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 \( \text{NP} \)-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 \( L \) and the size of the data domain \( D \), we show an \( \mathcal{O}(L+D)^{\mathcal{O}(L+D)} \)-time algorithm. It improves on a previous algorithm, thereby settling an open problem. When parameterized by the number of states of the contributor \( C \), we reuse an \( \mathcal{O}^*(2^C) \)-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

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 \( f(k) \cdot \text{poly}(n) \). 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 \( C \), we showed that reachability can be solved in time \( O^*(2^k) \). The notation \( O^* \) suppresses polynomial factors in the running time. Interestingly, this is the best one can hope for. An algorithm with a subexponential dependence on \( C \), to be precise an algorithm running in time \( 2^{o(C)} \), 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 \( L \) and the size of the data domain \( D \). We gave an algorithm running in time \( (LD)^{O(LD)} \). Interestingly, the lower bound is only \( 2^{o((L+D) \log(L+D))} \). 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 \( L \) and \( D \). We give an algorithm running in time \( (L+D)^{O(L+D)} \). This matches the lower bound and closes the gap for reachability. When parameterized by the size of the contributors, we obtain an \( O^*(2^k) \) 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 \( O(\text{Reach}(L, D, C) \cdot \text{Cycle}(L, D, C)) \) where \( \text{Reach}(L, D, C) \) denotes the running time of the invoked reachability algorithm and \( \text{Cycle}(L, D, C) \) 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 \( L \) and \( D \) are given as parameters. It runs in time \( (L+D)^{O(L+D)} \) and significantly improves upon the \( (LD)^{O(LD)} \)-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 $\text{Cycle}(L, D, C)$ 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 $S = (D, a^0, P_L, P_C)$ 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, $la$ denotes a write of $a$ to the memory, $?a$ denotes a read of $a$. The leader is given by the tuple $P_L = (Op(D), Q_L, q_0^L, \delta_L)$ where $Q_L$ is the set of states, $q_0^L \in Q_L$ 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 $q \xrightarrow{w} q'$ for $(q, w, q') \in \delta_L$. The contributor is defined similarly, by $P_C = (Op(D), Q_C, q_0^C, \delta_C)$.

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 $(q, a, pc) \in CF = Q_L \times D \times Q_C$. 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 $CF = \bigcup_{i \in \mathbb{N}} CF^i$. A configuration is called initial if it is of the form $(q_0^L, a^0, pc^0)$ where $pc^0(i) = q_0^C$ for each $i \in [1..t]$. We use projections to access the components of a configuration. Let $\pi_L$ and $\pi_D$ 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 $\pi_C$ 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 $S$ may change due to an interaction with the memory or an internal transition. We capture such changes by a labeled transition relation among configurations, $\rightarrow \subseteq CF \times (Op(D) \cup \{\varepsilon\}) \times CF$. It contains transitions induced by the leader and by the contributor. We focus on the former. If there is a write $q \xrightarrow{a} q'$ of the leader, we get $(q, a, pc) \xrightarrow{b_a} (q', b, pc)$. Similarly, a read $q \xrightarrow{a/\varepsilon} q'$ induces $(q, a, pc) \xrightarrow{b_a} (q', a, pc)$. Note that the current memory value has to match the read symbol. An internal transition $q \xrightarrow{a} q'$ yields $(q, a, pc) \xrightarrow{b_a} (q', a, pc)$ 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 $p \xrightarrow{b_a/z_a} (q, a, pc) \xrightarrow{b_a/z_a} (q, b/a, pc')$, 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 $la$ and immediately reads $?a$ again. In this case, we could replace the corresponding read transition by $\varepsilon$.

The transition relation $\rightarrow$ is generalized to words, denoted by $c \xrightarrow{w} c'$ with $w \in Op(D)^*$. We call such a sequence a computation of $S$. We also write $c \rightarrow^* c'$ if there is a word $w$ with $c \xrightarrow{w} c'$, and $c \rightarrow^+ c'$ if $w$ has length at least 1. An infinite computation is a sequence $\sigma = c_0 \rightarrow c_1 \rightarrow \ldots$ of infinitely many transitions. We call it initialized if $c_0$ is an initial configuration. Since $\sigma$ 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 $\text{Inf}(\sigma) = \{q \in Q_L \mid \exists^\infty i : q = \pi_L(c_i)\}$.

