express for instance a global deadlock with an objective requiring the local run of every process p to be finite.

Moreover, the flexibility of boolean formulas allows us to relate configurations between processes: say each process has to decide between 0 and 1, then we can express agreement by demanding that they all select 0 or all 1.

Regular objectives are furthermore closed under boolean combinations. They can be complemented by simply taking the negation of the formula φ, and intersected in polynomial time by taking the product automaton for each process and adapting the formula.

Complexity blows up quickly with more expressive objectives Regular objectives only restrict the shape of local runs without any requirement on their interleaving. Restrictions on interleavings would lead to PSPACE-hardness very quickly. As we will see in Section 5, as soon as we can have a system where processes are required to synchronize in some way, we also obtain PSPACE-hardness.

Objectives that are sensitive to interleavings of local runs can be used to test the emptiness of the intersection of languages of n DFAs, even without any locks. We can take $Proc = \{p_1, \dots, p_n\}$ and $\Sigma_p = \{a_p, b_p, c_p\}$ for all p and ask for a global run in $(a_{p_1} \cdots a_{p_n} + b_{p_1} \cdots b_{p_n})^*(c_{p_1} \cdots c_{p_n})^\omega$ in the LSS constructed from those DFAs.

In this work we study the problem of finding a run satisfying some given regular objective.

Definition 6. *We define the* regular verification problem *as:*
Input: a sound LSS $\mathcal{S}$ and a regular objective $RO = ((\mathcal{B}_p)_{p \in Proc}, \varphi)$
Output: Is there a process-fair run of $\mathcal{S}$ satisfying RO?

Note that we define the problem existentially: we are looking for a bad run, hence the given objective should express the set of runs that we want to avoid. We use this formulation as it simplifies a bit our proofs, and as regular objectives are easy to complement.

We also define the problem in the particular case of process deadlocks: Here, we ask whether there is a run in which some given process p is eventually blocked forever. We define it as our standard example of a "simple" objective. We will show that we can decide it in PTIME in a particular case, and we will use it for complexity lower bounds, thus showing that those complexities are already inherent to the systems.

Definition 7. *We define the* process deadlock problem *as:*
Input: a sound LSS $\mathcal{S}$ and a process p.
Output: Is there a process-fair run of $\mathcal{S}$ whose projection on p is finite?

As our last definition in this part, we introduce exclusive LSS, in which a process that can acquire a lock cannot do any other operation from the same state.

Definition 8 (*Exclusive*). *A process is exclusive if its transition system $\mathcal{A}_p$ is such that for all states s, if s has an outgoing transition acquiring some lock t, then all other outgoing transitions acquire that same lock t. An LSS is exclusive if all its processes are.*

3 Patterns for 2LSS

In this section we define patterns for 2LSS. These are summaries of bounded size of the operations executed during a run, which contain enough information to tell if local runs can be combined into a global one. Let us first define a couple of useful functions over local runs.

Definition 9. *Given a finite local run $w_p = a_0 \cdots a_n$ of a process p, we define $\textsc{Owns}(w_p)$ as the set of locks p holds after executing $op(a_0) \cdots op(a_n)$.*

We extend the function $\textsc{Owns}$ to infinite runs by setting $\textsc{Owns}(a_1 a_2 \cdots)$ as the set of locks kept indefinitely by p after some point. Formally, we define $\textsc{Owns}(a_1 a_2 \cdots) = \bigcup_{i \in \mathbb{N}} \bigcap_{j > i} \textsc{Owns}(a_0 \cdots a_j)$.

The trace of an infinite run $w_p = a_1 a_2 \cdots$, denoted by $tr(w_p)$, is the infinite word $A_0 A_1 \cdots \in (2^T)^\omega$ with $A_i = \textsc{Owns}(a_1 \cdots a_i)$ the set of locks held by p after executing the first i actions of w_p.

We also define $\textsc{Inf}(w)$ as the set of sets of locks that p owns infinitely often when executing w_p:

$$\textsc{Inf}_p(a_1 a_2 \cdots) = \{A \subseteq T_p \mid A = \textsc{Owns}(a_1 \cdots a_i) \text{ for infinitely many } i\}$$

We start with patterns of *finite runs* as in [6]. We redefine them here with a formalism adapted to our purpose.

Definition 10 (*Finitary patterns*). *Finitary patterns are defined for finite local runs of a 2LSS. Let p be a process, $T_p = \{t_1, t_2\}$ its locks. Let $w = a_1 a_2 \cdots a_n$ be a finite local run of p. The pattern of w is defined as the set $\textsc{Owns}(w)$ along with an information on its strength:*

- *If $\textsc{Owns}(w) = \{t_1\}$ (resp. $\{t_2\}$) and the last operation on locks in w is $\texttt{rel}_{t_2}$ (resp. $\texttt{rel}_{t_1}$) then we say that w has a strong pattern, denoted as $\overline{\textsc{Owns}(w)}$*
- *Otherwise we say that w has a weak pattern, denoted $\overset{- - - -}{\textsc{Owns}(w)}$*

In [6] the global deadlock problem was studied, so only patterns of finite runs were of interest. We define patterns of infinite runs as we have to account for the runs of processes that do not get blocked.

Definition 11 (*Infinitary patterns*). *Let w be an infinite local run of a process p accessing locks $T_p = \{t_1, t_2\}$. Let $tr(w) = A_0 A_1 \cdots \in (2^{T_p})^\omega$. The pattern of w is given by $\textsc{Inf}(w)$ along with an information on its strength:*

- *We say that w has a* strong pattern $\overline{\mathrm{INF}(w)}$ *when* $tr(w) \in (2^{\{t_1,t_2\}})^* \{t_1, t_2\}\{t_1\}^\omega$ *(the process has both locks at some point, releases one of them and does not do any other operation on locks afterwards). Observe that in this case* $\mathrm{INF}(w) = \{\{t_1\}\}$.
- *Otherwise, w has the* weak pattern $\overline{\mathrm{INF}}(w)$.

We say that w is switching *if $\emptyset \notin \mathrm{INF}(w)$ and $\mathrm{OWNS}(w) = \emptyset$. This means that eventually, p never releases both locks, but releases each one infinitely often. In particular, $T_p \in \mathrm{INF}(w)$.*

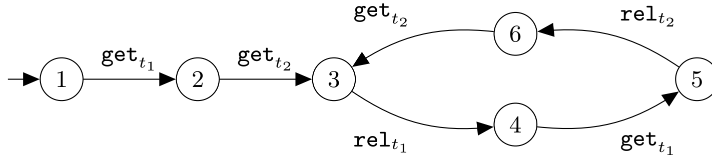

Fig. 1. A process with a single infinite run whose pattern is $\overline{\{\{t_1\}, \{t_2\}, \{t_1, t_2\}\}}$ (switching). It also has finite runs of patterns $\overline{\emptyset}$, $\overline{\{t_1\}}$, $\overline{\{t_1, t_2\}}$, $\overline{\{t_2\}}$ and $\overline{\{t_1\}}$

Example 1. Consider the process p displayed in Figure 1. It has a single infinite run, which eventually cycles between states 4 (in which it has only t_2), 6 (in which it has only t_1), and 3 and 5 (in which it has both), hence it has as infinitary pattern $\overline{\{\{t_1\}, \{t_2\}, \{t_1, t_2\}\}}$, i.e., it is switching.

This system is sound, i.e., for all finite runs w, $\mathrm{OWNS}(w)$ is determined by its end state. Furthermore the pattern is strong if $\mathrm{OWNS}(w)$ is a singleton and the last operation in w is a `rel`, which is also determined by the end state in this system. We can infer that all runs ending in state 1 have pattern $\overline{\emptyset}$, in state 2 $\overline{\{t_1\}}$, in state 3 and 5 $\overline{\{t_1, t_2\}}$, in state 4 $\overline{\{t_2\}}$, and in state 6 $\overline{\{t_1\}}$.

Note that for each of the patterns defined above, the set of runs matching that pattern is a regular language. Although this fact is clear, we formalise it in the following lemma. This allows us to give explicitly (very small) automata recognising those languages, and we think that the proof of this lemma may help the reader understand the relation between finitary and infinitary patterns.

Lemma 1. *Let $p \in Proc$ be a process.*

For each pattern **pat** *described in Definitions 10 and 11 we can define a (deterministic) ELA $\mathcal{A}^p_{\mathbf{pat}}$ (with 12 states) over the alphabet $\Sigma_p \cup \{\square\}$ recognizing the language consisting of $w^{\square}$ with w a local run of p whose pattern is* **pat**.

Proof. For all patterns we use the same states and transitions, which keep track of the finitary patterns. They are described in Figure 2 with $T_p = \{t_1, t_2\}$. We labelled edges with operations instead of actions as the transitions of an action

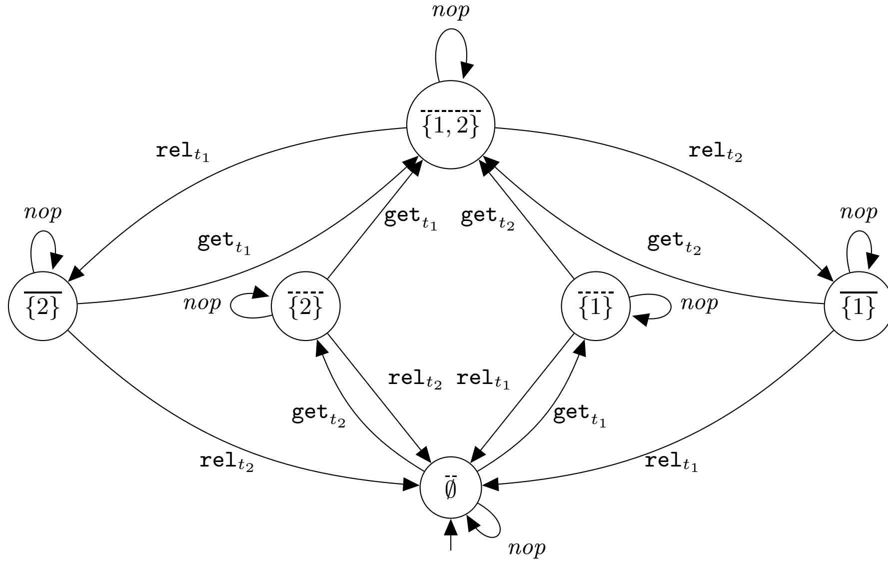

Fig. 2. The automaton structure for pattern recognition. Every state s has a transition $\square$ to its copy $s^{\square}$, with a $\square$ self-loop, which is not displayed.

a depend only on $op(a)$ here. For each state s we have a transition labelled by $\square$ leading to a copy $s^{\square}$ of that state with only a self-loop labelled by $\square$. The desired pattern is then expressed as an Emerson-Lei condition to obtain an ELA.

For a finitary pattern **pat** the formula $\mathrm{inf}_{\mathbf{pat}^{\square}}$ suffices, to indicate that the automaton read a run of pattern **pat** and then only $\square$.

For an infinitary pattern such that $\textsc{Inf}(w_p) = \{\{t\}\}$ for some t, we have to distinguish strong and weak. If the pattern is strong we use the formula $\mathrm{inf}_{\overline{\{t\}}} \wedge \bigwedge_{s \neq \overline{\{t\}}} \neg \mathrm{inf}_s$ saying that we stay in state $\overline{\{t\}}$ indefinitely, otherwise we use $\mathrm{inf}_{\overset{\text{-\,-}}{\{t\}}} \bigwedge_{s \neq \overset{\text{-\,-}}{\{t\}}} \neg \mathrm{inf}_s$ saying that we stay in $\overset{\text{-\,-}}{\{t\}}$ indefinitely.

