Parameterized Verification of Asynchronous Shared-Memory Systems

Javier Esparza[1], Pierre Ganty[2]⋆, and Rupak Majumdar[3]

[1]TU Munich [2]IMDEA Software Institute [3]MPI-SWS

Abstract. We characterize the complexity of the safety verification problem for parameterized systems consisting of a leader process and arbitrarily many anonymous and identical contributors. Processes communicate through a shared, bounded-value register. While each operation on the register is atomic, there is no synchronization primitive to execute a sequence of operations atomically.
We analyze the complexity of the safety verification problem when processes are modeled by finite-state machines, pushdown machines, and Turing machines. The problem is coNP-complete when all processes are finite-state machines, and is PSPACE-complete when they are pushdown machines. The complexity remains coNP-complete when each Turing machine is allowed boundedly many interactions with the register. Our proofs use combinatorial characterizations of computations in the model, and in case of pushdown-systems, some language-theoretic constructions of independent interest.

1 Introduction

We conduct a systematic study of the complexity of safety verification for *parameterized asynchronous shared-memory systems*. These systems consist of a *leader* process and arbitrarily many identical *contributors*, processes with no identity, running at arbitrarily relative speeds and subject to faults (a process can crash). The shared-memory consists of a read/write register that all processes can access to perform either a read operation or a write operation. The register is bounded: the set of values that can be stored is finite. We do insist that read/write operations execute atomically but sequences of operations do not: no process can conduct an atomic sequence of reads and writes while excluding all other processes. The parameterized verification problem for these systems asks to check if a safety property holds no matter how many contributors are present. Our model subsumes the case in which all processes are identical by having the leader process behave like yet another contributor. The presence of a distinguished leader adds (strict) generality to the problem.

We analyze the complexity of the safety verification problem when leader and contributors are modeled by finite state machines, pushdown machines, and even Turing machines. Using combinatorial properties of the model that allow simulating arbitrarily many contributors using finitely many ones, we show that if leader and contributors are finite-state machines the problem is coNP-complete. The case in which leader and contributors are pushdown machines was first considered by Hague [18], who gave a coNP

⋆ Supported by the Spanish projects with references TIN2010-20639 and TIN2012-39391-C04.

lower bound and a 2EXPTIME upper bound. We close the gap and prove that the problem is PSPACE-complete. Our upper bound requires several novel language-theoretic constructions on bounded-index approximations of context-free languages. Finally, we address the bounded safety problem, i.e., deciding if no error can be reached by computations in which no contributor nor the leader execute more than a given number k of steps (this does not bound the length of the computation, since the number of contributors is unbounded). We show that (if k is given in unary) the problem is coNP-complete not only for pushdown machines, but also for arbitrary Turing machines. Thus, the safety verification problem when the leader and contributors are poly-time Turing machines is also coNP-complete.

These results show that non-atomicity substantially reduces the complexity of verification. In the atomic case, contributors can ensure that they are the only ones that receive a message: the first contributor that reads the message from the store can also erase it within the same atomic action. This allows the leader to distribute identities to contributors. As a consequence, the safety problem is at least PSPACE-hard for state machines, and undecidable for pushdown machines (in the atomic case, the safety problem of two pushdown machines is already undecidable). A similar argument shows that the bounded safety problem is PSPACE-hard. In contrast, we get several coNP upper bounds, which opens the way to the application of SAT-solving or SMT-techniques.

Besides intellectual curiosity, our work on this model is motivated by practical distributed protocols implemented on wireless sensor networks. In these systems, a central co-ordinator (the base station) communicates with an arbitrary number of mass-produced tiny agents (or motes) that run concurrently and asynchronously. The motes have limited computational power, and for some systems such as vehicular networks anonymity is a requirement [21]. Further, they are susceptible to crash faults. Implementing atomic communication primitives in this setting is expensive and can be problematic: for instance, a process might crash while holding a lock. Thus, protocols in these systems work asynchronously and without synchronization primitives. Our algorithms provide the foundations for safety verification of these systems.

Related Works. Parameterized verification problems have been extensively studied both theoretically and practically. It is a computationally hard problem: the reachability problem is undecidable even if each process has a finite state space [2]. For this reason, special cases have been extensively studied. They vary according to the main characteristics of the systems to verify like the communication topology of the processes (array, tree, unordered, etc); their communication primitives (shared memory, unreliable broadcasts, (lossy) queues, etc); or whether processes can distinguish from each other (using ids, a distinguished process, etc). Prominent examples include broadcast protocols [13,15,10,9], where finite-state processes communicate via broadcast messages, asynchronous programs [16,25], where finite-state processes communicate via unordered channels, finite-state processes communicating via ordered channels [1], micro architectures [22], cache coherence protocols [11,7], communication protocols [12], multithreaded shared-memory programs [5,8,20,24].

Besides the model of Hague [18], the closest model to ours that has been previously studied [17] is that of distributed computing with identity-free, asynchronous processors and non-atomic registers. The emphasis there was the development of distributed

algorithm primitives such as time-stamping, snapshots, and consensus, using either unbounded registers or an unbounded number of bounded registers.

It was left open if these primitives can be implemented using a bounded number of bounded registers. Our decidability results indicate that this is not possible: the safety verification problem would be undecidable if such primitives could be implemented.

2 Formal Model: Non-Atomic Networks

We describe our formal model, called *non-atomic networks*. We take a language-theoretic view, identifying a system with the language of its executions.

Preliminaries. A *labeled transition system* (LTS) is a quadruple $\mathcal{T} = (\Sigma, Q, \delta, q_0)$, where Σ is a finite set of *action labels*, Q is a (non necessarily finite) set of *states*, $q_0 \in Q$ is the *initial state*, and $\delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times Q$ is the *transition relation*, where ε is the empty string. We write $q \xrightarrow{a} q'$ for $(q, a, q') \in \delta$. For $\sigma \in \Sigma^*$, we write $q \xrightarrow{\sigma} q'$ if there exist $q_1, \ldots, q_n \in Q$ and $a_0, \ldots, a_n \in \Sigma \cup \{\varepsilon\}$, $q \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \cdots q_n \xrightarrow{a_n} q'$ such that $a_0 \cdots a_n = \sigma$. The sequence $q \cdots q'$ is called a *path* and σ its *label*. A *trace* of $\mathcal{T}$ is a sequence $\sigma \in \Sigma^*$ such that $q_0 \xrightarrow{\sigma} q$ for some $q \in Q$. Define $L(\mathcal{T})$, the *language* of $\mathcal{T}$, as the set of traces of $\mathcal{T}$. Note that $L(\mathcal{T})$ is *prefix closed*: $L(\mathcal{T}) = \mathit{Pref}(L(\mathcal{T}))$ where $\mathit{Pref}(L) = \{s \mid \exists u\colon\ s\,u \in L\}$

To model concurrent executions of LTSs, we introduce two operations on languages: the shuffle and the asynchronous product. The *shuffle* of two words $x, y \in \Sigma^*$ is the language $x \sqcup\!\sqcup y = \{x_1 y_1 \ldots x_n y_n \in \Sigma^* \mid \text{each } x_i, y_i \in \Sigma^* \text{and } x = x_1 \cdots x_n \wedge y = y_1 \cdots y_n\}$. The shuffle of two languages L_1, L_2 is the language $L_1 \sqcup\!\sqcup L_2 = \bigcup_{x \in L_1, y \in L_2} x \sqcup\!\sqcup y$. Shuffle is associative, and so we can write $L_1 \sqcup\!\sqcup \cdots \sqcup\!\sqcup L_n$ or $\sqcup\!\sqcup_{i=1}^n L_i$.

The *asynchronous product* of two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, denoted $L_1 \parallel L_2$, is the language L over the alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ such that $w \in L$ iff the projections of w to Σ_1 and Σ_2 belong to L_1 and L_2, respectively.[1] If a language consists of a single word, e.g. $L_1 = \{w_1\}$, we abuse notation and write $w_1 \parallel L_2$. Asynchronous product is also associative, and so we write $L_1 \parallel \cdots \parallel L_n$ or $\|_{i=1}^n L_i$.

Let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ be LTSs, where $\mathcal{T}_i = (\Sigma_i, Q_i, \delta_i, q_{0i})$. The *interleaving* $\sqcup\!\sqcup_{i=1}^n \mathcal{T}_i$ is the LTS with actions $\bigcup_{i=1}^n \Sigma_i$, set of states $Q_1 \times \cdots \times Q_n$, initial state $(q_{01}, \ldots, q_{0n})$, and a transition $(q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n)$ iff $(q_i, a, q'_i) \in \delta_i$ for some $1 \leq i \leq n$ and $q_j = q'_j$ for every $j \neq i$. Interleaving models parallel composition of LTSs that do not communicate at all. The language $L(\mathcal{T}_1 \sqcup\!\sqcup \cdots \sqcup\!\sqcup \mathcal{T}_n)$ of the interleaving is $\sqcup\!\sqcup_{i=1}^n L(\mathcal{T}_i)$.

The *asynchronous parallel composition* $\|_{i=1}^n \mathcal{T}_i$ of $\mathcal{T}_1, \ldots, \mathcal{T}_n$ is the LTS having $\bigcup_{i=1}^n \Sigma_i$ as set of actions, $Q_1 \times \cdots \times Q_n$ as set of states, $(q_{01}, \ldots, q_{0n})$ as initial state, and a transition $(q_1, \ldots, q_n) \xrightarrow{a} (q'_1, \ldots, q'_n)$ if and only if

1. $a \neq \varepsilon$ and for all $1 \leq i \leq n$ either $a \notin \Sigma_i$ and $q_i = q'_i$ or $a \in \Sigma_i$ and $(q_i, a, q'_i) \in \delta_i$, or;
2. $a = \varepsilon$, and there is $1 \leq i \leq n$ such that $(q_i, \varepsilon, q'_i) \in \delta_i$ and $q_j = q'_j$ for every $j \neq i$.

Asynchronous parallel composition models the parallel composition of LTSs in which an action a must be simultaneously executed by every LTSs having a in its alphabet. $L(\mathcal{T}_1 \parallel \cdots \parallel \mathcal{T}_n)$, the language of the asynchronous parallel composition, is $\|_{i=1}^n L(\mathcal{T}_i)$.

[1] Observe that the $L_1 \parallel L_2$ depends on L_1, L_2 **and** also their underlying alphabet Σ_1 and Σ_2.

Non-atomic networks. We fix a finite non-empty set $\mathcal{G}$ of *global values*. A *read-write alphabet* is any set of the form $A \times \mathcal{G}$, where A is a set of *read* and *write actions*, or just *reads* and *writes*. We denote a letter $(a, g) \in A \times \mathcal{G}$ by $a(g)$, and write $\mathcal{G}(a_1, \ldots, a_n)$ instead of $\{a_1, \ldots, a_n\} \times \mathcal{G}$.

In what follows, we consider LTSs over read-write alphabets. We fix two LTSs $\mathcal{D}$ and C, called the *leader* and the *contributor*, with alphabets $\mathcal{G}(r_d, w_d)$ and $\mathcal{G}(r_c, w_c)$, respectively, where r_d, r_c are called reads and w_c, w_d are called writes. We write $w_\star$ (respectively, $r_\star$) to stand for either w_c or w_d (respectively, r_c or r_d). We also assume that for each value $g \in \mathcal{G}$ there is a transition in the leader or contributor which reads or writes g (if not, the value is never used and is removed from $\mathcal{G}$).

Additionally, we fix an LTS $\mathcal{S}$ called a *store*, whose states are the global values of $\mathcal{G}$ and whose transitions, labeled with the read-write alphabet, represent possible changes to the global values on reads and writes. No read is enabled initially. Formally, the *store* is an LTS $\mathcal{S} = (\Sigma, \mathcal{G} \cup \{g_0\}, \delta_S, g_0)$, where $\Sigma = \mathcal{G}(r_d, w_d, r_c, w_c)$, g_0 is a designated initial value not in $\mathcal{G}$, and δ_S is the set of transitions $g \xrightarrow{r_\star(g)} g$ and $g' \xrightarrow{w_\star(g)} g$ for all $g \in \mathcal{G}$ and all $g' \in \mathcal{G} \cup \{g_0\}$. Observe that fixing $\mathcal{D}$ and C also fixes $\mathcal{S}$.

Definition 1. *Given a pair $(\mathcal{D}, C)$ of a leader $\mathcal{D}$ and contributor C, and $k \geq 1$, define $\mathcal{N}_k$ to be the LTS $\mathcal{D} \parallel \mathcal{S} \parallel \sqcup_k C$, where $\sqcup_k C$ is $\sqcup_{i=1}^{k} C$. The* (non-atomic) $(\mathcal{D}, C)$-network *$\mathcal{N}$ is the set $\{\mathcal{N}_k \mid k \geq 1\}$, with* language $L(\mathcal{N}) = \bigcup_{k=1}^{\infty} L(\mathcal{N}_k)$. *We omit the prefix $(\mathcal{D}, C)$ when it is clear from the context.*

Notice that $L(\mathcal{N}_k) = L(\mathcal{D}) \parallel L(\mathcal{S}) \parallel \sqcup_k L(C)$ and $L(\mathcal{N}) = L(\mathcal{D}) \parallel L(\mathcal{S}) \parallel \sqcup_\infty L(C)$, where $\sqcup_\infty L(C)$ is given by $\bigcup_{k=1}^{\infty} \sqcup_k L(C)$.

The safety verification problem. A trace of a $(\mathcal{D}, C)$-network $\mathcal{N}$ is unsafe if it ends with an occurrence of $w_c(\#)$, where # is a special value of $\mathcal{G}$. Intuitively, an occurrence of $w_c(\#)$ models that the contributor raises a flag because some error has occurred. A $(\mathcal{D}, C)$-network $\mathcal{N}$ is *safe* iff its language contains no unsafe trace, namely $L(\mathcal{N}) \cap \Sigma^* w_c(\#) = \emptyset$. (We could also require the leader to write #, or to reach a certain state; all these conditions are easily shown equivalent.)

Given a machine M having an LTS semantics over some read-write alphabet, we denote its LTS by $[\![M]\!]$. Given machines M_D and M_C over read-write alphabets, The *safety verification problem* for machines M_D and M_C consists of deciding if the $([\![M_D]\!], [\![M_C]\!])$-network is safe. Notice that the size of the input is the size of the machines, and not the size of the LTSs thereof, which might even be infinite.

