Journal of Automated Reasoning (2020) 64:1419–1444
https://doi.org/10.1007/s10817-020-09572-x

Fine-Grained Complexity of Safety Verification

Peter Chini[1] · **Roland Meyer**[1] · **Prakash Saivasan**[1]

Received: 19 June 2020 / Accepted: 26 June 2020 / Published online: 14 July 2020

Abstract
We study the fine-grained complexity of Leader Contributor Reachability (LCR) and Bounded-Stage Reachability (BSR), two variants of the safety verification problem for shared memory concurrent programs. For both problems, the memory is a single variable over a finite data domain. Our contributions are new verification algorithms and lower bounds. The latter are based on the Exponential Time Hypothesis (ETH), the problem Set Cover, and cross-compositions. LCR is the question whether a designated leader thread can reach an unsafe state when interacting with a certain number of equal contributor threads. We suggest two parameterizations: (1) By the size of the data domain $\mathtt{D}$ and the size of the leader $\mathtt{L}$, and (2) by the size of the contributors $\mathtt{C}$. We present algorithms for both cases. The key techniques are compact witnesses and dynamic programming. The algorithms run in $\mathcal{O}^*((\mathtt{L} \cdot (\mathtt{D}+1))^{\mathtt{L}\cdot\mathtt{D}} \cdot \mathtt{D}^{\mathtt{D}})$ and $\mathcal{O}^*(2^{\mathtt{C}})$ time, showing that both parameterizations are fixed-parameter tractable. We complement the upper bounds by (matching) lower bounds based on ETH and Set Cover. Moreover, we prove the absence of polynomial kernels. For BSR, we consider programs involving $\mathtt{t}$ different threads. We restrict the analysis to computations where the write permission changes $\mathtt{s}$ times between the threads. BSR asks whether a given configuration is reachable via such an $\mathtt{s}$-stage computation. When parameterized by $\mathtt{P}$, the maximum size of a thread, and $\mathtt{t}$, the interesting observation is that the problem has a large number of difficult instances. Formally, we show that there is no polynomial kernel, no compression algorithm that reduces the size of the data domain $\mathtt{D}$ or the number of stages $\mathtt{s}$ to a polynomial dependence on $\mathtt{P}$ and $\mathtt{t}$. This indicates that symbolic methods may be harder to find for this problem.

Keywords Parameterized verification · Parameterized complexity · Fine-grained complexity · Safety verification

✉ Peter Chini
p.chini@tu-bs.de

Roland Meyer
roland.meyer@tu-bs.de

Prakash Saivasan
p.saivasan@tu-bs.de

[1] TU Braunschweig, Brunswick, Germany

1 Introduction

We study the fine-grained complexity of two safety verification problems [1,18,32] for shared memory concurrent programs. The motivation to reconsider these problems are recent developments in fine-grained complexity theory [7,12,35,39]. They suggest that classifications such as NP or even FPT are too coarse to explain the success of verification methods. Instead, it should be possible to identify the precise influence that parameters of the input have on the verification time. Our contribution confirms this idea. We give new verification algorithms for the two problems that, for the first time, can be proven optimal in the sense of fine-grained complexity theory. To state the results, we need some background. As we proceed, we explain the development of fine-grained complexity theory.

There is a well-known gap between the success that verification tools see in practice and the judgments about computational hardness that worst-case complexity is able to give. The applicability of verification tools steadily increases by tuning them towards industrial instances. The complexity estimation is stuck with considering the input size or at best assuming certain parameters to be constant. However, the latter approach is not very enlightening if the runtime is n^k, where n is the input size and k the parameter.

The observation of a gap between practical algorithms and complexity theory is not unique to verification but made in every field that has to solve hard computational problems. Complexity theory has taken up the challenge to close the gap. So-called *fixed-parameter tractability* (FPT) [13,15] proposes to identify parameters k so that the runtime is $f(k)poly(n)$, where f is a computable function and $poly(n)$ denotes any polynomial dependent on n. These parameters are powerful in the sense that they dominate the complexity.

For an FPT-result to be useful, function f should only be mildly exponential, and of course k should be small in the instances of interest. Intuitively, they are what one needs to optimize. *Fine-grained complexity* is the study of upper and lower bounds on the function. Indeed, the fine-grained complexity of a problem is written as $O^*(f(k))$, emphasizing f and k and suppressing the polynomial part. For upper bounds, the approach is still to come up with an algorithm.

For lower bounds, fine-grained complexity has taken a new and very pragmatic perspective. For the problem of n-variable 3-SAT the best known algorithm runs in $\mathcal{O}(2^n)$, and this bound has not been improved since 1970. The idea is to take improvements on this problem as unlikely, known as the exponential-time hypothesis (ETH) [35]. Formally, it asserts that there is no $2^{o(n)}$-time algorithm for 3-SAT. ETH serves as a lower bound that is reduced to other problems [39]. An even stronger assumption about SAT, called SETH [7,35], and a similar one about Set Cover[12] allow for lower bounds like the absence of $\mathcal{O}^*((2-\delta)^n)$ algorithms.

In this work, we contribute fine-grained complexity results for verification problems on concurrent programs. The first problem is reachability for a leader thread that is interacting with an unbounded number of contributors (LCR) [18,32]. We show that, assuming a parameterization by the size of the leader $\mathtt{L}$ and the size of the data domain $\mathtt{D}$, the problem can be solved in $\mathcal{O}^*((\mathtt{L}\cdot(\mathtt{D}+1))^{\mathtt{L}\cdot\mathtt{D}}\cdot\mathtt{D}^{\mathtt{D}})$. At the heart of the algorithm is a compression of computations into witnesses. To check reachability, our algorithm then iterates over candidates for witnesses and checks each of them for being a proper witness. Interestingly, we can formulate a variant of the algorithm that seems to be suited for large state spaces.

Using ETH, we show that the algorithm is (almost) optimal. Moreover, the problem is shown to have a large number of hard instances. Technically, there is no polynomial kernel [5,6]. Experience with kernel lower bounds is still limited. This notion of hardness seems to

indicate that symbolic methods are hard to apply. The lower bounds that we present share similarities with the reductions from [8,29,30].

If we consider the size $\mathtt{C}$ of the contributors as a parameter, we obtain an $\mathcal{O}^*(2^{\mathtt{C}})$ upper bound. Our algorithm is based on dynamic programming. We use the technique to solve a reachability problem on a graph that is shown to be a compressed representation for LCR. The compression is based on a saturation argument which is inspired by thread-modular reasoning [23,24,31,34]. With the hardness assumption on Set Cover we show that the algorithm is indeed optimal. Moreover, we prove the absence of a polynomial kernel.

Parameterizations of LCR involving just a single parameter $\mathtt{D}$ or $\mathtt{L}$ are intractable. We show that these problems are W[1]-hard. This proves the existence of an FPT-algorithm for those parameterizations unlikely.

The second problem we study generalizes bounded context switching. Bounded stage reachability (BSR) asks whether a state is reachable if there is a bound $\mathtt{s}$ on the number of times the write permission is allowed to change between the threads [1]. Again, we show the new form of kernel lower bound. The result is tricky and highlights the power of the computation model.

The results are summarized by the table below. Main findings are highlighted in gray. We present two new algorithms for LCR. Moreover, we suggest kernel lower bounds as hardness indicators for verification problems. The corresponding lower bound for BSR is particularly difficult to achieve.

Problem	Upper Bound	Lower Bound	Kernel
LCR($\mathtt{D},\mathtt{L}$)	$\mathcal{O}^*((\mathtt{L}\cdot(\mathtt{D}+1))^{\mathtt{L}\cdot\mathtt{D}}\cdot\mathtt{D}^{\mathtt{D}})$	$2^{o(\sqrt{\mathtt{L}\cdot\mathtt{D}}\cdot\log(\mathtt{L}\cdot\mathtt{D}))}$	No poly.
LCR($\mathtt{C}$)	$\mathcal{O}^*(2^{\mathtt{C}})$	$(2-\delta)^{\mathtt{C}}$	No poly.
LCR($\mathtt{D}$), LCR($\mathtt{L}$)	Intractable		
BSR($\mathtt{P},\mathtt{t}$)	$\mathcal{O}^*(\mathtt{P}^{2\mathtt{t}})$	$2^{o(\mathtt{t}\cdot\log(\mathtt{P}))}$	No poly.
BSR($\mathtt{s},\mathtt{D}$)	Intractable		

The conference version of this paper appeared in [10]. The paper at hand presents new results. This includes an improved algorithm for LCR running in $\mathcal{O}^*(2^{\mathtt{C}})$ time instead of $\mathcal{O}^*(4^{\mathtt{C}})$ and a new $(2-\delta)^{\mathtt{C}}$ lower bound based on Set Cover. Together, upper and lower bound show that the optimal algorithm for the problem has been found. Moreover, we give proofs for the intractability of certain parameterizations of LCR and BSR. This justifies our choice of parameters.

We provide a full version of the paper at hand in [11], including missing proofs and details of formal constructions.

Related Work

Concurrent programs communicating through a shared memory and having a fixed number of threads have been extensively studied [2,16,27,33]. The leader contributor reachability problem as considered in this paper was introduced as parametrized reachability in [32]. In [18], it was shown to be NP-complete when only finite state programs are involved and PSPACE-complete for recursive programs. In [36], the parameterized pairwise reachability problem was considered and shown to be decidable. Parameterized reachability under a variant of round robin scheduling was proven decidable in [38].

The bounded stage restriction on the computations of concurrent programs as considered here was introduced in [1]. The corresponding reachability problem was shown to be NP-complete when only finite state programs are involved. The problem remains in NEXP-time and PSPACE-hard for a combination of counters and a single pushdown. The bounded stage restriction generalizes the concept of bounded context switching from [40], which was shown to be NP-complete in that paper. In [9], FPT-algorithms for bounded context switching were obtained under various parameterization. In [3], networks of pushdowns communicating through a shared memory were analyzed under topological restrictions.

There have been few efforts to obtain fixed-parameter tractable algorithms for automata and verification-related problems. FPT-algorithms for automata problems have been studied in [21,22,41]. In [14], model checking problems for synchronized executions on parallel components were considered and proven intractable. In [17], the notion of conflict serializability was introduced for the TSO memory model and an FPT-algorithm for checking serializability was provided. The complexity of predicting atomicity violation on concurrent systems was considered in [20]. The finding is that FPT-solutions are unlikely to exist. In [19], the problem of checking correctness of a program along a pattern is investigated. The authors conduct an analysis in several parameters. The results range from NP-hardness even for fixed parameters to FPT-algorithms.

2 Preliminaries

We introduce our model for programs, which is fairly standard and taken from [1,18,32], and give the basics on fixed-parameter tractability.

Programs

A program consists of finitely many threads that access a shared memory. The memory is modeled to hold a single value at a time. Formally, a *(shared memory) program* is a tuple $\mathcal{A} = (D, a^0, (P_i)_{i\in[1..t]})$. Here, D is the data domain of the memory and $a^0 \in D$ is the initial value. Threads are modeled as control-flow graphs that write values to or read values from the memory. These operations are captured by $Op(D) = \{!a, ?a \mid a \in D\}$. We use the notation $W(D) = \{!a \mid a \in D\}$ for the write operations and $R(D) = \{?a \mid a \in D\}$ for the read operations. A thread P_{id} is a non-deterministic finite automaton $(Op(D), Q, q^0, \delta)$ over the alphabet of operations. The set of states is Q with $q^0 \in Q$ the initial state. The final states will depend on the verification task. The transition relation is $\delta \subseteq Q \times (Op(D) \cup \{\varepsilon\}) \times Q$. We extend it to words and also write $q \xrightarrow{w} q'$ for $q' \in \delta(q, w)$. Whenever we need to distinguish between different threads, we add indices and write Q_{id} or δ_{id}.

