Complexity of Liveness in Parameterized Systems

Peter Chini
TU Braunschweig, Germany
p.chini@tu-braunschweig.de

Roland Meyer
TU Braunschweig, Germany
roland.meyer@tu-braunschweig.de

Prakash Saivasan
TU Braunschweig, Germany
p.saivasan@tu-braunschweig.de

Abstract

We investigate the fine-grained complexity of liveness verification for leader contributor systems. These consist of a designated leader thread and an arbitrary number of identical contributor threads communicating via a shared memory. The liveness verification problem asks whether there is an infinite computation of the system in which the leader reaches a final state infinitely often. Like its reachability counterpart, the problem is known to be NP-complete. Our results show that, even from a fine-grained point of view, the complexities differ only by a polynomial factor.

Liveness verification decomposes into reachability and cycle detection. We present a fixed point iteration solving the latter in polynomial time. For reachability, we reconsider the two standard parameterizations. When parameterized by the number of states of the leader $\mathtt{L}$ and the size of the data domain $\mathtt{D}$, we show an $(\mathtt{L}+\mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$-time algorithm. It improves on a previous algorithm, thereby settling an open problem. When parameterized by the number of states of the contributor $\mathtt{C}$, we reuse an $\mathcal{O}^*(2^{\mathtt{C}})$-time algorithm. We show how to connect both algorithms with the cycle detection to obtain algorithms for liveness verification. The running times of the composed algorithms match those of reachability, proving that the fine-grained lower bounds for liveness verification are met.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Problems, reductions and completeness

Keywords and phrases Liveness Verification, Fine-Grained Complexity, Parameterized Systems

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2019.37

Related Version A full version of the paper is available as [9] and `https://arxiv.org/abs/1909.12004`.

Acknowledgements We thank Arnaud Sangnier for helpful discussions.

1 Introduction

We study the fine-grained complexity of liveness verification for parameterized systems formulated in the leader contributor model. The model [26, 16] assumes a distinguished leader thread interacting (via a shared memory) with a finite but arbitrary number of indistinguishable contributor threads. The liveness verification problem [14] asks whether there is an infinite computation of the system in which the leader visits a set of final states infinitely often. Fine-grained complexity [13, 10] studies the impact of parameters associated with an algorithmic problem on the problem's complexity like the influence of the contributor size on the complexity of liveness verification. The goal is to develop deterministic algorithms that are provably optimal. We elaborate on the three ingredients of our study.

The leader contributor model has attracted considerable attention [26, 16, 14, 31, 17, 23, 7]. From a modeling point of view, a variety of systems can be formulated as anonymous entities interacting with a central authority, examples being client-server applications, resource-management systems, and distributed protocols on wireless sensor networks. From an algorithmic point of view, the model has led to positive surprises. Hague [26] proved decidability of reachability even in a setting where the system components are pushdown automata. La Torre et al. [31] generalized the result to any class of components that satisfies mild assumptions, the most crucial of which being computability of downward closures. As for the complexity, Esparza et al. [16, 17] proved PSPACE-completeness for Hague's model and NP-completeness in the setting where the components are given by finite-state automata. The liveness problem was first studied in [14]. Interestingly, liveness has the same complexity as reachability, it is NP-complete for finite-state systems. Fortin et al. [23] generalized the study to LTL-definable properties and gave conditions for NEXPTIME-completeness.

Fine-grained complexity is a field within parameterized complexity [13, 10]. Parameterized complexity intends to explain the following gap between theory and practice that is observed throughout algorithmics. Despite a high worst-case complexity, tools may have an easy time solving a problem. Parameterized complexity argues that measuring the complexity of a problem in terms of the size of the input, typically denoted by n, is too rough. One should consider further parameters k that capture the shape of the input or the solution sought. Then the gap is due to the fact that tools implement an algorithm running in time $f(k) \cdot poly(n)$. Here, f may be an exponential, but it only depends on the parameter, and that parameter is small in practice. Problems solvable by such an algorithm are called fixed-parameter tractable and belong to the complexity class FPT. Fine-grained complexity is the study of the precise function f that is needed, via upper and lower bound arguments.

The fine-grained complexity of the reachability problem for the leader contributor model was studied in our previous work [7]. We assumed that the components are finite state and considered two parameterizations. When parameterized by the size of the contributors $\mathtt{C}$, we showed that reachability can be solved in time $\mathcal{O}^*(2^{\mathtt{C}})$. The notation $\mathcal{O}^*$ suppresses polynomial factors in the running time. Interestingly, this is the best one can hope for. An algorithm with a subexponential dependence on $\mathtt{C}$, to be precise an algorithm running in time $2^{o(\mathtt{C})}$, would contradict the so-called exponential time hypothesis (ETH). The ETH [28] is a standard hardness assumption in parametrized complexity that is used to derive relative lower bounds. The second parameterization is by the size of the leader $\mathtt{L}$ and the size of the data domain $\mathtt{D}$. We gave an algorithm running in time $(\mathtt{LD})^{\mathcal{O}(\mathtt{LD})}$. Interestingly, the lower bound is only $2^{o((\mathtt{L}+\mathtt{D})\cdot\log(\mathtt{L}+\mathtt{D}))}$. Being away a quadratic factor in the exponent means a substantial gap for a deterministic algorithm.

In the present paper, we study the fine-grained complexity of the liveness verification problem. We assume finite-state components and consider the same parameterization as for reachability. The surprise is in the parameterization by $\mathtt{L}$ and $\mathtt{D}$. We give an algorithm running in time $(\mathtt{L}+\mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$. This matches the lower bound and closes the gap for reachability. When parameterized by the size of the contributors, we obtain an $\mathcal{O}^*(2^{\mathtt{C}})$ algorithm.

To explain the algorithms, note that a live computation decomposes into a prefix and an accepting cycle. Finding prefixes is a matter of reachability. We show how to combine reachability algorithms with a cycle detection to obtain algorithms that find live computations. The resulting algorithms will run in time $\mathcal{O}(\mathit{Reach}(\mathtt{L},\mathtt{D},\mathtt{C}) \cdot \mathit{Cycle}(\mathtt{L},\mathtt{D},\mathtt{C}))$ where $\mathit{Reach}(\mathtt{L},\mathtt{D},\mathtt{C})$ denotes the running time of the invoked reachability algorithm and $\mathit{Cycle}(\mathtt{L},\mathtt{D},\mathtt{C})$ that of the cycle detection. This result allows for considering reachability and cycle detection separately.

Our first main contribution is an algorithm for reachability when $\mathtt{L}$ and $\mathtt{D}$ are given as parameters. It runs in time $(\mathtt{L}+\mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$ and significantly improves upon the $(\mathtt{LD})^{\mathcal{O}(\mathtt{LD})}$-time algorithm from [7]. Moreover, it is optimal in the fine-grained sense. It closes the gap between

upper and lower bound. The algorithm works over sketches of computations. A sketch is valid if there is an actual computation corresponding to it. In [7], we performed a single validity check for each sketch. Here, we show that valid sketches can be build up inductively from small sketches. To this end, we interleave validity checks with compression phases. Our algorithm is a dynamic programming on small sketches, exploiting the inductive approach.

Our second main result is an algorithm for detecting cycles. We show that the problem is actually solvable in polynomial time. Technically, we employ a characterization of cycles via (certain) SCC decompositions of the contributor automaton. These decompositions can be computed by a fixed point iteration invoking Tarjan's algorithm [35] in polynomial time.

