Fine-Grained Complexity of Program Verification

Der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegte Dissertation

von
Peter Chini
geboren am 14.12.1986
in Rodalben

Eingereicht am:	01.09.2021
Disputation am:	-
1. Referent:	Prof. Roland Meyer
2. Referent:	Prof. Petteri Kaski
3. Referent:	Prof. Javier Esparza

2021

You have to put in the hours . . . 10,000 hours, are you crazy? - Petteri Kaski, 2016

Abstract

Tools to automatic program verification have been a central research topic over the last decades. Often disparaged as slow in the past, the steady enhancement of tools paid off in recent years. In fact, tool-based approaches to verification have proven themselves to be rather efficient and applicable. This development has led to a gap between theory and practice. While tools perform well, verification tasks are still deemed *computationally hard* by classical complexity theory. This has two reasons. On the one hand, tools identify and exploit structures that are characteristic for practical instances. On the other hand, worst-case complexity relies on artificial instances that do not occur in practice. In order to bridge the gap, we employ *Parameterized* and more specific *Fine-Grained Complexity* to tackle the latter reason.

The problem with classical complexity theory is that its measurements are solely based on the size of an instance. Structural characteristics and parameters are ignored. Parameterized complexity takes additional parameters of an instance into account and generates a finer picture of the complexity of a problem. It allows for measuring time and space requirements in terms of these parameters. Fine-grained complexity even goes one step further and proves the optimality of obtained algorithms.

We provide an interdisciplinary study connecting program verification and parameterized complexity. To this end, we conducted several fine-grained complexity analyses of typical tasks from the spectrum of program verification. This includes classical safety verification problems like bounded context switching and its variants, safety and liveness verification problems for parameterized systems, as well as consistency testing problems for shared memories. Our main findings are new and provably optimal verification algorithms for each of the considered problems. These reveal the precise time requirements for solving these tasks. Moreover, we provide new hardness results that distinguish between tractable and intractable parameters.

Altogether, our study shows that program verification greatly benefits from the algorithmic techniques and lower bound frameworks provided by parameterized and fine-grained complexity. Our results contribute to understanding the complexity of the considered verification tasks in more detail and ultimately help to identify simple and hard instances.

Zusammenfassung

Tools zur automatisierten Verifikation von Programmen sind schon seit einigen Jahrzehnten zentraler Gegenstand der Forschung. In der Vergangenheit oft als langsam erachtet, hat sich das Bild über Verifikationstools jedoch aus heutiger Sicht gewandelt. Die stetige Weiterentwicklung tool-basierter Ansätze hat dazu geführt, dass sich selbige als sehr effizient und einfach anwendbar erwiesen haben. Dies führte jedoch unweigerlich zu einer Diskrepanz zwischen Theorie und Praxis. Während Verifikationstools verlässlich und effizient Programme auf ihre Korrektheit überprüfen, gelten die zugrundeliegenden Verifikationsaufgaben immer noch als sehr rechenaufwendig. Dies hat zwei Gründe. Einerseits nutzen Tools strukturelle Eigenschaften, die typisch für Instanzen aus der Praxis sind, aus, um das implementierte Verifikationsverfahren zu beschleunigen. Andererseits basiert die Worst-Case-Komplexität von Verifikationsaufgaben auf künstlichen Instanzen, die in der Praxis ohnehin nicht vorkommen. Mit dieser Arbeit leisten wir einen ersten Beitrag zur Aufhebung dieser Diskrepanz. Dazu nutzen wir zwei neue Arten von Komplexitätstheorien, die bei letzterem Grund für obige Diskrepanz ansetzen: nämlich die *Parametrisierte Komplexitätstheorie* und die sogenannte *Feingranulare Komplexitätstheorie*.

Das Problem der klassischen Komplexitätstheorie ist, dass ihre Messungen ausschließlich auf der Eingabegröße einer Instanz beruhen. Strukturelle Besonderheiten und Parameter werden jedoch ignoriert. Parametrisierte Komplexitätstheorie misst Zeit- und Platzbedarf in Abhängigkeit dieser Parameter und bietet daher ein deutlich detaillierteres Bild der Komplexität eines Problems. Mit feingranularer Komplexitätstheorie ist es sogar möglich, zu beweisen, dass ein gefundener Algorithmus für das zugrundeliegende Problem optimal ist bezüglich Zeit- oder Platzverbrauch.

Diese Arbeit umfasst eine interdisziplinäre Studie, die Programmverifikation und parametrisierte Komplexitätstheorie vereint. Gegenstand der Studie sind mehrere feingranulare Komplexitätsanalysen von Problemen, die dem Spektrum der Programmverifikation zugeordnet werden. Dazu gehören klassische Probleme wie Bounded Context Switching und Variationen davon, sowie Safety- und Liveness-Verifikationsprobleme für parametrisierte Systeme und Testing-Probleme für verteilte Speicher. Die wichtigsten Ergebnisse sind neue, optimale, Verifikationsverfahren für jedes der betrachteten Probleme. Diese lösen die betrachteten Verifikationsproblem in der genau dafür benötigten Zeit. Zudem beweisen wir neue Hardness-Resultate, die zwischen effizient nutzbaren und übrigen Parametern unterscheiden.