The semantics of a program is given in terms of labeled transitions between configurations. A *configuration* is a pair $(pc, a) \in (Q_1 \times \cdots \times Q_t) \times D$. The program counter pc is a vector that shows the current state $pc(i) \in Q_i$ of each thread P_i. Moreover, the configuration gives the current value in memory. We call $c^0 = (pc^0, a^0)$ with $pc^0(i) = q_i^0$ for all $i \in [1..t]$ the *initial configuration*. Let C denote the set of all configurations. The program's transition relation among configurations $\to \subseteq C \times (Op(D) \cup \{\varepsilon\}) \times C$ is obtained by lifting the transition relations of the threads. To define it, let $pc_1 = pc[i = q_i]$, meaning thread P_i is in state q_i and otherwise the program counter coincides with pc. Let $pc_2 = pc[i = q_i']$. If thread P_i tries to read with the transition $q_i \xrightarrow{?a} q_i'$, then $(pc_1, a) \xrightarrow{?a} (pc_2, a)$. Note that the

memory is required to hold the desired value. If the thread has the transition $q_i \xrightarrow{!b} q_i'$, then $(pc_1, a) \xrightarrow{!b} (pc_2, b)$. Finally, $q_i \xrightarrow{\varepsilon} q_i'$ yields $(pc_1, a) \xrightarrow{\varepsilon} (pc_2, a)$. The program's transition relation is generalized to words, $c \xrightarrow{w} c'$. We call such a sequence of consecutive labeled transitions a *computation*. To indicate that there is a word justifying a computation from c to c', we write $c \to^* c'$. We may use an index $\xrightarrow{w}_i$ to indicate that the computation was induced by thread P_i. Where appropriate, we also use the program as an index, $\xrightarrow{w}_{\mathcal{A}}$.

Fixed-Parameter Tractability

We wish to study the fine-grained complexity of safety verification problems for the above programs. This means our goal is to identify parameters of these problems that satisfy two properties. First, in practical instances they are small. Second, assuming that these parameters are small, show that efficient verification algorithms can be obtained. *Parametrized complexity* is a branch of complexity theory that makes precise the idea of being efficient relative to a parameter.

Fix a finite alphabet Σ. A *parameterized problem* L is a subset of $\Sigma^* \times \mathbb{N}$. The problem is *fixed-parameter tractable* if there is a deterministic algorithm that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, decides $(x, k) \in L$ in time $f(k) \cdot |x|^{O(1)}$. We use FPT for the class of all such problems and say *a problem is* FPT to mean it is in that class. Note that f is a computable function only depending on the parameter k. It is common to denote the runtime by $\mathcal{O}^*(f(k))$ and suppress the polynomial part. We will be interested in the precise dependence on the parameter, in upper and lower bounds on the function f. This study is often referred to as *fine-grained complexity*.

Lower bounds on f are usually obtained from assumptions about SAT. The most famous is the *Exponential Time Hypothesis* (ETH). It assumes that there is no algorithm solving n-variable 3-SAT in $2^{o(n)}$ time. Then, the reasoning is as follows: If f drops below a certain bound, ETH would fail. Other standard assumptions for lower bounds are the *Strong Exponential Time Hypothesis* (SETH) and the hardness assumption of Set Cover. We postpone the definition of the latter and focus on SETH. This assumption is more restrictive than ETH. It asserts that n-variable SAT cannot be solved in $\mathcal{O}^*((2-\delta)^n)$ time for any $\delta > 0$.

While many parameterizations of NP-hard problems were proven to be fixed-parameter tractable, there are problems that are unlikely to be FPT. Such problems are hard for the complexity class W[1]. For a theory of relative hardness, the appropriate notion of reduction is called *parameterized reduction*. Given parameterized problems $L, L' \subseteq \Sigma^* \times \mathbb{N}$, we say that L is *reducible* to L' via a *parameterized reduction* if there is an algorithm that transforms an input (x, k) to an input (x', k') in time $g(k) \cdot |x|^{O(1)}$ such that $(x, k) \in L$ if and only if $(x', k') \in L'$. Here, g is a computable function and k' is computed by a function only dependent on k.

3 Leader Contributor Reachability

We consider the *leader contributor reachability problem* for shared memory programs. The problem was introduced in [32] and shown to be NP-complete in [18] for the finite state case.[1] We contribute two new verification algorithms that target two parameterizations of the

[1] The problem is called parameterized reachability in these works. We renamed it to avoid confusion with parameterized complexity.

problem. In both cases, our algorithms establish fixed-parameter tractability. Moreover, with matching lower bounds we prove them to be optimal even in the fine-grained sense.

An instance of the leader contributor reachability problem is given by a shared memory program of the form $\mathcal{A} = (D, a^0, (P_L, (P_i)_{i\in[1..t]}))$. The program has a designated *leader* thread P_L and several *contributor* threads $P_1, \ldots, P_t$. In addition, we are given a set of unsafe states for the leader. The task is to check whether the leader can reach an unsafe state when interacting with a number of instances of the contributors. It is worth noting that the problem can be reduced to having a single contributor. Let the corresponding thread P_C be the union of $P_1, \ldots, P_t$ (constructed using an initial ε-transition). We base our complexity analysis on this simplified formulation of the problem.

For the definition, let $\mathcal{A} = (D, a^0, (P_L, P_C))$ be a program with two threads. Let $F_L \subseteq Q_L$ be a set of unsafe states of the leader. For any $t \in \mathbb{N}$, define the program $\mathcal{A}^t = (D, a^0, (P_L, (P_C)_{i\in[1..t]}))$ to have exactly t copies of P_C. Further, let C^f be the set of configurations where the leader is in an unsafe state (from F_L). The problem of interest is as follows:

Leader Contributor Reachability (LCR)
Input: A program $\mathcal{A} = (D, a^0, (P_L, P_C))$ and a set of states $F_L \subseteq Q_L$.
Question: Is there a $t \in \mathbb{N}$ such that $c^0 \to^*_{\mathcal{A}^t} c$ for some $c \in C^f$?

We consider two parameterizations of LCR. First, we parameterize by $\mathtt{D}$, the size of the data domain D, and $\mathtt{L}$, the number of states of the leader P_L. We denote the parameterization by LCR($\mathtt{D}$, $\mathtt{L}$). The second parameterization that we consider is LCR($\mathtt{C}$), a parameterization by the number of states of the contributor P_C. For both, LCR($\mathtt{D}$, $\mathtt{L}$) and LCR($\mathtt{C}$), we present fine-grained analyses that include FPT-algorithms as well as lower bounds for runtimes and kernels.

While for LCR($\mathtt{D}$, $\mathtt{L}$) we obtain an FPT-algorithm, it is not likely that LCR($\mathtt{D}$) and LCR($\mathtt{L}$) admit the same. We prove that these parameterizations are W[1]-hard.

3.1 Parameterization by Memory and Leader

We give an algorithm that solves LCR in time $\mathcal{O}^*((\mathtt{L}\cdot(\mathtt{D}+1))^{\mathtt{L}\cdot\mathtt{D}}\cdot\mathtt{D}^{\mathtt{D}})$, which means LCR($\mathtt{D}$, $\mathtt{L}$) is FPT. We then show how to modify the algorithm to solve instances of LCR as they are likely to occur in practice. Interestingly, the modified version of the algorithm lends itself to an efficient implementation based on off-the-shelf sequential model checkers. We conclude with lower bounds for LCR($\mathtt{D}$, $\mathtt{L}$).

Upper Bound

We give an algorithm for the parameterization LCR($\mathtt{D}$, $\mathtt{L}$). The key idea is to compactly represent computations that may be present in an instance of the given program. To this end, we introduce a domain of so-called witness candidates. The main technical result, Lemma 6, links computations and witness candidates. It shows that reachability of an unsafe state holds in an instance of the program if and only if there is a witness candidate that is valid (in a precise sense). With this, our algorithm iterates over all witness candidates and checks each of them for being valid. To state the overall result, let $\mathit{Wit}(\mathtt{L}, \mathtt{D}) = (\mathtt{L}\cdot(\mathtt{D}+1))^{\mathtt{L}\cdot\mathtt{D}}\cdot\mathtt{D}^{\mathtt{D}}\cdot\mathtt{L}$ be the number of witness candidates and let $\mathit{Valid}(\mathtt{L}, \mathtt{D}, \mathtt{C}) = \mathtt{L}^3\cdot\mathtt{D}^2\cdot\mathtt{C}^2$ be the time it takes to check validity of a candidate. Note that it is polynomial.

Theorem 1 LCR *can be solved in time* $\mathcal{O}(Wit(\mathtt{L}, \mathtt{D}) \cdot Valid(\mathtt{L}, \mathtt{D}, \mathtt{C}))$.

Let $\mathcal{A} = (D, a^0, (P_L, P_C))$ be the program of interest and F_L be the set of unsafe states in the leader. Assume we are given a computation ρ showing that P_L can reach a state in F_L when interacting with a number of contributors. We explain the main ideas to find an efficient representation for ρ that still allows for the reconstruction of a similar computation. To simplify the presentation, we assume the leader never writes $!a$ and immediately reads $?a$ (same value). If this is the case, the read can be replaced by ε.

In a first step, we delete most of the moves in ρ that were carried out by the contributors. We only keep *first writes*. For each value a, this is the write transitions $fw(a) = c \xrightarrow{!a} c'$ where a is written by a contributor for the first time. The reason we can omit subsequent writes of a is the following: If $fw(a)$ is carried out by contributor P_1, we can assume that there is an arbitrary number of other contributors that all mimicked the behavior of P_1. This means whenever P_1 did a transition, they copycatted it right away. Hence, there are arbitrarily many contributors pending to write a. Phrased differently, the symbol a is available for the leader whenever P_L needs to read it. The idea goes back to the *Copycat Lemma* stated in [18]. The reads of the contributors are omitted as well. We will make sure they can be served by the first writes and the moves done by P_L.

After the deletion, we are left with a shorter expression ρ'. We turn it into a word w over the alphabet $Q_L \cup D_\perp \cup \bar{D}$ with $D_\perp = D \cup \{\perp\}$ and $\bar{D} = \{\bar{a} \mid a \in D\}$. Each transition $c \xrightarrow{!a/?a/\varepsilon}_L c'$ in ρ' that is due to the leader moving from q to q' is mapped (i) to $q.a.q'$ if it is a write and (ii) to $q.\perp.q'$ otherwise. A first write $fw(a) = c \xrightarrow{a} c'$ of a contributor is mapped to $\bar{a}$. We may assume that the resulting word w is of the form $w = w_1.w_2$ with $w_1 \in ((Q_L.D_\perp)^*.\bar{D})^*$ and $w_2 \in (Q_L.D_\perp)^*.F_L$. Note that w can still be of unbounded length.

In order to find a witness of bounded length, we compress w_1 and w_2 to w_1' and w_2'. Between two first writes $\bar{a}$ and $\bar{b}$ in w_1, the leader can perform an unbounded number of transitions, represented by a word in $(Q_L.D_\perp)^*$. Hence, there are states $q \in Q_L$ repeating between $\bar{a}$ and $\bar{b}$. We contract the word between the first and the last occurrence of q into just a single state q. This state now represents a loop on P_L. Since there are $\mathtt{L}$ states in the leader, this bounds the number of contractions. Furthermore, we know that the number of first writes is bounded by $\mathtt{D}$, each symbol can be written for the first time at most once. Thus, the compressed string w_1' is a word in the language $((Q_L.D_\perp)^{\leq \mathtt{L}}.\bar{D})^{\leq \mathtt{D}}$.

The word w_2 is of the form $w_2 = q.u$ for a state $q \in Q_L$ and a word u. We truncate the word u and only keep the state q. Then we know that there is a computation leading from q to a state in F_L where P_L can potentially write any symbol but read only those symbols which occurred as a first write in w_1'. Altogether, we are left with a word of bounded length.

Definition 2 The set of witness candidates is $\mathcal{E} = ((Q_L.D_\perp)^{\leq \mathtt{L}}.\bar{D})^{\leq \mathtt{D}}.Q_L$.

Before we elaborate on the precise relation between witness candidates and computations, we turn to an example. It shows how an actual computation is compressed to a witness candidate following the above steps.

Example 3 Consider the program $\mathcal{A} = (D, a^0, (P_L, P_C))$ with domain D, leader thread P_L, and contributor thread P_C given in Fig. 1. We follow a computation in $\mathcal{A}^2$ that reaches the unsafe state q_4 of the leader. Note that the transitions are labeled by L and C, depending on whether the leader or a contributor moved.