Our goal is to characterize the complexity of the safety verification problem considering various types of machines for the leader and the contributors. We first establish some fundamental combinatorial properties of non-atomic networks.

3 Simulation and Monotonicity

We prove two fundamental combinatorial properties of non-atomic networks: the Simulation and Monotonicity Lemmas. Informally, the Simulation Lemma states that a leader cannot distinguish an unbounded number of contributors from the parallel composition of at most $|\mathcal{G}|$ *simulators*—LTSs derivable from the contributors, one for each value

of $\mathcal{G}$. The Monotonicity Lemma states that non-minimal traces (with respect to a certain subword order) can be removed from a simulator without the leader "noticing", and, symmetrically, non-maximal traces can be removed from the leader without the simulators "noticing".

3.1 Simulation

First writes and useless writes. Let σ be a trace. The *first write* of g in σ by a contributor is the first occurrence of $w_c(g)$ in σ. A *useless write* of g by a contributor is any occurrence of $w_c(g)$ that is immediately overwritten by another write. For technical reasons, we additionally assume that useless writes are not first writes.

Example 1. In a network trace $w_d(g_1)_1\, w_c(g_2)_2\, w_c(g_3)_3\ r_d(g_3)_4\, w_c(g_2)_5\, w_c(g_1)_6$ where we have numbered occurrences, $w_c(g_2)_2$ is a first write of g_2, and $w_c(g_2)_5$ is a useless write of g_2 (even though $w_c(g_2)_2$ is immediately overwritten).

We make first writes and useless writes explicit by adding two new actions f_c and u_c to our LTSs, and adequately adapting the store.

Definition 2. *The* extension *of an LTS* $\mathcal{T} = (\mathcal{G}(r,w), Q, \delta, q_0)$ *is the LTS* $\mathcal{T}^E = (\mathcal{G}(r,w,f,u), Q, \delta^E, q_0)$*, where* f, u *are the* first write *and* useless write *actions, respectively, and*

$$\delta^E = \delta \cup \{(q, f(g), q'), (q, u(g), q') \mid (q, w(g), q') \in \delta\} \ .$$

We define an *extended store*, whose states are triples (g, W, b), where $g \in \mathcal{G}$, $W\colon \mathcal{G} \to \{0,1\}$ is the *write record*, and $b \in \{0,1\}$ is the *useless flag*. Intuitively, W records the values written by the contributors so far. If $W(g) = 0$, then a write to g must be a first write, and otherwise a regular write or a useless write. The useless flag is set to 1 by a useless write, and to 0 by other writes. When set to 1, the flag prevents the occurrence of a read. The flag only ensures that between a useless write and the following write no read happens, i.e., that a write tagged as useless will indeed be semantically useless. A regular or first write may be semantically useless or not.

Definition 3. *The* extended store *is the LTS* $S^E = (\Sigma_E, \mathcal{G}_E, \delta_{S^E}, c_0)$ *where*

- $\Sigma_E = \mathcal{G}(r_d, w_d, r_c, w_c, f_c, u_c)$*;*
- $\mathcal{G}_E$ *is the set of triples* (g, W, b)*, where* $g \in \mathcal{G} \cup \{g_0\}$*,* $W : \mathcal{G} \to \{0,1\}$*, and* $b \in \{0,1\}$*;*
- c_0 *is the triple* $(g_0, W_0, 0)$*, where* $W_0(g) = 0$ *for every* $g \in \mathcal{G}$*;*
- δ_{S^E} *has a transition* $(g, W, b) \xrightarrow{a} (g', W', b')$ *where* $g' \in \mathcal{G}$ *iff one of the following conditions hold:*
  - $a = r_\star(g)$*,* $g' = g$*,* $W' = W$*, and* $b = b' = 0$*;*
  - $a = w_d(g')$*,* $W' = W$ *and* $b' = 0$*;*
  - $a = f_c(g')$*,* $W(g') = 0$*,* $W' = W[W(g')/1]$*, and* $b' = 0$*;*
  - $a = w_c(g')$*,* $W(g') = 1$*,* $W' = W$*, and* $b' = 0$*;*
  - $a = u_c(g')$*,* $W(g') = 1$*,* $W' = W$*, and* $b' = 1$.

The extension of $\mathcal{N}_k$ *is* $\mathcal{N}_k^E = \mathcal{D} \parallel S^E \parallel \sqcup_k C^E$ *and the extension of* $\mathcal{N}$ *is the set* $\mathcal{N}^E = \{\mathcal{N}_k^E \mid k \geq 1\}$*. The languages* $L(\mathcal{N}_k^E)$ *and* $L(\mathcal{N}^E)$ *are defined as in Def. 1.*

It follows immediately from this definition that if $v \in L(\mathcal{N}^E)$ then the sequence v' obtained of replacing every occurrence of $f_c(g), u_c(g)$ in v by $w_c(g)$ belongs to $L(\mathcal{N})$. Conversely, every trace v' of $L(\mathcal{N})$ can be transformed into a trace v of $L(\mathcal{N}^E)$ by adequately replacing some occurrences of $w_c(g)$ by $f_c(g)$ or $u_c(g)$.

In the sequel, we use sequences of first writes to partition sets of traces. Define Υ to be the (finite) set of sequences over $\mathcal{G}(f_c)$ with no repetitions. By the very idea of "first writes" no sequence of Υ writes the same value twice, hence no word in Υ is longer than $|\mathcal{G}|$. Also define $\Upsilon_\#$ to be those words of Υ which ends with $f_c(\#)$. Given $\tau \in \Upsilon$, define P_τ to be the language given by $(\Sigma_E \setminus \mathcal{G}(f_c))^* \sqcup\!\sqcup \tau$. P_τ contains all the sequences over Σ_E in which the subsequence of first writes is exactly τ. For $S \subseteq \Upsilon$, $P_S = \bigcup_{\sigma \in S} P_\sigma$.

The Copycat Lemma. Intuitively, a *copycat* contributor is one that follows another contributor in all its actions: it reads or writes the same value immediately after the other reads or writes. Informally, the copycat lemma states that any trace of a non-atomic network can be extended with copycat contributors.

Consider first the non-extended case. Clearly, for every trace of $\mathcal{N}_k$ there is a trace of $\mathcal{N}_{k+1}$ in which the leader and the first k contributors behave as before, and the $(k+1)$-th contributor behaves as a copycat of one of the first k contributors, say the i-th: if the i-th contributor executes a read $r_c(g)$, then the $(k+1)$-th contributor executes the same read *immediately after*, and the same for a write.

Example 2. Consider the trace $r_c(g_0)\, w_d(g_1)\, r_c(g_1)\, w_c(g_2)$ of $\mathcal{D} \parallel \mathcal{S} \parallel C$. Then the sequence $r_c(g_0)^2\, w_d(g_1)\, r_c(g_1)^2\, w_c(g_2)^2$ is a trace of $\mathcal{D} \parallel \mathcal{S} \parallel (C \sqcup\!\sqcup C)$.

For the case of extended networks, a similar result holds, but the copycat copies a first write by a regular write: if the i-th contributor executes an action other than $f_c(g)$, the copycat contributor executes the same action immediately after, but if the i-th contributor executes $f_c(g)$, then the copycat executes $w_c(g)$.

Definition 4. *We say $u \in \mathcal{G}(r_d, w_d)^*$ is* compatible *with a multiset $M = \{v_1, \ldots, v_k\}$ of words over $\mathcal{G}(f_c, w_c, u_c, r_c)$ (possibly containing multiple copies of a word) iff $u \parallel L(S^E) \parallel \sqcup\!\sqcup_{i=1}^{k} v_i \neq \emptyset$. Let $\tau \in \Upsilon$. We say u is compatible with M following τ iff $P_\tau \cap (u \parallel L(S^E) \parallel \sqcup\!\sqcup_{i=1}^{k} v_i) \neq \emptyset$.*

Lemma 1. *Let $u \in \mathcal{G}(r_d, w_d)^*$ and let M be a multiset of words over $\mathcal{G}(r_c, f_c, w_c, u_c)$. If u is compatible with M, then u is compatible with every M' obtained by erasing symbols from $\mathcal{G}(r_c)$ and $\mathcal{G}(u_c)$ from the words of M.*

Proof. Erasing reads and useless writes (that no one reads) by contributors does not affect the sequence of values written to the store and read by someone, hence compatibility is preserved. □

Lemma 2 (Copycat Lemma). *Let $u \in \mathcal{G}(r_d, w_d)^*$, let M be a multiset over $L(C^E)$ and let $v' \in M$. Given a prefix v of v' we have that if u is compatible with M, then u is compatible with $M \oplus v[f_c(g)/w_c(g)]$.*[2]

[2] Throughout the paper, we use $\{\}, \oplus, \ominus$, and $\geq$ for the multiset constructor, union, difference and inclusion, respectively. The word $w[a/b]$ results from w by replacing all occurrences of a by b.

Example 3. $r_d(g_1)$ is compatible with $f_c(g_1)f_c(g_2)$. By the Copycat Lemma $r_d(g_1)$ is also compatible with $\{f_c(g_1)f_c(g_2), w_c(g_1)\,w_c(g_2)\}$. Indeed, $f_c(g_1)\,w_c(g_1)\,r_d(g_1)f_c(g_2)\,w_c(g_2) \in L(S^E)$ is a trace (even though $f_c(g_2)$ is useless).

The Simulation Lemma. The simulation lemma states that we can replace unboundedly many contributors by a finite number of LTSs that "simulate" them. In particular the network is safe iff its simulation is safe.

Let $v \in L(C^E)$. Let $\#v$ be the number of times that actions of $\mathcal{G}(f_c, w_c)$ occur in v, minus one if the last action of v belongs to $\mathcal{G}(f_c, w_c)$. E.g., $\#v = 1$ for $v = f_c(g_1)r_c(g_1)$ but $\#v = 0$ for $v = r_c(g_1)f_c(g_1)$. The next lemma is at the core of the simulation theorem.

Lemma 3. *Let $u \in L(\mathcal{D})$ and let $M = \{v_1, \ldots, v_k\}$ be a multiset over $L(C^E)$ compatible with u. Then u is compatible with a multiset $\tilde{M}$ over $L(C^E) \cap \mathcal{G}(r_c, u_c)^*\ \mathcal{G}(f_c, w_c)$.*

Proof. Since u is compatible with M, $u \parallel L(S^E) \parallel \sqcup\!\sqcup_{i=1}^{k} v_i \neq \emptyset$. Lemma 1 shows that we can drop from M all the v_i such that $v_i \in \mathcal{G}(r_c, u_c)^*$. Further, define $\#M = \sum_{i=1}^{k} \#v_i$. We proceed by induction on $\#M$. If $\#M = 0$, then all the words of M belong to $\mathcal{G}(r_c, u_c)^*\ \mathcal{G}(f_c, w_c)$, and we are done. If $\#M > 0$, then there is $v_i \in M$ such that $v_i = \alpha_i\,\sigma\,\beta_i$, where $\alpha_i \in \mathcal{G}(r_c, u_c)^*$, $\sigma \in \mathcal{G}(f_c, w_c)$, and $\beta_i \neq \varepsilon$. Let g be the value written by σ, and let $v_{k+1} = \alpha_i w_c(g)$. By Lemma 2, u is compatible with $\{v_1, \ldots, v_{k+1}\}$, and so there is $v' \in u \parallel L(S^E) \parallel \sqcup\!\sqcup_{i=1}^{k+1} v_i$ in which the write σ of v_i occurs in v' immediately before the write of v_{k+1}. We now let the writes occur in the reverse order, which amounts to replacing v_i by $v'_i = \alpha_i\,u_c(g)\,\beta_i$ and v_{k+1} by $v'_{k+1} = \alpha_i\,\sigma$. This yields a new multiset $M' = M \ominus \{v_i\} \oplus \{v'_i, v'_{k+1}\}$ compatible with u. Since $\#M' = \#M - 1$, we then apply the induction hypothesis to M', obtain $\tilde{M}$ and we are done. □

Definition 5. *For all $g \in \mathcal{G}$, let $L_g = L(C^E) \cap \mathcal{G}(r_c, u_c)^* f_c(g)$. Define S_g be an LTS over $\mathcal{G}(r_c, u_c, f_c, w_c)$ such that $L(S_g) = Pref(L_g \cdot w_c(g)^*)$. Define the LTS $\mathcal{N}^S = \mathcal{D} \parallel S^E \parallel \sqcup\!\sqcup_{g\in\mathcal{G}} S_g$ which we call the* simulation *of $\mathcal{N}^E$.*

Lemma 4. *Let $u \in L(\mathcal{D})$ and let $M = \{v_1, \ldots, v_k\}$ be a multiset over $L(C^E) \cap \mathcal{G}(r_c, u_c)^*\ \mathcal{G}(f_c, w_c)$ compatible with u. Then u is compatible with a set $S = \{s_g\}_{g\in\mathcal{G}}$ where $s_g \in L(S_g)$.*

Proof. Let us partition the multiset M as $\{M_g\}_{g\in\mathcal{G}}$ such that M_g contains exactly the traces of M ending with $f_c(g)$ or $w_c(g)$. Note that some M_g might be empty. Each non-empty M_g is of the form $M_g = \{x_1 f_c(g), x_2 w_c(g), \ldots, x_n w_c(g)\}$ where $n \geq 1$, and $x_i \in \mathcal{G}(r_c, u_c)^*$ for every $1 \leq i \leq n$. Define M'_g as empty if M_g is empty, and M'_g as M_g together with $n-1$ copies of $x_1 w_c(g)$. The copycat lemma shows that u is compatible with $\oplus_{g\in\mathcal{G}} M'_g$. Let us now define the multiset M''_g to be empty if M'_g is empty, and the multiset of exactly n elements given by $x_1 f_c(g)$ and $n-1$ copies of $x_1 w_c(g)$ if M'_g is not empty. Again we show that u is compatible with $\oplus_{g\in\mathcal{G}} M''_g$. The reason is that the number $n-1$ of actions $w_c(g)$ in each M''_g does not change (compared to M_g) and each $w_c(g)$ action can happen as soon as $f_c(g)$ has occurred.