Otherwise we only have to check the set of sets of locks owned infinitely often, hence we use the formula $\bigwedge_{J \in \textsc{Inf}(w_p)} \varphi_J \wedge \bigwedge_{J \notin \textsc{Inf}(w_p)} \neg \varphi_J$, where φ_J is $\mathrm{inf}_{\overset{\text{-\,-}}{J}} \vee \mathrm{inf}_{\overline{J}}$ if J is a singleton, and $\mathrm{inf}_{\overset{\text{-\,-}}{J}}$ otherwise.

We now present the key proposition on patterns for 2LSS. It states when a set of local runs can be combined into a global run. Note that the criterion depends only on the patterns of the local runs and the last states they reach. This will be the crucial ingredient in the proof that the regular verification problem is in NP for 2LSS.

Proposition 1. *Consider a family of local runs $(w_p)_{p \in Proc}$ (each of them can be finite or infinite).*

We write G_{Inf} for the undirected graph whose vertices are locks and with a p-labelled edge between t_1 and t_2 whenever $T_p = \{t_1, t_2\}$ and w_p is switching.

For all finite w_p let s_p be its end state. We define the set of locks that can be acquired from s_p: $\textsc{Blocks}_p = \{t \mid \exists a, op(a) = \mathtt{get}_t \text{ and } \delta_p(s_p, a) \text{ is defined}\}$.

The local runs $(w_p)_{p\in Proc}$ can be scheduled into a process-fair global run if and only if the following conditions are all satisfied.

1. *If w_p is finite then all outgoing transitions from its end state s_p acquire a lock.*
2. *All sets $\textsc{Owns}(w_p)$ are disjoint.*
3. *All $\textsc{Blocks}_p$ are included in $\bigcup_{p\in Proc} \textsc{Owns}(w_p)$.*
4. *The intersection $\textsc{Owns}(w_p) \cap \bigcup_{J\in\textsc{Inf}(w_q)} J$ is empty for all pairs of processes $p \neq q$ such that w_q is infinite.*
5. *There is a total order $\leq$ on locks such that for all p whose run w_p has a strong pattern $\overline{\{t_1\}}$ (for finite runs) or $\overline{\{\{t_1\}\}}$ (for infinite runs) we have $t_1 \leq t_2$; where t_2 is the other lock used by p.*
6. *There is no process p such that (1) $\{t, t'\} \in \textsc{Inf}(w_p)$ and (2) there is a path in G_{Inf} between t and t' not using a p-labelled edge. In particular, G_{Inf} is acyclic.*

Proof. **⇒:** We start with the left-to-right implication. Let $(w_p)_{p\in Proc}$ be a family of local runs, suppose they can be scheduled into a process-fair global run w.

For all finite local runs w_p, as w is process-fair, after some point p cannot ever execute any action.

As a consequence, s_p (the state reached after executing w_p) cannot have any outgoing transition executing `rel` or *nop*, as those can always be executed. The locks of $\textsc{Blocks}_p$ are never free after some point, as otherwise p would be enabled infinitely often on the run, so the run would not be process-fair. This shows condition 1.

All finite runs w_p stop while holding the locks of $\textsc{Owns}(w_p)$. All infinite w_p eventually acquire and never release the locks of their $\textsc{Owns}(w_p)$. Hence the $\textsc{Owns}(w_p)$ sets need to be pairwise disjoint, proving condition 2.

Furthermore, if a lock is not in some $\textsc{Owns}(w_p)$ then it is free infinitely often, and thus cannot be in $\textsc{Blocks}_p$ for any p, as w is process-fair. This proves condition 3.

All locks of $\bigcup_{J\in\textsc{Inf}(w_q)} J$ are held by q infinitely often, hence they cannot be in any $\textsc{Owns}(w_p)$ with $p \neq q$, which shows condition 4.

If a run w_p of a process p using locks t_1, t_2 has a pattern $\overline{\{t_1\}}$ or $\overline{\{\{t_1\}\}}$ then the last operation on t_1 (when p acquires it for the last time) is followed by at least one operation on t_2 in the run w. We satisfy condition 5 by setting $\leq$ as an order on locks such that $t \leq t'$ whenever t is only acquired finitely many times and there is an operation on t' after the last operation on t in w .

We demonstrate condition 6 by contradiction. Say there exist such locks and process, i.e., there exist $t = t_1, \ldots, t_n = t'$ and $p_1, \ldots, p_{n-1}$ without p such that for all $1 \leq i < n$, p_i accesses t_i and t_{i+1} and w_{p_i} is switching. Then all p_i are always holding a lock after some point.

As $\{t_n, t_1\} \in \textsc{Inf}(w_p)$, this means that p holds t_n and t_1 simultaneously infinitely often. Whenever that happens, processes $p_1, \dots, p_{n-1}$ have to share the remaining $(n-2)$ locks, hence one of them holds no lock, contradicting the fact that $\emptyset \notin \textsc{Inf}(w_{p_i})$ for all i.

$\Leftarrow$: For the other direction, suppose $(w_p)_{p \in Proc}$ satisfies all the conditions of the list. We construct a process-fair global run whose local projections are the w_p.

To do so, we will construct a sequence of finite runs $v_0, v_1, \dots$ such that $v_0 v_1 \cdots$ is such a global run.

We will ensure that the following property is satisfied for all $i \in \mathbb{N}$:

$$\begin{aligned} &\text{For all processes } p, \text{ after executing } v_0 \cdots v_i, \\ &\text{if } \textsc{Owns}(w_p) \in \textsc{Inf}(w_p) \text{ then } \textsc{Owns}((v_0 \cdots v_i)|_p) = \textsc{Owns}(w_p) \\ &\text{otherwise } w_p \text{ is switching and } p \text{ holds one lock.} \end{aligned} \tag{1}$$

We will also make sure that all p with an infinite w_p execute an action in infinitely many v_i.

The first run v_0 has to be constructed separately as we require it to satisfy some extra conditions. We construct v_0 such that for all p:

- If w_p is finite then $v_0|_p = w_p$.
- If w_p is infinite then $w_p = v_0|_p u_p$ with u_p such that for every prefix u'_p of u_p, $\textsc{Owns}(v_0|_p u'_p) \in \textsc{Inf}(w_p)$. Furthermore if $\emptyset \in \textsc{Inf}(w_p)$ then $\textsc{Owns}(v_0|_p) = \emptyset$. In other words, we execute a prefix of each infinite run such that what follows matches its asymptotic behaviour.

Construction of v_0

- First, for all infinite w_p such that $\emptyset \in \textsc{Inf}(w_p)$, there exist arbitrarily large finite prefixes of w_p ending with p having no lock. Hence we can select one of those prefixes $v_0|_p$, large enough for p to never hold a set of locks not in $\textsc{Inf}(w_p)$ later in the run. We execute all such $v_0|_p$ at the start. All locks are free afterwards.
- We then execute for all other p with weak patterns, their maximal prefix ending with p having no lock. All locks are still free.
- Then we execute all w_p with strong patterns, in increasing order according to $\leq$ (see condition 5) on the locks t_p such that $\textsc{Owns}(w_p) = \{t_p\}$. We execute in full the finite ones, while for the infinite ones we execute a prefix $v_0|_p$ such that in the end p owns only t_p and never acquires the other lock afterwards (recall that $\textsc{Inf}(w_p) = \{\{t_p\}\}$ in that case). Say we executed some of those local runs, let p be a process such that w_p has a strong pattern accessing locks $t_1 \leq t_2$, say we want to execute $v_0|_p$. By condition 2, all $\textsc{Owns}(w_p)$ are disjoint, hence there is no other process q with $\textsc{Owns}(w_q) = \{t_1\}$. The only locks that are not free at that point are the t such that $t < t_1$ and $\textsc{Owns}(w_q) = \{t\}$ for some q with a strong pattern. Therefore, both t_1 and t_2 are free, and $v_0|_p$ can be executed. In the end the aforementioned locks t_p are taken and all others are free.

- Then we consider the finite w_p with non-empty $\textsc{Owns}(w_p)$ and weak patterns. For those, we can execute the rest of the run (we already executed the maximal prefix leading to them holding no lock), as all they do is take the locks in $\textsc{Owns}(w_p)$, which are free by conditions 2 and 4.
- For the infinite w_p with non-empty $\textsc{Owns}(w_p)$ and weak patterns, there are two possibilities:
  - The first is that p eventually keeps the same set of locks forever and never executes any more operations on locks. Then its trace $tr(w_p)$ is of the form either $(2^{T_p})^*\emptyset\{t_1\}^\omega$ or $(2^{T_p})^*\{t_1, t_2\}^\omega$. In that case clearly we can just execute the run until we reach a point after which p only ever owns $\textsc{Owns}(w_p)$ forever. We can do this as all locks taken so far are either in $\textsc{Owns}(w_q)$ for some q with finite w_q or are in an element of some $\textsc{Inf}(w_q)$ for some q. Thus all locks from those $\textsc{Owns}(w_p)$ are free by conditions 2 and 4.
  - The other possibility is that $tr(w_p) \in (2^{T_p})^*(\{t_1\}^*\{t_1, t_2\})^\omega$ with $\{t_1, t_2\} = T_p$ and $\textsc{Owns}(w_p) = \{t_1\}$. This happens if p ultimately holds one lock forever and acquires and releases the other one infinitely many times. At that point all locks that are taken are in some $\textsc{Owns}(w_p)$, thus by condition 4 both locks of p are free. Hence we can execute enough steps of w_p to reach a point at which p holds only t_1 and will only hold sets of locks of $\textsc{Inf}(w_p)$ afterwards.
- Finally we consider the infinite switching runs w_p. All those processes must have $T_p \in \textsc{Inf}(w_p)$, hence by condition 4 all their locks are free. By condition 6, G_{Inf} is acyclic. We can therefore pick one of those processes p and a lock t such that no other such process accesses t. We execute w_p until p only owns t will only own sets of locks of $\textsc{Inf}(w_p)$ afterwards. All locks of the other such p are still free, hence we can iterate that step until we executed a prefix of each of those p.

We have constructed a finite run v_0 whose projection $v_0|_p$ on Σ_p is such that if w_p is finite then $w_p = v_0|_p$ and if w_p is infinite then $v_0|_p$ is a prefix of w_p such that all local configurations seen later in the run are in $\textsc{Inf}(w_p)$. Moreover v_0 satisfies property 1.

We now construct the remaining parts of the run. If all w_p are finite then v_0 proves the lemma (we can set all other v_i as ε). Otherwise we must describe the rest of the process-fair global run whose projections are the w_p. We start with a small construction that will help us define the v_i.

Suppose we constructed $v_0, \dots, v_i$ so that property 1 is satisfied for all $j \leq i$. Now suppose some lock t_0 is not in any $\textsc{Owns}(w_p)$ and is not free after executing $v_0 \cdots v_i$. Then there exists a switching run w_{p_1} with $t_0 \in T_{p_1}$.

Let t_1 be the other lock of p_1, say it is not free. By property 1, p_1 holds only one lock and thus does not hold t_1. By condition 4 t_1 cannot be in some $\textsc{Owns}(w_p)$, thus, again by property 1, as t_1 is not free, there exists a switching w_{p_2} such that $t_1 \in T_{p_2}$. Let t_2 be the other lock of p_2.

We construct this way a sequence of processes $p_1, p_2, \dots$ and of locks $t_0, t_1, \dots$ such that $T_{p_j} = \{t_{j-1}, t_j\}$ and w_{p_j} is switching for all j. This sequence cannot be infinite as each p_j labels an edge in G_{Inf}, which is finite and acyclic.

Hence there exists k such that t_k is free. We can therefore execute w_{p_k} until p_k holds t_k and not t_{k-1}, then execute $w_{p_{k-1}}$ until p_{k-1} holds t_{k-1} and not t_{k-2}, and so on until t_1 is free.

Hence if a lock t is not in any $\textsc{Owns}(w_p)$ but is not free after executing $v_0 \cdots v_i$ then we can prolong the prefix run so that t is free and some lock from the same connected component in G_{Inf} is not. For all such t and i we name this prolongation of the run $\pi_{t,i}$.

