Parameterized Verification of Asynchronous Shared-Memory Systems

JAVIER ESPARZA, Fakultät für Informatik, Technische Universität München, Germany
PIERRE GANTY, IMDEA Software Institute, Madrid, Spain
RUPAK MAJUMDAR, Max Planck Institute for Software Systems, Kaiserslautern, Germany

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 the case of pushdown systems, some language-theoretic constructions of independent interest. Our results are surprising, because parameterized verification problems on slight variations of our model are known to be undecidable. For example, the problem is undecidable for finite-state machines operating with synchronization primitives, and already for two communicating pushdown machines. Thus, our results show that a robust, decidable class can be obtained under the assumptions of anonymity and asynchrony.

CCS Concepts: • **Theory of computation → Distributed computing models; Problems, reductions and completeness**; **Grammars and context-free languages**; **Regular languages;**

Additional Key Words and Phrases: Parameterized systems, asynchronous shared memory, decidability, formal languages

ACM Reference Format:
Javier Esparza, Pierre Ganty, and Rupak Majumdar. 2016. Parameterized verification of asynchronous shared-memory systems. J. ACM 63, 1, Article 10 (February 2016), 48 pages.
DOI: http://dx.doi.org/10.1145/2842603

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 arbitrary relative speeds. 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

N-Greens Software: Next-GeneRation Energy-EfficieNt Secure Software. Comunidad de Madrid. (S2013/ICE-2731) and POLCA: Programming Large Scale Heterogeneous Infrastructures. EU FP7. (610686).
Authors' addresses: J. Esparza, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany; email: esparza@in.tum.de; P. Ganty, IMDEA Software Institute, Edificio IMDEA Software, Campus Montegancedo UPM, 28223-Pozuelo de Alarcón, Madrid, Spain; email: pierre.ganty@imdea.org; R. Majumdar, Max Planck Institute for Software Systems, Paul-Ehrlich-Str 26, 67663 Kaiserslautern, Germany; email: rupak@mpi-sws.org.

DOI: http://dx.doi.org/10.1145/2842603

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 aims 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, or 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 problem remains coNP-complete even if either the leader or the contributors (but not both) are allowed to be pushdown machines.

The case in which leader and contributors are finite-state machines, but the communication primitive is rendezvous, was studied by German and Sistla [1992]. They showed that the safety verification problem for their model is EXPSPACE-complete, but reduces to polynomial time in case there is no distinguished leader process. In contrast, our results show that the complexity is vastly reduced in the setting of nonatomic shared memory. In the absence of a distinguished leader process, we show that the complexity of safety verification is polynomial time, similar to German and Sistla [1992], even if each process is modeled by a pushdown machine.

The case in which leader and contributors are both pushdown machines was first considered by Hague [2011], who gave a coNP 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, that is, deciding if no error can be reached by computations in which no contributor or 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 polytime Turing machines is also coNP-complete.

These results show that nonatomicity substantially reduces the complexity of verification. Dually, they characterize the limitations of the parameterized computation model when atomic operations are not allowed. 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 PSPACE-complete already for a fixed number of state machines, and EXPSPACE-complete for parameterized state machines. Moreover, the safety verification problem is undecidable already for two fixed pushdown machines. A similar argument shows that the bounded safety problem is PSPACE-hard. In contrast, we get several coNP upper bounds in the finite-state and the bounded cases. Interestingly, our PSPACE-completeness result for pushdown systems shows that some computational power can be regained in the model by adding "memory" to the processes.

Besides intellectual curiosity, our work is motivated by several recent designs for large-scale distributed protocols in constrained environments. First, our model is an abstraction for programmable self-assembly in large robot swarms [Wood et al. 2013; Werfel et al. 2014; Rubenstein et al. 2014]. In these systems, thousands of low-cost, low-power robots run the same program and communicate with a small number of

"seeds" (leaders, in our nomenclature) and attempt to implement a joint behavior. The design constraints in these systems preclude the possibility of assigning individual identifiers. Instead, each robot is autonomous and interacts asynchronously and anonymously with other robots in the system. They have very limited sensing, computation, and actuation capabilities and are susceptible to crash faults. Owing to the basic sensing and communication capabilities and the faulty nature of interactions, there is no easy way to implement atomic communication primitives: for example, a robot can crash while holding a lock, or other robots in the system can completely miss sensing synchronization messages. Instead, systems such as Robobees [Wood et al. 2013] build on distributed protocols inspired by similar behavior in nature, such as swarm behavior in insects or self-assembly at the molecular level.

Second, our model can be seen as a simple abstraction for distributed protocols implemented on wireless sensor networks, such as vehicular networks. In these systems, a central coordinator (the base station) communicates with an arbitrary number of participating vehicles, and the system runs concurrently and asynchronously. Anonymity is desirable in these systems for privacy reasons [Laurendeau and Barbeau 2007] and implementing atomic communication primitives can be problematic: for instance, a vehicle might leave the network while holding a lock. Again, protocols in these systems work asynchronously and without synchronization primitives.

On the one hand, our results show bounds on what can be implemented in these models. On the other hand, our algorithms provide the foundations for safety verification of (abstractions of) these systems, and our coNP upper bounds opens the way to the application of SAT-solving or SMT-techniques. However, the focus of this article is understanding the theoretical limitations of the model rather than practical modeling and analysis 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 [Apt and Kozen 1986]. For this reason, special cases have been extensively studied. They vary according to the main characteristics of the systems to verify such as the communication topology of the processes (array, tree, unordered, and so on); their communication primitives (shared memory, unreliable broadcasts, rendezvous, lossy queues, and so on); or whether processes can distinguish from each other (using IDs, a distinguished process, and so on). Prominent examples include broadcast protocols [Esparza et al. 1999; Finkel and Leroux 2002; Dimitrova and Podelski 2008; Delzanno et al. 2010], in which finite-state processes communicate via broadcast messages; asynchronous programs [Ganty and Majumdar 2012; Viswanathan and Chadha 2009], in which finite-state processes communicate via unordered channels; finite-state processes communicating via ordered channels [Abdulla et al. 1996]; microarchitectures [McMillan 1998]; cache coherence protocols [Emerson and Kahlon 2003; Delzanno 2003]; communication protocols [Emerson and Namjoshi 1998]; and multithreaded shared-memory programs [Clarke et al. 2008; Delzanno et al. 2002; Kaiser et al. 2010; La Torre et al. 2010].

In a seminal article, German and Sistla [1992] studied a parameterized model with rendezvous as communication primitive, in which processes are finite-state machines. They show that safety verification is polynomial time if there are no leaders, but EXPSPACE-complete in the presence of a leader. Remarkably, the absence of atomic coordination in our model brings down the complexity to coNP in the presence of a leader. In the absence of a leader, safety verification is polynomial time in our model as well, even when processes are pushdown machines.

In addition to the model of Hague [2011], several models close to ours have been previously studied in distributed computing. For example, Guerraoui and Ruppert [2007] studied a model of distributed computing with identity-free, asynchronous processors

and nonatomic registers. The emphasis there was the development of distributed algorithm primitives such as timestamping, snapshots, and consensus, using either unbounded registers or an unbounded number of bounded registers. Population protocols [Angluin et al. 2007] are a different model for asynchronous, identity-free, processes. In these protocols, arbitrarily many identity-free processes interact asynchronously in rendezvous fashion under the control of a fair scheduler. The emphasis in these works is on describing what functions can be computed in the model. While we present our complexity results for the safety verification problem, we can dually view our results as the computational power of our model. Thus, for example, our results show that asynchronous shared-memory systems, in which the leader and all contributors are pushdown machines and the global store is finite-state, can compute all PSPACE predicates.

2. FORMAL MODEL: NONATOMIC NETWORKS

In this article, we follow the "systems-as-languages" approach: system actions are modeled as symbols in an alphabet, executions are modeled as words, that is, finite sequences of symbols, and the system itself is modeled as the language of its executions. Operations that combine systems into larger ones are modeled as operations on languages.

2.1. Systems as Languages

We recall some basic notions of language theory. An *alphabet* Σ is a finite, nonempty set of symbols. A *word* over Σ is a finite sequence of symbols of Σ, and a *language* is a set of words. As customary, Σ^* denotes the language of all words over Σ. The length of a word w is denoted by $|w|$. The empty word is denoted by ε, and its length is 0. The concatenation of two words $x = a_1 \cdots a_n$ and $y = b_1 \cdots b_m$ is the word $xy = a_1 \cdots a_n b_1 \cdots b_m$. The ith symbol ($1 \leq i \leq |w|$) of a word w is denoted by $(w)_i$, and the subsequence $(w)_i(w)_{i+1} \ldots (w)_j$, where ($1 \leq i, j \leq |w|$), is denoted by $(w)_{i..j}$. We use *regular expressions* to denote languages. A regular expression over the alphabet Σ is defined inductively using the grammar

$$r \ ::= \ \emptyset \mid \varepsilon \mid a \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^*,$$

where $a \in \Sigma$. A regular expression r denotes a language $L(r)$ defined inductively as follows: $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, $L(a) = \{a\}$, $L(r_1 + r_2) = L(r_1) \cup L(r_2)$, $L(r_1 \cdot r_2) = \{x\,y \mid x \in L(r_1), y \in L(r_2)\}$, and $L(r^*) = \{x_1 \ldots x_k \mid k \geq 0$ and $x_i \in L(r)$ for each $i = 1, \ldots, k\}$.

We often speak of "the system L," where L is a language, meaning the system whose set of executions is L.

Combining systems: Shuffle. Intuitively, the shuffle of two systems is the result of putting them side by side, letting them run in parallel, without any kind of communication between them, and looking at them as one single system. In the systems-as-languages approach, the two systems are two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, and their shuffle, denoted by $L_1 \between L_2$, is the language over $(\Sigma_1 \cup \Sigma_2)$ defined as follows. Given two words $x \in \Sigma_1^*, y \in \Sigma_2^*$, we say that $z \in (\Sigma_1 \cup \Sigma_2)^*$ is an *interleaving* of x and y if there exist (possibly empty) words $x_1, \ldots, x_n \in \Sigma_1^*$ and $y_1, \ldots, y_n \in \Sigma_2^*$ such that $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$, and $z = x_1 y_1 \ldots x_n y_n$. The *shuffle* of L_1 and L_2 is the language $L_1 \between L_2 = \bigcup_{x \in L_1, y \in L_2} x \between y$, where $x \between y$ denotes the set of all interleavings of x and y. Shuffle is associative and commutative; thus, we can write $L_1 \between \cdots \between L_n$ or $\between_{i=1}^{n} L_i$.

Example 2.1. For $L_1 = a^*$ and $L_2 = b^*$, we get $L_1 \between L_2 = (a + b)^*$. For $L_1 = a^n$ and $L_2 = a^m$ we get $L_1 \between L_2 = a^{n+m}$.

Combining systems: Asynchronous product. The asynchronous product of two systems $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ also puts them side by side and lets them run in parallel. However, actions in the alphabet of both systems must now be executed jointly: if the first system is ready to execute an action $a \in \Sigma_1 \cap \Sigma_2$, but the second is not, then the first system must wait for the second. The language of the resulting system, called the asynchronous product of L_1 and L_2, is denoted by $L_1 \parallel L_2$, and defined as follows. Let $Proj_\Sigma(w)$ be the word obtained by erasing from w all occurrences of symbols not in Σ. The *asynchronous product* of L_1 and $L_2 \subseteq \Sigma_2^*$, denoted $L_1 \parallel L_2$, is the language over the alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ such that $w \in L_1 \parallel L_2$ if and only if $Proj_{\Sigma_1}(w) \in L_1$ and $Proj_{\Sigma_2}(w) \in L_2$. If a language consists of a single word, for example, $L_1 = \{w_1\}$, we abuse notation and write $w_1 \parallel L_2$ instead of $\{w_1\} \parallel L_2$.

Example 2.2. Let $\Sigma_1 = \{a, c\}$ and $\Sigma_2 = \{b, c\}$. For $L_1 = (ac)^*$ and $L_2 = (bc)^*$ we get $L_1 \parallel L_2 = ((ab + ba)c)^*$. For $L_1 = c^n$ and $L_2 = c^m$ we get $L_1 \parallel L_2 = c^n$ if $n = m$, and $L_1 \parallel L_2 = \emptyset$ otherwise.

Observe that $L_1 \parallel L_2$ does not only depend on L_1, L_2, but also on the alphabets Σ_1 and Σ_2. For example, if $\Sigma_1 = \{a\}$ and $\Sigma_2 = \{b\}$, then $\{a\} \parallel \{b\} = \{ab, ba\}$, but if $\Sigma_1 = \{a, b\} = \Sigma_2$, then $\{a\} \parallel \{b\} = \emptyset$. Thus, we should more properly write $L_1 \parallel_{\Sigma_1, \Sigma_2} L_2$. However, since the alphabets Σ_1 and Σ_2 will be clear from the context, we will omit them.

We describe systems as combinations of shuffles and asynchronous products; for instance, we write $L_1 \parallel (L_2 \between L_3)$. In these expressions, we assume that $\between$ binds tighter than $\parallel$; thus, $L_1 \between L_2 \parallel L_3$ is the language $(L_1 \between L_2) \parallel L_3$, and not $L_1 \between (L_2 \parallel L_3)$.

Like shuffle, asynchronous product is also associative and commutative; thus, we write $L_1 \parallel \cdots \parallel L_n$ or $\parallel_{i=1}^n L_i$. Moreover, unlike shuffle, it is idempotent, that is, $L \parallel L = L$. More generally, it follows immediately from the definition of the asynchronous product that, if $L, L' \subseteq \Sigma^*$ and $L \subseteq L'$, then $L \parallel L' = L$. The following lemma generalizes this property even further.

LEMMA 2.3. *Let $L \subseteq \Sigma^*$ and $\Sigma' \subseteq \Sigma$. We have that*

(1) $L \parallel Proj_{\Sigma'}(L) = L$ and, more generally,
(2) $L \parallel L' = L$ for every $L' \subseteq (\Sigma')^*$ such that $Proj_{\Sigma'}(L) \subseteq L'$,
(3) $Proj_{\Sigma'}(L' \parallel L) = L' \parallel Proj_{\Sigma'}(L)$ for all languages $L' \subseteq (\Sigma')^*$.

PROOF. (1) To prove $L \subseteq L \parallel Proj_{\Sigma'}(L)$, observe that for every $w \in L$, we have $Proj_{\Sigma'}(w) \in Proj_{\Sigma'}(L)$ by definition of projection; thus, $w \in L \parallel Proj_{\Sigma'}(L)$ by the definition of the asynchronous product. To prove $L \parallel Proj_{\Sigma'}(L) \subseteq L$, let $w \in L \parallel Proj_{\Sigma'}(L)$. By the definition of asynchronous product, $w \in (\Sigma \cup \Sigma')^* = \Sigma^*$ and $Proj_\Sigma(w) \in L$. But $Proj_\Sigma(w) = w$.

(2) To prove $L \parallel L' \subseteq L$, take a word $w \in L \parallel L'$. This word is over $(\Sigma \cup \Sigma')^* = \Sigma^*$. By definition of asynchronous product, $Proj_\Sigma(w) \in L$, but $Proj_\Sigma(w) = w$. To prove $L \parallel L' \supseteq L$, let $w \in L$. Then; thus, $Proj_{\Sigma'}(w) \in L'$ by hypothesis. Since $Proj_\Sigma(w) = w \in L$, we get $w \in L \parallel L'$ by the definition of the asynchronous product.

(3) Let $w \in Proj_{\Sigma'}(L' \parallel L)$. This means that there is a $w' \in L' \parallel L$ whose projection $Proj_{\Sigma'}(w') = w$. It follows from the definition of asynchronous product and $\Sigma' \subseteq \Sigma$ that $Proj_{\Sigma'}(w') \in L'$ and $w' \in L$. Moreover, $Proj_{\Sigma'}(w') = w \in Proj_{\Sigma'}(L)$.

Conversely, let $w \in L' \parallel Proj_{\Sigma'}(L)$. Then, $w \in L'$ and there is a $w' \in L$ such that $Proj_{\Sigma'}(w') = w$. It follows that $w' \in L' \parallel L$ and $w \in Proj_{\Sigma'}(L' \parallel L)$. □

In what follows, we extend regular expressions with the $\parallel$ and $\between$ operators, with the obvious semantics.

In the following, we assume the following results known from language theory (e.g., see Hopcroft and Ullman [1979]). The regular languages are closed under shuffle product and asynchronous product. Moreover, given finite-state machines for two languages, we can construct in polynomial time a finite-state machine for their shuffle product or asynchronous product. A similar result holds for the shuffle or asynchronous product of a context-free language and a regular language (see Appendix A): given a pushdown automaton (or a context-free grammar) for the context-free language, and given a finite-state machine for the regular language, we can construct in polynomial time a pushdown automaton or a context-free grammar for their shuffle or asynchronous product. The constructions are similar to those employed to prove closure under the more familiar operation of intersection.

2.2. Nonatomic Networks

A nonatomic network is an infinite family of systems parameterized by a number k. The kth element of the family has $k+1$ components communicating through a global store by means or read and write actions. The store is modeled as one single global variable with a possibly very large, but finite, set of values corresponding to all possible configurations of the store. One of the $k+1$ components is the leader, while the other k are the contributors. All contributors have exactly the same possible behaviors (they are copies of the same language), while the leader may behave differently. The network is called nonatomic because components cannot atomically execute sequences of actions, only one single read or write.

Formally, we fix a finite 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 define $\mathcal{G}(a_1, \ldots, a_n) = \{a_i(g) \mid 1 \leq i \leq n,\ g \in \mathcal{G}\}$.

We fix two languages $\mathcal{D}$ and $\mathcal{C}$, called the *leader* and the *contributor*, with alphabets $\Sigma_\mathcal{D} = \mathcal{G}(r_d, w_d)$ and $\Sigma_\mathcal{C} = \mathcal{G}(r_c, w_c)$, respectively, where r_d, r_c are called *reads* and w_c, w_d are called *writes*. We write $w_\star$ (resp., $r_\star$) to stand for either w_c or w_d (respect., r_c or r_d). We further assume that (i) for each value $g \in \mathcal{G}$, there is a word in the leader or contributor that reads or writes g (if not, the value is never used and is removed from $\mathcal{G}$); and (ii) that both $\mathcal{D}$ and $\mathcal{C}$ are prefix-closed languages.

Additionally, we fix a language $\mathcal{S}$, called the *store*, over the alphabet $\Sigma_\mathcal{D} \cup \Sigma_\mathcal{C}$. It models the sequences of read and write operations supported by an atomic register: a write $w_\star(g)$ writes g to the register, while a read $r_\star(g)$ succeeds when the register's current value is g. In other words, the store is always willing to execute any write, but it is only willing to execute a read action whose value coincides with the value of the last write. Initially, the store is only willing to execute a write. Formally, $u \in (\Sigma_\mathcal{D} \cup \Sigma_\mathcal{C})^*$ belongs to $\mathcal{S}$ if and only if for each position p such that $(u)_p = r_\star(g)$ for some $g \in \mathcal{G}$, there exists an earlier position $p' < p$ such that $(u)_{p'} = w_\star(g)$ and $(u)_{p'+1..p} \in (r_\star(g))^*$. Alternatively, $\mathcal{S}$ can also be defined as the language of the regular expression $\sum_{g\in\mathcal{G}}(w_\star(g)\,(r_\star(g))^*)^*$. Observe that $\mathcal{S}$ is completely determined by $\Sigma_\mathcal{D}$ and $\Sigma_\mathcal{C}$, and that $\mathcal{S}$ is prefix-closed.

Definition 2.4. Let $\mathcal{D} \subseteq \Sigma_\mathcal{D}^*$ and $\mathcal{C} \subseteq \Sigma_\mathcal{C}^*$ be a leader and a contributor, and let $k \geq 1$. The *k-instance of the (nonatomic) $(\mathcal{D}, \mathcal{C})$-network* is the language $\mathcal{N}_k = (\mathcal{D} \parallel \mathcal{S} \parallel \ \lbag\!\rbag_k \mathcal{C})$ where $\lbag\!\rbag_k\mathcal{C}$ is an abbreviation for $\lbag\!\rbag_{i=1}^k \mathcal{C}$. The *nonatomic $(\mathcal{D}, \mathcal{C})$-network* $\mathcal{N}$ is the language $\mathcal{N} = \bigcup_{k=1}^\infty \mathcal{N}_k$. We omit the prefix $(\mathcal{D}, \mathcal{C})$ when it is clear from the context, and call $\mathcal{N}$ and $\mathcal{N}_k$ the network and the k-instance of the network, respectively. It follows from the properties of shuffle and asynchronous product that

$$\mathcal{N} = (\mathcal{D} \parallel \mathcal{S} \parallel \ \lbag\!\rbag_\infty \mathcal{C}),$$

where $\lbag\!\rbag_\infty\mathcal{C}$ is an abbreviation of $\bigcup_{k=1}^\infty \lbag\!\rbag_k\mathcal{C}$.

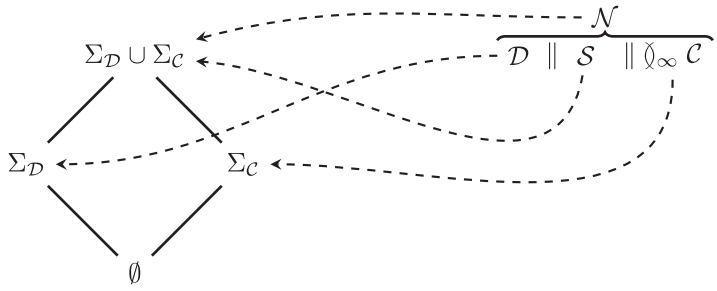


Fig. 1. Lattice formed by the alphabets of $\mathcal{N}$'s components: $\mathcal{D} \subseteq (\Sigma_{\mathcal{D}})^*$, $\mathcal{C} \subseteq (\Sigma_{\mathcal{C}})^*$, and $\mathcal{S}, \mathcal{N} \subseteq (\Sigma_{\mathcal{D}} \cup \Sigma_{\mathcal{C}})^*$.

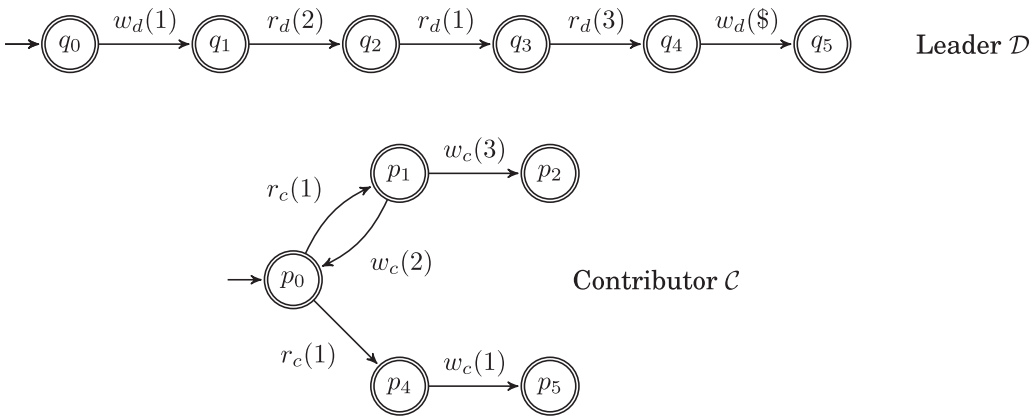


Fig. 2. Leader and contributor of a nonatomic network.

Figure 1 gives a graphical presentation of the alphabets of the constituent languages of $\mathcal{N}$ and their relationships. It is also worth pointing that, because $\mathcal{D}$, $\mathcal{C}$, and $\mathcal{S}$ are prefix-closed languages, every k-instance $\mathcal{N}_k$ as well as $\mathcal{N}$ are also prefix-closed.

Notation 2.5. We often speak of the projection of a language L onto the alphabet of the leader or the contributor. To lighten the notation, we write $Proj_{\mathcal{D}}(L)$ and $Proj_{\mathcal{C}}(L)$, instead of $Proj_{\Sigma_{\mathcal{D}}}(L)$ and $Proj_{\Sigma_{\mathcal{C}}}(L)$.

It is convenient to introduce a notion of *compatibility* between a word of the leader and a multiset of words of the contributor. Intuitively, compatibility means that all the words can be interleaved into a legal sequence of reads and writes, that is, a sequence supported by an atomic register. Formally:

Definition 2.6. Let $u \in \Sigma_{\mathcal{D}}^*$, and let $M = \{v_1, \ldots, v_k\}$ be a multiset of words over $\Sigma_{\mathcal{C}}^*$ (possibly containing multiple copies of a word). We say that u is *compatible* with M if and only if $(u \parallel \mathcal{S} \parallel \Diamond_{i=1}^{k} v_i) \neq \emptyset$.

Because it makes the exposition easier to follow, our examples use (read/write) finite-state automata to denote languages. A *finite-state automaton* (FSA) is a tuple $A = (\mathcal{Q}, \mathcal{G}(r, w), \Delta, q_0, F)$, where $\mathcal{Q}$ is a finite set of *states* with an *initial state* $q_0 \in \mathcal{Q}$ and a set of accepting states $F \subseteq \mathcal{Q}$, $\mathcal{G}(r, w)$ is a read-write alphabet, and $\Delta \subseteq (\mathcal{Q} \times (\mathcal{G}(r, w) \cup \{\varepsilon\}) \times \mathcal{Q})$ is a set of *transitions*. The language $L(A)$ is defined as usual.

Example 2.7. Figure 2 shows an example of a nonatomic network with $\mathcal{G} = \{1, 2, 3, \$\}$. The languages $\mathcal{D}$ and $\mathcal{C}$ are represented using FSAs. In particular, $\mathcal{D}$ contains six words

(the six prefixes of the word $w_d(1)\,r_d(2)\,r_d(1)\,r_d(3)\,w_d(\$)$), while $\mathcal{C}$ contains infinitely many words.