Now define S consisting of one trace s_g for each $g \in \mathcal{G}$ such that $s_g = \varepsilon$ if $M''_g = \emptyset$; and $s_g = x_1 f_c(g)\, w_c(g)^{n-1}$ if M''_g consists of $x_1 f_c(g)$ and $n-1$ copies of $x_1 w_c(g)$.

We have that u is compatible with S because the number of $f_c(g)$ and $w_c(g)$ actions in M''_g and s_g does not change and each $w_c(g)$ action can happen as soon as $f_c(g)$ has

occurred. Note that it need not be the case that $s_g \in L(C^E)$. However each $s_g \in L(S_g)$ (recall that each $L(S_g)$ is prefix closed). □

Corollary 1. *Let $u \in L(\mathcal{D})$ and let $M = \{v_1, \ldots, v_k\}$ be a multiset over $L(C^E)$ compatible with u. Then u is compatible with a set $S = \{s_g\}_{g \in \mathcal{G}}$ where $s_g \in L(S_g)$.*

In Lemmas 1,2,3 and 4 and Corollary 1 compatibility is preserved. We can further show that it is preserved following a given sequence of first writes. For example, in Lem. 3 if u is compatible with M following τ then u is compatible with $\tilde{M}$ following τ.

Lemma 5 (Simulation Lemma). *Let $\tau \in \Upsilon$:*

$$L(\mathcal{N}^E) \cap P_\tau \neq \emptyset \quad \textit{iff} \quad L(\mathcal{N}^S) \cap P_\tau \neq \emptyset \ .$$

Proof. ($\Rightarrow$): The hypothesis and the definition of $\mathcal{N}^E$ shows that there is $k \geq 1$ such that $P_\tau \cap (L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup_k L(C^E)) \neq \emptyset$.

Therefore we conclude that there exists $u \in L(\mathcal{D})$ and $M = \{v_1, \ldots, v_k\}$ over $L(C^E)$ such that u is compatible with M following τ. Corollary 1 shows that u is compatible following τ with a set $S = \{s_g\}_{g \in \mathcal{G}}$ where $s_g \in L(S_g)$. Therefore we have $P_\tau \cap (u \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g)) \neq \emptyset$, hence that $P_\tau \cap (L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g)) \neq \emptyset$ and finally that $P_\tau \cap L(\mathcal{N}^S) \neq \emptyset$.

($\Leftarrow$): The hypothesis and the definition of $\mathcal{N}^S$ shows that $P_\tau \cap (L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g)) \neq \emptyset$. Hence we find that there exists $u \in L(\mathcal{D})$ and a set $\{x_g\}_{g \in \mathcal{G}}$ where $x_g \in L(S_g)$ such that $P_\tau \cap (u \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} x_g) \neq \emptyset$. The prefix closure of $L(S_g)$ shows that either x_g does not have a first write or $x_g = v_g f_c(g) w_c(g)^{n_g}$ for some $v_g f_c(g) \in L_g$ and $n_g \in \mathbb{N}$. In the former case, that is $x_g \in \mathcal{G}(r_c, u_c)^*$, Lemma 1 shows that discarding the trace does not affect compatibility. Then define the multiset M containing for each remaining trace $x_g = v_g f_c(g) w_c(g)^{n_g}$ the trace $v_g f_c(g)$ and n_g traces $v_g w_c(g)$. M contains no other element. Using a copycat-like argument, it is easy to show M is compatible with u and further that compatibility follows τ. Finally, because $v_g f_c(g) \in L(C^E) \cap \mathcal{G}(r_c, u_c)^* \mathcal{G}(f_c)$ and because C^E is the extension of C we find that every trace of M is also a trace of C^E, hence that there exists $k \geq 1$ such that $P_\tau \cap (L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup_k L(C^E)) \neq \emptyset$, and finally that $L(\mathcal{N}^E) \cap P_\tau \neq \emptyset$. □

Let us now prove an equivalent safety condition.

Proposition 1. *A $(\mathcal{D}, C)$-network $\mathcal{N}$ is safe iff $L(\mathcal{N}^S) \cap P_{\Upsilon_\#} = \emptyset$.*

Proof. From the semantics of non-atomic networks, $\mathcal{N}$ is unsafe if and only if $L(\mathcal{N}) \cap (\Sigma^* w_c(\#)) \neq \emptyset$, equivalently, $L(\mathcal{N}^E) \cap (\Sigma_E^* f_c(\#)) \neq \emptyset$ (by definition of extension), which in turn is equivalent to $L(\mathcal{N}^E) \cap P_{\Upsilon_\#} \neq \emptyset$ (by definition of $P_{\Upsilon_\#}$), if and only if $L(\mathcal{N}^S) \cap P_{\Upsilon_\#} \neq \emptyset$ (by the simulation lemma). □

3.2 Monotonicity

Before stating the monotonicity lemma, we need some language-theoretic definitions. For an alphabet Σ, define the *subword ordering* $\preceq \subseteq \Sigma^* \times \Sigma^*$ on words as $u \preceq v$ iff u results from v by deleting some occurrences of symbols. Let $L \subseteq \Sigma^*$, define $S \subseteq L$ to be

- *cover* of L if for every $u \in L$ there is $v \in S$ such that $u \preceq v$;

– *support* of L if for every $u \in L$ there is $v \in S$ such that $v \preceq u$.

Observe that for every $u, v \in S$ such that $u \prec v$: if S is a cover then so is $S \setminus \{u\}$, and if S is a support then so is $S \setminus \{v\}$.

Recall that $\mathcal{N}^S = \mathcal{D} \parallel S^E \parallel \sqcup_{g \in \mathcal{G}} S_g$. It is convenient to introduce a fourth, redundant component that does not change $L(\mathcal{N}^S)$, but exhibits important properties of it. Recall that the leader cannot observe the reads of the contributors, and does not read the values introduced by useless writes. We introduce a local copy $S^E_{\mathcal{D}}$ of the store with alphabet $\mathcal{G}(r_d, w_d, f_c, w_c)$ that behaves like S^E for writes and first writes of the contributors, but has neither contributor reads nor useless writes in its alphabet. Formally:

Definition 6. *The* leader store $S^E_{\mathcal{D}}$ *is the LTS* $(\Sigma^E_D, \mathcal{G}^{\mathcal{D}}_E, \delta^E_D, c_0)$*,*

- $\Sigma^E_D = \mathcal{G}(r_d, w_d, f_c, w_c)$*;*
- $\mathcal{G}^{\mathcal{D}}_E$ *is the set of pairs* (g, W)*, where* $g \in \mathcal{G} \cup \{g_0\}$ *and* $W : \mathcal{G} \to \{0, 1\}$*;*
- c_0 *is the pair* (g_0, W_0)*, where* $W_0(g) = 0$ *for every* $g \in \mathcal{G}$*;*
- δ^E_D *has a transition* $(g, W) \xrightarrow{a} (g', W')$ *where* $g' \in \mathcal{G}$ *iff one of the following conditions hold: a)* $a{=}w_d(g')$ *and* $W' = W$*; b)* $a{=}r_d(g)$*,* $g' = g$*, and* $W' = W$*; c)* $a{=}f_c(g')$*,* $W(g') = 0$*, and* $W' = W[W(g')/1]$*; d)* $a{=}w_c(g')$*,* $W(g') = 1$*, and* $W' = W$*.*

It follows easily from this definition that $L(S^E_{\mathcal{D}})$ is the projection of $L(S^E)$ onto Σ^E_D, and so $L(S^E) = L(S^E_{\mathcal{D}}) \parallel L(S^E)$ holds. Now, define $\mathcal{DS} = \mathcal{D} \parallel S^E_{\mathcal{D}}$, we find that:

$$\begin{aligned} L(\mathcal{N}^S) &= L(\mathcal{D} \parallel S^E \parallel \sqcup_{g \in \mathcal{G}} S_g) && \text{def. 5} \\ &= L(\mathcal{D}) \parallel L(S^E_{\mathcal{D}}) \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g) \\ &= L(\mathcal{D} \parallel S^E_{\mathcal{D}}) \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g) \\ &= L(\mathcal{DS} \parallel S^E \parallel \sqcup_{g \in \mathcal{G}} S_g) && (1) \end{aligned}$$

Lemma 6 (Monotonicity Lemma). *Let* $\tau \in \Upsilon$ *and let* $\hat{L}_\tau$ *be a cover of* $L(\mathcal{DS}) \cap P_\tau$*. For every* $g \in \mathcal{G}$*, let* $\underline{L}_g$ *be a support of* L_g*, and let* $\underline{S}_g$ *be an LTS such that* $L(\underline{S}_g) = \mathit{Pref}(\underline{L}_g \cdot w_c^*(g))$*:*

$$(L(\mathcal{DS}) \cap P_\tau) \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g) \neq \emptyset \;\; \textit{iff} \;\; \hat{L}_\tau \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(\underline{S}_g) \neq \emptyset \;\; .$$

The proof of the monotonicity lemma breaks down into monotonicity for the contributors (Lemma 7) and for the leader (Lemma 9).

Lemma 7 (Contributor Monotonicity Lemma). *For every* $g \in \mathcal{G}$*, let* $\underline{L}_g$ *be a support of* L_g*, and let* $\underline{S}_g$ *be an LTS such that* $L(\underline{S}_g) = \mathit{Pref}(\underline{L}_g w_c^*(g))$*. Let* $u \in \mathcal{G}(r_d, w_d)^*$ *and* $\tau \in \Upsilon$*:*

$$(u \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g)) \cap P_\tau \neq \emptyset \;\; \textit{iff} \;\; (u \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(\underline{S}_g)) \cap P_\tau \neq \emptyset \;\; .$$

Proof. ($\Leftarrow$): It suffices to observe that since $\underline{L}_g \subseteq L_g$ we have $L(\underline{S}_g) \subseteq L(S_g)$ and we are done. ($\Rightarrow$): Since $L_g \subseteq \mathcal{G}(r_c, u_c)^* f_c(g)$ and $\underline{L}_g \subseteq L_g$ we find that for every word $w' \in L_g \setminus \underline{L}_g$ there exists a word $w \in \underline{L}_g$ resulting from w' by erasing symbols in $\mathcal{G}(u_c, r_c)$. Hence, Lemma 1 shows that erasing symbols in $\mathcal{G}(u_c, r_c)$ does not affect compatibility. The proof concludes by observing that compatibility is further preserved for τ, and we are done. □

The leader monotonicity lemma requires the following technical observation.

Lemma 8. *Let $\tau \in \Upsilon$ and $L \subseteq \mathcal{G}(r_c, f_c, w_c, u_c)^*$ satisfying the following condition: if $\alpha f_c(g)\beta_1\beta_2 \in L$, then $\alpha f_c(g)\beta_1 w_c(g)\beta_2 \in L$. For every $v, v' \in P_\tau \cap L(S^E_{\mathcal{D}})$:*

$$\text{if } v \parallel L(S^E) \parallel L \neq \emptyset \qquad \text{and } v' \geq v, \qquad \text{then } v' \parallel L(S^E) \parallel L \neq \emptyset \ .$$

Because $v, v' \in P_\tau \cap L(S^E_{\mathcal{D}})$ over alphabet $\Sigma^E_D = \mathcal{G}(r_d, w_d, f_c, w_c)$ and $v' \geq v$ we find that v can be obtained from v' by erasing factors that are necessarily of the form $w_\star(g)\ r_d(g)^*$ or $r_d(g)$. In particular $v, v' \in P_\tau$ shows that $Proj_{\mathcal{G}(f_c)}(v) = Proj_{\mathcal{G}(f_c)}(v') = \tau$.[3] The proof of Lem. 8 is by induction on the number of those factors.

Lemma 9 (Leader Monotonicity Lemma). *Let $\tau \in \Upsilon$ and $L \subseteq \mathcal{G}(r_c, f_c, w_c, u_c)^*$ satisfying: if $\alpha f_c(g)\beta_1\beta_2 \in L$, then $\alpha f_c(g)\beta_1 w_c(g)\beta_2 \in L$. For every cover $\hat{L}_\tau$ of $P_\tau \cap L(\mathcal{DS})$:*

$$(P_\tau \cap L(\mathcal{DS})) \parallel L(S^E) \parallel L \neq \emptyset \quad \textit{iff} \quad \hat{L}_\tau \parallel L(S^E) \parallel L \neq \emptyset \ .$$

Proof. ($\Leftarrow$): It follows from $\hat{L}_\tau \subseteq (P_\tau \cap L(\mathcal{DS}))$. ($\Rightarrow$): We conclude from the hypothesis that there exists $w \in P_\tau \cap L(\mathcal{DS})$ such that $w \parallel L(S^E) \parallel L \neq \emptyset$. Since $\hat{L}_\tau$ is a cover $P_\tau \cap L(\mathcal{DS})$, we find that there exists $w' \in \hat{L}_\tau$ such that $w' \geq w$ ans $w' \in P_\tau \cap L(\mathcal{DS})$. Finally, $\mathcal{DS} = \mathcal{D} \parallel S^E_{\mathcal{D}}$ shows that $w, w' \in P_\tau \cap L(S^E_{\mathcal{D}})$, hence that $w' \parallel L(S^E) \parallel L \neq \emptyset$ following Lem. 8, and finally that $\hat{L}_\tau \parallel L(S^E) \parallel L \neq \emptyset$ because $w' \in \hat{L}_\tau$. □

4 Complexity of safety verification of non-atomic networks

Recall that the *safety verification problem* for machines M_D and M_C consists in deciding if the $(\llbracket M_D \rrbracket, \llbracket M_C \rrbracket)$-network is safe. Notice that the size of the input is the size of the machines, and not the size of its LTSs, which might even be infinite. We study the complexity of safety verification for different machine classes.

