

Distributed controller synthesis for deadlock avoidance

Hugo Gimbert¹, Corto Mascle², Anca Muscholl³, and Igor Walukiewicz⁴

^{1,4}Université de Bordeaux, CNRS, France
^{2,3}Université de Bordeaux, France

Abstract: We consider the distributed control problem for systems synchronizing over locks. The goal is to find a local controller for each of the processes so that global deadlocks of the system are avoided. Without restrictions this problem is shown to be undecidable, even for a fixed number of processes and locks. We identify two restrictions that make the distributed control problem decidable. The first one is to allow each process to use at most two locks. The problem is shown to be Σ_2^P -complete in this case, and even in PTIME under some additional assumptions. For example, the dining philosophers problem satisfies these assumptions. The second restriction is the nested usage of locks. In this case the distributed control problem is NEXPTIME complete. The drinking philosophers problem falls in this case.

1 Introduction

Automatic synthesis of distributed systems has a big potential since such systems are difficult to write, test, or verify. The state space and the number of different behaviors grow exponentially with the number of processes. This is where distributed synthesis can be more useful than centralized synthesis, because an equivalent, sequential system may be very big. The other important point is that distributed synthesis produces by definition a distributed system, while a synthesized sequential system may not be implementable on a given distributed architecture. Unfortunately, very few settings are known for which distributed synthesis is decidable, and those that we know require at least exponential time.

The problem was first formulated by Pnueli and Rosner [31]. Subsequent research showed that, essentially, the only decidable architectures are pipelines, where each process can send messages only to the next process in the pipeline [12, 23, 28]. In addition, the complexity is non-elementary in the size of the pipeline.

These negative results motivated the study of distributed synthesis for asynchronous automata, and in particular synthesis with so called causal information. In this setting the problem becomes decidable for co-graph action alphabets [13], and for tree architectures of processes [15, 29]. Yet the complexity is again non-elementary, this time w.r.t. the depth of the tree. Worse, it has been recently established that distributed synthesis with causal information is undecidable for unconstrained architectures [18]. Distributed synthesis for (safe) Petri nets [11] has encountered a similar line of limited advances, and due to [18], is also undecidable in the general case, since it is inter-reducible to distributed synthesis for asynchronous automata [3]. This situation raised the question if there is any setting for distributed synthesis that covers some standard examples of distributed systems, and is manageable algorithmically.

In this work we consider distributed systems with locks; each process can take or release a lock from a pool of locks. Locks are one of the most classical concepts in distributed systems. They are also probably the most frequently used synchronization mechanism in concurrent programs. We formulate our results in a control setting rather than synthesis – this avoids the need for a specification formalism. The objective is to find a local strategy for each process so that the global system does not get stuck. For unrestricted systems with locks we hit again an undecidability barrier, as for the models discussed above. Undecidability was known already for the verification of systems where each process is modelled as a pushdown automaton [22], since unrestricted usage of locks allows for inter-process communication. Yet, we are able to find quite interesting restrictions making distributed control synthesis for systems with locks decidable, and even algorithmically manageable.

The first restriction is to limit the number of locks available to each process to two. The classical example is the dining philosophers problem, where each philosopher has two locks corresponding to the left and the right fork. Observe that we do not limit the total number of locks in the system. We show that the complexity of this synthesis problem is at the second level of the polynomial hierarchy. The problem gets even simpler when we restrict the strategies such that they cannot block a process when all locks are available. We call such strategies *locally live*. We obtain an NP-algorithm for locally live strategies, and even a PTIME algorithm when the access to locks is *exclusive*. This means that once a process tries to acquire some lock it cannot switch to another action before getting it. In other words, a process that tries to get a lock is blocked as long as this lock is not available.

The second restriction is nested lock usage. This is a very common restriction in concurrent programs [21], and sometimes it is enforced syntactically by associating locks with program blocks. Nested lock usage simply says that acquiring and releasing locks should follow a stack discipline. Verification of concurrent programs with nested locks has been shown decidable in [21, 22], and this triggered further work on extensions of lock usage policies [4, 20, 24]. In distributed computing, the drinking philosophers setting [5] is an example of nested lock usage. We show that in this case the distributed synthesis problem is NEXPTIME-complete, where the exponent in the algorithm depends only on

the number of locks. A decision procedure for the verification of such systems, based on similar ideas on lock orderings, appeared already in [22]. Note that we study here a more general problem, namely distributed control. Our results are stated for finite-state processes only, in order to keep it simple, but they hold for pushdown processes as well.

As mentioned above, we formalize the distributed synthesis problem as a control problem [32]. A process is given as a transition graph where transitions can be local actions, or acquire/release of a lock. Some transitions are **controllable**, and some are not. A controller for a process decides which **controllable** transitions to allow, depending on the local history. In particular, the controller of a process does not see the states of other processes. Our techniques are based on analyzing patterns of taking and releasing locks. In decidable cases there are finite sets of patterns characterizing potential deadlocks.

The notion of patterns resembles locking disciplines [8], which are commonly used to prevent deadlocks. An example of a locking discipline is “take the left fork before the right one” in the dining philosophers problem. Our results allow to check if a given locking discipline may result in a deadlock, and in some cases even list all deadlock-avoiding locking disciplines.

To summarize, the main results of our work are:

- Σ_2^P -completeness of the deadlock avoidance control problem for systems where each process has access to at most 2 locks (**2LSS**).
- An NP algorithm for **2LSS** with locally live strategies.
- A PTIME algorithm for **2LSS** with locally live strategies and exclusive lock access.
- A NEXPTIME algorithm and the matching lower bound for the nested lock case.
- Undecidability of the deadlock avoidance control problem for systems with unrestricted access to locks (with three processes and four locks in total).

Related work Distributed synthesis is an old idea motivated by the Church synthesis problem [6]. Actually, the logic CTL has been proposed with distributed synthesis in mind [7]. Given this long history, there are relatively few results on distributed synthesis. Three main frameworks have been considered: synchronous networks of input/output automata, asynchronous automata, Petri games.

The synchronous network model has been proposed by Pnueli and Rosner [30, 31]. They established that controller synthesis is decidable for pipeline architectures and undecidable in general. The undecidability result holds for very simple architectures with only two processes. Subsequent work has shown that in terms of network shape pipelines are essentially the only decidable case [12, 23, 28]. Several ways to circumvent undecidability have been considered. One was to restrict to local specifications, specifying the desired behavior

of each automaton in the network separately. Unfortunately, this does not extend the class of decidable architectures substantially [28]. A furthergoing proposal was to consider only input-output specifications. A characterization, still very restrictive, of decidable architectures for this case is given in [14].

The asynchronous (Zielonka) automaton setting was proposed as a reaction to these negative results [13]. The main hope was that causal memory helps to prevent undecidability arising from partial information, since the synchronization of processes in this model makes them share information. Causal memory indeed allowed to get new decidable cases: co-graph action alphabets [13], connectedly communicating systems [27], and tree architectures [15, 29]. There is also a weaker condition covering these three cases [17]. This line of research suffered however from a very recent result showing undecidability in the general case [18].

Distributed synthesis in the Petri net model, called Petri games, has been proposed recently in [11]. The idea is that some tokens are controlled by the system and some by the environment. Once again causal memory is used. Without restrictions this model is inter-reducible with the asynchronous automata model [3], hence the undecidability result [18] applies. The problem is EXPTIME-complete for one environment token and arbitrary many system tokens [11]. This case stays decidable even for global safety specifications, such as deadlock, but undecidable in general [10]. As a way to circumvent the undecidability, bounded synthesis has been considered in [9, 19], where the bound on the size of the resulting controller is fixed in advance. The approach is implemented in the tool ADAMSYNT [16].

The control formulation of the synthesis problem comes from the control theory community [32]. It does not require to talk about a specification formalism, while retaining most useful aspects of the problem. A frequently considered control objective is avoidance of undesirable states. In the distributed context, deadlock avoidance looks like an obvious candidate, since it is one of the most basic desirable properties. The survey [36] discusses the relation between the distributed control problem and Church synthesis. Some distributed versions of the control problem have been considered, also hitting the undecidability barrier very quickly [1, 33–35].

We would like to mention two further results that do not fit into the main threads outlined above. In [37] the authors consider a different synthesis problem for distributed systems: they construct a centralized controller for a scheduler that would guarantee absence of deadlocks. This is a very different approach to deadlock avoidance. Another recent work [2] adds a new dimension to distributed synthesis by considering communication errors in a model with synchronous processes that can exchange their causal memory. The authors show decidability of the synthesis problem for 2 processes.

Outline of the paper In the next section we define systems with locks, strategies, and the control problem. We introduce locally live strategies as well as the 2-lock, exclusive, and nested locking restrictions. This permits to state

the main results of the paper. The following three sections consider systems with the 2-lock restriction. First, we briefly give intuitions behind the Σ_2^P -completeness in the general case. Section 3.2 presents an NP algorithm for 2LSS with locally live strategies. Section 3.3 gives a PTIME algorithm for the exclusive case with locally live strategies. Next in Section 4 we consider systems with nested locks, and show that the problem is NEXPTIME-complete in this case. Finally, in Section 5 we prove that without any restrictions the problem is undecidable.

2 Preliminaries

A **lock-sharing system** (LSS for short) is a parallel composition of processes sharing a pool of locks. Processes do not communicate, but they may acquire or release locks from the pool. Some transitions of processes are **uncontrollable**, intuitively the environment decides if such a transition is taken. The goal is to find a strategy for each process so that the system never deadlocks. The challenge is that the strategies are local, in the sense that each process only knows its previous actions, without having a global view of the system.

⌈ A *process* p is an automaton $\mathcal{A}_p = (S_p, \Sigma_p, T_p, \delta_p, \text{init}_p)$ with a set of locks T_p that it can acquire or release. The transition function $\delta_p : S_p \times \Sigma_p \rightarrow \text{Op}(T_p) \times S_p$ associates with a state from S_p and an action from Σ_p an operation on some lock and a new state; it is a partial function. The lock operations consist in acquiring (acq_t) or releasing (rel_t) some lock t from T_p , or doing nothing: $\text{Op}(T_p) = \{\text{acq}_t, \text{rel}_t \mid t \in T_p\} \cup \{\text{nop}\}$. Figure 1 gives an example.

⌈ 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 $(\text{init}_p, \emptyset)$, namely the initial state with no locks. A transition between local configurations $(s, B) \xrightarrow{a, op} (s', B')$ exists when $\delta_p(s, a) = (op, s')$ and one of the following holds:

- $op = \text{nop}$ and $B = B'$;
- $op = \text{acq}_t$, $t \notin B$ and $B' = B \cup \{t\}$;
- $op = \text{rel}_t$, $t \in B$, and $B' = B \setminus \{t\}$.

⌈ A *local run* $(a_1, op_1)(a_2, op_2) \dots$ of \mathcal{A}_p is a finite or infinite sequence over $\Sigma_p \times \text{Op}(T_p)$ such that there exists a sequence of local configurations $(\text{init}_p, \emptyset) = (s_0, B_0) \xrightarrow{(a_1, op_1)}_p (s_1, B_1) \xrightarrow{(a_2, op_2)}_p \dots$. While the run is determined by the sequence of actions, we prefer to make lock operations explicit. We write Runs_p

⌈ for the set of local runs of \mathcal{A}_p . We call a local run *neutral* if it starts and ends with the same set of locks.

⌈ A **lock-sharing system** (LSS) $\mathcal{S} = ((\mathcal{A}_p)_{p \in \text{Proc}}, \Sigma^s, \Sigma^e, T)$ is a set of processes together with a partition of actions between **controllable** and **uncontrollable** ones, and a set T of locks. We have $T = \bigcup_{p \in \text{Proc}} T_p$, for the set of all locks. **Controllable** and **uncontrollable** actions belong to the system and to the environment, respectively. We write $\Sigma = \bigcup_{p \in \text{Proc}} \Sigma_p$ for the set of actions of all processes and

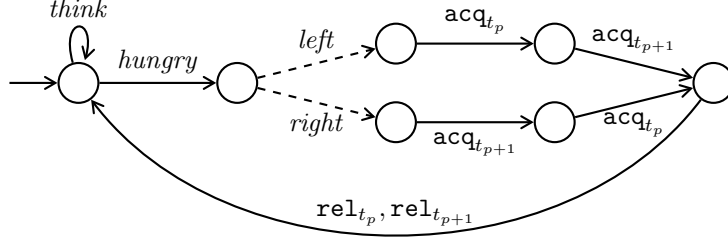


Figure 1: A dining philosopher p . Dashed transitions are [controllable](#).

require that (Σ^s, Σ^e) partitions Σ . The sets of states and action alphabets of processes should be disjoint: $S_p \cap S_q = \emptyset$ and $\Sigma_p \cap \Sigma_q = \emptyset$ for $p \neq q$. The sets of locks are not disjoint, in general, since processes may share locks.

Example 1. The dining philosophers problem can be formulated as a control problem for a [lock-sharing system](#) $\mathcal{S} = ((\mathcal{A}_p)_{p \in Proc}, \Sigma^s, \Sigma^e, T)$.

We set $Proc = \{1, \dots, n\}$ and $T = \{t_1, \dots, t_n\}$ as the set of locks. For every $p \in Proc$, process \mathcal{A}_p is as in Figure 1, with the convention that $t_{n+1} = t_1$. Actions in Σ^s are marked by dashed arrows. These are [controllable](#) actions. The remaining actions are in Σ^e . Once the environment makes a philosopher p hungry, p has to get both the left (t_p) and the right (t_{p+1}) fork to eat. She may however choose the order in which she takes them; actions *left* and *right* are [controllable](#).

A global configuration of \mathcal{S} 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.

The semantics of such systems is *asynchronous*, as transitions between two configurations done by a single process: $C \xrightarrow{(p, a, op)} C'$ if $(s_p, B_p) \xrightarrow{(a, op)}_p (s'_p, B'_p)$ 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 by the sequence of transition labels. A global run w *determines a local run* of each process: $w|_p$ is the sequence of p 's actions in w .

⌈ A *local strategy* σ_p says which actions p can take depending on a local run so far: $\sigma_p : Runs_p \rightarrow 2^{\Sigma_p}$, provided $\Sigma^e \cap \Sigma_p \subseteq \sigma_p(u)$, for every $u \in Runs_p$. A *control strategy* for a [lock-sharing system](#) is a tuple of local strategies, one for each process: $\sigma = (\sigma_p)_{p \in Proc}$. This requirement says that a strategy cannot block environment actions.

A local run u of a system *respects* σ_p if for every non-empty prefix $v(a, op)$ of u , we have $a \in \sigma_p(v)$. Observe that local runs are affected only by the local strategy. A global run w *respects* σ if for every process p , the local run $w|_p$ respects σ_p . We often say just σ -run, instead of “run respecting σ ”.

As an example consider the system for two philosophers from Example 1. Suppose that both local strategies always say to take the *left* transition. So $hungry^1, left^1, acq_{t_1}^1, acq_{t_2}^1$ is a local run of process 1 respecting the strategy; similarly $hungry^2, left^2, acq_{t_2}^2, acq_{t_1}^2$ for process 2. (We use superscripts to indicate the process doing an action.)

The global run $hungry^1, hungry^2, left^1, left^2, acq_{t_1}^1, acq_{t_2}^2$ respects the strategy. It deadlocks, since each philosopher needs a lock the other one owns.

Definition 1 (Deadlock avoidance control problem). A σ -run w *leads to a deadlock in σ* if w cannot be prolonged to a σ -run. A control strategy σ is *winning* if no σ -run leads to a deadlock in σ . The *deadlock avoidance control problem* is to decide if for a given system there is some winning control strategy.

In this work we consider several variants of the deadlock avoidance control problem. Maybe surprisingly, in order to get more efficient algorithms we need to exclude strategies that can block a process by itself:

▮ **Definition 2** (Locally live strategy). A local strategy σ_p for process p is *locally live* if every local σ_p -run u can be prolonged: there is some $b \in \Sigma_p$ such that ub is a local run respecting σ_p . A strategy σ is locally live if each of its associated local strategies is so.

In other words, a *locally live* strategy guarantees that a process does not block if it runs alone. Coming back to Example 1: a strategy always offering one of the *left* or *right* actions is *locally live*. A strategy that offers none of the two is not. Observe that blocking one process after the hungry action is a very efficient strategy to avoid a deadlock, but it is not the intended one. This is why we consider *locally live* to be a desirable property rather than a restriction.

Note that being *locally live* is not exactly equivalent to a strategy always proposing at least one outgoing transition. In our semantics, a process blocks if it tries to acquire a lock that it already owns, or to release a lock it does not own. But it becomes equivalent thanks to the following remark:

Remark 1. We can assume that each process keeps track in its state of which locks it owns. Note that this assumption does not compromise the complexity results when the number of locks a process can access is fixed. We will not use this assumption in Section 4, where a process can access arbitrarily many locks (in a nested fashion).

Without any restrictions our synthesis problem is undecidable. We present the proof of the following theorem in Section 5. We warn the reader that in order to use as few processes and locks as possible, the proof gets quite technical. A shorter proof using more locks can be found in the conference version of this paper.

Theorem 3. *The deadlock avoidance control problem for LSS and at most 4 locks in total is undecidable. It remains so when restricted to locally live strategies.*

We propose two cases when the control problem becomes decidable. The two are defined by restricting the usage of locks.

First we require that each process accesses at most two different locks. In the following definition, we require each process to use exactly two locks, as it is more convenient to avoid case distinctions on the number of locks used by a process. This is not more restrictive as we can always add some dummy locks (which are never actually used) to the set of locks of a process to make sure that it has exactly two.

□ **Definition 4 (2LSS).** A process $\mathcal{A}_p = (S_p, \Sigma_p, T_p, \delta_p, \text{init}_p)$ *uses two locks* if $|T_p| = 2$. A system $\mathcal{S} = ((\mathcal{A}_p)_{p \in \text{Proc}}, \Sigma^s, \Sigma^e, T)$ is a **2LSS** if every process uses two locks.

Note that in the above definition we do not bound the total number of locks in the system, just the number of locks per process. The process from Figure 1 is a **2LSS**. Our first main result says that the control problem is decidable for **2LSS**.

Theorem 5. *The deadlock avoidance control problem for 2LSS is Σ_2^P -complete.*

For the lower bound we use strategies that take a lock and then block. This does not look like a very desired behavior, and this is the reason for introducing the concept of **locally live** strategies. The second main result says that restricting to **locally live** strategies helps.

Theorem 6. *The deadlock avoidance control problem for 2LSS is in NP when strategies are required to be locally live.*

We do not know if the above problem is in PTIME. We can get a PTIME algorithm under one more assumption.

Definition 7 (Exclusive systems). A process p is *exclusive* if for every state $s \in S_p$: if s has an outgoing transition with some acq_t operation then all outgoing transitions have the same acq_t operation. A system is *exclusive* if all its processes are.

Example 2. The process from Figure 1 is *exclusive*, while the one from Figure 2 is not. The latter has a state with one $\text{acq}_{t_{p+1}}$ and one rel_{t_p} outgoing transition. Observe that in this state the process cannot block, and has the possibility to take a lock at the same time. **Exclusive** systems do not have such a possibility, so their analysis is much easier.