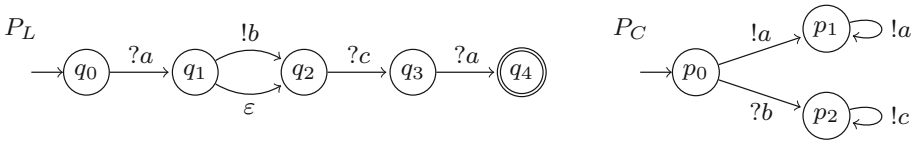

Fig. 1 Leader thread P_L (left) and contributor thread P_C (right) over the data domain $D = \{a^0, a, b, c\}$. The only unsafe state of the leader is given by $F_L = \{q_4\}$

$$(q_0, p_0, p_0, a^0) \xrightarrow{!a}_C (q_0, p_1, p_0, a) \xrightarrow{?a}_L (q_1, p_1, p_0, a) \xrightarrow{!b}_L$$
$$(q_2, p_1, p_0, b) \xrightarrow{?b}_C (q_2, p_1, p_2, b) \xrightarrow{!c}_C (q_2, p_1, p_2, c) \xrightarrow{?c}_L$$
$$(q_3, p_1, p_2, c) \xrightarrow{!a}_C (q_3, p_1, p_2, a) \xrightarrow{?a}_L (q_4, p_1, p_2, a).$$

We construct a witness candidate out of the computation. To this end, we only keep the first writes of the contributors. These are the write $!a$ in the first transition and the write $!c$ in the fifth transition. Both are marked red. They will be represented in the witness candidate by the symbols $\bar{a}, \bar{c} \in \bar{D}$.

Now we map the transitions of the leader to words. Writes are preserved, reads are mapped to $\bot$. Then we obtain the witness candidate

$$\bar{a} \,.\, q_0 \,.\, \bot \,.\, q_1 \,.\, b \,.\, \bar{c} \,.\, q_2.$$

Note that we omit the last two transitions of the leader. The reason is as follows. After the first write $\bar{c}$, the leader is in state q_2. From this state, the leader can reach q_4 while only reading from first writes that have already appeared in the witness candidate, namely a and c. Hence, we can truncate the witness candidate at that point and do not have to keep the remaining computation to q_4.

To characterize computations in terms of witness candidates, we define the notion of validity. This needs some notation. Consider a word $w = w_1 \dots w_\ell$ over some alphabet Γ. For $i \in [1..\ell]$, we set $w[i] = w_i$ and $w[1..i] = w_1 \dots w_i$. If $\Gamma' \subseteq \Gamma$, we use $w \downarrow_{\Gamma'}$ for the projection of w to the letters in Γ'.

Consider a witness candidate $w \in \mathcal{E}$ and let $i \in [1..|w|]$. We use $\bar{D}(w, i)$ for the set of all first writes that occurred in w up to position i. Formally, we define it to be $\bar{D}(w, i) = \{a \mid \bar{a} \text{ is a letter in } w[1..i]\downarrow_{\bar{D}}\}$. We abbreviate $\bar{D}(w, |w|)$ as $\bar{D}(w)$. Let $q \in Q_L$ and $S \subseteq D$. Recall that the state represents a loop in P_L. The set of all letters written within a loop from q to q when reading only symbols from S is $\texttt{Loop}(q, S) = \{a \mid a \in D \text{ and } \exists v_1, v_2 \in (W(D) \cup R(S))^* : q \xrightarrow{v_1 ! a v_2}_L q\}$.

The definition of validity is given next. Technical details of the three requirements are made precise in the text below.

Definition 4 A witness candidate $w \in \mathcal{E}$ is *valid* if it satisfies the following properties: (1) First writes are unique. (2) The word w encodes a run on P_L. (3) There are supportive computations on the contributors.

(1) If $w \downarrow_{\bar{D}} = \bar{c_1} \dots \bar{c_\ell}$, then the $\bar{c_i}$ are pairwise different.

(2) Let $w \downarrow_{Q_L \cup D_\bot} = q_1 a_1 q_2 a_2 \dots a_\ell q_{\ell+1}$. If $a_i \in D$, then $q_i \xrightarrow{!a_i}_L q_{i+1} \in \delta_L$ is a write transition of P_L. If $a_i = \bot$, then we have an ε-transition $q_i \xrightarrow{\varepsilon}_L q_{i+1}$. Alternatively, there is a read $q_i \xrightarrow{?a}_L q_{i+1}$ of a symbol $a \in \bar{D}(w, \texttt{pos}(a_i))$ that already occurred within

a first write (the leader does not read its own writes). Here, we use $\texttt{pos}(a_i)$ to access the position of a_i in w. State $q_1 = q_L^0$ is initial. There is a run from $q_{\ell+1}$ to a state $q_f \in F_L$. During this run, reading is restricted to symbols that occurred as first writes in w. Formally, there is a word $v \in (W(D) \cup R(\bar{D}(w)))^*$ leading to an unsafe state q_f. We have $q_{\ell+1} \xrightarrow{v}_L q_f$.

(3) For each prefix $v\bar{a}$ of w with $\bar{a} \in \bar{D}$ there is a computation $q_C^0 \xrightarrow{u!a}_C q$ on P_C so that the reads in u can be obtained from v. Formally, let $u' = u \downarrow_{R(D)}$. Then there is an embedding of u' into v, a monotone map $\mu : [1..|u'|] \to [1..|v|]$ that satisfies the following. Let $u'[i] = ?a$ with $a \in D$. The read is served in one of the following three ways. We may have $v[\mu(i)] = a$, which corresponds to a write of a by P_L. Alternatively, $v[\mu(i)] = q \in Q_L$ and $a \in \texttt{Loop}(q, \bar{D}(w, \mu(i)))$. This amounts to reading from a leader's write that was executed in a loop. Finally, we may have $a \in \bar{D}(w, \mu(i))$, corresponding to reading from another contributor.

Our goal is to prove that a valid witness candidate exists if and only if there is a computation leading to an unsafe state. Before we state the corresponding lemma, we provide some intuition for the three requirements along an example.

Example 5 Reconsider the program $\mathcal{A}$ from Fig. 1. We elaborate on why the three requirements for validity are essential. To this end, we present three witness candidates, each violating one of the requirements. Thus, these candidates cannot correspond to an actual computation of the program.

The witness candidate $w_1 = \bar{a} \,.\, q_0 \,.\, \bot \,.\, q_1 \,.\, b \,.\, \bar{a} \,.\, q_2$ clearly violates requirement (1) due to the repetition of $\bar{a}$. Since first writes are unique there cannot exist a computation of program $\mathcal{A}$ following candidate w_1.

Requirement (2) asks for a proper run on the leader thread P_L. Hence, the witness candidate $w_2 = \bar{a} \,.\, q_0 \,.\, a \,.\, q_1 \,.\, b \,.\, \bar{c} \,.\, q_2$ violates the requirement although it satisfies (1). The subword $q_0 \,.\, a \,.\, q_1$ of w_2 encodes that the leader should take the transition $q_0 \xrightarrow{!a}_L q_1$. But this transition does not exist in P_L. Consequently, there is no computation of $\mathcal{A}$ which corresponds to the witness candidate w_2.

For requirement (3), consider the candidate $w_3 = \bar{a} \,.\, q_0 \,.\, \bot \,.\, q_1 \,.\, \bot \,.\, \bar{c} \,.\, q_2$. It clearly satisfies (1). Requirement (2) is also fulfilled. In fact, the subwords encoding transitions of the leader are $q_0 \,.\, \bot \,.\, q_1$ and $q_1 \,.\, \bot \,.\, q_2$. The first subword corresponds to transition $q_0 \xrightarrow{?a}_L q_1$ which can be taken since a already appeared as a first write in w_3. The second subword refers to the transition $q_1 \xrightarrow{\varepsilon}_L q_2$.

To explain that w_3 does not satisfy requirement (3), we show that c cannot be provided as a first write. To this end, assume that w_3 satisfies (3). Then, for the prefix $v.\bar{c}$ with $v = \bar{a} \,.\, q_0 \,.\, \bot \,.\, q_1 \,.\, \bot$, there is a computation of the form $p_0 \xrightarrow{u!c}_C p_2$. The reads in u are either first writes in v or writes provided by the leader (potentially in loops). Symbol b is not provided as such: It is neither a first write in v nor a symbol written by the leader (in a loop) along v. However, a computation u leading to state p_2 in P_C needs to read b once. Hence, such a computation does not exist and c cannot be provided as a first write.

The witness candidate $w = \bar{a} \,.\, q_0 \,.\, \bot \,.\, q_1 \,.\, b \,.\, \bar{c} \,.\, q_2$ from Example 3 satisfies all the requirements. In particular (3) is fulfilled since b is written by the leader in the transition $q_1 \xrightarrow{!b} q_2$. Hence, in this case, c can be provided as a first write.

Lemma 6 *There is a $t \in \mathbb{N}$ so that $c^0 \to_{\mathcal{A}^t}^* c$ with $c \in C^f$ if and only if there is a valid witness candidate $w \in \mathcal{E}$.*

Our algorithm iterates over all witness candidates $w \in \mathcal{E}$ and tests whether w is valid. The number of candidates $\mathit{Wit}(\mathtt{L}, \mathtt{D})$ is $(\mathtt{L} \cdot (\mathtt{D}+1))^{\mathtt{L}\cdot\mathtt{D}} \cdot \mathtt{D}^{\mathtt{D}} \cdot \mathtt{L}$. This is due to the fact that we can force a witness candidate to have maximum length via inserting padding symbols. Hence, the number of candidates constitutes the first factor of the complexity estimation stated in Theorem 1. The polynomial factor $\mathit{Valid}(\mathtt{L}, \mathtt{D}, \mathtt{C})$ is due to the following lemma.

Lemma 7 *Validity of $w \in \mathcal{E}$ can be checked in time $\mathcal{O}(L^3 \cdot D^2 \cdot C^2)$.*

Practical Algorithm

We improve the above algorithm so that it should work well on practical instances. The idea is to factorize the leader along its *strongly connected components* (SCCs), the number of which is assumed to be small in real programs. Technically, our improved algorithm works with *valid SCC-witnesses*. They symbolically represent SCCs rather than loops in the leader. To state the complexity, we first define the *straight line depth*, the number of SCCs the leader may visit during a computation. The definition needs a graph construction.

Let $\mathcal{V} \subseteq \bar{D}^{\leq \mathtt{D}}$ contain only words that do not repeat letters. Let $r = \bar{c}_1 \dots \bar{c}_\ell \in \mathcal{V}$ and $i \in [0..\ell]$. By $P_L \downarrow_i$ we denote the automaton obtained from P_L by removing all transitions that read a value outside $\{c_1, \dots, c_i\}$. Let $\mathtt{SCC}(P_L \downarrow_i)$ denote the set of all SCCs in this automaton. We construct the directed graph $G(P_L, r)$ as follows. The vertices are the SCCs of all $P_L \downarrow_i$, $i \in [0..\ell]$. There is an edge between $S, S' \in \mathtt{SCC}(P_L \downarrow_i)$, if there are states $q \in S, q' \in S'$ with $q \xrightarrow{?a/!a/\varepsilon} q'$ in $P_L \downarrow_i$. If $S \in \mathtt{SCC}(P_L \downarrow_{i-1})$ and $S' \in \mathtt{SCC}(P_L \downarrow_i)$, we only get an edge if we can get from S to S' by reading c_i. Note that the graph is acyclic.

The depth $d(r)$ of P_L relative to r is the length of the longest path in $G(P_L, r)$. The *straight line depth* is $\mathtt{d} = \max\{d(r) \mid r \in \mathcal{V}\}$. The *number of SCCs* $\mathtt{s}$ is the size of $\mathtt{SCC}(P_L \downarrow_0)$. With these values at hand, the number of SCC-witness candidates (the definition of which can be found in the full version of the paper) can be bounded by $\mathit{Wit}_{SCC}(\mathtt{s}, \mathtt{D}, \mathtt{d}) \leq (\mathtt{s} \cdot (\mathtt{D}+1))^{\mathtt{d}} \cdot \mathtt{D}^{\mathtt{D}} \cdot 2^{\mathtt{D}+\mathtt{d}}$. The time needed to test whether a candidate is valid is $\mathit{Valid}_{SCC}(\mathtt{L}, \mathtt{D}, \mathtt{C}, \mathtt{d}) = \mathtt{L}^2 \cdot \mathtt{D} \cdot \mathtt{C}^2 \cdot \mathtt{d}^2$.

Theorem 8 LCR *can be solved in time $\mathcal{O}(\mathit{Wit}_{SCC}(s, D, d) \cdot \mathit{Valid}_{SCC}(L, D, C, d))$.*

For this algorithm, what matters is that the leader's state space is strongly connected. The number of states has limited impact on the runtime.

Lower Bound

We prove that the algorithm from Theorem 1 is only a root-factor away from being optimal: A $2^{o(\sqrt{\mathtt{L}\cdot\mathtt{D}}\cdot\log(\mathtt{L}\cdot\mathtt{D}))}$-time algorithm for LCR would contradict ETH. We achieve the lower bound by a reduction from k × k Clique, the problem of finding a clique of size k in a graph the vertices of which are elements of a $k \times k$ matrix. Moreover, the clique has to contain one vertex from each row. Unless ETH fails, the problem cannot be solved in time $2^{o(k\cdot\log(k))}$ [39].