Since $\mathit{Cycle}(\mathtt{L}, \mathtt{D}, \mathtt{C})$ is polynomial, liveness has the same complexity as reachability also in the fine-grained sense. With the above result, we obtain the mentioned algorithms for liveness by composing the reachability algorithms with the cycle detection.

Related Work. The parameterized complexity has also been studied for other verification problems. Farzan and Madhusudan [18] consider the problem of predicting atomicity violations. Depending on the synchronization, they obtain an efficient fine-grained algorithm resp. prove an FPT-algorithm unlikely. In [15], the authors give an efficient (fine-grained) algorithm for the problem of checking TSO serializability. In [5], we studied the fine-grained complexity of bounded context switching [33], including lower bounds on the complexity. In [7], we gave a parameterized analysis of the bounded write-stage restriction, a generalization of bounded context switching [2]. The problem turns out to be hard for different parameterizations, and has a large number of hard instances. In a series of papers [20, 19, 36], Fernau et al. studied FPT-algorithms for problems from automata theory.

Related to leader contributor systems are broadcast networks (ad-hoc networks) [34, 12]. These consist of an arbitrary number of finite-state contributors that communicate via message passing. There is no leader. This has an impact on the complexity of safety [11, 24] and liveness [6, 3] verification, which drops from NP (leader contributor systems) to P.

More broadly, the verification of parameterized systems is an active field of research [4]. Prominent approaches are well-structuredness arguments [1, 21] and cut-off results [25]. Well-structuredness means the transition relation is monotonic wrt. a well-quasi ordering on the configurations, a combination that leads to surprising decidability results. A cut-off is a bound on the size of system instances such that correctness of the bounded instances entails correctness of all instances. Our algorithm uses different techniques. We give a reduction from liveness to reachability combined with a polynomial-time cycle check. Reductions from liveness to reachability or safety are recently gaining popularity in verification [29, 32, 27]. For reachability, we then rely on techniques from parameterized complexity [13, 10], namely identifying combinatorial objects to iterate over and dynamic programming.

2 Leader Contributor Systems and the Liveness Problem

We introduce leader contributor systems and the leader contributor liveness problem of interest following [26, 16, 14]. Moreover, we give a short introduction to fine-grained complexity. For standard textbooks, we refer to [22, 10, 13].

Leader Contributor Systems. A *leader contributor system* consists of a designated leader thread communicating with a number of identical contributor threads via a shared memory. Formally, the system is a tuple $\mathcal{S} = (D, a^0, P_L, P_C)$ where D is the finite domain of the shared memory and $a^0 \in D$ is the initial memory value. The leader P_L and the contributor P_C

are abstractions of concrete threads making visible the interaction with the memory. They are defined as finite state automata over the alphabet $Op(D) = \{!a, ?a \mid a \in D\}$ of memory operations. Here, $!a$ denotes a write of a to the memory, $?a$ denotes a read of a. The leader is given by the tuple $P_L = (Op(D), Q_L, q_L^0, \delta_L)$ where Q_L is the set of states, $q_L^0 \in Q_L$ is the initial state, and $\delta_L \subseteq Q_L \times (Op(D) \cup \{\varepsilon\}) \times Q_L$ is the transition relation. We extend the relation to words in $Op(D)^*$ and usually write $q \xrightarrow{w}_L q'$ for $(q, w, q') \in \delta_L$. The contributor is defined similarly, by $P_C = (Op(D), Q_C, q_C^0, \delta_C)$.

The possible interactions of a thread with the memory depend on the current memory value and the internal state of the thread. To keep track of this information, we use *configurations*. These are tuples of the form $(q, a, pc) \in CF^t = Q_L \times D \times Q_C^t$. Here, pc is a vector storing the current state of each contributor, and there are $t \in \mathbb{N}$ contributors participating in the computation. The number of participating contributors can be arbitrary, but will be fixed throughout the computation. Therefore, the set of all configurations is given by $CF = \bigcup_{t \in \mathbb{N}} CF^t$. A configuration is called *initial* if it is of the form (q_L^0, a^0, pc^0) where $pc^0(i) = q_C^0$ for each $i \in [1..t]$. We use projections to access the components of a configuration. Let π_L and π_D denote the projections to the leader state resp. the memory content, $\pi_L((q, a, pc)) = q$ and $\pi_D((q, a, pc)) = a$. The map π_C projects a configuration to the set of contributor states present in pc, $\pi_C((q, a, pc)) = \{pc(i) \mid i \in [1..t]\}$.

The current configuration of $\mathcal{S}$ may change due to an interaction with the memory or an internal transition. We capture such changes by a labeled transition relation among configurations, $\to \subseteq CF \times (Op(D) \cup \{\varepsilon\}) \times CF$. It contains transitions induced by the leader and by the contributor. We focus on the former. If there is a write $q \xrightarrow{!b}_L q'$ of the leader, we get $(q, a, pc) \xrightarrow{!b} (q', b, pc)$. Similarly, a read $q \xrightarrow{?a}_L q'$ induces $(q, a, pc) \xrightarrow{?a} (q', a, pc)$. Note that the current memory value has to match the read symbol. An internal transition $q \xrightarrow{\varepsilon}_L q'$ yields $(q, a, pc) \xrightarrow{\varepsilon} (q', a, pc)$. For the transitions induced by the contributors, let $pc(i) = p$ and $pc' = pc[i = p']$, meaning $pc'(i) = p'$ and pc' coincides with pc in all other components. A transition $p \xrightarrow{!b/?a/\varepsilon}_C p'$ yields $(q, a, pc) \xrightarrow{!b/?a/\varepsilon} (q, b/a, pc')$, like for the leader. Note that transitions are only defined among configurations involving the same number of contributors. It is convenient to assume that the leader never writes $!a$ and immediately reads $?a$ again. In this case, we could replace the corresponding read transition by ε.

The transition relation $\to$ is generalized to words, denoted by $c \xrightarrow{w} c'$ with $w \in Op(D)^*$. We call such a sequence a *computation* of $\mathcal{S}$. We also write $c \to^* c'$ if there is a word w with $c \xrightarrow{w} c'$, and $c \to^+ c'$ if w has length at least 1. An *infinite computation* is a sequence $\sigma = c^0 \to c^1 \to \dots$ of infinitely many transitions. We call it *initialized* if c^0 is an initial configuration. Since σ involves infinitely many configurations but the set Q_L is finite, there are states of the leader that occur infinitely often along the computation. We denote the set of these states by $\mathrm{Inf}(\sigma) = \{q \in Q_L \mid \exists^\infty i : q = \pi_L(c^i)\}$.

Leader Contributor Liveness. The *leader contributor liveness problem* is the task of deciding whether the leader satisfies a liveness specification while interacting with a number of contributors. Formally, given a leader contributor system $\mathcal{S} = (D, a^0, P_L, P_C)$ and a set of final states $F \subseteq Q_L$ encoding the specification, the problem asks whether there is an initialized infinite computation σ such that the leader visits F infinitely often along σ. Since F is finite, this is equivalent to $\mathrm{Inf}(\sigma) \cap F \neq \emptyset$. In this case, σ is called a *live computation*.

Leader Contributor Liveness (LCL)
Input: A leader contributor system $\mathcal{S} = (D, a^0, P_L, P_C)$ and final states $F \subseteq Q_L$.
Question: Is there an infinite initialized computation σ such that $\mathrm{Inf}(\sigma) \cap F \neq \emptyset$?