Theorem 8. *The deadlock avoidance control problem for exclusive 2LSS is in PTIME, when strategies are required to be locally live.*

Without local liveness, the problem stays Σ_2^P -hard for **exclusive 2LSS**. Our last result uses a classical restriction on the usage of locks:

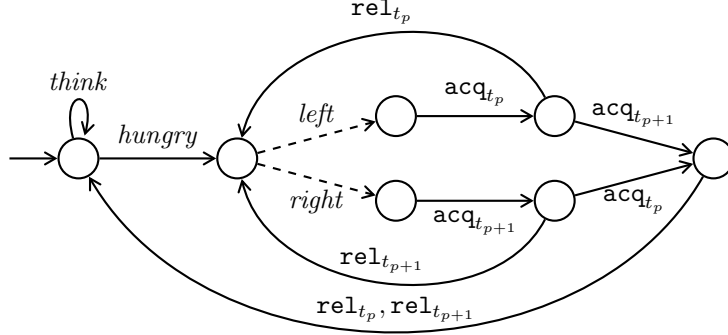


Figure 2: A flexible philosopher p . She can release a fork if the other fork is not available.

Definition 9 (Nested-locking). A local run is *nested-locking* if the order of acquiring and releasing locks in the run respects a stack discipline, i.e., the only lock a process can release is the last one it acquired.

⌈ A process is nested-locking if all its local runs are, and an LSS is nested-locking if all its processes are.

The process from Figure 1 is *nested-locking*, while the one from Figure 2 is not.

Theorem 10. The *deadlock avoidance control problem for LSS* is NEXPTIME-complete.

3 Two locks per process

We describe how to solve the deadlock avoidance control problem for 2LSS, that are systems where every process uses two locks. We present the three results announced in the previous section, namely, Theorems 5, 6, and 8.

The general case, treated in Theorem 5, puts no restriction on *strategies* or on the system, besides it being a 2LSS. Our approach is to summarize the possibilities of a strategy into a polynomial size witness. We show that from a computational complexity perspective we cannot do better than guessing them to solve the problem.

The next case is when we require *strategies* to be *locally live*. This eliminates a mechanism we use to get the lower bound for the general case. The mechanism is to make a process take one of its locks and then reach a state with no available actions. Indeed, with *locally live strategies*, if a process takes a lock and never releases it, it either lives forever (then the system wins) or it is blocked when trying to get its other lock (meaning that its other lock is not available).

Finally, we consider the restriction of the deadlock avoidance problem to *exclusive* systems (still with *locally live strategies*). Here, whenever a process can execute an action acquiring a lock it is the only thing it can do. This means that a process gets blocked whenever it tries to get a lock and the lock is not free. Observe, that in general this is not the case (Figure 2).

In this section we fix a **2LSS** $\mathcal{S} = (\{\mathcal{A}_p\}_{p \in Proc}, \Sigma^s, \Sigma^e, T)$ over the set of processes $Proc$. We assume that the **2LSS** satisfies Remark 1. We also fix a *control strategy* $\sigma = (\sigma_p)_{p \in Proc}$.

The three following subsections present the three cases.

3.1 The general case in Σ_2^P

If a run leads to a (global) *deadlock*, then all the corresponding local runs lead to local configurations in which all actions allowed by σ acquire a lock. This is because local actions and lock release actions are non-blocking.

Our first insight is to summarize local runs by *patterns* describing the most recent lock operations. We will see later that this information is sufficient to decide if the *strategy* is *winning* (Lemma 13). A *pattern of a local run* of a process p in a **2LSS** describes which of the four following situations are possible for process p at the end of a local run:

- p owns both locks;
- p owns no lock;
- p owns exactly one of its locks, say t , and either
 - its last operation on locks was \mathbf{acq}_t ; or
 - the last operation on locks was $\mathbf{rel}_{t'}$ with $t \neq t'$.

Before defining patterns we need to introduce the runs on which we use them, i.e., runs that lead potentially to deadlocks:

Definition 11 (Risky run). Consider a local σ -run u of a process p . We say that u is σ -*risky* w.r.t. σ if after executing u all transitions allowed by σ are \mathbf{acq} transitions. We simply write *risky* when σ is clear from context.

We write $\mathbf{Owns}_{p,\sigma}(u)$ for the set of locks owned by p after u , or simply $\mathbf{Owns}_p(u)$ when the *strategy* is clear from context. We also define $\mathbf{Blocks}_{p,\sigma}(u) = \{t : \mathbf{acq}_t \in \sigma_p(u)\}$ (again, we write simply $\mathbf{Blocks}_p(u)$ when the *strategy* is clear from context).

Note that if a σ -run u is *risky*, and the *strategy* σ is *locally live*, then $\mathbf{Blocks}_p(u) \neq \emptyset$; if σ is not *locally live* then $\mathbf{Blocks}_p(u)$ can be empty. So $\mathbf{Blocks}_p(u)$ is the (possibly empty) set of locks that can block process p after a *risky* run u . If the run is not *risky* then the process can do some local action or a release action.

We can now define patterns formally.

Definition 12 (Patterns). Consider a **risky** local σ -run u of process p . We say that u has a **strong pattern** $\text{Owns}_p(u) \implies \text{Blocks}_p(u)$ if $\text{Owns}_p(u) \neq \emptyset$ and the last operation on locks in u is a release. Otherwise we say that u has a **weak pattern** $\text{Owns}_p(u) \dashrightarrow \text{Blocks}_p(u)$. We also write $\text{Owns}_p(u) \longrightarrow \text{Blocks}_p(u)$ if we do not specify if a pattern is **strong** or **weak**.

◻ We say that σ **admits a pattern** of p if there exists some **risky** σ -run of p having this pattern.

We write \mathbb{P}_p^σ for the set of patterns of p admitted by σ . We write $\mathbb{P}^\sigma = (\mathbb{P}_p^\sigma)_{p \in \text{Proc}}$ for the family of patterns admitted by σ , and call \mathbb{P}^σ the **behavior** of σ .

Note that by definition $\text{Owns}_p(u) \cap \text{Blocks}_p(u) = \emptyset$. Since in a 2LSS any process uses two locks, a **strong pattern** must have the form $\text{Owns}_p(u) = \{t_1\}$ and $\text{Blocks}_p(u) = \{t_2\}$ or \emptyset , where t_1, t_2 are the two locks used by p .

A **strong pattern** represents the situation where p acquires locks t_1 and t_2 , before releasing t_2 and then trying to take it back (in this case $\text{Blocks}_p(u) = \{t_2\}$). The reason for blocking in this case is that some other process has taken t_2 after p released it, hence after p has taken t_1 . **Weak patterns** do not impose such a constraint on acquiring locks. Distinguishing between **weak** or **strong patterns** of local runs is crucial when we want to form a global run from local runs.

The next lemma gives a characterization of winning strategies in terms of patterns.

Lemma 13. *Let $\sigma = (\sigma_p)_{p \in \text{Proc}}$ be a **strategy** and $\mathbb{P}^\sigma = (\mathbb{P}_p^\sigma)$ its **behavior**. Then σ is **not winning** if and only if for every p there is a **pattern** $\text{Owns}_p \longrightarrow \text{Blocks}_p$ in \mathbb{P}_p^σ such that all conditions below hold:*

- $\bigcup_{p \in \text{Proc}} \text{Blocks}_p \subseteq \bigcup_{p \in \text{Proc}} \text{Owns}_p$,
- the sets Owns_p are pairwise disjoint,
- there exists a total order $<$ on T such that for all p , if p admits a **strong pattern** $\{t\} \implies \text{Blocks}_p$ then $t < t'$, where t' is the other lock used by p .

Proof. Suppose σ is not **winning**, let u be a global σ -run ending in a **deadlock**, and for each process p let u_p be the corresponding local run.

For every p , the local run u_p has to be **risky**, otherwise u_p could be extended into a longer run consistent with σ . Thus u_p has a **pattern** $\text{Owns}_p \longrightarrow \text{Blocks}_p$ to \mathbb{P}_p^σ .

We check that these **patterns** meet all requirements of the lemma. Clearly as we are in a **deadlock**, the only actions available to each process need to acquire locks that are already taken, hence the first condition is satisfied. Furthermore, no two processes can own the same lock, implying the second condition. Finally, let $<$ be a total order on locks compatible with the order in u between the last operation on each lock, that is: $t < t'$ if the last operation on t in u is before the last one on t' . If one of t, t' is untouched throughout the run then the order is taken arbitrarily.

Consider a process p such that u_p has a **strong pattern** $\{t\} \Longrightarrow \text{Blocks}_p$. So u_p is of the form $u_1(a, \text{acq}_t)u_2(b, \text{rel}_{t'})u_3$ with no action on t in u_2 or u_3 . Hence $t < t'$ since the last action on t is before the last action on t' .

We now prove the other direction of the lemma. Suppose that for each p there is a **pattern** $\text{Owns}_p \longrightarrow \text{Blocks}_p$ in \mathbb{P}_p^σ such that those **patterns** satisfy all three conditions of the lemma. Let $<$ be a total order on locks witnessing the third condition.

For all p there exists a **risky** local run u_p yielding **pattern** $\text{Owns}_p \longrightarrow \text{Blocks}_p$. We combine these runs into a global run. We start by executing one by one in some arbitrary order all the u_p such that $\text{Owns}_p = \emptyset$. After executing each such run, all locks are free, hence we can execute the next one. This leaves all locks free at the end.

For all p such that $\text{Owns}_p = \{t\}$ and $\text{Owns}_p \dashrightarrow \text{Blocks}_p$ is weak, we can decompose u_p as $u_1^p(a, \text{acq}_t)u_2^p$ with u_1^p **neutral** and u_2^p not containing any operation on locks. We execute u_1^p , which leaves all locks free as it is **neutral**.

At this point we consider all the processes p where u_p has a **strong pattern** $\{t_p\} \Longrightarrow \text{Blocks}_p$. We execute these runs u_p according to the order $<$ on t_p . This is possible, as for each such p we have $t_p < t'_p$, where t'_p is the other lock of p . The linear order guarantees that before executing u_p all locks greater or equal to t_p according to $<$ are free. In particular, t_p and t'_p are free, thus we can execute u_p .

After the first two steps, all locks are free except for locks t_p of processes p with a **strong pattern** $\{t_p\} \Longrightarrow \text{Blocks}_p$. We come back to the u_p with **weak patterns**. We execute the remaining parts of the u_p , namely $(a, \text{acq}_t)u_2^p$ as above. As u_2^p contains no operation on locks, we only need t to be free to execute this run. As all Owns_p are disjoint, and all locks taken at that point belong to some other Owns_p , t is free, hence all those runs can be executed.

Finally, the remaining runs are the ones such that $\text{Owns}_p = \{t, t'\}$ contains both locks of p . As all Owns_p are disjoint, both these locks are free, hence u_p can be executed as p can only use these two locks.

We have executed all local runs, therefore we reach a configuration where all processes have to acquire a lock from $\bigcup_{p \in \text{Proc}} \text{Blocks}_p$ to keep running, and all locks in $\bigcup_{p \in \text{Proc}} \text{Owns}_p$ are taken. As $\bigcup_{p \in \text{Proc}} \text{Blocks}_p \subseteq \bigcup_{p \in \text{Proc}} \text{Owns}_p$, we have reached a **deadlock**. \square

Thanks to Lemma 13, in order to decide if there is a **winning strategy** for a given system it is enough to come up with a set of **patterns** \mathbb{P}_p for each process p and show two properties:

- there exists a **strategy** σ such that $\mathbb{P}_p^\sigma \subseteq \mathbb{P}_p$ for each process p ;
- the sets of **patterns** \mathbb{P}_p do not meet the conditions given by Lemma 13.

Note that in the first condition we only require an inclusion because by the previous lemma, the less **patterns** a **strategy** allows, the less likely it is to create a **deadlock**.

We start by showing that given a set of **patterns** for each process, we can check the first condition in polynomial time.

Lemma 14. *Given a **behavior** $(\mathbb{P}_p)_{p \in Proc}$, it is decidable in PTIME whether there exists a **strategy** σ such that for every p : $\mathbb{P}_p^\sigma \subseteq \mathbb{P}_p$.*

Proof. First of all note that we only need to check for each p that there exists a **local strategy** σ_p that does not allow any **risky** runs whose **pattern** is not in \mathbb{P}_p .

Let $p \in Proc$, let \mathcal{A}_p be its transition system. We extend it in a similar way as in Remark 1, by adding some information in the states.

We already assumed that the states contained the information of which locks are currently owned by p . For states where p owns a lock t_1 , we store an additional bit of information saying whether p released its other lock t_2 since acquiring t_1 for the last time.

This way, the **risky** nature of local runs and their **patterns** depend only on the state in which they end and its outgoing transitions. For instance if a state has no outgoing transitions and is such that when reaching it p holds t_1 and released t_2 since acquiring it, then the **pattern** of runs ending there is $\{t_1\} \Rightarrow \emptyset$. If this **pattern** is not in \mathbb{P}_p then we declare this state as bad.

Formally, a state is called good if there exists a set of outgoing transitions containing all environment transitions and such that:

- either it contains a transition with no acquire operation,
- or the set of locks acquired by those transitions is such that runs ending in that state have their **pattern** in \mathbb{P}_p .

Otherwise, a state is called bad. If all states of the system are good then clearly there is a suitable **local strategy**.

We simply iteratively delete bad states and all their in-going transitions. If one of those transitions is controlled by the environment, we declare its source state as bad (as reaching that state would allow the environment to take that transition, leading us to a bad state). Note that deleting transitions may create more bad states by reducing the choice of the system. If we end up deleting all states, we conclude that there is no suitable **local strategy**. Otherwise the subsystem we obtain has only good states, allowing us to get a **strategy** matching the input set of **patterns**. \square

Proposition 15. *The **deadlock avoidance control problem** for 2LSS is decidable in Σ_2^P .*

Proof. The algorithm first guesses a set of **patterns** \mathbb{P}_p for each process p (each set is of bounded size). By Lemma 14, we can then check in polynomial time if there exists a **strategy** $\sigma = (\sigma_p)_{p \in Proc}$ with σ_p admitting only **patterns** in \mathbb{P}_p . Next, in CONP we can check that there is no selection of **patterns** $\pi_p \in \mathbb{P}_p$ for each p , and total order on locks $<$ satisfying Lemma 13.

For the correctness of the algorithm observe that if there exists a **winning strategy** then it suffices to guess the sets of **patterns** it allows. Conversely suppose the algorithm guessed a **behavior** not meeting the requirements of Lemma 13. Then by the first step of the algorithm we have a **strategy** that is **winning**. \square

Theorem 5. *The **deadlock avoidance control problem** for **2LSS** is Σ_2^P -complete.*

Proof. The upper bound follows from Proposition 15.

For the lower bound we reduce from the $\exists\forall$ -SAT problem. Suppose we are given a formula in 3-disjunctive normal form $\bigvee_{i=1}^k \alpha_i$, so each α_i is a conjunction of three literals $\ell_1^i \wedge \ell_2^i \wedge \ell_3^i$ over a set of variables $\{x_1, \dots, x_n, y_1, \dots, y_m\}$. The question is whether the formula $\varphi = \exists x_1, \dots, x_n \forall y_1, \dots, y_m, \bigvee_{i=1}^k \alpha_i$ is true.

We construct a **2LSS** for which there is a **winning strategy** iff the formula is true. The **2LSS** will use locks:

$$\{t_i \mid 1 \leq i \leq k\} \cup \{x_i, \overline{x_i} \mid 1 \leq i \leq n\} \cup \{y_j, \overline{y_j} \mid 1 \leq j \leq m\}.$$

For all $1 \leq i \leq n$ we have a process p_i with six states, as depicted in Fig. 3. In that process the system has to take both x_i and $\overline{x_i}$, and then may release one of them before being blocked in a state with no outgoing transitions. Similarly, for all $1 \leq j \leq m$ we have a process q_j , in which the environment has to take y_j or $\overline{y_j}$, and then is blocked.

For each clause α_i we have a process $p(\alpha_i)$ which just has one transition acquiring lock t_i towards a state with a local loop on it. Hence to block all those processes the environment needs to have all t_i taken by other processes. Those processes are not necessary but help to clarify the proof.

She can do that with our last kind of processes. For each clause α_i and each literal ℓ of α_i there is a process $p_i(\ell)$. There the process has to acquire t_i and then ℓ before entering a state with a self-loop.

In order to block all processes $p(\alpha_i)$, each t_i has to be taken by a process $p_i(\ell)$ for some literal ℓ of α_i . For process $p_i(\ell)$ to be blocked, lock ℓ has to be taken before, by some p_i or q_j .

A **strategy** for the system amounts to choosing whether p_i should release x_i or $\overline{x_i}$, for each $i = 1, \dots, n$. It may also choose to release neither. Since the environment has a global view of the system, it can afterwards choose one of $y_j, \overline{y_j}$ in process q_j , for each $j = 1, \dots, m$. Those choices represent a valuation, the free lock remaining being the satisfied literal.

If the formula φ is true, then the system chooses the valuation of the x_i 's in order to make φ true. As soon as processes p_i, q_j have reached their final state, we also have a valuation for the y_j 's. At this point there is at least one clause α_i true, so with all its literals $\ell_1^i, \ell_2^i, \ell_3^i$ true. Observe that among the 4 processes $p(\alpha_i)$ and $p_i(\ell_j^i)$ at least one can reach its self-loop, namely the one that acquires t_i first. Hence, the system does not deadlock.

Otherwise, if the formula φ is not true, then for each choice of the system for the x_i 's, the environment can chose afterwards a suitable valuation of the y_j 's that falsifies φ ("afterwards" means that we look at a suitable scheduling of the acquire actions). For such a valuation, for every α_i there is some literal ℓ_i of α_i that is false. Consider the scheduling that lets $p_i(\ell_i)$ acquire t_i first. Since ℓ_i is taken, this implies that $p_i(\ell_i)$ is blocked. Also, $p(\alpha_i)$ is blocked because of t_i . The other two processes $p_i(\ell)$ with $\ell \neq \ell_i$ are also blocked because of t_i . So overall the entire system is blocked.

□

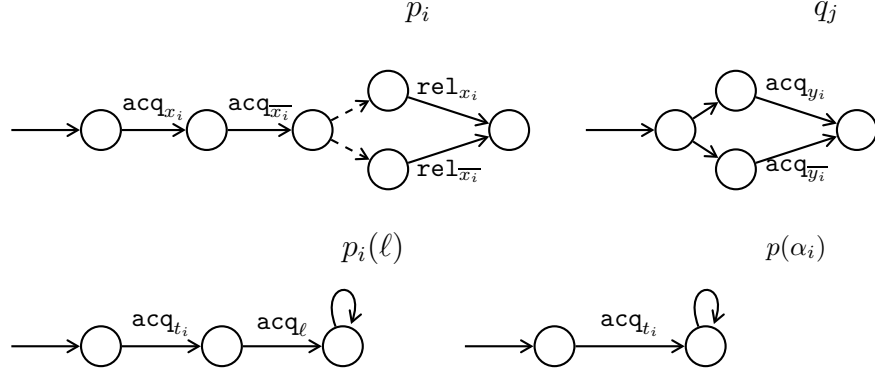


Figure 3: The processes used in the reduction in Theorem 5. Transitions of the system are dashed.