Given two classes of machines `D`, `C` (like finite-state machines or pushdown machines, see below), we define *the class of* (`D`,`C`)*-networks* as the set $\{(\llbracket D \rrbracket, \llbracket C \rrbracket)\text{-network} \mid D \in \mathtt{D}, C \in \mathtt{C}\}$ and denote by $\mathtt{Safety}(\mathtt{D}, \mathtt{C})$ the restriction of the safety verification problem to pairs of machines $M_D \in \mathtt{D}$ and $M_C \in \mathtt{C}$. We study the complexity of the problem when leader and contributors are *finite-state machines* (FSM) and *pushdown machines* (PDM).[4] In this paper a FSM is just another name for a finite-state LTS, and the LTS $\llbracket A \rrbracket$ of a FSM A is A, i.e. $\llbracket A \rrbracket = A$. We define the size $|A|$ of a FSM A as the size of its transition relation. A (read/write) *pushdown machine* is a tuple $P = (Q, \mathcal{G}(r, w), \Gamma, \Delta, \gamma_0, q_0)$, where Q is a finite set of *states* including the *initial state* q_0, Γ is a *stack alphabet* that contains the *initial stack symbol* γ_0, and $\Delta \subseteq (Q \times \Gamma) \times (\mathcal{G}(r, w) \cup \{\varepsilon\}) \times (Q \times \Gamma^*)$ is a set of *rules*. A *configuration* of a PDM P is a pair $(q, y) \in Q \times \Gamma^*$. The LTS $\llbracket P \rrbracket$ over $\mathcal{G}(r, w)$ associated to P has $Q \times \Gamma^*$ as states, (q_0, γ_0) as initial state, and a transition $(q, \gamma y) \xrightarrow{a} (q', y'y)$ iff $(q, \gamma, a, q', y') \in \Delta$. Define the size of a rule $(q, \gamma, a, q', y') \in \Delta$ as $|y'| + 5$ and the size $|P|$ of a PDM as the sum of the size of rules in Δ.

Determinism. We show that lower bounds (hardness) for the safety verification problems can be achieved already for *deterministic* machines. An LTS $\mathcal{T}$ over a read-write

[3] $Proj_{\Sigma'}(w)$ returns the projection of w onto alphabet Σ'.

[4] We also define FSA and PDA as the automaton (i.e. language acceptor) counterpart of FSM and PDM, respectively. As expected, definitions are identical except for an additional accepting component given by a subset of states in which the automaton accepts.

alphabet is *deterministic* if for every state s and every pair of transitions $s \xrightarrow{a_1} s_1$ and $s \xrightarrow{a_2} s_2$, if $s_1 \neq s_2$ then a_1 and a_2 are reads, and they read different values. Intuitively, for any state of a store $\mathcal{S}$, a deterministic LTS $\mathcal{T}$ can take at most one transition in $\mathcal{S} \parallel \mathcal{T}$. A $(\mathcal{D}, C)$-network is deterministic if $\mathcal{D}$ and C are deterministic LTSs. Given a class $\mathcal{X}$ of machines, we denote by d$\mathcal{X}$ the subclass of machines M of $\mathcal{X}$ such that $[\![M]\!]$ is a deterministic LTS over the read-write alphabet. Notice that this notion does not coincide with the usual definition of a deterministic automaton.

The observation is that a network with non-deterministic processes can be simulated by deterministic ones while preserving safety; intuitively, the inherent non-determinism of interleaving can simulate non-deterministic choice in the machines.

Lemma 10 (Determinization Lemma). *There is a polynomial-time procedure that takes a pair $(\mathcal{D}, C)$ of LTSs and outputs a pair $(\mathcal{D}', C')$ of deterministic LTSs such that the $(\mathcal{D}, C)$-network is safe iff the $(\mathcal{D}', C')$-network is safe.*

We prove the lemma by eliminating non-determinism as follows. Suppose $\mathcal{D}$ is non-deterministic by having transitions $(q, r_d(g), q')$ and $(q, r_d(g), q'')$. To resolve this non-determinism, we define $\mathcal{D}'$ and C' by modifying $\mathcal{D}$ and C as follows: we add new states q_1, q_2, q_3, q_4 to $\mathcal{D}$ and replace the two transitions $(q, r_d(g), q')$ and $(q, r_d(g), q'')$ by the transitions $(q, r_d(g), q_1)$, $(q_1, w_d(\mathbf{nd}), q_2)$, $(q_2, r_d(0), q_3)$, $(q_3, w_d(g), q')$, $(q_2, r_d(1), q_4)$ and $(q_4, w_d(g), q'')$. Let q_0 be the initial state of C. We add two new states $\hat{q}$ and $\tilde{q}$ to C and the transitions $(q_0, r_c(\mathbf{nd}), \hat{q})(\hat{q}, w_c(0), \tilde{q})(\tilde{q}, w_c(1), q_0)$. Finally, we extend the store to accommodate the new values $\{0, 1, \mathbf{nd}\}$. It follows that $\mathcal{D}'$ has one fewer pair of non-deterministic transitions than $\mathcal{D}$. Similar transformations can eliminate other non-deterministic transitions (e.g., two writes from a state) or non-determinism in C.

4.1 Complexity of Safety Verification for FSMs and PDMs

We characterize the complexity of the safety verification problem of non-atomic networks depending on the nature of the leader and the contributors. We show:

`Safety`(dFSM, dFSM), `Safety`(PDM, FSM), `Safety`(FSM, PDM)	coNP-complete
`Safety`(dPDM, dPDM), `Safety`(PDM, PDM)	PSPACE-complete

Theorem 1. `Safety`(dFSM, dFSM) *is* coNP*-hard.*

We show hardness by a reduction from 3SAT to the complement of the safety verification problem. Given a 3SAT formula, we design a non-atomic network in which the leader and contributors first execute a protocol that determines an assignment to all variables, and uses subsets of contributors to store this assignment. For a variable x, the leader writes x to the store, inviting proposals for values. On reading x, contributors non-deterministically write either "x is 0" or "x is 1" on the store, possibly over-writing each other. At a future point, the leader reads the store, reading the proposal that was last written, say "x is 0." The leader then writes "commit x is 0" on the store. Every contributor that reads this commitment moves to a state where it returns 0 every time the value of x is asked for. Contributors that do not read this message are stuck and do not participate further. The commitment to 1 is similar. This protocol ensures that each variable gets assigned a consistent value.

Then, the leader checks that each clause is satisfied by querying the contributors for the values of variables (recall that contributors return consistent values) and checking each clause locally. If all clauses are satisfied, the leader writes a special symbol #. The safety verification problem checks that # is never written, which happens iff the formula is unsatisfiable. Finally, Lemma 10 ensures all processes are deterministic.

Theorem 2. `Safety`(PDM, FSM) *is in* coNP.

Proof. Fix a $(\mathcal{D}, C)$-network $\mathcal{N}$, where P_D is a PDM generating $\mathcal{D} = [\![P_D]\!]$, and C is a FSM. Hence $L(\mathcal{D})$ is a context-free language and $L(C)$ is regular. Prop. 1 and Def. 5 (of $\mathcal{N}^S$) show that the $(\mathcal{D}, C)$-network $\mathcal{N}$ is accepting iff $L(\mathcal{D} \parallel S^E \parallel \sqcup_{g\in\mathcal{G}} S_g) \cap P_{\Upsilon_\#} \neq \emptyset$. Since C is given by a FSM, so is C^E. Further, $L_g = L(C^E) \cap \mathcal{G}(r_d, u_c)^* f_c(g)$ has a support captured by those paths in C^E starting from the initial state and whose label ends by $f_c(g)$ and in which no state is entered more than once. Therefore if C^E has k states then the set of paths starting from the initial state, of length at most $k+1$ and whose label ends with $f_c(g)$ is a support, call it $\underline{L}_g$, of L_g. Next, Lem. 7 shows that deciding $L(\mathcal{D} \parallel S^E \parallel \sqcup_{g\in\mathcal{G}} S_g) \cap P_{\Upsilon_\#} \neq \emptyset$ is equivalent to $L(\mathcal{D} \parallel S^E \parallel \sqcup_{g\in\mathcal{G}} \mathit{Pref}(\underline{L}_g \cdot w_c(g)^*)) \cap P_{\Upsilon_\#} \neq \emptyset$.

Note that this last check does not directly provide a NP algorithm for non-safety because, due to the write records, S^E is exponentially larger than $|\mathcal{G}|$. So, we proceed by pushing down sequences of first writes and obtain the following equivalent statement: $L(\mathcal{D}) \parallel (L(S^E) \cap P_{\Upsilon_\#}) \parallel (\sqcup_{g\in\mathcal{G}} L(\mathit{Pref}(\underline{L}_g \cdot w_c(g)^*)) \cap P_{\Upsilon_\#}) \neq \emptyset$.

Now, we get an NP algorithm as follows: (*a*) guess $\tau \in \Upsilon_\#$ (this can be done in time polynomial in $|\mathcal{G}|$); (*b*) construct in polynomial time a FSA A_1 for $L(S^E) \cap P_\tau$ (A_1 results from S^E by keeping the $|\tau|$ write records corresponding to τ); (*c*) for each $g \in \tau$, guess $z_g \in \underline{L}_g$ (the guess can be done in polynomial time); (*d*) guess $z \in (\sqcup_{g\in\mathcal{G}} z_g) \cap P_\tau$ (this fixes a sequence of reads, useless writes and first writes of the contributors according to τ); (*e*) construct in polynomial time a FSA A_2 such that $L(A_2)$ is the least language containing z and if $\alpha f_c(g) \beta_1 \beta_2 \in L(A_2)$ then $\alpha f_c(g) \beta_1 w_c(g) \beta_2 \in L(A_2)$ (intuitively we add selfloops with write actions of $\mathcal{G}(w_c)$ to the FSA accepting z such that $w_c(g)$ occurs provided $f_c(g)$ has previously occurred); (*f*) construct in time polynomial in $|P_D|$ a context-free grammar (CFG) G_D such that $L(G_D) = L(P_D)$; (*g*) construct in polynomial time a CFG G such that $L(G) = L(G_D) \parallel L(A_1) \parallel L(A_2)$ (this can be done in time polynomial in $|G_D| + |A_1| + |A_2|$ as stated in Prop. 2, Sect. E); (*h*) check in polynomial time whether $L(G) \neq \emptyset$. □

We continue with the following results showing that even if all processes but the leader are given a stack then the safety verification problem remains in coNP. A detailed proof is given in Appendix C.

Theorem 3. `Safety`(FSM, PDM) *is in* coNP.

The complexity of the problem becomes higher when all the processes are PDMs.

Theorem 4. `Safety`(dPDM, dPDM) *is* PSPACE-*hard.*

PSPACE-hardness is shown by reduction from the acceptance problem of a polynomial-space deterministic Turing machine. The proof is technical. The leader and contributors simulate steps of the Turing machine in rounds. The stack is used to store configurations

of the Turing machine. In each round, the leader sends the current configuration of the Turing machine to contributors by writing the configuration one element at a time on to the store and waiting for an acknowledgement from some contributor that the element was received. The contributors receive the current configuration and store the next configuration on their stacks. In the second part of the round, the contributors send back the configuration to the leader. The leader and contributors use their finite state to make sure all elements of the configuration are sent and received.

Additionally, the leader and the contributors use the stack to count to 2^n steps. If both the leader and some contributor count to 2^n in a computation, the construction ensures that the Turing machine has been correctly simulated for 2^n steps, and the simulation is successful. The counting eliminates bad computation sequences in which contributors conflate the configurations from different steps due to asynchronous reads and writes.

Next we sketch the upper PSPACE bound that uses constructions on approximations of context-free languages. The details of the proof are available in Appendix E.

Theorem 5. `Safety`(PDM, PDM) *is in* PSPACE.

Proof. Let P_D and P_C be PDMs respectively generating $\mathcal{D} = [\![P_D]\!]$ and $C = [\![P_C]\!]$, hence $L(\mathcal{D})$ and $L(C)$ are context-free languages. Proposition 1 shows that the $(\mathcal{D}, C)$-network $\mathcal{N}$ is accepting iff $L(\mathcal{N}^S) \cap P_{\Upsilon_\#} \neq \emptyset$ iff $L(\mathcal{DS} \parallel S^E \parallel \sqcup_{g\in\mathcal{G}} S_g) \cap P_{\Upsilon_\#} \neq \emptyset$ (by (1)). From the construction of the Simulation Lemma, for each $g \in \mathcal{G}$ the language $L_g = L(C^E) \cap \mathcal{G}(r_d, u_c)^* f_c(g)$ is context-free, and so is $L(S_g)$. Given P_C we compute in polynomial time a PDA P_g such that $L(P_g) = L_g$. Next,

$$
\begin{aligned}
& L(\mathcal{DS} \parallel S^E \parallel \sqcup_{g\in\mathcal{G}} S_g) \cap P_{\Upsilon_\#} \neq \emptyset \\
\text{iff } & (L(\mathcal{DS}) \cap P_{\Upsilon_\#}) \parallel L(S^E) \parallel \sqcup_{g\in\mathcal{G}} L(S_g) \neq \emptyset \\
\text{iff } & (L(\mathcal{DS}) \cap P_{\Upsilon_\#}) \parallel L(S^E) \parallel \sqcup_{g\in\mathcal{G}} \mathit{Pref}(L(P_g) \cdot w_c(g)^*) \neq \emptyset \qquad (2) \\
\text{iff } & (\textstyle\bigcup_{\tau\in\Upsilon_\#} \hat{L}_\tau) \parallel L(S^E) \parallel \sqcup_{g\in\mathcal{G}} \mathit{Pref}(\overline{L(P_g)} \cdot w_c(g)^*) \neq \emptyset \qquad (3)
\end{aligned}
$$

(2) follows from definition of S_g and $L_g = L(\overline{P_g})$; (3) follows from Lem. 6 and by letting $\hat{L}_\tau$ and $\overline{L(P_g)}$ be a cover and support of $L(\mathcal{DS}) \cap P_\tau$ and $L(P_g)$, respectively.

Next, for all $g \in \mathcal{G}$ we compute a FSA A_g such that $L(A_g)$ is a support of $L(P_g)$. Our first language-theoretic construction shows that the FSA A_g can be computed in time exponential but space polynomial in $|P_g|$. Then, because $L(S^E)$ is a regular language, we compute in space polynomial in $|P_D| + |P_C|$ a FSA A_C such that $L(A_C) = L(S^E) \parallel \sqcup_{g\in\mathcal{G}} \mathit{Pref}(L(A_g) \cdot w_c(g)^*)$. Hence, by (3) and because of $\Upsilon_\#$ (guessing and checking $\tau \in \Upsilon_\#$ is done in time polynomial in $|\mathcal{G}|$) we find that it suffices to prove $\hat{L}_\tau \parallel L(A_C) \neq \emptyset$ is decidable in space polynomial in $|P_D| + |P_C|$.