Fine-Grained Complexity. The problem LCL is known to be NP-complete [14]. Despite its hardness, it may still admit efficient deterministic algorithms the running times of which depend exponentially only on certain parameters. To find parameters that allow for the construction of such algorithms, one examines the *parameterized complexity* of LCL. Note that the name does not refer to parameterized systems. It stems from measuring the complexity not only in the size of the input but also in the mentioned parameters.

Let Σ be an alphabet. Unlike in classical complexity theory where we consider problems over Σ^*, a *parameterized problem* P is a subset of $\Sigma^* \times \mathbb{N}$. Inputs to P are pairs (x, k) with the second component k being referred to as the *parameter*. Problem P is called *fixed-parameter tractable* if it admits a deterministic algorithm deciding membership in P for pairs (x, k) in time $f(k) \cdot |x|^{\mathcal{O}(1)}$. Here, f is a computable function that only depends on k. Since f usually dominates the polynomial, the running time of the algorithm is denoted by $\mathcal{O}^*(f(k))$.

While finding an upper bound for the function f amounts to coming up with an efficient algorithm, lower bounds on f are obtained relative to hardness assumptions. One of the standard assumptions is the *exponential time hypothesis* (ETH) [28]. It asserts that 3-SAT cannot be solved in time $2^{o(n)}$ where n is the number of variables in the input formula. The lower bound is transported to the problem of interest via a reduction from 3-SAT. Then, f cannot drop below a certain bound unless ETH fails. It is a task of *fine-grained complexity* to find the *optimal* function f, where upper and lower bound match.

We conduct fine-grained complexity analyses for two parameterizations of LCL. First, we consider LCL(L, D), the parameterization by the number of states in the leader L and the size of the data domain D. We show an $(\mathtt{L} + \mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$-time algorithm, matching the lower bound for LCL from [7]. The second parameterization LCL(C) is by the number of states of the contributor C. We give an algorithm running in time $\mathcal{O}^*(2^{\mathtt{C}})$. It also matches the known lower bound [7]. Therefore, both algorithms are optimal in the fine-grained sense. The parameterizations LCL(L) and LCL(D) are unlikely to be fixed-parameter tractable. These problems are hard for W[1], a complexity class comprising intractable problems [7].

3 Dividing Liveness along Interfaces

A live computation naturally decomposes into a prefix and a cycle. This means that solving LCL amounts to finding both, a prefix computation and a cyclic computation. However, we need to guarantee that the computations can be linked. The prefix should lead to a configuration that the cycle loops on. Since there are infinitely many configurations, we introduce the finite domain of interfaces. An interface abstracts a configuration to its leader state, memory value, and set of contributor states. Hence, an interface can be seen as a summary of those configurations that are suitable for linking prefix and cycle.

Our algorithm to solve LCL works as follows. We start a reachability algorithm for the leader contributor model on the final states that the live computation should visit. After a modification, the algorithm outputs all interfaces witnessing prefixes to those states. Let $\mathit{Reach}(\mathtt{L}, \mathtt{D}, \mathtt{C})$ denote the running time of the reachability algorithm. We show that the obtained set of interfaces will be of size at most $\mathit{Reach}(\mathtt{L}, \mathtt{D}, \mathtt{C})$. We iterate over the interfaces and pass each to a cycle detection which works over interfaces instead of configurations. If a cycle was found, a live computation exists. Let $\mathit{Cycle}(\mathtt{L}, \mathtt{D}, \mathtt{C})$ be the time needed for a single cycle detection. Then, the running time of the algorithm can be estimated as follows.

▶ **Theorem 1.** LCL *can be solved in time* $\mathcal{O}(\mathit{Reach}(\mathtt{L}, \mathtt{D}, \mathtt{C}) \cdot \mathit{Cycle}(\mathtt{L}, \mathtt{D}, \mathtt{C}))$.

The first step in proving Theorem 1 is to decompose live computations into prefixes and cycles. To be precise, we aim for a decomposition where the cycle is saturated in the sense that the initial configuration already contains all contributor states that will be encountered

along the cycle. Knowing these states in advance eases technical arguments when finding cycles in Section 5. Formally, a cyclic computation $\tau = c \to^* c$ is called *saturated* if for each configuration c' in τ, we have $\pi_C(c') \subseteq \pi_C(c)$. We write $c \to^*_{sat} c$ for a saturated cycle. The following lemma yields the desired decomposition. If not stated otherwise, proofs and details for the current section are provided in the full version of the paper.

▶ **Lemma 2.** *There is an infinite initialized computation σ with $\mathrm{Inf}(\sigma) \cap F \neq \emptyset$ if and only if there is a finite initialized computation $c^0 \to^* c \to^+_{sat} c$ with $\pi_L(c) \in F$.*

We would like to decompose LCL into finding prefix and cycle. But we need to ensure that the found computations can be linked at an explicit configuration. For avoiding the latter, we introduce interfaces. An *interface* is a triple $I = (S, q, a) \in \mathcal{P}(Q_C) \times Q_L \times D$ consisting of a set of contributor states S, a state of the leader q, and a memory value a. A configuration c *matches* the interface I if $\pi_C(c) = S$, $\pi_L(c) = q$, and $\pi_D(c) = a$. We denote this by $I(c)$, interpreting I as a predicate. The set of interfaces is denoted by IF. The following lemma shows that the notion allows for decomposing LCL. We can search for prefixes and cycles separately. The lemma provides the arguments needed to complete the proof of Theorem 1.

▶ **Lemma 3.** *Let $I \in \mathrm{IF}$. There is a computation $c^0 \to^* c \to^+_{sat} c$ with $I(c)$ if and only if there are computations $d^0 \to^* d$ and $f \to^+_{sat} f$ with $I(d) \wedge I(f)$.*

In the following, we turn to our main contributions. We present algorithms for reachability and cycle detection and obtain precise values for $\mathit{Reach}(\mathtt{L}, \mathtt{D}, \mathtt{C})$ and $\mathit{Cycle}(\mathtt{L}, \mathtt{D}, \mathtt{C})$. Further, we modify the reachability algorithms to output interfaces. Then we invoke Theorem 1 to derive algorithms for LCL. The first problem that we consider is finding prefixes.

Leader Contributor Reachability (LCR)
Input: A leader contributor system $\mathcal{S} = (D, a^0, P_L, P_C)$ and final states $F \subseteq Q_L$.
Question: Is there an initialized computation $c^0 \to^* c$ with $\pi_L(c) \in F$?

The problem LCR is NP-complete [16]. Its complexity $\mathit{Reach}(\mathtt{L}, \mathtt{D}, \mathtt{C})$ depends on the parameterization. There are two standard parameterizations [7, 8]: $\mathsf{LCR}(\mathtt{L}, \mathtt{D})$ and $\mathsf{LCR}(\mathtt{C})$.

For the parameterization by L and D, we present an algorithm solving $\mathsf{LCR}(\mathtt{L}, \mathtt{D})$ in time $(\mathtt{L}+\mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$. The algorithm solves an open problem [7] by matching the known lower bound: unless ETH fails, LCR cannot be solved in time $2^{o((\mathtt{L}+\mathtt{D})\cdot\log(\mathtt{L}+\mathtt{D}))}$. The algorithm and its modification for obtaining interfaces are presented in Section 4.