**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 $S = (D, a^0, P_L, P_C)$ and a set of final states $F \subseteq Q_L$ encoding the specification, the problem asks whether there is an initialized infinite computation $\sigma$ such that the leader visits $F$ infinitely often along $\sigma$. Since $F$ is finite, this is equivalent to $\text{Inf}(\sigma) \cap F \neq \emptyset$. In this case, $\sigma$ is called a live computation.
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 \( \Sigma \) be an alphabet. Unlike in classical complexity theory where we consider problems over \( \Sigma^* \), 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 \( f(k) \cdot |x|^{O(1)} \). 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 \( O^*(f(k)) \).

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 \( \text{LCL}(L, D) \), the parameterization by the number of states in the leader \( L \) and the size of the data domain \( D \). We show an \( (L+D)^{O(L+D)} \)-time algorithm, matching the lower bound for \( \text{LCL} \) from [7]. The second parameterization \( \text{LCL}(C) \) is by the number of states of the contributor \( C \). We give an algorithm running in time \( O^*(2^2) \). It also matches the known lower bound [7]. Therefore, both algorithms are optimal in the fine-grained sense. The parameterizations \( \text{LCL}(L) \) and \( \text{LCL}(D) \) 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 \( \text{Reach}(L, D, C) \) denote the running time of the reachability algorithm. We show that the obtained set of interfaces will be of size at most \( \text{Reach}(L, D, C) \). 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 \( \text{Cycle}(L, D, C) \) 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 \( O(\text{Reach}(L, D, C) \cdot \text{Cycle}(L, D, C)) \).

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 \( \tau = c \rightarrow^* c \) is called saturated if for each configuration \( c' \) in \( \tau \), we have \( \pi_L(c') \subseteq \pi_L(c) \). We write \( c \rightarrow^+_\text{sat} c \) 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 \( \sigma \) with \( \text{Inf}(\sigma) \cap F \neq \emptyset \) if and only if there is a finite initialized computation \( c^0 \rightarrow^* c \rightarrow^+_\text{sat} c \) 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 \( \text{IF} \). 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 \( I \in \text{IF} \). There is a computation \( c^0 \rightarrow^* c \rightarrow^+_\text{sat} c \) with \( I(c) \) if and only if there are computations \( d^0 \rightarrow^* d \) and \( f \rightarrow^+_\text{sat} f \) with \( I(d) \land I(f) \).

In the following, we turn to our main contributions. We present algorithms for reachability and cycle detection and obtain precise values for Reach\((L, D, C)\) and Cycle\((L, D, C)\). 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.

<table>
<thead>
<tr>
<th>Leader Contributor Reachability (LCR)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A leader contributor system ( \mathcal{S} = (D, a^0, P_L, P_C) ) and final states ( F \subseteq Q_L ).</td>
</tr>
<tr>
<td><strong>Question:</strong> Is there an initialized computation ( c^0 \rightarrow^* c ) with ( \pi_L(c) \in F )?</td>
</tr>
</tbody>
</table>

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

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

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

For LCR\((C)\), 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]).** LCR\((C)\) can be solved in time \( O(2^C \cdot C^4 \cdot L^2 \cdot D^2) \).

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.

<table>
<thead>
<tr>
<th>Saturated Cycle (CYC)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A leader contributor system ( \mathcal{S} = (D, a^0, P_L, P_C) ) and an interface ( I \in \text{IF} ).</td>
</tr>
<tr>
<td><strong>Question:</strong> Is there a computation ( c \rightarrow^+_\text{sat} c ) with ( I(c) )?</td>
</tr>
</tbody>
</table>

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 $O(D^2 \cdot (C^2 + L^2 \cdot D^2))$.

The theorem shows that Cycle$(L, D, C)$ is polynomial. Hence, by Theorem 1, we obtain that LCL can be solved in time $O^*(Reach(L, D, C))$. 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$(L, D, C)$ into account, Theorem 1 yields the following.

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

Corollary 8. LCL$(C)$ can be solved in time $O(2^C \cdot L \cdot D^2 \cdot (L \cdot C^4 + D \cdot C^2 + L^2 \cdot D^3))$.