Consider the 1-instance $\mathcal{N}_1 = (\mathcal{D} \parallel \mathcal{S} \parallel \mathcal{C})$ of the network. For instance, we have $u = w_d(1)\,r_c(1)\,w_c(2)\,r_d(2) \in \mathcal{N}_1$. To prove it, we have to show that the projections of u onto $\Sigma_{\mathcal{D}}$, $\Sigma_{\mathcal{C}}$ and their union are words of $\mathcal{D}$, $\mathcal{C}$, and $\mathcal{S}$, respectively. Observe first that, since the alphabet of the store (i.e., $\Sigma_{\mathcal{D}}, \cup\, \Sigma_{\mathcal{C}}$) contains all reads and writes, the projection of any word onto $\Sigma_{\mathcal{D}} \cup \Sigma_{\mathcal{C}}$ is always the word itself. Thus, we have to show $u \in \mathcal{S}$, which, since $\mathcal{S} = \sum_{g \in \mathcal{G}} (w_\star(g)\,(r_\star(g))^*)^*$, is indeed the case. The projections of u onto $\Sigma_{\mathcal{D}}$ and $\Sigma_{\mathcal{C}}$ are $w_d(1)\,r_d(2)$ and $r_c(1)\,w_c(2)$, which are words of $\mathcal{D}$ and $\mathcal{C}$. In the terminology of Definition 2.6, $w_d(1)\,r_d(2)$ is compatible with the singleton set $\{r_c(1)\,w_c(2)\}$.

The 2-instance $\mathcal{N}_2 = (\mathcal{D} \parallel \mathcal{S} \parallel (\mathcal{C} \between \mathcal{C}))$ contains the word

$$w_d(1)\,r_c(1)\,r_c(1)\,w_c(2)\,r_d(2)\,w_c(1)\,r_c(1)\,r_d(1)\,w_c(3)\,r_d(3)\,w_d(\$). \tag{1}$$

To prove this, we first observe that (1) is a legal sequence of reads and writes, and so it belongs to $\mathcal{S}$. Now, we take

$$\begin{aligned} v &= w_d(1)\,r_d(2)\,r_d(1)\,r_d(3)\,w_d(\$) \\ v_1 &= r_c(1)\,w_c(1) \\ v_2 &= r_c(1)\,w_c(2)\,r_c(1)\,w_c(3) \end{aligned} \tag{2}$$

and show that v is compatible with $\{v_1, v_2\}$ by proving that (1) belongs to $(v \parallel \mathcal{S} \parallel (v_1 \between v_2))$. For this, we label the contributor actions of (1) by 1 and 2 such that the projection onto the actions labeled by 1 yield v_1, and analogously for v_2, while the projection onto the actions without a label yields v:

$$w_d(1)\,\underbrace{r_c(1)}_{1}\,\underbrace{r_c(1)}_{2}\,\underbrace{w_c(2)}_{2}\,r_d(2)\,\underbrace{w_c(1)}_{1}\,\underbrace{r_c(1)}_{2}\,r_d(1)\,\underbrace{w_c(3)}_{2}\,r_d(3)\,w_d(\$) \tag{3}$$

3. THE SAFETY VERIFICATION PROBLEM

In our systems-as-languages approach, $\mathcal{D}$ is the language of possible executions of the code of the leader. We assume that this code has been instrumented so that, when an error occurs, the leader writes a special error value in the store, modeled by the symbol \$. Formally, a word u of a $(\mathcal{D}, \mathcal{C})$-network $\mathcal{N}$ is unsafe if it ends with an occurrence of $w_d(\$)$; $\mathcal{N}$ is *safe* if and only if it contains no unsafe word. Observe that, if $\mathcal{N}$ is safe, then no instance of the network—whatever the number of contributors—can produce an error.

Safety can also be defined in other ways: we could declare a word unsafe if it ends with an occurrence of $w_c(\$)$, or even with an occurrence of $w_\star(\$)$. As we will see, our complexity results are the same for any of these choices.

Safety as emptiness. From this point on, we will make the following assumption: every word u in $\mathcal{D}$ ends with $w_d(\$)$, formally $(u)_{|u|} = w_d(\$)$ for every $u \in \mathcal{D}$. This comes with no loss of generality and allows the safety checking problem to be reduced to a language emptiness problem: $\mathcal{N}$ is safe if and only if $\mathcal{N} = \emptyset$. However, observe that $\mathcal{D}$ is no longer prefix closed as seen when comparing the leader in Figure 3 and Figure 2. This also holds for $\mathcal{N}$.

Example 3.1. Consider again the network of Example 2.7. A bit of reasoning shows that no word of the 1-instance $\mathcal{N}_1$ is unsafe. During the execution of such a word, the leader must move at some point from q_2 to q_3 by reading a 1, then later from q_3 to q_4 by reading a 3. The 1 and the 3 read by the leader must be previously written by the contributor. However, after writing the 1, the contributor is necessarily in state p_5, from where it cannot reach p_1 to write a 3.

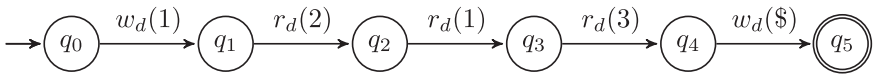


Fig. 3. Leader $\mathcal{D}$ for which only those words ending with $w_d(\$)$ are kept.

This example provides an important intuition. If we view $\mathcal{D}$ as the language of a (possibly infinite) state machine, then the safety problem becomes the problem of deciding whether the leader, helped by the contributors, can reach a state from which it can write \$ (state q_4 in the example). For this, the leader must overcome obstacles in the form of read actions: often, the leader can only perform an action $r_d(g)$ if it is helped by a contributor that performs $w_c(g)$. In the example, the leader must overcome obstacles at states q_1, q_2, and q_3.

We study the complexity of safety verification for networks in which the read-write languages $\mathcal{D}$ and $\mathcal{C}$ of leader and contributor are generated by an abstract machine, like an FSA or a pushdown automaton (PDA). More precisely, given two classes of machines D, C (like FSA or PDA), we study the safety problem Safety(D, C) defined as follows:

Safety(D, C)

Given: machines $D \in$ D and $C \in$ C
Decide: Is $(L(D) \parallel \mathcal{S} \parallel \{\!|_\infty L(C))$ safe?

In the next sections, we prove that Safety(FSA, FSA), Safety(PDA, FSA), and Safety(FSA, PDA) are coNP-complete, while Safety(PDA, PDA) is PSPACE-complete. Remarkably, the complexity in all cases is *lower* than that of the problem:

Safety_non_parametric(D, C)

Given: machines $D \in$ D, $C \in$ C, number k
Decide: Is $(L(D) \parallel \mathcal{S} \parallel \{\!|_k L(C))$ safe?

In particular, it is well known that networks with a PDA-leader, a store, and *one single PDA-contributor* (i.e., the inputs of Safety_non_parametric(PDA, PDA) with $k = 1$) are Turing powerful. In such a network, leader and contributor can use their stacks to simulate a queue, and it is well known that finite-state machines whose transitions act over one queue are Turing-powerful. Having arbitrarily many contributors turns out to be a disadvantage for the network. Intuitively, the reason is that the leader cannot know whether the value it reads in the store has been written by a contributor that has read its last message, or perhaps by some other contributor that has missed the last part of the computation.

4. THE SIMULATION LEMMA

Given a network $\mathcal{N} = (\mathcal{D} \parallel \mathcal{S} \parallel \{\!|_\infty \mathcal{C})$, the Simulation Lemma constructs a *finite* set $\{Sim_g \mid g \in \mathcal{G}\}$ of *simulators*, one for each value of $\mathcal{G}$, such that

$$Proj_{\mathcal{D}}(\mathcal{D} \parallel \mathcal{S} \parallel \{\!|_\infty \mathcal{C}) = Proj_{\mathcal{D}}(\mathcal{D} \parallel \mathcal{S} \parallel \{\!|_{g \in \mathcal{G}} Sim_g). \tag{4}$$

Recall from Lemma 2.3(3) that $Proj_{\mathcal{D}}(\mathcal{D} \parallel L) = (\mathcal{D} \parallel Proj_{\mathcal{D}}(L))$ holds for every language L; thus, in particular, we can rewrite Equation (4) as

$$(\mathcal{D} \parallel Proj_{\mathcal{D}}(\mathcal{S} \parallel \{\!|_\infty \mathcal{C})) = (\mathcal{D} \parallel Proj_{\mathcal{D}}(\mathcal{S} \parallel \{\!|_{g \in \mathcal{G}} Sim_g)).$$

Thus, informally, the Simulation Lemma states that the leader cannot distinguish $(\mathcal{S} \parallel \{\!|_\infty \mathcal{C})$ from $(\mathcal{S} \parallel \{\!|_{g \in \mathcal{G}} Sim_g)$: when composed with these two systems, it executes

exactly the same sequences. We can thus say that the simulators are able to simulate an unbounded number of contributors.

Notation 4.1 (*The special action* $w_c(\#)$). In order to define the simulators, we assume without loss of generality that the alphabet $\Sigma_{\mathcal{C}}$ contains a special action $w_c(\#)$. This action is best understood as a *corrupting* write. It adds the capability for simulators to overwrite the store with a value that nobody can read. After a corrupting write, a read can only happen after the store is overwritten with a value in $\mathcal{G}$. The alphabet of the simulator Sim_g is thus $\mathcal{G}(r_c) \cup \{w_c(g), w_c(\#)\}$. That is, Sim_g can read arbitrary values, but can only write the value g or corrupt the store.

Informally, the behavior of the simulator is as follows. As long as $\mathcal{C}$ does not execute $w_c(g)$, the simulator behaves like $\mathcal{C}$, just replacing every write $w_c(g')$ by a corrupting write, that is, by $w_c(\#)$. If $\mathcal{C}$ executes $w_c(g)$, then the simulator also executes $w_c(g)$; from this moment on, it keeps executing $w_c(g)$ *an arbitrary number of times*. The rationale for this definition is the following intuition (formalized in Lemma 4.8 as the Copycat Lemma): If a contributor can execute a sequence of reads and writes as a component of a network, then *arbitrarily many* contributors can execute the same sequence by copying the behavior of the first: if the first contributor reads a value, then the replicators read the same value immediately after; if the first contributor writes a value, the replicators write the same value immediately after. This is possible because replicating a read or a write does not change the value of the store. Note that the Copycat Lemma would not hold in a setting with atomic actions: if the first contributor read a value and wrote a different value to the store atomically, the next contributors would be stuck.

Example 4.2. Consider the multiset of words

$$\begin{aligned} v &= w_d(1)\, r_d(2)\, r_d(1)\, r_d(3)\, w_d(\$) \\ v_1 &= r_c(1)\, w_c(1) \\ v_2 &= r_c(1)\, w_c(2)\, r_c(1)\, w_c(3) \\ v_3 &= r_c(1)\, w_c(2)\, r_c(1)\, w_c(3) \end{aligned} \tag{5}$$

obtained from Equation (2) by replicating v_2. We construct a word of $(v \parallel \mathcal{S} \parallel (v_1 \between v_2 \between v_3))$ from the word of Equation (1) and Equation (3) by inserting the actions of v_3 right after the actions of v_2, as follows:

$$w_d(1)\, \underbrace{r_c(1)}_{1}\, \underbrace{r_c(1)}_{2}\, \underbrace{w_c(2)}_{2}\, r_d(2)\, \underbrace{w_c(1)}_{1}\, \underbrace{r_c(1)}_{2}\, r_d(1)\, \underbrace{w_c(3)}_{2}\, r_d(3)\, w_d(\$)$$

$$w_d(1)\, \underbrace{r_c(1)}_{1}\, \underbrace{r_c(1)}_{2}\, \underbrace{r_c(1)}_{3}\, \underbrace{w_c(2)}_{2}\, \underbrace{w_c(2)}_{3}\, r_d(2)\, \underbrace{w_c(1)}_{1}\, \underbrace{r_c(1)}_{2}\, \underbrace{r_c(1)}_{3}\, r_d(1)\, \underbrace{w_c(3)}_{2}\, \underbrace{w_c(3)}_{3}\, r_d(3)\, w_d(\$)$$

Intuitively, the new word describes a behavior of the network $\mathcal{N}_3$ in which the third contributor behaves as a *copycat* of the second one: it reads or writes the same value immediately after the second contributor reads or writes. In the same way, we can add further copies $v_4, v_5, \ldots$ of v_2 to the multiset.

Therefore, once a contributor executes one $w_c(g)$ action (this would be $w_c(3)$ in our example), arbitrarily many replicators are ready to supply the leader with as many $w_c(g)$ actions as it may need for its $r_d(g)$-obstacles, and that is what the simulator does. Formally, Sim_g is defined as follows.

Definition 4.3. Given a leader $\mathcal{D}$ and a contributor $\mathcal{C}$, define for every $g \in \mathcal{G}$, $\mathcal{C}_g$ as the language $\mathcal{C}/\{w_c(g)\}$, where L_1/L_2, the *right quotient operation*, returns the set $\{w \mid \exists x \in L_2 : w\,x \in L_1\}$. That is, $\mathcal{C}_g$ is the set of words u satisfying $u\, w_c(g) \in \mathcal{C}$. Let $\mathcal{C}_g^{\#}$ be the result of replacing every occurrence of a symbol in $\mathcal{G}(w_c)$ in the words of $\mathcal{C}_g$, by $w_c(\#)$ (if $\mathcal{C}_g = \emptyset$, then $\mathcal{C}_g^{\#} = \emptyset$ as well). The *g-simulator* of $\mathcal{C}$ is the language

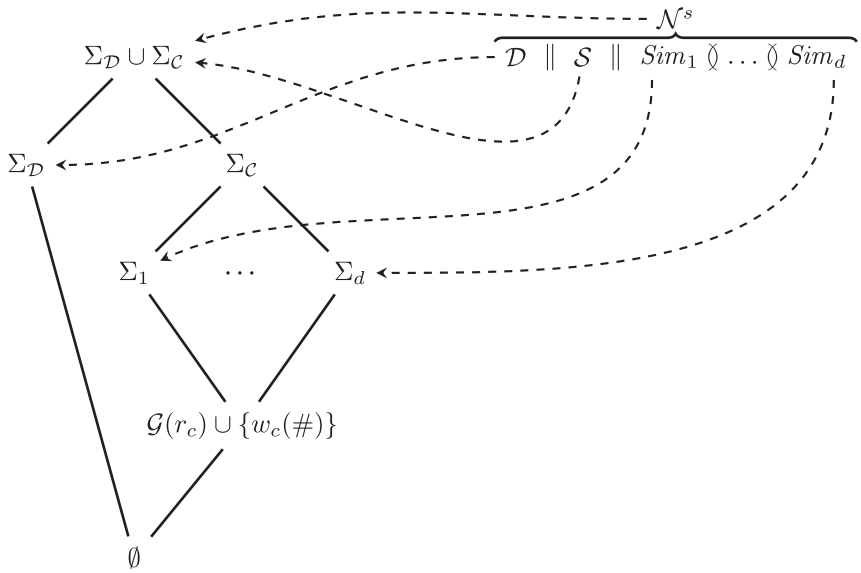


Fig. 4. Lattice formed by the alphabets of $\mathcal{N}^s$'s components: $\mathcal{D} \subseteq (\Sigma_{\mathcal{D}})^*$, $Sim_g \subseteq (\Sigma_g)^*$ for each $g \in \mathcal{G} = \{1, \ldots, d\}$, and $\mathcal{S}, \mathcal{N}^s \subseteq (\Sigma_{\mathcal{D}} \cup \Sigma_{\mathcal{C}})^*$, where $\Sigma_{\mathcal{D}} = \mathcal{G}(r_d, w_d)$, $\Sigma_{\mathcal{C}} = \mathcal{G}(r_c, w_c) \cup \{w_c(\#)\}$, and $\Sigma_g = \mathcal{G}(r_c) \cup \{w_c(\#), w_c(g)\}$ for all $g \in \mathcal{G}$.

$Sim_g = \varepsilon + \mathcal{C}_g^{\#}\, w_c(g)^+$ over the alphabet $\mathcal{G}(r_c) \cup \{w_c(g), w_c(\#)\}$, also sometimes denoted Σ_g. The *simulating network* $\mathcal{N}^s$ is defined as $(\mathcal{D} \parallel \mathcal{S} \parallel \lozenge_{g \in \mathcal{G}}\, Sim_g)$.

The alphabets of the components of $\mathcal{N}^s$ are shown in Figure 4.

An immediate consequence of the Simulation Lemma is that $Proj_{\mathcal{D}}(\mathcal{N}) = Proj_{\mathcal{D}}(\mathcal{N}^s)$. Hence, because we previously reduced safety checking to language emptiness, this means that $\mathcal{N}$ is safe if $\mathcal{N}^s$ is safe. Further observe that the language Sim_g is not necessarily prefix closed although it does contain ε, the empty string. We thus find that $\mathcal{N}^s$ is not prefix closed; however, the projections on $\Sigma_{\mathcal{D}}$ of $\mathcal{N}$ and $\mathcal{N}^s$ are prefix closed. Finally, observe that $\lozenge_{g \in \mathcal{G}}\, Sim_g$ has the property to be *closed by writes*. Formally, it is expressed as follows: whenever $\alpha\, w_c(g)\, \beta_1 \beta_2 \in (\lozenge_{g \in \mathcal{G}}\, Sim_g)$, we have $\alpha\, w_c(g)\, \beta_1\, w_c(g) \beta_2 \in (\lozenge_{g \in \mathcal{G}}\, Sim_g)$.

Example 4.4. Figure 5 gives the four simulators $Sim_1, Sim_2, Sim_3, Sim_{\$}$ for the contributor of Figure 2. Because $Sim_{\$} = \{\varepsilon\}$, we deliberately forget to mention it in subsequent developments. The fact that Sim_g is not necessarily prefix closed can be seen on its corresponding FSA, in which not all states are final. □

We are now ready to formulate the Simulation Lemma.

Lemma 4.5 (Simulation Lemma). *Let $\mathcal{D}$ and $\mathcal{C}$ be a leader and a contributor, and let $\{Sim_g \mid g \in \mathcal{G}\}$ be the set of simulators defined in Definition 4.3. We have that*

$$\big(\mathcal{D} \parallel Proj_{\mathcal{D}}(\mathcal{S} \parallel \lozenge_{\infty}\, \mathcal{C})\big) \;=\; \big(\mathcal{D} \parallel\; Proj_{\mathcal{D}}(\mathcal{S} \parallel \lozenge_{g \in \mathcal{G}}\, Sim_g)\big). \tag{6}$$

Before proving the Simulation Lemma, we illustrate it by means of a couple of examples.

Example 4.6. Consider again the example of Figure 2, but with a different leader $\mathcal{D}'$ given by the prefix-closure of the word

$$u = w_d(1)\,(r_d(2)\, r_d(1)\, r_d(3))^{100}\, w_d(\$). \tag{7}$$

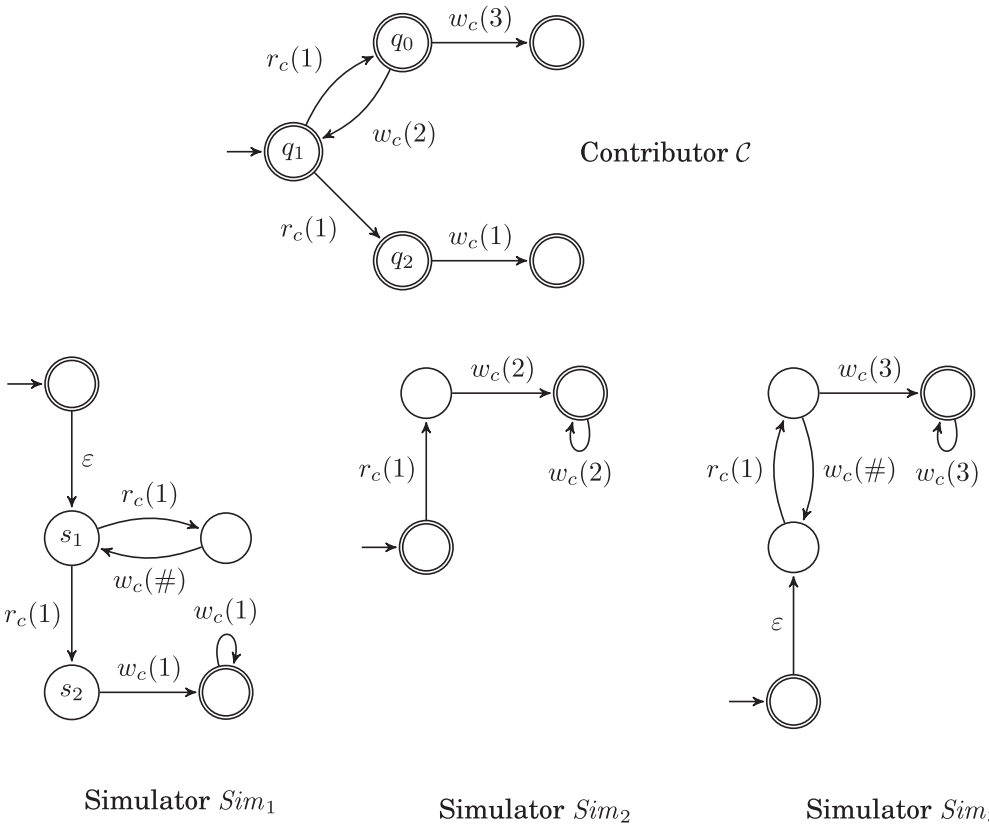


Fig. 5. Simulators for the nonatomic network of Figure 2, represented as finite automata. Simulator $Sim_\$$ is given by $\{\varepsilon\}$, and is not depicted.

It is easy to see that in the new network the leader can execute u given at Equation (7). For example, the leader and 200 contributors can execute together

$$w_d(1)\, r_c(1)^{200}\, (w_c(2)\, r_d(2)\, w_c(1)\, r_c(1)\, r_d(1)\, w_c(3)\, r_d(3))^{100}\, w_d(\$),$$

whose projection onto $\Sigma_{\mathcal{D}}$ yields u. Note that, to execute the 300 contributor writes (100 times $w_c(1)$, 100 times $w_c(2)$ and 100 times $w_c(3)$) as required by u, 200 contributors suffices. By visiting state q_0 twice, the same contributor can perform $w_c(2)$ and $w_c(3)$. By the Simulation Lemma, the leader must also be able to execute u in the simulating network $(\mathcal{D}' \parallel \mathcal{S} \parallel (Sim_1 \between Sim_2 \between Sim_3))$, and this is indeed the case: taking

$$v_1 = r_c(1)\, w_c(1)^{100} \quad v_2 = r_c(1)\, w_c(2)^{100} \quad v_3 = r_c(1)\, w_c(3)^{100}$$

we have that $(u \parallel \mathcal{S} \parallel (v_1 \between v_2 \between v_3)) \neq \emptyset$.

Example 4.7. Let us discuss a slight variation of Example 4.6, in which the contributor transition from q_1 to q_2 executes a $w_c(2)$ instead of $r_c(1)$ and transition from q_0 to q_1 executes a $w_c(3)$ instead of $w_c(2)$. After this change, the leader and 101 contributors can execute together

$$w_d(1)\, r_c(1)\, (w_c(2)\, r_d(2)\, w_c(1)\, r_d(1)\, w_c(3)\, r_d(3))^{100}\, w_d(\$),$$

The fact that we need 99 contributors less than in Example 4.6 is due to the fact that 1 contributor will execute 100 times $w_c(3)$ and 100 contributors execute $w_c(2)$ and

$w_c(1)$. With the change, the simulators become as follows, Sim_3 remains the same; Sim_2 now consists of a single accepting state with a self-loop executing $w_c(2)$; and, in Sim_1, the transition from s_1 to s_2 executes a $w_c(\#)$ instead of $r_c(1)$. To avoid ambiguities, we denote those languages Sim'_i, $i = 1, 2, 3$.

Again, by the Simulation Lemma, the leader must also be able to execute u in the simulating network $\big(\mathcal{D}' \parallel \mathcal{S} \parallel (Sim'_1 \between Sim'_2 \between Sim'_3)\big)$, and this is indeed the case: taking

$$v'_1 = w_c(\#)\, w_c(1)^{100} \quad v'_2 = w_c(2)^{100} \quad v'_3 = r_c(1)\, w_c(3)^{100},$$

we have that $(u \parallel \mathcal{S} \parallel (v'_1 \between v'_2 \between v'_3)) \neq \emptyset$.

4.1. Proof of the Simulation Lemma

We start by giving a precise formulation of the Copycat Lemma discussed in the main text. It intentionally comes with no proof because we think it is trivial. To avoid ambiguities, let us recall that $\Sigma_{\mathcal{C}} = \mathcal{G}(w_c, r_c) \cup \{w_c(\#)\}$.

Lemma 4.8 (Copycat Lemma). *Let $v \in \Sigma^*_{\mathcal{D}}$ and let M be a multiset of words of $\Sigma^*_{\mathcal{C}}$. If v is compatible with M and v' is a prefix of some word of M, then v is also compatible with $M \oplus \{v'\}$ (the multiset obtained by adding one copy of v' to M).*

Recall that, in order to produce an unsafe word, contributors may have to help the leader to overcome the read obstacles in its path to the action $w_d(\$)$. Now, assume that a contributor executes a sequence of actions ending with a write $w_c(g)$. As a consequence of the Copycat Lemma, arbitrarily many contributors may execute the same sequence, but stopping short of $w_c(g)$. These contributors can then help the leader to overcome *any number* of future obstacles $r_d(g)$. This is the idea that we exploit to prove the Simulation Lemma.

We start the proof with a simple result, which we first illustrate by means of an example.

Lemma 4.9. *Let $u \in \Sigma^*_{\mathcal{D}}$ and let M be a multiset of words over $\Sigma^*_{\mathcal{C}}$. If u is compatible with M, then u is compatible with any multiset M' obtained by erasing any number of occurrences of actions of $\mathcal{G}(r_c) \cup \{w_c(\#)\}$ from the words of M.*

Proof. Erasing reads and corrupting writes by contributors does not affect the sequence of values written to the store and read by the leader, hence compatibility is preserved. □

Example 4.10. Consider the words

$$u = w_d(1)\, r_d(2)\, r_d(1) \quad v_1 = r_c(1)\, w_c(\#)\, w_c(1) \quad v_2 = r_c(1)\, w_c(2)\, r_c(1).$$

Then u is compatible with $\{v_1, v_2\}$, as shown by the sequence

$$u' = w_d(1) \underbrace{r_c(1)}_{1} \underbrace{r_c(1)}_{2} \underbrace{w_c(\#)}_{1} \underbrace{w_c(2)}_{2} r_d(2) \underbrace{w_c(1)}_{1} \underbrace{r_c(1)}_{2} r_d(1).$$

By Lemma 4.9, the word u is also compatible with the multiset $M' = \{v'_1, v'_2\}$, where $v'_1 = w_c(1)$ and $v'_2 = w_c(2)$, because they can be obtained from v_1 and v_2 by removing only reads and corrupting writes. We can, for instance, interleave u, v'_1, v'_2 into

$$u'' = w_d(1) \underbrace{w_c(2)}_{2} r_d(2) \underbrace{w_c(1)}_{1} r_d(1),$$

which is a legal sequence of reads and writes. □

We are now in a position to prove the Simulation Lemma.