To compute a cover $\hat{L}_\tau$ of $L(\mathcal{DS}) \cap P_\tau$, we need results about the k-index approximations of a context-free language [4]. Given a CFG G in CNF and $k \geq 1$, we define the *k-index approximation* of $L(G)$, denoted by $L^{(k)}(G)$, consisting of the words of $L(G)$ having a derivation in which every intermediate word contains at most k occurrences of variables. We further introduce an operator $\bowtie$ which, given G and FSA A, computes in polynomial time a context-free grammar $G \bowtie A$ such that $L(G \bowtie A) = L(G) \parallel L(A)$. We prove the following properties:

1. $L^{(3m)}(G)$ is a cover of $L(G)$, where m is the number of variables of G;

2. for every FSA A and $k \geq 1$, $L^{(k)}(G \bowtie A) = L^{(k)}(G) \parallel L(A)$;
3. $L^{(k)}(G) \neq \emptyset$ on input G, k can be decided in NSPACE($k \log(|G|)$).

Equipped with these results, the proof proceeds as follows. Let G_D be a context-free grammar such that $L(G_D) = L(P_D)$. It is well-known that G_D can be computed in time polynomial in $|P_D|$. Next, given τ, we compute a grammar G_D^τ recognizing $P_\tau \cap L(\mathcal{DS})$ as follows. The definition of $\mathcal{DS}$ shows that $P_\tau \cap L(\mathcal{DS}) = L(\mathcal{D}) \parallel (L(S^E_{\mathcal{D}}) \cap P_\tau)$. We then compute a FSA $S^\tau_{\mathcal{D}}$ such that $L(S^\tau_{\mathcal{D}}) = L(S^E_{\mathcal{D}}) \cap P_\tau$. It can be done in time polynomial in $|P_D| + |P_C|$ because it is a restriction of S^E where write records are totally ordered according to τ and there are exactly $|\tau|$ of them. Therefore we obtain, $P_\tau \cap L(\mathcal{DS}) = L(G_D) \parallel L(S^\tau_{\mathcal{D}})$ because $L(G_D) = L(\mathcal{D})$. Define G_D^τ as the CFG $G_D \bowtie S^\tau_{\mathcal{D}}$ which can be computed in polynomial time in G_D and $S^\tau_{\mathcal{D}}$, hence in $|P_D| + |P_C|$. Clearly $L(G_D^\tau) = P_\tau \cap L(\mathcal{DS})$. Further, $L^{(k)}(G_D^\tau)$ is a cover of $L(G_D^\tau)$ for some $k \leq p(|P_D|)$, where p is a suitable polynomial.

By item 2, $L^{(k)}(G_D^\tau) \parallel L(A_C) = L^{(k)}(G_D^\tau \bowtie A_C)$, where the grammar $G_D^\tau \bowtie A_C$ can be constructed in exponential time and space polynomial in $|P_D| + |P_C|$. Now we apply a generic result of complexity (see e.g. Lemma 4.17, [3]), slightly adapted: given functions $f_1, f_2 \colon \Sigma^* \to \Sigma^*$ and $g \colon \Sigma^* \times \Sigma^* \to \Sigma^*$ if f_i can be computed by a s_{f_i}-space-bounded Turing machine, and g can be computed by a $s_{g_1}(|x_1|) \cdot s_{g_2}(|x_2|)$-space-bounded Turing machine, then $g(f_1(x), f_2(x))$ can be computed in $\log(|f_1(x)| + |f_2(x)|) + s_{f_1}(|x|) + s_{f_2}(|x|) + s_{g_1}(|f_1(x)|) \cdot s_{g_2}(|f_2(x)|))$ space. We have

- f_1 is the function that computes $G_D^\tau \bowtie A_C$ on input (P_D, P_C), and f_2 is the function that on input P_D computes $3m$, where m is the number of variables of G_D^τ. So the output size of f_1 is exponential in the input size, while it is polynomial for f_2. Moreover, s_{f_i} for $i = 1, 2$ is polynomial.
- g is the function that on input $(G_D^\tau \bowtie A_C, 3m)$ yields 1 if $L^{(3m)}(G_D^\tau \bowtie A_C) \neq \emptyset$, and 0 otherwise, where m is the number of variables of G_D^τ. By (3) s_{g_1} is logarithmic, and s_{g_2} is linear.

Finally, the generic complexity result shows that $g \circ f$ can be computed in space polynomial in $|P_D| + |P_C|$, and we are done. □

We note that our three language-theoretic constructions (the construction of automaton A_g that is a cover of $L(P_g)$ of size at most exponential in $|P_g|$, and results 1, 2, and 3 in the proof above) improve upon previous constructions, and are all required for the optimal upper bound. Hague [18] shows an alternate doubly exponential construction using a result of Ehrenfeucht and Rozenberg in place of Theorem 7. This gave a 2EXPTIME algorithm. Even after using our exponential time construction for A_g, we can only get an EXPTIME algorithm, since the non-emptiness problem for (general) context-free languages is P-complete [19]. Our bounded-index approximation for the cover and the space-efficient emptiness algorithm for bounded-index languages are crucial to the PSPACE upper bound.

4.2 The bounded safety problem

Given $k > 0$, we say that a $(\mathcal{D}, C)$-network is *k-safe* if all traces in which the leader and each contributor make at most k steps are safe; i.e., we put a bound of k steps on the

runtime of each contributor, and consider only safety within this bound. Here, a step consists of a read or a write of the shared register. The bound does not limit the total length of traces, because the number of contributors is unbounded. The *bounded safety* problem asks, given $\mathcal{D}$, C, and k written in unary, if the $(\mathcal{D}, C)$-network is k-safe.

Given a class of (D,C)-networks, we define BoundedSafety(D, C) as the restriction of the k-safety problem to pairs of machines $M_D \in \mathtt{D}$ and $M_C \in \mathtt{C}$, where we write k in unary. A closer look to Theorem 1 shows that its proof reduces the satisfiability problem for a formula ϕ to the bounded safety problem for a (D,C)-network and a number k, all of which have polynomial size $|\phi|$. This proves that BoundedSafety(dFSM, dFSM) is coNP-hard. We show that, surprisingly, bounded safety remains coNP-complete for pushdown systems, and, even further, for arbitrary Turing machines. Notice that the problem is already coNP-complete for one single Turing machine.

We sketch the definition of the Turing machine model (TM), which differs slightly from the usual one. Our Turing machines have two kind of transitions: the usual transitions that read and modify the contents of the work tape, and additional transitions with labels in $\mathcal{G}(r, w)$ for communication with the store. The machines are input-free, i.e., the input tape is always initially empty.

Theorem 6. BoundedSafety(TM, TM) *is* coNP*-complete.*

Proof. Co-NP-hardness follows from Theorem 1. To prove BoundedSafety(TM, TM) is in NP we use the simulation lemma. Let M_D, M_C, k be an instance of the problem, where M_D, M_C are Turing machines of sizes n_D, n_C with LTSs $\mathcal{D} = [\![M_D]\!]$ and $C = [\![M_C]\!]$, and let $n_D + n_C = n$. In particular, we can assume $|\mathcal{G}| \leq n$, because we only need to consider actions that appear in M_D and M_C. If the $(\mathcal{D}, C)$-network is not k-safe, then by definition there exist $u \in L(\mathcal{D})$ and a multiset $M = \{v_1, \ldots, v_k\}$ over $L(C^E)$ such that u is compatible with M following some $\tau \in \Upsilon_\#$; moreover, all of $u, v_1, \ldots, v_m$ have length at most k. By Cor. 1 and Lem. 1 (showing we can drop traces without a first or regular write), there exists a set $S = \{s_{g_1}, \ldots, s_{g_m}\}$ with $m \leq |\mathcal{G}| \leq n$, where $s_{g_i} \in L_{g_i} \cdot w_c(g_i)^*$, and numbers $i_1, \ldots, i_m$ such that u is compatible with $\{s_{g_1} w_c(g_1)^{i_1}, \ldots, s_{g_m} w_c(g_m)^{i_m}\}$ following τ. Since each of the s_{g_i} is obtained by suitably renaming the actions of a trace, we have $|s_{g_i}| \leq k$. Moreover, since the $w_c(g_j)^{i_j}$ parts provide the writes necessary to execute the reads of the s_g sequences, and there are at most $k \cdot (m + 1) \leq k \cdot (n + 1)$ of them, the numbers can be chosen so that $i_1, \ldots, i_m \leq O(n \cdot k)$ holds.

We present a nondeterministic polynomial algorithm that decides if the $(\mathcal{D}, C)$-network is k-unsafe. The algorithm guesses $\tau \in \Upsilon_\#$ and traces $u, s_{g_1}, \ldots, s_{g_m}$ of length at most k. Since there are at most $n + 1$ of those traces, this can be done in polynomial time. Then, the algorithm guesses numbers $i_1, \ldots, i_m$. Since the numbers can be chosen so that $i_1, \ldots, i_m \leq O(n \cdot k)$, this can also be done in polynomial time. Finally, the algorithm guesses an interleaving of $u, s_{g_1} w_c(g_1)^{i_1}, \ldots, s_{g_m} w_c(g_m)^{i_m}$ and checks compatibility following τ. This can be done in $O(n^2 \cdot k)$ time. If the algorithm succeeds, then there is a witness that $(L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup\!\sqcup_{g \in \mathcal{G}} L(S_g)) \cap P_\tau \neq \emptyset$ holds, which shows, by Prop. 1 and Def. 5 (of $\mathcal{N}^S$) that the $(\mathcal{D}, C)$-network is unsafe. □

A TM is poly-time if it takes at most $p(n)$ steps for some polynomial p, where n is the size of (the description of) the machine in some encoding. As a corollary, we get that the safety verification problem when leaders and contributors are poly-time Turing

machines is coNP-complete. Note that the coNP upper bound holds even though the LTS corresponding to a poly-time TM is exponentially larger than its encoding.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS '96. pp. 313–321. IEEE Computer Society (1996)
2. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Information Processing Letters 22(6), 307 – 309 (1986)
3. Arora, S., Barak, B.: Computational Complexity–A Modern Approach. CUP (2009)
4. Brainerd, B.: An analog of a theorem about context-free languages. Information and Control 11(56), 561 – 567 (1967)
5. Clarke, E.M., Talupur, M., Veith, H.: Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In: TACAS '08. LNCS, vol. 4963, pp. 33–47. Springer (2008)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press and McGraw-Hill (1990)
7. Delzanno, G.: Constraint-based verification of parameterized cache coherence protocols. Formal Methods in System Design 23(3), 257–301 (2003)
8. Delzanno, G., Raskin, J.F., Van Begin, L.: Towards the automated verification of multi-threaded java programs. In: TACAS '02. LNCS, vol. 2280, pp. 173–187. Springer (2002)
9. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: CONCUR '10. LNCS, vol. 6269, pp. 313–327. Springer (2010)
10. Dimitrova, R., Podelski, A.: Is lazy abstraction a decision procedure for broadcast protocols? In: VMCAI '08. LNCS, vol. 4905, pp. 98–111. Springer (2008)
11. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: CHARME '03. pp. 247–262. LNCS, Springer (2003)
12. Emerson, E.A., Namjoshi, K.S.: Verification of parameterized bus arbitration protocol. In: CAV '98. LNCS, vol. 1427, pp. 452–463. Springer (1998)
13. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS '99. pp. 352–359. IEEE Computer Society (1999)
14. Esparza, J., Ganty, P., Kiefer, S., Luttenberger, M.: Parikh's theorem: A simple and direct automaton construction. Information Processing Letters 111, 614–619 (2011)
15. Finkel, A., Leroux, J.: How to compose Presburger-accelerations: Applications to broadcast protocols. In: FSTTCS '02. LNCS, vol. 2556, pp. 145–156. Springer (2002)
16. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. 34(1), 6:1–6:48 (2012)
17. Guerraoui, R., Ruppert, E.: Anonymous and fault-tolerant shared-memory computing. Distributed Computing 20(3), 165–177 (2007)
18. Hague, M.: Parameterised pushdown systems with non-atomic writes. In: Proc. of FSTTCS '11. LIPIcs, vol. 13, pp. 457–468. Schloss Dagstuhl (2011)
19. Jones, N.D., Laaser, W.T.: Complete problems for deterministic polynomial time. In: Proc. of STOC '74. pp. 40–46. ACM (1974)
20. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: CAV '10. LNCS, vol. 6174. Springer (2010)
21. Laurendeau, C., Barbeau, M.: Secure anonymous broadcasting in vehicular networks. In: LCN '07. pp. 661–668. IEEE Computer Society (2007)
22. McMillan, K.L.: Verification of an implementation of tomasulo's algorithm by compositional model checking. In: CAV '98. LNCS, vol. 1427, pp. 110–121. Springer (1998)

23. Sipser, M.: Introduction to the Theory of Computation. Int'l Thomson Publishing (1996)
24. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: CAV '10. LNCS, vol. 6174, pp. 629–644. Springer (2010)
25. Viswanathan, M., Chadha, R.: Deciding branching time properties for asynchronous programs. Theoretical Computer Science 410(42), 4169–4179 (2009)

A Combinatorics

Proof (of Lem 8). Because $v, v' \in P_\tau \cap L(S^E_{\mathcal{D}})$ over alphabet $\Sigma^E_D = \mathcal{G}(r_d, w_d, f_c, w_c)$ and $v' \geq v$ we find that v can be obtained from v' by erasing factors that are necessarily of the form $w_\star(g)\ r_d(g)^*$ or $r_d(g)$. In particular $v, v' \in P_\tau$ shows that $Proj_{\mathcal{G}(f_c)}(v) = Proj_{\mathcal{G}(f_c)}(v') = \tau$.[5]

The proof is by induction on the number m of those factors. If $m = 0$ then $v' = v$ and we are done. Now, let $m > 0$ and let $v_\dagger$ be the trace which results from erasing one factor $\sigma \in (\mathcal{G}(w_d, w_c)\mathcal{G}(r_d)^*) \cup (\mathcal{G}(r_d))$ from v'. That is $v' = v^1_\dagger \sigma v^2_\dagger$ where $v^1_\dagger v^2_\dagger = v_\dagger \geq v$. Without loss of generality we can assume that if $\sigma \notin \mathcal{G}(r_d)$ then the first symbol of $v^2_\dagger$ is a write action (it belongs to $\mathcal{G}(f_c, w_c, w_d)$). Also observe that by $S^E_{\mathcal{D}}$, if $\sigma \in \mathcal{G}(r_d)$ then the last write in $v^1_\dagger$ writes the value needed by σ.