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 $2^{o((L+D) \log (L+D))}$ nor in time $2^{O(C)}$.

4 Reachability Parameterized by Leader and Domain

We present the algorithm for LCR$(L, D)$. It runs in time $(L + D)^{O(L + D)}$ and therefore proves Theorem 4. Moreover, with the results from Section 3 and 5, the algorithm can be utilized for solving LCL in time $(L + D)^{O(L + D)}$. Like in [7], the algorithm relies on a notion of 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 $(L + D)^{O(L + D)}$, 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 $w = (w, q, \sigma)$. The word $w = (q_1, a_1)(q_2, a_2)\ldots(q_n, a_n)$ represents the run of the leader. It is a sequence from $(Q_L \times (D \cup \{\perp\}))^*$, 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 $\sigma: [1..k] \rightarrow [1..n]$, a monotonically increasing map where $k \leq D$. The number of first-write positions $k$ is called the order of $w$. We denote it by $ord(w) = k$. Moreover, we use Wit for the set of all witnesses. A witness $w = (w, q, \sigma) \in Wit$ is called initialized if $w$ begins in the initial state $q_0^L$ 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 $\beta \in D^S$ that are all different. Formally, $\beta_i \neq \beta_j$ for $i \neq j$. We use $\text{FW}$ 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 $\beta$ of length $\text{ord}(x)$. For being valid, $x$ has to be leader valid along $\beta$ and contributor valid along $\beta$. 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 $\beta$ if $w$ encodes a run of the leader that reaches state $q$. Reading during the run is restricted to symbols from $\beta$: the $\ell$-th symbol $\beta_\ell$ 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 $q_i \xrightarrow{a_i} q_{i+1}$. If $a_i = \bot$, the leader either has an $\epsilon$-transition or reads a symbol available at position $i$, from the set $S_\beta(i) = \{\beta_\ell \mid \sigma(\ell) \leq i\}$. We use $L\text{Valid}_\beta(x)$ to indicate that $x$ is leader valid along $\beta$.

**Contributor Validity.** The witness is contributor valid along $\beta$ if the contributors can provide the first writes for $w$ in the order indicated by $\sigma$. Let us focus on the $i$-th first write $\beta_i$. Providing $\beta$ is a question of reachability of the set $Q_i = \{p \mid \exists p' : p \xrightarrow{\beta_\ell} C p'\}$ in the contributor automaton. More precisely, 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 $\text{Expr}(x, \beta_1 \ldots \beta_{i-1})$ be the language of available reads. We say that $x$ is valid for the $i$-th first write of $\beta$ if $Q_i$ is reachable by a contributor while reading is restricted to $\text{Expr}(x, \beta_1 \ldots \beta_{i-1})$. We use $C\text{Valid}_\beta(x)$ to indicate this validity. If $x$ is valid for all first writes, it is contributor valid along $\beta$. Formally, $C\text{Valid}_\beta(x) = \bigwedge_{i \in [1..\text{ord}(x)]} C\text{Valid}_\beta(x)$.

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

**Lemma 9.** Let $x \in \text{Wit}$ and $\beta \in \text{FW}$. $\text{Valid}_\beta(x)$ 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 $c^0 \rightarrow^* c$ with $\pi_L(c) = q$ if and only if there is an initialized $x = (w, q, \sigma) \in \text{Wit}$ and $\beta \in \text{FW}$ so that $\text{Valid}_\beta(x)$.

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 $x = x_1 \times x_2 \times \cdots \times x_{k+1}$ 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 be assumed to be a short witness. There is no need for recording loops of the leader between first writes.