Proof of the Simulation Lemma (Lemma 4.5) ($\subseteq$): We present simultaneously the proof and an example. Fix $y \in \Sigma_{\mathcal{D}}^*$ such that $y \parallel \mathcal{S} \parallel \between_\infty \mathcal{C} \neq \emptyset$. We show that $y \parallel \mathcal{S} \parallel \between_{g \in \mathcal{G}} Sim_g \neq \emptyset$.

Example: $y = w_d(2) r_d(1) r_d(1)$.

By assumption, there exists $x \in \mathcal{S} \parallel \between_\infty \mathcal{C}$ such that $y \parallel \mathcal{S} \parallel x \neq \emptyset$. So $x \in \mathcal{S} \parallel (x_1 \between \cdots \between x_K)$ for some $K \geq 0$.

Example:
$$\begin{cases} x \in (x_1 \between x_2) \text{ and } z \in (y \parallel \mathcal{S} \parallel x) \\ x = r_c(2)\, r_c(2)\, w_c(1)\, w_c(3)\, r_c(3)\, r_c(3)\, w_c(1) \\ x_1 = r_c(2)\, w_c(1)\, w_c(3)\, r_c(3) \\ x_2 = r_c(2)\, r_c(3)\, w_c(1) \\ z = w_d(2)\, r_c(2)\, r_c(2)\, w_c(1)\, r_d(1)\, w_c(3)\, r_c(3)\, r_c(3)\, w_c(1)\, r_d(1). \end{cases}$$

Define $g_1, \ldots, g_n$ to be the sequence of values written to the store by the contributors in the order of their first writes in x. Hence, we have that all values g_i are pairwise distinct. We also define $p_1, \ldots, p_n$ as the positions in x in which these writes happen; hence, $p_1 < \cdots < p_n$ and $(x)_{p_i} = w_c(g_i)$.

Example:
$$\begin{cases} g_1 = 1 \text{ and } g_2 = 3 \\ x = r_c(2)\, r_c(2)\, \underbrace{w_c(1)}_{p_1}\, \underbrace{w_c(3)}_{p_2}\, r_c(3)\, r_c(3)\, w_c(1) \\ x_1 = r_c(2)\, w_c(1)\, w_c(3)\, r_c(3) \\ x_2 = r_c(2)\, r_c(3)\, w_c(1). \end{cases}$$

Further define $id_1, \ldots, id_n$ to be temporarily assigned contributor identifiers such that id_i is the contributor writing g_i at position p_i in x, and x_{id_i} is the sequence that it executes.

Example: $id_1 = 1$ and $id_2 = 1$, that is, the first writes of both $w_c(1)$ and $w_c(3)$ are executed by contributor 1.

Next, we use the Copycat Lemma to show that without loss of generality we can make further assumptions on x. First, we can assume that all identifiers id_i are pairwise distinct. This is not the case in our example, where $id_1 = id_2 = 1$, and we show how to repair this. If $id_i = id_j$ for some $i < j$, we first add a new contributor that executes a copy of x_{id_i}. By the Copycat Lemma, adding this copycat preserves compatibility, hence nonemptiness. Then, we assign the first write to g_j to the copycat, which allows breaking of the equality $id_i = id_j$.

Example:
Add: $x_3 = r_c(2)\, w_c(1)\, w_c(3)\, r_c(3)$, copy of x_1.
Redefine: $x := r_c(2)\, r_c(2)\, r_c(2)\, \underbrace{w_c(1)}_{p_1}\, \underbrace{w_c(3)}_{p_2}\, w_c(3)\, r_c(3)\, r_c(3)\, w_c(1)$.
We have: $x \in (x_1 \between x_2 \between x_3)$.
Assign: first $w_c(1)$ of x to x_1, first $w_c(3)$ of x to x_3.
Now: $id_1 = 1$ and $id_2 = 3$.

Next, we delete from x the following contributor reads:

(1) for every $i = 1, \ldots, n$, all the reads of contributor id_i after position p_i;
(2) all the reads of all contributors not in $\{id_1, \ldots, id_n\}$.

Lemma 4.9 shows that nonemptiness is preserved, that is, $y \parallel \mathcal{S} \parallel x \neq \emptyset$ still holds.

Example: We delete $r_c(3)$ in x_1 and x_3, and all reads of x_2.

$$x := r_c(2)\, r_c(2)\, \underbrace{w_c(1)}_{p_1}\, \underbrace{w_c(3)}_{p_2}\, w_c(3)\, w_c(1)$$
$$x_1 := r_c(2)\, w_c(1)\, w_c(3)$$
$$x_2 := w_c(1)$$
$$x_3 := r_c(2)\, w_c(1)\, w_c(3).$$

Now, we stepwise transform $x_{id_1}, \ldots, x_{id_n}$ further, so that at the end of the process, x_{id_i} contains only writes $w_c(g_i)$ and possibly $w_c(\#)$. Further, we ensure that each transformation preserves nonemptiness.

In a first step, for every contributor i different from $id_1, \ldots, id_n$, and for every $j = 1, \ldots, n$, we delete every occurrence of $w_c(g_j)$ in x_i, and append it to x_{id_j}. Since x_{id_j} executes the first occurrence of $w_c(g_j)$, this transformation preserves nonemptiness.

Example: We "transfer" the occurrence of $w_c(1)$ in x_2' to x_1'.

$$x_1 := r_c(2)\, w_c(1)\, w_c(3)\, w_c(1)$$
$$x_2 := \varepsilon$$
$$x_3 := r_c(2)\, w_c(1)\, w_c(3).$$

After this step, the only nonempty sequences are $x_{id_1}, \ldots, x_{id_n}$. Moreover, by the definition of $\mathcal{C}_{g_i}$, the prefix of x_{id_i} ending at position $p_i - 1$ belongs to $\mathcal{C}_{g_i}$.

For the second step, recall that contributor id_i executes the occurrence of $w_c(g_i)$ at position p_i. Consider the writes $w_c(g_j)$ in x_{id_i}, where $j \neq i$. We classify them into *early writes* (before the action at p_i) and *late writes* (after the action). Now, for each $i = 1, \ldots, n$ and every $j \neq i$, we proceed as follows:

(a) delete every late write $w_c(g_j)$ in x_{id_i}, and append it to x_{id_j}.
(b) replace every early write $w_c(g_j)$ by $w_c(\#)$ in x_{id_i}, and append it to x_{id_j}.

Example: We have $id_1 = 1$, and $id_2 = 3$. In x_1, there are no early writes, and $w_c(3)$ is the only late write. Thus, we delete $w_c(3)$, and add it to x_3.

$$x_1 := r_c(2)\, w_c(1)\, w_c(1)$$
$$x_3 := r_c(2)\, w_c(1)\, w_c(3)\, w_c(3).$$

Now, in x_3, there is now only an early write, namely, $w_c(1)$. Thus, we replace it by $w_c(\#)$, and add it to x_1.

$$x_1 := r_c(2)\, w_c(1)\, w_c(1)\, w_c(1)$$
$$x_3 := r_c(2)\, w_c(\#)\, w_c(3)\, w_c(3).$$

After (a), we have $x_{id_i} \in \mathcal{C}_{g_i} w_c(g_i)^+$. Indeed, all reads and all writes $w_c(g_j)$ have been removed, thus only $w_c(g_i)$s remain. After both (a) and (b), we have $x_{id_i} \in \mathcal{C}^{\#}_{g_i} w_c(g_i)^+$ (this follows immediately from the definition of $\mathcal{C}^{\#}_{g_i}$).

($\supseteq$): Let $u \in \mathcal{D} \parallel \mathit{Proj}_{\mathcal{D}}(\mathcal{S} \parallel \lozenge_{g \in \mathcal{G}} \mathit{Sim}_g)$. Then, there is a set $\{s_g \mid g \in \mathcal{G}\}$, where $s_g \in \mathit{Sim}_g$, that is compatible with u.

Let M be the multiset containing n_g copies of the word $v_g\ w_c(g)$ for each word s_g of the form $s_g = v_g\, w_c(g)^{n_g}$. Using a copycat-like argument, it is easy to see that M is compatible with u. Moreover, since the values written by a corrupting write are never read, u is also compatible with the multiset M' containing n_g copies of $u_g\ w_c(g)$. Finally,

since $u_g\, w_c(g) \in \mathcal{C}$, we have that M' is a multiset over $\{\!\!\{\}\!\!\}_\infty\ \mathcal{C}$; thus, we finally obtain $u \in \mathcal{D} \parallel \mathit{Proj}_{\mathcal{D}}(\mathcal{S} \parallel \{\!\!\{\}\!\!\}_{i=1}^{\infty}\ \mathcal{C})$. □

5. SAFETY(FSA, FSA) IS coNP-COMPLETE

We study the safety verification problem for FSAs, that is, we consider the problem Safety(FSA, FSA):

Given: FSAs D and C recognizing languages $\mathcal{D} = L(D)$ and $\mathcal{C} = L(C)$,
Decide: Is the network $(\mathcal{D} \parallel \mathcal{S} \parallel \{\!\!\{\}\!\!\}_\infty\ \mathcal{C})$ safe?

We prove that the problem is coNP-complete.

In order to state results about complexity, we must define the size of the input precisely. We define the size of an FSA $(Q, \mathcal{G}(r, w), \Delta, q_0, F)$ as the maximum between $|Q|$, $|\mathcal{G}(r, w)|$ and $|\Delta|$ where for any set S, the notation $|S|$ denotes the cardinality of S.

5.1. SAFETY(FSA, FSA) is coNP-Hard

We give a reduction from 3SAT to the complement of the safety verification problem. Intuitively, the leader uses the contributors to guess and store an assignment, and then checks if each clause is satisfied. (Observe that the leader cannot store the assignment in its internal state, because then the number of states of D would be at least equal to the number of assignments, and so exponential in the size of the formula.) Since a variable may appear in several clauses, during the check phase the leader may have to ask the contributors several times which value it guessed. We design a protocol to ensure that it always receives the same answer.

THEOREM 5.1. Safety(FSA, FSA) *is coNP-hard.*

PROOF. By reduction from 3SAT to the complement of Safety(FSA, FSA). Given a 3SAT formula with n variables and m clauses, we construct in polynomial time two regular expressions D and C with languages $\mathcal{D}$ and $\mathcal{C}$, respectively, such that the leader of the $(\mathcal{D}, \mathcal{C})$-network writes \$ if and only if the formula is satisfiable. The regular expressions can be translated into FSAs in the standard way.

The leader uses the contributors to guess and store an assignment, then checks if each clause is satisfied. Each contributor stores one value of the assignment. More precisely, the regular expressions for D and C capture exactly the prefix closure of D' and C', respectively, where

$$\begin{aligned} D' &= \mathrm{Guess}_1; \cdots ; \mathrm{Guess}_n; \mathrm{Check}_1; \cdots ; \mathrm{Check}_m; w_d(\$) \\ C' &= \mathrm{Store}_1 + \cdots + \mathrm{Store}_n, \end{aligned}$$

where the regular expressions Guess_i, Store_i, and Check_j are described next, and we use ";" to clearly denote concatenation.

Guessing and retrieving truth values. During Guess_i, the leader guesses a value for variable x_i. Since x_i may appear in several clauses, during the checks it may ask the contributors several times which value it guessed. We use the following protocol to ensure that it always receives the same answer. We choose the following set $\mathcal{G}$ of store values:

$$\begin{aligned} \mathcal{G} \;=\; & \{\$\} \cup \{\mathtt{set}[\mathtt{x_i}], \mathtt{get}[\mathtt{x_i}] \mid \mathtt{i} = 1, \ldots, n\} \cup \\ & \{\mathtt{propose}[\mathtt{x_i} \mapsto \mathtt{j}],\ \mathtt{commit}[\mathtt{x_i} \mapsto \mathtt{j}],\ \mathtt{return}[\mathtt{x_i} \mapsto \mathtt{j}] \mid \mathtt{i} = 1, \ldots, n,\ \mathtt{j} = 0, 1\} \end{aligned}$$

and we set

$$\begin{aligned}\text{Guess}_i &= w_d(\texttt{set}[\texttt{x}_\texttt{i}]);\ \textstyle\sum_{\texttt{j}=0,1}\Big(r_d(\texttt{propose}[\texttt{x}_\texttt{i}\mapsto\texttt{j}]);\ w_d(\texttt{commit}[\texttt{x}_\texttt{i}\mapsto\texttt{j}])\Big)\\ \text{Store}_i &= r_c(\texttt{set}[\texttt{x}_\texttt{i}]);\quad \Big(\textstyle\sum_{\texttt{j}=0,1} w_c(\texttt{propose}[\texttt{x}_\texttt{i}\mapsto\texttt{j}])\Big);\\ &\qquad \textstyle\sum_{\texttt{j}=0,1}\Big(r_c(\texttt{commit}[\texttt{x}_\texttt{i}\mapsto\texttt{j}]);\Big(r_c(\texttt{get}[\texttt{x}_\texttt{i}]);w_c(\texttt{return}[\texttt{x}_\texttt{i}\mapsto\texttt{j}])\Big)^*\Big).\end{aligned}$$

Thus, in order to assign a value to x_i, the leader writes $\texttt{set}[\texttt{x}_\texttt{i}]$ to the global store. The contributors who read $\texttt{set}[\texttt{x}_\texttt{i}]$ propose a value for x_i by writing $\texttt{propose}[\texttt{x}_\texttt{i}\mapsto\texttt{j}]$ for some $j\in\{0,1\}$. The leader reads the store at some (nondeterministic) point, and reads the last write of the contributors, which proposes either 0 or 1. If it reads $\texttt{propose}[\texttt{x}_\texttt{i}\mapsto\texttt{0}]$ (the 1 case is identical), then it commits to setting x_i to 0 by writing $\texttt{commit}[\texttt{x}_\texttt{i}\mapsto\texttt{0}]$). Contributors who read $\texttt{commit}[\texttt{x}_\texttt{i}\mapsto\texttt{0}]$ move on to the next phase, in which they return 0 every time that they are asked the value of x_i. Contributors who miss the commit message are stuck. This protocol ensures that the leader and contributors can reach consensus on the value of x_i. Notice that an arbitrary number of contributors can participate and potentially overwrite each other, but x_i is still assigned one single value.

Checking clauses. The leader uses this protocol to "assign" nondeterministically chosen consensus values to each variable $x_1,\ldots,x_n$, and proceeds to check that the assignment satisfies the clauses. To check a clause, it gets the literals from the contributors and checks if the clause is satisfied. Assume, for instance, that the j-th clause is $x_2\vee\neg x_3$. Then

$$\begin{aligned}\text{Check}_j = w_d(\texttt{get}[\texttt{x}_2]);\ \big(\ & r_d(\texttt{return}[\texttt{x}_2\mapsto\texttt{1}])+\\ & r_d(\texttt{return}[\texttt{x}_2\mapsto\texttt{0}]);\ w_d(\texttt{get}[\texttt{x}_3]);\ r_d(\texttt{return}[\texttt{x}_3\mapsto\texttt{0}])\ \big).\end{aligned}$$

Suppose that the formula is satisfiable. Then, there is an execution of the protocol in which 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 $(\mathcal{D},\mathcal{C})$-network accepts. On the other hand, if the formula is unsatisfiable, then the leader never succeeds in checking all clauses and \$ never gets written. Note that the regular expressions D' and C' have size $O(m+n)$. Then, regular expressions are transformed into equivalent FSAs in linear time. Those are prefix closed by setting every state as accepting. Hence, we have obtained FSA representations equivalent to the regular expressions D and C, and we are done. □

Note that the FSAs for $\mathcal{D}$ and $\mathcal{C}$ provided by the reduction from SAT satisfy the following property: for every two transitions (q,a_1,q_1) and (q,a_2,q_2), if $q_1\neq q_2$, then a_1 and a_2 are read actions, and they read different values. We call such FSAs *deterministic*, because at every configuration of the network, at most one of the transitions of the FSA can occur. Thus, loosely speaking, all the nondeterminism necessary to solve SAT in polynomial time is provided by the inherent nondeterminism of interleaving, and not by the FSAs themselves.

5.2. Safety(FSA, FSA) is in coNP: Preliminary Approach

We now show the upper bound. By the Simulation Lemma, $\mathcal{N}=(\mathcal{D}\parallel\mathcal{S}\parallel\langle\!\rangle_\infty\mathcal{C})$ is unsafe if and only if the simulating network $\mathcal{N}^s=(\mathcal{D}\parallel\mathcal{S}\parallel\langle\!\rangle_{g\in\mathcal{G}}Sim_g)$ is unsafe. Moreover, by reducing safety to language emptiness (recall that this is done by keeping in $\mathcal{D}$ only those words ending with $w_d(\$)$), we further have the equivalence

$$\mathcal{N}=\emptyset \text{ if and only if } \mathcal{N}^s=\emptyset.$$

This immediately suggests the following nondeterministic algorithm for checking language nonemptiness:

(1) guess a word $u \in \mathcal{D}$ (it necessarily ends with $w_d(\$)$);
(2) guess a set of words $M = \{s_g \in Sim_g \mid g \in \mathcal{G}\}$;
(3) nondeterministically check that u is compatible with the set M by guessing a word $v \in (\Sigma_{\mathcal{D}} \cup \Sigma_{\mathcal{C}})^*$ and checking that $v \in (u \parallel \mathcal{S} \parallel \{\!|_{g\in\mathcal{G}} s_g)$.

The number of words to guess is $|\mathcal{G}| + 1$. Since we assume that every value of $\mathcal{G}$ appears in some transition of $\mathcal{D}$ or $\mathcal{C}$ (otherwise, that value can be removed), the number of words to be guessed is linear in the size of the FSAs D and C. For example, for the network of Figure 2, we get the simulators of Figure 5; thus, we need to guess four words (in Sim_1, Sim_2, Sim_3, and $Sim_\$$). Further, by the definition of the g-simulator (Definition 4.3), we can easily construct an FSA A_g recognizing Sim_g from the FSA C recognizing $\mathcal{C}$ by means of the following four steps:

(i) Compute an FSA for $\mathcal{C}_g = \mathcal{C}/\{w_c(g)\}$. It is well known that regular language is closed for the right quotient operation. Moreover, there exists a polynomial time algorithm that takes in input C and returns an FSA for $\mathcal{C}_g$.
(ii) Given the FSA of point (i), rename all occurrences of $\mathcal{G}(w_c)$ with $w_c(\#)$. The alphabet of the resulting FSA thus becomes $\mathcal{G}(r_c) \cup \{w_c(\#)\}$ and its language is $\mathcal{C}_g^{\#}$.
(iii) Given the FSA of point (ii) with language $\mathcal{C}_g^{\#}$, compute an FSA for $\mathcal{C}_g^{\#}\, w_c(g)^+$.
(iv) Finally, given the FSA of point (iii), compute an FSA for $\varepsilon + \left(\mathcal{C}_g^{\#}\, w_c(g)^+\right)$. This can be done, for instance, by adding a new (initial and also final) state q whose unique transition is labeled with ε and leads to the initial state of the FSA for $\mathcal{C}_g^{\#}$.

Applying these steps to the FSA for $\mathcal{C}$ shown in Figure 2 yields the FSAs shown in Figure 5. Clearly, these FSAs have linear size in the size of C.

However, the algorithm given by Steps (1) through (3) is not necessarily polynomial, because both u and the words of M could be arbitrarily long. To solve these problems, we first have a closer look at the words of M. We prove the Contributor Monotonicity Lemma, a general lemma valid for arbitrary languages, showing that we can pick s_g as a minimal word according to a certain order.

5.3. Contributor Monotonicity Lemma

Before stating the lemma, we need some language-theoretic definitions.

Let Σ be an alphabet and $u, v \in \Sigma^*$. We say that u is a (scattered) subword of v, denoted $u \preceq v$, if u can be obtained from v by deleting some occurrences of symbols. Thus, for instance, $ab \preceq bacb$. Given $\Sigma' \subseteq \Sigma$, we say that u is a (scattered) Σ'-subword of v, denoted $u \preceq_{\Sigma'} v$, if u can be obtained from v by deleting some occurrences of symbols that do *not* belong to Σ', but leaving all symbols from Σ'. Equivalently, $u \preceq_{\Sigma'} v$ if $u \preceq v$ and $Proj_{\Sigma'}(u) = Proj_{\Sigma'}(v)$. In particular, we have $u \preceq v$ if and only if $u \preceq_{\emptyset} v$.

Example 5.2. Let $\Sigma = \{a, b, c\}$ and $\Sigma' = \{a\}$. We have $ab \preceq bacb$, $ab \preceq_{\Sigma'} bacb$, $b \preceq bacb$, and $b \not\preceq_{\Sigma'} bacb$.

Given two languages $L, L' \subseteq \Sigma^*$, we say that L' is a *support* of L if $L' \subseteq L$ and for every $u \in L$, there is $v \in L'$ such that $v \preceq u$. The notion of Σ'-support is defined similarly, replacing $\preceq$ by $\preceq_{\Sigma'}$. If $\Sigma' = \{a\}$, then we often write $\preceq_a$ and a-support instead of $\preceq_{\{a\}}$ and $\{a\}$-support. Observe that, for every $u, v \in L'$ such that $u \prec v$, if L' is a support, then so is $L' \setminus \{v\}$.

Example 5.3. For example, $\{b, ab\}$ is a support of $L = \{a^n b^m \mid 0 \le n \le 5, 1 \le m \le 7\}$, but $\{\varepsilon\}$ is not, because $\varepsilon \notin L$.

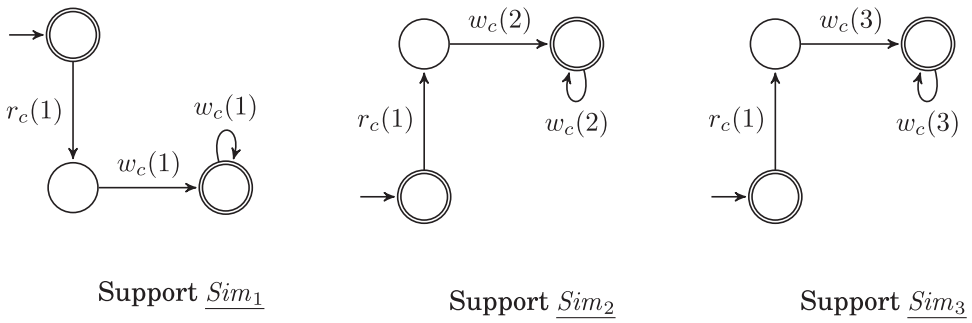


Fig. 6. Supports of the simulators of Figure 5, represented as finite automata.

In Figure 5, we have $Sim_1 = (\varepsilon + ((r_c(1)\, w_c(\#))^*\, r_c(1)\, w_c(1)^+))$, which has several supports, including $\{\varepsilon\}$ and $\{\varepsilon, r_c(1)w_c(1)\}$. Note that each $w_c(1)$-support must include $\{r_c(1)\, w_c(1)^i \mid i > 0\}$.

We can now state the Contributor Monotonicity Lemma. Intuitively, it states that the leader cannot distinguish the simulator Sim_g from a $w_c(g)$-support of it.

Lemma 5.4 (Contributor Monotonicity Lemma). *For every $g \in \mathcal{G}$, let $\underline{Sim_g}$ be a $w_c(g)$-support of Sim_g. We have that*

$$\Big(\mathcal{D} \parallel Proj_{\mathcal{D}}\big(\mathcal{S} \parallel \lozenge_{g\in\mathcal{G}}\, Sim_g\big)\Big) \;=\; \Big(\mathcal{D} \parallel Proj_{\mathcal{D}}\big(\mathcal{S} \parallel \lozenge_{g\in\mathcal{G}}\, \underline{Sim_g}\big)\Big)$$

Proof. ($\supseteq$): By the definition of support, we have $\underline{Sim_g} \subseteq Sim_g$, which immediately implies the result.

($\subseteq$): Let $u \in \mathcal{D} \parallel Proj_{\mathcal{D}}\big(\mathcal{S} \parallel \lozenge_{g\in\mathcal{G}}\, Sim_g\big)$. Then u is compatible with some set $\{s_g \in Sim_g \mid g \in \mathcal{G}\}$. By the definition of $w_c(g)$-support, $\underline{Sim_g}$ contains a word s'_g such that either $s'_g = \varepsilon = s_g$, or s'_g is obtained from s_g by erasing reads and corrupting writes. Since erasing these symbols does not affect compatibility (Lemma 4.9), u is compatible with $\{s'_g \mid g \in \mathcal{G}\}$, and we are done. □

Example 5.5. Figure 6 shows three possible supports for the simulators of Figure 5. Observe that the languages of the supports are of the form $(\varepsilon + L\, w_c(g)^+)$, where L is a finite language. In particular, if $\mathcal{C}$ is the language of a state machine with n states, then we can always find a support for Sim_g of the form $(\varepsilon + L\, w_c(g)^+)$ such that the words of L have at most length $n-1$. In other words, in this case, we can replace the simulator by a "faster" support that starts writing g after at most $(n-1)$ steps.

5.4. Membership in coNP

We now combine the proof outline in Section 5.2 with the Contributor Monotonicity Lemma to derive an upper bound for the safety verification problem for FSAs.

Theorem 5.6. `Safety(FSA, FSA)` *is in coNP.*

Proof. Let D and C be FSAs recognizing $\mathcal{D}$ and $\mathcal{C}$. Applying Steps (i) through (iv) (see Section 5.2) to $\mathcal{C}$, we obtain for every $g \in \mathcal{G}$ an FSA A_g recognizing Sim_g. One of the accepting states is the initial state, while each of the others has an input transition labeled by $w_c(g)$, and a self-loop labeled by $w_c(g)$ as unique output transition (see Figure 5). Further, these are the only transitions labeled by $w_c(g)$.