▶ **Theorem 4.** $\mathsf{LCR}(L, D)$ *can be solved in time $(L + D)^{\mathcal{O}(L+D)}$.*

For $\mathsf{LCR}(\mathtt{C})$, we modify the reachability algorithm from [7, 8] so that it outputs interfaces that witness prefixes. We recall the result on the complexity of the algorithm.

▶ **Theorem 5** ([7, 8])**.** $\mathsf{LCR}(C)$ *can be solved in time $\mathcal{O}(2^C \cdot C^4 \cdot L^2 \cdot D^2)$.*

The second task to solve LCL is detecting cycles. We formalize the problem. It takes an interface and asks for a saturated cycle on a configuration that matches the interface.

Saturated Cycle (CYC)
Input: A leader contributor system $\mathcal{S} = (D, a^0, P_L, P_C)$ and an interface $I \in \mathrm{IF}$.
Question: Is there a computation $c \to^+_{sat} c$ with $I(c)$?

We present an algorithm solving CYC in polynomial time. Key to the algorithm is a fixed point iteration over certain subgraphs of the contributor. Details are postponed to Section 5.

▶ **Theorem 6.** CYC *can be solved in time* $\mathcal{O}(D^2 \cdot (C^2 + L^2 \cdot D^2))$.

The theorem shows that $Cycle(\mathtt{L}, \mathtt{D}, \mathtt{C})$ is polynomial. Hence, by Theorem 1, we obtain that LCL can be solved in time $\mathcal{O}^*(Reach(\mathtt{L}, \mathtt{D}, \mathtt{C}))$. This means that liveness verification and safety verification in the leader contributor model only differ by a polynomial factor. Taking the precise values for $Reach(\mathtt{L}, \mathtt{D}, \mathtt{C})$ into account, Theorem 1 yields the following.

▶ **Corollary 7.** $\mathsf{LCL}(L, D)$ *can be solved in time* $(L + D)^{\mathcal{O}(L+D)}$.

▶ **Corollary 8.** $\mathsf{LCL}(C)$ *can be solved in time* $\mathcal{O}(2^C \cdot L \cdot D^2 \cdot (L \cdot C^4 + D \cdot C^2 + L^2 \cdot D^3))$.

For the latter result, we are actually more precise in determining the time complexity than stated in Theorem 1. Both obtained algorithms are optimal. They match the corresponding lower bounds for LCL that carry over from reachability [7]. Unless ETH fails, LCL cannot neither be solved in time $2^{o((\mathtt{L}+\mathtt{D})\cdot\log(\mathtt{L}+\mathtt{D}))}$ nor in time $2^{o(\mathtt{C})}$.

4 Reachability Parameterized by Leader and Domain

We present the algorithm for $\mathsf{LCR}(\mathtt{L}, \mathtt{D})$. It runs in time $(\mathtt{L} + \mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$ and therefore proves Theorem 4. Moreover, with the results from Section 3 and 5, the algorithm can be utilized for solving LCL in time $(\mathtt{L} + \mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$. Like in [7], the algorithm relies on a notion of witnesses. These are sketches of computations. A witness is valid if there is an actual computation following the sketch. Validity can be checked in polynomial time.

The algorithm from [7] iterates over all witnesses and tests validity for each. Hence, the time complexity of the algorithm is proportional to $(\mathtt{LD})^{\mathcal{O}(\mathtt{LD})}$, the number of considered witnesses. Key to our new algorithm is the fact that we can restrict to so-called short witnesses. These are sketches of loop-free computations. We show that validity of witnesses can be checked inductively from validity of short witnesses. We exploit the inductivity by a dynamic programming. It runs in time proportional to $(\mathtt{L} + \mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$, the number of short witnesses. This yields the desired complexity as stated in Theorem 4.

4.1 Witnesses and Validity

We introduce witnesses and recall the notion of validity. Afterwards, we elaborate on the main idea of our new algorithm: restricting to short witnesses for checking validity.

Intuitively, a witness is a compact way to represent computations of a leader contributor system. From a computation, a witness only stores the actions of the leader and the positions where memory symbols were written by a contributor for the first time. We call these positions *first writes*. From such a position on, we can assume an unbounded supply of the corresponding memory symbol. There is always a copy of a contributor waiting to provide it.

Formally, a *witness* is a triple $x = (w, q, \sigma)$. The word $w = (q_1, a_1)(q_2, a_2) \dots (q_n, a_n)$ represents the run of the leader. It is a sequence from $(Q_L \times (D \uplus \{\bot\}))^*$, containing leader states potentially combined with a memory value. The state $q \in Q_L$ is the target of the leader run. First-write positions are specified by $\sigma : [1..k] \to [1..n]$, a monotonically increasing map where $k \leq \mathtt{D}$. The number of first-write positions k is called the *order* of x. We denote it by $ord(x) = k$. Moreover, we use Wit for the set of all witnesses. A witness $x = (w, q, \sigma) \in Wit$ is called *initialized* if w begins in the initial state q_L^0 of the leader automaton.

If a witness corresponds to an actual computation, we call it valid. This means, the witness encodes a proper run of the leader and moreover, the first writes along the run can be provided by the contributors. Since the definition of witnesses only specifies first-write positions but not values, we need the notion of first-write sequences. The latter will allow for the definition of validity.

A *first-write sequence* is a sequence of data values $\beta \in D^{\leq \mathtt{D}}$ that are all different. Formally, $\beta_i \neq \beta_j$ for $i \neq j$. We use FW to denote the set of all those sequences. Given a witness $x = (w, q, \sigma)$, we define its validity with respect to a first-write sequence β of length $ord(x)$. For being valid, x has to be *leader valid along* β and *contributor valid along* β. We make both notions more precise. Details regarding this section including formal definitions are available in the full version of the paper.

Leader Validity. The witness is *leader valid along* β if w encodes a run of the leader that reaches state q. Reading during the run is restricted to symbols from β: the ℓ-th symbol β_ℓ is available for reading once the run arrives at position $\sigma(\ell)$. Formally, the encoding depends on the memory values a_i. If $a_i \neq \bot$, the leader has a transition $q_i \xrightarrow{!a_i}_L q_{i+1}$. If $a_i = \bot$, the leader either has an ε-transition or reads a symbol available at position i, from the set $S_\beta(i) = \{\beta_\ell \mid \sigma(\ell) \leq i\}$. We use $\mathrm{LValid}_\beta(x)$ to indicate that x is leader valid along β.

Contributor Validity. The witness is *contributor valid along* β if the contributors can provide the first writes for w in the order indicated by σ. Let us focus on the i-th first write β_i. Providing β_i is a question of reachability of the set $Q_i = \{p \mid \exists p' : p \xrightarrow{!\beta_i}_C p'\}$ in the contributor automaton. More precise, we need a contributor that reaches Q_i while reading only symbols available along w. This means that reading is restricted to earlier first writes and symbols written by the leader during w up to position $\sigma(i)$.

Let $Expr(x, \beta_1 \dots \beta_{i-1})$ be the language of available reads. We say that x is *valid for the i-th first write of* β if Q_i is reachable by a contributor while reading is restricted to $Expr(x, \beta_1 \dots \beta_{i-1})$. We use $\mathrm{CValid}^i_\beta(x)$ to indicate this validity. If x is valid for all first writes, it is *contributor valid along* β. Formally, $\mathrm{CValid}_\beta(x) = \bigwedge_{i \in [1..ord(x)]} \mathrm{CValid}^i_\beta(x)$.