3.2 Locally live strategies in NP

We now consider the case of 2LSS with **locally live strategies**. Such a strategy ensures that no process blocks by reaching a state with no outgoing transitions. Hence a process can only block if all its available transitions need to acquire a lock, but all these locks are taken. This restriction prevents a construction like the one used to obtain the lower bound of Proposition 5.

In the last subsection we were guessing a **behavior** of a strategy and then checking in CONP if the condition from Lemma 13 does not hold. Here we show that this check can be done in PTIME.

The argument is unfortunately quite lengthy. We represent a **behavior** as a lock graph $G_{\mathbb{P}}$, with vertices corresponding to locks and edges to patterns. Then, thanks to local liveness, instead of Lemma 13 we get Lemma 20 characterizing when a strategy is not winning by the existence of a subgraph of $G_{\mathbb{P}}$, called sufficient deadlock scheme. The main part of the proof is a polynomial time algorithm for deciding the existence of sufficient deadlock schemes.

As we are in a **locally live** framework, some **patterns** of local runs are impossible. We do not have patterns of the form $O \rightarrow \emptyset$ as a local run can block only because it requires some locks that are taken. This leaves us with two possible types of patterns $\{t_1\} \rightarrow \{t_2\}$ and $\emptyset \rightarrow B$. We represent the set of patterns of the first type as a graph. An edge labeled by p from t_1 to t_2 represents a pattern $\{t_1\} \rightarrow \{t_2\}$ of a process p . Recall that this corresponds to a local run ending in a situation when p holds t_1 and all actions need to acquire t_2 .

Definition 16 (Lock graph $G_{\mathbb{P}}$). For a **behavior** $\mathbb{P} = (\mathbb{P}_p)_{p \in \text{Proc}}$, we define a labeled graph $G_{\mathbb{P}} = \langle T, E_{\mathbb{P}} \rangle$, called **lock graph**, whose nodes are locks and whose edges are either weak or strong. Edges are labeled by processes.

There is a **weak edge** $t_1 \xrightarrow{p} t_2$ in $G_{\mathbb{P}}$ whenever there is a **weak pattern** $\{t_1\} \dashrightarrow \{t_2\}$ in \mathbb{P}_p . There is a **strong edge** $t_1 \xRightarrow{p} t_2$ whenever there is a **strong pattern** $\{t_1\} \Rightarrow \{t_2\}$ in \mathbb{P}_p and there is no **weak pattern** $\{t_1\} \dashrightarrow \{t_2\}$ in \mathbb{P}_p . We write

$t_1 \xrightarrow{p} t_2$ when the type of the edge is irrelevant.

- ⌈ A path (resp. cycle) in $G_{\mathbb{P}}$ is *simple* if all its edges are labeled by different processes. A cycle is *weak* if it contains some *weak edge*, and *strong* otherwise.

Note that the information contained in the graph does not use *patterns* of the form $\emptyset \rightarrow B$. The next definition provides some notions to incorporate them next to our graph representation.

Definition 17. For a *behavior* $\mathbb{P} = (\mathbb{P}_p)_{p \in Proc}$, a process p is called *solid* if there is no *pattern* of the form $\emptyset \rightarrow Blocks_p$ in \mathbb{P}_p ; otherwise it is called *fragile*.

- ⌈ A process p is *Z-lockable* in \mathbb{P} if there is a *pattern* $\emptyset \rightarrow B$ in \mathbb{P}_p with $B \subseteq Z$. Note that a process is *fragile* if and only if it is *Z-lockable* for some Z .
- ⌈ A *solid edge* of $G_{\mathbb{P}}$ is one that is labeled by a *solid* process. A *solid cycle* is one that only has *solid edges*.

What the previous definition says is that a *solid* process needs to take a lock to be blocked, whereas a *fragile* one can be blocked without owning a lock. So *solid* processes must be taken into account in the deadlock schemes defined next:

Definition 18 (*Z-deadlock scheme*). Consider a *behavior* $\mathbb{P} = (\mathbb{P}_p)_{p \in Proc}$, and the associated *lock graph* $G_{\mathbb{P}}$. Let $Z \subseteq T$ be a set of locks. We set $Proc_Z$ as the set of processes whose both accessible locks are in Z .

- ⌈ A *Z-deadlock scheme for \mathbb{P}* is a partial function $ds_Z : Proc_Z \rightarrow E_{\mathbb{P}} \cup \{\perp\}$ assigning 0 or 1 edge of $G_{\mathbb{P}}$ to each process of $Proc_Z$ such that:
 - For all $p \in Proc_Z$, if $ds_Z(p) \neq \perp$ then $ds_Z(p)$ is a p -labelled edge of $G_{\mathbb{P}}$.
 - If $p \in Proc_Z$ is *solid* then $ds_Z(p) \neq \perp$.
 - For all $t \in Z$ there exists a unique $p \in Proc_Z$ such that $ds_Z(p)$ is an outgoing edge of t .
 - The subgraph of $G_{\mathbb{P}}$ restricted to $ds_Z(Proc_Z)$ does not contain any *strong cycle*.

The idea of the previous definition is that a *Z-deadlock scheme* witnesses a way to reach a configuration in which all locks of Z are taken, and all processes using those locks are blocked. Each process from $Proc_Z$ is mapped to an edge telling which lock it holds in the *deadlock* configuration and which one it needs in order to advance. A process not holding any lock is mapped to \perp . For every lock in Z there is a unique outgoing edge in ds_Z , corresponding to the process owning that lock. Note that this implies that the subgraph induced by ds_Z is a union of cycles, with some non-branching paths going into these cycles.

Definition 19 (*Sufficient deadlock scheme*). A *sufficient deadlock scheme* for a *behavior* \mathbb{P} is a *Z-deadlock scheme* ds_Z for \mathbb{P} such that for every process $p \in Proc$ either $ds_Z(p)$ is an edge of the lock graph $G_{\mathbb{P}}$, or p is *Z-lockable* in \mathbb{P} .

We now prove a key result: a **strategy** is not **winning** if and only if its lock graph admits a **sufficient deadlock scheme**. This allows us to check the existence of a **winning strategy** by non-deterministically guessing a **behavior**, verifying that there exists a **strategy** respecting it, computing the corresponding **lock graph** and then checking if there exists a **sufficient deadlock scheme** for it. The rest of this section provides a method to do these steps in polynomial time, which leads to an NP algorithm.

Lemma 20. *Consider a **locally live control strategy** σ and $\mathbb{P}^\sigma = (\mathbb{P}_p^\sigma)_{p \in \text{Proc}}$ the **behavior** of σ . The **strategy** σ is **not winning** if and only if there is a **sufficient deadlock scheme** for \mathbb{P}^σ .*

Proof. Suppose σ is not **winning**. Then by Lemma 13, there exist **patterns** $\text{Owns}_p \longrightarrow \text{Blocks}_p \in \mathbb{P}_p^\sigma$ for each p such that:

- $\bigcup_{p \in \text{Proc}} \text{Blocks}_p \subseteq \bigcup_{p \in \text{Proc}} \text{Owns}_p$,
- the sets Owns_p are pairwise disjoint,
- there exists a total order \leq on T such that for all p , if $\text{Owns}_p \longrightarrow \text{Blocks}_p$ is a **strong pattern** $\{t\} \Longrightarrow \text{Blocks}_p$ then $t \leq t'$ where t' is the other lock used by p (besides t).

Let $Z = \bigcup_{p \in \text{Proc}} \text{Owns}_p$, and for all $p \in \text{Proc}_Z$, define $ds(p)$ as:

- \perp if $\text{Owns}_p = \emptyset$,
- $t_1 \xrightarrow{p} t_2$ if $\mathbb{P}_p = \{t_1\} \longrightarrow \{t_2\}$ (whether it is **strong** or **weak** is irrelevant for now). This edge exists by definition of $G_{\mathbb{P}}$.

Note that there are no other possible cases above, as σ is **locally live** and thus Blocks_p cannot be empty.

We show that ds_Z is a sufficient deadlock scheme for \mathbb{P}^σ by checking the four conditions from Definition 18. The first condition holds by definition of ds_Z . For the second condition let $p \in \text{Proc}_Z$ and suppose p is \mathbb{P} -solid. Thus, Owns_p is not empty and $ds(p) \neq \perp$. For the third condition let $t \in Z$. As Z is the disjoint union of the sets Owns_p there exists a unique $p \in \text{Proc}_Z$ such that $t \in \text{Owns}_p$, so a unique edge $ds(p)$ outgoing from from t . For the last condition note that for all **strong edges** $t \xrightarrow{p} t'$ the pattern $\text{Owns}_p \Longrightarrow \text{Blocks}_p$ must be strong as well, hence $t \leq t'$. As \leq is a total order on locks, there cannot be any **strong cycle**.

Finally, suppose that $p \notin \text{Proc}_Z$ or $ds(p) = \perp$. In both cases $\text{Owns}_p = \emptyset$, thus p is Blocks_p -lockable, and hence Z -lockable as $\text{Blocks}_p \subseteq Z$. As a consequence, ds is a **sufficient deadlock scheme** for \mathbb{P} .

For the other direction, suppose we have a **sufficient Z -deadlock scheme** ds for \mathbb{P}^σ . As $ds(\text{Proc}_Z)$ does not contain any **strong cycle**, we can pick a total order \leq on locks such that for all **strong edges** $t_1 \xrightarrow{p} t_2$ belonging to $ds(\text{Proc}_Z)$, we have $t_1 \leq t_2$.

By definition of **sufficient Z -deadlock scheme**, for each process $p \in \text{Proc}$ we can find a **pattern** $\text{Owns}_p \longrightarrow \text{Blocks}_p \in \mathbb{P}_p$ with the following properties.

- If $ds(p) = \perp$ or $p \notin Proc_Z$ then p is Z -lockable. Hence we can choose $Blocks_p \subseteq Z$ such that $\emptyset \longrightarrow Blocks_p \in \mathbb{P}_p$.
- If $ds(p) = t_1^p \xrightarrow{p} t_2^p$ then there exists a pattern $\{t_1^p\} \longrightarrow \{t_2^p\} \in \mathbb{P}_p$ with $\{t_1^p, t_2^p\} \subseteq Z$.

As all locks of Z have exactly one outgoing edge in $ds(Proc_Z)$, and as all $Owns_p$ with $p \notin Proc_Z$ or $ds(p) = \perp$ are empty, the sets $Owns_p$ are pairwise disjoint and $\bigcup_{p \in Proc} Blocks_p \subseteq Z \subseteq \bigcup_{p \in Proc} Owns_p$.

By the definition of \leq , for all strong patterns $\{t_1^p\} \Longrightarrow \{t_2^p\}$ we have $t_1^p \leq t_2^p$. Recall that $Blocks_p$ is never empty because σ is locally live, so $\{t\} \Longrightarrow Blocks_p$ can only have the form $\{t_1^p\} \Longrightarrow \{t_2^p\}$.

By Lemma 13, the above shows that σ is not winning. \square

From now on we fix a behavior \mathbb{P} . We will show how to decide if there is a sufficient deadlock scheme for \mathbb{P} in PTIME. For this we need to be able to certify in PTIME that there is no Z -deadlock scheme as in Definition 18. Our approach will be to eliminate edges from $G_{\mathbb{P}}$ and construct a Z -deadlock scheme incrementally, on bigger and bigger sets of locks Z . We show that this process either exhibits a set Z that is big enough to provide a sufficient deadlock scheme for \mathbb{P} , or it fails, and in this case there is no sufficient deadlock scheme for \mathbb{P} .

The next lemma shows that a Z -deadlock scheme can be constructed incrementally. Suppose we already have a set Z on which we know how to construct a Z -deadlock scheme. Then the lemma says that in order to get a sufficient deadlock scheme for \mathbb{P} it is enough to focus on $G_{\mathbb{P}} \setminus Z$.

Lemma 21. *Let $Z \subseteq T$ be such that there is no solid edge from Z to $T \setminus Z$ in $G_{\mathbb{P}}$. Suppose that $ds_Z : Proc_Z \rightarrow E \cup \{\perp\}$ is a Z -deadlock scheme for \mathbb{P} . If there exists some sufficient deadlock scheme for \mathbb{P} then there is one which is equal to ds_Z over $Proc_Z$.*

Proof. Suppose ds is a sufficient deadlock scheme for \mathbb{P} , so ds is a B -deadlock scheme for some $B \subseteq T$ such that for every $p \in Proc$ either $ds(p)$ is an edge in $G_{\mathbb{P}}$ or p is B -lockable in \mathbb{P} . We construct a $(B \cup Z)$ -deadlock scheme ds' which is equal to ds_Z over $Proc_Z$. Then we show that the deadlock scheme is sufficient.

For every process $p \in Proc$, we define $ds'(p)$ as:

- $ds_Z(p)$ if $p \in Proc_Z$,
- \perp if p labels an edge from Z to $T \setminus Z$,
- $ds(p)$ otherwise.

We check that ds' is a $(B \cup Z)$ -deadlock scheme. The assumption is that there are no solid edges from Z to $T \setminus Z$, thus all processes mapped to \perp are fragile. Every lock $t \in B \cup Z$ has at most one outgoing edge in ds' , since it can only come from ds_Z , if $t \in Z$, or from ds , if $t \in B \setminus Z$. We verify that there is at least one outgoing edge. By definition of Z -deadlock scheme there is one outgoing edge from every lock in Z . A lock $t \in B \setminus Z$ has exactly one outgoing edge in

$ds(Proc)$, and this edge is conserved in ds' . Finally, there cannot be any **strong cycle** in $ds'(Proc)$ as there are none within Z or $B \setminus Z$ and there are no edges from Z to $T \setminus Z$ in ds' .

It remains to show that ds' is a **sufficient deadlock scheme** for \mathbb{P} . Let $p \in Proc$ be an arbitrary process. We make a case distinction on the locks of p . The first case is when both locks are in Z . If p is **solid** then $ds'(p) = ds_Z(p) \neq \perp$. If p is **fragile** then it is Z -**lockable** by definition of **fragile**. The second case is when one lock is in $B \setminus Z$ and the other in $B \cup Z$. If p is **solid** then $ds(p)$ must be defined because ds is a **sufficient deadlock scheme**. We must have $ds'(p) = ds(p)$ as there are no solid edges from Z to $T \setminus Z$. If p is **fragile** then p is $B \cup Z$ -**lockable**. The final case is when one lock of p is not in $B \cup Z$. Since ds is a **sufficient deadlock scheme**, p is B -**lockable**, so it is $B \cup Z$ -**lockable**. \square

Recall that we have fixed a **behavior** \mathbb{P} , and that $G_{\mathbb{P}}$ is its **lock graph**. We will describe several polynomial-time algorithms operating on a graph $H = (T, E_H)$ and a set Z of locks. Graph H will be obtained by erasing some edges from $G_{\mathbb{P}}$. We will say that H has a **sufficient deadlock scheme** to mean that there is a **deadlock scheme** using only edges in H that is **sufficient** for \mathbb{P} . Each of those algorithms will either eliminate some edges from H or extend Z , while maintaining the following three invariants.

Invariant 1. $G_{\mathbb{P}}$ has a **sufficient deadlock scheme** if and only if H does.

Invariant 2. There are no **solid edges** from Z to $T \setminus Z$ in H .

Invariant 3. There exists a Z -**deadlock scheme** in $G_{\mathbb{P}}$.

Invariant 1 expresses that the edges we removed from $G_{\mathbb{P}}$ to get H were not useful for the **deadlock scheme**. Invariant 2, along with Lemma 21, guarantees that we can always extend a **deadlock scheme** over Z to a **sufficient** one if it exists. Invariant 3 maintains the existence of a **deadlock scheme** over Z .

We extend Z as much as we can while maintaining those. In the end we either obtain a **sufficient deadlock scheme** (that is, Z is large enough so that all processes outside of $Proc_Z$ are Z -**lockable**), or a non-sufficient one that we cannot extend anymore.

We may also at some point observe contradictions in the edges of H that forbid any **sufficient deadlock scheme**, in which case we can conclude immediately thanks to Invariant 1.

We start with H being the given $G_{\mathbb{P}}$ and $Z = \emptyset$. The invariants are clearly satisfied.

Definition 22 (Double and solo solid edges). Consider a **solid process** p . We say that there is a **double solid edge** $t_1 \xleftrightarrow{p} t_2$ in H if both $t_1 \xrightarrow{p} t_2$ and $t_1 \xleftarrow{p} t_2$ exist in H . We say that $t_1 \xrightarrow{p} t_2$ in H is a **solo solid edge** if there is no $t_1 \xleftarrow{p} t_2$ in H .

Our first algorithm looks for a **solo solid edge** $t_1 \xrightarrow{p} t_2$ and erases all other outgoing edges from t_1 . It is correct as a **deadlock scheme** for H has to map p

to the edge $t_1 \xrightarrow{p} t_2$ and there must be at most one outgoing edge from every lock.

We repeat this algorithm till no edges are removed. If some call of the algorithm fails then there is no **sufficient deadlock scheme** for H . Otherwise the resulting H satisfies the property:

(Trim) if a lock t in $H \setminus Z$ has an outgoing **solo solid edge** then it has no other outgoing edges.

◻ We call H **trimmed** if it satisfies property (Trim).

Algorithm 1 Trimming the graph

```

1: Look for  $t \in H \setminus Z$  with a solo solid edge  $t \xrightarrow{p} t' \in E_H$  and some other
   outgoing edges
2: If there is no such edge then stop and report success.
3: for every edge  $t \xrightarrow{q} t'' \in E_H$  from  $t$  with  $q \neq p$  do
4:   if  $q$  is solid and  $t \xleftarrow{q} t'' \notin E_H$  then
5:     return “no  $H$ -deadlock scheme”
6:   else
7:     Erase  $t \xrightarrow{q} t''$ 
8:   end if
9: end for

```

Lemma 23. *Suppose (H, Z) satisfies Invariants 1 to 3. If Algorithm 1 fails then H has no **sufficient deadlock scheme**. After a successful execution of the algorithm all the invariants are still satisfied. If a successful execution does not remove an edge from H then H satisfies (Trim).*

Proof. Let H' be the graph after an execution of Algorithm 1. Observe that the algorithm does not change Z . If $H = H'$ then (Trim) holds. If the algorithm fails then there is a lock with two outgoing **solo solid edges**. Thus it is impossible to find a **sufficient deadlock scheme** for H .

Finally, if the algorithm succeeds but H' is smaller than H , we must show that all the invariants hold. Since the algorithm does not change Z , Invariants 2 and 3 continue to hold. For Invariant 1, suppose $t \xrightarrow{p} t'$ is the edge found by the algorithm. Observe that if H has a **sufficient deadlock scheme** ds_H then $ds_H(p)$ must be this edge. So ds_H is also a **sufficient deadlock scheme** for H' . In the other direction, a **sufficient deadlock scheme** for H' is also **sufficient** for H , as H' is a subgraph of H and $Proc_H = Proc_{H'}$. The latter holds because H' has the same locks as H . ◻

Our second algorithm searches for cycles formed by **solid edges** and eventually adds them to Z . If such a cycle is **weak** then it can be added to Z . If the cycle is **strong**, it may still be the case that its reversal is **weak** (see p_1, p_2, p_3 in Figure 4). More precisely it may be the case that for every **solid edge** $t_i \xrightarrow{p_i} t_{i+1}$ in the cycle there is also a reverse edge $t_i \xleftarrow{p_i} t_{i+1}$ (which is **solid** by definition,

since p_i is so). If the reversed cycle is also **strong** then there is no H -**deadlock scheme**. Otherwise, it is **weak** and it can be added to Z . The result still satisfies the invariants thanks to the (Trim) property of H .