Recall the nondeterministic procedure of Section 5.2:

(1) guess a word $u \in \mathcal{D}$ (remember that such words necessarily end with $w_d(\$)$);
(2) guess a set of words $M = \{s_g \in Sim_g \mid g \in \mathcal{G}\}$;
(3) nondeterministically check that u is compatible with the set M by guessing a word $v \in (\Sigma_{\mathcal{D}} \cup \Sigma_{\mathcal{C}})^*$ and checking that $v \in (u \parallel \mathcal{S} \parallel \between_{g\in\mathcal{G}} s_g)$.

By the Contributor Monotonicity Lemma, in Step (2), we can restrict the guess of M to words s_g in a $w_c(g)$-support $\underline{Sim_g}$ of Sim_g. To profit from this, we choose $\underline{Sim_g}$ as the words accepted by runs of A_g that visit every nonfinal state at most once. It is easy to see that, by the properties of A_g described earlier, this is indeed a $w_c(g)$-support. In our example, these supports are recognized by the FSAs shown in Figure 6. It follows that we can safely limit the guess of Step (2) to words s_g of the form $s_g = s'_g\, w_c(g)^{n_g}$, where $s'_g \in \mathcal{C}^{\#}_g$, $|s'_g|$ is bounded by the number of states of C (if $s_g = \varepsilon$, then $s'_g = \varepsilon$ and $n_g = 0$). Thus, intuitively, we can always choose s_g as the concatenation of a short head and a possibly very long but very simple tail, consisting of n_g repetitions of $w_c(g)$.

However, the nondeterministic procedure still has two problems. The word u could be arbitrarily long, thus not "guessable" in polynomial time; the same could hold for the numbers n_g. In the rest of the proof, we show that, as a matter of fact, we do not have to guess either n_g or u. We reason as follows: if $\mathcal{N}$ is unsafe, then there is a word $u \in \mathcal{D}$ compatible with a set $\{s'_g\, w_c(g)^{n_g} \mid g \in \mathcal{G}\}$, where $s'_g \neq \varepsilon$ implies $n_g > 0$. Notice that s'_g, which belongs to $\mathcal{C}^{\#}_g$, is over alphabet $\mathcal{G}(r_c) \cup \{w_c(\#)\}$. Thus, s'_g contains no occurrences of $w_c(g)$ and there is a word $s \in \between_{g\in\mathcal{G}} s'_g\, w_c(g)^{n_g}$ such that $(u \parallel \mathcal{S} \parallel s) \neq \emptyset$. Let s' be the result of removing from s, for every $g \in \mathcal{G}$, all occurrences of $w_c(g)$ but the first one. We call s' the *skeleton* of s. Now, we replace this procedure with the following:

(a) Guess a subset $\mathcal{G}'$ of $\mathcal{G}$ and guess a set of words $\{s'_g \in \mathcal{C}^{\#}_g \mid g \in \mathcal{G}', |s'_g| \leq |C|\}$. Intuitively, the set $\mathcal{G}'$ contains all values written at least once by the contributors to the global store along some run leading to a write $w_d(\$)$. This step can be performed in quadratic time, because every s'_g has linear length in $|C|$, and $\mathcal{G}$ has linear size in $|C| + |D|$.
(b) Guess a skeleton $s' \in \between_{g\in\mathcal{G}'} (s'_g\, w_c(g))$. This can also be done in quadratic time.
(c) Construct an FSA recognizing the following language:

$$Sk(s') = \begin{cases} \{(s')_{1..i_1}\}\big((s')^*_{i_1} \between Sk((s')_{(i_1+1)..|s'|})\big), & \text{where } i_1 \text{ is the least position } (s')_{i_1} \in \mathcal{G}(w_c) \\ \{s'\} & \text{if } (s')_i \in \mathcal{G}(w_c) \text{ for no } i. \end{cases}$$

For this, it suffices to construct the obvious FSA that accepts only the word s', and add $w_c(g)$-self-loops whenever the FSA read $w_c(g)$ before. (See Example 5.7, to follow). The FSA has quadratic size in $|C| + |D|$, and can be constructed in quadratic time. Intuitively, Sk exploits the closed-by-write property of simulators: once a value is written to the store (the write at $(s')_{i_1}$), it is subsequently available arbitrarily many times (the language $(s')^*_{i_1}$). Note that, by the guesses in Steps (a), (b), and (c), every word of $\between_{g\in\mathcal{G}} \underline{Sim_g}$ is covered $Sk(s')$ for some skeleton s'.

(d) Construct an FSA recognizing $\big(\mathcal{D} \parallel \mathcal{S} \parallel Sk(s')\big)$. To see that this can be achieved in polynomial time, observe first that there is an FSA S recognizing $\mathcal{S}$ with $\mathcal{G} \cup \{\#\}$ as set of states, and the following transitions: for every value $g \in \mathcal{G}$, transitions $(g, r_d(g), g)$, $(g, r_c(g), g)$, and $(g, w_c(\#), \#)$; for every two values $g \in \mathcal{G}$ and $g' \in \mathcal{G} \cup \{\#\}$, transitions $(g, w_d(g'), g')$ and $(g, w_c(g'), g')$. All states are accepting and # is initial. We then apply the standard asynchronous product construction of two regular languages; the construction takes FSAs with n and m states, and returns an FSA for the asynchronous product with $O(nm)$ states.
(e) Check language nonemptiness for the FSA obtained at Step (d).

This procedure runs in nondeterministic polynomial time and succeeds whenever $\mathcal{N}^s$ is unsafe. □

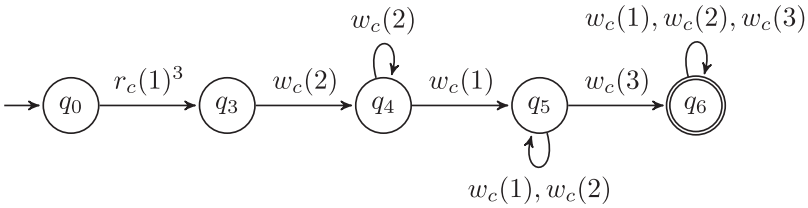
Example 5.7. Consider again the network of Figure 2. We apply Steps (a) through (e) to show that it is nonsafe. At Step (a), we guess words of Sim_1, Sim_2, Sim_3:

$$\begin{aligned} s'_1 &= r_c(1)\, w_c(1) \\ s'_2 &= r_c(1)\, w_c(2) \\ s'_3 &= r_c(1)\, w_c(3) \end{aligned}$$

At Step (b), we guess the skeleton

$$s' = r_c(1) r_c(1) r_c(1)\, w_c(2)\, w_c(1)\, w_c(3).$$

At Step (c), we construct an FSA for $\big(\mathcal{D} \parallel \mathcal{S} \parallel \mathit{Sk}(s')\big)$ by adding self-loops to the straight-line FSA for s':



Finally, at Steps (d) and (e), we apply the asynchronous product construction to this FSA, the FSA for $\mathcal{D}$ of Figure 2, and the 5-state automaton for $\mathcal{S}$. The resulting FSA is nonempty. Let us see that, for instance, it accepts the word

$$w_d(1) r_c(1)^3\, w_c(2) r_d(2)\, w_c(1) r_d(1)\, w_c(3) r_d(3)\, w_d(\$). \tag{8}$$

Indeed, (8) is a legal sequence of reads and writes, thus it belongs to $\mathcal{S}$; its projection onto the actions of the leader is accepted by $\mathcal{D}$, thus it belongs to $(\mathcal{D} \parallel \mathcal{S})$; finally, its projection onto the actions of the contributors is equal to the skeleton s' (thus, in this case, we have $s = s'$), thus it belongs to $\big(\mathcal{D} \parallel \mathcal{S} \parallel \mathit{Sk}(s')\big)$.

5.5. A Useful Intuition

Let $\Sigma = \{a_1, \ldots, a_n\}$ be an alphabet. Then, there is an FSA of size polynomial in n that recognizes the language $a_1^* \parallel \ldots \parallel a_n^*$: the FSA has a single (initial as well as final) state, and self-loops for each letter. However, consider the slightly modified language

$$L^+ = a_1^+ \parallel \ldots \parallel a_n^+, \tag{9}$$

where $a_i^+ = a_i a_i^*$ denotes one or more occurrences of a_i for each i. The smallest FSA for L^+ is exponential in n; intuitively, the FSA must remember the subset of Σ that has been seen and check at the end that all letters in Σ have been seen. Now, fix any linear ordering $<$ on Σ, and consider the subset $L^+_<$ of L^+, in which we additionally require that the first occurrences of letters in a word satisfy the ordering $<$. With this stipulation, there is an FSA of size polynomial in n for this language. For example, if $a_1 < a_2 < \cdots < a_n$, then the FSA recognizes $a_1 a_1^* a_2 (a_1 + a_2)^* \ldots a_n (a_1 + \ldots a_n)^*$. Moreover, the union of these languages over of all orderings of Σ is exactly the language L^+.

In the following proofs, we often require constructing FSAs for languages similar to L^+ and checking that their intersection with some other language is nonempty. Instead of directly constructing the exponentially large FSA, we use nondeterminism to guess an ordering of first occurrences, and check that the corresponding intersection with $L^+_<$ is nonempty. For example, in Theorem 5.6, the skeleton that we guess fixes the ordering

of first occurrences of writes. Similar constructions appear in more explicit form in the following sections.

6. SAFETY(PDA, FSA) IS coNP-COMPLETE

We consider the case in which only the leader is a PDA, while the contributors are FSAs. The proof is a rather straightforward adaptation of the proof of Theorem 5.6, and serves as a gentle introduction to the more involved cases of the next sections.

We first recall basic definitions and notations on PDAs. A (read/write) PDA is a 7-tuple $P = (\mathcal{Q}, \mathcal{G}(r,w), \Gamma, \Delta, Z_0, q_0, F)$, where $\mathcal{Q}$ is a finite set of *states* including the *initial state* q_0 and a set $F \subseteq \mathcal{Q}$ of accepting states, Γ is a *stack alphabet* that contains the *initial stack symbol* Z_0, and $\Delta \subseteq (\mathcal{Q} \times \Gamma) \times (\mathcal{G}(r,w) \cup \{\varepsilon\}) \times (\mathcal{Q} \times \Gamma^*)$ is a set of *rules*. A *configuration* of PDA P is a triple $(q, w, \alpha) \in \mathcal{Q} \times \mathcal{G}(r,w)^* \times \Gamma^*$, where q is the state of the PDA, w is the input tape and α is the pushdown tape. Let $\vdash_P$ be the relation on the configurations of P defined as follows: for $Z \in \Gamma$, $x \in \Sigma \cup \{\varepsilon\}$, we have $(q, xw, \alpha Z) \vdash_P (q', w, \alpha\gamma)$, called a *move*, if and only if $(q, Z, x, q', \gamma) \in \Delta$. Furthermore, let $\vdash_P^*$ denote the reflexive transitive closure of $\vdash_P$. A configuration (p, w, α) is *accepting* whenever $w = \varepsilon$ and $p \in F$. A sequence

$$(p_1, x_1 x_2 \cdots x_k\, w, \alpha_1) \vdash_P (p_2, x_2 \cdots x_k\, w, \alpha_2) \vdash_P \cdots \vdash_P (p_{k+1}, w, \alpha_{k+1})$$

of $k \geq 0$ moves is a *run* if $p_1 = q_0$ and $\alpha_1 = Z_0$. Furthermore, the *run* is said to be accepting if its last configuration is accepting. The *language* of a PDA P is the set $L(P) = \{w \in \mathcal{G}(r,w)^* \mid \exists q \in \mathcal{Q},\ \alpha \in \Gamma^* : (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha)$ and (q, ε, α) is accepting$\}$. In other words, $L(P)$ is the set of words for which there is an accepting run of P. The size of a rule $(q, \gamma, a, q', x) \in \Delta$ is defined as $|x| + 5$, and the size $|P|$ of a PDA as the sum of the size of rules in Δ.

THEOREM 6.1. Safety(PDA, FSA) *is in coNP.*

PROOF. Recall Steps (a) through (e) in the nondeterministic algorithm for the complement of Safety(FSA, FSA) in Theorem 5.6. We adapt this algorithm to a nondeterministic algorithm for the complement of Safety(PDA, FSA). Steps (a) through (c) do not change. Steps (d) and (e) are replaced by:

(d') Construct a PDA recognizing all words in $\big(\mathcal{D} \parallel \mathcal{S} \parallel Sk(s')\big)$. To see that this can be achieved in polynomial time, we apply a well-known result stating that the asynchronous product of a context-free language recognized by a PDA P and a regular language recognized by an FSA A is a context-free language, and a PDA for the asynchronous product can be constructed in polynomial time from P and A.

(e') Check language nonemptiness for the PDA obtained at Step (d'). Recall that PDA emptiness can be decided in polynomial time in the size of the PDA. □

7. SAFETY(FSA, PDA) IS coNP-COMPLETE

We show that even if all contributors (but not the leader) have access to a stack, the safety verification problem remains in coNP. The lower bound follows, of course, from Section 5.1.

As we saw in the last section, if the contributor C is an FSA, then we can solve the safety verification problem by guessing a sequence $s_g = s'_g\, w_c(g)^{n_g} \in Sim_g$ for every $g \in \mathcal{G}$, where the length of s'_g is bounded by the number of states of C. This worked because the set of these sequences constitutes a support of Sim_g; thus, we could apply the Contributor Monotonicity Lemma. However, this is no longer true if the contributor

is a PDA. In this case, the shortest sequence s'_g satisfying $s'_g\, w_c(g)^{n_g} \in Sim_g$ for some n_g may have exponential length in the size of the PDA.

Example 7.1. For every $n \geq 0$, let $\mathcal{C}_n$ consist of the prefixes of the word $(r_c(1)r_c(2))^{2^n} w_c(3)$. The language $\mathcal{C}_n$ is recognized by a PDA with $O(n)$ states, and we have, for each $n \geq 0$, a simulator $Sim_3 = (\varepsilon + (r_c(1)r_c(2))^{2^n} w_c(3)^+)$.

We thus need to apply a different strategy. The strategy uses two generic lemmas, valid for arbitrary networks, to which we devote the next two sections.

7.1. The Decomposition Lemma

Consider again the leader of Figure 2, consisting of the word

$$\underbrace{w_d(1)}_{1}\ \underbrace{r_d(2)}_{2}\ \underbrace{r_d(1)}_{3}\ \underbrace{r_d(3)}_{4}\ \underbrace{w_d(\$)}_{5}.$$

Given a sequence of the leader, its reads can be seen as obstacles that the leader must overcome in order to execute the sequence. To overcome the obstacles, the leader needs help in the form of contributor writes. The leader sequence can be "completed" with contributor writes in many different ways. For instance, the most "economic" completion of this sequence is

$$\underbrace{w_d(1)}_{1}\ w_c(2)\ \underbrace{r_d(2)}_{2}\ w_c(1)\ \underbrace{r_d(1)}_{3}\ w_c(3)\ \underbrace{r_d(3)}_{4}\ \underbrace{w_d(\$)}_{5},$$

but many other completions are possible, such as

$$\begin{array}{cccccccccccccc}
 & w_d(1) & w_c(3) & w_c(2) & r_d(2) & w_c(1)^5 & r_d(1) & w_c(3) & w_c(2) & w_c(3) & r_d(3) & w_d(\$) & & \\
w_c(2) & w_d(1) & w_c(2) & & r_d(2) & w_c(1) & r_d(1) & w_c(3) & & & r_d(3) & w_d(\$) & & \\
 & \underbrace{w_d(1)}_{1} & w_c(2) & & \underbrace{r_d(2)}_{2} & w_c(1) & \underbrace{r_d(1)}_{3} & w_c(3) & & & \underbrace{r_d(3)}_{4} & w_c(3) & \underbrace{w_d(\$)}_{5} & w_c(1)
\end{array}$$

In order to state the Decomposition Lemma, we need to define the set of all possible completions of the sequences of the leader. Observe that this is the set of sequences that obey the store rules and whose projection onto $\Sigma_{\mathcal{D}}$ yields a word of $\mathcal{D}$.

Definition 7.2. The *completion* of $\mathcal{D}$, denoted by $c\mathcal{D}$, is the set of sequences $u \in (\mathcal{G}(w_\star, r_d))^*$ such that $u \in \mathcal{S}$ and $Proj_{\mathcal{D}}(u) \in \mathcal{D}$.

The following proposition gives a characterization of $c\mathcal{D}$, and proves that replacing the leader by its completion does not change the behavior of a network.

PROPOSITION 7.3.

(1) For every leader $\mathcal{D}$:

$$c\mathcal{D} = Proj_{\mathcal{G}(w_\star, r_d)}(\mathcal{D} \parallel \mathcal{S}).$$

(2) For every $\mathcal{N}^s = (\mathcal{D} \parallel \mathcal{S} \parallel \ \mathop{\Vert}_{g \in \mathcal{G}} Sim_g)$:

$$\mathcal{N}^s = (c\mathcal{D} \parallel \mathcal{S} \parallel \ \mathop{\Vert}_{g \in \mathcal{G}} Sim_g).$$

PROOF. (1) We start with the left to right ($\subseteq$) inclusion.

$$\begin{array}{rl}
c\mathcal{D} = \{u \in \mathcal{G}(w_\star, r_d)^* \mid u \in \mathcal{S},\ Proj_{\mathcal{D}}(u) \in \mathcal{D}\} & \text{by definition} \\
\subseteq Proj_{\mathcal{G}(w_\star, r_d)}\{u \mid u \in \mathcal{S},\ Proj_{\mathcal{D}}(u) \in \mathcal{D}\} & \text{by property of } Proj
\end{array}$$

For the other direction ($\supseteq$), let $u \in \mathcal{S} \parallel \mathcal{D}$, that is, u is such that $u \in \mathcal{S}$ and $Proj_{\mathcal{D}}(u) \in \mathcal{D}$. Now, let $u' = Proj_{\mathcal{G}(w_\star, r_d)}(u)$. We want to show that $u' \in \mathcal{S}$ and $Proj_{\mathcal{D}}(u') \in \mathcal{D}$. Since

$Proj_{\mathcal{D}}(u) \in \mathcal{D}$ and $\Sigma_{\mathcal{D}} \subseteq \mathcal{G}(w_\star, r_d)$, we have that $Proj_{\mathcal{D}}(u') \in \mathcal{D}$. Let us turn to $u' \in \mathcal{S}$. We have that either $u' = u$ and we are done or that $u' \neq u$. Moreover the definition of *Proj* shows that u' results from u by deleting actions not in $\mathcal{G}(w_\star, r_d)$. These are necessarily contributor reads ($\mathcal{G}(r_c)$) or corrupting writes ($w_c(\#)$). Observe that deleting a read action does not alter the current value the store is holding, while deleting a corrupting write does not disable any action it can only enable reads. Therefore, u' is a legal sequence of actions complying with the store, that is, $u' \in \mathcal{S}$, and we are done.

(2) By the definition of completion, we have $Proj_{\mathcal{D}}(c\mathcal{D}) \subseteq \mathcal{D}$; thus, by Lemma 2.3(2) with $L = c\mathcal{D}$ and $L' = \mathcal{D}$, we get $(c\mathcal{D} \parallel \mathcal{D}) = c\mathcal{D}$. Now we have that

$$
\begin{aligned}
\mathcal{N}^s &= (\mathcal{D} \parallel \mathcal{S} \parallel \ \lozenge_{g\in\mathcal{G}}\, Sim_g) \\
&= \big((\mathcal{D} \parallel \mathcal{S}) \parallel Proj_{\mathcal{G}(w_\star,r_d)}(\mathcal{D} \parallel \mathcal{S}) \parallel \ \lozenge_{g\in\mathcal{G}}\, Sim_g\big) \quad \text{Lem. 2.3(1) with } L = (\mathcal{D} \parallel \mathcal{S}),\ \Sigma' = \mathcal{G}(w_\star, r_d) \\
&= \big((c\mathcal{D} \parallel \mathcal{D}) \parallel \mathcal{S} \parallel \ \lozenge_{g\in\mathcal{G}}\, Sim_g\big) \qquad \text{by Point (1)} \\
&= (c\mathcal{D} \parallel \mathcal{S} \parallel \ \lozenge_{g\in\mathcal{G}}\, Sim_g) \qquad ((c\mathcal{D} \parallel \mathcal{D}) = c\mathcal{D}) \quad \square
\end{aligned}
$$

Because they play an important role, let us clarify the alphabets involved in the equation $\mathcal{N}^s = (c\mathcal{D} \parallel \mathcal{S} \parallel \ \lozenge_{g\in\mathcal{G}}\, Sim_g)$. The language $\mathcal{N}^s$ is over $\Sigma_{\mathcal{D}} \cup \Sigma_C$, which coincides with $\mathcal{G}(w_d, r_d) \cup (\mathcal{G}(w_c, r_c) \cup \{w_c(\#)\})$. The alphabet of $c\mathcal{D}$ is $\mathcal{G}(w_\star, r_d)$, which includes $\Sigma_{\mathcal{D}}$. The alphabet of $\mathcal{S}$ is that of $\mathcal{N}^s$. Finally, for each $g \in \mathcal{G}$, the alphabet of Sim_g is given by $\mathcal{G}(r_c) \cup \{w_c(g), w_c(\#)\}$. We are now ready to state the Decomposition Lemma:

LEMMA 7.4 (DECOMPOSITION LEMMA). *For every $v \in c\mathcal{D}$:*

$$(v \parallel \mathcal{S} \parallel \ \lozenge_{g\in\mathcal{G}}\, Sim_g) \neq \emptyset \qquad \textit{if and only if} \qquad (v \parallel \mathcal{S} \parallel Sim_g) \neq \emptyset \quad \textit{for every } g \in \mathcal{G}.$$

The proof can be found in Appendix B. Here, we only provide some intuition and an illustrating example.

Example 7.5. Consider the simulating network with $\mathcal{G} = \{1, 2\}$ such that

$$
\begin{aligned}
\mathcal{D} &= w_d(1)\, w_d(2)\, r_d(1)\, r_d(2)\, w_d(\$) \\
Sim_1 &= \big(\varepsilon + r_c(1)\, w_c(\#)\, w_c(1)^+\big) \\
Sim_2 &= \big(\varepsilon + r_c(1)\, r_c(2)\, r_c(1)\, w_c(2)^+\big)
\end{aligned}
$$

Consider further the word

$$v = w_d(1)\, w_d(2)\, w_c(1)\, r_d(1)\, w_c(2)\, r_d(2)\, w_d(\$).$$

We have that $v \in c\mathcal{D}$. By definition of $c\mathcal{D}$, it is a word of $Proj_{\mathcal{G}(w_\star,r_d)}(\mathcal{S} \parallel \mathcal{D})$ which, as previously argued in Proposition7.3, $c\mathcal{D}$ complies with the store rules. The projection of v onto $\mathcal{G}(w_d, r_d)$, the alphabet of $\mathcal{D}$, yields the only word of $\mathcal{D}$. Further, we have that $\big(v \parallel \mathcal{S} \parallel (Sim_1 \lozenge Sim_2)\big) \neq \emptyset$. Let

$$
\begin{aligned}
u_1 &= r_c(1)\, w_c(\#)\, w_c(1) \\
u_2 &= r_c(1)\, r_c(2)\, r_c(1)\, w_c(2)
\end{aligned}
$$

be words of Sim_1 and Sim_2, respectively. The word

$$w_d(1)\, \underbrace{r_c(1)}_{1}\, \underbrace{r_c(1)}_{2}\, \underbrace{w_c(\#)}_{1}\, w_d(2)\, \underbrace{r_c(2)}_{2}\, \underbrace{w_c(1)}_{1}\, r_d(1)\, \underbrace{r_c(1)}_{2}\, \underbrace{w_c(2)}_{2}\, r_d(2)\, w_d(\$) \tag{10}$$

(where the underbraces indicate which of Sim_1 and Sim_2 executes which action) belongs to $\big(v \parallel \mathcal{S} \parallel (u_1 \lozenge u_2)\big)$. We have that $(v \parallel \mathcal{S} \parallel u_1) \neq \emptyset$ and $(v \parallel \mathcal{S} \parallel u_2) \neq \emptyset$. Indeed, the

words

$$w_d(1)\,\underbrace{r_c(1)}_{1}\,\underbrace{w_c(\#)}_{1}\,w_d(2)\,\underbrace{w_c(1)}_{1}\,r_d(1)\,w_c(2)\,r_d(2)\,w_d(\$) \tag{11}$$

$$w_d(1)\,\underbrace{r_c(1)}_{2}\,w_d(2)\,\underbrace{r_c(2)}_{2}\,w_c(1)\,r_d(1)\,\underbrace{r_c(1)}_{2}\,\underbrace{w_c(2)}_{2}\,r_d(2)\,w_d(\$) \tag{12}$$

obtained from Equation (10) by removing the actions executed by Sim_2 and Sim_1, respectively, belong to $(v \parallel \mathcal{S} \parallel u_1)$ and $(v \parallel \mathcal{S} \parallel u_2)$. Let us check that this is the case for Equation (11). First, note that Equation (11) obeys the store rules, that is, Equation (11) belongs to $\mathcal{S}$. Next, recall that the alphabets of $\mathcal{D}$, and Sim_1 are $\Sigma_{\mathcal{D}} = \mathcal{G}(w_d, r_d)$, and $\Sigma_1 = \mathcal{G}(r_c) \cup \{w_c(1), w_c(\#)\}$, respectively. Therefore, we get projections

$$\begin{aligned} &\text{onto } \Sigma_{\mathcal{D}}\colon \quad w_d(1)\,w_d(2)\,r_d(1)\,r_d(2)\,w_d(\$)\\ &\text{onto } \Sigma_1\colon \quad r_c(1)\,w_c(\#)\,w_c(1), \end{aligned}$$

which are words of $\mathcal{D}$, and Sim_1, respectively. The check for Equation (12) is similar.