Since $v' \in P_\tau \cap L(S^E_{\mathcal{D}})$, it is routine to check that $v_\dagger \in P_\tau \cap L(S^E_{\mathcal{D}})$. Therefore we conclude from the induction hypothesis and $v_\dagger \geq v$ that there exists a trace $w_\dagger \in L(S^E) \parallel L$ such that $v_\dagger \parallel w_\dagger \neq \emptyset$, equivalently that $Proj_{\Sigma^E_D}(w_\dagger) = v_\dagger$ since $\Sigma^E_D \subseteq \Sigma_E$. Observe that $w_\dagger$ can be divided into $w^1_\dagger w^2_\dagger$ such that $Proj_{\Sigma^E_D}(w^i_\dagger) = v^i_\dagger$ for $i = 1, 2$. Let $w^1_\ddagger$ be the (possibly empty) suffix of $w^1_\dagger$ starting at the last occurrence of an action of Σ^E_D and let $w^2_\ddagger$ be the prefix of $w^2_\dagger$ which ends at the first occurrence of an action of Σ^E_D. Then $L(S^E)$ shows that $w^1_\ddagger w^2_\ddagger$ belongs to $\mathcal{G}(r_c)^*\mathcal{G}(u_c)^*$.

Let us now consider the added factor σ. First, let us notice that $Proj_{\Sigma^E_D}(w^1_\dagger \sigma w^2_\dagger) = v^1_\dagger \sigma v^2_\dagger = v'$. Thus it suffices to show that $w^1_\dagger \sigma w^2_\dagger \in L(S^E) \parallel L$.

Now, if $\sigma \in \mathcal{G}(r_d)$, we can choose $w^1_\dagger$ and $w^2_\dagger$ such that $w^1_\ddagger = \varepsilon$. Notice that as for $v^1_\dagger$ and $v^2_\dagger$, then the last write that occurs in $w^1_\dagger$ writes the value needed by σ and so $w^1_\dagger \sigma w^2_\dagger \in L(S^E)$. Hence we find that $w^1_\dagger \sigma w^2_\dagger \in L(S^E) \parallel L$ because the alphabet of L is disjoint from $\mathcal{G}(r_d)$.

On the other hand if $\sigma \notin \mathcal{G}(r_d)$ we can choose $w^1_\dagger$ and $w^2_\dagger$ such that $w^2_\ddagger = \varepsilon$. Then $\sigma = w_\star(g) r_d(g)^i$ for some $i \in \mathbb{N}$. Also $w^1_\dagger \sigma w^2_\dagger \in L(S^E)$ because, as assumed above, the first symbol of $w^2_\dagger$ is a write action.

Finally, if $\sigma = w_d(g) r_d(g)^i$ we find that $w^1_\dagger \sigma w^2_\dagger \in L(S^E) \parallel L$ because the alphabet of L is disjoint from $\mathcal{G}(w_d, r_d)$. Else if $\sigma = w_c(g) r_d(g)^i$ we find, by S^E, that $f_c(g)$ must occur in $w^1_\dagger$, hence that $w_c(g)$ can be matched in L following the hypothesis on L. □

B coNP lower bound

Proof (of Theorem 1). We give a reduction from 3SAT to the complement of the safety verification problem. Given a 3SAT formula with n variables and m clauses, we con-

[5] $Proj_{\Sigma'}(w)$ returns the projection of w onto alphabet Σ'.

struct a deterministic leader $\mathcal{D}$ and a deterministic contributor C such that the $(\mathcal{D}, C)$-network writes a special symbol # iff the formula is satisfiable. The leader uses the contributors to guess and store an assignment, and then checks if each clause is satisfied.

Gadget to guess and retrieve a bit. The reduction uses the following protocol between the leader and the contributors to guess a bit and maintain the guess consistently. To assign a value to `bit`, the leader writes `bit` to the global store. The contributors who read `bit` from the store then write consecutively `propose-bit-is-`i, $i = 0, 1$, on the store. The leader reads the store at some (non-deterministic) point, and reads the last write by one of the contributors proposing either 0 or 1. If it reads `propose-bit-is-0` (the 1 case is identical), it writes back that it commits to setting the bit to 0 (writing `commit-bit-is-0`). Contributors who read `commit-bit-is-0` move on to the next phase, where they deliver `bit-is-0` each time they are asked the value of `bit`. That is, they wait to read a `get-value-of-bit` message, and reply with `bit-is-0`.

Similarly, if the leader commits to a 1, contributors who read the message come to the consensus that the bit is 1. Contributors who miss the commit message are stuck. This protocol ensures that the leader and contributors can reach consensus on the value of a bit, and even though they are deterministic, the value of the bit is chosen non-deterministically, based on when the leader reads a value (`propose-bit-is-`i, $i = 0, 1$) from the store. Notice that an arbitrary number of contributors can participate and potentially overwrite each other, but the bit is fixed to a chosen value. *Reduction from 3SAT.* The leader uses the above protocol to "assign" non-deterministically chosen consensus values to each variable $x_1, \ldots, x_n$. Then, it checks sequentially that each clause is satisfied by this assignment. To do this, it gets the literals from the contributors and checks if the clause is satisfied. To get the assigned value to a variable x, the leader writes `get-value-of-`x on the store. The contributors that are storing an assignment to x (i.e., those who completed the consensus protocol for x) and who read this message, write the consensus value (using values x`-is-0` or x`-is-1`). Even if several contributors write, they write the same value.

Suppose the formula is satisfiable. Then, there is an execution of the protocol where the contributors reach a consensus for each bit corresponding to a satisfying assignment, and the leader succeeds in checking all clauses. Then, the value # gets written to the store and the $(\mathcal{D}, C)$-network is accepting. On the other hand, if the formula is not satisfiable, then the leader never succeeds checking all clauses and # never gets written. Note that the size of $\mathcal{D}$ and C is $O(m + n)$ and that they are deterministic. □

C coNP upper bound

We first recall some basic structural property of FSAs. Define the reachability relation of a FSA as follows: state s_1 is reachable from s_2 iff there is a (possibly empty) path from s_2 to s_1. We define a strongly connected component (`scc`) of a FSA as an equivalence class for the mutual reachability relation.

Lemma 11. *For every FSA $A = (\Sigma, Q, \delta, q_0, F)$ there exists a finite collection $A_1, \ldots, A_d$ such that each A_i satisfies the following properties:*

1. *each A_i results from removing states and transitions from A; and*
2. *$\bigcup_{i=1}^{d} L(A_i) = L(A)$; and*
3. *for each $v_1, v_2 \in L(A_i)$ there exists $v \in L(A_i)$ such that $v \succeq v_1$ and $v \succeq v_2$.*

Proof. A path in a FSA is said to be *accepting* if its first and last state are initial and final respectively. Let π be an accepting path in A such that no state repeats in π. Define A_π to be the FSA that consists exactly of (*1*) the states and transitions of π; and (*2*) the states and transitions of the `sccs` visited by π.

Clearly, A_π results from removing states and transitions from A. Define the set $\{A_1, \ldots, A_d\}$ such that each accepting path π with no repeating state induces exactly one automaton $A_\pi \in \{A_1, \ldots, A_d\}$. This set is finite since there are only finitely many states and transitions in A. Furthermore, it is easily checked that $\bigcup_{i=1}^{d} L(A_i) = L(A)$.

We turn to point 3. Because no state is repeated, π fixes a total order on the `sccs` it visits (and are also included in A_π). So any two accepting paths in A_π must visit the `sccs` in that order. Therefore, it suffices to show that given a `scc`, any two traces drawn upon that `scc` are covered by a third one. This is easily seen from the definition of subword ordering and the fact that in a `scc` all states are mutually reachable. □

Next, let $g \in \mathcal{G}$ and define $LL_g = L_g \cdot w_c(g)^*$. Observe that $LL_g \subseteq \mathcal{G}(r_c, u_c)^* f_c(g)\, w_c(g)^*$. Hence, the alphabet of LL_g is given by $\mathcal{G}(r_c, u_c) \cup \{f_c(g), w_c(g)\}$.

Lemma 12. *Let $v \in \mathcal{G}(r_d, w_d, f_c, w_c)^*$, $g \in \mathcal{G}$, $G_1 \subseteq \mathcal{G}$ such that $g \notin G_1$. We have: if $v \parallel L(S^E) \parallel (\sqcup\!\sqcup_{g' \in G_1} LL_{g'}) \neq \emptyset$ and $v \parallel L(S^E) \parallel LL_g \neq \emptyset$ then $v \parallel L(S^E) \parallel (\sqcup\!\sqcup_{g \in G_1 \cup \{g\}} LL_g) \neq \emptyset$.*

Proof. Let $x \in \sqcup\!\sqcup_{g' \in G_1} LL_{g'}$ and $y \in LL_g$. We prove: if $v \parallel L(S^E) \parallel x \neq \emptyset$ and $v \parallel L(S^E) \parallel y \neq \emptyset$ then $v \parallel L(S^E) \parallel (x \sqcup\!\sqcup y) \neq \emptyset$.

Let $v = v_0 \ldots v_{n-1}$, where $v_i \in \mathcal{G}(r_d, w_d, f_c, w_c)$ for each i, $0 \leq i < n$, and consider the asynchronous product $v \parallel L(S^E)$. We define the LTS Q over alphabet Σ_E as follows. The states of Q are the set $\{(V_i, g) \mid 0 \leq i \leq n,\ g \in \mathcal{G} \cup \{\$\}\}$. The initial state is $(V_0, \$)$. There is a transition $(V_i, g) \xrightarrow{a} (V_j, g')$ between two states of Q iff one of the following condition holds:

- $a = r_d(g) = v_i$, $j = i + 1$, $g' = g$;
- $a = r_c(g)$, $j = i$, $g' = g$;
- $a = w_d(g') = v_i$, $j = i + 1$;
- $a = f_c(g') = v_i$, $j = i + 1$;
- $a = w_c(g') = v_i$, $j = i + 1$;
- $a = u_c$, $j = i$, $g' = \$$.

It is easy to see that $L(Q)$ is the prefix closure of the language $v \parallel L(S^E)$.

Since $v \parallel L(S^E) \parallel x \neq \emptyset$ and $v \parallel L(S^E) \parallel y \neq \emptyset$ by hypothesis, there exist two words σ_x and σ_y of Q such that $\sigma_x \parallel x \neq \emptyset$ and $\sigma_y \parallel y \neq \emptyset$. Let σ'_x be the word resulting from erasing all symbols in $\mathcal{G}(u_c, r_c)$ from σ_x, and define σ'_y similarly. It can be easily checked that $\sigma'_x = \sigma'_y = v$. Therefore, there exist $\alpha_i, \beta_i \in \mathcal{G}(u_c, r_c)^*$ for every i, $0 \leq i \leq n$ such that

$$\sigma_x = \alpha_0 v_0 \alpha_1 v_1 \ldots \alpha_{n-1} v_{n-1} \alpha_n$$
$$\sigma_y = \beta_0 v_0 \beta_1 v_1 \ldots \beta_{n-1} v_{n-1} \beta_n$$

The definition of Q further shows that if v_{i-1} is a read or write to g then $\alpha_i = r_c(g)^{i_1} u_c^{i_2}$ for some positive integer i_1, i_2 and similarly $\beta_i = r_c(g)^{j_1} u_c^{j_2}$ for some positive integer j_1, j_2. Moreover, if $i_2 + j_2 > 0$ then v_i must be a write. Define for each $0 \leq i \leq n$ the word $\gamma_i = r_c(g)^{i_1+j_1} u_c^{i_2+j_2}$, and let $\sigma_{sh} = \gamma_0 v_0 \dots \gamma_{n-1} v_{n-1} \gamma_n$. It is routine to check that $\sigma_{sh} \in L(Q)$ holds, and so $L(Q) \parallel \sigma_{sh} \neq \emptyset$. Let σ'_{sh} be the result of erasing from σ_{sh} all symbols of $\mathcal{G}(r_d, w_d)$ (which necessarily correspond to symbols of v). Again we find that $L(Q) \parallel \sigma'_{sh} \neq \emptyset$. Finally, by the definition of γ_i and since $g \notin G_1$ we get $\sigma'_{sh} \in (x \sqcup\!\sqcup y)$, and so $L(Q) \parallel (x \sqcup\!\sqcup y) \neq \emptyset$. By the definition of Q this implies $v \parallel L(S^E) \parallel (x \sqcup\!\sqcup y) \neq \emptyset$, and we are done. □

Proof (of Thm. 3).

1. guess $\tau = g_1 \dots g_d \in \Upsilon_{\#}$, in particular $g_d = \#$;
2. compute a FSA Q_τ such that $L(Q_\tau) = P_\tau \cap L(\mathcal{DS})$;
3. compute the sccs of Q_τ;
4. guess an accepting path π_τ of Q_τ where no state repeats;
5. compute the FSA q_τ consisting exactly of the states and transitions of Q_τ visited by π_τ and the states and transitions of the sccs of Q_τ visited by π_τ;
6. check whether $L(q_\tau) \parallel L(S^E) \parallel LL_{g_i} \neq \emptyset$ for each i, $1 \leq i \leq d$.

Non-deterministic polynomial time. The two guesses are of polynomial size, the first in the size of $\mathcal{G}$, the second in the size of the FSA Q_τ which can be computed in time polynomial in the size of $\mathcal{G}$ and $\mathcal{D}$. Computing the sccs is done in time linear in the size of Q_τ using Tarjan's algorithm [6]. Each of the d checks can be made in polynomial time. To see this, we first notice that we can safely replace $L(S^E)$ by $L(S^E) \cap P_\tau$ for which an FSM can be computed in polynomial time. Therefore it is clear that each check can be made in polynomial time.