Technically, we construct from an instance (G, k) of k × k Clique an instance $(\mathcal{A} = (D, a^0, (P_L, P_C)), F_L)$ of LCR such that $\mathtt{D} = \mathcal{O}(k)$ and $\mathtt{L} = \mathcal{O}(k)$. Furthermore, we show that G contains the desired clique of size k if and only if there is a $t \in \mathbb{N}$ such that $c^0 \to^*_{\mathcal{A}^t} c$ with $c \in C^f$. Suppose we had an algorithm for LCR running in time $2^{o(\sqrt{\mathtt{L}\cdot\mathtt{D}}\cdot\log(\mathtt{L}\cdot\mathtt{D}))}$. Combined with the reduction, this would yield an algorithm for k × k Clique with runtime $2^{o(\sqrt{k^2}\cdot\log(k^2))} = 2^{o(k\cdot\log k)}$. But unless the exponential time hypothesis fails, such an algorithm cannot exist.

Proposition 9 LCR *cannot be solved in time* $2^{o(\sqrt{L \cdot D} \cdot \log(L \cdot D))}$ *unless* ETH *fails.*

We assume that the vertices V of G are given by tuples (i, j) with $i, j \in [1..k]$, where i denotes the row and j denotes the column in the matrix. In the reduction, we need the leader and the contributors to communicate on the vertices of G. However, we cannot store tuples (i, j) in the memory as this would cause a quadratic blow-up $\mathtt{D} = \mathcal{O}(k^2)$. Instead, we communicate a vertex (i, j) as a string $\mathtt{row}(i).\mathtt{col}(j)$. We distinguish between row- and column-symbols to avoid stuttering, the repeated reading of the same symbol. With this, it cannot happen that a thread reads a row-symbol twice and takes it for a column.

The program starts its computation with each contributor choosing a vertex (i, j) to store. For simplicity, we denote a contributor storing the vertex (i, j) by $P_{(i,j)}$. Note that there can be copies of $P_{(i,j)}$.

Since there are arbitrarily many contributors, the chosen vertices are only a superset of the clique we want to find. To cut away the false vertices, the leader P_L guesses for each row the vertex belonging to the clique. Contributors storing other vertices than the guessed ones will be switched off bit by bit. To this end, the program performs for each $i \in [1..k]$ the following steps: If (i, j_i) is the vertex of interest, P_L first writes $\mathtt{row}(i)$ to the memory. Each contributor that is still active reads the symbol and moves on for one state. Then P_L communicates the column by writing $\mathtt{col}(j_i)$. Again, the active contributors $P_{(i',j')}$ read.

Upon transmitting (i, j_i), the contributors react in one of the following three ways: (1) If $i' \neq i$, the contributor $P_{(i',j')}$ stores a vertex of a different row. The computation in $P_{(i',j')}$ can only go on if (i', j') is connected to (i, j_i) in G. Otherwise it will stop. (2) If $i' = i$ and $j' = j_i$, then $P_{(i',j')}$ stores exactly the vertex guessed by P_L. In this case, $P_{(i',j')}$ can continue its computation. (3) If $i' = i$ and $j' \neq j$, thread $P_{(i',j')}$ stores a different vertex from row i. The contributor has to stop.

After k such rounds, there are only contributors left that store vertices guessed by P_L. Furthermore, each two of these vertices are connected. Hence, they form a clique. To transmit this information to P_L, each $P_{(i,j_i)}$ writes $\#_i$ to the memory, a special symbol for row i. After P_L has read the string $\#_1 \ldots \#_k$, it moves to its final state. A formal construction can be found in the full version of the paper.

Note that the size $\mathcal{O}(k)$ of the data domain cannot be avoided, even if we encoded the row and column symbols in binary. The reason is that P_L needs a confirmation of k contributors that were not stopped during the guessing and terminated correctly. Since contributors do not have final states, we need to transmit this information in the form of k different memory symbols.

Absence of a Polynomial Kernel

A kernelization of a parameterized problem is a compression algorithm. Given an instance, it returns an equivalent instance the size of which is bounded by a function only in the parameter. From an algorithmic perspective, kernels put a bound on the number of hard instances. Indeed, the search for small kernels is a key interest in algorithmics, similar to the search for FPT-algorithms. It can be shown that kernels exist if and only if a problem admits an FPT-algorithm [13].

Let Q be a parameterized problem. A *kernelization* of Q is an algorithm that given an instance (B, k), runs in polynomial time in B and k, and outputs an equivalent instance (B', k') such that $|B'| + k' \leq g(k)$. Here, g is a computable function. If g is a polynomial, we say that Q admits a *polynomial kernel*.

Unfortunately, for many problems the community failed to come up with polynomial kernels. This lead to the contrary approach, namely disproving their existence [5,6,28]. The absence of a polynomial kernel constitutes an exponential lower bound on the number of hard instances. Like computational hardness results, such a bound is seen as an indication of general hardness of the problem. Technically, the existence of a polynomial kernel for the problem of interest is shown to imply NP $\subseteq$ coNP/poly. However, the inclusion is considered unlikely as it would cause a collapse of the polynomial hierarchy to the third level [42].

In order to link the existence of a polynomial kernel for LCR(D, L) with the above inclusion, we follow the framework developed in [6]. Let Γ be an alphabet. A *polynomial equivalence relation* is an equivalence relation $\mathcal{R}$ on Γ^* with the following properties: Given $x, y \in \Gamma^*$, it can be decided in time polynomial in $|x| + |y|$ whether $(x, y) \in \mathcal{R}$. Moreover, for $n \in \mathbb{N}$ there are at most polynomially many equivalence classes in $\mathcal{R}$ restricted to $\Gamma^{\leq n}$.

The key tool for proving kernel lower bounds are cross-compositions. Let $L \subseteq \Gamma^*$ be a language and $Q \subseteq \Gamma^* \times \mathbb{N}$ be a parameterized language. We say that L *cross-composes* into Q if there exists a polynomial equivalence relation $\mathcal{R}$ and an algorithm $\mathcal{C}$, together called the *cross-composition*, with the following properties: $\mathcal{C}$ takes as input $\varphi_1, \ldots, \varphi_I \in \Gamma^*$, all equivalent under $\mathcal{R}$. It computes in time polynomial in $\sum_{\ell=1}^{I} |\varphi_\ell|$ a string $(y, k) \in \Gamma^* \times \mathbb{N}$ such that $(y, k) \in Q$ if and only if there is an $\ell \in [1..I]$ with $\varphi_\ell \in L$. Furthermore, parameter k is bounded by $p(\max_{\ell \in [1..I]} |\varphi_\ell| + \log(I))$, where p is a polynomial.

It was shown in [6] that a cross-composition of any NP-hard language into a parameterized language Q prohibits the existence of a polynomial kernel for Q unless NP $\subseteq$ coNP/poly. In order to make use of this result, we show how to cross-compose 3-SAT into LCR(D, L). This yields the following:

Theorem 10 LCR(D, L) *does not admit a polynomial kernel unless* NP $\subseteq$ coNP/*poly*.

The difficulty in coming up with a cross-composition is the restriction on the size of the parameters. In our case, this affects D and L: Both parameters are not allowed to depend polynomially on I, the number of given 3-SAT-instances. We resolve the polynomial dependence by encoding the choice of such an instance into the contributors via a binary tree.

Proof (*Idea*) Assume some encoding of Boolean formulas as strings over a finite alphabet. We use the polynomial equivalence relation $\mathcal{R}$ defined as follows: Two strings φ and ψ are equivalent under $\mathcal{R}$ if both encode 3-SAT-instances, and the numbers of clauses and variables coincide.

Let the given 3-SAT-instances be $\varphi_1, \ldots, \varphi_I$. Every two of them are equivalent under $\mathcal{R}$. This means all φ_ℓ have the same number of clauses m and use the same set of variables $\{x_1, \ldots, x_n\}$. We assume that $\varphi_\ell = C_1^\ell \wedge \cdots \wedge C_m^\ell$.

We construct a program proceeding in three phases. First, it chooses an instance φ_ℓ, then it guesses an evaluation for all variables, and in the third phase it verifies that the evaluation satisfies φ_ℓ. While the second and the third phase do not cause a dependence of the parameters on I, the first phase does. It is not possible to guess a number $\ell \in [1..I]$ and communicate it via the memory as this would provoke a polynomial dependence of D on I.

To implement the first phase without a polynomial dependence, we transmit the indices of the 3-SAT-instances in binary. The leader guesses and writes tuples $(u_1, 1), \ldots, (u_{\log(I)}, \log(I))$ with $u_\ell \in \{0, 1\}$ to the memory. This amounts to choosing an instance φ_ℓ with binary representation $\mathtt{bin}(\ell) = u_1 \ldots u_{\log(I)}$.

It is the contributors' task to store this choice. Each time the leader writes a tuple (u_i, i), the contributors read and branch either to the left, if $u_i = 0$, or to the right, if $u_i = 1$. Hence, in the first phase, the contributors are binary trees with I leaves, each leaf storing the index

of an instance φ_ℓ. Since we did not assume that I is a power of 2, there may be computations arriving at leaves that do not represent proper indices. In this case, the computation deadlocks.

The size of D and P_L in the first phase is $\mathcal{O}(\log(I))$. Note that this satisfies the size-restrictions of a cross-composition.

For guessing the evaluation in the second phase, the program communicates on tuples (x_i, v) with $i \in [1..n]$ and $v \in \{0, 1\}$. The leader guesses such a tuple for each variable and writes it to the memory. Any participating contributor is free to read one of the tuples. After reading, it stores the variable and the evaluation.

In the third phase, the satisfiability check is performed as follows: Each contributor that is still active has stored in its current state the chosen instance φ_ℓ, a variable x_i, and its evaluation v_i. Assume that x_i when evaluated to v_i satisfies C^ℓ_j, the j-th clause of φ_ℓ. Then the contributor loops in its current state while writing the symbol $\#_j$. The leader waits to read the string $\#_1 \dots \#_m$. If P_L succeeds, we are sure that the m clauses of φ_ℓ were satisfied by the chosen evaluation. Thus, φ_ℓ is satisfiable and P_L moves to its final state. For details of the construction and a proof of correctness, we refer to the full version. □

3.2 Parameterization by Contributors

The size of the contributors $\mathtt{C}$ has substantial influence on the complexity of LCR. We show that the problem can be solved in time $\mathcal{O}^*(2^{\mathtt{C}})$ via dynamic programming. Moreover, we present a matching lower bound proving it unlikely that LCR can be solved in time $\mathcal{O}^*((2-\delta)^{\mathtt{C}})$, for any $\delta > 0$. The result is obtained by a reduction from Set Cover. Finally, a lower bound for the kernel of LCR($\mathtt{C}$) is provided.

Upper Bound

Our algorithm is based on dynamic programming. Intuitively, we cut a computation of the program along the states reached by the contributors. To this end, we keep a table with an entry for each subset of the contributors' states. The entry of set $S \subseteq Q_C$ contains those states of the leader that are reachable under a computation where the behavior of the contributors is limited to S. We fill the table by a dynamic programming procedure and check in the end whether a final state of the leader occurs in an entry. The result is as follows.

Theorem 11 LCR *can be solved in time* $\mathcal{O}(2^C \cdot C^4 \cdot L^2 \cdot D^2)$.

To define the table, we first need a more compact way of representing computations that allows for fast iteration. The observation is that keeping one set of states for all contributors suffices. Let $S \subseteq Q_C$ be the set of states reachable by the contributors in a given computation. By the *Copycat Lemma* [18], we can assume for each $q \in S$ an arbitrary number of contributors that are currently in q. This means that we do not have to distinguish between different contributor instances.

Formally, we reduce the search space to $V = Q_L \times D \times \mathcal{P}(Q_C)$. Instead of explicit configurations, we consider tuples (q, a, S), where $q \in Q_L$, $a \in D$, and $S \subseteq Q_C$. Between these tuples, we define an edge relation E. If P_L writes $a \in D$ with transition $q \xrightarrow{!a} q'$, we get $(q, b, S) \to_E (q', a, S)$ for each $b \in D$ and $S \subseteq Q_C$. Reads of the leader are similar. Contributors also change the memory but saturate set S instead of changing the state: If there is a transition $p \xrightarrow{!a} p'$ in P_C with $p \in S$, then $(q, b, S) \to_E (q, a, S \cup \{p'\})$ for each $b \in D$ and $q \in Q_L$. Reads are similar.