Imagine now that we are given Equation (11) and Equation (12) as words of $(v \parallel \mathcal{S} \parallel u_1)$ and $(v \parallel \mathcal{S} \parallel u_2)$, and we are asked to construct a word of $\big(v \parallel \mathcal{S} \parallel (u_1 \between u_2)\big)$. For this, we first align the occurrences of the letters of v in Equation (11) and Equation (12), then construct the new word by "merging" the blocks between two consecutive occurrences:

$$\begin{array}{llllllll} w_d(1) & r_c(1)\,w_c(\#) & w_d(2) & w_c(1) & r_d(1) & w_c(2) & r_d(2) & w_d(\$)\\ w_d(1) & r_c(1) & w_d(2) & r_c(2) & w_c(1) & r_d(1) & r_c(1) & w_c(2) \quad r_d(2) \quad w_d(\$) \end{array}$$

$$w_d(1)\ \ r_c(1)\,r_c(1)\,w_c(\#)\ \ w_d(2)\ \ r_c(2)\ \ w_c(1)\quad r_d(1)\ \ r_c(1)\ \ w_c(2)\quad r_d(2)\ \ w_d(\$)$$

7.2. The Leader Monotonicity Lemma

Using the Decomposition Lemma, we can show $\mathcal{N}^s$ is unsafe by guessing a word $v \in c\mathcal{D}$ and checking that $v \parallel \mathcal{S} \parallel Sim_g$ is nonempty for every $g \in \mathcal{G}$. Unfortunately, the shortest v can be exponential in the input. Consider, for example, the family $\mathcal{D} = w_d(1)^* r_d(2)\,w_d(\$)$ and $\{\mathcal{C}(n)\}_{n\geq 0}$, where each $\mathcal{C}(n)$ consists of the prefixes of the word $r_c(1)^{2^n} w_c(2)$. The shortest v witnessing nonemptiness of $\mathcal{D} \parallel \mathcal{S} \parallel \between_\infty \mathcal{C}(n)$ has length $O(2^n)$. In order to get a coNP upper bound, we need another combinatorial argument that shows a monotonicity property for the leader.

A first attempt at monotonicity would suggest that if $v \in c\mathcal{D}$ is a witness for nonemptiness, $v' \in c\mathcal{D}$, and $v' \succeq v$, then v' is also a witness for nonemptiness. Unfortunately, this does not hold.

Example 7.6. Let $\mathcal{G} = \{1, \$\}$, and let

$$\mathcal{D} = w_d(1)\,w_d(\$) \qquad Sim_1 = (\varepsilon + r_c(1)\,w_c(1)^+) \quad Sim_\$ = \varepsilon + \emptyset = \{\varepsilon\}$$

Since $\mathcal{D}$ performs no reads, we have that

$$c\mathcal{D} = (w_d(1)\,w_d(\$)) \between w_c(1)^* \between w_c(\$)^* = \mathcal{G}(w_c)^*\,w_d(1)\,\mathcal{G}(w_c)^*\,w_d(\$)\,\mathcal{G}(w_c)^*.$$

Clearly, $w_d(1)\,w_d(\$) \parallel \mathcal{S} \parallel Sim_1 \between Sim_\$$ is nonempty. However, consider the larger string $w_c(1)\,w_d(1)\,w_d(\$)$. The language $(w_c(1)\,w_d(1)\,w_d(\$)) \parallel \mathcal{S} \parallel Sim_1 \between Sim_\$$ is empty, since no behavior of the simulator starts with a write.

The solution to this problem is to appropriately rename certain actions. More precisely, we want to distinguish the first time a contributor writes a value g to the store from all the other occurrences of $w_c(g)$.

Definition 7.7. Given a word u over any alphabet Σ including write actions for a set $\mathcal{G}' \subseteq \mathcal{G}$, that is $\mathcal{G}'(w_c)$, let $u[\mathtt{f}]$ be the word obtained as follows: for every $g \in \mathcal{G}$, rename the first occurrence of $w_c(g)$ in u (if any), to $f_c(g)$, where f_c is a new action standing for "first write" by a contributor (or simulator). Given a language L over an alphabet Σ including $\mathcal{G}(w_c)$, we define $L[\mathtt{f}]$ as the language over the alphabet $\Sigma \cup \mathcal{G}'(f_c)$ satisfying $L[\mathtt{f}] = \{u[\mathtt{f}] \mid u \in L\}$.

Example 7.8. For the word

$$u = w_d(1)\, r_c(1)^{200} \left(w_c(2)\, r_d(2)\, w_c(\#)\right)^{100}$$

we get

$$u[\mathtt{f}] = w_d(1)\, r_c(1)^{200}\, f_c(2)\, r_d(2)\, w_c(\#) \left(w_c(2)\, r_d(2)\, w_c(\#)\right)^{99}.$$

The following proposition states that renaming the first writes does not change the behavior of the network; it only provides more fine-grained information about them:

Proposition 7.9. *For every simulating network* $\mathcal{N}^s = (c\mathcal{D} \parallel \mathcal{S} \parallel \{\!|_{g\in\mathcal{G}} \mathit{Sim}_g)$*, we have* $\mathcal{N}^s[\mathtt{f}] = (c\mathcal{D}[\mathtt{f}] \parallel \mathcal{S}[\mathtt{f}] \parallel \{\!|_{g\in\mathcal{G}} \mathit{Sim}_g[\mathtt{f}])$.

Proof. Let $u[\mathtt{f}] \in \mathcal{N}^s[\mathtt{f}]$. By definition, we then have $u \in \mathcal{N}^s$; thus, there are words $v \in c\mathcal{D}$, $v' \in \mathcal{S}$, and $v_g \in \mathit{Sim}_g$ for every $g \in \mathcal{G}$ such that $u = (v \parallel v' \parallel \{\!|_{g\in\mathcal{G}}\, v_g)$. Since $w_c(g)$ belongs to the alphabets of $c\mathcal{D}$, $\mathcal{S}$, and Sim_g, we have that $\mathit{Proj}_{\mathcal{G}(w_c)}(v) = \mathit{Proj}_{\mathcal{G}(w_c)}(v') = \mathit{Proj}_{\mathcal{G}(w_c)}(\{\!|_{g\in G}\, v_g)$; thus, $u[\mathtt{f}] = (v[\mathtt{f}] \parallel v'[\mathtt{f}] \parallel \{\!|_{g\in\mathcal{G}}\, v_g[\mathtt{f}])$.

Let $u \in (c\mathcal{D}[\mathtt{f}] \parallel \mathcal{S}[\mathtt{f}] \parallel \{\!|_{g\in\mathcal{G}} \mathit{Sim}_g[\mathtt{f}])$. Because the alphabet of $\mathcal{N}^s[\mathtt{f}]$ coincides with that of $\mathcal{S}[\mathtt{f}]$, we find that $u \in \mathcal{S}[\mathtt{f}]$; thus, u contains at most one occurrence of $f_c(g)$, which comes moreover before any occurrence of $w_c(g)$. Thus, there is u' such that $u = u'[\mathtt{f}]$. It follows that $u' \in (c\mathcal{D} \parallel \mathcal{S} \parallel \{\!|_{g\in\mathcal{G}} \mathit{Sim}_g)$; thus, $u \in \mathcal{N}^s[\mathtt{f}]$. □

We can now state the Leader Monotonicity Lemma.

Lemma 7.10 (Leader Monotonicity Lemma). *Let* $L \subseteq (\Sigma_{\mathcal{C}})^*$ *be a language satisfying the closed-by-write condition: if* $\alpha\, w_c(g)\, \beta_1\beta_2 \in L$*, then* $\alpha\, w_c(g)\, \beta_1\, w_c(g)\, \beta_2 \in L$*. For every* $v, v' \in \mathit{Proj}_{\mathcal{G}(w_\star, r_d)}(\mathcal{S})[\mathtt{f}]$*:*

$$\text{if } v \parallel \mathcal{S}[\mathtt{f}] \parallel L[\mathtt{f}] \neq \emptyset \qquad \text{and } v' \succeq_{\mathcal{G}(f_c)} v, \qquad \text{then } v' \parallel \mathcal{S}[\mathtt{f}] \parallel L[\mathtt{f}] \neq \emptyset.$$

The proof is technical and can be found in Appendix C. Note that the word $w_c(1)\, w_d(1)\, w_d(\$)$ from Example 7.6 is ruled out by the lemma because $f_c(1)\, w_d(1)\, w_d(\$) \not\succeq_{\mathcal{G}(f_c)} w_d(1)\, w_d(\$)$.

7.3. Proof of the coNP Upper Bound

We now prove the coNP upper bound for Safety(FSA, PDA). Recall Proposition 7.3:

$$\mathcal{N}^s = (c\mathcal{D} \parallel \mathcal{S} \parallel \{\!|_{g\in\mathcal{G}} \mathit{Sim}_g).$$

We can show that $\mathcal{N}^s$ is unsafe by guessing an unsafe sequence $v \in c\mathcal{D}$ and checking that $(v \parallel \mathcal{S} \parallel \{\!|_{g\in\mathcal{G}} \mathit{Sim}_g) \neq \emptyset$. By the Decomposition Lemma, we can guess an unsafe sequence $v \in c\mathcal{D}$ and check for every $g \in \mathcal{G}$ that $(v \parallel \mathcal{S} \parallel \mathit{Sim}_g) \neq \emptyset$.

We cannot directly guess v (it can have exponential length) and must proceed differently. Instead of checking $(c\mathcal{D} \parallel \mathcal{S} \parallel \{\!|_{g\in\mathcal{G}} \mathit{Sim}_g) \neq \emptyset$, we consider the equivalent problem of checking

$$(c\mathcal{D}[\mathtt{f}] \parallel \mathcal{S}[\mathtt{f}] \parallel \{\!|_{g\in\mathcal{G}} \mathit{Sim}_g[\mathtt{f}]) \neq \emptyset.$$

Since both $\mathcal{D}$ and $\mathcal{S}$ are recognized by FSAs, we have that $c\mathcal{D}$, given by $Proj_{\mathcal{G}(w_\star, r_d)}(\mathcal{D} \parallel \mathcal{S})$, is also recognized by an FSA whose size is polynomial in the size of the FSAs for $\mathcal{D}$ and $\mathcal{S}$. Further, from a PDA C for $\mathcal{C}$, and for each $g \in \mathcal{G}$, we can compute a PDA for $Sim_g[\mathtt{f}]$ that is polynomial in the size of C. To see this, recall that $Sim_g = (\varepsilon + \mathcal{C}_g^{\#}\, w_c(g)^+)$, where $\mathcal{C}_g^{\#}$ is (a renaming of) the set of words $\mathcal{C}_g = \mathcal{C}/\{w_c(g)\}$. It is an exercise in automata theory to transform the PDA C recognizing $\mathcal{C}$ into a PDA P_g recognizing Sim_g, with linear blow-up.

Given FSAs cD and S recognizing $c\mathcal{D}$ and $\mathcal{S}$, respectively, and given a PDA P_g for each $g \in \mathcal{G}$ recognizing Sim_g, it is also easy to construct FSAs $cD[\mathtt{f}]$ and $S[\mathtt{f}]$ for $c\mathcal{D}[\mathtt{f}]$ and $\mathcal{S}[\mathtt{f}]$, and a PDA $P_g[\mathtt{f}]$ for $Sim_g[\mathtt{f}]$. Since

$$Sim_g[\mathtt{f}] = \varepsilon + \mathcal{C}_g^{\#}\, f_c(g)\, w_c(g)^*,$$

the new PDA $P_g[\mathtt{f}]$ has linear size in P_g.

However, $cD[\mathtt{f}]$ and $S[\mathtt{f}]$ are *exponentially larger* than cD and S. The reason is that, in both cases, the FSAs need to store the set of values $g \in \mathcal{G}$ for which there has already been a first write $f_c(g)$.

To get around this exponential blow-up, our nondeterministic algorithm guesses the sequence τ of first writes of the word v. That is, the algorithm guesses a subset of $\mathcal{G}$ and an ordering of this subset. For example, if $\mathcal{G} = \{1, 2, 3\}$, then there are 16 possibilities for τ: 6 sequences giving all possible permutations of $\{f_c(1), f_c(2), f_c(3)\}$, 6 sequences giving possible orderings $\{f_c(i)\, f_c(j) \mid i, j \in \{1, 2, 3\} \text{ and } i \neq j\}$, the 3 sequences $\{f_c(1), f_c(2), f_c(3)\}$, and finally, the sequence ε.

Since $|\tau| \leq |\mathcal{G}|$, it can be guessed in polynomial time. We can then compute in polynomial time an FSA $cD[\mathtt{f}]^\tau$ recognizing the words of $c\mathcal{D}[\mathtt{f}]$ whose projection onto $\mathcal{G}(f_c)$ is equal to τ. The same holds for $\mathcal{S}[\mathtt{f}]^\tau$. After this preprocessing, we are left with our new task: nondeterministically check in polynomial time

$$(c\mathcal{D}[\mathtt{f}]^\tau \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel \mathop{\Diamond}_{g \in \mathcal{G}} Sim_g[\mathtt{f}]) \neq \emptyset. \tag{13}$$

Using the Decomposition Lemma, we can perform this check one PDA Sim_g at a time. However, we cannot directly guess a $v \in c\mathcal{D}[\mathtt{f}]^\tau$, as such a witness can be exponentially long. Instead, we proceed as follows.

The algorithm guesses a *subautomaton* $E[\mathtt{f}]^\tau$ of $cD[\mathtt{f}]^\tau$ with language $\mathcal{E}[\mathtt{f}]^\tau$ (i.e., an FSA derived from $cD[\mathtt{f}]^\tau$ by deleting some states and transitions) satisfying the following properties:

(A) If Equation (13) holds, then $(\mathcal{E}[\mathtt{f}]^\tau \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel \mathop{\Diamond}_{g \in \mathcal{G}} Sim_g[\mathtt{f}]) \neq \emptyset$.
(B) For every two words $v_1, v_2 \in \mathcal{E}[\mathtt{f}]^\tau$, there is another word $v_3 \in \mathcal{E}[\mathtt{f}]^\tau$ such that $v_3 \succeq_{\mathcal{G}(f_c)} v_1$ and $v_3 \succeq_{\mathcal{G}(f_c)} v_2$. Since the projection of all words of $\mathcal{E}[\mathtt{f}]^\tau$ onto first writes is equal to τ, the condition $v_3 \succeq_{\mathcal{G}(f_c)} v_1$ and $v_3 \succeq_{\mathcal{G}(f_c)} v_2$ is equivalent to $v_3 \succeq v_1$ and $v_3 \succeq v_2$.

The algorithm then checks in polynomial time that, for each $g \in \mathcal{G}$, we have that $(\mathcal{E}[\mathtt{f}]^\tau \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel Sim_g[\mathtt{f}]) \neq \emptyset$ and uses the Leader Monotonicity Lemma to assemble a witness.

Before giving the technical details of the algorithm, the following lemma about FSAs shows that such a choice of $\mathcal{E}[\mathtt{f}]^\tau$ is always possible.

LEMMA 7.11. *For every FSA A, there exists a finite collection $A_1, \ldots, A_d$ of FSAs satisfying the following properties:*

(*1*) *each A_i is a subautomaton of A, that is, results from removing states and transitions from A;*
(*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 $v \succeq v_2$.*

Furthermore, given an arbitrary subautomaton A' of A, we can check in polynomial time that it belongs to the collection.

We show how Lemma 7.11 can be used to prove the main result of this section.

THEOREM 7.12. $\texttt{Safety(FSA, PDA)} \in$ coNP

PROOF. As in the preceding discussion, our starting point is the check

$$(c\mathcal{D}[\![\texttt{f}]\!] \parallel \mathcal{S}[\![\texttt{f}]\!] \parallel \mathbin{\text{⧢}}_{g\in\mathcal{G}} \mathit{Sim}_g[\![\texttt{f}]\!]) \neq \emptyset.$$

The algorithm nondeterministically guesses the (at most length $|\mathcal{G}|$) sequence τ of first writes and we are left with the following check:

$$(c\mathcal{D}[\![\texttt{f}]\!]^\tau \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathbin{\text{⧢}}_{g\in\mathcal{G}} \mathit{Sim}_g[\![\texttt{f}]\!]) \neq \emptyset. \tag{14}$$

As discussed earlier, the FSAs for $c\mathcal{D}[\![\texttt{f}]\!]^\tau$ and $\mathcal{S}[\![\texttt{f}]\!]^\tau$ are of size polynomial in the input, and for each $g \in \mathcal{G}$, the PDA $\mathit{Sim}_g[\![\texttt{f}]\!]$ is also polynomial in the input. All these machines can be computed in polynomial time in the input.

We discharge the check (14) in the following two steps.

(1) First, using Lemma 7.11, the algorithm guesses a subautomaton $E[\![\texttt{f}]\!]^\tau$ of $cD[\![\texttt{f}]\!]^\tau$ such that for every two words $v_1, v_2 \in \mathcal{E}[\![\texttt{f}]\!]^\tau$, there is another word $v_3 \in \mathcal{E}[\![\texttt{f}]\!]^\tau$ such that $v_3 \succeq_{\mathcal{G}(f_c)} v_1$ and $v_3 \succeq_{\mathcal{G}(f_c)} v_2$. Note that, since the projection of all words of $\mathcal{E}[\![\texttt{f}]\!]^\tau$ onto first writes is equal to τ, the condition $v_3 \succeq_{\mathcal{G}(f_c)} v_1$ and $v_3 \succeq_{\mathcal{G}(f_c)} v_2$ is equivalent to $v_3 \succeq v_1$ and $v_3 \succeq v_2$. Moreover, the guessed $E[\![\texttt{f}]\!]^\tau$ is such that if (14) holds, then $(\mathcal{E}[\![\texttt{f}]\!]^\tau \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathbin{\text{⧢}}_{g\in\mathcal{G}} \mathit{Sim}_g[\![\texttt{f}]\!]) \neq \emptyset$.

(2) Check for every $g \in \mathcal{G}$ that $(\mathcal{E}[\![\texttt{f}]\!]^\tau \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathit{Sim}_g[\![\texttt{f}]\!]) \neq \emptyset$. First, Lemma 2.3(1) shows that it is equivalent to check $\big(\mathcal{E}[\![\texttt{f}]\!]^\tau \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel (\mathit{Proj}_{\mathcal{C}}(\mathcal{S})[\![\texttt{f}]\!]^\tau \parallel \mathit{Sim}_g[\![\texttt{f}]\!])\big) \neq \emptyset$. Observe that the language $\mathit{Proj}_{\mathcal{C}}(\mathcal{S})[\![\texttt{f}]\!]^\tau \parallel \mathit{Sim}_g[\![\texttt{f}]\!]$ is over alphabet $\Sigma_{\mathcal{C}}$ and is closed by writes: if $\alpha\, w_c(g)\, \beta_1\beta_2 \in \mathcal{S}^\tau \parallel \mathit{Sim}_g$, then $\alpha\, w_c(g)\, \beta_1\, w_c(g)\, \beta_2 \in \mathcal{S}^\tau \parallel \mathit{Sim}_g$.

We sketch why this check can be performed in nondeterministic polynomial time in the input. The FSA $E[\![\texttt{f}]\!]^\tau$ can be computed in nondeterministic polynomial time by Lemma 7.11: guess a subautomaton and check in polynomial time that it belongs to the collection $\{A_1, \ldots, A_d\}$. There is an FSA for $\mathcal{S}$ and for $\mathit{Proj}_{\mathcal{C}}(\mathcal{S})$ with $O(|\mathcal{G}|)$ states; thus, since $|\tau| \leq |\mathcal{G}|$, we can construct FSAs $S[\![\texttt{f}]\!]^\tau$ and $S_C[\![\texttt{f}]\!]^\tau$ for $\mathcal{S}[\![\texttt{f}]\!]^\tau$ and $\mathit{Proj}_{\mathcal{C}}(\mathcal{S})[\![\texttt{f}]\!]^\tau$ with $O(|\mathcal{G}|^2)$ states in polynomial time. Finally, as mentioned earlier, we can construct a PDA P_g recognizing Sim_g from the PDA C recognizing $\mathcal{C}$, in polynomial time in C. Next, we can compute in polynomial time a PDA N recognizing $(\mathcal{E}[\![\texttt{f}]\!]^\tau \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathit{Proj}_{\mathcal{C}}(\mathcal{S})[\![\texttt{f}]\!]^\tau \parallel \mathit{Sim}_g[\![\texttt{f}]\!])$, since the asynchronous product of a PDA and a fixed number of FSAs can be computed in polynomial time in the size of the inputs. Finally, we can check the nonemptiness of N in polynomial time.

We prove that the algorithm is correct. It suffices to show: There is a word $v \in \mathcal{E}[\![\texttt{f}]\!]^\tau$ such that $(v \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathbin{\text{⧢}}_{g\in\mathcal{G}} \mathit{Sim}_g[\![\texttt{f}]\!]) \neq \emptyset$ if and only if $(\mathcal{E}[\![\texttt{f}]\!]^\tau \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathit{Sim}_g[\![\texttt{f}]\!]) \neq \emptyset$ for every $g \in \mathcal{G}$.

($\Rightarrow$): This part is identical to the argument for the Decomposition Lemma. Let $x \in (v \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathbin{\text{⧢}}_{g\in\mathcal{G}} \mathit{Sim}_g[\![\texttt{f}]\!])$. Then, there are words $u_g \in \mathit{Sim}_g[\![\texttt{f}]\!]$ for each $g \in \mathcal{G}$ that appear as scattered subwords in x and for which $x \in (v \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathbin{\text{⧢}}_{g\in\mathcal{G}} u_g)$. Further, the scattered subwords are disjoint for all $g' \neq g$. Now, for each g, take x_g to be the scattered subword of x obtained by removing all $u_{g'}$ for $g' \neq g$. Then $x_g \in (\mathcal{E}[\![\texttt{f}]\!]^\tau \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathit{Sim}_g[\![\texttt{f}]\!])$.

($\Leftarrow$): By hypothesis, there is a set $\{u_g \in \mathcal{E}[\![\texttt{f}]\!]^\tau \mid g \in \mathcal{G}\}$ of witnesses such that $(u_g \parallel \mathcal{S}[\![\texttt{f}]\!]^\tau \parallel \mathit{Sim}_g[\![\texttt{f}]\!]) \neq \emptyset$ for every $g \in \mathcal{G}$. It also follows from Lemma 2.3(1) that

$(u_g \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel Proj_{\mathcal{C}}(\mathcal{S})[\mathtt{f}]^\tau \parallel Sim_g[\mathtt{f}]) \neq \emptyset$ for every $g \in \mathcal{G}$. Property (B) of $\mathcal{E}[\mathtt{f}]^\tau$, and repeated applications of the Leader Monotonicity Lemma (Lemma 7.10) with $L = Proj_{\mathcal{C}}(\mathcal{S})[\mathtt{f}]^\tau \parallel Sim_g[\mathtt{f}]$ show that we can $\mathcal{G}(f_c)$-cover all the separate witnesses with one $v \in \mathcal{E}[\mathtt{f}]^\tau$ such that $v \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel Proj_{\mathcal{C}}(\mathcal{S})[\mathtt{f}]^\tau \parallel Sim_g[\mathtt{f}] \neq \emptyset$ for each $g \in \mathcal{G}$. For instance, let $u_{g_1}, u_{g_2} \in \mathcal{E}[\mathtt{f}]^\tau$ such that $(u_{g_i} \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel Sim_g[\mathtt{f}]) \neq \emptyset$ for $i = 1, 2$. Property (B) of $\mathcal{E}[\mathtt{f}]^\tau$ shows that there exists $u_{\hat{g}} \succeq_{\mathcal{G}(f_c)} u_{g_i}$ for $i = 1, 2$; hence, we find that $(u_{\hat{g}} \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel Sim_g[\mathtt{f}]) \neq \emptyset$ by Lemma 7.10. Coming back to $v \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel Proj_{\mathcal{C}}(\mathcal{S})[\mathtt{f}]^\tau \parallel Sim_g[\mathtt{f}] \neq \emptyset$, we find that $v \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel Sim_g[\mathtt{f}] \neq \emptyset$ for each $g \in \mathcal{G}$ following Lemma 2.3(1). We rewrite it as $v' \parallel \mathcal{S} \parallel Sim_g \neq \emptyset$ for each $g \in \mathcal{G}$, where v' is the result of replacing all actions in $\mathcal{G}(f_c)$ by the corresponding action in $\mathcal{G}(w_c)$. Since $v' \in c\mathcal{D}$, we can apply the Decomposition Lemma, which shows that $v' \parallel \mathcal{S} \parallel \between_{g \in G} Sim_g \neq \emptyset$ and finally that $c\mathcal{D} \parallel \mathcal{S} \parallel \between_{g \in G} Sim_g \neq \emptyset$. □

To conclude the proof, we prove Lemma 7.11.

Proof (of Lemma 7.11). Let π be an accepting run 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 strongly connected components (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 run 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). Thus, any two accepting runs in A_π must visit the sccs in that order. Therefore, it suffices to show that given an scc, any two words 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 an scc, all states are mutually reachable. □

8. SAFETY(PDA, PDA) IS PSPACE-COMPLETE

8.1. SAFETY(PDA, PDA) 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 acknowledgment 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 that 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.

Theorem 8.1. Safety(PDA, PDA) *is PSPACE-hard.*

Proof of Theorem 8.1. 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 $c_0 \to c_1 \to \cdots \to c_{2^n}$, 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 these assumptions, configurations of M can be encoded by words of fixed length.

We define a $(\mathcal{D}, \mathcal{C})$-network that simulates M and such that $\mathcal{D} = L(D)$ and $\mathcal{C} = L(C)$, where D and C are PDAs. The leader and the contributors cooperatively 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 PDA, 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 that 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.

Simulating one step of the TM. The network iteratively simulates one step of M as follows: the leader sends the current configuration of M to the contributors, then the contributors send back the next configuration to the leader. More precisely, assume that the reverse of a configuration of the Turing machine is stored as a word w of length n in the stack of the leader. 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. If the leader could be sure that there is a unique contributor, the task would be easy: the leader pops its stack, writing its contents one symbol at a time in the store, waiting after each step for an acknowledgement of the contributor. Then, it proceeds to read the new configuration from the store, also one symbol at a time, acknowledging after each read, and pushes the symbols into its stack. The contributor behaves symmetrically, with the difference that what it pushes into its stack is not the configuration sent by the leader, but a successor configuration, which it computes on-the-fly, as the symbols arrive. This procedure works independently of the length of the configuration; thus, the leader and its single contributor can simulate an arbitrary Turing machine.