Correctness. We want to show that

$$L(q_\tau) \parallel L(S^E) \parallel (\sqcup\!\sqcup_{g \in \mathcal{G}} L(S_g)) \neq \emptyset \text{ iff } L(q_\tau) \parallel L(S^E) \parallel LL_{g_i} \neq \emptyset \text{ for each } i, 1 \leq i \leq d \ .$$

For the only if direction it suffices to notice that by Lem. 1 and definition of P_τ we can restrict the shuffle to those values occurring in τ only. Hence, the definition of $L(S_g)$ and the shuffle operator show the rest.

For the if direction, let $v_1, \dots, v_d$ be the d words from $L(q_\tau)$ such that $v_i \parallel L(S^E) \parallel LL_{g_i} \neq \emptyset$ for each i, $1 \leq i \leq d$. By repeatedly applying Lemma 11 point 3 we find there exists $v \in L(q_\tau)$ such that $v \geq v_i$ for every i. Furthermore, Lemma 8 shows that $v \parallel L(S^E) \parallel LL_{g_i} \neq \emptyset$ for each i.

Finally, we conclude from repeated applications of Lemma 12 that $v \parallel L(S^E) \parallel (\sqcup\!\sqcup_{i=1}^{d} LL_{g_i}) \neq \emptyset$, hence that $L(q_\tau) \parallel L(S^E) \parallel (\sqcup\!\sqcup_{g \in \mathcal{G}} L(S_g)) \neq \emptyset$ by definition of $L(S_g) = \mathit{Pref}(LL_g)$ and we are done. □

D PSPACE lower bound

Proof (of Theorem 4). We give a reduction from the acceptance problem of a linear space-bounded deterministic Turing machine to the complement of the safety verification problem. Fix a deterministic TM M that on input of size n uses at most n tape cells and accepts in exactly 2^n steps. We are given an input x and want to check if M accepts

x. An accepting run is a sequence of TM configurations $c_0 \to c_1 \to \ldots \to c_{2^n}$, where c_0 is the initial configuration (the input x is written on the tape, the head points to the leftmost cell, and the TM is in its initial state), there is a transition of the TM from c_i to c_{i+1} for $i = 0, \ldots, 2^n - 1$, and c_{2^n} is accepting. Following the above assumptions, configurations of M can be encoded by words of fixed length.

We define a $(\mathcal{D}, C)$-network that simulates M and such that $\mathcal{D} = [\![P_D]\!]$ and $C = [\![P_C]\!]$ where P_D and P_C are dPDMs. The leader and the contributors co-operatively simulate computations of M using their stack, and also use the stack to count up to 2^n steps. We start by describing the basic gadgets used in the simulation.

Counting to 2^n using n stack symbols. We show how a contributor can use its stack to count down from 2^n. Consider a stack alphabet of with $n + 2$ symbols $\{\mathsf{l}_0, \ldots, \mathsf{l}_n\} \cup \{\$\}$, where \$ is a special bottom-of-stack marker. Given a stack over this alphabet, the contributor PDM, provided the top of the stack is l_i for some $0 \leq i \leq n$, performs a `decrement` operation defined as follows:

1. While the top of the stack is l_i for some $i > 0$, do $\mathtt{pop}(\mathsf{l}_i)$; $\mathtt{push}(\mathsf{l}_{i-1})$; $\mathtt{push}(\mathsf{l}_{i-1})$;
2. $\mathtt{pop}(\mathsf{l}_0)$ and `return`;

Suppose initially the stack contains $\mathsf{l}_n\$$ (the bottom of the stack is to the right). Then, we reach a stack with \$ on the top exactly after popping l_0 2^n times, that is after performing 2^n times the `decrement` operation.

Computing one step of the TM. In the construction, we simulate one step of the M by sending a configuration from the leader to contributors, and then sending back the next configuration from contributors to the leader.

Assume the reverse of a configuration of the Turing machine is stored as a word w of length n in the stack of the leader and the stack of the contributors is empty. We want to ensure that at the end of the protocol, the stack of the leader contains the reversal of a successor configuration of M, and the stack of the contributor is again empty.

We use the following protocol. The leader and contributors use their finite set of control states to count till n. The leader pops one symbol of w at a time from its stack, writes it on to the global store, and waits for an acknowledgment from some contributor that the symbol has been received. Conversely, the contributors read the letters of w one symbol at a time from the global store. Moreover, using its finite state, the contributors compute the successor configuration of the configuration that is received from the leader and store it on to their stacks. Additionally, a contributor sends an acknowledgment for the receipt of each symbol read from the leader.

After n steps of the leader and contributors, the stack of the leader is empty and the stack of the contributor contains w' where w' can be reached from w^R by executing one step in M.

Notice that at the end of this part of the protocol, in spite of asynchronous reads and writes, the leader is certain that all n symbols were received in order, but not necessarily by the same contributor.

The second part of the protocol sends this configuration back from a contributor to the leader. Again, the leader and the contributor use their finite state to count till n. The contributor sends n symbols one at a time to the leader, and waits for an acknowledgement to check that the leader read the same symbol it transferred. After n steps, the leader's stack contains the reverse of w' and the contributor's stack is again empty.

Moreover, the contributor is certain that the entire configuration has been correctly received by the leader.

Notice that even in the presence of non-atomic reads and writes, if the leader and *some* contributor successfully reach the end of the protocol, then the leader and that particular contributor has faithfully simulated one step of the machine. However, we cannot ensure that the same contributor participated in one whole round of the protocol, always reading and writing the latest values and faithfully simulating one step of the Turing machine. For example, it is possible that several contributors, that have simulated the Turing machine for different number of steps, participate in the protocol. The simulation catches these discrepancies by counting, as described below.

The reduction. Initially, all contributors push $\mathsf{l}_n\$$ onto their stacks. The leader pushes $\mathsf{l}_n\$$ onto the stack, and additionally, the reverse of the starting configuration of the Turing machine.

Then, the leader and contributors execute the protocol described above. At the end of the first part of a round, the leader perform a `decrement`. At the end of the second part of a round, the contributor perform a `decrement`.

The network accepts the computation (e.g., by outputting a special symbol #) if (1) both the leader and some contributor count up to 2^n, and (2) at that point, the Turing machine is in an accepting configuration.

Notice that if the leader interacts with the same contributor for 2^n rounds, then both of them will simultaneously reduce the counter on the stack to \$ at the same time, and thus, would have correctly simulated the Turing machine for 2^n steps. So, if the stack encodes an accepting configuration, the Turing machine accepts.

Conversely, if the leader interacts with multiple contributors in different rounds, then there will not be any contributor whose count reaches 2^n simultaneously with the leader. All such computations are not faithful simulations of the Turing machine and none of them therefore lead to accepting the computation.

Finally, we note that in the above reduction, all processes are deterministic machines. □

E Language Theoretic Constructions

We now complete the proof of Theorem 5 by providing the language-theoretic constructions. We assume familiarity with basic formal language theory [23].

A *context-free grammar* (CFG) is a tuple $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ where $\mathcal{X}$ is a finite set of *variables* containing the *axiom* X_0, Σ is an alphabet, $\mathcal{P} \subseteq \mathcal{X} \times (\Sigma \cup \mathcal{X})^*$ is a finite set of *productions* (the production (X, w) may also be noted $X \to w$). The size of a production $X \to w$ is $|w| + 2$. The size $|G|$ of a CFG G is the sum of all the sizes of productions in $\mathcal{P}$. A CFG $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ is in Chomsky normal form (CNF) iff $\mathcal{P} \subseteq (\mathcal{X} \times (\Sigma \cup \mathcal{X}^2)) \cup \{(X_0, \varepsilon)\}$. A CFG can be converted to CNF in time polynomial in its size.

Given two strings $u, v \in (\Sigma \cup \mathcal{X})^*$ we define a *step* relation $u \Rightarrow v$ if there exists a production $(X, w) \in \mathcal{P}$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. A step is further said to be *leftmost* if $y \in \Sigma^*$, that is the production is applied on the leftmost variable of u. We use $\Rightarrow^*$ to denote the reflexive transitive closure of $\Rightarrow$. The

language of G is $L(G) = \{w \in \Sigma^* \mid X_0 \Rightarrow^* w\}$ and we call any sequence of steps from X_0 to $w \in \Sigma^*$ a *derivation*. A derivation is *leftmost* if it is a sequence of leftmost steps.

Given a CFG with relation $\Rightarrow$ between strings, for every $k \geq 1$ we define the subrelation $\overset{(k)}{\Rightarrow}$ of $\Rightarrow$ as follows: $u \overset{(k)}{\Rightarrow} v$ iff $u \Rightarrow v$ and both u and v contain at most k occurrences of variables. We denote by $\overset{(k)}{\Rightarrow}^*$ the reflexive transitive closure of $\overset{(k)}{\Rightarrow}$. The *k-index language* of G is $L^{(k)}(G) = \{w \in \Sigma^* \mid X_0 \overset{(k)}{\Rightarrow}^* w\}$ and we call the sequence of steps from X_0 to $w \in \Sigma^*$ a *k-index derivation*.

The following properties holds: $\overset{(k)}{\Rightarrow} \subseteq \overset{(k+1)}{\Longrightarrow}$ for all $k \geq 1$; if $B \overset{(k-1)}{\Longrightarrow}^* w$ then $BC \overset{(k)}{\Rightarrow}^* wC$; moreover, if $BC \overset{(k)}{\Rightarrow}^* w$, then there exist w_1, w_2 such that $w = w_1 w_2$ and either (i) $B \overset{(k-1)}{\Longrightarrow}^* w_1$, $C \overset{(k)}{\Rightarrow}^* w_2$, or (ii) $C \overset{(k-1)}{\Longrightarrow}^* w_2$ and $B \overset{(k)}{\Rightarrow}^* w_1$.

Asynchronous product of CFGs and FSAs. Given a CFG G and a FSA A, we now define a CFG $G \bowtie A$ such that $L(G \bowtie A) = L(G) \parallel L(A)$. Without loss of generality, we assume the set of accepting states of A is a singleton. We further show that the k-index language of $G \bowtie A$ is the asynchronous product of the k-index language of G and the language of A.

Definition 7. *Given a CFG $G = (\mathcal{X}, \Sigma_g, \mathcal{P}, X_0)$ in CNF and a FSA $A = (\Sigma_a, Q, \delta, q_0, \{q_f\})$, we define $G \bowtie A$ as the CFG $G^{\bowtie} = (\mathcal{X}^{\bowtie}, \Sigma_{\bowtie}, \mathcal{P}^{\bowtie}, X_0^{\bowtie})$. First replace in G every production of the form $X \to \sigma(\in \Sigma_g \cup \{\varepsilon\})$ by two productions $X \to \sigma\bot$ and $\bot \to \varepsilon$ where $\bot$ is a variable not in $\mathcal{X}$. This modified grammar is again referred to as $G = (\mathcal{X}, \Sigma_g, \mathcal{P}, X_0)$.*

Then define $G^{\bowtie} = (\mathcal{X}^{\bowtie}, \Sigma_{\bowtie}, \mathcal{P}^{\bowtie}, X_0^{\bowtie})$ as follows:

- $\mathcal{X}^{\bowtie} = Q \times \mathcal{X} \times Q$; $\Sigma_{\bowtie} = \Sigma_g \cup \Sigma_a$; $X_0^{\bowtie} = \langle q_0, X_0, q_f \rangle$;
- $\mathcal{P}^{\bowtie}$ *contains no more than the following transitions:*
  - *if $X \to \sigma\bot \in \mathcal{P}$ and $\sigma \notin \Sigma_a$ then $\langle q, X, q' \rangle \to \sigma\langle q, \bot, q' \rangle \in \mathcal{P}^{\bowtie}$*
  - *if $\sigma \notin \Sigma_g$ and $(q, \sigma, q') \in \delta$ then $\langle q, X, q'' \rangle \to \sigma\langle q', X, q'' \rangle \in \mathcal{P}^{\bowtie}$*
  - *if $X \to \sigma\bot \in \mathcal{P}$, $(q, \sigma, q') \in \delta$ and $\sigma \neq \varepsilon$ then $\langle q, X, q'' \rangle \to \sigma\langle q', \bot, q'' \rangle \in \mathcal{P}^{\bowtie}$*
  - *if $X \to YZ \in \mathcal{P}$ then $\langle q_1, X, q_2 \rangle \to \langle q_1, Y, q' \rangle\langle q', Z, q_2 \rangle \in \mathcal{P}^{\bowtie}$*
  - *if $q \in Q$ then $\langle q, \bot, q \rangle \to \varepsilon \in \mathcal{P}^{\bowtie}$*

Proposition 2. *Let G, A and $G^{\bowtie}$ as in def. 7. We have $L^{(k)}(G \bowtie A) = L^{(k)}(G) \parallel L(A)$ for every $k \geq 1$, hence $L(G \bowtie A) = L(G) \parallel L(A)$. Moreover, $G^{\bowtie}$ is computable in time polynomial in $|G| + |A|$.*

Proof. It suffices to show that for each $q, q' \in Q$, $X \in \mathcal{X}$ and $k \geq 1$: $\langle q, X, q' \rangle \overset{(k)}{\Rightarrow}^* w$ iff $w \in w^a \parallel w^g$ and $X \overset{(k)}{\Rightarrow}^* w^g$ in G and $q \xrightarrow{w^a} q'$ in A.From Def. 7 it is clear that $G^{\bowtie}$ is computable in time polynomial in $|G|$ and $|A|$. □

Computing a FSA supporting $L(G)$. We first show that, given a CFG G, one can construct a FSA accepting a support of $L(G)$ and whose size is at most exponentially larger than the size of G.

Theorem 7. *Given a CFG $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ in CNF with n variables, we can compute a FSA A with $O(2^{n \log(n)})$ states such that $L(A)$ is a support of $L(G)$.*

The proof of Theorem 7 requires the following technical lemma.

Lemma 13. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_t)$ be in CNF and $D\colon X_0 \Rightarrow^* v \in \Sigma^*$ a leftmost derivation for some $X_0 \in \mathcal{X}$. There exists $v' \preceq v$ and a leftmost n-index derivation $D'\colon X_0 \overset{(n)}{\Longrightarrow}{}^* v'$, where n is the number of distinct variables appearing in D.*

Proof. By induction on the number m of sequences of steps of the form $X \Rightarrow^* wX\alpha \Rightarrow^* ww'\alpha$ with $w \neq \varepsilon$ occurring in D.