Unsere Studie zeigt, dass Programmverifikation von algorithmischen Techniken und Frameworks für untere Schranken, die aus der parametrisierten Komplexitätstheorie bekannt sind, profitieren kann. Die Ergebnisse tragen dazu bei, die Komplexität von Verifikationsproblemen genauer zu verstehen und helfen letztlich, einfache und schwierige Instanzen zu unterscheiden.

Preface

Parts of this thesis have already been published within one of our peer-reviewed works [91, 93, 94, 95, 96, 98]. The thesis can be seen as a summary and an extension of these papers. Other peer-reviewed works by us that were not incorporated into this thesis are [89] and [90]. All the cliparts that appear in the following have been taken from `www.nicepng.com`.

Acknowledgments

Foremost, I would like to thank my supervisor Roland Meyer for his guidance during the last years. The initial idea of this interdisciplinary study is due to him and I am grateful for his support, the shared ideas and experience, and of course for countless sessions in front of the board that often lasted for hours. Working with him always encouraged me to tune our results and it made sentences like, *do we already have* k^k, unforgettable.

I would also like to thank Petteri Kaski and Javier Esparza for accepting to review this thesis.

I am further beholden to my friend and companion in the jungle of research, Prakash Saivasan. We have been struggling together through many of the results and we always managed to push our upper and lower bounds to the optimum — even if this caused sleepless nights sometimes.

My parents, Christine and Klaus-Peter, as well as my parents-in-law, Gisela and Richard, also deserve my thanks. Without their steady support, I would not have had the opportunity to begin or complete this thesis.

I would also like to thank my sister Lisa, my brother Sebastian with his family Rahel, Paul, and Martha, as well as many friends and colleagues that I met along the way: Simon Böhm, Sebastian Wolff, Sebastian Muskalla, Timo Seibel, Philipp Schon, Yannick Bender, Pascal Elarbi, Anja Flucke, Christina Gillmann, Hannah, Daniel & Mattis Schmitt, Thorsten & Sonja May, Fadi Almouhanna, Elisabeth Neumann, Johannes Mohr, Emanuele D'Osualdo, Florian Furbach, Rehab Massoud, Mike Becker, Thomas Haas, and Sören van der Wall.

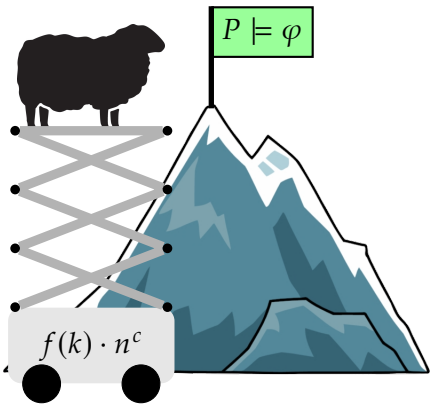
Finally I thank you, Charlotte, for always supporting me and especially for helping me through the last years while facing your own PhD.

Contents

List of Figures

List of Tables

1. Introduction



Software systems appear essentially everywhere in our everyday life. They are not only the backbone of computers and mobile devices but also the foundation of large cloud applications like streaming services and social media. Unfortunately, many software systems suffer from bugs and this is not just subjective perception. In fact, *software developers make 100 to 150 errors for every thousand lines of code on average* [213]. While most software errors do not cause severe damage in general, there are some bugs in software and hardware systems that either negatively affected the life of millions of people or were even lethal. An example is the infamous bug in the *Therac-25* radiation therapy machine [24, 254]. Between 1985 and 1987, a software bug caused several lethal radiation overdoses. A further example is a bug in an energy management system by General Electric which led to the so-called *Northeast blackout* in 2003 [107]. This shortage of electricity affected around 55 million people in the US and Canada and lasted for two days. There are many more examples of software bugs with fatal consequences like these.

What the two mentioned bugs have in common is that they appeared in concurrent software systems. The performance-oriented design and intricate behaviors make these systems hard to program and prone to errors. Finding errors in concurrent systems is a topic within a research field called *Program Verification* [24, 102]. Roughly, program verification aims to show that a given program behaves according to its specification — this includes finding bugs and also proving their absence. The probably simplest form of verification are type systems of programming languages as well as assertions that are placed into the code. Program verification is a significant part of quality assurance and essential to each software development process.

Methods to program verification are being developed since the late 1960s. One of the most prominent methods is to manually construct a *proof* for a given program with so-called *Hoare triples* [164, 210]. While the manual proof construction works well for sequential programs, for concurrent programs one would need to construct a proof for each possible interleaving of threads. Due to the large number of such interleavings, the construction is typically

not feasible. Subsequently, the development of automated verification procedures for concurrent programs began. Such a procedure takes a concurrent program, or a model of it, as its input and reports bugs automatically. Often, various procedures are implemented within a *verification tool*. Such a tool does not only have the advantage of being more efficient than searching a proof by hand, but also simplifies integrating verification into software development. Over the last decades, substantial effort has been devoted to the development of verification tools. Consequently, the applicability steadily increased and has seen great success in recent years [102]. A prime example is the verification tool INFER [75] by Facebook which is currently employed by many leading IT and software development companies.