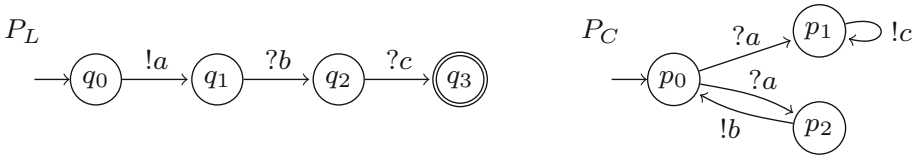

Fig. 2 Leader thread P_L (left) and contributor thread P_C (right). The data domain is given by $D = \{a^0, a, b, c\}$ and the only unsafe state is $F_L = \{q_3\}$

The set V together with the relation E form a finite directed graph $\mathcal{G} = (V, E)$. We call the node $v^0 = (q_L^0, a^0, \{q_C^0\})$ the *initial node*. Computations are represented by paths in $\mathcal{G}$ starting in v^0. Hence, we reduced LCR to the problem of checking whether the set of nodes $F_L \times D \times \mathcal{P}(Q_C)$ is reachable from the initial node in $\mathcal{G}$.

Lemma 12 *There is a $t \in \mathbb{N}$ so that $c^0 \to_{\mathcal{A}^t}^* c$ with $c \in C^f$ if and only if there is a path in $\mathcal{G}$ from v^0 to a node in $F_L \times D \times \mathcal{P}(Q_C)$.*

Before we elaborate on the algorithm solving reachability on $\mathcal{G}$ we turn to an example. It shows how $\mathcal{G}$ is constructed from a program and illustrates Lemma 12.

Example 13 We consider the program $\mathcal{A} = (D, a^0, (P_L, P_C))$ from Fig. 2. The nodes of the corresponding graph $\mathcal{G}$ are given by $V = Q_L \times D \times \mathcal{P}(\{p_0, p_1, p_2\})$. Its edges E are constructed following the above rules. For instance, we get an edge $(q_1, a, \{p_0\}) \to_E (q_1, a, \{p_0, p_1\})$ since P_C has a read transition $p_0 \xrightarrow{?a} p_1$. Intuitively, the edge describes that currently, the leader is in state q_1, the memory holds a, and an arbitrary number of contributors is waiting in p_0. Then, some of these read a and move to p_1. Hence, we might assume an arbitrary number of contributors in the states p_0 and p_1.

The complete graph $\mathcal{G}$ is presented in Fig. 3. For the purpose of readability, we only show the nodes that are reachable from $v^0 = (q_0, a^0, \{p_0\})$. Moreover, we omit self-loops and we present the graph as a collection of subgraphs. The latter means that for each subset S of $\mathcal{P}(\{p_0, p_1, p_2\})$, we consider the induced subgraph $\mathcal{G}[Q_L \times D \times \{S\}]$. It contains the nodes $Q_L \times D \times \{S\}$ and all edges that start and end in this set. Note that we omit the last component from a node (q, a, S) in $\mathcal{G}[Q_L \times D \times \{S\}]$. The induced subgraphs are connected by edges that saturate S.

The red marked nodes are those which contain the unsafe state q_3 of the leader. Consider a path from v^0 to one of these nodes. It starts in $\mathcal{G}[Q_L \times D \times \{\{p_0\}\}]$. To reach one of the red nodes, the path has to traverse via $\mathcal{G}[Q_L \times D \times \{\{p_0, p_1\}\}]$ to $\mathcal{G}[Q_L \times D \times \{Q_C\}]$ or via $\mathcal{G}[Q_L \times D \times \{\{p_0, p_2\}\}]$. Phrased differently, the states of the contributors need to be saturated two times along the path. This means that in an actual computation, there must be contributors in p_0, p_1, and p_2. These can then provide the symbols b and c which are needed by the leader to reach q_3.

Constructing $\mathcal{G}$ for a program and solving reachability takes time $\mathcal{O}^*(4^C)$ [10]. Hence, we have to solve reachability without constructing $\mathcal{G}$ explicitly. Our algorithm computes a table T which admits a recurrence relation that simplifies the reachability query: Instead of solving reachability directly on $\mathcal{G}$, we can restrict to so-called *slices* of $\mathcal{G}$. These are subgraphs of polynomial size where reachability queries can be decided efficiently.

We define the table T. For each set $S \subseteq Q_C$, we have an entry $T[S]$, given by:

$$T[S] = \{(q, a) \in Q_L \times D \mid v^0 \to_E^* (q, a, S)\}.$$

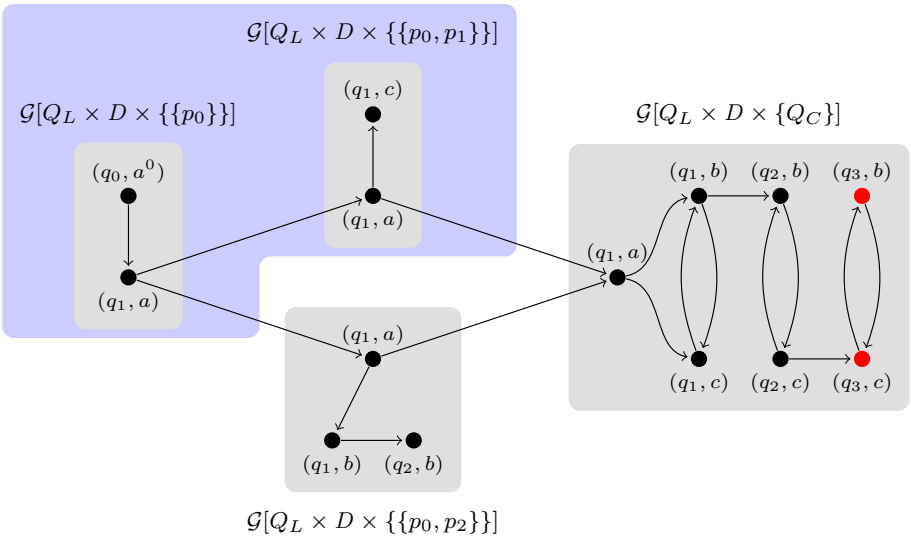

Fig. 3 Graph $\mathcal{G}$ summarizing the computations of the program in Fig. 2. Self-loops and nodes not reachable from $v^0 = (q_0, a^0, \{p_0\})$ are omitted. We further omit the third component of nodes since it is clear from the context. Nodes that are marked red involve the unsafe state q_3 of the leader. The blue highlighted area shows the slice $\mathcal{G}_{\{p_0\},\{p_0,p_1\}}$. (Color figure online)

Intuitively, $T[S]$ contains all nodes in $\mathcal{G}[Q_L \times D \times \{S\}]$ that are reachable from v^0.

Assume we have already computed T. By Lemma 12 we get: There is a $t \in \mathbb{N}$ so that $c^0 \to^*_{\mathcal{A}^t} c \in C^f$ if and only if there is an $S \subseteq Q_C$ such that $T[S] \cap F_L \times D \neq \emptyset$. The latter can be checked in time $\mathcal{O}(2^{\mathtt{C}} \cdot \mathtt{L}^2 \cdot \mathtt{D}^2)$ as there are $2^{\mathtt{C}}$ candidates for S.

It remains to compute the table. Our goal is to employ a dynamic programming based on a recurrence relation over T. To formulate the relation, we need the notion of *slices* of $\mathcal{G}$. Let $W \subseteq Q_C$ be a subset and $p \in Q_C \setminus W$ be a state. We denote by S the union $S = W \cup \{p\}$. The *slice* $\mathcal{G}_{W,S}$ is the induced subgraph $\mathcal{G}[Q_L \times D \times \{W, S\}]$. We denote its set of edges by $E_{W,S}$.

The main idea of the recurrence relation is saturation. When traversing a path π in $\mathcal{G}$, the set of contributor states gets saturated over time. Assume we cut π each time after a new state gets added. Then we obtain subpaths, each being a path in a slice: If $p \in Q_C$ gets added to $W \subseteq Q_C$, the corresponding subpath is in $\mathcal{G}_{W,W\cup\{p\}}$. Phrased differently, for a set $S \subseteq Q_C$, the entry $T[S]$ is the union of those nodes that are reachable from $T[S \setminus \{p\}]$ in $\mathcal{G}_{S\setminus\{p\},S}$, for each $p \in S$.

Formally, we define sets $R(W, S)$ for each $W \subseteq Q_C$, $p \in Q_C \setminus W$, and $S = W \cup \{p\}$. These sets collect the nodes that are reachable from $T[W]$ in the slice $\mathcal{G}_{W,S}$:

$$R(W, S) = \{(q, a) \in Q_L \times D \mid \exists (q', a') \in T[W] \text{ with } (q', a', W) \to^*_{E_{W,S}} (q, a, S)\}.$$

Lemma 14 *Table T admits the recurrence relation $T[S] = \bigcup_{p\in S} R(S \setminus \{p\}, S)$.*

We illustrate the lemma and the introduced notions on an example. Afterwards, we show how to compute the table T by exploiting the recurrence relation.

Example 15 Reconsider the program given in Fig. 2. The table T has eight entries, one for each subset of Q_C. The entries that are non-empty can be seen in the graph of Fig. 3.

Each of the subgraphs contains exactly those nodes that are reachable from v^0. For instance $T[\{p_0, p_1\}] = \{(q_1, a), (q_1, c)\}$.

Let $W = \{p_0\}$ and $S = \{p_0, p_1\}$. Then, the slice $\mathcal{G}_{W,S}$ is shown in the figure as blue highlighted area. Note that it also contains the edge from $(q_1, a, \{p_0\})$ to $(q_1, a, \{p_0, p_1\})$, leading from $\mathcal{G}[Q_L \times D \times \{\{p_0\}\}]$ to $\mathcal{G}[Q_L \times D \times \{\{p_0, p_1\}\}]$.

The set $R(W, S)$ contains those nodes in $\mathcal{G}[Q_L \times D \times \{S\}]$ that are reachable from $T[W]$ in the slice $\mathcal{G}_{W,S}$. According to the graph, these are (q_1, a, S) and (q_1, c, S) and hence we get $T[S] = R(W, S)$.

In general, not all nodes in $T[S]$ are reachable from a single set $T[W]$. But if a node is reachable, then it is reachable from some set $T[S \setminus \{p\}]$ with $p \in S$. Note that this is covered by the recurrence relation in Lemma 14. It branches over all sets $S \setminus \{p\}$ and hence collects all nodes that are reachable from such a set.

We apply the recurrence relation in a bottom-up dynamic programming to fill the table T. Let $S \subseteq Q_C$ be a subset and assume we already know $T[S \setminus \{p\}]$, for each $p \in S$. Then, for a fixed p, we compute $R(S \setminus \{p\}, S)$ by a fixed-point iteration on the slice $\mathcal{G}_{S\setminus\{p\},S}$. The number of nodes in the slice is $\mathcal{O}(\mathtt{L} \cdot \mathtt{D})$. Hence, the iteration takes time at most $\mathcal{O}(\mathtt{L}^2 \cdot \mathtt{D}^2)$. It is left to construct $\mathcal{G}_{S\setminus\{p\},S}$. We state the time needed in the following lemma. The proof is postponed so as to finish the complexity estimation of Theorem 11.

Lemma 16 *Slice $\mathcal{G}_{S\setminus\{p\},S}$ can be constructed in time $\mathcal{O}(C^3 \cdot L^2 \cdot D^2)$.*

Wrapping up, we need $\mathcal{O}(\mathtt{C}^3 \cdot \mathtt{L}^2 \cdot \mathtt{D}^2)$ time for computing a set $R(S \setminus \{p\}, S)$. Due to the recurrence relation of Lemma 14, we have to compute at most $\mathtt{C}$ sets $R(S \setminus \{p\}, S)$ for a given $S \subseteq Q_C$. Hence, an entry $T[S]$ can be computed in time $\mathcal{O}(\mathtt{C}^4 \cdot \mathtt{L}^2 \cdot \mathtt{D}^2)$. The estimation also covers the base case $S = \{p_C^0\}$, where $T[S]$ can be computed by a fixed-point iteration on the induced subgraph $\mathcal{G}[Q_L \times D \times \{S\}]$. Since the table T has $2^{\mathtt{C}}$ entries, the complexity estimation of Theorem 11 follows. It is left to prove Lemma 16.