With leader and contributor validity in place, we can define x to be *valid along* β if $\mathrm{LValid}_\beta(x) \wedge \mathrm{CValid}_\beta(x)$. Again, we use predicate notation. We write $\mathrm{Valid}_\beta(x)$ if x is valid along β. Validity of a witness along a first-write sequence can be checked in polynomial time.

▶ **Lemma 9.** *Let $x \in Wit$ and $\beta \in \mathrm{FW}$. $\mathrm{Valid}_\beta(x)$ can be evaluated in polynomial time.*

The algorithm from [7] iterates over witnesses and invokes Lemma 9 to check validity. The following lemma proves the correctness: validity indicates the existence of a computation.

▶ **Lemma 10.** *Let $q \in Q_L$. There is an initialized computation $c^0 \to^* c$ with $\pi_L(c) = q$ if and only if there is an initialized $x = (w, q, \sigma) \in Wit$ and a $\beta \in FW$ so that $\mathrm{Valid}_\beta(x)$.*

For obtaining a tractable algorithm, we would like to restrict to short witnesses when checking validity. These are witnesses encoding a loop-free run of the leader. The following two observations are crucial to our development.

Leader validity can be checked inductively on short witnesses. A witness x can be written as a product $x = x_1 \times x_2 \times \dots \times x_{k+1}$ of smaller witnesses. Each x_i encodes that part of the leader run of x happening between two first-write positions $\sigma(i-1)$ and $\sigma(i)$. The *witness concatenation* $\times$ appends these runs. Each x_i can assumed to be a short witness. There is no need for recording loops of the leader between first writes. We can cut them out.

Assume $y = x_1 \times \dots \times x_i$ encodes a proper run ρ of the leader that reads from the available first writes $\beta_1, \dots, \beta_{i-1}$. Formally, $\mathrm{LValid}_{\beta_1 \dots \beta_{i-1}}(y)$. Then, leader validity of $y \times x_{i+1}$ along $\beta_1 \dots \beta_i$ mainly depends on the newly added witness x_{i+1}. The reason is that we prolong ρ, a run of the leader that was already verified. All that we have to remember from ρ is where

it ends. This means that we can shrink y to a short witness. We consecutively cut out loops from the leader, denoted by Shrink^*, until we obtain a loop free witness. Formally, if $\mathrm{LValid}_{\beta_1\dots\beta_{i-1}}(y)$ holds true, we have the equality

$$\mathrm{LValid}_{\beta_1\dots\beta_i}(y \times x_{i+1}) = \mathrm{LValid}_{\beta_1\dots\beta_i}(\mathit{Shrink}^*(y) \times x_{i+1}).$$

Hence, checking leader validity can be restricted to (concatenations of) short witnesses.

Like leader validity, we can restrict contributor validity to short witnesses. The main reason is that testing validity for the i-th first write only requires limited knowledge about earlier first writes. As long as we guarantee that earlier first writes can be provided along a run of the leader, we do not have to keep track of their precise positions anymore. This means that we can shrink the run when testing validity for the i-th first write.

Assume that $y = x_1 \times \cdots \times x_i$ is known to be contributor valid. Formally, $\mathrm{CValid}_{\beta_1\dots\beta_{i-1}}(y)$ is true. Note that the first writes considered in y are $\beta_1, \dots, \beta_{i-1}$. We want to check contributor validity of $y \times x_{i+1}$. Since there is only one new first write that we add, namely β_i, we have to evaluate $\mathrm{CValid}^i_{\beta_1\dots\beta_i}(y \times x_{i+1})$. Satisfying contributor validity means that β_i can be provided along $y \times x_{i+1}$ assuming that $\beta_1, \dots, \beta_{i-1}$ were already provided. In fact, it is not important where these earlier first writes appeared exactly. We just need the fact that after y, they can assumed to be there. This allows for shrinking y and forgetting about the precise positions of the earlier first writes. Formally, if $\mathrm{CValid}_{\beta_1\dots\beta_{i-1}}(y)$, we have

$$\mathrm{CValid}^i_{\beta_1\dots\beta_i}(y \times x_{i+1}) = \mathrm{CValid}^i_{\beta_1\dots\beta_i}(\mathit{Shrink}^*(y) \times x_{i+1}).$$

In the next section, we turn the above observations into a recursive definition of validity for short witnesses. The recursion only involves short witnesses of lower order. Since the number of these is bounded by $(\mathtt{L}+\mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$, we can employ a dynamic programming that checks validity of short witnesses in time proportional to their number.

4.2 Algorithm and Correctness

Before we can formulate the recursion, we need to introduce short witnesses and a concatenation operator on the same. A *short witness* is a witness $z = (w, q, \sigma) \in \mathit{Wit}$ where the leader states in $w = (q_1, a_1) \dots (q_n, a_n)$ are all distinct. We use Wit^{sh} to denote the set of all short witnesses. Moreover, let $\mathrm{Ord}(k)$ denote the set of those short witnesses that are of order k.

Let $x = (w, q, \sigma) \in \mathrm{Ord}(i)$ and $y = (w', q', \sigma') \in \mathrm{Ord}(j)$ be two short witnesses. Assume that the first state in w' is q, meaning that y starts with the target state of x. Then, the *short concatenation* of x and y is defined to be the short witness $x \otimes y = \mathit{Shrink}^*(x \times y) \in \mathrm{Ord}(i+j)$.

The price to pay for the smaller number of short witnesses is a more expensive check for validity. Rather than checking validity once for each short witness, we build them up by a recursion along the order, and check validity for each composition. Let z be a short witness. If $\mathit{ord}(z) = 0$, there are no first-write positions. Only leader validity is important:

$$\mathrm{Valid}^{sh}_{\varepsilon}(z) = \mathrm{LValid}_{\varepsilon}(z).$$

For a short witness z of order $k+1$, we define validity along $\beta = \beta_1 \dots \beta_{k+1} \in \mathrm{FW}$ by

$$\mathrm{Valid}^{sh}_{\beta}(z) = \bigvee_{\substack{x \in \mathrm{Ord}(k) \\ y \in \mathrm{Ord}(1)}} [z = x \otimes y] \wedge \mathrm{LValid}_{\beta}(x \times y) \wedge \mathrm{CValid}^{k+1}_{\beta}(x \times y) \wedge \mathrm{Valid}^{sh}_{\beta'}(x).$$

Here $\beta' = \beta_1 \dots \beta_k$ is the prefix of β where the last element is omitted.

The idea behind the recursion is to cut off the last first write β_{k+1}, check its validity, and recurse on the remaining part. To this end, z is decomposed into two short witnesses $x \in \mathrm{Ord}(k)$ and $y \in \mathrm{Ord}(1)$. Intuitively, x is the compression of a larger witness that is already known to be valid and y is the short witness responsible for the last first write. By our considerations above, we already know that it suffices to check validity for β_{k+1} with x instead of its expanded form. These are the evaluations $\mathrm{LValid}_\beta(x \times y)$ and $\mathrm{CValid}^{k+1}_\beta(x \times y)$. To guarantee validity along β', we recurse on $\mathrm{Valid}^{sh}_{\beta'}(x)$.

The following lemma shows the correctness of the recursion. Using Lemma 10, we can work with short witnesses to discover computations in the given leader contributor system.