The current success of tools stands in contrast to the computational complexity of program verification tasks. Measured in classical complexity, these tasks are typically located in the spectrum of classes above NP. This shows a well-known gap between practice and theory which occurs due to two reasons. On the one hand, tools are tuned towards industrial instances. They identify certain structures or parameters within the programs which they can exploit algorithmically. On the other hand, classical complexity theory is stuck with relying solely on the input size as a measure while other parameters are ignored. Consequently, worst-case complexity estimations are typically obtained from made-up instances that do not occur in practice but nevertheless push the hardness of the verification task.

In this thesis, we examine the capability of *Parameterized Complexity* as a theory for explaining and ultimately bridging the gap. Parameterized complexity [112, 141, 167] is a new field within complexity theory that allows for fine-grained complexity judgments. The theory does not only base its measurements on the input size, like classical complexity theory does, but also determines the influence of structural *parameters* on a problem's time and space consumption. We conducted several parameterized complexity analyses of typical verification tasks, providing an interdisciplinary study that connects parameterized complexity and program verification.

Our results show that parameterized complexity is capable of bringing theory and practice of program verification closer together. We provide *new verification algorithms* which are based on the rich source of algorithmic techniques known from parameterized complexity. Many of our provided algorithms are *provably optimal*. Parameterized complexity comes with a framework for proving lower bounds. We adapted the framework to program verification and were able to prove that algorithms faster than ours are highly unlikely to exist. We furthermore reveal *new hardness results* that serve two purposes. First, they determine parameters of the considered verification tasks that are not sufficient to guarantee an efficient algorithm. Moreover, they classify these parameters along the *level of intractability* that they cause. Altogether, we obtain a clear picture of the parameterized complexity for each of the verification tasks we considered in this thesis. Our

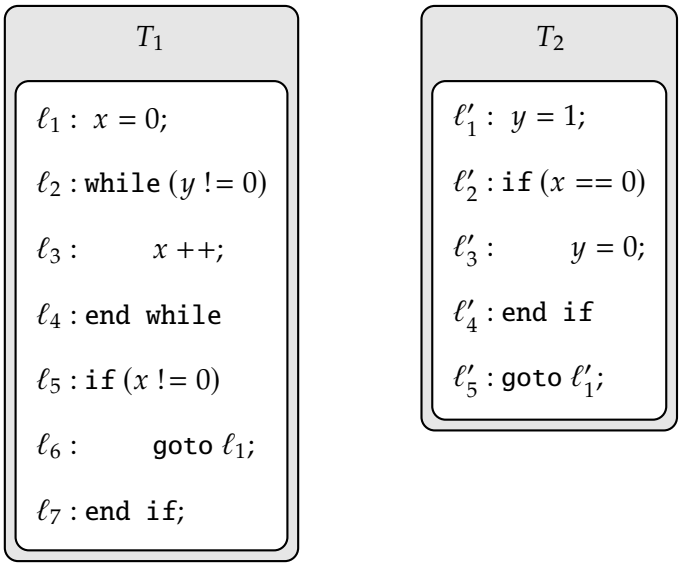


Figure 1.1: A simple program with two concurrent threads T_1, T_2 that act on integer variables x, y. Each instruction is decorated by a label.

results separate tractable from intractable parameters and ultimately help to distinguish between hard and simple instances.

Before we give an overview of our contribution, we first elaborate on the two main ingredients of our interdisciplinary study, namely program verification and parameterized complexity, in more detail.

1.1 Program Verification

When designing a program, there is always some sort of intended behavior that we associate with it. For instance, it should not get stuck in a deadlock, it should not reach a certain point, or it should eventually give a response after receiving an input. *Program Verification* [24, 102] is the task of proving that a given program follows its intended behavior. If it does, we consider the given program *correct*. Otherwise, the program contains a bug.

For proving correctness of a program, we need a description of what its intended behavior should be. This is captured in the program's *specification*. Consequently, a program is correct if it satisfies its specification and a bug occurs if it shows behavior outside its specification. Then, program verification can be understood as the following decision problem: given a program P and a specification φ, decide whether P satisfies φ, written

$$P \models \varphi.$$

Consider the program P in Figure 1.1. It consists of two concurrently running threads, T_1 and T_2, that manipulate the integer variables x and y. Each instruction has a label to identify its location in the code. A specification φ could postulate that the threads cannot simultaneously reach ℓ_6 and ℓ'_5. We show that P does not satisfy φ — meaning that P contains a bug.

It can quickly be seen that there is a computation of P which brings both threads to the locations prohibited by φ. In fact, T_1 may start by setting x to 0, followed by T_2 which performs ℓ'_1 and ℓ'_2. It does not proceed with setting y to 0. Instead, the computation is interfered by T_1 which jumps into its `while`-loop and increases x at least once. Then T_2 executes ℓ'_3 and ℓ'_4 and goes to ℓ'_5. Note that this breaks the loop of thread T_1. Now x is different from 0 and T_1 can reach ℓ_6. Altogether, P does not satisfy its specification φ.