However, in the presence of an unbounded number of contributors, the leader has no guarantee that all messages it reads have been sent by the same contributor, and that this contributor has read all previous messages from the leader. To solve this problem, 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 onto 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.

Note 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.

Note that, even in the presence of nonatomic 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 numbers of steps, participate in the protocol. The simulation catches these discrepancies by counting, as described here.

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

Then, the leader and contributors execute the protocol described earlier. 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.

Note that if the leader interacts with the same contributor for 2^n rounds, then both will simultaneously reduce the counter on the stack to \$ at the same time, thus would have correctly simulated the Turing machine for 2^n steps. Thus, 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; therefore, none leads to accepting the computation.

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

8.2. `Safety`(`PDA`, `PDA`) is in PSPACE

We sketch the upper PSPACE bound, which uses constructions on approximations of context-free languages. The details of the proof are available in Appendix A.

Let D and C be PDAs recognizing $\mathcal{D}$ and $\mathcal{C}$. By the Simulation Lemma, our task consists of deciding

$$(\mathcal{D} \parallel \mathcal{S} \parallel \between_{g\in\mathcal{G}} Sim_g) \stackrel{?}{=} \emptyset,$$

which, by Proposition 7.3, is equivalent to

$$(c\mathcal{D} \parallel \mathcal{S} \parallel \between_{g\in\mathcal{G}} Sim_g) \stackrel{?}{=} \emptyset. \tag{15}$$

Recall that $\mathcal{S}$ is regular language recognized by FSAs S with $O(|\mathcal{G}|)$ states. From D and S, we can construct in polynomial time (in $|D| + |S|$) a PDA cD recognizing $c\mathcal{D}$. Further, recall that $Sim_g = \varepsilon + \mathcal{C}_g^{\#}\, w_c(g)^+$, where $\mathcal{C}_g^{\#}$ is (a relabeling of) the set of words u containing no occurrence of $w_c(g)$ and satisfying $u\, w_c(g) \in \mathcal{C}$. One can transform the

PDA for $\mathcal{C}$ into a PDA for $\mathcal{C}_g^{\#}$ with linear blow-up, and this into a PDA P_g for Sim_g. Thus, Equation (15) is equivalent to

$$(L(cD) \parallel L(S) \parallel \ \between_{g\in\mathcal{G}}\ L(P_g)) \stackrel{?}{=} \emptyset,$$

which suggests the following naïve nondeterministic algorithm:

—Guess step by step a sequence

$$\rho = \ \Gamma_0 \stackrel{a_0}{\Rightarrow} \Gamma_1 \stackrel{a_1}{\Rightarrow} \cdots \stackrel{a_{n-1}}{\Longrightarrow} \Gamma_n,$$

where, for each $0 \leq i \leq n$, a_i is an action, and Γ_i is a triple consisting of a value g, a configuration γ_d of cD, and a set $\{\gamma_g \mid g \in \mathcal{G}\}$ of configurations of the PDAs P_g.

—At each step, check that Γ_{i+1} is a successor of Γ_i under the action a_i according to the network semantics.

However, since storing a configuration of a PDA may require an arbitrary amount of memory, there is no guarantee that the algorithm runs in (nondeterministic) polynomial space.

In the following, we show how we refine this algorithm to get a PSPACE upper bound. This requires several technical ingredients.

Ingredient 1: Bounding Stacks of the Contributors. The problem concerning the configurations of the P_g can be solved by resorting to the Contributor Monotonicity Lemma. In Appendix A, we prove the following result (Theorem A.3):

(Support Theorem) Given a PDA P with a stack alphabet of size k recognizing a context-free language L, let $L^k \subseteq L$ be the set of words of L that are accepted by P by means of a run in which the size of the stack never exceeds $O(k)$. Then, L^k is a support of L.

We cannot directly combine this result with the Contributor Monotonicity Lemma, because the latter is only applicable to $w_c(g)$-supports. However, recall that we have $Sim_g = \varepsilon + \mathcal{C}_g^{\#}\, w_c(g)^+$, where the words of $\mathcal{C}_g^{\#}$ contain no occurrences of $\mathcal{G}(w_c)$. It is easy to see that, for every support $\underline{\mathcal{C}_g^{\#}}$ of $\mathcal{C}_g^{\#}$, the set $\underline{Sim_g} = \varepsilon + \underline{\mathcal{C}_g^{\#}}\, w_c(g)^+$ is a $w_c(g)$-support of Sim_g. Let $C_g^{\#}$ be a PDA of polynomial size in $|C|$ recognizing $\mathcal{C}_g^{\#}$, and let $\underline{\mathcal{C}_g^{\#}}$ be the subset of words of $\mathcal{C}_g^{\#}$ accepted by runs of $C_g^{\#}$ with a linearly bounded stack. Then $\underline{\mathcal{C}_g^{\#}}$ is a support of $\mathcal{C}_g^{\#}$; thus, $\underline{Sim_g} = \varepsilon + \underline{\mathcal{C}_g^{\#}}\, w_c(g)^+$ is a $w_c(g)$-support of Sim_g. Further, since $C_g^{\#}$ and P_g are very similar PDAs, it is easy to see that $\underline{Sim_g}$ is also recognized by runs of P_g with a linearly bounded stack. Applying the Contributor Monotonicity Lemma, we obtain that (15) is equivalent to checking

$$(c\mathcal{D} \parallel \mathcal{S} \parallel \between_{g\in\mathcal{G}}\ Sim_g^N) \stackrel{?}{=} \emptyset \tag{16}$$

where N is the number of stack symbols of Sim_g, which is bounded by some polynomial in $|C|$. (To be precise, N depends on g, but to avoid clutter in the notation, let us take the maximum over all $g \in \mathcal{G}$.) Thus, it suffices to guess a sequence ρ in which the stacks of the simulators are linearly bounded (in $|C|$).

Ingredient 2: Generalized Leader Monotonicity Lemma. The second problem with the naïve algorithm is that the stack of the leader can be unbounded. Here, we need a second technical ingredient: a generalization of the Leader Monotonicity Lemma

(Lemma 7.10) that allows replacing the language of the leader by a related language, its *cover*.

Given two languages L, L' over the same alphabet Σ, we say that L' is a *cover* of L if $L' \subseteq L$ and for every $u \in L$, there is $v \in L'$ such that $u \preceq v$, where $\preceq$ is the subword ordering. Further, given $\Sigma' \subseteq \Sigma$, we say that L' is a Σ'-*cover* of L if $L' \subseteq L$ and for every $u \in L$, there is $v \in L'$ such that $u \preceq_{\Sigma'} v$, where $\preceq_{\Sigma'}$ is the Σ'-subword ordering, in which we are only allowed to remove symbols in $\Sigma \setminus \Sigma'$. Observe that, for every $u, v \in L'$ such that $u \prec v$, if L' is a cover, then so is $L' \setminus \{u\}$.

We can now state the Generalized Leader Monotonicity Lemma.

LEMMA 8.2 (GENERALIZED LEADER MONOTONICITY LEMMA). *For every $\mathcal{G}(f_c)$-cover $\widehat{c\mathcal{D}\texttt{[f]}}$ of $c\mathcal{D}\texttt{[f]}$:*

$$\left(c\mathcal{D}\texttt{[f]} \parallel \mathcal{S}\texttt{[f]} \parallel \between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]}\right) \neq \emptyset \quad \textit{if and only if} \quad \left(\widehat{c\mathcal{D}\texttt{[f]}} \parallel \mathcal{S}\texttt{[f]} \parallel \between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]}\right) \neq \emptyset.$$

PROOF. ($\Leftarrow$): Follows from $\widehat{c\mathcal{D}}\texttt{[f]} \subseteq c\mathcal{D}\texttt{[f]}$.

($\Rightarrow$): We first observe that $\between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]}$ has the closure by write property, that is, if $\alpha\, w_c(g)\, \beta_1\beta_2 \in \between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]}$, then $\alpha\, w_c(g)\, \beta_1\, w_c(g)\, \beta_2 \in \between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]}$ for every $g \in \mathcal{G}$. Since $c\mathcal{D}\texttt{[f]} \parallel \mathcal{S}\texttt{[f]} \parallel \between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]} \neq \emptyset$, there exists $w \in c\mathcal{D}\texttt{[f]}$ such that $w \parallel \mathcal{S}\texttt{[f]} \parallel \between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]} \neq \emptyset$. Since $\widehat{c\mathcal{D}}\texttt{[f]}$ is a $\mathcal{G}(f_c)$-cover of $c\mathcal{D}\texttt{[f]}$, there exists $w' \in \widehat{c\mathcal{D}}\texttt{[f]}$ such that $w' \succeq_{\mathcal{G}(f_c)} w$ and $w' \in c\mathcal{D}\texttt{[f]}$. Proposition 7.3 shows that $c\mathcal{D} = \mathit{Proj}_{\mathcal{G}(w_\star, r_d)}(\mathcal{D} \parallel \mathcal{S})$; thus, we have that $w, w' \in \mathit{Proj}_{\mathcal{G}(w_\star, r_d)}(\mathcal{S})\texttt{[f]}$. We can thus apply Lemma 7.10 and obtain $w' \parallel \mathcal{S}\texttt{[f]} \parallel \between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]} \neq \emptyset$. Since $w' \in \widehat{c\mathcal{D}}\texttt{[f]}$, we finally get $\widehat{c\mathcal{D}}\texttt{[f]} \parallel \mathcal{S}\texttt{[f]} \parallel \between_{g\in\mathcal{G}} \mathit{Sim}_g\texttt{[f]} \neq \emptyset$. □

Example 8.3. Consider again the network of Example 7.6. Introducing first-writes we get

$$c\mathcal{D}\texttt{[f]} = (w_d(1)\, w_d(\$)) \between \left(\varepsilon + (f_c(1)\, w_c(1)^*)\right) \between \left(\varepsilon + (f_c(\$)\, w_c(\$)^*)\right).$$

The system

$$\widehat{c\mathcal{D}\texttt{[f]}}_1 = (w_d(1)\, w_d(\$)) \between \left(f_c(1)\, w_c(1)^*\right) \between \left(f_c(\$)\, w_c(\$)^*\right)$$

is a cover of $c\mathcal{D}\texttt{[f]}$, *but not a $\mathcal{G}(f_c)$-cover*: all the words covering $w_d(1)\, w_d(\$)$ contain one occurrence of $f_c(1)$ and $f_c(\$)$. On the other hand, the system

$$\begin{aligned}\widehat{c\mathcal{D}\texttt{[f]}}_2 \;=\; & (w_d(1)\, w_d(\$)) \between \left(f_c(1)\, w_c(1)^*\right) \between \left(\varepsilon + (f_c(\$)\, w_c(\$)^*)\right) \\ & + (w_d(1)\, w_d(\$)) \between \left(f_c(\$)\, w_c(\$)^*\right) \\ & + w_d(1)\, w_d(\$)\end{aligned}$$

is a $\mathcal{G}(f_c)$-cover of $c\mathcal{D}\texttt{[f]}$. To see that a $\mathcal{G}(f_c)$-cover is necessary for the Lemma, note that $(\widehat{c\mathcal{D}\texttt{[f]}}_2 \parallel \mathcal{S}\texttt{[f]} \parallel \mathit{Sim}_1\texttt{[f]} \between \mathit{Sim}_\$\texttt{[f]})$ is unsafe (as $w_d(1)\, w_d(\$)$ witnesses) while $(\widehat{c\mathcal{D}\texttt{[f]}}_1 \parallel \mathcal{S}\texttt{[f]} \parallel \mathit{Sim}_1\texttt{[f]} \between \mathit{Sim}_\$\texttt{[f]}) = \emptyset$. □

Putting it All Together. We now give the details of the construction.

THEOREM 8.4. `Safety(PDA, PDA)` *is in PSPACE.*

PROOF. We give a nondeterministic polynomial space algorithm. Our starting point is Equation (15). Using the Support Theorem and the Contributor Monotonicity Lemma, we reduce the proof obligation to checking Equation (16). At this point, we want to invoke the Generalized Leader Monotonicity Lemma; thus, we focus on checking

$$(c\mathcal{D}\texttt{[f]} \parallel \mathcal{S}\texttt{[f]} \parallel \between_{g\in\mathcal{G}} \mathit{Sim}^N_g\texttt{[f]}) \stackrel{?}{=} \emptyset.$$

As in the proof of Theorem 7.12, the PDA $cD[\mathtt{f}]$ for $c\mathcal{D}$ and the FSA for and $S[\mathtt{f}]$ are *exponentially larger* than D and S. We solve this problem in a similar way: we nondeterministically guess a sequence τ of first-writes. We have that $|\tau| \leq |\mathcal{G}|$; thus, τ can be guessed in polynomial time. It remains to nondeterministically check in polynomial space

$$(c\mathcal{D}[\mathtt{f}]^\tau \parallel \mathcal{S}[\mathtt{f}]^\tau \parallel \between_{g\in\mathcal{G}} \mathit{Sim}_g^N[\mathtt{f}]) \neq \emptyset.$$

We construct machines recognizing $c\mathcal{D}[\mathtt{f}]^\tau$, $\mathcal{S}[\mathtt{f}]^\tau$, and $\mathit{Sim}_g^N[\mathtt{f}]$.

—There is a polynomial time algorithm that constructs an FSA S^τ recognizing $\mathcal{S}[\mathtt{f}]^\tau$ from an FSA S and τ.

This is an elementary exercise.

—There is a polynomial time algorithm that constructs a context-free grammar $G^\tau_{c\mathcal{D}}$ recognizing $c\mathcal{D}[\mathtt{f}]^\tau$ (the reason for constructing a CFG instead of a PDA will become clear later).

Given the FSA S^τ, the algorithm constructs a PDA for $c\mathcal{D}[\mathtt{f}] = \mathit{Proj}_{\mathcal{G}(w_\star,r_d)}(\mathcal{D} \parallel \mathcal{S})[\mathtt{f}]^\tau$. Finally, it applies the polynomial transformation between PDAs and CFGs.

—There is a polynomial space algorithm that constructs an FSA A_g recognizing $\mathit{Sim}_g^N[\mathtt{f}]$.

Note that A_g is an FSA, not a PDA. Since the alphabet of $\mathit{Sim}_g[\mathtt{f}]$ only contains writes for g, we can easily construct in polynomial time a PDA $P_g[\mathtt{f}]$ recognizing $\mathit{Sim}_g[\mathtt{f}]$. Since in order to accept $\mathit{Sim}_g^N[\mathtt{f}]$ we can bound the stack depth of $P_g[\mathtt{f}]$ to $O(N)$, we can recognize $\mathit{Sim}_g^N[\mathtt{f}]$ by an FSA A_g with $2^{O(N)}$ states. The FSA has a state for each configuration of the PDA with stack depth $O(N)$, and transitions that simulate the moves of P_g. Moreover, A_g can be computed in (exponential time and) polynomial space. The algorithm just goes through all configurations with stack depth $O(N)$, and outputs for each of them all its successor configurations. For this, the algorithm needs to store only one configuration of the PDA, and the last rule used to compute a successor.

—There is an exponential time and polynomial space algorithm that constructs an FSA $A_{\between}$ recognizing $\between_{g\in\mathcal{G}} \mathit{Sim}_g^N[\mathtt{f}]$.

Summarizing, we are left with the problem of deciding

$$\left(L(G^\tau_{c\mathcal{D}}) \parallel L(S^\tau) \parallel L(A_{\between})\right) \stackrel{?}{=} \emptyset \tag{17}$$

in polynomial space, where $G_{c\mathcal{D}}$ and S^τ are a CFG and an FSA constructed by a polynomial time algorithm, and $A_{\between}$ is an FSA of exponential size constructed by an algorithm running in polynomial space.

Now, by the closure properties of regular and context-free languages with respect to asynchronous product, there exists an operator $\bowtie$ that, given a CFG G and an FSA A, constructs in polynomial time in $|G| + |A|$ a CFG $G \bowtie A$ recognizing $L(G) \parallel L(A)$. Then, there is an exponential time and polynomial space algorithm to construct the context-free grammar $G^\tau_{\bowtie} = \left((G^\tau_{c\mathcal{D}} \bowtie S^\tau) \bowtie A_{\between}\right)$, which recognizes Equation (17). Thus, we are left with the problem of deciding

$$L(G^\tau_{\bowtie}) = L(G^\tau_{c\mathcal{D}} \bowtie S^\tau \bowtie A_{\between}) \stackrel{?}{=} \emptyset \tag{18}$$

in polynomial space in the size of D and C.

Suppose that the emptiness problem for $G^{\tau}_{\bowtie}$ could be decided in polylogarithmic space. Then, we could resort to a generic result of complexity theory (e.g., see Lemma 4.17, Arora and Barak [2009]):

(Space Composition Theorem) Given functions $f, g : \Sigma^* \to \Sigma^*$, if f and g can be computed by s_f- and s_g-space-bounded Turing machines, respectively, then $g(f(x))$ can be computed in $\big(\log(|f(x)|) + s_f(|x|) + s_g(|f(x)|)\big)$ space.

We apply this result by choosing f and g as follows:

—f is the function that computes $G^{\tau}_{\bowtie}$ on input (D, C) (and nondeterministically guessing τ).
—g is the function that, on input $G^{\tau}_{\bowtie}$, returns 1 if $G^{\tau}_{\bowtie}$ is nonempty, and 0 otherwise.

Then, $g \circ f$ is the characteristic function of $\texttt{Safety}(\texttt{PDA}, \texttt{PDA})$. By the Space Composition Theorem, $g \circ f$ can be computed in polynomial space. We conclude that, if emptiness of $G^{\tau}_{\bowtie}$ could be checked in polylogarithmic space, then $\texttt{Safety}(\texttt{PDA}, \texttt{PDA}) \in$ PSPACE would follow.

At this point, however, we seem to reach a dead end, because $G^{\tau}_{\bowtie}$ is a context-free grammar. The emptiness problem for context-free grammars is P-complete [Jones and Laaser 1974], and there is no known polylogarithmic space algorithm for the problem. Thus, since $G_{\bowtie}$ is exponential in the size of the input, an algorithm that first constructs $G_{\bowtie}$ and then checks for emptiness runs in exponential space.

In the final step of the proof, we get around this problem by substituting $L(G^{\tau}_{\bowtie})$ by a cover that has nicer structural properties. Loosely speaking, we use leader monotonicity to show that the emptiness check can be replaced by the so-called *k-index emptiness check*, which is solvable in polylogarithmic space

We need some results of language theory about the k-index approximations of a context-free language [Brainerd 1967] (see Appendix A for formal definitions). Given a CFG $G = (\mathcal{X}, \Sigma_g, \mathcal{P}, X_0)$, we denote by $L^{(k)}(G)$ the set of words of $L(G)$ that can be derived from X_0 by means of a derivation in which every intermediate string contains at most k occurrences of nonterminals. Formally, let $\overset{(k)}{\Rightarrow}$ be the subrelation of the step relation $\Rightarrow$ defined as follows: $u \overset{(k)}{\Rightarrow} v$ if and only if $u \Rightarrow v$ and $|Proj_{\mathcal{X}}(v)| \leq k$, that is, v contains at most k occurrences of nonterminals. Further, let $\overset{(k)}{\Rightarrow}^*$ be the reflexive transitive closure of $\overset{(k)}{\Rightarrow}$. The *k-index language* of G is the set $L^{(k)}(G) = \{w \in \Sigma^* \mid X_0 \overset{(k)}{\Rightarrow}^* w\}$. We call the sequence of steps from X_0 to $w \in \Sigma^*$ a *k-index derivation*. Appendix A proves, or provides references for, the following results:

(a) $L^{(3m)}(G)$ is a cover of $L(G)$, where m is the number of nonterminals of G.
(b) For every FSA A and $k \geq 1$, $L^{(k)}(G \bowtie A) = L^{(k)}(G) \parallel L(A)$.
(c) There is a nondeterministic algorithm that, given a CFG G and a number $k \geq 1$, checks emptiness of $L^{(k)}(G)$ using $O(k \log(|G|))$ space. Thus, by Savitch's theorem, there is a deterministic $O(k^2 \log(|G|)^2)$ space algorithm for this problem.

Now, we proceed as follows. Recall that $G^{\tau}_{c\mathcal{D}}$ is a context-free grammar recognizing $c\mathcal{D}[\texttt{f}]^{\tau}$ that can be constructed in polynomial time in the size of the PDA D. Let $m_{c\mathcal{D}}$ be the number of nonterminals of $G^{\tau}_{c\mathcal{D}}$. Since $G^{\tau}_{c\mathcal{D}}$ can be constructed in polynomial time, $m_{c\mathcal{D}} < p(|D|)$ for some adequate polynomial p. Note that guessing τ is important: if we did not fix the ordering, $G_{c\mathcal{D}}$ and $m_{c\mathcal{D}}$ would not be polynomial in the size of D. Starting

with Equation (17), we get:

$$\begin{aligned}
&\left(L(G^{\tau}_{c\mathcal{D}}) \parallel L(S^{\tau}) \parallel L(A_{\lozenge})\right) \neq \emptyset \\
\Leftrightarrow\ &\{\text{Generalized Leader Monotonicity, (a) }\} \\
&\left(L^{(3m_{c\mathcal{D}})}(G^{\tau}_{c\mathcal{D}}) \parallel L(S^{\tau}) \parallel L(A_{\lozenge})\right) \neq \emptyset \\
\Leftrightarrow\ &\{\text{(b), Definition of } \bowtie\} \\
&L^{(3m_{c\mathcal{D}})}(G^{\tau}_{\bowtie}) \neq \emptyset
\end{aligned}$$

Since $G^{\tau}_{\bowtie}$ has exponential size, and $m_{c\mathcal{D}}$ is bounded by a polynomial, by (c) we can apply the Space Composition Theorem as follows:

—f is the function that, on input (D, C), returns $G^{\tau}_{\bowtie}$ and $3m_{c\mathcal{D}}$.
—g is the function that, on input (G, k), returns 1 if $L^{(k)}(G) \neq \emptyset$, and 0 otherwise.

It follows that $L^{3m_{c\mathcal{D}}}(G^{\tau}_{\bowtie}) \neq \emptyset$ can be decided in nondeterministic polynomial space. By Savitch's theorem, we conclude that *Safety*(PDA, PDA) is in PSPACE. □

We note that our language-theoretic constructions (the Support Theorem and results (a), (b), and (c) in the earlier proof) improve upon previous constructions, and are all required for the optimal upper bound. Hague [2011] shows an alternate doubly exponential construction using a result of Ehrenfeucht and Rozenberg in place of Theorem A.3. This gave a 2EXPTIME algorithm. Even after using our exponential time construction for A_g, we can only get an EXPTIME algorithm, since the nonemptiness problem for (general) context-free languages is P-complete [Jones and Laaser 1974]. Our bounded-index approximation for the cover and the space-efficient emptiness algorithm for bounded-index languages are crucial to the PSPACE upper bound.

9. DISCUSSIONS

We have classified the complexity of safety verification for parameterized systems with a distinguished leader communicating through finite-state shared memory without any synchronization. Our upper bounds are surprising, because the complexity of the problem can be significantly higher (even undecidable) in slightly different settings [Esparza 2014].

We now discuss two additional directions. The first is the special *symmetric* case, in which all participants run the same "code." We show that this case can be solved in polynomial time. The second is where we bound the number of steps that each individual participant can perform. We show that, if the bound is given in unary, the problem remains coNP-complete even when each participant is an arbitrary Turing machine.

9.1. The Symmetric Case

In the symmetric case, we have $\mathcal{D} = \mathcal{C}$, that is, the leader and the contributors have the same language. We consider the safety verification problem in this setting and show that it is in polynomial time when the leader and the contributors are given by a PDA.

Because all processes are identical, we can alternately assume that the distinguished leader process is the language $\{\varepsilon\}$, and consider the problem whether the $(\{\varepsilon\}, \mathcal{C})$-network $\mathcal{N}$ is safe. Note that, in this case, we modify the definition of safety and require that *some* participant writes the special value \$. By replacing $\mathcal{S}$ by $Proj_{\mathcal{C}}(\mathcal{S})$, we can completely discard $\mathcal{D}$ and $\Sigma_{\mathcal{D}}$ and rewrite the safety verification problem as follows:

$$\mathcal{S} \parallel \left((Sim_{\$} \setminus \{\varepsilon\}) \lozenge_{g \in \mathcal{G}, g \neq \$} Sim_g\right) \stackrel{?}{=} \emptyset.$$

ALGORITHM 1: Decision procedure when all processes are identical and given by PDA

begin
  Data: A $(\{\varepsilon\}, \mathcal{C})$-network where $\mathcal{C}$ is given by the PDA C and $\$ \in \mathcal{G}$
  Result: Whether the $(\{\varepsilon\}, \mathcal{C})$-network is safe
  $wAcc := \varepsilon$ and $Seen := \emptyset$;
  repeat
    $keepgoing := false$;
    foreach $g \in \mathcal{G} \setminus Seen$ **do**
      if $\mathcal{S}[\![f]\!] \parallel wAcc \ \left(\lozenge_{g' \in Seen} w_c(g')^* \lozenge Sim_g[\![f]\!] \right) \neq \emptyset$ **then**
        $keepgoing := true$;
        $wAcc := wAcc \cdot f_c(g)$ and $Seen := Seen \cup \{g\}$;
        if $g = \$$ **then return** unsafe;
        ;
  until $keepgoing = false$;
  return safe;

The safety verification decision procedure for this case is given by Algorithm 1. We show it is correct and runs in polynomial time in Theorem 9.2. The central argument for the correctness of Algorithm 1 relies on the following lemma. In what follows, for each $g \in \mathcal{G}$, we replace Sim_g by $Sim_g \setminus \{\varepsilon\}$.