▶ **Lemma 11.** *Let $q \in Q_L$ and $\beta \in \mathrm{FW}$. There is an $x = (w, q, \sigma) \in Wit$ with $\mathrm{Valid}_\beta(x)$ if and only if there is an $z = (w', q, \sigma') \in Wit^{sh}$ with $\mathrm{Valid}^{sh}_\beta(z)$. In this case, $init(x) = init(z)$.*

Note that in the lemma, $init(x)$ refers to the first state of w. Similarly for z.

It remains to give the algorithm. For each first-write sequence β and each short witness z, we compute $\mathrm{Valid}^{sh}_\beta(z)$ by a dynamic programming. To this end, we maintain a table indexed by first-write sequences and short witnesses. An entry for $\beta \in \mathrm{FW}$ and $z \in Wit^{sh}$ is computed as follows. Let $|\beta| = ord(z) = k$. We iterate over all short witnesses $x \in \mathrm{Ord}(k-1), y \in \mathrm{Ord}(1)$ and check whether $z = x \otimes y$ holds. If so, we compute $\mathrm{LValid}_\beta(x \times y) \wedge \mathrm{CValid}^k_\beta(x \times y)$ and look up the value of $\mathrm{Valid}^{sh}_{\beta'}(x)$ in the table. Details on the precise complexity are presented in the full version of the paper.

▶ **Proposition 12.** *The set of all valid short witnesses can be computed in time $(L + D)^{\mathcal{O}(L+D)}$.*

It is left to explain how interfaces can be obtained from the algorithm. From a valid short witness, target state and last memory value can be read off. Contributor states can be obtained by synchronizing the contributor along the witness. This takes polynomial time. Details can be found in the full version of the paper.

5 Finding Cycles in Polynomial Time

We give an efficient algorithm solving CYC in time $\mathcal{O}(\mathtt{D}^2 \cdot (\mathtt{C}^2 + \mathtt{L}^2 \cdot \mathtt{D}^2))$. This proves Theorem 6. The algorithm relies on a characterization of cycles in terms of stable SCC decompositions. These are decompositions of the contributor automaton into strongly connected subgraphs that are stable in the sense that they write exactly the symbols they intend to read. With a fixed point iteration, we show how to find stable SCC decompositions in the mentioned time.

Our algorithm is technically simple. It relies on a fixed point iteration calling Tarjan's algorithm [35] to obtain SCC decompositions. Hence, the algorithm is easy to implement and shows that stable SCC decompositions are the ideal structure for detecting cycles. Moreover, we can modify the algorithm to detect cycles where the leader necessarily makes a move.

We also discovered that cycles can be detected by a non-trivial polynomial-time reduction to the problem of finding cycles in dynamic graphs. Although the latter can be solved in polynomial time [30], the obtained algorithm for CYC does not admit an efficient polynomial-time complexity. The reason is that the algorithm in [30] repeatedly solves linear programs that grow large due to the reduction. Compared to this method, our algorithm is more efficient and technically simpler due to being tailored to the actual problem.

5.1 From Saturated Cycles to Stable SCC decompositions

We characterize cycles in terms of stable SCC decompositions. These are decompositions of the contributor automaton that can provide themselves with all the symbols that a cycle along this structure may read. For the definition, we generalize properties of a fixed cycle to

the fact that a saturated cycle exists. We link the latter with an alphabet Γ, a variable for the set of reads in a saturated cycle. Then we define stable SCC decompositions depending on Γ. Hence, the search for a cycle amounts to finding a Γ with a stable SCC decomposition.

Throughout the section, we fix an interface $I = (S, q, a)$ and a saturated cycle $\tau = c \to_{sat}^{+} c$ with $I(c)$. We assume that the set $\text{Writes}(\tau) = \{b \in D \mid d \xrightarrow{!b} d' \in \tau\}$ is non-empty, τ contains at least one write. If τ contains only reads, then either a contributor or the leader run in an $?a$-loop, a cycle which is easy to detect. We generalize two properties of τ.

Property 1: Strongly connectedness. Considering the saturated cycle τ, we can observe how the current state of a particular contributor P changes over time. Assume P starts in a state p and visits a state p' during τ. Since it runs along the cycle, the contributor will eventually move from p' back to p again. This means that in the contributor automaton, there is a path from p to p' and vice versa. Phrased differently, p and p' are strongly connected.

To make this notion more precise, we define a subgraph of the contributor automaton. Intuitively, it is the restriction of P_C to the states and transitions visited along τ. Rather than defining it for a single computation τ, we generalize to a set of *enabled reads* $\Gamma \subseteq D$. The directed graph $G_S(\Gamma) = (S, E(\Gamma))$ has as vertices the contributor states S and as edges the set $E(\Gamma)$. The latter are transitions of P_C between states in S that are either reads enabled by Γ or writes of arbitrary symbols. Formally, we have

$$(p, p') \in E(\Gamma) \text{ if } p \xrightarrow{?b}_C p' \text{ with } b \in \Gamma \text{ or } p \xrightarrow{!b}_C p' \text{ with } b \in D.$$

For the cycle $\tau = c \to_{sat}^{+} c$, the induced graph is $G_S(\Gamma)$ where $\Gamma = \text{Writes}(\tau)$. With the graph in place, we can define our notion of strongly connected states.

▶ **Definition 13.** *Let $p, p' \in S$ be two states and $\Gamma \subseteq D$. We say that p and p' are* strongly Γ-connected *if p and p' are strongly connected in the graph $G_S(\Gamma)$.*

Like the classical notion, the above definition generalizes to sets. We say that a set $V \subseteq S$ is *strongly Γ-connected* if each two states in V are strongly Γ-connected.

The saturated cycle τ runs along the SCC decomposition of its induced graph $G_S(\Gamma)$. Following a particular contributor P in τ, we collect the visited states in a set $S_P \subseteq S$. Then, S_P is strongly Γ-connected and thus contained in an inclusion maximal strongly connected set, an SCC of $G_S(\Gamma)$. Hence, the contributors in τ stay within SCCs of the graph. We associate with τ the SCC decomposition. Again, we generalize to a given alphabet.

Let $\Gamma \subseteq D$ and $V \subseteq S$ strongly Γ-connected. We call V a *strongly Γ-connected component* (Γ-SCC) if it is inclusion maximal. The latter means that for each $V \subseteq V'$ with V' strongly Γ-connected, we already have $V = V'$. We consider the unique partition of S into Γ-SCCs. Note that by a partition, we mean a collection $(S_1, \ldots, S_\ell)$ of pairwise disjoint subsets of S such that $S = \bigcup_{i \in [1..\ell]} S_i$. The order of a partition is not important for our purpose.

▶ **Definition 14.** *The partition of S into Γ-SCCs is called* Γ-SCC decomposition *of S.*

We denote the Γ-SCC decomposition by $SCCdcmp_S(\Gamma)$. It consists of the vertices of the SCC decomposition of $G_S(\Gamma)$. Hence, we can obtain it from an application of Tarjan's algorithm [35], a fact that becomes important when computing $SCCdcmp_S(\Gamma)$ in Section 5.2.

Property 2: Stability. Let $SCCdcmp_S(\Gamma) = (S_1, \ldots, S_\ell)$ be the Γ-SCC decomposition associated with the saturated cycle τ. The writes in τ can be linked with the S_i. If a write occurs between states $p, p' \in S_i$, we associate it with the set S_i. The writes of the leader all occur on a cyclic computation $q \to_L^* q$. The point of assigning writes to sets is the following. Writes that belong to a set can occur on a cycle through a set of the decomposition.