While in the example disproving correctness was simple, the general task of (dis)proving that a given program satisfies its specification requires more elaborate methods. One of the earliest and still prominent methods goes back to Floyd and Hoare [164, 210]. They provided a logic that allows for the manual construction of correctness proofs along the statements of a program. While this method works well for sequential programs, constructing a proof by hand can be quite *tedious* [100] when concurrency is involved. In fact, the sheer number of interleavings of threads in a concurrent program can make finding such a proof an infeasible task. Clarke and Emerson [100] had the idea to replace the manual construction by an automated process, a verification tool that takes a program P and a specification φ and automatically checks whether $P \models \varphi$. Such a tool is not only faster than finding a proof by hand but it can also be integrated into software development.

Unfortunately, the computational requirements of automatically proving correctness are high and an immediate algorithm is out of reach. In fact, since most programming languages are able to simulate general Turing machines, testing whether an arbitrary program satisfies a specification is typically an undecidable problem [316]. To overcome the undecidability, it is common to approximate the program's behavior. In this thesis, we consider two corresponding approaches: *Model Checking* and *Testing*.

1.1.1 Model Checking

The idea behind *model checking* is to capture a program's semantics by a precise mathematical model and to formalize the specification. Then, instead of working with the actual program and the specification directly, it is rather checked whether the model satisfies the formalization. The idea goes back to the 1980's work of Clarke and Emerson [100] and Queille and Sifakis [294].

Typically, in model checking programs are captured by an automaton model [212] while the specification is described in terms of a logic [148] like LTL [290, 332] or CTL [36, 100]. Since the 1980s, model checking has evolved to one of the major verification techniques [24, 101, 102]. Besides

its immanent mathematical elegance, the popularity of model checking is due to its steadily increasing applicability. The latter relies on striking ideas like *Symbolic Model Checking* [67, 274], *Partial Order Reduction* [184, 289, 318], and *Bounded Model Checking* [41, 42] that pushed the applicability of model checking towards industrial-scaled systems [101]. In 2007, Clarke, Emerson, and Sifakis received the Turing award for their pioneering work.

In this thesis, we consider tasks of two different types that are common in program verification and model checking. We will explain them in more detail below. The basis of our study are concurrent programs.

Safety Verification We speak of *safety verification* if the specification that our program has to satisfy encodes a so-called *safety property*. Intuitively, such a property states that *nothing bad should happen* [232, 249]. For instance, a deadlock or the simultaneous arrival of two threads in a critical section [24]. The term *safety property* goes back to Lamport [249]. Following his work, there were several attempts to mathematically capture what a safety property actually is. Consequently, there are various definitions [10, 15, 120, 255] with ranging expressiveness and some of them coincide under certain assumptions [14].

What the definitions have in common is that a finite execution of the program suffices to detect whether the safety property is violated. This means, a safety property partitions the states of a program into *unsafe* and *safe* states. Hence, proving that such a property holds amounts to showing that no unsafe state is reachable. It fails if we can show that an unsafe state can be reached. The latter formulation allows for capturing safety verification algorithmically — as a reachability problem. The safety verification problems in this thesis are all formulated in terms of reachability.

As an example, reconsider the program P from Figure 1.1 along with its specification φ. It encodes that the threads T_1 and T_2 cannot reach the locations ℓ_6 and ℓ'_5 simultaneously. This is clearly a safety property — we may think of T_1 and T_2 both arriving at a critical section which should be exclusive to one thread. As we have seen, the property does not hold since the unsafe state is indeed reachable. Consequently, P is unsafe.

Liveness Verification If the specification of our program encodes a so-called *liveness property*, we speak of *liveness verification*. Unlike safety properties, liveness properties do not avoid the occurrence of a *bad* event. Instead, they express that *something good eventually happens* [232, 249]. Typical examples are that each thread of a program eventually enters its critical section or that it visits its critical section infinitely often [24]. *Liveness properties* were first mentioned by Lamport [249] but they were heavily influenced by Pnueli's work [290] who realized that program verification questions at that point mainly targeted programs with a clear beginning and end. Programs that need to satisfy a liveness property typically do not terminate.

We follow the approach to liveness verification by Vardi and Wolper [320]. The authors reduced checking for liveness properties to an automata theoretical problem. To this end, Vardi and Wolper suggested to describe the property that needs to be checked in terms of a Büchi automaton [145], a finite-state automata over infinite words. The approach covers properties defined in LTL [320] and MSO [70, 71] and furthermore allows for capturing liveness verification algorithmically — in terms of a repeated reachability problem. In fact, let A_φ be the Büchi automaton of a liveness property φ. A program P satisfies φ if each infinite execution of P is accepted by A_φ. Phrased differently, φ is not satisfied if $A_{\neg\varphi}$ and P share an infinite execution. The latter is a repeated reachability problem on the system $A_{\neg\varphi} \times P$. Since the problems that we consider in this thesis typically rely on finite-state models, our liveness verification tasks are all stated in terms of repeated reachability.