Figure 4 presents a case where an application of Algorithm 2 followed by Algorithm 1 allows us to detect an inconsistency in the **solid edges**, proving the absence of any **deadlock scheme**.

Algorithm 2 Find **solid cycles** and add them to Z if possible.

```

1: Look for a simple cycle of solid edges  $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$  not inter-
   sectioning  $Z$  and all  $t_i$  distinct
2: If there is no such cycle, stop and report success.
3: if all the edges on the cycle are strong then
4:   if for some  $j$  there is no reverse edge  $t_j \xleftarrow{p_j} t_{j+1} \in E_H$  then
5:     return “no  $H$ -deadlock scheme”
6:   else if all edges  $t_j \xleftarrow{p_j} t_{j+1}$  are strong then
7:     return “no  $H$ -deadlock scheme”
8:   end if
9: end if
10:  $Z \leftarrow Z \cup \{t_1, \dots, t_k\}$ 
11: For every  $t_i$  remove from  $E_H$  all edges outgoing from  $t_i$  except for  $t_i \xleftarrow{p_i} t_{i+1}$ .
12: if some solid process  $p$  has no edge in  $H$  then
13:   return “no  $H$ -deadlock scheme”
14: end if
15: repeat
16:   Apply Algorithm 1
17: until no more edges are removed from  $H$ 

```

Lemma 24. *Suppose (H, Z) satisfies the Invariants 1 to 3 and H is **trimmed**. If the execution of Algorithm 2 does not fail then the resulting H and Z also satisfy the invariants and (Trim). If the execution fails then there is no **sufficient deadlock scheme** for H .*

Proof. Suppose that the algorithm finds a **simple cycle** $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$ where all p_i are **solid processes**, and all t_i are distinct. By definition of a **simple cycle**, all p_i are distinct as well. If there is a **sufficient deadlock scheme** for H then it should assign either $t_i \xrightarrow{p_i} t_{i+1}$ or $t_i \xleftarrow{p_i} t_{i+1}$ to p_i .

We examine the cases when the algorithm fails. The first reason for failure may appear when all the edges on the cycle are **strong**. If for some j there is no reverse edge $t_j \xleftarrow{p_j} t_{j+1}$ in E_H then a **sufficient deadlock scheme** for H , call it ds_H , should assign the edge $t_j \xrightarrow{p_j} t_{j+1}$ to p_j . In consequence, as ds_H has to give each t_i at most one outgoing edge, all the edges in the cycle should be in the image of ds_H . This is impossible as the cycle is **strong**. When there are reverse edges $t_i \xleftarrow{p_i} t_{i+1} \in E_H$ for all i , the algorithm fails if all of them are **strong**. Indeed, there cannot be a **sufficient deadlock scheme** for H in this case.

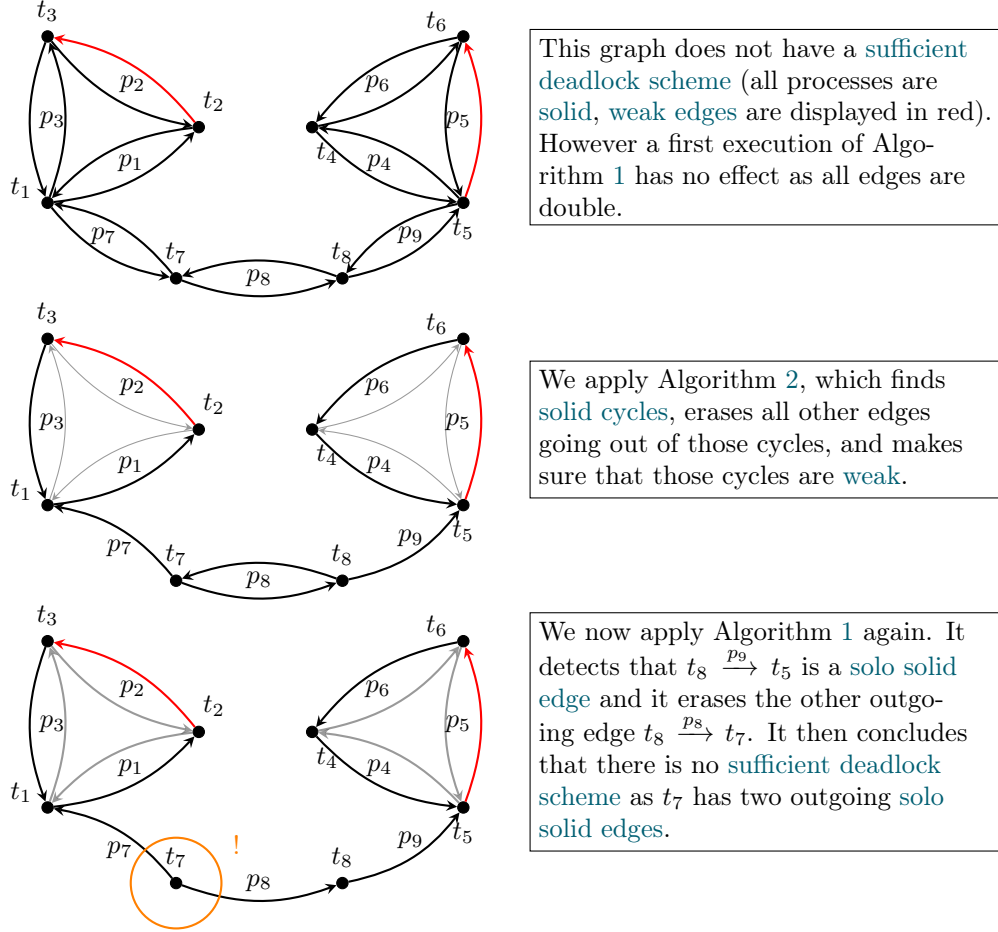


Figure 4: An example of application of Algorithm 1 and Algorithm 2.

The last reason for failure is when there is some **solid process** p and all p -labeled edges were removed by the algorithm. These must be edges of the form $t_i \xrightarrow{p} t$ that are not on the cycle, for some $i = 1, \dots, k$. Those edges cannot belong to a **deadlock scheme** as it has to contain the cycle in one direction or the other and thus cannot contain other outgoing edges from that cycle. As a **deadlock scheme** cannot assign any edge to p , and p is **solid**, there cannot be a **sufficient deadlock scheme** in that case.

If the algorithm does not fail then either the cycle $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$ is **weak**, or its reverse is. Thanks to Lemma 23, we only need to show that our three invariants hold after line 11. Let (H', Z') be the values at that point. So $Z' = Z \cup \{t_1, \dots, t_k\}$, and H' is H after removing edges in line 11. We show that the invariants hold.

For Invariant 2, we observe that thanks to (Trim) for every lock in Z' there is exactly one outgoing edge in H' . So there is no **solid edge** from Z' to $H \setminus Z'$ as there was none from Z .

For Invariant 3, we extend our Z -**deadlock scheme** to Z' . We choose the cycle found by the algorithm or its reversal depending on which one is **weak**. For every p_i we define $ds_{Z'}(p_i)$ to be the edge in the chosen cycle. We set $ds_{Z'}(p) = \perp$ for all $p \in Proc_{Z'} \setminus Proc_Z$ other than p_1, \dots, p_k . We must show that such a p is necessarily **fragile**. Indeed, in this case p must have one of its locks t in Z , and the other one, t' , in $Z' \setminus Z$. By Invariant 2, there is no **solid edge** from t to t' in H . In H' all edges from t' to t are removed. So p is **fragile** as the algorithm does not fail at line 12.

For Invariant 1 suppose there is a **sufficient deadlock scheme** for H' . Then it is also a **sufficient deadlock scheme** for H , as H' is a subgraph of H over the same set of locks. For the other direction take ds_H , a **sufficient deadlock scheme** for H . By Lemma 21, as we showed that Invariant 2 is maintained, we can assume that ds_H is equal to $ds_{Z'}$ on Z' . We define a **deadlock scheme** $ds_{H'}$ for H' . If $ds_H(p) = \perp$ then $ds_{H'}(p) = \perp$. If the source edge of $ds_H(p)$ is in $H \setminus Z'$ then $ds_{H'}(p) = ds_H(p)$. This edge is guaranteed to exist in H' . If the two locks of p are both in Z' let $ds_{H'}(p) = ds_H(p) = ds_{Z'}(p)$. The remaining case is when $ds(p)$ is an edge $t \xrightarrow{p} t'$ with $t \in Z'$ and $t' \in H \setminus Z'$. In this case p is **fragile** as Z' has no **solid edges** leaving it. We can then set $ds_{H'} = \perp$. It can be verified that $ds_{H'}$ is a **sufficient deadlock scheme** for H' . \square

Lemma 25. *If Algorithm 2 succeeds but does not increase Z nor decrease H then (H, Z) satisfies three properties:*

H1 H is **trimmed**.

H2 H has no **solid cycle** that intersects $T \setminus Z$.

H3 Every **solid process** has an edge in H .

Proof. Since H was not modified, Algorithm 1 did not find any **solo solid edge** $t \xrightarrow{p} t'$ with other outgoing edges from t , hence property **H1** is satisfied.

By Lemma 24, Invariant 2 is satisfied, hence any **solid cycle** intersecting $T \setminus Z$ in H must be entirely in $T \setminus Z$. However if such a cycle existed then Algorithm 2

would not have stopped in line 2, and thus would have either failed or increased Z . There is therefore no such cycle intersecting $T \setminus Z$, hence property H2 is also satisfied.

If H3 were not satisfied then Algorithm 2 would have failed on lines 12-13. \square

Since in the rest of the algorithm we increase Z but do not modify H , the three properties stated in the previous lemma will continue to hold.

Definition 26. For any pair (H, Z) we define an equivalence relation between locks: $t_1 \equiv_H t_2$ if $t_1, t_2 \in T \setminus Z$ and there is a path of double solid edges in H between t_1 and t_2 .

Intuitively, once we have trimmed the graph and eliminated simple cycles of solid edges with Algorithm 2, the equivalence classes of \equiv_H are “trees” made of double solid edges (c.f. Lemma 28 below) with no outgoing edges (except for singletons, c.f. Lemma 27).

Lemma 27. If H satisfies property H1 and $t_1 \xrightarrow{p} t_2$ is in H for a solid process p then either the \equiv_H -equivalence class of t_1 is a singleton, or $t_1 \xleftarrow{p} t_2$ is in H , hence $t_1 \equiv_H t_2$.

Proof. If the \equiv_H -equivalence class of t_1 is not a singleton then there is a double solid edge from t_1 . By the (Trim) property, there cannot be any outgoing solo solid edge from t_1 , so $t_1 \xleftarrow{p} t_2$ must be in H , too. \square

Lemma 28. Suppose that H satisfies properties H1 and H2. Let $t_1, t_2 \in T \setminus Z$. If $t_1 \equiv_H t_2$ then H has a unique simple path of solid edges from t_1 to t_2 .

Proof. If $t_1 = t_2$ then any non-empty simple path of solid edges from t_1 to t_2 would contradict property H2, hence the empty path is the only simple path from t_1 to t_2 .

If $t_1 \neq t_2$ then by definition of \equiv_H there is a path of double solid edges from t_1 to t_2 , hence there is a simple path from t_1 to t_2 .

Suppose there exist two distinct simple paths from t_1 to t_2 , then by Lemma 27 all the locks on those paths are in the \equiv_H -equivalence class of t_1 and t_2 . Hence as $t_1 \notin Z$, there is a cycle of double solid edges intersecting $H \setminus Z$, contradicting property H2. \square

Our third algorithm looks for an edge $t_1 \xrightarrow{p} t_2$ with $t_1 \notin Z$ and $t_2 \in Z$, and adds the full \equiv_H -equivalence class C of t_1 to Z . This step is correct, as we can extend a Z -deadlock scheme to $(Z \cup C)$ -deadlock scheme by orienting edges in C , as displayed in the example in Figure 5.

Algorithm 3 Extending Z by locks that can reach Z

```

1: while there exists  $t_1 \xrightarrow{p} t_2 \in E_H$  with  $t_1 \notin Z$  and  $t_2 \in Z$  do
2:    $Z \leftarrow Z \cup \{t \in T \mid t \equiv_H t_1\}$ 
3: end while

```

Lemma 29. *Suppose H satisfies properties [H1](#), [H2](#) and [H3](#), and (H, Z) satisfies Invariants 1 to 3. After executing Algorithm 3, the new H and Z also satisfy all these properties, and H has no edges from $T \setminus Z$ to Z .*

Proof. Let (H', Z') be the pair obtained after executing Algorithm 3. Observe that $H' = H$, hence Invariant 1 holds. For the same reason [H1](#) and [H3](#) are still satisfied. Furthermore, as Z can only increase, [H2](#) continues to hold.

Let Z_{m+1} be the value of Z at the end of the m -th iteration. So $Z_{m+1} = Z_m \cup \{t \in T \mid t \equiv_H t_1\}$, where $t_1 \xrightarrow{p} t_2$ is the edge found in the guard of the while statement. We verify that Z_{m+1} satisfies Invariants 2 and 3 if Z_m does.

For Invariant 2, Lemma 27 says that there are no outgoing [solid edges](#) from the \equiv_H -equivalence class of t_1 , unless that class is a singleton. If it is a singleton, there are no outgoing solid edges from t_1 or $t_1 \xrightarrow{p} t_2$ is the only outgoing edge of t_1 . In both cases, there are no solid edges from Z_{m+1} to $T \setminus Z_{m+1}$ in H .

For Invariant 3 we extend a Z_m -deadlock scheme to Z_{m+1} . So we are given ds_m and construct ds_{m+1} . If the two locks of some process q are in Z_m then $ds_{m+1}(q) = ds_m(q)$. We set $ds_{m+1}(p)$ to be the edge $t_1 \xrightarrow{p} t_2$ found by the algorithm, so here $t_1 \in Z_{m+1} \setminus Z_m$ and $t_2 \in Z_m$. Let C be the \equiv_H -equivalence class of t_1 : $C = \{t \in T \mid t \equiv_H t_1\}$. By Lemma 28 there is a unique [simple path](#) from $t \in C$ to t_1 . Let $t \xrightarrow{q} t'$ be the first edge on this path. We set $ds_{m+1}(q)$ to be this edge. We set $ds_{m+1}(q) = \perp$ for all remaining processes q .

We verify that ds_{m+1} is a Z_{m+1} -[deadlock scheme](#). By the above definition every lock in C has a unique outgoing edge in ds_{m+1} , hence every lock in Z_{m+1} does. It is also immediate that the image of ds_{m+1} does not contain a [strong cycle](#) as it would need to be already in the image of ds_m (every lock has exactly one outgoing edge in ds_{m+1} and the path obtained by following those edges from an element of C leads to Z_m). It is maybe less clear that $ds_{m+1} \neq \perp$ for every [solid](#) $q \in Proc_{Z_{m+1}}$. Let q be a [solid process](#) in $Proc_{Z_{m+1}}$, and suppose ds_{m+1} is not defined by the procedure from the previous paragraph. If both locks of q are in Z_m then $ds_{m+1}(q)$ must be defined because $ds_m(q)$ is. If $q = p$, the process labeling the transition chosen by the algorithm, then $ds_{m+1}(q)$ is defined. Otherwise both locks of q are in C . Say these are t and t' . If neither $t \xrightarrow{q} t'$ is on the shortest path from t to t_1 , nor is $t \xleftarrow{q} t'$ on the shortest path from t' to t_1 then there must be a cycle in C . But this is impossible as we assumed that there are no [solid cycles](#) intersecting $T \setminus Z$ (property [H2](#)) and $Z \subseteq Z_m$. Hence $ds_{m+1}(q)$ is defined, and ds_{m+1} is a Z_{m+1} -[deadlock scheme](#).

All that is left to prove is that H has no edges from $T \setminus Z$ to Z , which is immediate as otherwise Algorithm 3 would not have stopped. \square

Our last algorithm looks for [weak cycles](#) in the remaining graph. If it finds one, it adds to Z not only all locks in the cycle but also their \equiv_H -equivalence classes.

Lemma 30. *Suppose H satisfies [H1](#), [H2](#) and [H3](#), (H, Z) satisfies Invariants 1 to 3, and moreover there are no edges from $T \setminus Z$ to Z . After an execution of Algorithm 4, H still satisfies [H1](#), [H2](#) and [H3](#), and the new (H, Z) satisfies Invariants 1 to 3.*

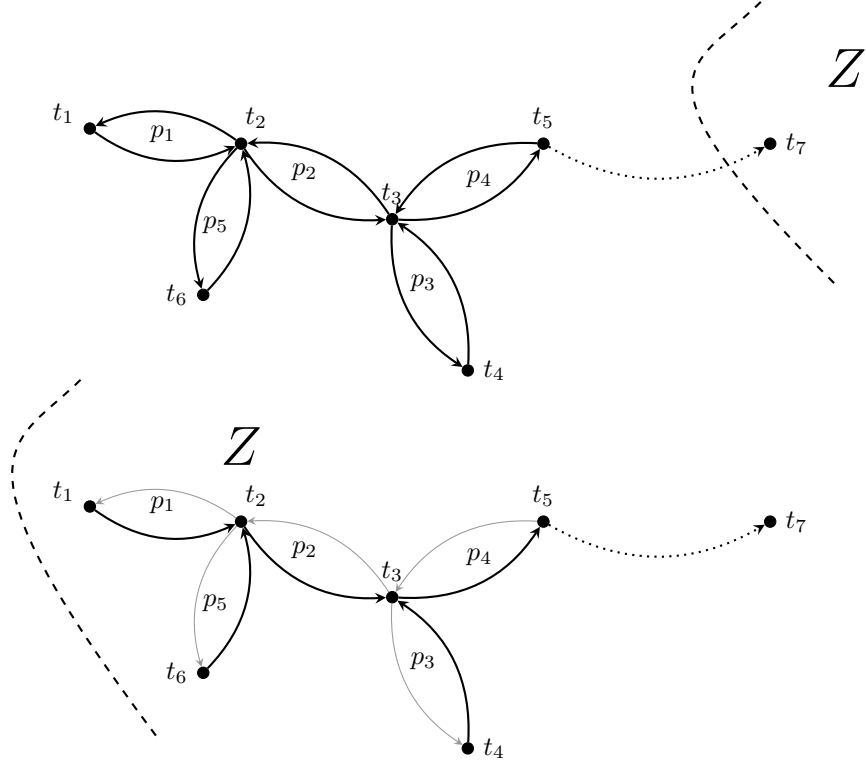


Figure 5: Illustration of Algorithm 3

Algorithm 4 Incorporating weak cycles

- 1: **if** there exists a weak cycle $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$ with $t_k \xrightarrow{p_k} t_1$ weak and $t_i \notin Z$ for some i , **then**
 - 2: $Z \leftarrow Z \cup \bigcup_{i=1}^k \{t \mid t \equiv_H t_i\}$
 - 3: **end if**
-

Proof. Let (H', Z') be the pair obtained after execution of Algorithm 3. Observe that $H' = H$, hence Invariant 1 holds. For the same reason H1 and H3 are still satisfied. Furthermore, as Z can only increase, so is H2. It remains to verify Invariants 2 and 3.