Say we already constructed $v_0, \ldots, v_i$, and that property 1 is satisfied for all $j \leq i$. We construct v_{i+1}. Let p be either a process that never executed an action, or if there are no such processes, the process whose last action in $v_0 \cdots v_i$ is the earliest.

We prolong the current run so as to execute some actions of w_p. If the next action of w_p applies an operation *nop* we can execute it right away. The next action cannot execute a `rel` operation: After executing v_0 all processes with infinite w_p only own sets of locks of $\textsc{Inf}(w_p)$. By property 1, if w_p is switching then after executing $v_0 \cdots v_i$ the process p holds one lock and will not release it as it would be left with no lock and $\emptyset \notin \textsc{Inf}(w_p)$. If p is not switching then after executing $v_0 \cdots v_i$ it holds $\textsc{Owns}(w_p)$ and cannot release any lock as it keeps those forever.

Hence we are left with the case where the next action of p acquires a lock t. If t is not free we apply $\pi_{t,i}$ to free it (and block another lock of the same connected component of G_{Inf}). Note that after executing $\pi_{t,i}$ all processes with switching runs still hold one lock, and the others have not moved.

- If w_p is switching then p was already holding a lock t', and it can then take t and then run w_p until it holds only one lock again, thus respecting property 1.
- Otherwise p was holding $\textsc{Owns}(w_p)$ (by Property 1) and we have to let him take t and then continue until p holds exactly $\textsc{Owns}(w_p)$ again.
  - If we can do it right away we do so.
  - Otherwise it means that p needs its other lock t' to reach that next step, and that this lock is taken. More precisely, it means that $\textsc{Owns}(w_p) = \emptyset$ and $\emptyset, \{t, t'\} \in \textsc{Inf}(w_p)$.
    As $\{t, t'\} \in \textsc{Inf}(w_p)$, by condition 6, t and t' are not in the same connected component of G_{Inf}. Hence we can execute $\pi_{t',i}$, without locking t back, as $\pi_{t,i}$ and $\pi_{t',i}$ use disjoint sets of locks and processes.
    This ensures that both t and t' are free, which allows p to take t and proceed to the next point at which it holds $\emptyset$.

  In both cases we end up in a configuration where p owns $\textsc{Owns}_p$, all processes with switching runs hold exactly one lock, and the other processes did not move, thus respecting property 1.

We have constructed v_{i+1}, ensuring that property 1 is satisfied for $i+1$. Furthermore $v_{i+1}|_p$ is non-empty for p a process with infinite w_p which either never executed anything before or executed its last action the earliest. This

ensures that all p with infinite w_p execute infinitely many actions in $v_0 v_1 \cdots$. Hence we obtain a global run $v = v_0 v_1 \cdots$ such that for all p we have $v|_p = w_p$.

Furthermore we ensured that v is process-fair as all p with finite runs are blocked: all such w_p lead to a state from which only locks of $\textsc{Blocks}_p$ can be taken, by condition 1, and by condition 3 all $\textsc{Blocks}_p$ are included in $\bigcup_{p \in Proc} \textsc{Owns}(w_p)$, the set of locks that are never free from some point on.

As a result, there exists a process-fair run whose local projections are the w_p, proving the right-to-left implication.

Example 2. Consider two processes p and q with the same transition system, displayed in Figure 1. We can prove that all process-fair runs of those two will end in a global deadlock using patterns.

Say there is a run whose projection on one of them (say, p) is infinite, then that projection w_p has pattern $\overline{\{\{t_1\}, \{t_2\}, \{t_1, t_2\}\}}$, meaning it will take and release both locks infinitely often without releasing both at the same time after some point.

Then q does not have a compatible run: It cannot have the same infinitary pattern by condition 5 of Proposition 1. Furthermore, by condition 3 it cannot have any finitary pattern besides $\overline{\emptyset}$. However, its only run with that pattern is the empty one, which ends in the initial state, from which there is a transition executing $\texttt{get}_{t_1}$, meaning that by condition 2 we should have $t_1 \in \textsc{Owns}(w_p)$, which is not the case. Thus there cannot be such a run.

4 Process deadlocks

While the complexity lower bounds presented in this work are robust to many restrictions, we can still find some interesting properties that can be verified on some systems in polynomial time. In [6] (Lemma 22 and Proposition 24) it was proven that verifying if a “locally live” strategy on a 2LSS allows a run leading to a global deadlock (in which all processes are blocked) can be done in polynomial time. An immediate consequence of this is that verifying if a sound 2LSS in which all states have at least one outgoing transition has a run yielding a global deadlock can be done in PTIME.

From the results in [6] we can also extract the NP-completeness of finding a global deadlock in a 2LSS when we allow states with no outgoing transitions.

4.1 A PTIME algorithm for exclusive 2LSS

Here we are interested in a different problem, the process deadlock problem. We provide a polynomial-time algorithm based on a key lemma that lists the different ways a process can be blocked.

Let $\mathcal{S}$ be a sound exclusive 2LSS and p a process of $\mathcal{S}$.

Lemma 2. *Let $(s_p, B_p)_{p \in Proc}$ be a global configuration and for each process p let u'_p be a local run starting in (s_p, B_p) and such that u'_p is either infinite or leads to a state with no outgoing transitions.*

There exists a process-fair global run w from $(s_p, B_p)_{p\in Proc}$ such that for all p its projection w_p on Σ_p is a prefix of u'_p.

Proof. We construct w by iterating the following step: For each p we set $u'_p = v_p w_p$ with v_p the prefix of u'_p executed so far. We select uniformly at random a process $p \in Proc$. If it can execute the first action of w_p then we let it do so, otherwise we do nothing.

We iterate this procedure indefinitely. This produces a (possibly finite) global run of the system such that its local projections are prefixes of the u'_p. We prove that it is process-fair.

Let $p \in Proc$, assume that p has an available action at infinitely many steps. As our LSS is exclusive, whenever p has an available action and is in some state s, either all outgoing transitions are executing an operation nop or `rel` (and thus can all be executed as the system is sound), or they all acquire the same lock t (as the system is exclusive). Hence if one outgoing transition can be executed , they all can and thus in particular the next action of u'_p is available. As a result, p can execute the next action of u_p at infinitely many steps, and thus will progress infinitely many times in u_p.

In conclusion, with this procedure we either reach a global deadlock, or we always have an available action, implying that at least one process will be able to progress infinitely many times and that the resulting run u is infinite. In the latter case, all processes that can execute an action at infinitely many steps of the run will do so, proving that the run is process-fair.

Definition 12. *Define the graph G whose vertices are locks and with an edge $t_1 \xrightarrow{p} t_2$ if and only if the process p has a local run w_p ending in a state where all outgoing transitions acquire t_2 and such that* $\textsc{Owns}(w_p) = \{t_1\}$. *We say that w_p witnesses the edge $t_1 \xrightarrow{p} t_2$.*

Lemma 3. *For all $p \in Proc$, if there is a p-labelled edge $t_1 \xrightarrow{p} t_2$ in G then either $t_1 \xrightarrow{p} t_2$ is witnessed by a run with a weak pattern or its reverse $t_2 \xrightarrow{p} t_1$ is in G and is witnessed by a run with a weak pattern.*

Proof. As p has an edge $t_1 \xrightarrow{p} t_2$ in G, there is a local run which acquires both locks of p at the same time. Let w_p be such a run of minimal length. The last operation in w_p must be a `get`, by minimality, hence w_p is of the form $w'_p a$ with $op(a) = \texttt{get}_t$ for some $t \in \{t_1, t_2\}$. Furthermore, suppose the last operation in w'_p besides nop is a `rel`, then there is a previous configuration in w_p in which p holds both of its locks, contradicting the minimality of w_p. Hence w'_p has a weak pattern, and it leads to a state where p may acquire t, thus has to acquire t as the system is exclusive. Furthermore p is then holding its other lock, therefore w'_p witnesses an edge in G.

Lemma 4. *If p has a reachable transition acquiring some lock t and there is a path from t to a cycle in G then there is a process-fair global run with a finite projection on p.*

Proof. Let $t = t_0 \xrightarrow{p_1} t_1 \xrightarrow{p_2} \cdots \xrightarrow{p_k} t_k$ be such a path in G and let $t_k = t'_1 \xrightarrow{p'_1} \cdots t'_n \xrightarrow{p'_n} t'_{n+1} = t'_1 = t_k$ be such a cycle.

For all $1 \leq i \leq k$ we choose a run w_i witnessing $t_{i-1} \xrightarrow{p_i} t_i$. Similarly for all $1 \leq j \leq n$ we choose a run w'_j witnessing $t'_j \xrightarrow{p'_i} t'_{j+1}$, and we choose it so that it has a weak pattern whenever possible.

Case 1: If there exists j such that w'_j has a weak pattern, then we proceed as follows: Let $w'_j = u_j v_j$ so that u_j is the maximal neutral prefix of w_j. We execute u_j, leaving all locks free. Let m be the maximal index such that $t_m \in \{t'_1, \cdots, t'_n\}$. We execute all w_i in increasing order for $1 \leq i \leq m$.

Then we execute $w'_{j+1} \cdots w'_n w'_1 \cdots w'_{j-1}$ and then v_j. Then we end up in a configuration where all p_i with $i \leq m$ are holding t_{i-1} and need t_i to advance, while all p'_i are holding t'_i and need t'_{i+1} to advance. As $t_j \in \{t'_1, \ldots, t'_n\}$, all those processes are blocked, and in particular $t = t_0$ is held by a process which will never release it.

As p has a reachable transition taking t, we can define w_p as a shortest run that ends in a state where some outgoing transitions takes a lock of $\{t_0, \ldots, t_m, t'_1, \ldots, t'_n\}$. By minimality this run can be executed, as all other locks are free. By exclusiveness, it reaches a state where all transitions take the same non-free lock.

By Lemma 2 we can extend this run into a process-fair one, whose projection on p can only be w_p, as p will never be able to advance further.

Case 2: Now suppose there is no j such that w'_j has a weak pattern, then as we took all w'_j with weak patterns whenever possible, it means there is no local run with a weak pattern witnessing any of the $t'_j \xrightarrow{p'_j} t'_{j+1}$. We can then apply Lemma 3 to show that the reverse cycle $t'_1 = t'_{n+1} \xrightarrow{p'_n} t'_n \cdots \xrightarrow{p'_1} t'_1$ exists in G and all its edges are witnessed by runs with weak patterns. Hence we can apply the arguments from the previous case using this cycle to conclude.

Lemma 5. *If p has a reachable transition acquiring some lock t and there is a path in G from t to some t' such that there is a process q with an infinite local run w_q acquiring t' and never releasing it (i.e., such that $t' \in$ Owns(w_q)), then there is a process-fair global run with a finite projection on p.*

Proof. Let $t = t_0 \xrightarrow{p_1} t_1 \xrightarrow{p_2} \cdots \xrightarrow{p_k} t_k = t'$ be the shortest path from t to t'. Let w_p be a local run of p acquiring t at some point, either infinite or leading to a state with no outgoing transition. For each $1 \leq i \leq k$ we select a local run w_i of p_i witnessing $t_{i-1} \xrightarrow{p_i} t_i$. Furthermore we select those w_i with weak patterns whenever possible. Let t_q be the other lock used by q besides t', and let w_q be an infinite run of q in which t' is eventually taken and never released. We can decompose w_q as $u_q v_q$ where u_q is the largest neutral prefix of w_q. We distinguish several cases:

Case 1: $t_q \notin \{t_0, \ldots, t_k\}$, or t_q is not used in v_q. Then we can execute u_q, leaving all locks free, then $w_1 \cdots w_k$, which can be done as the execution of $w_1 \cdots w_i$ leaves $t_i, \ldots, t_k$ free and thus w_{i+1} can be executed. Let $v_q = v'_q v''_q$ with v'_q a prefix of v_q large enough so that t' is held by q and never released later. Then as no t_i is used in v_q, we can execute v'_q. Let w be the run constructed so far. Then by Lemma 2 we can construct a process-fair run w' starting in the last configuration of w whose projection on q is a prefix of v''_q (thus t_k is never released and thus neither are $t_0, \ldots, t_{k-1}$) and whose projection on p is a prefix of w_p (and thus finite as w_p tries to acquire t, which is never free). As a consequence, ww' is a process-fair run whose projection on p is finite.