We generalize from τ to a given alphabet $\Gamma \subseteq D$. Let $SCCdcmp_S(\Gamma) = (S_1, \dots, S_\ell)$ be the Γ-SCC decomposition of S. The *writes of the decomposition* is the set of all symbols that occur as writes either between the states of S_i or in a cycle $q \to_L^* q$ on the leader while preserving the memory content a. Formally, we define the writes to be the union $\mathrm{Writes}(S_1, \dots, S_\ell) = \mathrm{Writes}_C(S_1, \dots, S_\ell) \cup \mathrm{Writes}_L(S_1, \dots, S_\ell)$ where

$$\mathrm{Writes}_C(S_1, \dots, S_\ell) = \{b \mid p \xrightarrow{!b}_C p' \text{ with } p, p' \in S_i\} \text{ and}$$
$$\mathrm{Writes}_L(S_1, \dots, S_\ell) = \{b \mid \exists u, v : (q, a) \xrightarrow{u.!b.v}_{L'} (q, a)\}.$$

Here, $\to_{L'}$ denotes the transition relation of the automaton $P_{L'}$, a restriction of the leader P_L to reads within $\mathrm{Writes}_C(S_1, \dots, S_\ell)$. The automaton also keeps track of the memory content. We define $P_{L'} = (Op(D), Q_L \times D, (q_L^0, a^0), \delta_{L'})$ with the transitions

$$\begin{aligned}
&(s, b) \xrightarrow{!b'}_{L'} (s', b') && \text{if } s \xrightarrow{!b'}_L s',\\
&(s, b) \xrightarrow{?b}_{L'} (s', b) && \text{if } s \xrightarrow{?b}_L s' \text{ and } b \in \mathrm{Writes}_C(S_1, \dots, S_\ell),\\
&(s, b) \xrightarrow{\varepsilon}_{L'} (s, b') && \text{if } b' \in \mathrm{Writes}_C(S_1, \dots, S_\ell).
\end{aligned}$$

The last transitions change the memory content due to a write of a contributor.

The following lemma states that writes behave monotonically. This fact will become important in Section 5.2. We provide a proof in the full version of the paper.

▶ **Lemma 15.** *Let $\Gamma \subseteq \Gamma' \subseteq D$. We have* $\mathrm{Writes}(SCCdcmp_S(\Gamma)) \subseteq \mathrm{Writes}(SCCdcmp_S(\Gamma'))$.

During the cycle τ, reads are always preceded by corresponding writes. Hence, the writes of the Γ-SCC decomposition, where $\Gamma = \mathrm{Writes}(\tau)$, provide all symbols needed for reading. In fact, we have $\mathrm{Writes}(SCCdcmp_S(\Gamma)) \supseteq \Gamma$. The following definition generalizes this property.

▶ **Definition 16.** *Let $\Gamma \subseteq D$. The Γ-SCC decomposition $SCCdcmp_S(\Gamma)$ of S is called* stable *if it provides Γ as its writes, meaning* $\mathrm{Writes}(SCCdcmp_S(\Gamma)) = \Gamma$.

Note that the definition asks for equality instead of inclusion. The reason is that we can express stability as a fixed point of a suitable operator. This will be essential in Section 5.2.

Characterization. The following proposition characterizes the existence of saturated cycles via stable SCC decompositions. It is a major step towards the polynomial-time algorithm.

▶ **Proposition 17.** *There is a saturated cycle $\tau = c \to_{sat}^+ c$ with $I(c)$ if and only if there exists a non-empty subset $\Gamma \subseteq D$ such that $SCCdcmp_S(\Gamma)$ is stable.*

Proof. Assume the existence of a saturated cycle τ. Our candidate set is $\Gamma = \mathrm{Writes}(\tau)$. We already argued above that $\mathrm{Writes}(SCCdcmp_S(\Gamma)) \supseteq \Gamma$. If equality holds, $SCCdcmp_S(\Gamma)$ is stable and Γ is the set we are looking for. Otherwise, we have $\mathrm{Writes}(SCCdcmp_S(\Gamma)) \supsetneq \Gamma$.

In the latter case, we consider $\Gamma' = \mathrm{Writes}(SCCdcmp_S(\Gamma))$ instead of Γ. Since $\Gamma' \supseteq \Gamma$, we can apply Lemma 15 and obtain that $\mathrm{Writes}(SCCdcmp_S(\Gamma'))$ contains Γ'.

Iterating this process yields a sequence of sets $(\Gamma_i)_i$ that is strictly increasing, $\Gamma_i \subsetneq \Gamma_{i+1}$, and that satisfies $\mathrm{Writes}(SCCdcmp_S(\Gamma_i)) \supseteq \Gamma_i$. The sequence is finite since $\Gamma_i \subseteq D$ for all i. Hence, there is a last set Γ_d which necessarily fulfills $\mathrm{Writes}(SCCdcmp_S(\Gamma_d)) = \Gamma_d$.

For the other direction, we need to construct a saturated cycle from a set Γ with stable SCC decomposition. Idea and formal proof are given in the full version of the paper. ◀

5.2 Computing Stable SCC decompositions

The search for a saturated cycle reduces to finding an alphabet Γ with a stable SCC decomposition. Following the definition of stability, we can express Γ as a fixed point that can be computed by a Kleene iteration [37] in polynomial time. We define the suitable operator. It acts on the powerset lattice $\mathcal{P}(D)$ and for a given set X, it computes the writes of the X-SCC decomposition. Formally, it is defined by

$$\mathrm{Writes}_{SCC}(X) = \mathrm{Writes}(\mathit{SCCdcmp}_S(X)).$$

The operator is monotone and can be evaluated in polynomial time.

▶ **Lemma 18.** *For $X \subseteq X'$ subsets of D, we have* $\mathrm{Writes}_{SCC}(X) \subseteq \mathrm{Writes}_{SCC}(X')$. *Moreover,* $\mathrm{Writes}_{SCC}(X)$ *can be computed in time* $\mathcal{O}(\mathtt{D} \cdot (\mathtt{C}^2 + \mathtt{L}^2 \cdot \mathtt{D}^2))$.

Monotonicity follows from Lemma 15. For the evaluation, let X be given. We apply Tarjan's algorithm on $G_S(X)$ to compute the X-SCC decomposition $\mathit{SCCdcmp}_S(X)$. This takes linear time. It is left to compute the writes $\mathrm{Writes}(\mathit{SCCdcmp}_S(X))$. For details on the computation and the precise complexity we refer to the full version.

The following lemma states that the non-trivial fixed points of the operator Writes_{SCC} are precisely the sets with a stable SCC decomposition. Hence, searching for a cycle reduces to searching for a fixed point.

▶ **Lemma 19.** *For $\Gamma \neq \emptyset$ we have,* $\Gamma = \mathrm{Writes}_{SCC}(\Gamma)$ *if and only if* $\mathit{SCCdcmp}_S(\Gamma)$ *is stable.*

Correctness immediately follows from the definition of stability. For finding a suitable set Γ, we employ a Kleene iteration to compute the greatest fixed point of Writes_{SCC}. It starts from $\Gamma = D$, the top element of the lattice. At each step, it evaluates $\mathrm{Writes}_{SCC}(\Gamma)$ by invoking Lemma 18. This takes time $\mathcal{O}(\mathtt{D} \cdot (\mathtt{C}^2 + \mathtt{L}^2 \cdot \mathtt{D}^2))$. Termination is after at most $\mathtt{D}$ steps since at least one element is removed from the set Γ each iteration. Hence, the time to compute the greatest fixed point of Writes_{SCC} is $\mathcal{O}(\mathtt{D}^2 \cdot (\mathtt{C}^2 + \mathtt{L}^2 \cdot \mathtt{D}^2))$.