Consider the **weak cycle** found by the algorithm $t_1 \xrightarrow{p_1} t_2 \cdots \xrightarrow{p_k} t_{k+1} = t_1$, and note that $t_i \notin Z$ for all i . Let $Z' = Z \cup \bigcup_{i=1}^k \{t \mid t \equiv_H t_i\}$ as in line 2.

For Invariant 2, consider some t_i on the cycle. Lemma 27 says that there are no outgoing solid edges from the \equiv_H -equivalence class of t_i , unless that class is a singleton. If this class is a singleton, there are no outgoing **solid edges** from t_i or $t_i \xrightarrow{p} t_{i+1}$ is the only outgoing edge of t_i . In both cases, there are no **solid edges** from Z' to $T \setminus Z'$ in H .

For Invariant 3 we extend a Z -**deadlock scheme** ds_Z to Z' . For every lock $t \in Z' \setminus Z$ let j be the biggest index among $1, \dots, k$ with $t \equiv_H t_j$. If $t = t_j$ then set $ds_{Z'}(p_j)$ to be the edge $t_j \xrightarrow{p_j} t_{j+1}$. Otherwise, take the unique path from t to t_j in the \equiv_H -equivalence class of the two locks; this is possible thanks to Lemma 28. If the path starts with $t \xrightarrow{p} t'$ then set $ds_{Z'}(p)$ to this edge. Then set $ds_{Z'}(p) = \perp$ for all remaining processes p .

We claim that $ds_{Z'}$ is a Z' -**deadlock scheme**. First, there is an outgoing $ds_{Z'}$ edge from every lock in Z' because of the definition. Moreover it is unique.

We need to show that $ds_{Z'}(p)$ is defined for every **solid process** p . This is clear if the two locks, t and t' , of p are in Z . If both locks are not in Z then either $t \equiv_H t'$ or there is a **solo solid edge** between the two, say $t \xrightarrow{p} t'$. In the latter case this is the only edge from t , as H is **trimmed**. As the \equiv_H -equivalence class of t is then a singleton, this must be an edge on the cycle and $ds_{Z'}(p)$ is defined to be this edge. Suppose $t \equiv_H t'$ and $ds_{Z'}(p)$ is not defined. Let j be the biggest index among $1, \dots, k$ such that $t \equiv_H t_j$. If neither $t \xrightarrow{p} t'$ is on the shortest path from t to t_j , nor $t \xleftarrow{p} t'$ is on the shortest path from t' to t_j then there must be a cycle in C . But this is impossible as we assumed that there are no **solid cycles** intersecting $T \setminus Z$ in H (Property H2). The remaining case is when one of the locks of p is in Z and another in $Z' \setminus Z$. There is no **solid edge** leaving Z by Invariant 2. There is no **solid edge** entering Z by the assumption of the lemma. So p is a **solid process** labeling no edge in H which contradicts H3.

The last thing to verify for a Z' -**deadlock scheme** is that there is no **strong cycle** in $ds_{Z'}$. We first check that $ds_{Z'}$ contains $t_k \xrightarrow{p_k} t_1$. This is because t_k is necessary the last from its \equiv_H -equivalence class. A **strong cycle** cannot contain locks from Z as there are no edges entering Z in $ds_{Z'}$. Let $t'_1 \xrightarrow{p'_1} t'_2 \cdots \xrightarrow{p'_l} t'_{l+1} = t'_1$ be a hypothetical **strong cycle** in $Z' \setminus Z$ using transitions in $ds_{Z'}$.

Consider x such that $t'_j \equiv_H t'_j$ for $j \leq x$ but $t'_1 \not\equiv_H t'_{x+1}$. By definition of $ds_{Z'}$ we must have that t'_x is the last lock among t_1, \dots, t_k equivalent to t'_1 , say it is t_y . As each lock only has one outgoing transition in the image of $ds_{Z'}$, and as there is a path from t_y to t_k in that image, t_k must be on that cycle, and thus the **weak edge** $t_k \xrightarrow{p_k} t_1$ as well, contradicting the assumption that this is a **strong cycle**. \square

We conclude with our complete algorithm (if one of our sub-algorithms returns a result, then the entire algorithm stops):

Algorithm 5 Algorithm to check the existence of a [sufficient deadlock scheme](#)

```

1:  $H \leftarrow G_{\mathbb{P}}$ 
2:  $Z \leftarrow \emptyset$ 
3: repeat
4:   Apply Algorithm 1
5: until No more edges are removed from  $H$ 
6: repeat ▷  $H$  is trimmed
7:   Apply Algorithm 2
8: until No more edges are removed from  $H$ 
9: repeat ▷ From now on  $H$  satisfies properties H1, H2 and H3
10:  Apply Algorithm 3 ▷ no edges from  $T \setminus Z$  to  $Z$ 
11:  Apply Algorithm 4
12: until  $Z$  does not increase anymore
13: if there is a process  $p \in Proc \setminus Proc_Z$  that is not  $Z$ -lockable then
14:   return “ $\sigma$  is winning”
15: else
16:   return “ $\sigma$  is not winning”
17: end if

```

Lemma 31. *Algorithm 5 terminates in polynomial time, and fails if and only if there is no [sufficient deadlock scheme](#) for \mathbb{P}^σ .*

Proof. Let (H', Z') be the values at the end of the execution of the algorithm.

Suppose the algorithm fails. If it is before line 13 then using the previous lemmas and Invariant 1 we get that $G_{\mathbb{P}}$ does not have a [sufficient deadlock scheme](#). If the algorithm fails in line 14 then there exists a process p with one of its locks outside of Z and not Z -lockable. Suppose towards a contradiction H has a [sufficient deadlock scheme](#) ds_H . It must have an edge from a lock of p that is not in Z , say from t . By definition, every lock with an incoming edge in ds_H must also have an outgoing edge in ds_H . Following these edges we get a cycle in H . During the last iteration of lines 9-12, Z was not increased, hence by Lemma 29 there are no edges from $T \setminus Z$ to Z . This cycle is therefore outside Z . It has to be a [weak cycle](#) by definition of a [deadlock scheme](#), which is a contradiction because Algorithm 4 did not increase Z in its last application.

If the algorithm succeeds then there is a Z -[deadlock scheme](#), say ds_Z (c.f. Invariant 3). We construct a [sufficient deadlock scheme](#) (Z, ds) for $G_{\mathbb{P}}$ as follows. First, we set $ds(p) = ds_Z(p)$ for all $p \in Proc_Z$. Consider $p \in Proc \setminus Proc_Z$, as the algorithm did not fail in lines 13-14, p is Z -lockable, thus we set $ds(p) = \perp$.

Finally, this algorithm operates in polynomial time as all steps of all loops in the algorithms either decrease H or increase Z . Furthermore, the condition on line 13 is easily verifiable by checking in the [behavior](#) $(\mathbb{P}_p^\sigma)_{p \in Proc}$ of σ whether there exists $\emptyset \longrightarrow B \in \mathbb{P}_p$ such that $B \subseteq Z$. \square

Theorem 6. *The deadlock avoidance control problem for 2LSS is in NP when strategies are required to be locally live.*

Proof. We start by guessing a behavior $\mathbb{P} = (\mathbb{P}_p)_{p \in \text{Proc}}$. Its size is polynomial in the number of processes. We can check in polynomial time that there exists a strategy respecting the patterns in \mathbb{P} thanks to Lemma 14.

If yes, then we compute the lock graph $G_{\mathbb{P}}$ for \mathbb{P} and check if there is a sufficient deadlock scheme for \mathbb{P} in polynomial time thanks to Lemma 31.

By Lemma 20, this algorithm answers yes if and only if the system has a winning locally live strategy. \square

3.3 The exclusive case in PTIME

In this section we study **exclusive 2LSS**. These systems enjoy enough properties to be able to decide the deadlock avoidance control problem with **locally live** strategies in polynomial time.

In an **exclusive** system, if a state has an outgoing acq_t transition, then all its outgoing transitions are labeled with acq_t . So in such a state the process is necessarily blocked until t becomes available.

Behaviors of exclusive systems have some special properties, see Lemma 33. First, whenever a strategy allows a **strong edge** $\{t_1\} \Longrightarrow \{t_2\}$ for a process p , it also allows a reverse **weak edge** $\{t_2\} \dashrightarrow \{t_1\}$. This will imply that the **strong cycle** condition in our **deadlock schemes** can be satisfied automatically, because any **strong cycle** can be replaced by a reverse cycle of **weak edges**. Second, all processes that have some pattern are **fragile**.

The above observations simplify the analysis of the **lock graph**. First, we get a much simplified NP argument (Proposition 34). This allows us to eliminate guessing and obtain a PTIME algorithm (Proposition 38).

Throughout this section we fix an **exclusive 2LSS** \mathcal{S} , and consider only **locally live strategies**. As we have seen in the previous section, whether or not a strategy σ is **winning** is determined by its **behavior** \mathbb{P}^σ . More precisely, σ is **winning** if and only if \mathbb{P}^σ does not admit a **sufficient deadlock scheme**, see Lemma 20. In this section we show that the latter property can be decided in PTIME for **exclusive 2LSS**.

Definition 32. We call a **behavior** \mathbb{P} **exclusive** if

- whenever \mathbb{P}_p contains $\{t_1\} \Longrightarrow \{t_2\}$ then it contains either $\{t_1\} \dashrightarrow \{t_2\}$ or $\{t_2\} \dashrightarrow \{t_1\}$, and
- whenever \mathbb{P}_p contains $\{t_1\} \longrightarrow \{t_2\}$ then p is $\{t_1, t_2\}$ -**lockable** in \mathbb{P} .

Remark 2. Say we have a **strong cycle** $t_1 \xrightarrow{p_1} t_2 \xrightarrow{p_2} \dots \xrightarrow{p_k} t_{k+1} = t_1$ in the **lock graph** $G_{\mathbb{P}}$ of an **exclusive behavior** \mathbb{P} , then all p_i have a **pattern** $t_i \Longrightarrow t_{i+1}$ but not $t_i \dashrightarrow t_{i+1}$. Then by definition of **exclusive behavior**, they all have a pattern $t_{i+1} \dashrightarrow t_i$, hence there is a **weak cycle** $t_1 = t_{k+1} \dashrightarrow \dots \dashrightarrow t_2 \dashrightarrow t_1$.

Lemma 33. *If σ is a **locally live** strategy in an **exclusive 2LSS** and $\mathbb{P}^\sigma = (\mathbb{P}_p)_{p \in \text{Proc}}$ is its **behavior**, then \mathbb{P}^σ is **exclusive**.*

Proof. Consider the first statement. Suppose there is a **strong pattern** $t_1 \Rightarrow t_2 \in \mathbb{P}_p$, then there is a process p and a local σ -run of p of the form

$$u = u_1(a_1, \text{acq}_{t_1})u_2(a_1, \text{rel}_{t_2})u_3(a_3, \text{acq}_{t_2}),$$

with no rel_{t_1} in u_2 or u_3 . Hence, there is a point in the run at which p holds both locks.

If there were always a release between two acquire operations in u_p then p would acquire and then release each lock without ever holding both. In consequence, there must be two acquires in u with no release in-between. As the process is **exclusive**, the state from which the second lock is taken only has outgoing transitions taking it. Thus there is a **weak edge** between the first lock taken and the second one. As $t_1 \xRightarrow{p} t_2$ is strong, the only possibility is that there is a **weak edge** $t_2 \xrightarrow{p} t_1$.

For the second statement, suppose that $t_1 \xrightarrow{p} t_2$ is an edge in $G_{\mathbb{P}}$. Thus there exists a local σ -run u of p acquiring t_1 . The run u is of the form $u_1(a, \text{acq}_{t_i})u_2$ for some $i \in \{1, 2\}$ and u_1 containing only local actions. As \mathcal{S} is **exclusive**, this means that u_1 makes p reach a configuration where all outgoing transitions acquire t_i , and p owns no lock. Since σ is **locally live** this means that p is $\{t_i\}$ -**lockable**, hence also $\{t_1, t_2\}$ -**lockable**. \square

Now consider a decomposition of the lock graph $G_{\mathbb{P}}$ into strongly connected components (SCC for short).

▮ An SCC of $G_{\mathbb{P}}$ is a **direct deadlock** if it contains a **simple cycle**. A **deadlock SCC** is a **direct deadlock SCC** or an SCC from which a **direct deadlock SCC** can be reached.

Figure 6 illustrates these concepts: the left graph has a **direct deadlock SCC** formed by the three locks at the top. The two remaining locks form a **deadlock SCC**, because there is a path towards a **direct deadlock SCC**. Observe that the two locks at the bottom are not a **direct deadlock SCC** because there is only one process between the two locks and thus no **simple cycle** within the SCC.

▮ Let $BT_{\mathbb{P}}$ be the set of all locks appearing in some **deadlock SCC**.

Proposition 34. *Consider an **exclusive behavior** \mathbb{P} . There is a **sufficient deadlock scheme** for \mathbb{P} if and only if all processes are $BT_{\mathbb{P}}$ -**lockable**.*

The proof follows from the lemmas below.

Lemma 35. *If all processes are $BT_{\mathbb{P}}$ -**lockable** then there is a **sufficient deadlock scheme** for \mathbb{P} .*

Proof. We construct a **deadlock scheme** for $G_{\mathbb{P}}$ as follows: For all **direct deadlock SCCs** we select a **simple cycle** inside. By Remark 2 and Lemma 33, this cycle is **weak** or has a reverse **weak cycle**. We select a direction in which the cycle is

weak, and for all t in the cycle we set p_t as the process labeling the edge from t in the cycle.

Then while there is an edge $t \xrightarrow{t'}$ in $G_{\mathbb{P}}$ such that p_t is not yet defined but $p_{t'}$ is, we set $p_t = p$. When this ends we have defined p_t for all locks in $BT_{\mathbb{P}}$. We define ds as $ds(p_t) = t \xrightarrow{p_t} \bar{t}$ for all $t \in BT_{\mathbb{P}}$, and $ds(p) = \perp$ for all other $p \in Proc$. We show that ds is a **sufficient deadlock scheme** for \mathbb{P} .

Clearly, for all $p \in Proc$, the value $ds(p)$ is either \perp or a p -labeled edge of $G_{\mathbb{P}}$. Furthermore, as all processes are $BT_{\mathbb{P}}$ -lockable, in particular the ones mapped to \perp by ds are. It is also clear that all locks of $BT_{\mathbb{P}}$ have a unique outgoing edge. By construction of ds we ensured that we had no strong cycle in it. \square

Lemma 36. *If ds_Z is a **sufficient deadlock scheme** for \mathbb{P} then $Z \subseteq BT_{\mathbb{P}}$.*

Proof. Suppose there is some $t \in Z \setminus BT_{\mathbb{P}}$, then there exists p such that $ds_Z(p) = t \xrightarrow{p} t'$, for some $t' \in Z$. By definition of $BT_{\mathbb{P}}$, there are no edges from $T \setminus BT_{\mathbb{P}}$ to $BT_{\mathbb{P}}$ in $G_{\mathbb{P}}$, hence $t' \in Z \setminus BT_{\mathbb{P}}$. By iterating this process we eventually find a **simple cycle** in $G_{\mathbb{P}}$ outside of $BT_{\mathbb{P}}$, which is impossible, as this cycle should be part of a **direct deadlock SCC**, and thus included in $BT_{\mathbb{P}}$. \square

Lemma 37. *If some process p is not $BT_{\mathbb{P}}$ -lockable then there is no **sufficient deadlock scheme** for \mathbb{P} .*

Proof. Suppose there exists p that is not $BT_{\mathbb{P}}$ -lockable. Towards a contradiction assume that there is some **sufficient deadlock scheme** ds_Z for \mathbb{P} .

As p is not $BT_{\mathbb{P}}$ -lockable, then by Lemma 36 it is not Z -lockable either. Hence, $ds(p)$ is an edge $t_1 \xrightarrow{p} t_2$ in $G_{\mathbb{P}}$, with $t_1, t_2 \in Z$, and thus $t_1, t_2 \in BT_{\mathbb{P}}$.

By Lemma 33, p is $\{t_1, t_2\}$ -lockable, and therefore also $BT_{\mathbb{P}}$ -lockable, yielding a contradiction. \square

This concludes the proof of Proposition 34.

Deciding the existence of a winning strategy for **exclusive** systems.

Until now we have assumed that we were given a **strategy** σ , and we described how to check if it is **winning**, by constructing $BT_{\mathbb{P}}$ and checking that every process is $BT_{\mathbb{P}}$ -lockable, where $\mathbb{P} = \mathbb{P}^{\sigma}$. Now we want to decide if there is any **winning strategy**. We use the insights above, but we cannot simply enumerate all **exclusive behaviors**, as they are exponentially many.

For every process p and every set of edges between two locks of p we can check if there is a **local strategy** inducing only edges within this set, as a consequence of Lemma 14.

◻ We call an edge $t_1 \xrightarrow{p} t_2$ **unavoidable** if all **local strategies** of p induce this edge. Let G_u be the graph whose nodes are locks and whose edges are the **unavoidable** edges.

We will compute a set of locks BT_u in a similar way as $BT_{\mathbb{P}}$ in the previous section except that we will use slightly more general basic SCCs of G_u .

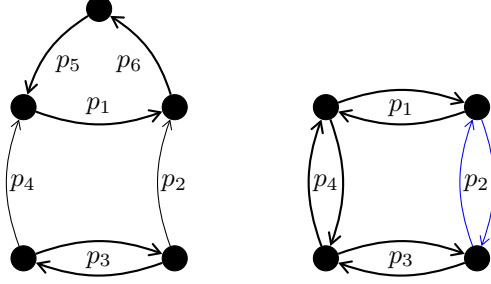


Figure 6: An illustration of *semi-deadlock SCCs*. The blue double edge is not in G_u , but every strategy of the system will induce one of those two edges.

□ A *direct semi-deadlock SCC* of G_u is either a *direct deadlock SCC* or an SCC containing only double edges, and two locks t_1 and t_2 such that for some process p using t_1 and t_2 , every strategy for p induces at least one edge between t_1 and t_2 . Then a *semi-deadlock SCC* of G_u is either a *direct semi-deadlock SCC* or an SCC from which a *direct semi-deadlock SCC* can be reached.

Let BT_u be the set of locks appearing in *semi-deadlock SCCs*.

In the graph on the right part of Figure 6 the black edges are in G_u , the double blue ones are not, but indicate that every local strategy of process p_2 induces one of the two edges in G_P . The four locks do not form a *direct deadlock SCC* of G_u as there is no *simple cycle*. However they do form a *direct semi-deadlock* one, as p_2 will induce an edge no matter its strategy, forming a *simple cycle*.