Proof The slice $\mathcal{G}_{S\setminus\{p\},S}$ consists of the two subgraphs $\mathcal{G}_{S\setminus\{p\}} = \mathcal{G}[Q_L \times D \times \{S \setminus \{p\}\}]$ and $\mathcal{G}_S = \mathcal{G}[Q_L \times D \times \{S\}]$, and the edges leading from $\mathcal{G}_{S\setminus\{p\}}$ to $\mathcal{G}_S$. We elaborate on how to construct $\mathcal{G}_S$. The construction of $\mathcal{G}_{S\setminus\{p\}}$ is similar.

First, we write down the nodes of $\mathcal{G}_S$. This can be done in time $\mathcal{O}(\mathtt{L} \cdot \mathtt{D})$. Edges in the graph are either induced by transitions of the leader or by the contributor. The former ones can be added in time $\mathcal{O}(|\delta_L| \cdot \mathtt{D}) = \mathcal{O}(\mathtt{L}^2 \cdot \mathtt{D}^2)$ since a single transition of P_L may lead to $\mathtt{D}$ edges. To add the latter edges, we browse δ_C for transitions of the form $s \xrightarrow{!a} s'$ with $s, s' \in S$. Each such transition may induce $\mathtt{L} \cdot \mathtt{D}$ edges. Adding them takes time $\mathcal{O}(|\delta_C| \cdot \mathtt{C} \cdot \mathtt{L} \cdot \mathtt{D}) = \mathcal{O}(\mathtt{C}^3 \cdot \mathtt{L} \cdot \mathtt{D}^2)$ since we have to test membership of s, s' in S. Note that we can omit transitions $s \xrightarrow{?a} s'$ with $s, s' \in S$ as their induced edges are self-loops in $\mathcal{G}_S$.

To complete the construction, we add the edges from $\mathcal{G}_{S\setminus\{p\}}$ to $\mathcal{G}_S$. These are induced by transitions $r \xrightarrow{?a/!a} p \in \delta_C$ with $r \in S \setminus \{p\}$. Since each of these may again lead to $\mathtt{L} \cdot \mathtt{D}$ different edges, adding all of them takes time $\mathcal{O}(\mathtt{C}^3 \cdot \mathtt{L} \cdot \mathtt{D}^2)$. In total, we estimate the time for the construction by $\mathcal{O}(\mathtt{C}^3 \cdot \mathtt{L}^2 \cdot \mathtt{D}^2)$. □

Lower Bound

We prove it unlikely that LCR can be solved in $\mathcal{O}^*((2-\delta)^{\mathtt{C}})$ time, for any $\delta > 0$. This shows that the algorithm from Sect. 3.2 has an optimal runtime. The lower bound is achieved by a

reduction from Set Cover, one of the 21 original NP-complete problems by Karp [37]. We state its definition.

Set Cover	
Input:	A family of sets $\mathcal{F} \subseteq \mathcal{P}(U)$ over a universe U, and $r \in \mathbb{N}$.
Question:	Are there sets $S_1, \ldots, S_r$ in $\mathcal{F}$ such that $U = \bigcup_{i \in [1..r]} S_i$?

Besides its NP-completeness, it is known that Set Cover admits an $\mathcal{O}^*(2^n)$-time algorithm [25], where n is the size of the universe U. However, no algorithm solving Set Cover in time $\mathcal{O}^*((2-\delta)^n)$ for a $\delta > 0$ is known so far. Actually, it is conjectured in [12] that such an algorithm cannot exist unless the SETH breaks.

While a proof for the conjecture in [12] is still missing, the authors provide evidence in the form of relative hardness. They obtain lower bounds for prominent problems by tracing back to the *assumed* lower bound of Set Cover. These bounds were not known before since SETH is hard to apply: No suitable reductions from SAT to these problems are known so far. Hence, Set Cover can be seen as an alternative source for lower bounds whenever SETH seems out of reach. This made the problem a standard assumption for hardness which is widely used [4,9,12].

To obtain the desired lower bound for LCR, we establish a polynomial time reduction from Set Cover that strictly preserves the parameter n. Formally, if $(\mathcal{F}, U, r)$ is an instance of Set Cover, we construct an instance $(\mathcal{A} = (D, a^0, (P_L, P_C)), F_L)$ of LCR where $\mathsf{C} = n + c$ with c a constant. Note that even a linear dependence on n is not allowed. Moreover, the instance satisfies the equivalence: There is a set cover if and only if there is a $t \in \mathbb{N}$ such that $c^0 \to^*_{\mathcal{A}^t} c$ with $c \in C^f$. Assume we had an $\mathcal{O}^*((2-\delta)^{\mathsf{C}})$-time algorithm for LCR. With the reduction, this would immediately yield an $\mathcal{O}^*((2-\delta)^{n+c}) = \mathcal{O}^*((2-\delta)^n)$-time algorithm for Set Cover.

Proposition 17 *If* LCR *can be solved in* $\mathcal{O}^*((2-\delta)^{\mathsf{C}})$ *time for a* $\delta > 0$*, then* Set Cover *can be solved in* $\mathcal{O}^*((2-\delta)^n)$ *time.*

For the proof of the proposition, we elaborate on the aforementioned reduction. The main idea is the following: We let the leader guess r sets from $\mathcal{F}$. The contributors store the elements that got covered by the chosen sets. In a final communication phase, the leader verifies that it has chosen a valid cover by querying whether all elements of U have been stored by the contributors.

Leader and contributors essentially communicate over the elements of U. For guessing r sets from $\mathcal{F}$, the automaton P_L consists of r similar phases. Each phase starts with P_L choosing an internal transition to a set $S \in \mathcal{F}$. Once S is chosen, the leader writes a sequence of all $u \in S$ to the memory.

A contributor in the program consists of $\mathsf{C} = n + 1$ states: An initial state and a state for each $u \in U$. When P_L writes an element $u \in S$ to the memory, there is a contributor storing this element in its states by reading u. Hence, each element that got covered by S is recorded in one of the contributors.

After r rounds of guessing, the contributors hold those elements of U that are covered by the chosen sets. Now the leader verifies that it has really picked a cover of U. To this end, it needs to check whether all elements of U have been stored by the contributors. Formally, the leader can only proceed to its final state if it can read the symbols $u^\#$, for each $u \in U$. A contributor can only write $u^\#$ to the memory if it stored the element u before. Hence, P_L reaches its final state if and only if a valid cover of U was chosen.

Absence of a Polynomial Kernel

We prove that 3-SAT can be cross-composed into LCR(C). This shows that the problem is unlikely to admit a polynomial kernel. The result is the following.

Proposition 18 LCR(C) *does not admit a polynomial kernel unless* NP $\subseteq$ coNP/*poly*.

For the cross-composition, let $\varphi_1, \ldots, \varphi_I$ be the given 3-SAT-instances, each two equivalent under $\mathcal{R}$, where $\mathcal{R}$ is the polynomial equivalence relation from Theorem 10. Then, each formula has the same number of clauses m and variables $x_1, \ldots, x_n$. Let us fix the notation to be $\varphi_\ell = C_1^\ell \wedge \cdots \wedge C_m^\ell$.

The basic idea is the following. Leader P_L guesses the formula φ_ℓ and an evaluation for the variables. The contributors store the latter. At the end, leader and contributors verify that the chosen evaluation indeed satisfies formula φ_ℓ.

For guessing φ_ℓ, the leader has a branch for each instance. Note that we can afford the size of the leader to depend on I since the cross-composition only restricts parameter C. Hence, we do not face the problem we had in Theorem 10.

Guessing the evaluation of the variables is similar to Theorem 10: The leader writes tuples (x_i, v_i) with $v_i \in \{0, 1\}$ to the memory. The contributors store the evaluation in their states. After this guessing-phase, the contributors can write the symbols $\#_j^\ell$, depending on whether the currently stored variable with its evaluation satisfies clause C_j^ℓ. As soon as the leader has read the complete string $\#_1^\ell \ldots \#_m^\ell$, it moves to its final state, showing that the evaluation satisfied all clauses of φ_ℓ.

Note that parameter C is of size $\mathcal{O}(n)$ and does not depend on I at all. Hence, the size-restrictions of a cross-composition are met.

3.3 Intractability

We show the W[1]-hardness of LCR(D) and LCR(L). Both proofs rely on a parameterized reduction from k - Clique, the problem of finding a clique of size k in a given graph. This problem is known to be W[1]-complete [15]. Our result is the following.

Proposition 19 *Both parameterizations,* LCR(D) *and* LCR(L)*, are* W[1]*-hard.*

We first reduce k - Clique to LCR(L). More precisely, we construct from an instance (G, k) of k - Clique in polynomial time an instance $(\mathcal{A} = (D, a^0, (P_L, P_C)), F_L)$ of LCR with $\mathtt{L} = \mathcal{O}(k)$. This meets the requirements of a parameterized reduction.

Program $\mathcal{A}$ operates in three phases. In the first phase, the leader chooses k vertices of the graph and writes them to the memory. Formally, it writes a sequence $(v_1, 1).(v_2, 2) \ldots (v_k, k)$ with v_i vertices of G. During this selection, the contributors non-deterministically choose to store a suggested vertex (v_i, i) in their state space.

In the second phase, the leader again writes a sequence of vertices using different symbols: $(w_1^\#, 1)(w_2^\#, 2) \ldots (w_k^\#, k)$. Note that the vertices w_i do not have to coincide with the vertices from the first phase. It is then the contributor's task to verify that the new sequence constitutes a clique. To this end, for each i, the program does the following: If a contributor storing (v_i, i) reads the value $(w_i^\#, i)$, the computation on the contributor can only continue if $w_i = v_i$. If a contributor storing (v_j, j) with $j \neq i$ reads $(w_i^\#, i)$, the computation can only continue if $v_j \neq w_i$ and if there is an edge between v_j and w_i.

Finally, in the third phase, we need to ensure that there was at least one contributor storing (v_i, i) and that the above checks were all positive. To this end, a contributor that has

successfully gone through the second phase and stores (v_i, i) writes the symbol $\#_i$ to the memory. The leader pends to read the sequence $\#_1 \dots \#_k$. This ensures the selection of k different vertices, each two being adjacent.

For proving W[1]-hardness of LCR(D), we reuse the above construction. However, the size of the data domain is $|V| \cdot k$, where V is the set of vertices of G. Hence, it is not a parameterized reduction for parameter D. The factor $|V|$ appears since leader and contributors communicate on the pure vertices. The main idea of the new reduction is to decrease the size of D by transmitting the vertices in binary. To this end, we add binary branching trees to the contributors that decode a binary encoding. We omit the details and refer to the full version of the paper.

4 Bounded-Stage Reachability

The *bounded-stage reachability problem* is a simultaneous reachability problem. It asks whether all threads of a program can reach an unsafe state when restricted to s-stage computations. These are computations where the write permission changes s times. The problem was first analyzed in [1] and shown to be NP-complete for finite state programs. We give matching upper and lower bounds in terms of fine-grained complexity and prove the absence of a polynomial kernel.

Let $\mathcal{A} = (D, a^0, (P_i)_{i \in [1..t]})$ be a program. A *stage* is a computation in $\mathcal{A}$ where only one of the threads writes. The remaining threads are restricted to reading the memory. An s-*stage computation* is a computation that can be split into s parts, each of which forming a stage. We state the decision problem.

Bounded-Stage Reachability (BSR)
Input: A program $\mathcal{A} = (D, a^0, (P_i)_{i \in [1..t]})$, a set $C^f \subseteq C$, and $s \in \mathbb{N}$.
Question: Is there an s-stage computation $c^0 \to^*_{\mathcal{A}} c$ for some $c \in C^f$?

We focus on a parameterization of BSR by P, the maximum number of states of a thread, and t, the number of threads. Let it be denoted by BSR(P, t). We prove that the parameterization is FPT and present a matching lower bound. The main result in this section is the absence of a polynomial kernel for BSR(P, t). The result is technically involved and shows what makes the problem hard.

Parameterizations of BSR involving only D and s are intractable. We show that BSR remains NP-hard even if both, D and s, are constants. This proves the existence of an FPT-algorithm for those cases unlikely.

4.1 Parameterization by Number of States and Threads

We first give an algorithm for BSR, based on a product construction of automata. Then, we present a lower bound under ETH. Interestingly, the lower bound shows that we cannot avoid building the product. We conclude with proving the absence of a polynomial kernel. As before, we cross-compose from 3-SAT but now face the problem that two important parameters in the construction, P and t, are not allowed to depend polynomially on the number of 3-SAT-instances.

Upper Bound

We show that BSR(P, t) is fixed-parameter tractable. The idea is to reduce to reachability on a product automaton. The automaton stores the configurations, the current writer, and counts up to the number of stages s. To this end, it has $\mathcal{O}^*(\mathtt{P}^{\mathtt{t}})$ many states. Details can be found in the full version of the paper.