Case 2: $t_q = t_j$ for some $0 \leq j \leq k$ and w_q acquires t_j at some point and never releases it. Then we apply the same reasoning as in the previous case for the path $t = t_0 \xrightarrow{p_1} \cdots \xrightarrow{p_j} t_j$.

Case 3: $t_q = t_j$ for some $0 \leq j \leq k$ and t_j is used in v_q but not kept indefinitely.

Subcase 3.1: there is an edge $t' \xrightarrow{q} t_j$. Then we have a path from t to a cycle $t_j \xrightarrow{p_{j+1}} \cdots \xrightarrow{p_k} t' \xrightarrow{q} t_j$. Hence by Lemma 4, there is a process-fair global run with a finite projection on p.

Subcase 3.2: One of the runs w_i has a weak pattern $\overline{\{t_{i-1}\}}$. We decompose w_i as $u_i v_i$ with u_i its largest neutral prefix. Then we execute u_i, then $w_{i+1} \cdots w_k$. After that we execute a prefix w'_q of w_q such that at the end q holds only t', and does not release it later. This prefix exists as q never keeps t_j indefinitely in w_q. We decompose w_q as $w'_q w''_q$. Then we execute $w_1 \cdots w_{i-1} w'_i$. All those runs can be executed as before executing each $w_{i'}$ both locks of $p_{i'}$ are free, and before executing v_i, t_{i-1} is free, which is all that is needed to execute v_i as w_i has a weak pattern. Let w be the run constructed so far. Then by Lemma 2 we can construct a process-fair run from the configuration reached by w whose projection on q is a prefix of w''_q and whose projection on p is a prefix of w_p. As a consequence, $t' = t_k$ is never released in w''_q and thus neither are $t_0, \ldots, t_{k-1}$. As w_p tries to take $t = t_0$ at some point, its prefix executed in w' is finite. Hence ww' is a process-fair run with a finite projection on p.

Subcase 3.3: There is no edge $t' \xrightarrow{q} t_j$ and all w_i have strong patterns. When executing the v_q part of w_q, q holds a lock at all times, and holds t_j at some point and t' at some point, hence it has to have both at the same time at some moment. Hence there is a moment at which q holds one of the locks and is about to get the other. As the system is exclusive, it means all its available transitions take that lock. Hence there is an edge $t' \xrightarrow{q} t_j$ or $t_j \xrightarrow{q} t'$ in the graph. As we assumed that there is no edge $t' \xrightarrow{q} t_j$, there is one $t_j \xrightarrow{q} t'$. Furthermore, as we selected the w_i so that they had weak patterns whenever possible, it means that for all i there is no run with a weak pattern witnessing $t_{i-1} \xrightarrow{p_i} t_i$. By Lemma 3 this means that there are edges $t_k \xrightarrow{p_k} t_{k-1} \xrightarrow{p_{k-1}} \cdots \xrightarrow{p_{j+1}} t_j$. With the edge

$t' \xrightarrow{q} t_j$, we obtain a cycle in G with a path from t to it. By Lemma 4, there is a process-fair global run with a finite projection on p.

This concludes our case distinction, proving the lemma.

Lemma 6. *If p has a reachable transition acquiring some lock t and there is a path in G from t to some t' such that there is a process q with a local run w_q with $t' \in \text{OWNS}(w_q)$ and going to a state with no outgoing transitions, then there is a process-fair global run with a finite projection on p.*

Proof. Let s_q be the state reached by w_q, we add a self-loop on it with a fresh letter #. As there are no other outgoing transitions from s_q this does not break the exclusiveness. It does not change G either. Then $w_q\#^\omega$ is an infinite run acquiring t' and never releasing it.

Hence by Lemma 5, there is a process-fair run w in this new system whose projection on p is finite. Let h be the morphism such that $h(\#) = \varepsilon$ and $h(a) = a$ for all other letters a. Then $h(w)$ is a process-fair run of the original system: it is a run as # does not change the configuration, meaning that all actions of $h(w)$ can be executed. For the same reason, if a process p' other than q only has finitely many actions in $h(w)$, then the same is true in w, thus there is a point after which no configuration allows p' to move in w, and thus in $h(w)$ as well. As for q, either it only executes # from some point on, meaning it has reached s_q and will be immobilised in $h(w)$, or it never executes any #, in which case $h(w) = w$ and it follows the same configurations in both.

Lemma 7. *There is a process-fair run whose projection on p is finite if and only if there is a local run w_p of p leading to a state where all outgoing transitions take some lock t and either*

1. *p has a local run leading to a state with no outgoing transitions.*
2. *or there is a path from t to a cycle in G*
3. *or there is a path in G from t to some lock t' and there is a process q with a local run w_q with an infinitary pattern with $t' \in \text{OWNS}(w_q)$.*
4. *or there is a path in G from t to some lock t' and there is a process q with a local run w_q such that $t' \in \text{OWNS}(w_q)$ and leading to a state with no outgoing transitions.*

Proof. We start with the left-to-right implication: Say there is a run w whose projection on p is finite. For each process $p' \in \mathit{Proc}$ let $w_{p'}$ be its local run.

Then w_p has to end in a state where all available transitions acquire a lock t. If there are no transitions at all, condition 1 is satisfied. If there is at least one such transition, then t is held forever by some other process p_1.

We construct a path $t = t_0 \xrightarrow{p_0} t_1 \xrightarrow{p_1} \cdots$ in G so that all t_i are held indefinitely by some process after some point in the run. Say we already constructed those up to i.

There is a process p_i holding t_i indefinitely. If w_{p_i} is infinite, then condition 3 is satisfied. Otherwise, w_{p_i} is finite, and with a finitary pattern such that $t_i \in \text{OWNS}(w_{p_i})$.

If this local run ends up in a state with no outgoing transition then condition 2 is satisfied, otherwise it must have no choice but to acquire some lock t'_{i+1}. Hence we construct an infinite path $t'_0 \xrightarrow{p'_0} t'_1 \xrightarrow{p'_1} \cdots$ in G.

The set of processes is finite, hence there exist $i < j$ such that $t_i = t_j$, meaning we have reached a cycle. Thus condition 4 is satisfied.

For the other direction, suppose there exists t as in the statement of the lemma, so that one of the conditions is satisfied.

If condition 1 is satisfied, then we have a finite run w_p leading to a state with no outgoing transition. We execute it and then prolong it into a global process-fair run by choosing a process uniformly at random and executing one of its available actions if there is any (similarly to the proof of 2). We obtain a process-fair run in which p only has finitely many actions. If condition 2 is satisfied then we have the result by Lemma 4. If condition 3 is satisfied then we have the result by Lemma 6. If condition 4 is satisfied then we have the result by Lemma 5.

To conclude the proof of Proposition 2, by Lemma 7, we only have to check the four conditions listed in its statement. Here is our algorithm:

We start by looking, in the transition system of process p, for a reachable local state with no outgoing transition. If there is one, we accept.

Then we compute all pairs (q, t) such that either there is an infinite run of process q keeping t indefinitely from some point on or there is a run w_q with $t \in \text{Owns}(w_q)$ leading to a state with no outgoing transitions. As our system is sound, the set of locks a process has is determined by its state. Let $owns_p$ be the function described in Definition 2. Then we compute all pairs (t, t') such that some process p' has a reachable state s with $owns_{p'}(s) = \{t\}$ and all outgoing transitions of s acquiring t'. We obtain the edges of G.

For both locks of p', we check that there is a reachable transition acquiring it, and there is a path in G to either a cycle or to a t from one of the pairs (q, t) computed above. If it is the case for one of them, we accept, otherwise we reject. This can all be done in polynomial time, proving the proposition.

Proposition 2. *The process deadlock problem is in* PTIME *for sound exclusive 2LSS.*

4.2 NP-hardness for general 2LSS

By contrast, when we lift the exclusive requirement, the problem becomes NP-hard (and NP-complete, as we will see later).

Proposition 3. *The process deadlock problem is* NP*-hard for sound 2LSS.*

Proof. We reduce from the 3SAT problem. We use a set of variables $x_1, \dots, x_n$. Let $\varphi = \bigwedge_{i=1}^{m} C_i$ with for each i, $C_i = \ell_i^1 \vee \ell_i^2 \vee \ell_i^3$ with $\ell_i^j \in \{x_k, \neg x_k \mid 1 \le k \le n\}$.

We construct a system with processes $Proc = \{p, p'\} \cup \{p_i^C \mid 1 \le i \le m\} \cup \{p_{i,j}^\ell \mid 1 \le i \le m, 1 \le j \le 3\} \cup \{p_k^x \mid 1 \le k \le n\}$. We also use locks $T =$

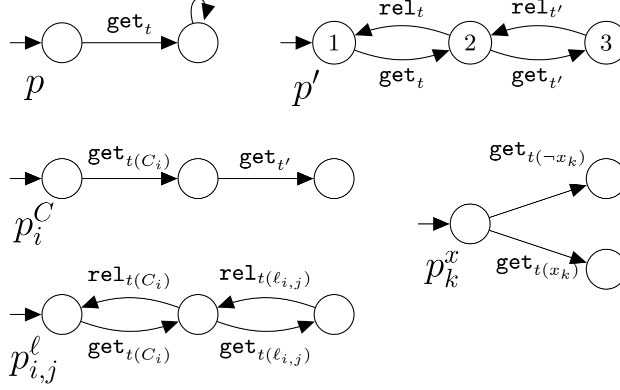

Fig. 3. Processes for the reduction in Proposition 3

$\{t, t'\} \cup \{t(C_i) \mid 1 \leq i \leq m\} \cup \{t(x_k), t(\neg x_k) \mid 1 \leq k \leq n\}$. The transition systems of these process are described in Figure 3.

In order to block process p we need to block it in its first state by having another process keep t forever. As a matter of fact, the only other process accessing t is p'. As a consequence, a process-fair run blocks p if and only if p' eventually keeps t forever.

Consider such a run w. Then eventually p' has to stop visiting its state 1. Furthermore, as w is process-fair, p' can never stay indefinitely in one of the other two states as it is always possible to execute a `rel` action. Hence p' goes through states 2 and 3 infinitely many times, meaning it takes and releases t' infinitely often.

This implies that none of the p_i^C keep t' indefinitely, which is only possible if all the $t(C_i)$ are taken and never released by other processes (if some $t(C_i)$ is free infinitely often, as w is process-fair p_i^C has to take $t(C_i)$ at some point, and then t' cannot be free infinitely often as p_i^C would have to take it eventually).

As a consequence, for each C_i there has to be a ℓ_i^j such that $p_{i,j}^\ell$ keeps $t(C_i)$ forever, which is only possible if $t(\ell_{i,j})$ is free infinitely often.

This means that the process p_k^x (with x_k the variable appearing in $\ell_{i,j}$) must have taken the lock associated with the negation of ℓ_i^j (it cannot stay in its initial state as the run is process-fair and ℓ_i^j is free infinitely often).

In conclusion, exactly one of $t(x_k), t(\neg x_k)$ is free infinitely often for each k, and for each clause C_i there is a literal in C_i whose lock is free infinitely often. Thus the valuation mapping each x_k to $\top$ if x_k is free infinitely often and $\bot$ otherwise satisfies φ.

Now suppose φ is satisfied by some valuation ν. We construct the following run: First of all for all k process p_k^x takes $t(\neg x_k)$ if $\nu(x_k) = \top$ and $t(x_k)$ otherwise. Then for each i we select some j_i such that ν satisfies ℓ_i^j and have process p_{i,j_i}^ℓ take C_i. Finally, p' takes t.