Proposition 38. *There is a winning strategy if and only if there exists some process p and a local strategy σ_p that prevents p from acquiring a lock from BT_u .*

Proof. One direction is easy: if all strategies make all processes acquire a lock from BT_u then there is no winning strategy. Let σ be a control strategy, \mathbb{P} its behavior and G_P its lock graph. Note that G_u is a subgraph of G_P , hence every SCC in G_P is a superset of an SCC in G_u . Observe that if an SCC in G_P contains a *direct semi-deadlock SCC* of G_u then it is a *direct deadlock SCC*. Indeed, if an SCC in G_u is a *direct semi-deadlock* but not a *direct deadlock* one then σ adds an edge $t_1 \xrightarrow{p} t_2$ to this SCC in G_P . As t_1, t_2 are in that SCC of G_u , there is a simple path from t_2 to t_1 not involving p . Hence, a *direct semi-deadlock SCC* becomes a *direct deadlock SCC*. This implies $BT_u \subseteq BT_P$.

Let $p \in Proc$, as there is a σ -run of p acquiring a lock of BT_u , either p is *BT_u -lockable* (and thus *BT_P -lockable*) or there is an edge labeled by p towards BT_u , meaning that both locks of p are in $BT_u \subseteq BT_P$ and thus that p is *BT_P -lockable* by Lemma 33. As a consequence, all processes are *BT_P -lockable*. We conclude by Proposition 34.

In the other direction we suppose that there exists a process p and a strategy σ_p forbidding p to acquire any lock of BT_u . We construct a strategy σ such that p is not *BT_P -lockable*. This will show that σ is winning by Proposition 34.

Let $FT_u = T \setminus BT_u$ be the set of locks not in BT_u . By definition of BT_u , in G_u no node of FT_u can reach a **direct semi-deadlock SCC**. In particular, there is no **direct semi-deadlock SCC** in G_u restricted to FT_u . We construct a **strategy** σ such that, when restricted to FT_u , the SCCs of $G_{\mathbb{P}}$ and G_u are the same, where $\mathbb{P} = \mathbb{P}^\sigma$.

Let us linearly order the SCC of G_u restricted to FT_u in such a way that if a component C_1 can reach a component C_2 then C_1 is before C_2 in the order.

We use strategy σ_p for p . For every process $q \neq p$ we have one of the two cases: (i) either there is a **local strategy** σ_q inducing only the edges that are already in G_u ; or (ii) every **local strategy** induces some edge that is not in G_u . In the second case there are no q -labeled edges in G_u , and for each of the two possible edges there is a **local strategy** inducing only this edge.

For a process q from the first case we take a **local strategy** σ_q that induces only the edges present in G_u .

For a process q from the second case,

- If both locks of q are in BT_u then take any **local strategy** for q .
- If one of the locks of q is in BT_u and the other in FT_u then choose a **strategy** inducing an edge from the BT_u lock to the FT_u lock.
- If both locks of q are in FT_u then choose a **strategy** inducing an edge from a smaller to a bigger SCC of G_u .

In the last case, both locks cannot be in the same SCC of G_u : As they are in FT_u , this would have to be an SCC with no simple cycles, i.e., a tree of double edges. But then the existence of q implies that this is a **direct semi-deadlock SCC**, which contradicts the fact that those locks are in FT_u .

Consider the graph $G_{\mathbb{P}}$ of the resulting strategy σ . Restricted to FT_u this graph has the same SCCs as G_u . Moreover, there are no extra edges in $G_{\mathbb{P}}$ added to any SCC included in FT_u , and there are no edges from FT_u to BT_u . As a result, we have $BT_u = BT_{\mathbb{P}}$.

As p acquires no lock from BT_u , it is not BT_u -lockable and thus not $BT_{\mathbb{P}}$ -lockable either. □

Theorem 8. *The **deadlock avoidance control problem** for **exclusive 2LSS** is in PTIME, when **strategies** are required to be **locally live**.*

Proof. We can compute BT_u in polynomial time and then check the condition from Proposition 38. □

4 Nested locks

We switch to another decidable case, the one of **nested-locking LSS**, in which the system has to ensure that locks are acquired and released in a stack-like manner. So a process can release only the last lock it has acquired.

Definition 39. A *stair decomposition* of a local run u is

$$u = u_1 \text{acq}_{t_1} u_2 \text{acq}_{t_2} \cdots u_k \text{acq}_{t_k} u_{k+1}$$

where u_1, \dots, u_{k+1} are *neutral* runs, and each u_i does not use any locks from $\{t_1, \dots, t_{i-1}\}$. We omit the actions associated with each operation as they are irrelevant here.

Lemma 40. Every *nested-locking* local run u has a unique *stair decomposition*.

Proof. We set $u = u_1 \text{acq}_{t_1} u_2 \text{acq}_{t_2} \cdots u_k \text{acq}_{t_k} u_{k+1}$ such that $\{t_1, \dots, t_k\}$ is the set of locks held by the process, call it p , at the end of the run, and the distinguished acq_{t_i} are the last acquisitions of these locks in u . Consequently, there is no operation on t_i in u_{i+1}, \dots, u_{k+1} .

Observe that u_{k+1} must be *neutral* because the process owns $\{t_1, \dots, t_k\}$ at the end of u . If some u_i , $i \leq k$, were not *neutral*, then there would exist some $t \notin \{t_1, \dots, t_i\}$ such that p holds t after $u_1 \text{acq}_{t_1} \cdots u_i \text{acq}_{t_i}$. Then p has to release t at some point later in the run: if $t \notin \{t_1, \dots, t_k\}$ then p does not hold it at the end; otherwise $t \in \{t_{i+1}, \dots, t_k\}$, and t is taken again later in the run. Both cases contradict the *nested-locking* assumption, because t would be released before t_i , which has been acquired after t . \square

We now define patterns of *risky* local runs that will serve as witnesses of reachable *deadlocks*, in a similar manner as in Definition 12.

Definition 41. Consider a local *risky* σ -run u of process p , and its *stair decomposition* $u = u_1 \text{acq}_{t_1} u_2 \text{acq}_{t_2} \cdots u_k \text{acq}_{t_k} u_{k+1}$. We associate with u a *stair pattern* $(T_{\text{owns}}, T_{\text{blocks}}, \preceq)$, where $T_{\text{owns}} = \{t_1, \dots, t_k\}$, T_{blocks} is the set of locks taken by outgoing transitions in the state reached by u , and \preceq is the smallest partial order on T_p satisfying the following: for all i , for all $t \in T_p$, if the last operation on t in u comes after the last acq_{t_i} then $t_i \preceq t$. A *behavior* of σ is a family of sets of *stair patterns* $(\mathbb{P}_p^\sigma)_{p \in \text{Proc}}$, where \mathbb{P}_p^σ is the set of *stair patterns* of local *risky* σ -runs of p .

Lemma 42. A *control strategy* σ with *behavior* $(\mathbb{P}_p^\sigma)_{p \in \text{Proc}}$ is *not winning* if and only if for every $p \in \text{Proc}$ there is a *stair pattern* $(T_{\text{owns}}^p, T_{\text{blocks}}^p, \preceq^p) \in \mathbb{P}_p^\sigma$ such that:

- $\bigcup_{p \in \text{Proc}} T_{\text{blocks}}^p \subseteq \bigcup_{p \in \text{Proc}} T_{\text{owns}}^p$,
- the sets T_{owns}^p are pairwise disjoint,
- there exists a total order \preceq , on the set of all locks T , compatible with all \preceq^p .

Proof. Suppose σ is not *winning*, and let w be a run leading to a *deadlock*. For all p let T_{owns}^p be the set of locks owned by p after w . Take u^p the local run of p in w . Since w leads to a *deadlock* every u^p is *risky*. For every p , consider the *stair pattern* $(T_{\text{owns}}^p, T_{\text{blocks}}^p, \preceq^p)$ of u^p . This way we ensure it is a *pattern* from \mathbb{P}_p^σ .

We need to show that these patterns satisfy the requirements of the lemma. Since the configuration reached after w is a **deadlock**, every process waits for locks that are already taken so $T_{blocks}^p \subseteq \bigcup_{q \in Proc} T_{owns}^q$, for every process p , proving the first condition.

We have that T_{owns}^p is the set of locks that p has at the end of the run w . So the sets T_{owns}^p are pairwise disjoint.

For the last requirement of the lemma take an order \preceq on T satisfying: $t \preceq t'$ if the last operation on t appears before the last operation on t' in w .

Let $p \in Proc$, let $u^p = u_1^p \text{acq}_{t_1^p} u_2^p \text{acq}_{t_2^p} \cdots u_k^p \text{acq}_{t_k^p} u_{k+1}^p$ be the **stair decomposition** of u^p . As p never releases t_i^p , the distinguished $\text{acq}_{t_i^p}$, is the last operation on t_i^p in the global run. Consequently, for all t we have $t_i^p \preceq t$ whenever t is used in $u_{i+1}^p \text{acq}_{t_{i+1}^p} \cdots u_k^p \text{acq}_{t_k^p} u_{k+1}^p$. As a result, \preceq is compatible with all \preceq^p .

For the converse implication, suppose that there are **patterns** satisfying all the conditions of the lemma. We need to construct a run w ending in a **deadlock**. For every process p we have a **stair pattern** $(T_{owns}^p, T_{blocks}^p, \preceq^p)$ coming from a local σ -run u^p of p , with $u^p = u_1^p \text{acq}_{t_1^p} u_2^p \text{acq}_{t_2^p} \cdots u_k^p \text{acq}_{t_k^p} u_{k+1}^p$ as **stair decomposition**. There is also a linear order \preceq compatible with all \preceq_p . Let \prec be its strict part. Let t_1, \dots, t_k be the sequence of locks from $\bigcup_p T_{owns}^p$ listed according to \prec . Let $\{p_1, \dots, p_n\} = Proc$. We claim that we can get a suitable global run w as $u_1^{p_1} \dots u_1^{p_n} w'$ where w' is obtained from $t_1 \dots t_k$ by substituting each t_i^p by $\text{acq}_{t_i^p} u_{i+1}^p$. Observe that every t_j from the sequence $t_1 \dots t_k$ corresponds to exactly one t_i^p , as the sets $T_{owns}^{p_1}, \dots, T_{owns}^{p_n}$ are disjoint.

All u_i^p are **neutral**, hence after executing $u_1^{p_1} \dots u_1^{p_n}$ all locks are free. Let $t_i^p \in T_p$, suppose furthermore that all $\text{acq}_{t_j^q} u_{j+1}^q$ with $t_j^q \prec t_i^p$ have been executed after $u_1^{p_1} \dots u_1^{p_n}$. Then the set of taken locks is $\{t_j^q \mid t_j^q \prec t_i^p\}$. As \preceq is compatible with all \preceq^p , all locks t used in $\text{acq}_{t_i^p} u_{i+1}^p$ are such that $t_i^p \preceq t$. Moreover, since all t_j^q that were taken before are such that $t_j^q \prec t_i^p$, the run $\text{acq}_{t_i^p} u_{i+1}^p$ uses only locks that are free and can therefore be executed.

As a result, w can be executed. It leads to a **deadlock** as $T_{blocks}^p \subseteq \bigcup_q T_{owns}^q$. \square

Lemma 43. *Given a **nested-locking LSS** \mathcal{S} , a process $p \in Proc$ and a set of **patterns** \mathbb{P}_p , we can check in polynomial time in $|\mathcal{A}_p|$ and $2^{|T|}$ whether there exists a **strategy** σ with $\mathbb{P}_p^\sigma \subseteq \mathbb{P}_p$.*

Proof. Fix a process p . We extend the states of p to keep track of the set of locks held by p as well as the order \prec induced by the **stair pattern** of the run seen so far (as in Definition 41). This increases the number of states by the factor $|T|! \cdot 2^{|T|}$.

As the set of locks owned by p is now a function of the current state, this also allows us to eliminate all non-realizable transitions which acquire a lock that p owns or release one it does not have.

Consider a state s where all outgoing transitions have a lock acquisition operation. Thanks to the previous paragraph, s determines the set of locks T_{owns}^s and an order \prec_s such that every local run ending in s has a pattern

$(T_{own,s}^s, T, \prec_s)$, where T depends on the choices a strategy for p makes in s . We mark s bad if none of these possible patterns is in \mathbb{P}_p .

We iteratively delete all bad states and all their ingoing transitions, as we need to ensure that we never reach them. If we delete an **uncontrollable** transition then we mark its source state as bad because reaching that state would make the environment able to reach a bad state. If this process marks the initial state bad then there is no local strategy with patterns included in \mathbb{P}_p . Otherwise, we look for new bad states as in the previous paragraph. Indeed, a state may satisfy the conditions of the previous paragraph after removing some of its outgoing transitions, for example a transition not accessing locks. If some new state is marked bad then we repeat the whole procedure.

When this double loop stabilizes and if the initial state is not marked bad, then the remaining transitions form a local strategy with all patterns in \mathbb{P}_p . \square

Proposition 44. *The **deadlock avoidance control problem** is decidable for **nest-locked-locking lock-sharing systems** in non-deterministic exponential time.*

Proof. First of all one can check that an **LSS** is **nested-locking** by considering each process p , expanding its set of states to include the information of which locks are held by p and in which order they were taken in the states. This causes an exponential blow-up of the size of the system. It is then easy to check that all release operations free the lock that was acquired last among the ones that are held by the process.

The decision procedure for the existence of a winning strategy guesses a **behavior** \mathbb{P}_p for each process p . The size of the guess is at most $2^{2|T|} \cdot |T|! \leq 2^{O(|T| \log(|T|))}$. Then it checks if there exist **local strategies** yielding subsets of those **behaviors**. This takes exponential time by Lemma 43. If the result is negative then the procedure rejects. Otherwise, it checks if some condition from Lemma 42 does not hold. If it finds one then it accepts, otherwise it rejects.

Clearly, if there is a **winning strategy** then the procedure can accept by guessing the family of **behaviors** corresponding to this **strategy**. For these **behaviors** the check from Lemma 43 does not fail, and one of the conditions of Lemma 42 must be violated.

Conversely, if the decision procedure concludes that there exists a **winning strategy**, then let $(\mathbb{P}_p)_{p \in Proc}$ be the guessed family of **behaviors**. We know that there exists a **strategy** σ with **behaviors** $(\mathbb{P}'_p)_{p \in Proc}$ such that $\mathbb{P}'_p \subseteq \mathbb{P}_p$ for all $p \in Proc$. Furthermore, as there are no **patterns** in $(\mathbb{P}_p)_{p \in Proc}$ satisfying the requirements of Lemma 42, there cannot be any in the \mathbb{P}'_p either. Hence σ is a **winning strategy**. \square

Theorem 10. *The **deadlock avoidance control problem** for **LSS** is NEXPTIME-complete.*

Proof. The upper bound is given by Proposition 44. For the lower bound, we reduce from the domino tiling problem over an exponential grid. In this problem, we are given an alphabet Σ with a special letter b , an integer n (in unary) and a set D of dominoes, each domino d being a 4-tuple $(up_d, down_d, right_d, left_d)$

of letters of Σ . The question is whether there exists a mapping $t : \{0, \dots, 2^n - 1\}^2 \rightarrow D$ representing a valid tiling of the grid, i.e. such that for all $x, y, x', y' \in \{0, \dots, 2^n - 1\}$:

- if $x' = x$ and $y' = y + 1$ then $up_{t(x,y)} = down_{t(x',y')}$
- if $x' = x + 1$ and $y' = y$ then $right_{t(x,y)} = left_{t(x',y')}$
- if $x = 0$ then $left_{t(x,y)} = b$
- if $x = 2^n - 1$ then $right_{t(x,y)} = b$
- if $y = 0$ then $down_{t(x,y)} = b$
- if $y = 2^n - 1$ then $up_{t(x,y)} = b$

The above problem is well-known to be NEXPTIME-complete.

Let n, Σ, D, b be an instance of the tiling problem. We construct a *LSS* as follows: We have three processes p, \bar{p} and q . Process p uses locks from $\{0_i^x, 1_i^x, 0_i^y, 1_i^y \mid 1 \leq i \leq n\}$, together with a lock t_d for each domino $d \in D$, and an extra lock called simply ℓ . Process \bar{p} will use similar locks but with a bar: $\overline{0_i^x}, \overline{1_i^x}, \overline{0_i^y}, \overline{1_i^y}, \overline{t_d}, \overline{\ell}$. Process q will use all the locks of p and \bar{p} .

Let us describe process q represented in Figure 8. In the initial state the environment can choose between several actions: *equality*, *vertical*, *horizontal*, b_{left} , b_{right} , b_{up} and b_{down} . Each of these actions leads to a different transition system, but the principle behind all the systems is the same. In the first phase, for each $1 \leq i \leq n$, the environment can choose to take either lock 0_i^x or 1_i^x , and then take either $\overline{0_i^x}$ or $\overline{1_i^x}$. In the second phase the same happens for y locks. After these two phases the environment has chosen two pairs of n -bit numbers, call them $\#x, \#y$ and $\#\bar{x}, \#\bar{y}$. Where the three systems differ is how the choice of \bar{x} 's and \bar{y} 's is limited in these two phases. This depends on the first action done by the environment:

- If it is *equality* then $\#x = \#\bar{x}$ and $\#y = \#\bar{y}$.
- If it is *vertical*, then $\#x = \#\bar{x}$ and $\#y + 1 = \#\bar{y}$.
- If it is *horizontal*, then $\#x + 1 = \#\bar{x}$ and $\#y = \#\bar{y}$.
- If it is b_{left} (resp. b_{right}) then $\#x = 0$ (resp. $\#x = 2^n - 1$).
- If it is b_{down} (resp. b_{up}) then $\#y = 0$ (resp. $\#y = 2^n - 1$).

All these constraints are easily implemented. For example, after *equality* the environment must take the same bits for \bar{x} as for x (similarly for y).

In the third phase, process q has to take and then immediately release locks ℓ and $\bar{\ell}$, before it reaches a state called *dominoes*.

Every state in the three phases before *dominoes* has a loop on it, meaning that q cannot *deadlock* while being in one of these states.

In state *dominoes*, the system chooses to take two dominoes d and \bar{d} such that:

- If the environment chose *equality* then $d = \bar{d}$.
- If it chose *vertical* then $up_d = down_{\bar{d}}$.
- If it chose *horizontal* then $right_d = left_{\bar{d}}$.
- If it chose b_{left} (resp. $b_{right}, b_{up}, b_{down}$) then $left_d = b$ (resp. $right_d, up_d, down_d$).

Each choice leads to a different state $s_{d,\bar{d}}$. From there transitions force the system to take every lock $t_{d'}$ and $\bar{t}_{d'}$, except for t_d and $t_{\bar{d}}$, in order to reach a state called *win* with a local loop on it and no other outgoing transitions.

We now describe process p represented in Figure 8. It starts by taking the lock ℓ , which it never releases. Then the environment chooses to take one of 0_i^x and 1_i^x and one of 0_i^y and 1_i^y for all $1 \leq i \leq n$. Finally, the system chooses a domino d and takes the lock t_d before reaching a state with no outgoing transitions.

Process \bar{p} behaves identically, but uses locks with a bar.