Assume $y = x_1 \times \cdots \times x_i$ encodes a proper run $\rho$ of the leader that reads from the available first writes $\beta_1, \ldots, \beta_{i-1}$. Formally, $L\text{Valid}_{\beta_1 \ldots \beta_{i-1}}(y)$. Then, leader validity of $y \times x_{i+1}$ along $\beta_1 \ldots \beta_i$ mainly depends on the newly added witness $x_{i+1}$. The reason is that we prolong $\rho$, a run of the leader that was already verified. All that we have to remember from $\rho$ 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 $\text{Shrink}^k$, until we obtain a loop free witness. Formally, if $\text{LValid}_{\beta_1 \cdots \beta_{i-1}}(y)$ holds true, we have the equality

$$\text{LValid}_{\beta_1 \cdots \beta_i}(y \times x_{i+1}) = \text{LValid}_{\beta_1 \cdots \beta_i}(\text{Shrink}^k(y) \times x_{i+1}).$$

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, $\text{CValid}_{\beta_1 \cdots \beta_{i-1}}(y)$ is true. Note that the first writes considered in $y$ are $\beta_1, \ldots, \beta_{i-1}$. We want to check contributor validity of $y \times x_{i+1}$. Since there is only one new first write that we add, namely $\beta_i$, we have to evaluate $\text{CValid}_{\beta_1 \cdots \beta_i}(y \times x_{i+1})$. Satisfying contributor validity means that $\beta_i$ can be provided along $y \times x_{i+1}$ assuming that $\beta_1, \ldots, \beta_{i-1}$ 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 $\text{CValid}_{\beta_1 \cdots \beta_{i-1}}(y)$, we have

$$\text{CValid}_{\beta_1 \cdots \beta_i}(y \times x_{i+1}) = \text{CValid}_{\beta_1 \cdots \beta_i}(\text{Shrink}^k(y) \times x_{i+1}).$$

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 $(L + D)^{O(L+D)}$, 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 $z = (w, q, \sigma) \in \text{Wit}$ where the leader states in $w = (q_1, a_1) \cdots (q_n, a_n)$ are all distinct. We use $\text{Wit}^\text{sh}$ to denote the set of all short witnesses. Moreover, let $\text{Ord}(k)$ denote the set of those short witnesses that are of order $k$.

Let $x = (w, q, \sigma) \in \text{Ord}(i)$ and $y = (w', q', \sigma') \in \text{Ord}(j)$ 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 $x \otimes y = \text{Shrink}^k(x \times y) \in \text{Ord}(i+j)$.

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 $\text{ord}(z) = 0$, there are no first-write positions. Only leader validity is important:

$$\text{Valid}^\text{sh}_z(z) = \text{LValid}_z(z).$$

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

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

Here $\beta' = \beta_1 \cdots \beta_k$ is the prefix of $\beta$ where the last element is omitted.
The idea behind the recursion is to cut off the last first write \( \beta_k \), check its validity, and recurse on the remaining part. To this end, \( z \) is decomposed into two short witnesses \( x \in \text{Ord}(k) \) and \( y \in \text{Ord}(1) \). 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 \( \beta_k \) with \( x \) instead of its expanded form. These are the evaluations \( \text{LValid}_{\beta}(x \times y) \) and \( \text{CValid}_{\beta}^{k+1}(x \times y) \).

To guarantee validity along this structure, we can modify the algorithm to detect cycles where the leader necessarily makes a move. These are decompositions of the contributor automaton into strongly connected subgraphs 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. The algorithm relies on a characterization of cycles in terms of stable SCC decompositions. Hence, the algorithm is easy to implement and shows how to find stable SCC decompositions in the mentioned time.

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

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

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

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.

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

We give an efficient algorithm solving CYC in time \( O(D^2 \cdot (C^2 + L^2 \cdot D^2)) \). 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 $\Gamma$, a variable for the set of reads in a saturated cycle. Then we define stable SCC decompositions depending on $\Gamma$. Hence, the search for a cycle amounts to finding a $\Gamma$ with a stable SCC decomposition.

Throughout the section, we fix an interface $I = (S, q, a)$ and a saturated cycle $\tau = c \rightarrow_{\text{sat}}^+ c$ with $I(c)$. We assume that the set $\text{Writes}(\tau) = \{ b \in D \mid d \xrightarrow{\text{sat}} d' \} \in \Gamma$ is non-empty, $\tau$ contains at least one write. If $\tau$ 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 $\tau$.

**Property 1: Strongly connectedness.** Considering the saturated cycle $\tau$, 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 $\tau$. 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 $\tau$. Rather than defining it for a single computation $\tau$, 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 $\Gamma$ 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 $\tau = c \rightarrow_{\text{sat}}^+ c$, 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 $\Gamma$-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 $\Gamma$-connected if each two states in $V$ are strongly $\Gamma$-connected.