Proposition 20 BSR *can be solved in time* $\mathcal{O}^*(P^{2t})$.

Lower Bound

By a reduction from k × k Clique, we show that a $2^{o(\mathtt{t}\cdot\log(\mathtt{P}))}$-time algorithm for BSR would contradict ETH. The above algorithm is optimal.

Proposition 21 BSR *cannot be solved in time* $2^{o(t\cdot\log(P))}$ *unless* ETH *fails.*

The reduction constructs from an instance of k × k Clique an equivalent instance $(\mathcal{A} = (D, a^0, (P_i)_{i\in[1..\mathtt{t}]}), C^f, \mathtt{s})$ of BSR. Moreover, it keeps the parameters small. We have that $\mathtt{P} = \mathcal{O}(k^2)$ and $\mathtt{t} = \mathcal{O}(k)$. As a consequence, a $2^{o(\mathtt{t}\cdot\log(\mathtt{P}))}$-time algorithm for BSR would yield an algorithm for k × k Clique running in time $2^{o(k\cdot\log(k^2))} = 2^{o(k\cdot\log(k))}$. But this contradicts ETH.

Proof (*Idea*) For the reduction, let $V = [1..k] \times [1..k]$ be the vertices of G. We define $D = V \cup \{a^0\}$ to be the domain of the memory. We want the threads to communicate on the vertices of G. For each row we introduce a reader thread P_i that is responsible for storing a particular vertex of the row. We also add one writer P_{ch} that is used to steer the communication between the P_i. Our program $\mathcal{A}$ is given by the tuple $(D, a^0, ((P_i)_{i\in[1..k]}, P_{ch}))$.

Intuitively, the program proceeds in two phases. In the first phase, each P_i non-deterministically chooses a vertex from the i-th row and stores it in its state space. This constitutes a clique candidate $(1, j_1), \ldots, (k, j_k) \in V$. In the second phase, thread P_{ch} starts to write a random vertex $(1, j_1')$ of the first row to the memory. The first thread P_1 reads $(1, j_1')$ from the memory and verifies that the read vertex is actually the one from the clique candidate. The computation in P_1 will deadlock if $j_1' \neq j_1$. The threads P_i with $i \neq 1$ also read $(1, j_1')$ from the memory. They have to check whether there is an edge between the stored vertex (i, j_i) and $(1, j_1')$. If this fails in some P_i, the computation in that thread will also deadlock. After this procedure, the writer P_{ch} guesses a vertex $(2, j_2')$, writes it to the memory, and the verification steps repeat. In the end, after k repetitions of the procedure, we can ensure that the guessed clique candidate is indeed a clique. Formal construction and proof are available in the full version of the paper. □

Absence of a Polynomial Kernel

We show that BSR(P, t) does not admit a polynomial kernel. To this end, we cross-compose the problem 3-SAT into BSR(P, t).

Theorem 22 BSR(P, t) *does not admit a polynomial kernel unless* NP ⊆ coNP/*poly*.

In the present setting, coming up with a cross-composition is non-trivial. Both parameters, P and t, are not allowed to depend polynomially on the number I of given 3-SAT-instances. Hence, we cannot construct an NFA that distinguishes the I instances by branching into I

different directions. This would cause a polynomial dependence of $\mathtt{P}$ on I. Furthermore, it is not possible to construct an NFA for each instance as this would cause such a dependence of $\mathtt{t}$ on I. To circumvent the problems, some deeper understanding of the model is needed.

Proof (*Idea*) Let $\varphi_1, \ldots, \varphi_I$ be given 3-SAT-instances, where each two are equivalent under $\mathcal{R}$, the polynomial equivalence relation of Theorem 10. Then each φ_ℓ has m clauses and n variables $\{x_1, \ldots, x_n\}$. We assume $\varphi_\ell = C_1^\ell \wedge \cdots \wedge C_m^\ell$.

In the program that we construct, the communication is based on 4-tuples of the form (ℓ, j, i, v). Intuitively, such a tuple transports the following information: The j-th clause in instance φ_ℓ, C_j^ℓ, can be satisfied by variable x_i with evaluation v. Hence, our data domain is $D = ([1..I] \times [1..m] \times [1..n] \times \{0, 1\}) \cup \{a^0\}$.

For choosing and storing an evaluation of the x_i, we introduce so-called variable threads $P_{x_1}, \ldots, P_{x_n}$. In the beginning, each P_{x_i} non-deterministically chooses an evaluation for x_i and stores it in its state space.

We further introduce a writer P_w. During a computation, this thread guesses exactly m tuples $(\ell_1, 1, i_1, v_1), \ldots, (\ell_m, m, i_m, v_m)$ in order to satisfy m clauses of potentially different instances. Each (ℓ_j, j, i_j, v_j) is written to the memory by P_w. All variable threads then start to read the tuple. If P_{x_i} with $i \neq i_j$ reads it, then the thread will just move one state further since the suggested tuple does not affect the variable x_i. If P_{x_i} with $i = i_j$ reads the tuple, the thread will only continue its computation if v_j coincides with the value that P_{x_i} guessed for x_i and, moreover, x_i with evaluation v_j satisfies clause $C_j^{\ell_j}$.

Now suppose the writer did exactly m steps while each variable thread did exactly $m + 1$ steps. This proves the satisfiability of m clauses by the chosen evaluation. But these clauses can be part of different instances: It is not ensured that the clauses were chosen from one formula φ_ℓ. The major difficulty of the cross-composition lies in how to ensure exactly this.

We overcome the difficulty by introducing so-called bit checkers P_b, where $b \in [1..\log(I)]$. Each P_b is responsible for the b-th bit of $\mathtt{bin}(\ell)$, the binary representation of ℓ, where φ_ℓ is the instance we want to satisfy. When P_w writes a tuple $(\ell_1, 1, i_1, v_1)$ for the first time, each P_b reads it and stores either 0 or 1, according to the b-th bit of $\mathtt{bin}(\ell_1)$. After P_w has written a second tuple $(\ell_2, 2, i_2, v_2)$, the bit checker P_b tests whether the b-th bit of $\mathtt{bin}(\ell_1)$ and $\mathtt{bin}(\ell_2)$ coincide, otherwise it will deadlock. This will be repeated any time P_w writes a new tuple to the memory.

Assume the computation does not deadlock in any of the P_b. Then we can ensure that the b-th bit of $\mathtt{bin}(\ell_j)$ with $j \in [1..m]$ never changed during the computation. This means that $\mathtt{bin}(\ell_1) = \cdots = \mathtt{bin}(\ell_m)$. Hence, the writer P_w has chosen clauses of just one instance φ_ℓ. Moreover, the current evaluation satisfies the formula. Since the parameters are bounded, $\mathtt{P} \in \mathcal{O}(m)$ and $\mathtt{t} \in \mathcal{O}(n + \log(I))$, the construction constitutes a proper cross-composition. For a formal construction and proof, we refer to the full version of the paper. □

Variable threads and writer thread are needed for testing satisfiability of clauses. The need for bit checkers comes from ensuring that all clauses stem from the same formula. We illustrate the notion with an example.

Example 23 Let four formulas $\varphi_1, \varphi_2, \varphi_3, \varphi_4$ with two clauses each be given. We show how the bit checkers are constructed. To this end, we first encode the index of the instances as binary numbers using two bits. The encoding is shown in Fig. 4 on the right hand side. Note the offset by one in the encoding.

We focus on the bit checker P_{b_1} responsible for the first bit. It is illustrated in Fig. 4 on the left hand side. Note that the label $\ell = 1, \ell = 3$ refers to transitions of the form $?(\ell, j, i, v)$

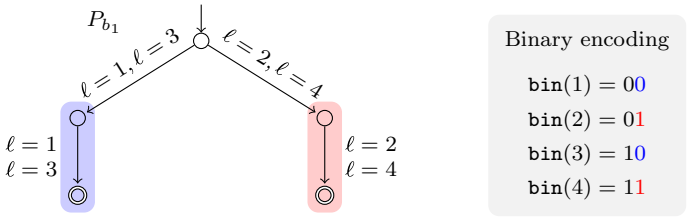

Fig. 4 A binary encoding (right) of the numbers 1 up to 4 using two bits. First bits are either marked blue, if they are 0, or red if they are 1. The bit checker P_{b_1} (left) focuses on the first bit. The label $\ell = 1, \ell = 3$ means that P_{b_1} has transitions on $?(\ell, j, i, v)$ for $\ell = 1, 3$ and arbitrary values for i, j, and v. The blue marked states store that the first bit b_1 is 0. Red marked states store that b_1 is 1. (Color figure online)

with ℓ either 1 or 3 and arbitrary values for i, j, and v. On reading the first of these tuples, P_{b_1} stores the first bit of ℓ in its state space. The blue marked states store that $b_1 = 0$, the red states store $b_1 = 1$. Then, the bit checker can only continue on reading tuples (ℓ, j, i, v) where the first bit of ℓ matches the stored bit. In the case of $b_1 = 0$, this means that P_{b_1} can only read tuples (ℓ, j, i, v) with ℓ either 1 or 3.

Assume the writer thread has output two tuples $(\ell_1, 1, i_1, v_1)$ and $(\ell_2, 2, i_2, v_2)$ and the bit checker P_{b_1} has reached a *last* state. Since the computation did not deadlock on P_{b_1}, we know that the first bits of ℓ_1 and ℓ_2 coincide. If the bit checker for the second bit does not deadlock as well, we get that $\ell_1 = \ell_2$. Hence, the writer has chosen two clauses from one instance φ_{ℓ_1}.

4.2 Intractability

We show that parameterizations of BSR involving only $\mathtt{s}$ and $\mathtt{D}$ are intractable. To this end, we prove that BSR remains NP-hard even if both parameters are constant. This is surprising as the number of stages $\mathtt{s}$ seems to be a powerful parameter. Introducing such a bound in simultaneous reachability lets the complexity drop from PSPACE to NP. But still, it is not enough to guarantee an FPT-algorithm.

Proposition 24 BSR *is* NP*-hard even if both* $\mathtt{s}$ *and* $\mathtt{D}$ *are constant.*

The proposition implies intractability: Assume there is an FPT-algorithm A for BSR running in time $f(\mathtt{s}, \mathtt{D}) \cdot poly(|x|)$, where x denotes the input. Then BSR′, the variant of BSR where $\mathtt{s}$ and $\mathtt{D}$ are constant, can also be solved by A. In this case, the runtime of A is $\mathcal{O}(poly(|x|))$ since $f(\mathtt{s}, \mathtt{D})$ is a constant on every instance of BSR′. But this contradicts the NP-hardness of BSR′ which is shown in the proposition.

Proof (*Idea*) We give a reduction from 3-SAT to BSR that keeps both parameters constant. Let φ be a 3-SAT-instance with m clauses and variables $x_1, \ldots, x_n$. We construct a program $\mathcal{A} = (D, a^0, P_1, \ldots, P_n, P_v)$ with $\mathtt{D} = 4$ different memory symbols that can only run 1-stage computations.

The program cannot communicate on literals directly, as this would cause a blow-up in parameter $\mathtt{D}$. Instead, variables and evaluations are encoded in binary in the following way. Let ℓ be a literal in φ. It consists of a variable x_i and an evaluation $v \in \{0, 1\}$. The padded binary encoding $\mathtt{bin}_{\#}(i) \in (\{0, 1\}.\#)^{\log(n)+1}$ of i is the usual binary encoding where each bit is separated by a #. The string $\mathtt{Enc}(\ell) = v\#\mathtt{bin}_{\#}(i)$ encodes that variable x_i has evaluation

v. We need the padding symbol # to prevent the threads in $\mathcal{A}$ from reading the same symbol more than once. Program $\mathcal{A}$ communicates by passing messages of the form $\texttt{Enc}(\ell)$. To this end, we need the data domain $D = \{a^0, \#, 0, 1\}$.

The program contains threads $P_i, i \in [1..n]$, called variable threads. Initially, these threads choose an evaluation for the variables and store it: Each P_i can branch on reading a^0 and choose whether it assigns 0 or 1 to x_i. Then, a verifier thread P_v starts to iterate over the clauses. For each clause C, it picks a literal $\ell \in C$ that should evaluate to true and writes its encoding $\texttt{Enc}(\ell)$ to the memory. Each of the P_i reads $\texttt{Enc}(\ell)$. Note that reading and writing $\texttt{Enc}(\ell)$ needs a sequence of transitions. In the construction, we ensure that all the needed states and transitions are provided. It is the task of each P_i to check whether the chosen literal ℓ is conform with the chosen evaluation for x_i. To this end, we distinguish two cases.