We need to show that if there is a tiling $t : \{0, \dots, 2^n - 1\}^2 \rightarrow D$ then there is a *winning strategy*. The *strategy* for q is to respond with the correct tiles: if the environment chooses $\#x, \#y, \#\bar{x}, \#\bar{y}$ the strategy chooses locks corresponding to d_1 and \bar{d}_2 with $d_1 = t(\#x, \#y)$ and $d_2 = t(\#\bar{x}, \#\bar{y})$. The strategy of p does the same but uses inverse encoding of numbers: considers 0 as 1, and 1 as 0. Similarly for \bar{p} .

Assume for contradiction that the *strategy* is not *winning*, so we have a run leading to a *deadlock*. First, observe that the environment must have process q go through state *dominoes* before p and \bar{p} start running, because all states before *dominoes* have a self-loop, so q cannot block there. If either p or \bar{p} starts before q has reached *dominoes*, then q can never reach it, as one of the locks $\ell, \bar{\ell}$ will never be available again.

If q reached state *dominoes* then process p has no choice but to take ℓ , and then the remaining locks among x, y . Similarly for \bar{p} . At this stage the *strategy* σ is defined so that the three processes will never take the same lock. So q cannot be blocked before reaching state *win*. Thus deadlock is impossible.

For the other direction, suppose there is a *winning strategy* σ for the system. Observe that the *strategy* σ_p for process p decides which domino to take after the environment has decided which x and y locks to take. So σ_p defines a function $t : \{0, \dots, 2^n - 1\}^2 \rightarrow D$. Similarly $\sigma_{\bar{p}}$ defines \bar{t} .

We first show that $t(i, j) = \bar{t}(i, j)$ for all $i, j \in \{0, \dots, 2^n - 1\}$. If not then consider for example the run where the environment chooses *equality* and then x, \bar{x} to be the representations of i , and y, \bar{y} to be representations of j . Suppose we have a run where process q reaches state *dominoes*, and assume that q 's strategy tells to go to state (d, \bar{d}) . Next the environment makes processes p and \bar{p} reach the states where they chose their dominoes, $t(i, j)$ and $\bar{t}(i, j)$ respectively. The two processes p and \bar{p} then reach a *deadlock* state. Since we assumed that $t(i, j) \neq \bar{t}(i, j)$, process q cannot reach state *win* from any state $s_{d,\bar{d}}$. Hence we have a *deadlock* run, a contradiction.

Once we know that the strategies σ_p and $\sigma_{\bar{p}}$ define the same tiling function it is easy to see that in order to be **winning** when the environment chooses one of the actions *vertical*, *horizontal* or b_{left} , b_{right} , b_{down} , b_{up} , the tiling function must be correct.

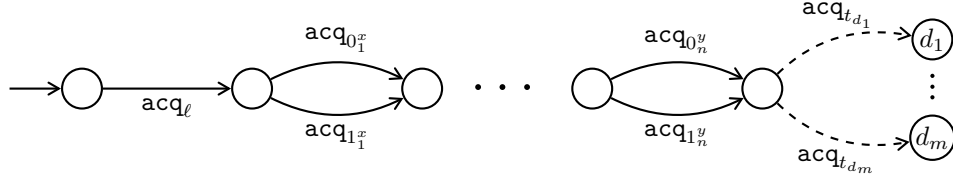


Figure 7: Transition system for process p for the proof of Theorem 10 (with $D = \{d_1, \dots, d_m\}$). Dashed arrows are controlled by the system.

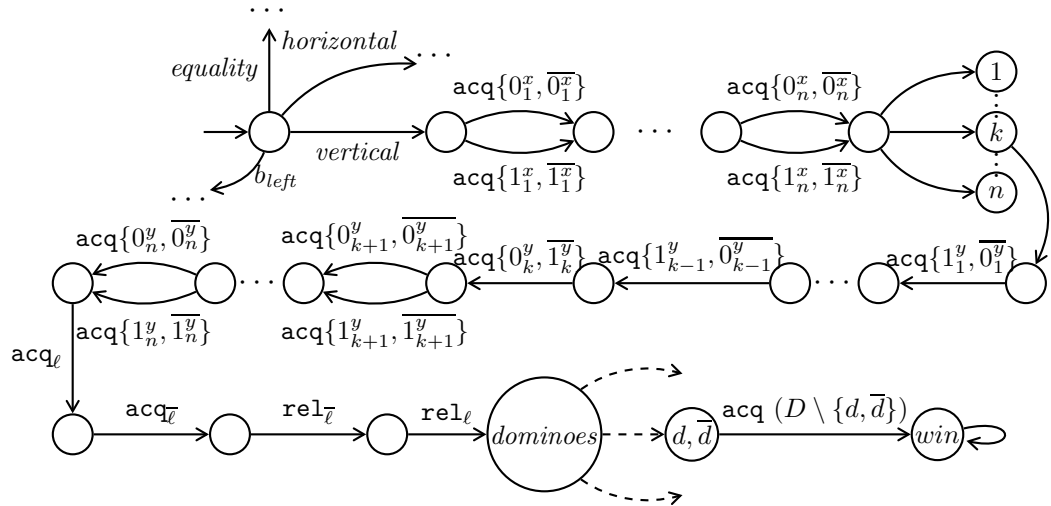


Figure 8: Transition system for process q in the proof of Theorem 10. Dashed arrows are controllable, every state before *dominoes* has a self-loop (not drawn) and $\text{acq } S$ means a sequence of forced transitions with the operations $\text{acq}_{t_{\ell}}$ for each $t \in S$ (in some order). For simplicity only the *vertical* case is shown.

□

5 Undecidability in the general case

We have seen that the control problem is decidable when each process can use at most 2 locks. Here we show that it becomes undecidable with 4 locks per

process. The case of 3 locks per process remains open.

The general principle of the proof is as follows. Using 3 locks, it is not hard to show that two processes are able to communicate. We will use altogether 4 locks in order to implement a reduction from the infinitary variant of the PCP problem.

We will construct three processes, C , P and \overline{P} . Processes P and \overline{P} are in charge of producing each an infinite sequence of bits, by acquiring and releasing their locks in a specific order. The third process C (the controller) has to produce two sequences of bits, one for each of P and \overline{P} . The system will deadlock if the first sequence of C does not match the one of P , or the second one the one of \overline{P} . At the start the environment makes a choice: C will either have to produce two identical sequences of bits, or two identical sequences of indices of pairs of the PCP instance. As the local strategies for P and \overline{P} are unaware of the environment's choice, the only way to win is to produce matching sequences of bits that describe an infinite PCP solution.

One additional difficulty is the initialization: as processes do not hold any lock at the beginning, the environment could cheat by having one process run alone a long time before the others start, in order to prevent the synchronization. That is why we make processes synchronize over a specific sequence of bits that does not appear later at the beginning, and add self-loops so that a process cannot be deadlocked while in that phase. This forces the environment to make them execute that phase together, after which they are in a configuration where they all hold locks and cannot be easily desynchronized.

Theorem 45. *The **deadlock avoidance control problem** for **LSS** with 3 processes and 4 locks is undecidable.*

Let $(u_i, v_i)_{i=1}^m$ be the PCP instance with $u_i, v_i \in \{0, 1\}^*$. An infinite solution is an infinite sequence of indices $i_1 i_2, \dots$ such that $u_{i_1} u_{i_2} \dots = v_{i_1} v_{i_2} \dots$. W.l.o.g. we assume that there is either a unique solution, or none.

The **LSS** we construct has three processes P, \overline{P}, C , using locks from the set

$$\{s_0, s_1, p, \overline{p}\}.$$

Process P uses only locks from $\{s_0, s_1, p\}$, process \overline{P} from $\{s_0, s_1, \overline{p}\}$, and C uses all 4 locks.

Each of the processes P, \overline{P} is supposed to synchronize with process C over an infinite binary sequence representing a PCP solution. Processes P and C synchronize over a sequence $u_{i_1} u_{i_2} \dots$, whereas \overline{P} and C synchronize over a sequence $v_{j_1} v_{j_2} \dots$.

A local choice of the environment tells C at the beginning whether she should check index equality $i_1 i_2 \dots = j_1 j_2 \dots$ or word equality $u_{i_1} u_{i_2} \dots = v_{j_1} v_{j_2} \dots$.

We first give a high-level description of how the processes P, \overline{P}, C synchronize over a single bit. Define for $b \in \{0, 1\}$, sequences of operations for P and \overline{P} , respectively:

$$\begin{aligned} B(b) &= \text{acq}_{s_b} \text{rel}_p \text{acq}_{s_{1-b}} \text{rel}_{s_b} \text{acq}_p \text{rel}_{s_{1-b}} \\ \overline{B}(b) &= \text{acq}_{s_b} \text{rel}_{\overline{p}} \text{acq}_{s_{1-b}} \text{rel}_{s_b} \text{acq}_{\overline{p}} \text{rel}_{s_{1-b}} \end{aligned}$$

and the matching sequences on C 's side:

$$\begin{aligned} C(b) &= \text{rel}_{s_b} \text{acq}_p \text{rel}_{s_{1-b}} \text{acq}_{s_b} \text{rel}_p \text{acq}_{s_{1-b}} \\ \overline{C}(b) &= \text{rel}_{s_b} \text{acq}_{\overline{p}} \text{rel}_{s_{1-b}} \text{acq}_{s_b} \text{rel}_{\overline{p}} \text{acq}_{s_{1-b}} \end{aligned}$$

At the beginning of a synchronization round between e.g. C and P , process C owns locks s_0 and s_1 , and P owns p . Having C, P execute $C(b)$ and $B(b)$, respectively, in lock-step manner, amounts to have C and P synchronize over bit b . However, this synchronization can be “spoiled” by the other process, here \overline{P} , who may interfere with the executions of C and P . This makes the definition of C 's behavior more complicated (see $C(P, b)$ in Figure 14, that contains $C(b)$ as the main branch).

We provide now a high-level description of the structure of the *LSS*. We start with processes P and \overline{P} , which are simpler than C because it is process C who controls the verification and who initiates the synchronizations. Process P is built as follows: after an initial phase it enters the choice state where it has two *controllable* actions, each taking him to a sequence of *uncontrollable* actions realizing one of the sequences $B(b)$, and then returning to the choice state (see Figure 15). Process \overline{P} is built similarly. As we will see, a strategy for a processes simply amount to making choices in the choice state, hence a strategy is a binary sequence. The correctness of this sequence is checked via synchronization with C .

The behaviour of C is as follows: after the initial phase, the environment tells C whether she should check word equality or index equality (see Figure 9). To check word equality, C repeats the following procedure: she chooses a bit b through a *controllable* action, then checks that b is the current bit of P , and then the same for \overline{P} (cf. Figure 12). For index equality the procedure is similar: C chooses an index i through a *controllable* action. Then she checks that the next bits of P correspond to u_i , and that the next bits of \overline{P} correspond to v_i (see Figure 13). Figure 9 shows the overall structure of C .

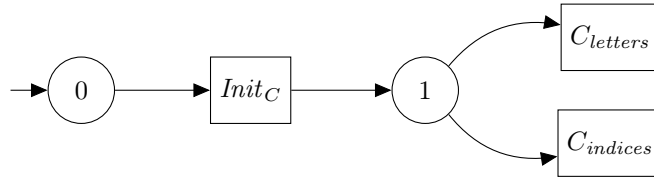


Figure 9: After the initialization phase of C , the environment chooses to check either that the two sequences of P, \overline{P} are equal or that they are produced by the same index sequence. The parts $C_{letters}$ and $C_{indices}$ are described in Figures 12, 13.

The initialization $Init_C$ of process C is depicted in Figure 10: process C first acquires s_0, s_1 , then synchronizes twice with P over bit 0 and twice with \overline{P} over bit 0. This is done using components $C^\top(P, 0)$ and $C^\top(\overline{P}, b)$ as shown

in Figure 11. These components are essentially the sequence $C(b)$ from page 41 with added self-loops on every state where C awaits a lock. As we can see from Figures 10 and 11, process C cannot block in $Init_C$ because there is a self-loop from every state with outgoing acquire transition.

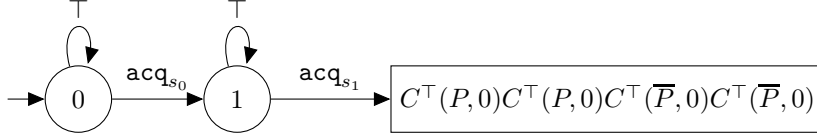


Figure 10: $Init_C$

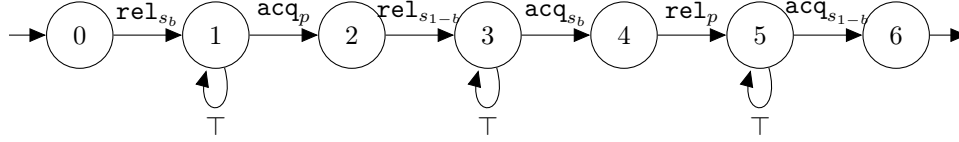


Figure 11: Component $C^T(P, b)$ of process C , $b \in \{0, 1\}$

To check word equality in $C_{letters}$, process C selects transitions matching the sequence of bits of both P and \bar{P} . Because of the lock ownership initialization, the three processes synchronize over the sequence $001b_1b_2\dots$ instead of the PCP solution $b_1b_2\dots$ (cf. Figure 12).

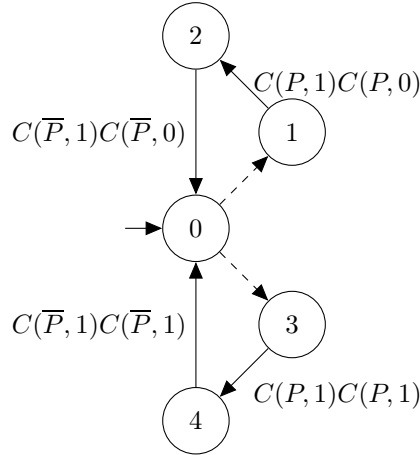


Figure 12: C either synchronizes with P and \bar{P} over bit 0 (upper branch) or over bit 1 (lower branch). The two outgoing transitions of state 0 are **controllable**.

To check index equality in $C_{indices}$, process C selects a sequence of indices

and checks for each index i with P and \overline{P} that their bit sequences match the sequences u_i and v_i , respectively, see Figure 13.

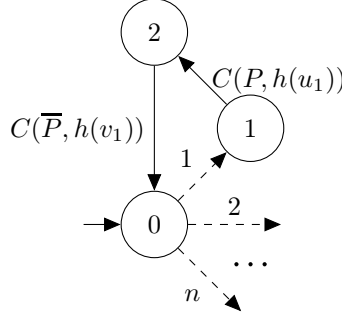


Figure 13: C either synchronizes with P and \overline{P} over some index i , with $h(0) = 10$ and $h(1) = 11$. The outgoing transitions of state 0 are **controllable**.

The component $C(P, b_1 \dots b_k)$ in Figure 13 stands for the sequential composition of $C(P, b_1), \dots, C(P, b_k)$, see $C(P, b)$ defined in Figure 14 (similarly for $C(\overline{P}, b_1 \dots b_k)$).

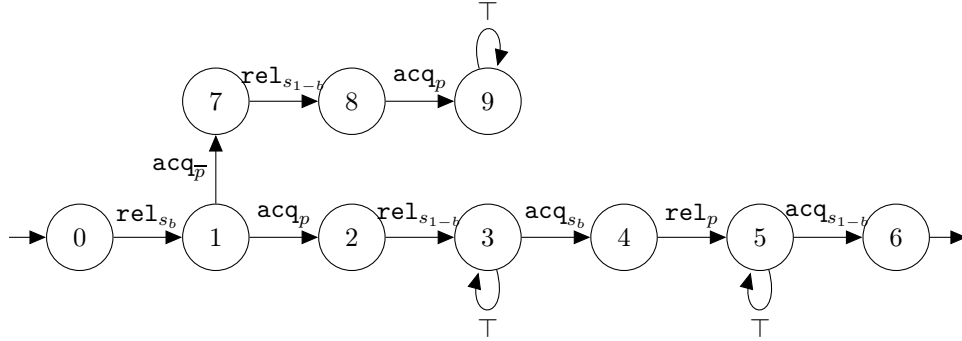


Figure 14: Component $C(P, b)$ of process C , with $b \in \{0, 1\}$

It remains to describe processes P (and \overline{P}), see Figure 15. We describe only P , since \overline{P} is symmetric.

The initialization sequences of P and \overline{P} are:

$$Init_P = \text{acq}_p B^\top(0) B^\top(0), \quad Init_{\overline{P}} = \text{acq}_{\overline{p}} \overline{B}^\top(0) \overline{B}^\top(0)$$

They use the (deadlock-free) component $B^\top(b)$ from Figure 16. Note that $B^\top(b)$ has the block $B(b)$ from page 41 as main branch, extended by self-loops on the transitions that need to acquire a lock, so that no deadlock will be possible in this part. In $Init_P$ process P first acquires lock p , then tries to synchronize with

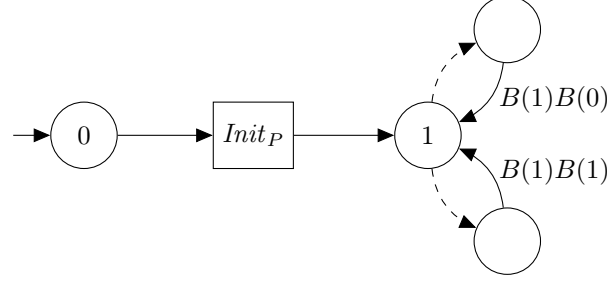


Figure 15: Process P , with $Init_P = \text{acq}_p B^\top(0) B^\top(0)$. The outgoing transitions of state 1 are **controllable**.

C twice over bit 0. After $Init_P$, process P repeatedly chooses between executing $B(1)B(0)$ or $B(1)B(1)$, see Figure 15. Process \overline{P} is defined analogously, with lock \overline{p} instead of p .

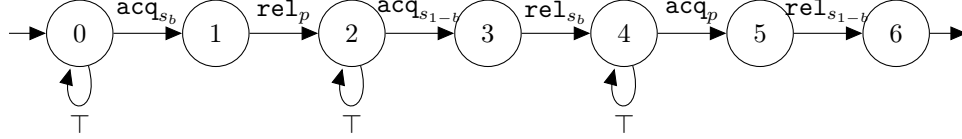


Figure 16: Component $B^\top(b)$ of process P , $b \in \{0, 1\}$

Remark 3. Note that in our *LSS* no **controllable** action uses locks. Hence it is always in the system's interest to allow at most one outgoing transition when it controls several, as this will not cause a deadlock and simply amounts to reducing the set of runs the environment can choose. Hence, if a strategy is **winning** then it remains so if we restrict it to allow only one **controllable** action. In particular, a **strategy** for P and \overline{P} , respectively, corresponds to the choice of a binary sequence (finite or infinite).

Remark 4. Our *LSS* is such that at any time, each of the processes that has already started its execution (by leaving the initial state) owns at least one lock. In particular, if all three processes have started then at most one lock can be free.

The next lemma describes the main properties of one round of synchronization between C and P , over bit b .