We consider an example of a liveness property. Suppose, we want to check whether the program P given in Figure 1.1 can reach the pair (ℓ_6, ℓ'_5) of labels infinitely often. Let the specification ψ capture the requirement. We have already seen earlier that P can reach the pair via some execution ρ. Since ℓ_6 and ℓ'_5 reset the corresponding thread to the beginning, the program can just repeat ρ and again arrive at (ℓ_6, ℓ'_5). This means we can perform the execution infinitely often. Hence, we found an infinite execution, namely ρ^ω, that satisfies the property ψ. However, not each infinite execution of P satisfies ψ. Note that for instance, thread T_1 can cycle forever in its loop while T_2 gets stuck at ℓ'_2. Since liveness properties need to be satisfied by all infinite executions of a program, we consequently have $P \not\models \psi$.

1.1.2 Testing

Software Testing [32, 102, 282] is an under-approximate method that runs the software or the program at hand on a particular input, a so-called *test*. The idea is to execute the program along a certain path. The actual outcome of this execution path is then compared to the anticipated outcome described by the specification. If the two outcomes do not match, we can report a bug. Otherwise, we can deduce that the tested execution is safe.

Testing is one of the predominant techniques for finding errors in software. In general, it accounts for around 50% of the total software development costs [24, 282]. The popularity of testing is due to the fact that integration into the software development process is comparatively simple and, unlike model checking, testing does not require a mathematical model of the system or program at hand — it only requires a (compiled) piece of code. Testing can hence be employed whenever system models cannot be derived. For instance if we do not have access to a piece of code, or if the software or hardware at hand is too intricate to being reflected by a *simple* model. The latter is typically the case in the context of shared-memory implementations [183] which is where we will consider testing problems in this thesis.

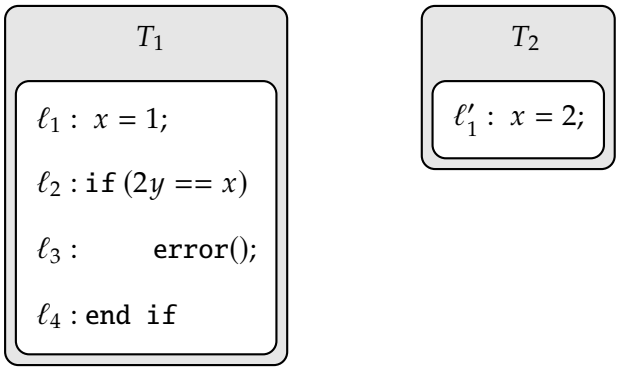


Figure 1.2: Program with two threads T_1, T_2 over integer variables x and y. The value of y is given as an input and is therefore unknown.

Despite its success, testing has its limitations. While the technique is capable of detecting bugs, it cannot prove their absence. In fact, one would need to test all inputs to a program in order to reason about all possible executions, something that is not feasible in practice. But there have been efforts to increase the expressiveness of testing. One of the most important approaches is *automatic test case generation* [282], a technique that autonomously generates test cases which should trigger *unsafe* executions of the program. Test case generation is typically divided into *static* [39, 109, 233, 321] and *dynamic* [72, 185, 186, 187, 188, 189, 238] generation. Roughly, the former technique derives test cases by inspecting the program code, the latter first runs the program and then generates test cases based on the outcome.

We consider a simple example. The program P given in Figure 1.2 consists of two threads T_1 and T_2 that act on the integer variables x and y. Assume that the value of y is not determined by the program but is given as an input, for instance through interaction with a user. The function `error()` called in thread T_1 represents an error state that the thread should not be able to reach.

We determine whether T_1 can actually reach the label ℓ_3 and throw the error. To this end, consider a test case that fixes the input $y = 1$ and lets thread T_2 run before T_1. In the resulting execution $\ell_1'.\ell_1.\ell_2.\ell_4$, the `if`-statement in T_1 fails since we have $2y = 2$ and $x = 1$. Hence, T_1 does not reach ℓ_3 with this particular execution. If we keep the value of y but change the scheduling of the threads so that T_2 interferes the computation of T_1 between ℓ_1 and ℓ_2, we obtain the corresponding execution $\ell_1.\ell_1'.\ell_2.\ell_3$. Note that this time, the `if`-statement in T_1 succeeds since $2y = 2 = x$ and consequently, the execution reaches `error()`. The test case therefore exposes that T_1 can indeed reach the error state and that the program P contains a bug.

1.2 Parameterized Complexity

In fields like program verification, where the goal is to solve computationally intricate problems, often a gap between theory and practice occurs. Although problems have a high worst-case complexity, tools may still solve them quite efficiently. Algorithms often exploit structures or parameters of a problem which results in a speed-up. Classical complexity theory is not precise enough to capture such parameters. The theory bases its measurements only on the input size and groups problems into classes like NP that do not distinguish between its members properly. The best one can hope for are results that assume certain parameters of the problem to be constant [225, 264] in order to obtain a polynomial-time algorithm. However, if the running time of the algorithm is n^k where n is the input size and k is the parameter, this still does not explain that the problem can be solved efficiently. Although for fixed k such an algorithm runs in polynomial time, the degree of the polynomial grows with increasing values of the parameter k.