(1) If ℓ involves a variable x_j with $j \neq i$, variable thread P_i just continues its computation by reading the whole string $\texttt{Enc}(\ell)$.
(2) If ℓ involves x_i, P_i has to ensure that the stored evaluation coincides with the one sent by the verifier. To this end, P_i can only continue its computation if the first bit in $\texttt{Enc}(\ell)$ shows the correct evaluation. Formally, there is only an outgoing path of transitions on $\texttt{Enc}(x_i)$ if P_i stored 1 as evaluation and on $\texttt{Enc}(\neg x_i)$ if it stored 0.

Note that each time P_v picks a literal ℓ, all P_i read $\texttt{Enc}(\ell)$, even if the literal involves a different variable. This means that the P_i count how many literals have been seen already. This is important for correctness: The threads will only terminate if they have read a word of fixed length and did not miss a single symbol. Phrased differently, there is no loss in the communication between P_v and the P_i.

Now assume P_v iterated through all m clauses and none of the variable threads got stuck. Then, each of them read exactly m encodings without running into a deadlock. Hence, the picked literals were all conform with the evaluation chosen by the P_i. This means that a satisfying assignment for φ is found.

During a computation of $\mathcal{A}$, the verifier P_v is the only thread that has write permission. Hence, each computation of $\mathcal{A}$ consists of a single stage. For a formal construction, we refer to the full version of the paper. □

5 Conclusion

We have studied several parameterizations of LCR and BSR, two safety verification problems for shared memory concurrent programs. In LCR, a designated leader thread interacts with a number of equal contributor threads. The task is to decide whether the leader can reach an unsafe state. The problem BSR is a generalization of bounded context switching. A computation gets split into stages, periods where writing is restricted to one thread. Then, BSR asks whether all threads can reach a final state simultaneously during an $\texttt{s}$-stage computation.

For LCR, we identified the size of the data domain $\texttt{D}$, the size of the leader $\texttt{L}$ and the size of the contributors $\texttt{C}$ as parameters. Our first algorithm showed that LCR($\texttt{D}$, $\texttt{L}$) is FPT. Then we modified the algorithm to obtain a verification procedure valuable for practical instances. The main insight was that due to a factorization along strongly connected components, the impact of $\texttt{L}$ can be reduced to a polynomial factor in the time complexity. We also proved the absence of a polynomial kernel for LCR($\texttt{D}$, $\texttt{L}$) and presented an ETH-based lower bound which shows that the upper bound is a root-factor away from being optimal.

For LCR($\texttt{C}$) we presented a dynamic programming, running in $\mathcal{O}^*(2^{\texttt{C}})$ time. The algorithm is based on slice-wise reachability. This reduces a reachability problem on a large graph to

reachability problems on subgraphs (slices) that are solvable in polynomial time. Moreover, we gave a tight lower bound based on Set Cover and proved the absence of a polynomial kernel.

Parameterizations different from LCR(D, L) and LCR(C) were shown to be intractable. We gave reductions from k - Clique and proved W[1]-hardness.

The parameters of interest for BSR are the maximum size of a thread P and the number of threads t. We have shown that a parameterization by both parameters is FPT and gave a matching lower bound. The main contribution was to prove it unlikely that a polynomial kernel exists for BSR(P, t). The proof relies on a technically involved cross-composition that avoids a polynomial dependence of the parameters on the number of given 3-SAT-instances.

Parameterizations involving other parameters like S or D were proven to be intractable for BSR. We gave an NP-hardness proof where S and D are constant.

Extension of the Model

In this work, the considered model for programs allows the memory to consist of a single cell. We discuss whether the presented results carry over when the number of memory cells increases. Having multiple memory cells is referred to as supporting *global variables*. Extending the definition of programs in Sect. 2 to global variables is straightforward.

For the problem LCR, allowing global variables is a rather powerful mechanism. Let LCR_{Var} denote the problem LCR where the input is a program featuring global variables. The interesting parameters for the problem are D, L, C, and V, the number of global variables. It turns out that LCR_{Var} is PSPACE-hard, even when C is constant. One can reduce the intersection emptiness problem for finite automata to LCR_{Var}. The reduction makes use only of the leader, contributors are not needed.

A program $\mathcal{A}$ with global variables can always be reduced to a program $\mathcal{A}'$ with a single memory cell [26]. Roughly, the reduction constructs the leader of $\mathcal{A}'$ in such a way that it can store the memory contents of $\mathcal{A}$ and manage contributor accesses to the memory. This means the new leader needs exponentially many states since there are D^V many possible memory valuations. The domain and the contributor of $\mathcal{A}'$ are of polynomial size. In fact, we can then apply the algorithm from Sect. 3.1 to the program $\mathcal{A}'$. The runtime depends exponentially only on the parameters D, L, and V. This shows that LCR_{Var}(D, L, V) is fixed-parameter tractable. It is an interesting question whether this algorithm can be improved. Moreover, it is open whether there are other parameterizations of LCR_{Var} that have an FPT-algorithm. A closer investigation is considered future work.

For BSR, allowing global variables also leads to PSPACE-hardness. The problem BSR_{Var}, defined similarly to LCR_{Var}, is PSPACE-hard already for a constant number of threads. In fact, the proof is similar to the hardness of LCR_{Var} where only one thread is needed. To obtain an algorithm for the problem, we modify the construction from Proposition 20. The resulting product automaton then also maintains the values of the global variables. This shows membership in PSPACE. But the size of the product now also depends exponentially on D and V. The interesting question is whether we can find an algorithm that avoids an exponential dependence on one of the parameters P, t, D or V. It is a matter of future work to examine the precise complexity of the different parameterizations.

Acknowledgements Open Access funding provided by Projekt DEAL.

References

1. Atig, M.F., Bouajjani, A., Kumar, K.N., Saivasan, P.: On bounded reachability analysis of shared memory systems. In: FSTTCS, LIPIcs, vol. 29, pp. 611–623. Schloss Dagstuhl (2014)
2. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. In: TACAS, LNCS, vol. 5505, pp. 107–123. Springer, Berlin (2009)
3. Atig, M.F., Bouajjani, A., Touili, T.: On the reachability analysis of acyclic networks of pushdown systems. In: CONCUR, LNCS, vol. 5201, pp. 356–371. Springer, Berlin (2008)
4. Björklund, A., Kaski, P., Kowalik, L.: Constrained multilinear detection and generalized graph motifs. Algorithmica **74**(2), 947–967 (2016)
5. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hermelin, D.: On problems without polynomial kernels. JCSS **75**(8), 423–434 (2009)
6. Bodlaender, H.L., Jansen, B.M.P., Kratsch, S.: Kernelization lower bounds by cross-composition. SIDAM **28**(1), 277–305 (2014)
7. Calabro, C., Impagliazzo, R., Paturi, R.: The complexity of satisfiability of small depth circuits. In: IWPEC, LNCS, vol. 5917, pp. 75–85. Springer, Berlin (2009)
8. Cantin, J.F., Lipasti, M.H., Smith, J.E.: The complexity of verifying memory coherence. In: SPAA, pp. 254–255. ACM (2003)
9. Chini, P., Kolberg, J., Krebs, A., Meyer, R., Saivasan, P.: On the complexity of bounded context switching. In: ESA, LIPIcs, vol. 87, pp. 27:1–27:15. Schloss Dagstuhl (2017)
10. Chini, P., Meyer, R., Saivasan, P.: Fine-grained complexity of safety verification. In: TACAS, LNCS, vol. 10806, pp. 20–37. Springer, Berlin (2018)
11. Chini, P., Meyer, R., Saivasan, P.: Fine-grained complexity of safety verification. CoRR 2018. arXiv:1802.05559
12. Cygan, M., Dell, H., Lokshtanov, D., Marx, D., Nederlof, J., Okamoto, Y., Paturi, R., Saurabh, S., Wahlström, M.: On problems as hard as CNF-SAT. ACM TALG **12**(3), 41:1–41:24 (2016)
13. Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms. Springer, Berlin (2015)
14. Demri, S., Laroussinie, F., Schnoebelen, P.: A parametric analysis of the state explosion problem in model checking. In: STACS, LNCS, vol. 2285, pp. 620–631. Springer, Berlin (2002)
15. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. Springer, Berlin (2013)
16. Durand-Gasselin, A., Esparza, J., Ganty, P., Majumdar, R.: Model checking parameterized asynchronous shared-memory systems. In: CAV, LNCS, vol. 9206, pp. 67–84. Springer, Berlin (2015)
17. Enea, C., Farzan., A.: On atomicity in presence of non-atomic writes. In: TACAS, LNCS, vol. 9636, pp. 497–514. Springer, Berlin (2016)
18. Esparza, J., Ganty, P., Majumdar, R.: Parameterized verification of asynchronous shared-memory systems. In: CAV, LNCS, vol. 8044, pp. 124–140. Springer, Berlin (2013)
19. Esparza, J., Ganty, P., Poch, T.: Pattern-based verification for multithreaded programs. ACM TOPLAS **36**(3), 9:1–9:29 (2014)
20. Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. In: TACAS, LNCS, vol. 5505, pp. 155–169. Springer, Berlin (2009)
21. Fernau, H., Heggernes, P., Villanger, Y.: A multi-parameter analysis of hard problems on deterministic finite automata. JCSS **81**(4), 747–765 (2015)
22. Fernau, H., Krebs, A.: Problems on finite automata and the exponential time hypothesis. In: CIAA, LNCS, vol. 9705, pp. 89–100. Springer, Berlin (2016)
23. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: ESOP, LNCS, vol. 2305, pp. 262–277. Springer, Berlin (2002)
24. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: SPIN, LNCS, vol. 2648, pp. 213–224. Springer, Berlin (2003)
25. Fomin, F.V., Kratsch, D., Woeginger, G.J.: Exact (exponential) algorithms for the dominating set problem. In: WG, LNCS, vol. 3353, pp. 245–256. Springer, Berlin (2004)

26. Fortin, M., Muscholl, A., Walukiewicz, I.: On parametrized verification of asynchronous, shared-memory pushdown systems. CoRR (2016). arXiv:1606.08707
27. Fortin, M., Muscholl, A., Walukiewicz, I.: Model-checking linear-time properties of parametrized asynchronous shared-memory pushdown systems. In: CAV, LNCS, vol. 10427, pp. 155–175. Springer, Berlin (2017)
28. Fortnow, L., Santhanam, R.: Infeasibility of instance compression and succinct PCPs for NP. JCSS **77**(1), 91–106 (2011)
29. Furbach, F., Meyer, R., Schneider, K., Senftleben, M.: Memory-model-aware testing: A unified complexity analysis. ACM TECS **14**(4), 63:1–63:25 (2015)
30. Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. **26**(4), 1208–1244 (1997)
31. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI, pp. 266–277. ACM (2007)
32. Hague, M.: Parameterised pushdown systems with non-atomic writes. In: FSTTCS, LIPIcs, vol. 13, pp. 457–468. Schloss Dagstuhl (2011)
33. Hague, M., Lin, A.W.: Synchronisation- and reversal-bounded analysis of multithreaded programs with counters. In: CAV, LNCS, vol. 7358, pp. 260–276. Springer, Berlin (2012)
34. Holík, L., Meyer, R., Vojnar, T., Wolff, S.: Effect summaries for thread-modular analysis–sound analysis despite an unsound heuristic. In: SAS, LNCS, vol. 10422, pp. 169–191. Springer, Berlin (2017)
35. Impagliazzo, R., Paturi, R.: On the complexity of k-sat. JCSS **62**(2), 367–375 (2001)
36. Kahlon, V.: Parameterization as abstraction: a tractable approach to the dataflow analysis of concurrent programs. In: LICS, pp. 181–192. IEEE (2008)
37. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, The IBM Research Symposia Series, pp. 85–103. Plenum Press (1972)
38. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: CAV, LNCS, vol. 6174, pp. 629–644. Springer, Berlin (2010)
39. Lokshtanov, D., Marx, D., Saurabh, S.: Slightly superexponential parameterized problems. In: SODA, pp. 760–776. SIAM (2011)
40. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS, LNCS, vol. 3440, pp. 93–107. Springer, Berlin (2005)
41. Wareham, T.: The parameterized complexity of intersection and composition operations on sets of finite-state automata. In: CIAA, LNCS, vol. 2088, pp. 302–310. Springer, Berlin (2000)
42. Yap, C.K.: Some consequences of non-uniform conditions on uniform classes. TCS **26**, 287–300 (1983)