We then repeat the following steps indefinitely: one by one each p^{ℓ}_{i,j_i} takes $t(\ell_i^j)$ and releases it, then p' takes and releases t'. This is all possible as all $t(\ell_i^{j_i})$ are free (those ℓ_i^j are satisfied by ν hence the corresponding p_k^x took their negations) and so is t' (none of the p_i^C ever moves thus they do not take t').

This run is process-fair as the processes that are eventually blocked are the p_k^x (which end up in states with no outgoing transitions), the p_i^C (which need C_i to advance, but those locks are never released) and p (which needs t to move on, but t is kept forever by p'). This concludes our reduction.

5 Regular objectives

5.1 The problem is Pspace-complete in general

In order to justify our approach, we prove that the general verification of LSS against regular objectives is Pspace-complete, even with strong restrictions on the system.

Proposition 4. *The regular verification problem is* Pspace-*complete for LSS in general.* Pspace-*hardness already holds for the process deadlock problem for sound exclusive LSS even with a fixed number of locks per process.*

The Pspace upper bound is easy to obtain: It suffices to guess a state s_p in each $\mathcal{A}_p$ and s'_p in each $\mathcal{B}_p$, and then guess a sequence of letters in Σ while keeping track of the states reached by that sequence in the $\mathcal{A}_p$ and $\mathcal{B}_p$.

If we reach a configuration with each $\mathcal{A}_p$ in state s_p and each $\mathcal{B}_p$ in s'_p, we start memorising the set of visited states in each $\mathcal{B}_p$. If we reach that configuration again, we stop and accept if and only if the set of visited states in the $\mathcal{B}_p$ satisfies φ. This comes down to guessing an ultimately periodic run in the global system and checking that it satisfies the objective.

The difficulty is to obtain the Pspace-hardness with a fixed number of locks per process. To do so we reduce the emptiness problem for the intersection of a set of deterministic automata.

Without loss of generality we will assume that there are at least two automata, that they are all over alphabet $\{0, 1\}$, and that their languages are all included in $11(00+01)^*$: we can always apply a small transformation to each automaton so that, if its language was $\mathcal{L}$, it becomes $11h(\mathcal{L})$ with h the morphism mapping 0 to 00 and 1 to 01. The intersection of those languages is empty if and only if the intersection of the original languages was empty.

Let $\mathcal{A}_1, \cdots, \mathcal{A}_n$ (with $n \geq 2$) be automata, with, for each $1 \leq i \leq n$, $\mathcal{A}_i = (S_i, \{0, 1\}, \delta_i, init_i, F_i)$. We construct a sound exclusive LSS $\mathcal{S}$ as follows:

For each $1 \leq i \leq n$ we have a process p_i which is in charge of simulating $\mathcal{A}_i$. The set of locks is $T = \{0_i, 1_i, key_i \mid 1 \leq i \leq n\}$. For all i, p_i accesses locks $0_i, 1_i, key_i$, as well as $0_{i+1}, 1_{i+1}, key_{i+1}$ if $i \leq n-1$ and $0_1, 1_1$ if $i = n$. Thus a process uses at most 6 locks in total.

For all $1 \leq i \leq n$ and t accessed by p_i, we have two actions $\mathtt{get}^i_t$ and $\mathtt{rel}^i_t$, with which p_i acquires and releases lock t, as well as actions nop^i and end^i with no effect on locks.

In the proof the following local sequences will be important:

$$\textsc{send}(0) = \mathtt{rel}^n_{0_n}\mathtt{get}^n_{0_0}\mathtt{rel}^n_{1_n}\mathtt{get}^n_{1_0}\mathtt{rel}^n_{0_0}\mathtt{get}^n_{0_n}\mathtt{rel}^n_{1_0}\mathtt{get}^n_{1_n}$$

$$\textsc{rec}_i(0) = nop^i\mathtt{get}^i_{0_{i+1}}\mathtt{rel}^i_{0_i}\mathtt{get}^i_{1_{i+1}}\mathtt{rel}^i_{1_i}\mathtt{get}^i_{0_i}\mathtt{rel}^i_{0_{i+1}}\mathtt{get}^i_{1_i}\mathtt{rel}^i_{1_{i+1}}$$

$\textsc{send}(1)$ and $\textsc{rec}_i(1)$ are defined analogously, by replacing 0 by 1 and 1 by 0 everywhere.

The following global sequences will be useful as well:

$$\textsc{acqs}(0) = \mathtt{get}^{n-1}_{0_n}\mathtt{rel}^{n-1}_{0_{n-1}}\cdots\mathtt{get}^1_{0_1}\mathtt{rel}^1_{0_0}$$

$$\textsc{rels}(0) = \mathtt{get}^1_{0_0}\mathtt{rel}^1_{0_1}\cdots\mathtt{get}^{n-1}_{0_{n-1}}\mathtt{rel}^{n-1}_{0_n}$$

$$\textsc{nop} = nop^1\cdots nop^{n-1}$$

$\textsc{acqs}(1)$ and $\textsc{rels}(1)$ are defined analogously, by replacing 0 by 1 and 1 by 0 everywhere.

The transition system of each process p_i is designed as follows: We start with $\mathcal{A}_i$, and we replace every transition labelled 0 with a sequence of transitions labelled by actions of $\textsc{send}(0)$ if $i = n$, and $\textsc{rec}_i(0)$ if $1 \leq i \leq n-1$ (there is at least one such i as $n \geq 2$).

Furthermore we add a few transitions so that each p_i with $i \leq n-1$ executes $\textsc{start}_i = \mathtt{get}^i_{key_{i+1}}\mathtt{get}^i_{0_i}\mathtt{get}^i_{1_i}\mathtt{get}^i_{key_i}\mathtt{rel}^i_{key_{i+1}}$ before entering the initial state of $\mathcal{A}_i$. If $i = n$ that sequence is $\textsc{start}_n = \mathtt{get}^n_{a_n}\mathtt{get}^n_{b_n}\mathtt{get}^n_{key_n}$. We also add a transition reading end^i from all states of F_i to a state $stop_i$ with no outgoing transition.

The objective is that the action end^i is executed for all $1 \leq i \leq n$.

One direction is easy. Say there is a word $u = b_1b_2\cdots b_m$ in the intersection of the languages of the $\mathcal{A}_i$. Then we start by executing all $\textsc{start}_i$ sequences for all i in increasing order, and then, for each $1 \leq j \leq m$ (in increasing order), we execute the sequence of operations

$$\begin{aligned} seq(b_j) = &\mathtt{rel}^n_{(b_j)_n}\textsc{nop}\ \textsc{acqs}(b_j)\mathtt{get}^n_{(b_j)_0}\mathtt{rel}^n_{(1-b_j)_n} \\ &\textsc{acqs}(1-b_j)\mathtt{get}^n_{(1-b_j)_0}\mathtt{rel}^n_{(b_j)_0}\textsc{rels}(b_j) \\ &\mathtt{get}^n_{(b_j)_n}\mathtt{rel}^n_{(1-b_j)_0}\textsc{rels}(1-b_j)\mathtt{get}^n_{(1-b_j)_n} \end{aligned}$$

This run projects on p_i as $\textsc{start}_i\textsc{rec}_i(b_1)\textsc{rec}_i(b_2)\cdots\textsc{rec}_(b_m)$ if $i \leq n-1$ and $\textsc{start}_i\textsc{send}(b_1)\textsc{send}(b_2)\cdots\textsc{send}(b_m)$ if $i = n$. As u is in the language of $\mathcal{A}_i$, p_i can execute this run locally. It can be easily checked that all operations in that run are valid in the current configuration, hence this sequence can be executed.

As u is accepted by all $\mathcal{A}_i$, after executing the sequence above each process p_i ends up in a state of F_i, and thus they can all execute end^i one after the other.

Conversely, suppose there is some run w whose local projection $w|_p$ on each process p is ends with end^i. Each $w|_{p_i}$ must start with the execution of $\textsc{start}_i$.

We prove the following lemma:

Lemma 8. *For all $j \in \mathbb{N}$, let w_j be the shortest prefix of w whose projection on p_n is $\textsc{start}_n\textsc{send}(b_1)\cdots\textsc{send}(b_j)$. Then the projection of w_j on every other p_i has $\textsc{rec}_i(b_1)\cdots\textsc{rec}_i(b_j)$ as a suffix.*

Proof. We prove this by induction on j. For $j = 0$ it is trivial. Let $j \in \mathbb{N}$, suppose the claim is true for j, we show it for $j + 1$.

First of all note that for all $i \geq 1$, if p_i has finished executing $\textsc{start}_i$ then it holds k_i and will never release it, hence p_{i-1} either has executed $\textsc{start}_{i-1}$ in full or has not begun executing it. In the second case, p_{i-1} will never be able to advance, which is impossible as $w|_p$ is not empty. As a result, after p_n has executed $\textsc{start}_n$, all other p_i must have executed $\textsc{start}_i$.

Another important remark is that after executing $\textsc{start}_i$, all p_i alternate between a `get` and a `rel` no matter which local run they execute. While p_n starts with a `rel`, all other processes start with a `get`. Therefore p_n always holds either 2 or 3 locks, while all others always hold either 3 or 4. There are n processes and $3n$ locks in total, hence at all times the global configuration is such that either all locks are taken and the next operation of some process p is `rel` (and `get` for all others) or one lock is free and all processes have a `get` as their next operation.

Now say process p_n has started executing $\textsc{send}(b_{j+1})$ by releasing $(b_{j+1})_n$. This means some other process must have taken $(b_{j+1})_n$, which can only be p_{n-1}. The only possibility is that p_{n-1} then releases $(b_{j+1})_{n-1}$, which can only be taken by p_{n-2}, ... We must end up executing $\textsc{acqs}(b_{j+1})$, which ends with $(b_{j+1})_0$ free, which can only be taken by p_n.

By continuing this reasoning we conclude that $w_{j+1} = w_j seq(b_{j+1})$, proving the lemma.

As all p_i execute end_i in w, in particular $w|_{p_n}$ ends with end^n, hence it is necessarily of the form $\textsc{start}_n\textsc{send}(b_1)\cdots\textsc{send}(b_m)end^n$. Let w' be w where all end^i have been erased. The lemma above allows us to conclude that for all i, $w'|_{p_i}$ has $\textsc{rec}_i(b_1)\cdots\textsc{rec}_i(b_m)$ as a suffix.

Recall that the languages of all $\mathcal{A}_i$ are included in $11(00+01)^*$. Moreover, we know that $b_1\cdots b_m$ is in the language of $\mathcal{A}_n$, hence $b_1 = b_2 = 1$. Furthermore, all $w'|_{p_i}$ $(i < n)$ are of the form $\textsc{start}_i\textsc{rec}_i(x_1)\cdots\textsc{rec}_i(x_r)$ with $x_1\cdots x_r$ in the language of $\mathcal{A}_i$. As the only moment a factor 11 can appear in a word of those languages is at the beginning, for $w'|_{p_i}$ to have $\textsc{rec}_i(b_1)\cdots\textsc{rec}_i(b_m)$ as a suffix, we must have $w'|_{p_i} = \textsc{start}_i\textsc{rec}_i(b_1)\cdots\textsc{rec}_i(b_m)$.

As a consequence, for all i we have $w|_{p_i} = \textsc{start}_i\textsc{rec}_i(b_1)\cdots\textsc{rec}_i(b_m)end^i$ and thus $b_1\cdots b_m$ must be accepted by all $\mathcal{A}_i$.

We have proven that this system had a run in which each p_i reads end^i if and only if there is a word accepted by all $\mathcal{A}_i$.