Parameterized Complexity [112, 141, 167] is a new field within complexity theory that is capable of closing the above gap. It measures the complexity of a problem in a way that overcomes the weakness of being bound to the input size only. Instead, parameterized complexity measures the influence of parameters on a problem's complexity. These parameters may either describe the shape of the input or the structure of the desired solution in more detail. Including them in the measurement provides a much finer picture of the complexity and helps to distinguish simpler from harder instances.

The early development of parameterized complexity is mainly attributed to Downey and Fellows [5, 135, 136, 157, 158]. Since its discovery, corresponding *parameterized complexity analyses* have been performed for all kinds of algorithmic problems from various fields within computer science and mathematics [112, 167, 141]. Nowadays, parameterized complexity has evolved to an integral part of complexity theory which provides many techniques that have become standard. Subsequently, we give a short overview following three fields within parameterized complexity: *Fixed-Parameter Tractability*, a corresponding *Hardness Theory*, and *Fine-Grained Complexity*.

1.2.1 Fixed-Parameter Tractability

One of the most important tasks of parameterized complexity is to examine whether a given problem is *fixed-parameter tractable* (FPT). The notion formalizes when we consider parameters of a problems influential enough to provide an efficient algorithm. Fixed-parameter tractability requires an algorithm that runs in time exponential, or worse, only in the considered parameters and polynomially in the input size. Formally,

$$f(k) \cdot n^d$$

where f is a function depending only on the considered parameters k, n is the size of the input, and $d \in \mathbb{N}$ is a constant. This restricts the *expensive part* of the algorithm to the parameters k. If these can assumed to be small in practical instances, the algorithm has almost polynomial running time.

The complexity class of all problems that are fixed-parameter tractable is denoted by FPT as well. The running time of the corresponding algorithms is typically abbreviated by $O^*(f(k))$ to emphasize the dominant part. Note that for a fixed parameter k, this yields a polynomial-time algorithm running in time n^d with a constant d independent from k. This stands in contrast to problems that can only be solved in time n^k. If k is constant in the latter case, the resulting algorithm runs in polynomial time as well but the degree of the polynomial depends on the actual value of k.

Since the formulation of fixed-parameter tractability [5, 136, 157, 158] many famous decision problems were shown to be FPT. However, finding an algorithm with the required running time can be tricky. As a consequence, the development of FPT-*techniques* [112, 170] began. These are algorithmic frameworks and techniques that have proven useful in developing FPT-algorithms. Some of these techniques are well-known classics, for instance *dynamic programming* [33, 106], others were developed only recently [113] and take advantage of algebra [48, 240] or probability theory [13]. We will employ these FPT-techniques to solve verification tasks. The particular techniques that we rely on will be introduced later.

1.2.2 Hardness Theory

While many classical decision problems were shown to be fixed-parameter tractable, one problem remained unrelenting. Namely finding a clique of size k in a given graph, denoted by Clique. Despite extensive effort, no algorithm for Clique running in time $f(k) \cdot n^d$ was found. The parameter k did not seem powerful enough to tame the computational hardness and to guarantee membership in FPT. This raised the question of whether Clique is FPT at all and in fact, we can say that it is highly unlikely.

To answer questions like the one above, parameterized complexity offers a theory of *relative hardness*. The idea of this theory is to separate tractable from *intractable* problems like it is achieved in classical complexity theory via P and NP — by relying on the hardness of certain decision problems. But instead of having only a single class comprising the intractable problems, Downey and Fellows [135, 136, 137, 138, 139] have shown that, in the parameterized world, there is a whole hierarchy of *intractability classes*. The latter is called W-*hierarchy*. It consists of infinitely many increasing levels, one for each degree of intractability. The t-th level is denoted by W[t] and the first level W[1] contains the class FPT. Consequently, the hierarchy can be drawn as follows:

$$\mathsf{FPT} \subseteq \mathsf{W}[1] \subseteq \mathsf{W}[2] \subseteq \ldots$$

Like for P and NP, it is widely believed that FPT and W[1] do not coincide [112]. Hence, a problem hard for W[1] is generally considered unlikely to be fixed-parameter tractable. For the problem Clique, this is the case. But for a corresponding statement, we need to be more precise and explicitly mention the underlying parameter: Clique(k) is W[1]-complete. Note that it could be the case that for a parameter different from k, the problem is FPT.

Like in classical complexity, the relative hardness theory in parameterized complexity relies on a suitable notion of reduction, so-called *parameterized reductions*. Roughly, these are reductions that keep the parameter small but allow to take FPT-time. As expected, FPT and the intractability classes W[t] are robust under this kind of reduction [112, 141, 167]. This means we can reason like in classical complexity theory. When we suspect that a problem at hand might be intractable, we can try to reduce from Clique(k). Once this is successful, we obtain that the problem is W[1]-hard.