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

Let $\Gamma \subseteq D$ and $V \subseteq S$ strongly $\Gamma$-connected. We call $V$ a strongly $\Gamma$-connected component ($\Gamma$-SCC) if it is inclusion maximal. The latter means that for each $V \subseteq V'$ with $V'$ strongly $\Gamma$-connected, we already have $V = V'$. We consider the unique partition of $S$ into $\Gamma$-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 $\Gamma$-SCCs is called $\Gamma$-SCC decomposition of $S$.

We denote the $\Gamma$-SCC decomposition by $\text{SCCDcmp}_G(\Gamma)$. 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 $\text{SCCDcmp}_G(\Gamma)$ in Section 5.2.

**Property 2: Stability.** Let $\text{SCCDcmp}_G(\Gamma) = (S_1, \ldots, S_\ell)$ be the $\Gamma$-SCC decomposition associated with the saturated cycle $\tau$. The writes in $\tau$ 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 $q \rightarrow_L^* q$. 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 $\tau$ to a given alphabet $\Gamma \subseteq D$. Let $SCCdcmp_S(\Gamma) = (S_1, \ldots, S_t)$ be the $\Gamma$-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 $q \rightarrow^*_L q$ on the leader while preserving the memory content $a$. Formally, we define the writes to be the union

$$W_{SCCdcmp}(S_1, \ldots, S_t) = W_C(S_1, \ldots, S_t) \cup W_L(S_1, \ldots, S_t)$$

Here, $\rightarrow^*_L$ denotes the transition relation of the automaton $P_L$, a restriction of the leader $P_L$ to reads within $W_{SCCdcmp}(S_1, \ldots, S_t)$. The automaton also keeps track of the memory content. We define $P_L = (Op(D), Q_L \times D, (q_0^L, a^0), \delta_L)$ with the transitions

$$(s, b) \xrightarrow{\mu} L, (s', b') \quad \text{if} \quad s \xrightarrow{\mu} L s',$$

$$(s, b) \xrightarrow{\mu} L, (s', b) \quad \text{if} \quad s \xrightarrow{\mu} L s' \text{ and } b \in W_C(S_1, \ldots, S_t),$$

$$(s, b) \xrightarrow{\mu} L, (s, b') \quad \text{if} \quad b' \in W_C(S_1, \ldots, S_t).$$

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 $W_{SCCdcmp}(\Gamma') \subseteq W_{SCCdcmp}(\Gamma)$.

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

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

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 $\tau = c \rightarrow^*_n c$ 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 $\tau$. Our candidate set is $\Gamma = W_{\tau}(\Gamma)$. We already argued above that $W_{SCCdcmp}(\Gamma) \supseteq \Gamma$. If equality holds, $SCCdcmp_S(\Gamma)$ is stable and $\Gamma$ is the set we are looking for. Otherwise, we have $W_{SCCdcmp}(\Gamma) \supseteq \Gamma$. In the latter case, we consider $\Gamma' = W_{SCCdcmp}(\Gamma)$ instead of $\Gamma$. Since $\Gamma' \supseteq \Gamma$, we can apply Lemma 15 and obtain that $W_{SCCdcmp}(\Gamma')$ contains $\Gamma'$.

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

For the other direction, we need to construct a saturated cycle from a set $\Gamma$ 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 \( \Gamma \) with a stable SCC decomposition. Following the definition of stability, we can express \( \Gamma \) 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

\[
\text{Writes}_{SCC}(X) = \text{Writes}(\text{SCC}dcmp_S(X)).
\]

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

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

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 \( \text{SCC}dcmp_S(X) \). This takes linear time. It is left to compute the writes \( \text{Writes}(\text{SCC}dcmp_S(X)) \). 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 \( \text{Writes}_{SCC} \) 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, \( \Gamma = \text{Writes}_{SCC}(\Gamma) \) if and only if \( \text{SCC}dcmp_S(\Gamma) \) is stable.

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

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 \( O^*(2^d) \)-time algorithm for LCR(\( C \)) was already known, we presented an algorithm solving LCR(\( L, D \)) in time \( (L + D)^{O(L+D)} \). 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 \( O(D^2 \cdot (C^2 + L^2 \cdot D^2)) \). 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