Recall that by Remark 3 we can assume that P 's strategy never allows to do both $B(0)$ and $B(1)$, and similarly for \overline{P} .

Lemma 46. Assume that C is in state 0 of $C(P, b)$ (c.f. Figure 14), P is in its choice state ready to execute $B(0)$ or $B(1)$, and \overline{P} is in its choice state ready to execute $\overline{B}(0)$ or $\overline{B}(1)$. Moreover, C owns s_0, s_1 , P owns p and \overline{P} owns \overline{p} .

If P 's strategy does not allow to do $B(b)$ then some execution from the above configuration reaches a deadlock. Otherwise, in all executions process C either stays in some \top -labeled loop, or reaches state 6. In the latter case, when C has reached state 6: process P has finished $B(b)$, process \bar{P} did not move, and the lock ownership is the same as before the execution.

Proof. Consider first the case where P 's strategy does not allow to do $B(b)$. If \bar{P} is not allowed to do $\bar{B}(b)$, then the system is **deadlocked** after C releases s_b . Otherwise we claim that the environment wins by following the branch 1, 7, 8, with \bar{P} doing the matching operations **acq** and **rel**. This is because the system is **deadlocked** after \bar{P} releases s_b : \bar{P} needs \bar{p} and P does not require s_b , which is the only free lock. Moreover, C 's transition $8 \xrightarrow{\text{acq}_p} 9$ is not possible, since P did not release p .

We are left with the case where P 's strategy allows to do $B(b)$ (and not $B(1-b)$). After C releases s_b the following can happen:

1. If \bar{P} takes s_b then C 's action acq_p in state 1 cannot happen. The communication along the path 1, 7, 8 is between C and \bar{P} . When \bar{P} releases s_b and C is in state 8, then the environment has no choice but to let P acquire s_b and C acquire p to reach 9, thus the system does not deadlock.
2. Otherwise P takes s_b and C can proceed until state 3.

We assume now that C is in state 3. There are two cases:

1. Suppose that P took s_{1-b} after C released it. Next P releases s_b , hence C can take s_b , release p , and proceed to state 5. Or \bar{P} takes s_b , in which case C 's action acq_{s_b} in state 3 can never happen: \bar{P} waits for s_{1-b} (owned by P) and P waits for p (owned by C). In the latter case C will loop forever in state 3.
2. Suppose that \bar{P} took s_{1-b} after C released it. Then C 's action acq_{s_b} in state 3 can never happen because P still owns s_b . So C will again loop forever in state 3.

Assume finally that C is in state 5. Since p was just released by C , process P can take it and release s_{1-b} . There are three possible cases:

1. If \bar{P} takes s_{1-b} then C 's action $\text{acq}_{s_{1-b}}$ from state 5 will never happen: \bar{P} waits for s_b (owned by C) and P waits either for s_{1-b} or s_b , to start the next block $B(b')$. So C will loop forever in state 5.
2. The situation is similar if P takes s_{1-b} again (as part of his next block): P waits for s_b (owned by C) and \bar{P} waits either for s_{1-b} or s_b , for his next block. Again, C will loop forever in state 5.
3. If C takes s_{1-b} then she proceeds to state 6, and the lock ownership is the initial one.

□

Remark 5. Note that each synchronization round, as described in Lemma 46, is initiated by C , since both P and \bar{P} need one of the locks s_b or s_{1-b} in order to start $B(b)$, $\bar{B}(b)$ respectively. In particular, when P and \bar{P} are ready to execute the next block ($B(b)$ or $\bar{B}(b)$), they need C to release a lock in order to start their execution.

Observe also that Lemma 46 requires the assumption that P owns p , \bar{P} owns \bar{p} , and C owns s_0, s_1 when a synchronization round starts. The encoding h and the initial bits 00 are used in order to enforce such an initial configuration from the starting configuration, where all locks are free.

We show now that it is possible to enforce the initial lock ownership of Lemma 46 from the initial configuration where all locks are free.

□ We denote by *Start* the configuration where C is in state 1 of Figure 9, holding the locks s_0 and s_1 , and P and \bar{P} are in state 1 of Figure 15, holding p and \bar{p} , respectively.

□ We will use a particular vocabulary in order to shorten the subsequent proof: we say that C and P execute $C^\top(P, b)$ and $B^\top(b)$ *synchronously* if they go from state 0 to 6 in Figures 11 and 16 respectively while alternating their operations: C releases s_b and P acquires it, then P releases p and C acquires it, etc. This wording is used similarly for $C^\top(\bar{P}, b)$ and $\bar{B}^\top(b)$.

We say that the *initialization phase* is executed *entirely synchronously* if the two copies of $B^\top(0)$ and of $C^\top(P, 0)$ are executed synchronously, and the two copies of $\bar{B}^\top(0)$ and of $C^\top(\bar{P}, 0)$ are executed synchronously as well.

Lemma 47. Assume that processes C, P, \bar{P} start their execution and the *initialization phase* is not executed *entirely synchronously*. Then one of the processes loops forever in its initialization part. Otherwise, the configuration *Start* is reached after executing the *initialization phase*.

Proof. We distinguish three cases.

Case 1: C starts last.

Let us assume that C does not loop forever in $Init_C$. So process C will ultimately execute the first two steps of $Init_C$, acquiring s_0 and s_1 . Recall that by Remark 4 at most one lock is available at this point, so P owns exactly p . Next C needs to execute $C^\top(P, 0)$ twice. If at any moment of this sequence \bar{P} interferes by acquiring s_0 or s_1 when C releases it, then \bar{P} releases \bar{p} , which would become the only free lock. But since $\text{acq}_{\bar{p}}$ does not occur in $C^\top(P, 0)$, C would then loop forever in $Init_C$, which we assumed does not happen. So \bar{P} does not interfere.

Furthermore, for C not to loop forever in $Init_C$, it must be the case that every time C releases a lock, P acquires it and releases the lock that C is waiting for next. As a result, either C loops forever in $Init_C$, or C and P execute *synchronously* twice $C(P, 0)$ and $B(0)$, which is only possible in $Init_P$.

After this part of the run P is in state 1 of Figure 15 and C is ready to run $C^\top(\bar{P}, 0)$ twice. By the same argument, either C loops forever in $Init_C$, or \bar{P} and C synchronize over 0 twice, bringing the *LSS* to configuration *Start*.

Case 2: P starts last.

Let us assume that P does not loop forever in $Init_P$. When P first acquires s_0 in $Init_P$ and releases p , we know by Remark 4 that p is the only free lock at this step. Hence the next operation must be C acquiring p , and then releasing s_1 . Thus C is currently executing *synchronously* with P either $C^\top(P, 0)$ or the main branch of $C(P, 0)$. The two processes must then proceed with the synchronization over bit 0, since otherwise P would loop forever in $Init_P$. Note that \bar{P} cannot interfere by taking s_0 or s_1 , since this would again make P loop forever in $Init_P$, because it does not need \bar{p} .

After P executed the first $B^\top(0)$ *synchronously* with C , it starts the second $B^\top(0)$. For the same reason as before, C has to execute *synchronously* with P either $C^\top(P, 0)$ or the main branch of $C(P, 0)$. However there was no synchronization between P and C over 1 in between, as P kept p between the two synchronizations. As C never executes $C(P, 0)$ twice consecutively outside $Init_C$, C was actually executing $Init_C$ and is now ready to enter the second part, the two executions of $C^\top(\bar{P}, 0)$.

Just like before, if P interferes in the execution of $C^\top(\bar{P}, 0)$, releasing p , then C will loop forever in $Init_C$. We use the same arguments for that part of the run as in case 1.

So in the end either one of the processes loops forever in its synchronizations phase, or all three reach the configuration *Start*.

Case 3: \bar{P} starts last.

We use similar arguments as in case 2, and consider the moment after \bar{P} executed its first action $acq_{\bar{p}}$. Next, process \bar{P} needs to execute twice $\bar{B}^\top(0)$. Assume that \bar{P} does not loop in $Init_{\bar{P}}$. This implies that \bar{P} executes the two copies of $\bar{B}^\top(0)$ *synchronously* with C executing twice $C^\top(0, \bar{P})$ (for the same reason as in case 2, C executes here $C^\top(0, \bar{P})$, not the main branch of $C(0, \bar{P})$). So C must have executed $C(0, P)$ twice before.

Case 3.1: P started before C first acquires p in $Init_C$.

After C first acquires p in $Init_C$ it either executes $C^\top(0, P)$ twice *synchronously* with P executing $B^\top(0)$, or C loops forever in $Init_C$.

Next, C has to execute the two copies of $C^\top(\bar{P}, 0)$. Similar arguments as in case 2 show that either \bar{P} loops forever in $Init_{\bar{P}}$, or C and \bar{P} execute twice *synchronously* $C^\top(\bar{P}, 0)$ and $\bar{B}^\top(0)$. So in the end either one of the processes loops forever in its synchronization sequence, or all three processes reach configuration *Start*.

Case 3.2: P starts after C first acquires p in $Init_C$.

This means that P has to wait until C first releases p to start. If P starts before C acquires p for the second time, by similar arguments as before he has to execute a synchronization over 0 along with C , but needs to wait until the end of $Init_C$ to execute its second synchronization over 0. If P starts after C releases p for the second time, then it cannot acquire p before the end of $Init_C$.

In both cases at the end of C 's initialization phase P has started (as \bar{P} has started, and P starts before \bar{P} in this case) but has at least one synchronization

over 0 left to do from $Init_P$. However C has to synchronize with P over 1 at that point.

We show now that P will loop in any case in $Init_P$. Consider Figure 14: C releases s_1 , which can only be acquired by \overline{P} , who releases \overline{p} . Then C releases s_0 . Next, \overline{P} may acquire it and release s_1 , which prevents processes to make any other operation, while P is still in its initial part. Or P may get it, release p , which C acquires. Thus process P again loops in $Init_P$.

We have considered all cases, and shown that in all of them either one of the processes loops forever in its initialization phase, or they reach configuration $Start$. □

Lemma 48. *There is a winning strategy for the LSS with processes C, P, \overline{P} if and only if the PCP instance has a solution.*

Proof. Suppose that the system wins. Consider a maximal run when the initialization phase is executed entirely synchronously. According to Lemma 47 this run goes through configuration $Start$, after which the processes follow their winning strategy. Note first that C 's strategy cannot stop in any of the states in which she chooses bits or indices, by proposing no controllable action. This is because in such states C holds both s_0 and s_1 . Moreover, as all three processes started, all of them always hold a lock (see Remark 4), hence P holds p and \overline{P} holds \overline{p} . So if C 's strategy would block then the entire system would block, contradicting the winning strategy. As a consequence, the strategies of P and \overline{P} cannot stop either.

By Lemma 46, the strategies of P and \overline{P} produce the same (infinite) binary sequence $b_1b_2\dots$, matching the one produced by C . This is because although Lemma 46 refers to a single synchronization round, such a round cannot interfere with the subsequent rounds: recall from Remark 5 that if C reaches the final state of $C(P, b)$, then this means that P has finished the execution of $B(b)$, and so P waits for C to initiate the next round.

Thus, if the environment chose to enter $C_{letters}$ in C then absence of deadlock means that all three processes proposed the same infinite binary sequence. If the environment chose to enter $C_{indices}$, then absence of deadlock means the sequence produced by P is $u_{i_1}u_{i_2}\dots$ and the one produced by \overline{P} is $v_{i_1}v_{i_2}\dots$, for $i_1i_2\dots$ the index sequence chosen by the winning strategy in C . As the sequences produced by P and \overline{P} are independent of the choice of the environment, they must be equal, and equal to $u_{i_1}u_{i_2}\dots$ and $v_{i_1}v_{i_2}\dots$. Hence, the PCP instance has a solution.

For the other direction assume that PCP has an infinite (unique) solution $u_{i_1}u_{i_2}\dots = v_{i_1}v_{i_2}\dots$. The strategy of C in Figures 12 and 13 consists in choosing the bits/indices provided by the solution. If the initialization phases are not executed entirely synchronously then the system wins by having some process loop forever in its initialization part (Lemma 47). Otherwise, configuration $Start$ is reached and we use Lemma 46 to show that the strategy is winning. □

Lemma 48 completes the proof of Theorem 45.

6 Conclusions

Motivated by a recent undecidability result for distributed control synthesis [18] we have considered a model for which the problem has not been investigated yet. With hindsight it is strange that the well-studied model of lock synchronization has not been considered in the context of distributed synthesis. One reason may be the “non-monotone” nature of the synthesis problem. It is not the case that for a less expressive class of systems the problem is necessarily easier because the controllers get less powerful, too.

The two decidable classes of lock-sharing systems presented here are rather promising. Especially because the low complexity results cover already non-trivial problems. All our algorithms are based on analyzing lock patterns. While in this paper we consider only finite state processes, the same method applies to more complex systems, as long as solving the centralized control problem in the style of Lemma 14 is decidable. This is for example the case for pushdown systems.

There are numerous directions that need to be investigated further. We have focused on deadlock avoidance because this is a central property, and deadlocks are difficult to discover by means of testing or verification. Another option is partial deadlock, where some, but not all, processes are blocked. The concept of *Z-deadlock scheme* should help here, but the complexity results may be different. Reachability, and repeated reachability properties need to be investigated, too.

We do not know if the upper bound from Theorem 6 is tight. The algorithm for verifying if there is a *deadlock* in a given *lock graph*, Algorithm 5, is already quite complicated, and it is not clear how to proceed when a *strategy* is not given.

Another research direction is to consider probabilistic controllers. It is well known that there are no symmetric solutions to the dining philosophers problem but there is a randomized one [25, 26]. Symmetric solutions are quite important for resilience issues as it is preferable that every process runs the same code. The Lehmann-Rabin algorithm is essentially the system presented in Figure 2 where the choice between *left* and *right* is made randomly. This is one of the examples where randomized strategies are essential. Distributed synthesis has a potential here because it is even more difficult to construct distributed randomized systems and prove them correct.

References

- [1] André Arnold and Igor Walukiewicz. Nondeterministic controllers of non-deterministic processes. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata*, volume 2 of *Texts in Logic and Games*, pages 29–52. Amsterdam University Press, 2007.
- [2] Béatrice Bérard, Benedikt Bollig, Patricia Bouyer, Matthias Függer, and Nathalie Sznajder. Synthesis in presence of dynamic links. In Jean-François

- Raskin and Davide Bresolin, editors, *Proceedings 11th International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2020, Brussels, Belgium, September 21-22, 2020*, volume 326 of *EPTCS*, pages 33–49, 2020. To appear in *Information and Computation*.
- [3] Raven Beutner, Bernd Finkbeiner, and Jesko Hecking-Harbusch. Translating asynchronous games for distributed synthesis. In *International Conference on Concurrency Theory (CONCUR'19)*, volume 140 of *LIPIcs*, pages 26:1–26:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
 - [4] Rémi Bonnet, Rohit Chadha, P. Madhusudan, and Mahesh Viswanathan. Reachability under contextual locking. *Log. Methods Comput. Sci.*, 9(3), 2013.
 - [5] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.
 - [6] Alonzo Church. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, volume I, pages 3–50. Cornell Univ., Ithaca, N.Y., 1957.
 - [7] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.
 - [8] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. Locking discipline inference and checking. In *ICSE 2016, Proceedings of the 38th International Conference on Software Engineering*, pages 1133–1144, Austin, TX, USA, May 2016.
 - [9] Bernd Finkbeiner. Bounded synthesis for Petri games. In Roland Meyer, André Platzer, and Heike Wehrheim, editors, *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, volume 9360 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2015.
 - [10] Bernd Finkbeiner, Manuel Giesekeing, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. Global winning conditions in synthesis of distributed systems with causal memory. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
 - [11] Bernd Finkbeiner and Ernst-Ruediger Olderog. Petri games: Synthesis of distributed systems with causal memory. *Inf. Comput.*, 253:181–203, 2017.
 - [12] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *LICS'05*, pages 321–330. IEEE Computer Society, 2005.

- [13] Paul Gastin, Benjamin Lerman, and Marc Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In *FSTTCS'04*, volume 3328 of *LNCS*, pages 275–286. Springer, 2004.
- [14] Paul Gastin, Nathalie Sznajder, and Marc Zeitoun. Distributed synthesis for well-connected architectures. *Formal Methods in System Design*, 34(3):215–237, June 2009.
- [15] Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor Walukiewicz. Asynchronous games over tree architectures. In *International Colloquium on Automata, Languages and Programming (ICALP'13)*, volume 7966 of *LNCS*, pages 275–286. Springer, 2013.
- [16] Manuel Giesekeing, Jesko Hecking-Harbusch, and Ann Yanich. A web interface for Petri nets with transits and Petri games. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 381–388. Springer, 2021.
- [17] Hugo Gimbert. On the control of asynchronous automata. In *FSTTCS'17*, volume 30 of *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
- [18] Hugo Gimbert. Distributed asynchronous games with causal memory are undecidable. *Log. Methods Comput. Sci.*, 18(3), 2022.
- [19] Jesko Hecking-Harbusch and Niklas O. Metzger. Efficient trace encodings of bounded synthesis for asynchronous distributed systems. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 369–386. Springer, 2019.
- [20] Vineet Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 27–36, 2009.
- [21] Vineet Kahlon and Aarti Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 101–110, 2006.
- [22] Vineet Kahlon, Franjo Ivancić, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV'05*, page 505–518, Berlin, Heidelberg, 2005. Springer-Verlag.

- [23] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *LICS'01*, pages 389–398. IEEE, 2001.
- [24] Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Contextual locking for dynamic pushdown networks. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 477–498. Springer, 2013.
- [25] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In John White, Richard J. Lipton, and Patricia C. Goldberg, editors, *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*, pages 133–138. ACM Press, 1981.
- [26] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [27] P. Madhusudan, P. S. Thiagarajan, and Shaofa Yang. The MSO theory of connectedly communicating processes. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 201–212. Springer, 2005.
- [28] P. Madhusudan and P.S. Thiagarajan. Distributed control and synthesis for local specifications. In *ICALP'01*, volume 2076 of *LNCS*, pages 396–407. Springer, 2001.
- [29] Anca Muscholl and Igor Walukiewicz. Distributed synthesis for acyclic architectures. In *FSTTCS'14*, volume 29 of *LIPIcs*, pages 639–651. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.
- [30] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proc. ACM POPL*, pages 179–190, 1989.
- [31] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS'90*, pages 746–757. IEEE Computer Society, 1990.
- [32] Peter J.G. Ramadge and Walter M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(2):81–98, 1989.
- [33] Karen Rudie and W. Murray Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Trans. on Automat. Control*, 37(11):1692–1708, 1992.
- [34] John G. Thistle. Undecidability in decentralized supervision. *Systems & Control Letters*, 54(5):503–509, 2005.
- [35] Stavros Tripakis. Undecidable problems in decentralized observation and control for regular languages. *Information Processing Letters*, 90(1):21–28, 2004.

- [36] Igor Walukiewicz. Synthesis with finite automata. In J. E. Pin, editor, *Handbook of Automata Theory*, volume 2, pages 1215–1258. 2021. <https://www.labri.fr/perso/igw/Papers/igw-synt-chapter.pdf>.
- [37] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott A. Mahlke. The theory of deadlock avoidance via discrete control. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 252–263. ACM, 2009.