6 Conclusion

We studied the fine-grained complexity of LCL, the liveness verification problem for leader contributor systems. To this end, we first decomposed LCL into the reachability problem LCR and the cycle detection CYC. We focused on the complexity of LCR. While an optimal $\mathcal{O}^*(2^{\mathtt{C}})$-time algorithm for $\mathsf{LCR}(\mathtt{C})$ was already known, we presented an algorithm solving $\mathsf{LCR}(\mathtt{L}, \mathtt{D})$ in time $(\mathtt{L} + \mathtt{D})^{\mathcal{O}(\mathtt{L}+\mathtt{D})}$. The algorithm is optimal in the fine-grained sense and therefore solves an open problem. It is a dynamic programming based on a notion of valid short witnesses. Moreover, we showed how to modify both algorithms for LCR so that they are compatible with a cycle detection and can be used in algorithms solving LCL.

Further, we determined the complexity of CYC. We presented an efficient fixed point iteration running in time $\mathcal{O}(\mathtt{D}^2 \cdot (\mathtt{C}^2 + \mathtt{L}^2 \cdot \mathtt{D}^2))$. It is based on a notion of stable SCC decompositions and invokes Tarjan's algorithm to find them. The result shows that LCL and LCR admit the same fine-grained complexity.

References

1. P. A. Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. In *LICS*, pages 160–170. IEEE, 1993.
2. M. F. Atig, A. Bouajjani, K. N. Kumar, and P. Saivasan. On Bounded Reachability Analysis of Shared Memory Systems. In *FSTTCS*, volume 29 of *LIPIcs*, pages 611–623. Schloss Dagstuhl, 2014.
3. N. Bertrand, P. Fournier, and A. Sangnier. Playing with Probabilities in Reconfigurable Broadcast Networks. In *FOSSACS*, volume 8412 of *LNCS*, pages 134–148. Springer, 2014.
4. R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
5. P. Chini, J. Kolberg, A. Krebs, R. Meyer, and P. Saivasan. On the Complexity of Bounded Context Switching. In *ESA*, volume 87, pages 27:1–27:15. Schloss Dagstuhl, 2017.
6. P. Chini, R. Meyer, and P.Saivasan. Liveness in Broadcast Networks. In *NETYS*, 2019.
7. P. Chini, R. Meyer, and P. Saivasan. Fine-Grained Complexity of Safety Verification. In *TACAS*, volume 10806 of *LNCS*, pages 20–37. Springer, 2018.
8. P. Chini, R. Meyer, and P. Saivasan. Fine-Grained Complexity of Safety Verification. *CoRR*, abs/1802.05559, 2018. `arXiv:1802.05559`.
9. P. Chini, R. Meyer, and P. Saivasan. Complexity of Liveness in Parameterized Systems. *CoRR*, abs/1909.12004, 2019. `arXiv:1909.12004`.
10. M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms*. Springer, 2015.
11. G. Delzanno, A. Sangnier, R. Traverso, and G. Zavattaro. On the Complexity of Parameterized Reachability in Reconfigurable Broadcast Networks. In *FSTTCS*, volume 18 of *LIPIcs*, pages 289–300. Schloss Dagstuhl, 2012.
12. G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized Verification of Ad Hoc Networks. In *CONCUR*, volume 6269 of *LNCS*, pages 313–327. Springer, 2010.
13. R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.
14. A. Durand-Gasselin, J. Esparza, P. Ganty, and R. Majumdar. Model Checking Parameterized Asynchronous Shared-Memory Systems. In *CAV*, volume 9206 of *LNCS*, pages 67–84. Springer, 2015.
15. C. Enea and A. Farzan. On Atomicity in Presence of Non-atomic Writes. In *TACAS*, volume 9636 of *LNCS*, pages 497–514. Springer, 2016.
16. J. Esparza, P. Ganty, and R. Majumdar. Parameterized Verification of Asynchronous Shared-Memory Systems. In *CAV*, pages 124–140, 2013.
17. J. Esparza, P. Ganty, and R. Majumdar. Parameterized Verification of Asynchronous Shared-Memory Systems. *JACM*, 63(1):10:1–10:48, 2016.
18. A. Farzan and P. Madhusudan. The Complexity of Predicting Atomicity Violations. In *TACAS*, volume 5505 of *LNCS*, pages 155–169. Springer, 2009.
19. H. Fernau, P. Heggernes, and Y. Villanger. A multi-parameter analysis of hard problems on deterministic finite automata. *JCSS*, 81(4):747–765, 2015.
20. H. Fernau and A. Krebs. Problems on Finite Automata and the Exponential Time Hypothesis. In *CIAA*, volume 9705 of *LNCS*, pages 89–100. Springer, 2016.
21. A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *TCS*, 256(1-2):63–92, 2001.
22. F. V. Fomin and D. Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. Springer, 2010.
23. M. Fortin, A. Muscholl, and I. Walukiewicz. Model-Checking Linear-Time Properties of Parametrized Asynchronous Shared-Memory Pushdown Systems. In *CAV*, volume 8044 of *LNCS*, pages 155–175. Springer, 2017.
24. P. Fournier. *Parameterized verification of networks of many identical processes*. PhD thesis, University of Rennes 1, 2015.

25 S. M. German and A. P. Sistla. Reasoning about Systems with Many Processes. *JACM*, 39(3):675–735, 1992.

26 M. Hague. Parameterised Pushdown Systems with Non-Atomic Writes. In *FSTTCS*, volume 13 of *LIPIcs*, pages 457–468. Schloss Dagstuhl, 2011.

27 M. Hague, R. Meyer, S. Muskalla, and M. Zimmermann. Parity to Safety in Polynomial Time for Pushdown and Collapsible Pushdown Systems. In *MFCS*, volume 117 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl, 2018.

28 R. Impagliazzo and R. Paturi. On the Complexity of k-SAT. *JCSS*, 62(2):367–375, 2001.

29 I. V. Konnov, M. Lazic, H. Veith, and J. Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, pages 719–734. ACM, 2017.

30 S. R. Kosaraju and G. F. Sullivan. Detecting Cycles in Dynamic Graphs in Polynomial Time (Preliminary Version). In *STOC*, pages 398–406. ACM, 1988.

31 S. La Torre, A. Muscholl, and I. Walukiewicz. Safety of Parametrized Asynchronous Shared-Memory Systems is Almost Always Decidable. In *CONCUR*, volume 42 of *LIPIcs*, pages 72–84. Schloss Dagstuhl, 2015.

32 O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham. Reducing liveness to safety in first-order logic. *PACMPL*, 2(POPL):26:1–26:33, 2018.

33 S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

34 A. Singh, C. R. Ramakrishnan, and S. A. Smolka. Query-Based Model Checking of Ad Hoc Network Protocols. In *CONCUR*, volume 5710 of *LNCS*, pages 603–619. Springer, 2009.

35 R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SICOMP*, 1(2):146–160, 1972.

36 T. Wareham. The Parameterized Complexity of Intersection and Composition Operations on Sets of Finite-State Automata. In *CIAA*, volume 2088 of *LNCS*, pages 302–310. Springer, 2000.

37 G. Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.