Basis. $m = 0$. The proof for this case is by induction on the number n of distinct variables appearing in D.

Basis. $n = 1$. Because G is in CNF and the assumption $m = 0$, D necessarily is such that $X_0 \Rightarrow v \in \Sigma$. Hence setting $v' = v$ we find that $X_0 \overset{(n)}{\Longrightarrow}{}^* v'$ which concludes the case.

Step. $n > 1$. Because G is in CNF and the assumption $m = 0$, it must be the case that D has the following form $X_0 \Rightarrow BC \Rightarrow^* w_1 C \Rightarrow^* w_1 w_2 = v$. Moreover $m = 0$ shows that X_0 does not appear in the subsequence of steps $D_1\colon B \Rightarrow^* w_1$ and $D_2\colon C \Rightarrow^* w_2$. The number of distinct variables appearing in D_1 and D_2 being at most $n - 1$ we conclude, by induction hypothesis, that there exists leftmost $(n-1)$-index derivations $D_1'\colon B \overset{(n-1)}{\Longrightarrow}{}^* w_1'$ and $D_2'\colon C \overset{(n-1)}{\Longrightarrow}{}^* w_2'$ with $w_1' w_2' \preceq w_1 w_2 = v$, hence that there exists a derivation $D'\colon X_0 \overset{(n)}{\Longrightarrow}{}^* BC \overset{(n)}{\Longrightarrow}{}^* w_1' C \overset{(n)}{\Longrightarrow}{}^* w_1' w_2' = v'$ such that $v' \preceq v$ and we are done.

Step. $m > 0$. Therefore in D there exists some variable X such that $X \Rightarrow^* wX\alpha \Rightarrow^* ww'\alpha$ and $w \neq \varepsilon$. Define the derivation D' given by D where the above subsequence of steps is replaced by $X \Rightarrow^* w'$. Clearly we have that the word v' produced by D' is a subword of v, the word produced by D. Moreover the above transformation on D allows to use the induction hypothesis on D', hence we find that there exists there exists a leftmost n-index derivation $D''\colon X_0 \overset{(n)}{\Longrightarrow}{}^* v''$ and $v'' \preceq v'$ and we are done since $v'' \preceq v' \preceq v$. □

Proof (of Theorem 7). From Lem. 13 it is easy to see that the words given by the leftmost n-index derivations is a support of $L(G)$. Recall that G is in CNF. Next we define a FSA $A = (\Sigma, Q, \delta, q_0, F)$ such that *i)* $Q = \left\{w \in \mathcal{X}^j \mid 0 \leq j \leq n\right\}$; *ii)* $\delta = \{(\alpha\, w, \gamma, \beta\, w) \mid (\alpha, \gamma\beta) \in \mathcal{P} \wedge \gamma \in \Sigma \cup \{\varepsilon\}\}$; *iii)* $q_0 = X_0$; *iv)* $F = \{\varepsilon\}$. It is easy to see that A simulates all the leftmost sequence of steps of index at most n and accepts only when those corresponds to n-index leftmost derivations, hence that $L(A)$ is a support of $L(G)$. Also since $n = |\mathcal{X}|$, Q has $O(n^n)$ states, or equivalently $O(2^{n \log(n)})$. □

From the construction, it is clear that there is a polynomial-space bounded algorithm (in $|G|$) that can implement the transition relation of A, that is, given an encoding of a state of A, produce iteratively the successors of the state.

Covering Context-free Languages by Bounded-Index Languages. Our second construction shows that, given a CFG G, we can construct an $O(|G|)$-index language that is a cover of $L(G)$.

Given a CFG $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ and $X, Y \in \mathcal{X}$, we say that Y *depends on* X if G has a production $X \to \alpha Y \beta$ for some $\alpha, \beta \in (\mathcal{X} \cup \Sigma)^*$, or if there is a variable Z such that Y depends on Z and Z depends on X. A *strongly connected component* (SCCs) of G if a maximal subset of mutually dependent variables.

Theorem 8. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ be a CFG in CNF with n variables and k SCCs. Then $L^{(n+2k)}(G)$ is a cover of $L(G)$.*

The proof of Theorem 8 uses the following technical lemma which follows from the fact that the commutative images of $L(G)$ and $L^{(n+1)}(G)$ coincide [14].

Lemma 14. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ be a CFG in CNF with n variables. For every $a \in \Sigma$, if $X_0 \Rightarrow^* w\,a\,v$ for some $w, v \in \Sigma^*$, then $X_0 \overset{(n+1)}{\Longrightarrow}{}^* w'\,a\,v'$ for some $w', v' \in \Sigma^*$.*

Proof (of Theorem 8). Let $w \in L(G)$. If $w = \varepsilon$, then since G is in CNF, we have $X_0 \Rightarrow \varepsilon$. So, w is also in $L^{(n+2k)}(G)$. Hence, assume $w \neq \varepsilon$. We prove by induction on k that w is a subword of some $w' \in L^{(n+2k)}(G)$.

Basis. $k = 1$. We proceed by induction on $|w|$. Let $w = a$ for some $a \in \Sigma$. We conclude from $a \in L(G)$ and G is in CNF that $X_0 \Rightarrow a$, hence that $X_0 \overset{(1)}{\Longrightarrow}{}^* a$ and finally that $a \in L^{(n+2k)}(G)$ and we are done. If $w = av$ for some $v \neq \varepsilon$ then $X_0 \Rightarrow X_1X_2 \Rightarrow^* a\,v$ for some $X_1, X_2 \in \mathcal{X}$. By Lemma 14, $X_1 \overset{(n+1)}{\Longrightarrow}{}^* v_1$ for some $v_1 \succeq a$, and by induction hypothesis $X_2 \overset{(n+2)}{\Longrightarrow}{}^* v_2$ for some $v_2 \succeq v$. So we get $X_0 \overset{(2)}{\Longrightarrow} X_1X_2 \overset{(n+2)}{\Longrightarrow}{}^* v_1X_2 \overset{(n+2)}{\Longrightarrow}{}^* v_1v_2$ and taking $w' = v_1v_2$ we have $X_0 \overset{(n+2)}{\Longrightarrow}{}^* w' \succeq w$.

Step. $k > 1$. Let $\mathcal{Y} \subset \mathcal{X}$ be the set of variables of a bottom strongly connected component of G, and let $\mathcal{P}_\mathcal{Y} \subset \mathcal{P}$ be the productions of G with a variable of $\mathcal{Y}$ on the left side. For every $Y \in \mathcal{Y}$, let $G_Y = (\mathcal{Y}, \Sigma, \mathcal{P}_\mathcal{Y}, Y)$; further, let $G' = (\mathcal{X}\backslash\mathcal{Y}, \Sigma \cup \mathcal{Y}, \mathcal{P}\backslash\mathcal{P}_\mathcal{Y}, X_0)$. Since $w \in L(G)$, there exist derivations $X_0 \Rightarrow^* w_1Y_1w_2 \dots w_rY_rw_{r+1}$ in $L(G')$ and $Y_i \Rightarrow^* v_i$ in $L(G_{Y_i})$ for every $i = 1, \dots, r$ such that $w_1v_1 \dots w_rv_rw_{r+1} = w$. Since G' has $(k-1)$ SCCs, by induction hypothesis there is $X_0 \overset{(i_1)}{\Longrightarrow}{}^* w'_1Y'_1w'_2 \dots w'_tY'_tw'_{t+1}$ in $L(G')$, where $i_1 = n - |\mathcal{Y}| + 2(k-1)$ and such that $w'_1Y'_1w'_2 \dots w'_tY'_tw'_{t+1} \succeq w_1Y_1w_2 \dots w_rY_rw_{r+1}$. In particular we have $Y'_1 \dots Y'_t \succeq Y_1 \dots Y_r$ which implies that there exists a monotonic injection $h\colon \{1, \dots, r\} \to \{1, \dots, t\}$ such that $Y'_{h(i)} = Y_i$ for all $i \in \{1, \dots, r\}$. Since every G_Y has one strongly connected component, for every $j = 1, \dots, r$ there is $v'_j \succeq v_j$ such that $Y'_{h(j)} = Y_j \overset{(i_2)}{\Longrightarrow}{}^* v'_j$, where $i_2 = |\mathcal{Y}| + 2$. On the other hand, for every ℓ not in the image of h there also is some word v'_ℓ such that $Y'_\ell \overset{(i_2)}{\Longrightarrow}{}^* v'_\ell$, where $i_2 = |\mathcal{Y}| + 2$.

So we have

$$\begin{aligned}
X_0 &\overset{(i_1)}{\Longrightarrow}{}^* w'_1Y'_1w'_2Y'_2 \dots w'_tY'_tw'_{t+1} \\
&\overset{(i_1+i_2)}{\Longrightarrow}{}^* w'_1v'_1w'_2Y_2 \dots w'_tY_tw'_{t+1} \\
&\overset{(i_1+i_2)}{\Longrightarrow}{}^* w'_1v'_1w'_2v'_2 \dots w'_tY_tw'_{t+1} \\
&\quad\dots \\
&\overset{(i_1+i_2)}{\Longrightarrow}{}^* w'_1v'_1w'_2v'_2 \dots w'_tv'_tw'_{t+1}
\end{aligned}$$

Let $w' = w'_1v'_1 \dots w'_tv'_tw'_{t+1}$. Since $i_1 + i_2 = n + 2k$, we get $X_0 \overset{(n+2k)}{\Longrightarrow}{}^* w' \succeq w$, and we are done. □

Checking Emptiness of k-index languages. Finally, we show that $L^{(k)}(G) \neq \emptyset$ is decidable in NSPACE($k \log(|G|)$). In contrast, non-emptiness checking for context-free languages is P-complete as shown by Jones [19].

Theorem 9. *Given a CFG G in CNF and $k \geq 1$, it is in* NSPACE($k \log(|G|)$) *to decide whether* $L^{(k)}(G) \neq \emptyset$.

Proof. We give a non-deterministic space algorithm. The algorithm, called *query*, takes two parameters, a variable $X \in \mathcal{X}$ and a number $\ell \geq 1$, and guesses an ℓ-index derivation of some word starting from X. To do so, the algorithm guesses a production $(X, w) \in \mathcal{P}$ with head X. If $X \to \sigma$ is chosen, for $\sigma \in \Sigma \cup \{\varepsilon\}$, it returns true. If $X \to BC$ is chosen, the algorithm non-deterministically looks (using a recursive call) (i) for an $(\ell-1)$-index derivation from B and an ℓ-index derivation from C, or (ii) for an $(\ell-1)$-index derivation from C and an ℓ-index derivation from B. When $\ell = 0$ or a recursive call returns false, then *query* returns false.

We show the following invariant: *query*(ℓ, X) has an execution returning true iff $X \overset{(\ell)}{\Longrightarrow}^* w$ for some $w \in \Sigma^*$. It follows that *query*(k, X_0) returns true iff $L^{(k)}(G) \neq \emptyset$. The right-to-left direction is proved on the number m of steps in a bounded-index derivation. If $m = 1$ then we have $X \overset{(\ell)}{\Longrightarrow} w$ with $\ell = 1$ and *query*(ℓ, X) returns true by picking $(X, w) \in \mathcal{P}$. If $m > 1$ then the sequence of steps is as follows $X \overset{(\ell)}{\Longrightarrow} BC \overset{(\ell)}{\Longrightarrow}^* w$ where $\ell \geq 2$. From there either $B \overset{(\ell-1)}{\Longrightarrow}^* w_1$, $C \overset{(\ell)}{\Longrightarrow}^* w_2$, or $C \overset{(\ell-1)}{\Longrightarrow}^* w_2$ and $B \overset{(\ell)}{\Longrightarrow}^* w_1$ where $w = w_1 w_2$ holds. Let us assume the latter holds (the other case is treated similarly). Then we have $C \overset{(\ell-1)}{\Longrightarrow}^* w_2$ and $B \overset{(\ell)}{\Longrightarrow}^* w_1$ and both sequence have no more than $m - 1$ steps. Therefore the induction hypothesis shows that *query*$(\ell - 1, C)$ and *query*(ℓ, B) return true, and so does *query*(ℓ, X) by picking $(X, BC) \in \mathcal{P}$.

The left-to-right direction is proved by induction on the number m of times productions are picked in an execution of *query* that returns true. If $m = 1$ then $\ell \geq 1$ and a production $(X, \sigma) \in \mathcal{P}$ where $\sigma \in \Sigma \cup \{\varepsilon\}$ must have been picked, hence the derivation $X \overset{(\ell)}{\Longrightarrow} \sigma$. If $m > 1$, *query* recursively called itself after picking $(X, BC) \in \mathcal{P}$. Let us assume case (i) was executed ((ii) is treated similarly). Following the assumption both calls *query*$(\ell - 1, B)$ and *query*(ℓ, C) return true (hence $\ell \geq 2$) and are such that productions are picked at most $m - 1$ times. Next, the induction hypothesis shows that $B \overset{(\ell-1)}{\Longrightarrow}^* w_1$ and $C \overset{(\ell)}{\Longrightarrow} w_2$ for some w_1 and w_2. Finally, $X \to BC$ of $\mathcal{P}$ and $\ell \geq 2$ shows that $X \overset{(\ell)}{\Longrightarrow} BC \overset{(\ell)}{\Longrightarrow}^* w_1 C \overset{(\ell)}{\Longrightarrow}^* w_1 w_2$ and we are done.

It remains to show that *query*(k, X_0) runs in NSPACE($k \log(|G|)$). Observe that for each non-deterministic choice (i) or (ii), there is one recursive call *query*$(B, \ell - 1)$ or *query*$(C, \ell - 1)$. The other call (e.g., to *query*(C, ℓ) in case (i)) is tail-recursive and can be replaced by a loop. Since the index that is passed to that recursive call decreases by 1 and *query* returns false when the index is 0, we find that along every execution at most k stack frames are needed and each frame keeps track of a grammar variable which can be encoded with $\log(|G|)$ bits. Hence we find that $L^{(k)}(G) \neq \emptyset$ can be decided in NSPACE($k \log(|G|)$). □