1.2.3 Fine-Grained Complexity

From a theoretical point of view, proving a problem fixed-parameter tractable might be sufficient for considering it as efficiently solvable. However, the definition of FPT still allows for large functions $f(k)$. In practice it can make a huge difference whether $f(k)$ is of the form k^k or 2^k. *Fine-Grained Complexity* is the study of the precise function $f(k)$ that is needed to solve a problem.

Obtaining upper bounds on the function $f(k)$ is achieved by applying the aforementioned FPT-techniques. There are many examples [113] where the right technique improved from a *slightly superexponential* dependence k^k to a *single exponential* one of the form c^k, where c is a constant. But sometimes, none of these techniques seems to be able to improve the running time. For these cases, fine-grained complexity offers a framework for proving relative lower bounds on $f(k)$. The idea is similar to relative hardness as explained above but different from intractability theory, fine-grained complexity does not provide complexity classes for which we can prove a problem complete. In order to provide relative lower bounds, fine-grained complexity relies on various hardness assumptions. The three best-known are the so-called *Exponential Time Hypothesis* (ETH), the *Strong Exponential Time Hypothesis* (SETH), and the newer *Set Cover Conjecture* (SCON) [110, 217, 218]. With the framework for relative lower bounds, it is then possible to find the provably *optimal* $f(k)$ that is required to solve the problem at hand.

The ETH is the assumption that 3-SAT cannot be solved in time $2^{o(n)}$ where n is the number of variables. Despite the algorithmic endeavor for 3-SAT, no algorithm has achieved the time $2^{o(n)}$ until now and it is believed that this is not possible. The exponential time hypothesis can be employed to derive lower bounds for $f(k)$ by reducing from 3-SAT to the problem of choice with a so-called *linear* reduction. A linear reduction ensures that the parameter k depends only linearly on the number of variables n. Hence,

not each reduction from 3-SAT known from NP-completeness automatically translates into a linear reduction. In fact, the latter are typically much harder to find. The SETH roughly assumes that the general problem SAT cannot be solved in time $O^*((2-\varepsilon)^n)$ for an $\varepsilon > 0$. The SCON assumes a similar lower bound for the problem Set Cover. All three assumptions are standard for obtaining lower bounds in parameterized complexity [110, 112].

A further goal of fine-grained complexity is to provide lower bounds for so-called *kernelizations*. Roughly, a *kernelization* is a polynomial-time preprocessing that takes an instance of a problem and reduces it to its hard part — its *kernel*. Such preprocessing techniques are typically applied when facing large instances like in SMT or optimization solvers [118, 215]. A kernelization is considered efficient if it can reduce a given instance to a kernel of size at most polynomial. Lower bounds can prove the absence of such polynomial kernels. This implies that preprocessing techniques have only limited success and cannot significantly simplify the particular problem at hand. We prove several kernel lower bounds for verification tasks.

1.3 Contribution

This thesis takes a first step towards closing the aforementioned gap between theory and practice in automatic program verification. We apply techniques from parameterized complexity as well as program verification to achieve the following two goals:

- *find efficient algorithms for verification tasks that are provably optimal in the fine-grained sense and that are easy to implement,*
- *find out what causes the computational hardness of verification problems. Provide a detailed picture of tractable and intractable parameters.*

Our observation is that program verification benefits from the above study. On the one hand, parameterized complexity offers algorithmic techniques that have not yet been employed to verify programs. Combining the techniques with lower bounds based on ETH, SETH, and SCON yields provably optimal verification algorithms. On the other hand, from the fine-grained hardness theory, we gain insights into the computational hardness of verification tasks that help to distinguish simple from hard instances.

We elaborate on our contribution in more detail. We consider safety verification, liveness verification, and testing along four different kinds of models, ranging from shared-memory concurrent programs to parameterized systems and weak memory. We do not list the complete contribution here but restrict to the most important results. For a complete overview, we refer to the introductions within the corresponding chapters 6, 7, 8, and 9.

1.3.1 Bounded Context Switching

Bounded Context Switching (BCS) [292, 293] is one of the most prominent under-approximate methods for verifying shared-memory concurrent programs. Since proving safety in this context is typically PSPACE-hard [241] or worse, under-approximating the behavior of the underlying program is common to reduce the complexity and find bugs more efficiently. BCS reduces the complexity to NP by limiting the number of times the threads can switch the processor — the number of so-called *context switches*. The method has proven to be reliable and experiments have shown that a majority of subtle concurrency bugs show up in few context switches [280].

We performed a fine-grained complexity analysis of BCS. The first question to solve is the choice of parameters. Since the number cs of context switches is typically small in practical examples [280], we initially considered BCS(cs), the parameterization of BCS in cs. Surprisingly, it turned out that parameterizing in cs alone is not sufficient to obtain a tractable algorithm. Instead, BCS(cs) is W[1]-complete. This means that introducing cs lets the complexity drop from PSPACE to NP but from a fine-grained point of view it is not sufficient to guarantee an FPT-algorithm.