LEMMA 9.1. *Let $1 \leq d \leq |\mathcal{G}|$ and let $\mathcal{G}_d = \{g_1, \ldots, g_{d-1}\} \cup \{\$\}$ be a set of distinct symbols all in $\mathcal{G}$. Let $\tau = f_c(g_1) \cdots f_c(g_{d-1})\, f_c(\$)$. We have that*

$$\mathcal{S}[\![f]\!]^{\tau} \parallel \lozenge_{g \in \mathcal{G}_d} Sim_g[\![f]\!] \neq \emptyset$$

if and only if

$$\begin{aligned}
& \mathcal{S}[\![f]\!]^{(\tau)_1} && \parallel Sim_{g_1}[\![f]\!] \neq \emptyset \\
\wedge\ & \mathcal{S}[\![f]\!]^{(\tau)_1 (\tau)_2} && \parallel f_c(g_1)\, (w_c(g_1)^* \lozenge Sim_{g_2}[\![f]\!]) \neq \emptyset \\
& \vdots \\
\wedge\ & \mathcal{S}[\![f]\!]^{(\tau)_{1..d-1}} && \parallel f_c(g_1) \cdots f_c(g_{d-2}) \left(w_c(g_1)^* \lozenge \cdots \lozenge w_c(g_{d-2})^* \lozenge Sim_{g_{d-1}}[\![f]\!] \right) \neq \emptyset \\
\wedge\ & \mathcal{S}[\![f]\!]^{\tau} && \parallel f_c(g_1) \cdots f_c(g_{d-1}) \left(w_c(g_1)^* \lozenge \cdots \lozenge w_c(g_{d-1})^* \lozenge Sim_{\$}[\![f]\!] \right) \neq \emptyset.
\end{aligned}$$

(For the sake of clarity, assume that Kleene star binds stronger than concatenation, which binds stronger than shuffle, which binds stronger than asynchronous product. The alphabet on the right-hand side of each asynchronous product is given by the least set containing the alphabet of the involved simulator and the other occurring symbols.)

Before proving the lemma, we show how it implies the correctness of Algorithm 1.

THEOREM 9.2. $\texttt{Safety}(\{\varepsilon\}, \texttt{PDA})$ *is PTIME-complete.*

PROOF. PTIME-hardness follows from the emptiness problem for PDAs [Jones and Laaser 1974]. Given a PDA over an alphabet Σ, we can modify its transitions to write $w_c(a)$ on every transition that reads $a \in \Sigma$ in the original PDA, and to write a new symbol \$ from its accepting state. Then, \$ can be written by the network if and only if some word is accepted by the PDA.

We now prove that Algorithm 1 is correct and runs in polynomial time. The algorithm computes an accumulation (in set *Seen*) of possible symbols that can be written by the network. If the symbol \$ is produced, it stops and returns unsafe. At each step, it maintains the invariant that

$$\mathcal{S}[\![\texttt{f}]\!]^{wAcc} \parallel \lozenge_{g \in Seen} Sim_g[\![\texttt{f}]\!] \neq \emptyset$$

using Lemma 9.1. This goes on until either \$ appears in *Seen* or no more values can be appended to *wAcc*.

Let us now turn to establish its correctness.

Soundness. If Algorithm 1 returns **unsafe**, then we have that \$ is the last symbol of *wAcc*. The sequence of checks in the **if** statement and Lemma 9.1 applied to $\tau = wAcc$ lets us conclude that $\mathcal{S}[\![\mathtt{f}]\!]^{wAcc} \parallel \mathbin{\text{Ⓢ}}_{g \in Seen} Sim_g[\![\mathtt{f}]\!] \neq \emptyset$, and we are done.

Completeness. Suppose that Algorithm 1 returns **safe** but (toward a contradiction) there exists a τ such that $\mathcal{S}[\![\mathtt{f}]\!]^{\tau} \parallel \mathbin{\text{Ⓢ}}_{g \in \{(\tau)_1,\ldots,(\tau)_{|\tau|}\}} Sim_g[\![\mathtt{f}]\!] \neq \emptyset$. Then, we have that at no point along the history of computation $(wAcc)_i = \$$ for some i. In particular, at no point in time *wAcc* contains all the first-writes of τ. Let i be the least position in τ such that $(\tau)_i$ is never added to *wAcc*. That is, the conditional of the **if** statement always fails for $(\tau)_i$. We derive a contradiction. We assume that all values of τ up to position $(i-1)$ eventually appear in *wAcc*. Thus, if the conditional always fails for $(\tau)_i$, we have that $\mathcal{S}[\![\mathtt{f}]\!]^{wAcc} \parallel wAcc \cdot \big(\mathbin{\text{Ⓢ}}_{g \in Seen} w_c(g)^* \mathbin{\text{Ⓢ}} Sim_{(\tau)_i}[\![\mathtt{f}]\!] \big) = \emptyset$. But this is a contradiction, since $\mathcal{S}[\![\mathtt{f}]\!]^{\tau} \parallel \mathbin{\text{Ⓢ}}_{g \in \{(\tau)_1,\ldots,(\tau)_{|\tau|}\}} Sim_g[\![\mathtt{f}]\!] \neq \emptyset$, which, by Lemma 9.1, implies that $\mathcal{S}[\![\mathtt{f}]\!]^{(\tau)_{1..i}} \parallel (\tau)_{1..i-1} \big(\mathbin{\text{Ⓢ}}_{g \in \{(\tau)_1,\ldots,(\tau)_{i-1}\}} w_c(g)^* \mathbin{\text{Ⓢ}} Sim_{(\tau)_i}[\![\mathtt{f}]\!] \big) \neq \emptyset$ and *Seen* includes all the values of τ up to position $i-1$.

Complexity. Observe that every time the condition in the **if** statement succeeds, *Seen* grows. Thus, the **if** statement can be taken at most $|\mathcal{G}|$ times. The outer **repeat** loop continues only if the inner loop finds at least one element that satisfies the **if** condition; thus, it can run at most $|\mathcal{G}|$ times. The **foreach** loop checks the **if** condition for up to $|\mathcal{G}|$ elements. Thus, the body of the **foreach** loop can execute at most $|\mathcal{G}|^2$ times.

We now show that the check in the **if** condition can be performed in polynomial time. First, recall that $\mathcal{S}[\![\mathtt{f}]\!]^{wAcc}$ and $Sim_g[\![\mathtt{f}]\!]$, for $g \in \mathcal{G}$, can both be computed in polynomial time in the input. Second, because $a_1^* \mathbin{\text{Ⓢ}} \cdots \mathbin{\text{Ⓢ}} a_n^*$ coincides with $\{a_1, \ldots, a_n\}^*$, where each a_i is an alphabet symbol, we can replace the conditional of the **if** statement by the equivalent

$$(\mathcal{S}[\![\mathtt{f}]\!]^{wAcc} \parallel wAcc \cdot \big(\{w_c(g') \mid g' \in Seen\}^* \mathbin{\text{Ⓢ}} Sim_g[\![\mathtt{f}]\!]\big) \neq \emptyset.$$

Given Sim_g, we can construct $\{w_c(g') \mid g' \in Seen\} \mathbin{\text{Ⓢ}} Sim_g[\![\mathtt{f}]\!]$ by adding self-loops labeled with $w_c(g')$ at every state. Also, given a word w and a PDA P, an easy polynomial-time construction produces a PDA that accepts $w \cdot L(P)$. Finally, we compute the asynchronous product of $\mathcal{S}[\![\mathtt{f}]\!]^{wAcc}$ and this PDA to get a PDA that is polynomial in the size of the input. Emptiness of this machine can be checked in polynomial time. Thus, overall, the runtime is bounded by a polynomial in $|\mathcal{C}|$. □

We complete the proof by proving Lemma 9.1.

Proof of Lemma 9.1. We prove the lemma by induction on the length d. If $d = 1$, then the result trivially holds.

For the inductive case, we require the following equivalence:

$$\mathcal{S}[\![\mathtt{f}]\!]^{(\tau)_{1..i+1}} \parallel (Sim_{g_1}[\![\mathtt{f}]\!] \mathbin{\text{Ⓢ}} \cdots \mathbin{\text{Ⓢ}} Sim_{g_i}[\![\mathtt{f}]\!] \mathbin{\text{Ⓢ}} Sim_{g_{i+1}}[\![\mathtt{f}]\!]) \neq \emptyset \tag{19}$$

if and only if

$$\mathcal{S}[\![\mathtt{f}]\!]^{(\tau)_{1..i}} \parallel (Sim_{g_1}[\![\mathtt{f}]\!] \mathbin{\text{Ⓢ}} \cdots \mathbin{\text{Ⓢ}} Sim_{g_i}[\![\mathtt{f}]\!]) \neq \emptyset \tag{20}$$

and

$$\mathcal{S}[\![\mathtt{f}]\!]^{(\tau)_{1..i+1}} \parallel (\tau)_{1..i} \big(w_c(g_1)^* \mathbin{\text{Ⓢ}} \cdots \mathbin{\text{Ⓢ}} w_c(g_i)^* \mathbin{\text{Ⓢ}} Sim_{g_{i+1}}[\![\mathtt{f}]\!]\big) \neq \emptyset. \tag{21}$$

From this equivalence, by applying the induction hypothesis to Equation (20), we conclude the first i elements of the conjunction and the lemma. We now prove the equivalence.

("only if") Suppose that Equation (19) holds. Then, Equation (20) holds as well: take the projection of any word in the language on $\{g_1, \ldots, g_i\}$. To see that Equation (21) also holds, note that, if Equation (19) holds, then, by erasing reads and corrupting writes of the simulators (see Lemma 4.9), we have that

$$\mathcal{S}[\mathtt{f}]^{(\tau)_{1..i+1}} \parallel (f_c(g_1)\, w_c(g_1)^* \between \cdots \between f_c(g_i)\, w_c(g_i)^* \between \mathit{Sim}_{g_{i+1}}[\mathtt{f}]) \neq \emptyset.$$

Since the writes on the store follow the order given by τ, we can pull the first-writes forward in the same order as τ. Thus,

$$\mathcal{S}[\mathtt{f}]^{(\tau)_{1..i+1}} \parallel f_c(g_1) \ldots f_c(g_i)\, (w_c(g_1)^* \between \cdots \between w_c(g_i)^* \between \mathit{Sim}_{g_{i+1}}[\mathtt{f}]) \neq \emptyset,$$

which is equivalent to

$$\mathcal{S}[\mathtt{f}]^{(\tau)_{1..i+1}} \parallel (\tau)_{1..i}\, (w_c(g_1)^* \between \cdots \between w_c(g_i)^* \between \mathit{Sim}_{g_{i+1}}[\mathtt{f}]) \neq \emptyset.$$

("if") Let $L_i = \mathcal{S}[\mathtt{f}]^{(\tau)_{1..i}} \parallel (\mathit{Sim}_{g_1}[\mathtt{f}] \between \cdots \between \mathit{Sim}_{g_i}[\mathtt{f}])$. Since Equation (20) holds, there exists $v \in L_i$. Observe that, since each $\mathit{Sim}_g[\mathtt{f}]$ is closed by writes, any extension of v by a sequence of writes from $\{w_c(g) \mid g \in \{g_1, \ldots, g_i\}\}$ is also in L_i. Additionally, by Equation (21), there exists $w' \in \mathcal{S}[\mathtt{f}]^{(\tau)_{1..i+1}} \parallel (\tau)_{1..i}\, (w_c(g_1)^* \between \cdots \between w_c(g_i)^* \between \mathit{Sim}_{g_{i+1}}[\mathtt{f}])$. Intuitively, it says that, whenever g_1 to g_i are available on demand, then g_{i+1} can also be made available on demand. Write w' as $(\tau)_{1..i}\, w''$ for some suffix w'' and consider $v' = v \cdot w''$. Observe that the sequence of first-writes of v' is exactly $(\tau)_{1..i+1}$. Intuitively, v' first makes available the values g_1 to g_i by the choice of v; then, whenever needed, these are consumed in w'', which makes available g_{i+1}. Together, we find that $v' \in L_{i+1}$, and we are done. □

9.2. The Bounded Safety Problem

Given $k > 0$, we say that a $(\mathcal{D}, \mathcal{C})$-network is *k-safe* if all words in which the leader and each contributor make at most k steps are safe; that is, we put a bound of k steps on the runtime of the leader as well as each contributor, and consider only safety within this bound. The bound does not limit the total length of words, because the number of contributors is unbounded. The *bounded safety* problem asks, given $\mathcal{D}$, $\mathcal{C}$, and k written in unary, if the $(\mathcal{D}, \mathcal{C})$-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 $M_\mathcal{D} \in \mathtt{D}$ and $M_\mathcal{C} \in \mathtt{C}$, where we write k in unary. A closer look at Theorem 5.1 shows that its proof reduces the satisfiability problem for a formula ϕ to the bounded safety problem for a $(L(M_\mathcal{D}), L(M_\mathcal{C}))$-network ($M_\mathcal{D} \in \mathtt{D}$, $M_\mathcal{C} \in \mathtt{C}$) and a number k, all of which have polynomial size $|\phi|$. This proves that BoundedSafety(FSA, FSA) is coNP-hard, already for deterministic FSAs. We show that, surprisingly, bounded safety remains coNP-complete for pushdown systems, and, even further, for arbitrary Turing machines. Note that the problem is already coNP-complete for one single (nondeterministic) 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 $\mathcal{G}(w_\star, r_\star)$ for communication with the store. The machines are input-free, that is, the input tape is always initially empty.

THEOREM 9.3. BoundedSafety(TM, TM) *is coNP-complete.*

PROOF. coNP-hardness follows from Theorem 5.1. To prove BoundedSafety(TM, TM) is in coNP, we use the Simulation Lemma. Let $M_{\mathcal{D}}, M_{\mathcal{C}}, k$ be an instance of the problem, where $M_{\mathcal{D}}, M_{\mathcal{C}}$ are Turing machines of sizes $n_{\mathcal{D}}, n_{\mathcal{C}}$ with languages $\mathcal{D} = L(M_{\mathcal{D}})$ and $\mathcal{C} = L(M_{\mathcal{C}})$, and let $n_{\mathcal{D}} + n_{\mathcal{C}} = n$. In particular, we can assume that $|\mathcal{G}| \leq n$, because we only need to consider actions that appear in $M_{\mathcal{D}}$ and $M_{\mathcal{C}}$. If the $(\mathcal{D}, \mathcal{C})$-network is not k-safe, then by definition there exist $u \in \mathcal{D}$ and a multiset $M = \{v_1, \ldots, v_m\}$ over $\mathcal{C}$ such that u is compatible with M; moreover, all of $u, v_1, \ldots, v_m$ have length at most k. We conclude from the Simulation Lemma (in particular, the constructions used in its proof) that there exists a set $\{s_{g_1}\, w_c(g_1)^{i_1}, \ldots, s_{g_m}\, w_c(g_m)^{i_m}\}$ such that $m \leq |\mathcal{G}| \leq n$, $s_{g_j}\, w_c(g_j)^{i_j} \in Sim_{g_j}, i_j > 0$ for all j. Since each of the s_{g_i} stems from a word of M, hence $\mathcal{C}$, by suitably renaming its actions, we have $|s_{g_i}| < k$. Moreover, since the $w_c(g_j)^{i_j}$ parts provide the writes necessary to execute the reads of u and 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 $(\mathcal{D}, \mathcal{C})$-network is k-unsafe. The algorithm will guess runs of length at most k for the TM of $\mathcal{D}$ and $\mathcal{C}$. The guess for the TM of $\mathcal{D}$ yields a word u ending with $w_d(\$)$ because we require the run of $\mathcal{D}$ to accept, and the n guesses for $\mathcal{C}$ results in the words $s_{g_1}, \ldots, s_{g_n}$ of length at most k. Note that the empty run is a valid guess. Otherwise, if the guessed run is nonempty, then its last action must be a write. Furthermore, and without loss of generality, we assume that if $s_{g_1} \neq \varepsilon$, then it ends with $w_c(g_1)$. Since there are $n+1$ of those words, the guess can be done in polynomial time. Then, the algorithm guesses numbers $i_1, \ldots, i_n$. 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 a word $x \in u \between s_{g_1}\, w_c(g_1)^{i_1} \between \ldots \between s_{g_m}\, w_c(g_m)^{i_m}$ and checks that it obeys the store rules, that is, $x \in \mathcal{S}$. This can be done in $O(n^2 \cdot k)$ time. If the algorithm succeeds, then there is a witness that $(\mathcal{D} \parallel \mathcal{S} \parallel \between_{g \in \mathcal{G}} Sim_g) \neq \emptyset$ holds, which shows, by the Simulation Lemma, that the $(\mathcal{D}, \mathcal{C})$-network is unsafe. □

A TM is polytime 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 polynomial time bounded nondeterministic Turing machines, is coNP-complete.

A. APPENDIX: LANGUAGE THEORETIC CONSTRUCTIONS

Context-Free Grammars. A *context-free grammar* (or CFG) is a tuple $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$, where $\mathcal{X}$ is a finite set of *nonterminals* including X_0, Σ is an alphabet, such that $\mathcal{X} \cap \Sigma = \emptyset$, and $\mathcal{P} \subseteq \mathcal{X} \times (\Sigma \cup \mathcal{X})^*$ is a finite set of *productions*. For a production $(X, w) \in \mathcal{P}$, often conveniently noted $X \to w$, we define its *size* as $|(X, w)| = |w| + 1$, and $|G| = \sum_{p \in \mathcal{P}} |p|$ defines the size of the grammar G. G is in Chomsky normal form (CNF) if and only if $\mathcal{P} \subseteq (\mathcal{X} \times (\Sigma \cup \mathcal{X}^2)) \cup \{(X_0, \varepsilon)\}$. A CFG can be converted to CNF in time polynomial in its size. Given two strings $u, v \in (\Sigma \cup \mathcal{X})^*$, a production $(X, w) \in \mathcal{P}$ and a position $1 \leq j \leq |u|$, we define a *step* $u \overset{(X,w)/j}{\Longrightarrow}_G v$ if and only if $(u)_j = X$ and $v = (u)_1 \cdots (u)_{j-1}\, w\, (u)_{j+1} \cdots (u)_{|u|}$. We omit (X, w) or j above the arrow when it is not important. *Step sequences* (including the empty sequence) are defined using the reflexive transitive closure of the step relation $\Rightarrow_G$, denoted $\Rightarrow_G^*$. We call any *step sequenceu* $\Rightarrow_G^*$ v *a derivation* whenever $u \in \mathcal{X}$ and $v \in \Sigma^*$. We omit the argument G in the following, when it is clear from the context. Finally, the language generated by G denoted $L(G)$ is the set $\{w \in \Sigma^* \mid X_0 \Rightarrow^* w\}$. A step $u \overset{(X,w)/j}{\Longrightarrow}_G v$ is said to be *leftmost*

whenever j is the least position such that $(u)_j \in \mathcal{X}$. As expected, a step sequence is k-index if all its steps are k-index. For a given integer constant $k > 0$, a word $u \in (\Sigma \cup \mathcal{X})^*$ is said to be of index k if u contains at most k occurrences of nonterminals (formally, $|Proj_{\mathcal{X}}(u)| \leq k$). A step $u \Rightarrow v$ is said to be k-index, denoted $u \overset{(k)}{\Longrightarrow} v$, if and only if both u and v are of index k. Given G and k, define $L^{(k)}(G)$ to be the set $\{w \in \Sigma^* \mid X_0 \overset{(k)}{\Longrightarrow}{}^* w\}$. It is routine to check that $L^{(k)}(G) \subseteq L(G)$ for every $k > 0$.

We now complete the proof of Theorem 8.4 by providing the language-theoretic constructions. We assume familiarity with basic formal language theory [Sipser 1996].

Asynchronous product of CFGs and FSAs. Given a CFG G in CNF and an 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 that 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 A.1. Given a CFG $G = (\mathcal{X}, \Sigma_g, \mathcal{P}, X_0)$ in CNF and an FSA $A = (Q, \Sigma_a, \delta, q_0, \{q_f\})$, we define $G^{\bowtie} = (\mathcal{X}^{\bowtie}, \Sigma_g \cup \Sigma_a, \mathcal{P}^{\bowtie}, X_0^{\bowtie})$ as follows:

—$\mathcal{X}^{\bowtie} = Q \times (\mathcal{X} \cup \{\varepsilon\}) \times Q$;
—$X_0^{\bowtie} = \langle q_0, X_0, q_f \rangle$;
—$\mathcal{P}^{\bowtie}$ contains no more than the following transitions:
(1) if $X \to \sigma \in \mathcal{P}$ and $\sigma \notin \Sigma_a$, then $\langle q, X, q' \rangle \to \sigma \langle q, \varepsilon, q' \rangle \in \mathcal{P}^{\bowtie}$ (case $\sigma \in \Sigma_g \setminus \Sigma_a$ or $\sigma = \varepsilon$)
(2) if $\sigma \notin \Sigma_g$ and $(q, \sigma, q') \in \delta$, then $\langle q, Z, q'' \rangle \to \sigma \langle q', Z, q'' \rangle \in \mathcal{P}^{\bowtie}$ for all $Z \in \mathcal{X} \cup \{\varepsilon\}$ (case $\sigma \in \Sigma_a \setminus \Sigma_g$ or $\sigma = \varepsilon$)
(3) if $X \to \sigma \in \mathcal{P}$, $(q, \sigma, q') \in \delta$, and $\sigma \neq \varepsilon$, then $\langle q, X, q'' \rangle \to \sigma \langle q', \varepsilon, q'' \rangle \in \mathcal{P}^{\bowtie}$ (case $\sigma \in \Sigma_g \cap \Sigma_a$)
(4) if $X \to Y\ Z \in \mathcal{P}$, then $\langle q_1, X, q_2 \rangle \to \langle q_1, Y, q' \rangle \langle q', Z, q_2 \rangle \in \mathcal{P}^{\bowtie}$ for every $q' \in Q$.
(5) $\langle q, \varepsilon, q \rangle \to \varepsilon \in \mathcal{P}^{\bowtie}$ for every $q \in Q$.

Proposition A.2. *Let G, A, and $G^{\bowtie}$ as in Definition A.1. 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 $k \geq 1$: $\langle q, X, q' \rangle \overset{(k)}{\Longrightarrow}{}^* w$ if and only if $w \in w^a \parallel w^g$ and $X \overset{(k)}{\Longrightarrow}{}^* w^g$ in G, and there is a run in A from q to q' that reads w^a, formally: $\exists q_1, \ldots, q_{n+1} \in Q$, $\exists \sigma_1, \ldots, \sigma_n \in \Sigma : (q_i, \sigma_i, q_{i+1}) \in \delta$ for all $i < n$, $w^a = \sigma_1 \ldots \sigma_n$. From Definition A.1, it is clear that $G^{\bowtie}$ is computable in time polynomial in $|G|$ and $|A|$. □

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

Theorem A.3. *Given a CFG $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ in CNF with n nonterminals, we can compute an FSA A with $O(2^{n \log(n)})$ states such that $L(A)$ is a support of $L(G)$.*

The proof of Theorem A.3 requires the following technical lemma.

Lemma A.4. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_l)$ be in CNF and $D : X_0 \Rightarrow^* v \in \Sigma^*$ a leftmost derivation for some $X_0 \in \mathcal{X}$. There exists $v' \preceq v$ and a leftmost n-index derivation $D' : X_0 \overset{(n)}{\Longrightarrow}{}^* v'$, where n is the number of distinct nonterminals 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 nonterminals 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 $X_0 \Rightarrow BC \Rightarrow^* w_1C \Rightarrow^* w_1w_2 = v$. Moreover $m = 0$ shows that X_0 does not appear in the subsequence of steps $D_1 : B \Rightarrow^* w_1$ and $D_2 : C \Rightarrow^* w_2$. The number of distinct nonterminals 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 $D'_1 : B \overset{(n-1)}{\Longrightarrow}{}^* w'_1$ and $D'_2 : C \overset{(n-1)}{\Longrightarrow}{}^* w'_2$ with $w'_1w'_2 \preceq w_1w_2 = v$, hence that there exists a derivation $D' : X_0 \overset{(n)}{\Longrightarrow}{}^* BC \overset{(n)}{\Longrightarrow}{}^* w'_1C \overset{(n)}{\Longrightarrow}{}^* w'_1w'_2 = v'$ such that $v' \preceq v$, and we are done.

Step. $m > 0$. Therefore, in D, there exists some nonterminal X such that $X \Rightarrow^* wX\alpha \Rightarrow^* ww'\alpha$ and $w \neq \varepsilon$. Define the derivation D' given by D, where the earlier 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 transformation on D allows use of the induction hypothesis on D'; hence, we find that there exists a leftmost n-index derivation $D'' : X_0 \overset{(n)}{\Longrightarrow}{}^* v''$ and $v'' \preceq v'$, and we are done since $v'' \preceq v' \preceq v$. □

PROOF OF THEOREM A.3. From Lemma A.4, 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 an FSA $A = (\Sigma, Q, \delta, q_0, F)$ such that (*i*) $Q = \{w \in \mathcal{X}^j \mid 0 \leq j \leq n\}$; (*ii*) $\delta = \{(\alpha\, w, \gamma, \beta\, w) \mid (\alpha, \gamma\beta) \in \mathcal{P} \wedge \gamma \in \Sigma \cup \{\varepsilon\}\}$; (*iii*) $q_0 = X_0$; and (*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 correspond 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 next 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 nonterminal Z such that Y depends on Z and Z depends on X. A *strongly connected component* (SCC) of G if a maximal subset of mutually dependent nonterminals.

THEOREM A.5. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ be a CFG in CNF with n nonterminals and k SCCs. Then, $L^{(n+2k)}(G)$ is a cover of $L(G)$.*

The proof of Theorem A.5 uses the following technical lemma, which follows from the fact that the commutative images of $L(G)$ and $L^{(n+1)}(G)$ coincide [Esparza et al. 2011].