As that condition is easily expressible as a regular objective, we obtain the PSPACE-hardness of the regular verification problem for sound exclusive LSS with 6 locks per process. However, our goal was to prove the PSPACE-hardness of the process deadlock problem for sound exclusive LSS.

To do so, we add a process q and locks ℓ_i for $1 \leq i \leq n+1$ so that the transition system of q simply takes ℓ_n and then goes to a state with a self-loop executing nop. We also add, for each $1 \leq i \leq n$, a sequence of transitions from $stop_i$ which take ℓ_i, then ℓ_{i-1} and release ℓ_i if $i \geq 2$, and simply take ℓ_i if $i = 1$, to end up in a state with no outgoing transition.

We show that there is a process-fair run with a finite projection on q if and only if there is one in the previous LSS such that each p_i executes end_i.

If the latter is true, then we just take the same run and prolong it so that each p_i takes ℓ_i. Then all ℓ_i are taken, and all processes need some ℓ_i to advance, we have reached a global deadlock (in particular the run is process-fair, and its projection on q is finite).

Conversely, suppose we have a process-fair run in the new LSS with a finite projection on q. Then q must be blocked, which is only possible if ℓ_n is held forever by p_n, which in turn is only possible if ℓ_{n-1} is held forever by p_{n-1}... We conclude that all p_i must be holding ℓ_i forever from some point on, and thus that they all read end_i.

We project that run to erase all actions getting an ℓ_i. We obtain a run of the previous system in which every process has executed end_i.

As a result, the new LSS has a process-fair run in which q is blocked if and only if the former LSS has a run in which every p_i has executed end_i, if and only if the $\mathcal{A}_i$ recognise a common word.

As a result, the process deadlock problem is PSPACE-complete for sound exclusive LSS with 8 locks per process.

5.2 ...but NP-complete for 2LSS

Then we prove that the complexity falls to NP when we demand that each process uses at most two locks.

Proposition 5. *The regular verification problem is* NP*-complete for 2LSS (the lower bound holds even for sound exclusive 2LSS).*

Proof. We start with the upper bound. Let $\mathcal{S} = ((\mathcal{A}_p)_{p \in Proc}, T, op)$ be a 2LSS and $((\mathcal{B}_p)_{p \in Proc}, \varphi)$ a regular objective. Our NP algorithm goes as follows: we guess a pattern $\mathbf{pat}_p$ for each process p, as well as a valuation ν of the $(\inf_{p,s})_{p \in Proc, s \in S_p}$. For each p let $\textsc{Owns}_p$ be the set of locks kept indefinitely by a run respecting $\mathbf{pat}_p$.

Then we check that those patterns respect the conditions of Proposition 1 and that this valuation satisfies φ (otherwise we stop). We then equip each $\mathcal{B}_p$ with the acceptance condition $\bigwedge_{\nu(\inf_{p,s})=\top} \inf_s \wedge \bigwedge_{\nu(\inf_{p,s})=\bot} \neg\inf_s$

We add a self-loop labelled $\square$ on each state in $\mathcal{A}_p$ whose outgoing transitions all acquire a lock of $\bigcup_{p\in Proc} \text{Owns}_p$.

Then, for each p we construct the product $\mathcal{C}$ of $\mathcal{A}_p$, $\mathcal{B}_p$ and $\mathcal{A}_{\mathbf{pat}_p}$ (from Lemma 1) to obtain an ELA recognising runs of p that match pattern $\mathbf{pat}_p$ and are in the language of $\mathcal{B}_p$. We guess an ultimately periodic run of the form uv^ω with u and v of polynomial size in the number of states of $\mathcal{C}$ and check that it is accepting (otherwise we stop). It is well-known that an ELA either has an empty language or accepts a run of that form. Then we accept.

We accept if and only if there is a valuation ν satisfying φ and a family of patterns $(\mathbf{pat}_p)_{p\in Proc}$ such that there exist local runs $(w_p)_{p\in Proc}$ of the processes matching those patterns and producing words whose runs in the $(\mathcal{B}_p)_{p\in Proc}$ match ν, and such that the finite ones end in states from which they can only take locks of Owns_p. By Proposition 1, this is true if and only if there is a global run of the system satisfying the given objective. Hence the problem is in NP.

For the lower bound, we could easily translate a SAT formula into a regular objective, with one process for each variable choosing to set it to $\top$ or $\bot$.

However, we want to show that the NP complexity lies already in the model with no need for complicated objectives. By Proposition 3, we know that the existence of a process-fair run blocking a given process p is NP-hard for sound 2LSS. However this is not the case if we are restricted to exclusive 2LSS.

In order to prove the lower bound for exclusive 2LSS we adapt the reduction from the proof of Proposition 3. Note that the only non-exclusive processes in Figure 3 are p' and $p^\ell_{i,j}$. In p' we add an extra state 4 and replace the transition from 2 to 3 with a *nop* transition from 2 to 4 and a $\texttt{get}_{t'}$ transition from 4 to 3. What may then happen is that p' gets stuck in 4 because t' is taken by some other process forever, which could not happen before as p' always had the option of releasing a lock in 2. To overcome this, we add to the objective that p' should have an infinite run. We do the same thing for $p^\ell_{i,j}$, by decomposing the $\texttt{get}_{t(\ell^j_i)}$ into two transitions and adding the requirement that all $p^\ell_{i,j}$ should run forever. The proof is then exactly the same as the one for Proposition 3.

6 Nested locks

In this section we address the verification problem for systems with a restricted lock acquisition policy. We require that each process acquires and releases locks as if they were stored in a stack. This is a classical restriction, as this way of managing locks is considered to be sound and suitable in many contexts.

An LSS is *nested* if all its runs are such that a process can only release the lock it acquired the latest among the ones it holds. In [2] (Theorem 5.5) the authors considered a type of system which can be translated to our sound nested exclusive LSS and proved an NP upper bound on the complexity of the following problem: Is there a reachable configuration where there are some processes $p_1, \ldots, p_k \in Proc$ and locks $t_1, \ldots, t_{k+1} = t_1 \in T$ with each p_i holding lock t_i and needing to get t_{i+1} to keep running? We will call such configurations *circular deadlocks*. They leave the question of a matching lower bound open.

We considerably generalise their result by proving an NP upper bound on the regular verification problem for nested LSS (note that the problem above can be solved by guessing a configuration with such a circular deadlock and using our NP algorithm to check reachability of that configuration). We then prove an NP lower bound on the process deadlock problem for sound nested exclusive LSS, thereby adding a matching NP lower bound to their result.

This shows that the nested requirement significantly improves the complexity of the regular verification problem. On the other hand, the NP-hardness is difficult to avoid: it holds even for a very restricted class of systems and for very simple objectives.

Lemma 9. *Every local run in a nested LSS can be decomposed as*

$$w = w_0 a_1 w_1 a_2 \cdots w_{k-1} a_k w_k w_{k+1} \cdots$$

where $a_1, \dots, a_k$ are the actions getting a lock that is not released later in w.

Furthermore, all w_i are neutral. Finally, for all $i \geq k+1$, all locks acquired in w_i are acquired infinitely many times in w. If w is finite, all w_i are empty for $i \geq k+1$. We call this decomposition the stair decomposition of w.

Proof. Let w be a local run of some process p. We start by decomposing it as

$$w = w_0 a_1 w_1 a_2 \cdots w_{k-1} a_k w_\infty$$

with $a_1, \dots, a_k$ the actions getting a lock that is not released later in the run. For all i let t_i be the lock taken by a_i, namely $op(a_i) = \mathtt{get}_{t_i}$.

We check that all $w_0, \dots, w_{k-1}$ are neutral. Consider some w_i. If a lock t is taken in w_i then it must be released later in the run because a_{i+1} is the next operation that takes a lock and does not release it. But because of the nesting discipline t cannot be released after a_{i+1}. So it must be released in w_i.

Now we look at w_∞. Every lock acquired in it must be released eventually. Thus if the run is finite we can set $w_k = w_\infty$ and $w_i = \varepsilon$ for all $i \geq k+1$.

If the run is infinite then we proceed as follows: Before executing w_∞, p holds $t_1, \dots, t_k$. We construct a sequence of neutral runs w'_j such that $w_\infty = w'_1 w'_2 \cdots$. Say we constructed $w'_1 \cdots w'_j$. As they are all neutral, after executing them p holds $t_1, \dots, t_k$. The next action a in w_∞ cannot release a lock as none of those locks are ever released. If a does not get a lock then we can simply set $w_{j+1} = a$. If a acquires lock t then let w_{j+1} be the infix of w_∞ starting with a and ending with the next action releasing t. This run is neutral as the system is nested. Then let j be such that $w'_1 \cdots w'_j$ contains all $\mathtt{get}_t$ operations with t acquired finitely many times in w_∞. We set $w_k = w'_1 \cdots w'_j$ and for all $i \geq k+1$, $w_i = w'_{j-k+i}$. We obtain our decomposition.

We now define patterns of local runs in a similar manner as in Section 3.

Definition 13. *Consider a (finite or infinite) local run w of process p, and its stair decomposition $w = w_0 a_1 \cdots w_{k-1} a_k w_k w_{k+1} \cdots$. For all i let t_i be the lock acquired by a_i.*

We say that w matches a stair pattern $(\textsc{Owns}^N(w), \leq^w, \textsc{Inf}^N(w))$ *when* $\textsc{Owns}^N(w) = \{t_1, \dots, t_k\}$*, the set of locks acquired infinitely many times is included in* $\textsc{Inf}^N(w)$*, and* $\leq_p^w$ *is a total order on* T *satisfying two conditions:*

- *if t is acquired finitely many times and t' infinitely many times then $t \leq_p^w t'$,*
- *if $t = t_i$ for some i and t' is acquired at some point after a_i then $t \leq_p^w t'$.*

The N in exponent above $\textsc{Owns}_p^N$ and $\textsc{Inf}_p^N$ is for nested, to avoid confusion with the notations defined in Section 3: while $\textsc{Owns}^N$ and $\textsc{Owns}$ correspond to the same idea, $\textsc{Inf}^N$ and $\textsc{Inf}$ are two different things.

Note that unlike the patterns defined for 2LSS, here a run may have several different patterns. We could define unique patterns but this would somehow make the statement of Lemma 10 and the proof of Lemma 11 more complicated.

Our next lemma characterises when local runs can be combined into a process-fair global one. Once again the characterisation uses only patterns and last states of the local runs.

Lemma 10. *Consider a family of (finite or infinite) local runs $(w_p)_{p\in Proc}$ of a nested LSS. For each $p \in Proc$ we consider a stair decomposition of w_p:*

$$w_p = w_{p,0} a_{p,1} \cdots w_{p,k_p-1} a_{p,k_p} w_{p,k_p} w_{p,k_p+1} \cdots$$

and for each $a_{p,i}$ let $t_{p,i}$ be the lock such that $op(a_{p,i}) = \mathtt{get}_{t_{p,i}}$.

Runs $(w_p)_{p\in Proc}$ can be scheduled into a process-fair global run if and only if there exist for each p a stair pattern $(\textsc{Owns}_p^N, \leq_p, \textsc{Inf}_p^N)$ that w_p matches and the following conditions are satisfied.

1. *The $\textsc{Owns}_p^N$ sets are pairwise disjoint.*
2. *All $\leq_p$ orders are the same.*
3. *For all p, if w_p is finite then it leads to a state where all outgoing transitions acquire a lock from $\bigcup_{p\in Proc} \textsc{Owns}_p^N$.*
4. *The set $\bigcup_{p\in Proc} \textsc{Owns}_p^N$ is disjoint from $\bigcup_{p\in Proc} \textsc{Inf}_p^N$.*

Proof. Suppose we have a process-fair global run w whose local projections are the $(w_p)_{p\in Proc}$. For each p let $\textsc{Owns}_p^N$ be the set of locks kept indefinitely in w_p and $\textsc{Inf}_p^N$ the set of locks acquired infinitely often in w_p. Let $\leq$ be a total order on locks such that for all $t, t' \in T$, if t is acquired finitely many times in w and there is an operation on t' after the last operation on t then $t \leq t'$. In particular, a lock acquired infinitely often is always greater than one acquired finitely many times. Further, for all p, i the action $a_{p,i}$ acquires $t_{p,i}$, which is not released later. Thus $a_{p,i}$ is the last action with an operation on $t_{p,i}$ in w. Hence if another lock t is used after $a_{p,i}$ in w_p, it is also used after $a_{p,i}$ in w, and therefore $t_{p,i} \leq t$. As a result, $(\textsc{Owns}_p^N, \leq, \textsc{Inf}_p^N)$ is a pattern of w_p for all p, and 2 is immediately satisfied.

As each p eventually holds $\textsc{Owns}_p^N$ and keeps those locks forever, the $\textsc{Owns}_p^N$ have to be disjoint, thus condition 1 is satisfied.

For condition 3, we use the fact that w is process-fair. For all p, if w_p is finite then it leads to a state where after some point in the run none of the outgoing transitions can be executed. Hence all these transitions acquire a lock that is never released after some point. This is the case for locks of $\bigcup_{p \in Proc} \textsc{Owns}_p^N$ but not for the others, which are free infinitely often. Hence condition 3 holds.

Finally, as all locks from $\bigcup_{p \in Proc} \textsc{Owns}_p^N$ are eventually never free while the locks from $\bigcup_{p \in Proc} \textsc{Inf}_p^N$ are free infinitely often, the two sets are necessarily disjoint, proving condition 4.

For the other implication, suppose that we have patterns $(\textsc{Owns}_p^N, \leq_p, \textsc{Inf}_p^N)$ such that all conditions are satisfied. Let $\leq$ be the total order on locks common to all patterns, which exists by condition 2. We start by executing one by one for each run w_p its prefix $w_{p,0}$, leaving all locks free are the $w_{p,0}$ are all neutral.

We use the notation T_O for the set $\bigcup_{p \in Proc} \textsc{Owns}_p^N$. We index the locks of T_O so that $T_O = \{t_1, \dots, t_m\}$ and $t_1 \leq t_2 \leq \cdots \leq t_m$. For each $t_i \in T_O$ there is a pair (p_i, j_i) such that $op(a_{p_i,j_i}) = \mathtt{get}_{t_i}$. Furthermore that pair is unique as a process p cannot have $a_{p,j}$ take t_i for two different j (by definition of stair decomposition) and as the $\textsc{Owns}_p^N$ are disjoint (by condition 1). We execute, for all $t_i \in T_O$, in increasing order on i, $a_{p_i,j_i} w_{p_i,j_i}$.

At first all locks are free. Then, for each i, just before we execute $a_{p_i,j_i} w_{p_i,j_i}$, the locks that are not free are exactly $\{t_{i'} \mid i' \leq i-1\}$. Hence for every lock $t_{i'}$ that is not free, we have $t_{i'} \leq t_i$ and $t_{i'} \neq t_i$.

By definition of $\leq_p$, all locks t' acquired in $a_{p_i,j_i} w_{p_i,j_i}$ are such that $t_i \leq_p t'$, hence $t_i \leq t'$ by condition 2. As a result, they are all free just before we execute $a_{p_i,j_i} w_{p_i,j_i}$. After we execute it, the set of non-free locks becomes $\{t_{i'} \mid i' \leq i\}$.

The projection of the resulting run on each p is $w_{p,0} a_{p,1} \cdots w_{p,k_p-1} a_{p,k_p} w_{p,k_p}$.

All that is left to do is executing the $w_{p,i}$ for $i \geq k_p + 1$ for each p. They only contain operations on locks that are acquired infinitely many times which are thus in $\textsc{Inf}_p^N$ as w_p matches pattern $(\textsc{Owns}_p^N, \leq_p, \textsc{Inf}_p^N)$, and therefore free by condition 4. As furthermore all $w_{p,i}$ are neutral by definition of stair decomposition, we can execute the next $w_{p,i}$ for each p again and again indefinitely, to obtain an infinite global run of the system.

This run is furthermore process-fair as the finite w_p lead to states whose outgoing transitions acquire locks of T_O, which are eventually all taken forever. Hence those processes do not have an available action infinitely often.

Before we can present our NP algorithm, we need one last technical lemma to show that we can recognise runs with a given pattern using a small automaton.

Lemma 11. *Given a process p and a stair pattern* **pat** *we can construct an ELA $\mathcal{A}^p_{\mathbf{pat}}$ such that for all nested local run w_p, $w_p^{\square}$ is accepted if and only if w_p matches stair pattern* **pat**. *The automaton $\mathcal{A}^p_{\mathbf{pat}}$ has at most $(|T_p| + 2)^2$ states and a formula of constant size for the accepting condition.*

Proof. Let $\mathbf{pat} = (\textsc{Owns}_p^N, \leq_p, \textsc{Inf}_p^N)$. We set $\textsc{Owns}_p^N = \{t_1, \dots, t_k\}$ so that $t_1 \leq_p \cdots \leq_p t_k$.

We define the automaton $\mathcal{A}^p_{\mathbf{pat}} = (S^p_{\mathbf{pat}}, \Sigma_p \cup \{\Box\}, \Delta^p_{\mathbf{pat}}, init^p_{\mathbf{pat}}, \varphi^p_{\mathbf{pat}})$ as follows: If there exist t, t' such that $t \in \textsc{Inf}^N_p$, $t' \notin \textsc{Inf}^N_p$ and $t \leq_p t'$ then no run can match this stair pattern, hence we simply set $\mathcal{A}^p_{\mathbf{pat}}$ as an automaton with an empty language. From now on we will assume that it is not the case.

The states of the automaton are $S^p_{\mathbf{pat}} = \{0, \ldots, k, \infty\} \times (T_p \cup \{neutral\})$, with $init^p_{\mathbf{pat}} = (0, neutral)$.

Intuition The first component of each state gives an index i such that the run read so far is of the form $w_0 a_1 w_1 \cdots a_i w_i$ with w_j neutral for all $j < i$, and for all $j \leq i$ $op(a_j) = \mathtt{get}_{t_j}$ and all locks t' used after a_j are such that $t_j \leq_p t'$. If the first component is ∞ it means we will only use locks of $\textsc{Inf}^N_p$ in the future.

The second component of a state (i, x) indicates which lock apart from $\{t_1, \ldots, t_i\}$ we acquired earliest among the ones we own. If we released all locks acquired since we took t_i, then the second component is *neutral*. We do not need to keep track of all locks acquired as we are only interested in nested runs: If we are in state $(i, neutral)$ and acquire some lock t, we go to state (i, t) to wait for it to be released: if we stay in state (i, t) indefinitely the run is not accepted, otherwise t is released we know that if the run we read is nested then all locks taken since we took t have been released before.

Formal proof: For each action $a \in \Sigma_p \cup \{\Box\}$ and state $s \in S^p_{\mathbf{pat}}$ we have the following transitions:

- If $s = (i, neutral)$ with $i < k$ then:
  - If $op(a) = \mathtt{get}_{t_{i+1}}$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{(i+1, neutral), (i, t_{i+1})\}$
  - If $op(a) = \mathtt{get}_t$ with $t_i \leq_p t$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{(i, t)\}$
- If $s = (k, neutral)$ then:
  - If $op(a) = \mathtt{get}_t$ with $t_k \leq t$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{(k, t)\}$
- If $s = (i, t)$ with $i \leq k$ then:
  - If $op(a) = \mathtt{rel}_t$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{(i, neutral)\}$
  - If $op(a) = \mathtt{get}_{t'}$ or $\mathtt{rel}_{t'}$ with $t_i \leq_p t'$ and $t' \neq t_i$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{s\}$
- If $s = (\infty, neutral)$ then:
  - If $op(a) = \mathtt{get}_t$ with $t \in \textsc{Inf}^N_p$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{(\infty, t)\}$
- If $s = (\infty, t)$ then:
  - If $op(a) = \mathtt{rel}_t$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{(\infty, neutral)\}$
  - If $op(a) = \mathtt{get}_{t'}$ or $\mathtt{rel}_{t'}$ with $t' \in \textsc{Inf}^N_p$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{s\}$
- If $op(a) = nop$ then $\Delta^p_{\mathbf{pat}}(s, a) = \{s\}$ for all s.
- If $a = \Box$ then $\Delta^p_{\mathbf{pat}}((k, neutral), a) = \Delta^p_{\mathbf{pat}}((\infty, neutral), a) = \{(\infty, neutral)\}$
- Otherwise $\Delta^p_{\mathbf{pat}}(s, a) = \emptyset$.
- We add an ε-transition from $(k, neutral)$ to $(\infty, neutral)$. It can be eliminated by adding a few transitions to the automaton, but we allow it as it simplifies the proof.

The acceptance condition $\varphi_{\mathbf{pat}}$ is simply $\inf_{(\infty, neutral)}$.

Let w be a local run of p matching the given stair pattern $\mathbf{pat}$, and let $w = w_0 a_1 \cdots w_{k-1} a_k w_k w_{k+1} \cdots$ be its stair decomposition. For all $i < k$ there is

a path in the automaton reading $w_{p,i}$ from state $(i, \mathit{neutral})$ to itself: every letter acquiring some t (thus getting to state (i,t)) is later followed by one releasing it. Letters using a lock lower than t_i for $\leq_p$ cannot appear in $w_{p,i}$ as otherwise w_p would not match **pat**. Furthermore, there are no $\square$ in $w_{p,i}$. As a result, when t is eventually released we are still in state (i,t) and we go back to state $(i, \mathit{neutral})$.

As a result, the run $w_0 a_1 \cdots w_{k-1} a_k w_k$ labels a path from $(0, \mathit{neutral})$ to $(k, \mathit{neutral})$ in the automaton. Then all letters that appear in the w_i for $i \geq k$ are greater than t_k, otherwise w_p would not match **pat**. If w_p is finite then all the following letters are $\square$, and we stay in $(\infty, \mathit{neutral})$ forever.

If w_p is infinite then by definition of the stair decomposition all the following letters use locks of $\textsc{Inf}_p^N$ or apply nop. Then we can take the ε transition to $(\infty, \mathit{neutral})$. Each w_j with $j \geq i+1$ labels a path from $(\infty, \mathit{neutral})$ to itself: all $\mathtt{get}_t$ operations that get the run to (∞, t) are matched by a later operation $\mathtt{rel}_t$ taking it back to $(\infty, \mathit{neutral})$. In both cases the run is accepting as it visits $(\infty, \mathit{neutral})$ infinitely many times.

Now let w be a nested local run of p such that $w^{\square}$ is accepted by $\mathcal{A}_{\mathbf{pat}}$. Then we consider an accepting computation of w in $\mathcal{A}$ and decompose w as $w = w_0 a_1 \cdots w_{k-1} a_k w_\infty$ with a_i the first letter in the run such that the computation gets to $(i, \mathit{neutral})$ after reading the prefix $w_0 a_1 \cdots w_{i-1} a_i$. By definition of the automaton, we must have $\mathit{op}(a_i) = \mathtt{get}_{t_i}$ for all i, and all operations executed after a_i must be on locks greater than t_i for $\leq_p$.

We show that for all $i \in \{0, \ldots, k, \infty\}$, any nested run labelling a path from $(i, \mathit{neutral})$ to itself must be neutral: Suppose it is not the case, let u be a nested run that is not neutral labelling a path from $(i, \mathit{neutral})$ to itself, of minimal size. The first operation on locks in u must be a $\mathtt{get}_t$, as otherwise u cannot be read from $(i, \mathit{neutral})$. In order to go back to $(i, \mathit{neutral})$, there must be a later operation $\mathtt{rel}_t$ in u. Hence, as u is nested, we have $u = avbu'$ with $\mathit{op}(a) = \mathtt{get}_t$, $\mathit{op}(b) = \mathtt{rel}_t$, v neutral, and u' labelling a path from $(i, \mathit{neutral})$ to itself. By minimality of u, u' must be neutral, hence so must be u: contradiction.