LEMMA A.6. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ be a CFG in CNF with n nonterminals. 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 A.5. Let $w \in L(G)$. If $w = \varepsilon$, then since G is in CNF, we have $X_0 \Rightarrow \varepsilon$. Thus, w is also in $L^{(n+2k)}(G)$. Hence, assume that $w \neq \varepsilon$. 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 A.6, $X_1 \overset{(n+1)}{\Longrightarrow}{}^* v_1$ for some $v_1 \succeq a$, and by the induction hypothesis on $|w|X_2 \overset{(n+2)}{\Longrightarrow}{}^* v_2$ for some $v_2 \succeq v$. Thus, 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. $k > 1$. Define a strongly connected component (or SCC) of a graph to be *bottom* when every edge whose source is in SCC also belongs the SCC. Now, let $\mathcal{Y} \subseteq \mathcal{X}$ be the set of nonterminals of a bottom strongly connected component of G, and let $\mathcal{P}_\mathcal{Y} \subseteq \mathcal{P}$ be the productions of G with a nonterminal 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 $G' = (\mathcal{X} \setminus \mathcal{Y}, \Sigma \cup \mathcal{Y}, \mathcal{P} \setminus \mathcal{P}_\mathcal{Y}, X_0)$. Since $w \in L(G)$, there exist derivations $X_0 \Rightarrow^* w_1Y_1w_2 \ldots w_rY_rw_{r+1}$ in $L(G')$ and $Y_i \Rightarrow^* v_i$ in $L(G_{Y_i})$ for every $i = 1, \ldots, r$ such that $w_1v_1 \ldots w_rv_rw_{r+1} = w$. Since G' has $(k-1)$ SCCs, by induction hypothesis, there is $X_0 \overset{(i_1)}{\Longrightarrow}{}^* w'_1Y'_1w'_2 \ldots w'_tY'_tw'_{t+1}$ in $L(G')$, where $i_1 = n - |\mathcal{Y}| + 2(k-1)$ and such that $w'_1Y'_1w'_2 \ldots w'_tY'_tw'_{t+1} \succeq w_1Y_1w_2 \ldots w_rY_rw_{r+1}$. In particular, we have $Y'_1 \ldots Y'_t \succeq Y_1 \ldots Y_r$, which implies that there exists a monotonic injection $h : \{1, \ldots, r\} \to \{1, \ldots, t\}$ such that $Y'_{h(i)} = Y_i$ for all $i \in \{1, \ldots, r\}$. Since every G_Y has one strongly connected component, for every $j = 1, \ldots, r$, 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 $i_2 = |\mathcal{Y}| + 2$.

Thus, we have

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

Let $w' = w'_1v'_1 \ldots w'_tv'_tw'_{t+1}$. 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, nonemptiness checking for context-free languages is P-complete, as shown by Jones and Laaser [1974].

We first need a lemma about the decomposition of derivations.

LEMMA A.7. *The following properties hold for k-index step sequences:*

(1) *If* $BC \overset{(k)}{\Longrightarrow}{}^* w$, *then there exist* w_1, w_2 *such that* $w = w_1\,w_2$ *and either (i)* $B \overset{(k-1)}{\Longrightarrow}{}^* w_1$, $C \overset{(k)}{\Longrightarrow}{}^* w_2$, *or (ii)* $C \overset{(k-1)}{\Longrightarrow}{}^* w_2$ *and* $B \overset{(k)}{\Longrightarrow}{}^* w_1$; *and vice versa.*

(2) *If either (a)* $B \overset{(k-1)}{\Longrightarrow}{}^* w_1$, $C \overset{(k)}{\Longrightarrow}{}^* w_2$, *or (b)* $C \overset{(k-1)}{\Longrightarrow}{}^* w_2$ *and* $B \overset{(k)}{\Longrightarrow}{}^* w_1$, *then* $BC \overset{(k)}{\Longrightarrow}{}^* w_1\,w_2$.

PROOF. Consider the step sequence $BC \overset{(k)}{\Longrightarrow}{}^* w$, which we rewrite $(BC\ =)\alpha_0 \overset{(k)}{\Longrightarrow} \alpha_1 \cdots \alpha_{n-1} \overset{(k)}{\Longrightarrow} \alpha_n (= w)$. We define a provenance annotation (the function *prov*) of each word in the step sequence. Intuitively, it maps each symbol of each word in the sequence onto a natural number i such that i refers to a position in α_0 giving the provenance information. The definition of *prov* is inductive. Define $prov(0) = 1 \ldots |\alpha_0|$,

which means that $(\alpha_0)_j$ stems from itself since $(prov(0))_j = j$. For the inductive case, given a step $\alpha_i \overset{(Z,z)/j}{\Longrightarrow}_{\alpha_{i+1}}$, define $prov(i+1) = (prov(i))_1 \cdots (prov(i))_{j-1} \cdot \big((prov(i))_j\big)^{|z|} \cdot (prov(i))_{j+1} \cdots (prov(i))_{|\alpha_i|}$. That is, either the provenance information is inherited from the nonterminal $(\alpha_i)_j (= Z)$ that has been rewritten yielding the $|z|$ copies of $(prov(i))_j$ in $prov(i+1)$ or it remains unchanged ($(prov(i))_k$ for all $k \neq j$). Note that $prov(i)$ is a word over alphabet $\{1, \ldots, |\alpha_0|\}$ and that $|prov(i)| = |\alpha_i|$ for all i.

Let us turn back to the step sequence $BC \overset{(k)}{\Longrightarrow}^* w$, whose last step $\alpha_{n-1} \overset{p/j}{\Longrightarrow} \alpha_n$ is the application of some production p at some position j of α_{n-1}. Since $\alpha_0 = BC$, necessarily we have that $(prov(n-1))_j$ is either 1 or 2. Assume that $(prov(n-1))_j = 2$ (the other case is treated similarly). Define a new step sequence by removing from $BC \overset{(k)}{\Longrightarrow}^* w$ all the steps $\alpha_i \overset{p/j}{\Longrightarrow} \alpha_{i+1}$ such that $(prov(i))_j = 2$. Intuitively, we discard all the steps that rewrite a nonterminal stemming from the initial C since $(\alpha_0)_2 = C$.

The resulting step sequence has the form $BC \overset{(k)}{\Longrightarrow}^* w_1 C$. Then, define w_2 to be the unique word satisfying $w = w_1 w_2$. Since $BC \overset{(k)}{\Longrightarrow}^* w_1 C$, by removing the occurrence of C in rightmost position at every step, we find that $B \overset{(k-1)}{\Longrightarrow}^* w_1$, and we are done.

For the second property, it suffices to observe that $B \overset{(k-1)}{\Longrightarrow}^* w_1$ yields $BC \overset{(k)}{\Longrightarrow}^* w_1 C$, hence that $BC \overset{(k)}{\Longrightarrow}^* w_1 w_2$ since $C \overset{(k)}{\Longrightarrow}^* w_2$. The other case is symmetric. □

THEOREM A.8. *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 nondeterministic space algorithm given Algorithm 2. The algorithm, called *query*, takes two parameters, a nonterminal $X \in \mathcal{X}$ 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 $(X, w) \in \mathcal{P}$. Observe that the nonterminal parameter coincides with the head of the production. If $X \to \sigma$ is chosen, for $\sigma \in \Sigma \cup \{\varepsilon\}$, it returns. If $X \to BC$ for some nonterminals B and C is chosen, the algorithm nondeterministically 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.

We show the following invariant: *query*(ℓ, X) has an execution that returns if and only if $X \overset{(\ell)}{\Longrightarrow}^* w$ for some $w \in \Sigma^*$. It follows that *query*(k, X_0) returns if and only if $L^{(k)}(G) \neq \emptyset$. 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 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 $\ell \geq 2$. Lemma A.7 shows that 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 that 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 sequences have no more than $m-1$ steps. Therefore, the induction hypothesis shows that *query*$(\ell-1, C)$ and *query*(ℓ, B) both return, as does *query*(ℓ, X) by picking $w = BC$.

The left-to-right direction is proved by induction on the number m of times that productions are picked in an execution of *query* that returns. 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 $w = BC$ such that $(X, w) \in \mathcal{P}$. Let us assume that Case (*i*) was executed (Case (*ii*) is treated similarly). Following the assumption, both calls *query*$(\ell-1, B)$ and *query*(ℓ, C) return

ALGORITHM 2: *query*(ℓ, X)

begin
  Data: ℓ: a strictly positive integer, X: a nonterminal of the CFG $G = (\mathcal{X}, \Sigma, \mathcal{P}, X_0)$
  Result: $\exists w \in \Sigma^*$: $X \overset{(\ell)}{\Rightarrow}{}^* w$: if yes, then *query*$(\ell, X)$ has an execution that returns; else, all its executions are blocked at either assume statement.
  if $\ell < 1$ **then assume** false;
  pick $w \in \mathcal{X}^2 \cup \Sigma \cup \{\varepsilon\}$;
  assume $(X, w) \in \mathcal{P}$;
  if $w \in \Sigma \cup \{\varepsilon\}$ **then**
    return;
  else if $w \in \mathcal{X}^2$ **then**
    if $*$ **then** /* non-det. choice */
      query$(\ell-1, (w)_1)$; /* case (i) */
      query$(\ell, (w)_2)$;
    else /* case (ii) */
      query$(\ell-1, (w)_2)$;
      query$(\ell, (w)_1)$;

(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, because $(X, BC) \in \mathcal{P}$ and $\ell \geq 2$, we find 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 nondeterministic choice (i) or (ii), there is one recursive call *query*$(\ell - 1, B)$ or *query*$(\ell - 1, C)$. The other call (e.g., to *query*(ℓ, C) 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 as long as the index is no less than 1, we find that, along every execution, at most k stack frames are needed and each frame keeps track of a nonterminal that can be encoded with $\log(|G|)$ bits. Hence, we find that $L^{(k)}(G) \neq \emptyset$ can be decided in NSPACE$(k \log(|G|))$. □

B. APPENDIX: PROOF OF THE DECOMPOSITION LEMMA

LEMMA B.1 (DECOMPOSITION LEMMA). *For every* $v \in c\mathcal{D}$*:*

$$v \parallel \mathcal{S} \parallel \lozenge_{g \in \mathcal{G}} Sim_g \neq \emptyset \quad \textit{if and only if} \quad v \parallel \mathcal{S} \parallel Sim_g \neq \emptyset \quad \textit{for every } g \in \mathcal{G}.$$

PROOF. ($\Rightarrow$): By the definition of asynchronous product, there exists $x \in (v \parallel \mathcal{S} \parallel \lozenge_{g \in \mathcal{G}} Sim_g)$. Observe that the alphabet of v is $\mathcal{G}(w_\star, r_d)$, the alphabet of $\mathcal{S}$ is $\Sigma_{\mathcal{D}} \cup \Sigma_{\mathcal{C}}$, and the alphabet of each Sim_g is $\mathcal{G}(r_c) \cup \{w_c(g), w_c(\#)\}$. Hence, there exists a set of words $\{u_g \mid g \in \mathcal{G}, u_g \in (\mathcal{C}_g^{\#} \cup \{\varepsilon\})\}$ such that $x \in (v \parallel \mathcal{S} \parallel \lozenge_{g \in \mathcal{G}} u_g\, w_c(g)^*)$. Moreover, each of the u_gs appears as a scattered subword in x, and if $g \neq g'$, then the corresponding scattered subwords are disjoint. Let x_g be the word obtained from x by removing the scattered subwords for all $g' \neq g$. It follows easily from the definitions that $x_g \in (v \parallel \mathcal{S} \parallel Sim_g)$, and we are done.

($\Leftarrow$): The proof goes by induction on $|\mathcal{G}|$. The case $|\mathcal{G}| = 1$ is trivial. When $|\mathcal{G}| > 1$, we prove that, for every $\mathcal{G}_1 \subset \mathcal{G}$, $\mathcal{G}_1 \neq \emptyset$ $g \notin \mathcal{G}_1$, $x \in \lozenge_{g' \in \mathcal{G}_1} Sim_{g'}$ and $y \in Sim_g$:

$$\text{if } v \parallel \mathcal{S} \parallel x \neq \emptyset \text{ and } v \parallel \mathcal{S} \parallel y \neq \emptyset, \text{ then } v \parallel \mathcal{S} \parallel (x \lozenge y) \neq \emptyset.$$

Let $v \in c\mathcal{D}$ of length $n \geq 0$ and assume $v \parallel \mathcal{S} \parallel x \neq \emptyset$ and $v \parallel \mathcal{S} \parallel y \neq \emptyset$. Then, there exist words $\sigma_x, \sigma_y \in \mathcal{S}$ such that $v \parallel \sigma_x \parallel x \neq \emptyset$ and $v \parallel \sigma_y \parallel y \neq \emptyset$. Following the definition of asynchronous product, the projections onto $\mathcal{G}(w_\star, r_d)$ of σ_x and σ_y coincide with v, and there exist $\alpha_i, \beta_i \in (\mathcal{G}(r_c) \cup \{w_c(\#)\})^*$ for every i, $1 \leq i \leq n+1$ such that

$$\sigma_x = \alpha_1 (v)_1 \alpha_2 (v)_2 \dots \alpha_n (v)_n \alpha_{n+1}$$
$$\sigma_y = \beta_1 (v)_1 \beta_2 (v)_2 \dots \beta_n (v)_n \beta_{n+1}.$$

Moreover, since $\sigma_x, \sigma_y \in \mathcal{S}$, they obey the store rules. Thus, if $(v)_{i-1}$ is a read or a write to g, then

$$\alpha_i = r_c(g)^{k_i} w_c(\#)^{\ell_i}$$
$$\beta_i = r_c(g)^{k'_i} w_c(\#)^{\ell'_i}$$

for some nonnegative integers $k_i, \ell_i, k'_i, \ell'_i$. Let

$$\gamma_i = r_c(g)^{k_i + k'_i} w_c(\#)^{\ell_i + \ell'_i}$$

and define

$$\sigma = \gamma_1 (v)_1 \gamma_2 \dots \gamma_n (v)_n \gamma_{n+1}.$$

Clearly, σ also obeys the store rules; thus, we have that $\sigma \in \mathcal{S}$ and also, by Lemma 2.3(1), $v \parallel \sigma = \sigma$, hence that $v \parallel \sigma \in \mathcal{S}$, and finally that $v \parallel \sigma \in v \parallel \mathcal{S}$. Define σ' as the projection of σ onto alphabets of the simulators for $\mathcal{G}_1 \cup \{g\}$, namely, $\bigcup_{g' \in \mathcal{G}_1 \cup \{g\}} \Sigma_{g'}$. By definition of x and y whose respective alphabets are $\bigcup_{g' \in \mathcal{G}_1} \Sigma_{g'}$ and Σ_g, clearly $\sigma' \in (x \between y)$. Then, we conclude from Lemma 2.3(1) that $\sigma' \parallel \sigma = \sigma$, hence that $v \parallel \sigma \parallel \sigma' \in v \parallel \mathcal{S}$, next that $v \parallel \sigma \parallel \sigma' \in v \parallel \mathcal{S} \parallel (x \between y)$, and thus that $v \parallel \mathcal{S} \parallel (x \between y) \neq \emptyset$, which concludes the proof. □

C. APPENDIX: PROOF OF THE LEADER MONOTONICITY LEMMA

We now give the proof of the Leader Monotonicity Lemma (Lemma 7.10).

LEMMA C.1. *Let $L \subseteq (\Sigma_C)^*$ be a language satisfying the following condition: if $\alpha\, w_c(g)\, \beta_1 \beta_2 \in L$, then $\alpha\, w_c(g)\, \beta_1\, w_c(g)\, \beta_2 \in L$. For every $v, v' \in Proj_{\mathcal{G}(w_\star, r_d)}(\mathcal{S})[\mathtt{f}]$:*

$$\textit{if } v \parallel \mathcal{S}[\mathtt{f}] \parallel L[\mathtt{f}] \neq \emptyset \qquad \textit{and } v' \succeq_{\mathcal{G}(f_c)} v, \qquad \textit{then } v' \parallel \mathcal{S}[\mathtt{f}] \parallel L[\mathtt{f}] \neq \emptyset.$$

PROOF OF LEMMA C.1. Assume that $v \parallel \mathcal{S}[\mathtt{f}] \parallel L[\mathtt{f}] \neq \emptyset$. Then, there exists a word $v_S \in L$ such that $v \parallel \mathcal{S}[\mathtt{f}] \parallel v_S \neq \emptyset$. Furthermore, since $v' \succeq_{\mathcal{G}(f_c)} v$, there exists an increasing injection $h : \{1, \dots, |v|\} \to \{1, \dots, |v'|\}$ such that $h(i) \geq i$ and $(v)_i = (v')_{h(i)}$ for all $i \in \{1, \dots, |v|\}$. In addition, by the definition of $\succeq_{\mathcal{G}(f_c)}$, we have that $Proj_{\mathcal{G}(f_c)}(v) = Proj_{\mathcal{G}(f_c)}(v')$.

The proof proceeds by induction on the number m of positions i in v such that $h(i+1) - h(i) > 1$. If $m = 0$, then $v = v'$, and we are done. If $m > 0$, let i be the least position such that $h(i+1) - h(i) > 1$. Consider the nonempty word segment in v' given by $(v')_{h(i)+1} \cdots (v')_{h(i+1)-1}$, and define

$$new_v = (v)_1 \dots (v)_i \cdot (v')_{h(i)+1} \cdots (v')_{h(i+1)-1} \cdot (v)_{i+1} \cdots (v)_{|v|}.$$

Intuitively, new_v is the result of inserting the segment $(v')_{h(i)+1} \cdots (v')_{h(i+1)-1}$ in v. Then, there exists an injection $h' : \{1, \dots, |new_v|\} \to \{1, \dots, |v'|\}$ such that $h'(i) \geq i$ and $(new_v)_i = (v')_{h(i)}$ for all $i \in \{1, \dots, |new_v|\}$. Moreover, h' can be chosen so that the number m' of positions i such that $h'(i+1) - h'(i) > 1$ satisfies $m' < m$. Thus, if $v' \succeq_{\mathcal{G}(f_c)} new_v$ and $new_v \parallel \mathcal{S}[\mathtt{f}] \parallel L[\mathtt{f}] \neq \emptyset$, then, following the induction hypothesis, we conclude that $v' \parallel \mathcal{S}[\mathtt{f}] \parallel L[\mathtt{f}] \neq \emptyset$. In the rest of the proof, we show that $new_v \parallel \mathcal{S}[\mathtt{f}] \parallel L[\mathtt{f}] \neq \emptyset$ since $v' \succeq_{\mathcal{G}(f_c)} new_v$ has been established earlier.

We first claim that $new_v \in \mathcal{S}[\mathtt{f}]$. For this, we first observe that, since $(v)_i = (v')_{h(i)}$, the values of the store after executing $(v)_i$ and $(v')_{h(i)}$ coincide. The same holds for $(v)_{i+1}$ and $(v')_{h(i+1)}$. Finally, since $v' \in \mathcal{S}$, the claim holds. Furthermore, we have $Proj_{\mathcal{G}(f_c)}(v) = Proj_{\mathcal{G}(f_c)}(v') = Proj_{\mathcal{G}(f_c)}(new_v)$.

Along the segment $(v')_{h(i)+1} \cdots (v')_{h(i+1)-1}$, either the value of the store is overwritten (with a w_d or a w_c) or not. In the former case, the segment consists only of r_d actions. Since the alphabet of L and $\mathcal{G}(r_d)$ are disjoint and $new_v \in \mathcal{S}$, we get $new_v \parallel \mathcal{S}[\mathtt{f}] \parallel v_S \neq \emptyset$. Thus, $new_v \parallel \mathcal{S}[\mathtt{f}] \parallel L \neq \emptyset$, and we are done.

When the store is overwritten, the segment consists of w_d, r_d, and w_c actions containing at least one occurrence of an action in $\mathcal{G}(w_\star)$. In this case, we show that there exists $v'_S \in L$ such that $new_v \parallel \mathcal{S}[\mathtt{f}] \parallel v'_S \neq \emptyset$. If the segment contains no occurrence of letters in $\mathcal{G}(w_c)$ actions (but at least one w_d action), then setting v'_S to v_S suffices because the alphabet of L and $\mathcal{G}(w_d)$ are disjoint. If the segment contains at least one action in $\mathcal{G}(w_c)$ actions, we observe that, since $new_v \in \mathcal{S}[\mathtt{f}]$, each occurrence of $w_c(g)$ in new_v is preceded by an occurrence of $f_c(g)$. To obtain v'_S, it thus suffices to add those occurrences of actions in $\mathcal{G}(w_c)$ to v_S. This results in a word of L because we assumed that, if $\alpha\ f_c(g)\ \beta_1\beta_2 \in L$, then $\alpha\ f_c(g)\ \beta_1\ w_c(g)\ \beta_2 \in L$. □

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for the careful reading and their effort in helping to improve the presentation. In particular, we thank the referee who suggested a much simpler proof of the Simulation Lemma.

REFERENCES

Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. 1996. General decidability theorems for infinite-state systems. In *LICS'96*. IEEE Computer Society, 313–321. DOI:http://dx.doi.org/10.1109/LICS.1996.561359

D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. 2007. The computational power of population protocols. *Distributed Computing* 20, 4, 279–304.

Krzysztof R. Apt and Dexter C. Kozen. 1986. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters* 22, 6, 307–309. DOI:http://dx.doi.org/10.1016/0020-0190(86)90071-2

Sanjeev Arora and Boaz Barak. 2009. *Computational Complexity–A Modern Approach*. Cambridge University Press, New York, NY.

Barron Brainerd. 1967. An analog of a theorem about context-free languages. *Information and Control* 11, 5–6, 561–567. DOI:http://dx.doi.org/10.1016/S0019-9958(67)90771-1

Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. 2008. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS'08*. Lecture Notes in Computer Science, Vol. 4963. Springer, Berlin, 33–47. DOI:http://dx.doi.org/10.1007/978-3-540-78800-3_4

Giorgio Delzanno. 2003. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design* 23, 3, 257–301. DOI:http://dx.doi.org/10.1023/A:1026276129010

Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. 2002. Towards the automated verification of multithreaded Java programs. In *TACAS'02. Lecture Notes in Computer Science*, Vol. 2280. Springer, Berline, 173–187. DOI:http://dx.doi.org/10.1007/3-540-46002-0_13

Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. 2010. Parameterized verification of ad hoc networks. In *CONCUR'10*. Lecture Notes in Computer Science, Vol. 6269. Springer, Berline, 313–327. DOI:http://dx.doi.org/10.1007/978-3-642-15375-4_22

Rayna Dimitrova and Andreas Podelski. 2008. Is lazy abstraction a decision procedure for broadcast protocols? In *VMCAI'08*. Lecture Notes in Computer Science, Vol. 4905. Springer, Berlin, 98–111. DOI:http://dx.doi.org/10.1007/978-3-540-78163-9_12

E. Allen Emerson and Vineet Kahlon. 2003. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME'03*. Lecture Notes in Computer Science, Vol. 2860. Springer, Berlin, 247–262. DOI:http://dx.doi.org/10.1007/978-3-540-39724-3_22

E. Allen Emerson and Kedar S. Namjoshi. 1998. Verification of parameterized bus arbitration protocol. In *CAV'98*. Lecture Notes in Computer Science, Vol. 1427. Springer, Berlin, 452–463. DOI:http://dx.doi.org/10.1007/BFb0028766

Javier Esparza. 2014. Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In *STACS'14: 31st International Symposium on Theoretical Aspects of Computer Science (Leibniz International Proceedings in Informatics (LIPIcs)*, Vol. 25. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 1–10. DOI:http://dx.doi.org/10.4230/LIPIcs.STACS.2014.1

Javier Esparza, Alain Finkel, and Richard Mayr. 1999. On the verification of broadcast protocols. In *LICS'99*. IEEE Computer Society, 352–359. DOI:http://dx.doi.org/10.1109/LICS.1999.782630

Javier Esparza, Pierre Ganty, Stefan Kiefer, and Michael Luttenberger. 2011. Parikh's theorem: A simple and direct automaton construction. *Information Processing Letters* 111, 614–619. DOI:http://dx.doi.org/10.1016/j.ipl.2011.03.019

Alain Finkel and Jérôme Leroux. 2002. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FSTTCS'02*. Lecture Notes in Computer Science, Vol. 2556. Springer, Berlin, 145–156. DOI:http://dx.doi.org/10.1007/3-540-36206-1_14

Pierre Ganty and Rupak Majumdar. 2012. Algorithmic verification of asynchronous programs. *ACM Transactions on Programming Languages and Systems* 34, 1, Article 6, 48 pages. DOI:http://dx.doi.org/10.1145/2160910.2160915

Steven M. German and A. Prasad Sistla. 1992. Reasoning about systems with many processes. *Journal of the ACM* 39, 3, 675–735. DOI:http://dx.doi.org/10.1145/146637.146681

Rachid Guerraoui and Eric Ruppert. 2007. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing* 20, 3, 165–177. DOI:http://dx.doi.org/10.1007/s00446-007-0042-0

Matthew Hague. 2011. Parameterised pushdown systems with non-atomic writes. In *Proceedings of FSTTCS'11 (LIPIcs)*, Vol. 13. Schloss Dagstuhl, 457–468. DOI:http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2011.457

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation* (1st ed.). Addison-Wesley, New York, NY.

Neil D. Jones and William T. Laaser. 1974. Complete problems for deterministic polynomial time. In *STOC'74*. ACM, New York, NY, 40–46. DOI:http://dx.doi.org/10.1145/800119.803883

Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2010. Dynamic cutoff detection in parameterized concurrent programs. In *CAV'10*. Lecture Notes in Computer Science, Vol. 6174. Springer, Berlin, 645–659. DOI:http://dx.doi.org/10.1007/978-3-642-14295-6_55

Christine Laurendeau and Michel Barbeau. 2007. Secure anonymous broadcasting in vehicular networks. In *LCN'07*. IEEE Computer Society, 661–668. DOI:http://dx.doi.org/10.1109/LCN.2007.37

Kenneth L. McMillan. 1998. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV'98*. Lecture Notes in Computer Science, Vol. 1427. Springer, Berlin, 110–121. DOI:http://dx.doi.org/10.1007/BFb0028738

M. Rubenstein, A. Cornejo, and R. Nagpal. 2014. Programmable self-assembly in a thousand-robot swarm. *Science* 345, 6198.

M. Sipser. 1996. *Introduction to the Theory of Computation*. International Thomson Publishing.

Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2010. Model-checking parameterized concurrent programs using linear interfaces. In *CAV'10*. Lecture Notes in Computer Science, Vol. 6174. Springer, Berlin, 629–644. DOI:http://dx.doi.org/10.1007/978-3-642-14295-6_54

Mahesh Viswanathan and Rohit Chadha. 2009. Deciding branching time properties for asynchronous programs. *Theoretical Computer Science* 410, 42, 4169–4179. DOI:http://dx.doi.org/10.1016/j.tcs.2009.01.021

J. Werfel, K. Petersen, and R. Nagpal. 2014. Designing collective behavior in a termite-inspired robot construction team. *Science* 343, 6172.

R. Wood, R. Nagpal, and G.-Y. Wei. 2013. Flight of the robobees. *Scientific American* 308, 3.

Received November 2014; revised October 2015; accepted October 2015