All runs w_i label a path from $(i, \mathit{neutral})$ to itself, thus they must be neutral.

If w is finite, then so is w_∞. Furthermore w_∞ is neutral as it must label a path from $(k, \mathit{neutral})$ to itself. Thus w must match the stair pattern **pat**.

Otherwise, we cut w_∞ in parts so that $w_\infty = w_k w_{k+1} \cdots$ with w_k labelling a path from $(k, \mathit{neutral})$ to itself and for all $j \geq k+1$, w_j labelling a path from $(\infty, \mathit{neutral})$ to itself.

This decomposition exists as, for w to be accepted, w_∞ must label a path starting in $(k, \mathit{neutral})$, taking at some point the ε transition from $(k, \mathit{neutral})$ to $(\infty, \mathit{neutral})$, and then going back infinitely many times to $(\infty, \mathit{neutral})$.

As each w_j labels a path from either $(k, \mathit{neutral})$ or $(\infty, \mathit{neutral})$ to itself, they are all neutral. Furthermore, the w_j with $j \geq k+1$ can only use locks of $\textsc{Inf}_p^N$, as the automaton only allows those operations from states with an ∞ first component.

As a result, w matches pattern **pat**.

We can finally give an NP upper bound for the problem over nested LSS.

Proposition 6. *The regular verification problem is decidable in* NP *for sound nested LSS.*

Proof. Let $\mathcal{S} = ((\mathcal{A}_p)_{p\in Proc}, T)$ be a sound nested LSS, and $((\mathcal{B}_p)_{p\in Proc}, \varphi)$ a regular objective.

The algorithm is similar to the one for Proposition 5: we guess a pattern $\mathbf{pat}_p = (\textsc{Owns}_p^N, \leq_p, \textsc{Inf}_p^N)$ for each process p, and a valuation ν of the variables $(\inf_{p,s})_{p\in Proc, s\in S_{\mathcal{B}_p}}$ (the variables of φ, see Definition 5). We check that ν satisfies φ. We transform each $\mathcal{A}_p$ into $\mathcal{A}_p^{\square}$, in which we added, on each state whose outgoing transitions all acquire a lock from $\bigcup_{p\in Proc} \textsc{Owns}_p^N$, a $\square$ self-loop. We also equip each $\mathcal{B}_p$ with an Emerson-Lei accepting condition expressing that the run matches ν.

We then guess, for each process p, a run in the product of $\mathcal{B}_p$, $\mathcal{A}_p$ and $\mathcal{A}_{\mathbf{pat}_p}$ (as described in Lemma 11) that matches valuation ν. It is folklore that if an Emerson-Lei automaton has an accepting run then it has one of the form uv^ω with u and v of polynomial size in the number of states of the automaton. Thus we can guess an accepting run within NP. An accepting run is one that respects ν in $\mathcal{B}_p$, and follows a run of $\mathcal{A}_p$ of pattern $\mathbf{pat}_p$ (and ends in a state with all outgoing transitions getting a lock of $\bigcup_{p\in Proc} \textsc{Owns}_p^N$ if it is finite).

By Lemma 10, we accept if and only if there is a process-fair global run of the LSS satisfying the objective.

We give a matching lower bound, robust to many restrictions. The reduction also solves a question left open in [2], as explained at the beginning of the section.

Proposition 7. *The process deadlock problem and the circular deadlock problem are* NP*-hard for sound nested exclusive LSS.*

Proof. We reduce the Independent Set Problem, in which we are given an undirected graph $G = (V, E)$ (edges are subsets of V of size 2) and an integer k and have to determine whether there is a subset of vertices $S \subseteq V$ such that $|S| = k$ and there are no edges between any two elements of S. Let $n = |V|$.

Let $G = (V, E)$ be an undirected graph, and $k \in \mathbb{N}$. We can assume that $V = \{1, \ldots, n\}$ for some $n \in \mathbb{N}$. We set $E = \{e_1, \ldots, e_m\}$, i.e., we put an arbitrary order on edges in E. Our set of processes is $Proc = \{p_1, \ldots, p_k\}$. For each $1 \leq j \leq m$ we have a lock t_j. We write T for the set $\{t_j \mid 1 \leq j \leq m\}$. Our set of locks is $T \cup \{\ell_1, \ldots, \ell_k\}$. For each $v \in V$ we write E_v for the set of edges adjacent to v and T_v for $\{t_j \mid e_j \in E_v\}$. Each process p_i uses locks of $T \cup \{\ell_i, \ell_{i+1}\}$, with the convention $\ell_{k+1} = \ell_1$.

Each process p_i has n transitions from its initial state, with operation *nop*, which lead to states $s_1, \ldots, s_n$. From each s_v a sequence of transitions (with no choice) acquires all locks $t_j \in T_v$ in increasing order of indices, then acquires ℓ_i, then ℓ_{i+1}, and then releases all those locks in reverse order (thus ensuring the nested property). We end up in a state end_i with a local self-loop. This system is clearly exclusive, as the only state with several outgoing transitions is the initial one, and none of them acquire any lock.

Suppose that this LSS has a run w leading to a circular deadlock. The structure of the LSS imposes that when executing w we eventually stay in the same configuration forever, with some processes blocked because they cannot acquire some lock and some looping indefinitely on their state end_i.

Let C be that configuration. If some p_i is stuck after acquiring ℓ_i, then it cannot have acquired ℓ_{i+1}, as otherwise it could release all of its locks and loop in end_i. Hence some other process holds ℓ_{i+1}, and it can only be p_{i+1} (with $p_{k+1} = p_1$). By iterating this reasoning, we conclude that all processes p_i are blocked while holding ℓ_i, as they cannot acquire ℓ_{i+1}. They must be holding disjoint sets of locks. By construction, each p_i is holding ℓ_i, plus the locks of some T_v, $v \in V$. Hence we have k disjoint T_v, i.e., we have a set of k vertices whose sets of adjacent edges are disjoint, i.e., an independent set of size k.

Now suppose no p_i is stuck after acquiring ℓ_i. Then all p_i that have acquired ℓ_i have reached end_i, and released all their locks, thus all ℓ_i are free. There must be at least one process blocked when trying to acquire an element of some T_v. Let j be the highest index in $\{1, \dots, m\}$ such that there is a process p_i blocked because it cannot acquire t_j. Then there is a process $p_{i'}$ which is holding t_j, and is itself unable to acquire some $t_{j'}$ (as all locks ℓ_r are free). However, as all processes acquire elements of T in increasing order of index, we must have $j' > j$, contradicting the maximality of j. Thus this case cannot happen, concluding the first part of our reduction.

Conversely suppose we have an independent set of vertices $S = \{v_1, \dots, v_k\} \subseteq V$ of size k. Then we construct the run w in which, one by one, each p_i first goes to s_{v_i} and then acquires $\{\ell_i\} \cup T_{v_i}$. This is possible as they all acquire disjoint sets of locks. We end up in a configuration where each p_r needs k_{r+1} to advance, but cannot do so as k_{r+1} is held by p_{r+1}. Hence w yields a circular deadlock (and even a global deadlock, which shows that it is process-fair). This ends our reduction, proving that the circular deadlock problem is NP-hard even for nested exclusive LSS. In the LSS above, we showed that if a run yields a circular deadlock then it yields a global deadlock. Hence we can apply the reduction to the process deadlock problem by picking an arbitrary process p_i. There is a process-fair run with a finite projection on p_i if and only if there is a solution to the initial Independent set problem.

Remark 2. The Independent set problem is NP-hard even on graphs of degree 3 [5] (Theorem 2.6). As in the reduction above the number of locks used by each process is bounded by the degree of the input graph, we conclude that the lower bound still holds for systems where each process uses at most 5 locks.

7 Conclusion

We have studied the verification problem for LSS against boolean combinations of regular local objectives. We established PSPACE-completeness for the general problem, and presented two subcases where the verification problem becomes NP-complete: 2LSS and nested LSS, as well as a PTIME algorithm for the process

deadlock problem for exclusive 2LSS. The NP and PTIME upper bounds use as their main ingredient the characterisations of whether local runs can be scheduled into global ones through patterns. All lower bounds are robust, as they hold with bounds on the number of locks per process and very simple objectives.

Concerning future work, most of our results can easily be extended to the case when processes are pushdown systems (except for the general case, which is undecidable instead of PSPACE-complete, see [9], Theorem 8). Another easy extension is to replace nested with bounded lock chains, a weaker condition defined in [7]. These essentially do not require new ideas, thus we chose to not include them to avoid unnecessary details and highlight the key ingredients. At the time of writing this paper, we are working towards implementing the algorithms described here (using a SAT solver for the NP-hard problems), in which we plan to include those extensions.

About open problems, we do not know if partial deadlocks can be detected in PTIME for exclusive LSS, or for 2LSS (not necessarily exclusive). Probabilistic algorithms have proven useful in distributed systems (see, for instance, the Lehmann-Rabin algorithm [10]), hence one may want to add probabilities to the model. Finally, versions of the problem with parameterized number of processes or locks could be of interest.

I would like to thank Anca Muscholl and Igor Walukiewicz for their support and useful comments.

References

1. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. p. 62–73. POPL '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/604131.604137, https://doi.org/10.1145/604131.604137
2. Brotherston, J., Brunet, P., Gorogiannis, N., Kanovich, M.: A compositional deadlock detector for android java. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering. p. 955–966. ASE '21, IEEE Press (2021). https://doi.org/10.1109/ASE51524.2021.9678572, https://doi.org/10.1109/ASE51524.2021.9678572
3. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8, https://doi.org/10.1007/978-3-319-10575-8
4. Emerson, E.A., Lei, C.L.: Modalities for model checking: branching time logic strikes back. Science of Computer Programming **8**(3), 275–306 (1987). https://doi.org/https://doi.org/10.1016/0167-6423(87)90036-0, https://www.sciencedirect.com/science/article/pii/0167642387900360
5. Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified np-complete graph problems. Theoretical Computer Science **1**(3), 237–267 (1976). https://doi.org/https://doi.org/10.1016/0304-3975(76)90059-1, https://www.sciencedirect.com/science/article/pii/0304397576900591
6. Gimbert, H., Mascle, C., Muscholl, A., Walukiewicz, I.: Distributed Controller Synthesis for Deadlock Avoidance. In: Bojanczyk, M., Merelli, E., Woodruff, D.P.

(eds.) 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 229, pp. 125:1–125:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ICALP.2022.125, https://drops.dagstuhl.de/opus/volltexte/2022/16466

7. Kahlon, V.: 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. pp. 27–36 (2009). https://doi.org/10.1109/LICS.2009.45
8. Kahlon, V., Gupta, A.: An automata-theoretic approach for model checking threads for LTL properties. In: 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06). pp. 101–110 (2006). https://doi.org/10.1109/LICS.2006.11
9. Kahlon, V., Ivancić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Proceedings of the 17th International Conference on Computer Aided Verification. p. 505–518. CAV'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11513988_49, https://doi.org/10.1007/11513988_49
10. Lehmann, D., Rabin, M.O.: On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 133–138 (1981)
11. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) 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. Lecture Notes in Computer Science, vol. 3440, pp. 93–107. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_7, https://doi.org/10.1007/978-3-540-31980-1_7
12. Taylor, R.N.: A general-purpose algorithm for analyzing concurrent programs. Commun. ACM **26**(5), 361–376 (may 1983). https://doi.org/10.1145/69586.69587, https://doi.org/10.1145/69586.69587
13. Zielonka, W.: Notes on finite asynchronous automata. RAIRO Theor. Informatics Appl. **21**(2), 99–135 (1987). https://doi.org/10.1051/ita/1987210200991, https://doi.org/10.1051/ita/1987210200991