For an FPT-algorithm, we need a second parameter. We chose the size m of the memory. Often, shared-memory communication is via setting flags which keeps memory requirements moderate [263]. With the new parameter, our main finding is that BCS can be solved in time

$$O^*(m^{cs} \cdot 2^{cs}).$$

The algorithm is a two step procedure, first reducing BCS to the simpler problem Shuffle Mem and then solving the latter. This requires an application of an algorithmic technique called *fast subset convolution* [48]. To the best of our knowledge, this is the first time the technique appears in the context of program verification. The algorithm shows that BCS(cs, m) is FPT. We complement the upper bound with an ETH-based lower bound of the form $m^{o(cs/\log(cs))}$ showing that our algorithm is almost optimal. As a part of our complexity analysis, we also contribute upper and lower bounds as well as optimal algorithms for Shuffle Mem and for variants of bounded context switching, like *Round Robin* [61, 248, 280] and *Bounded Stage Reachability* [20].

1.3.2 Liveness in Broadcast Networks

Broadcast networks [147] are a model for parameterized systems that consist of an arbitrary number of identical finite-state *clients* communicating via passing messages to each other. The precise number of involved clients is only fixed once a computation starts. Consequently, proving correctness for broadcast networks amounts to proving it for any number of clients.

Safety verification of broadcast networks is well-understood. Problems in this context are typically solvable in polynomial time [123, 124, 176]. We

examine the complexity of the *broadcast network liveness verification problem* (BNL) — the task of deciding whether one client can visit a final state repeatedly. The complexity of the problem has been left open, with an EXPSPACE upper bound [124] and a P-hardness lower bound [123]. To our surprise, we were able to show that BNL admits a polynomial-time algorithm.

Our algorithm solves BNL in time $O(p^4 \cdot t^2)$, where p is the number of client states and t the size of its transition relation. The algorithm is a fixed-point iteration that relies on a graph abstraction of broadcast networks which allows for switching from the infinite state space to a finite one. Although the algorithm is not a classical FPT-result, we have only obtained it while we were conducting a fine-grained complexity analysis following the two goals above. Moreover, our algorithm reveals new FPT-upper bounds for the more general *LTL model-checking problem* of broadcast networks [97].

We further investigate the complexity of the *fair liveness verification problem* (FBNL) where not only one, but all clients need to visit a final state repeatedly. We prove that FBNL can be solved in polynomial time as well. Altogether, our results show that safety and liveness verification tasks admit the same time complexity — up to a polynomial factor.

1.3.3 Safety and Liveness in Leader Contributor Systems

A *leader contributor system* is an appropriate model for parameterized systems with master/slave architecture. A leader contributor system consists of a particular *leader thread* and an arbitrary number of identical *contributor threads*. Communication is via accessing a single shared memory cell. We assume all threads in the system to be finite-state.

The safety verification problem in this context is the *leader contributor reachability problem* (LCR) [152, 198]. It asks whether the leader can reach an unsafe state with the help of a certain number of contributors. The problem is NP-complete [152]. We examine its fine-grained complexity by considering two parameterizations in more detail: LCR(d, l) and LCR(c). The former is the parameterization by the size d of the underlying data domain and the number l of leader states, the latter is the parameterization by the number c of contributor states. We show that other parameterizations are intractable.

For the former choice of parameters, d and l, we provide an algorithm that solves the safety verification problem LCR in time

$$(d + l)^{O(d+l)}.$$

Roughly, the algorithm is a dynamic programming on so-called *witnesses*. These are compressed executions of the system that can witness the reachability of an unsafe state. We also provide an ETH-based matching lower bound of the form $2^{o((d+l)\cdot\log(d+l))}$. It shows that our algorithm is optimal.

For the problem LCR(c), we provide an algorithm running in time

$$O^*(2^c).$$

The algorithm relies on a characterization of computations in terms of paths in a *saturation graph*. The graph represents arbitrarily many contributors by only a single set of states which gets saturated when simulating a computation. On the saturation graph, we then employ a dynamic programming to test the existence of a suitable path. The upper bound is matched by a lower bound of the form $O^*((2-\varepsilon)^c)$ based on the SCON.

Besides safety, we also consider liveness verification in leader contributor systems. The *leader contributor liveness problem* (LCL) asks whether the leader can reach a final state repeatedly [143]. Like LCR, the problem is NP-complete. To determine the fine-grained complexity of LCL, we decompose the problem into LCR and a cycle finding problem called CYC. We provide a fixed-point algorithm which solves CYC in polynomial time. This implies that the upper bounds from LCR carry over to LCL. The optimality of the derived algorithms is guaranteed since lower bounds for LCR naturally carry over to liveness. Hence, also in leader contributor systems, liveness and safety verification have same time complexity up to polynomial factor.

1.3.4 Memory Consistency

In processor verification, a major task is to test whether a shared-memory implementation provides the promised consistency guarantees. *Consistency algorithms* check this. They take a test case and decide whether it can be executed under a particular *memory model* like *Sequential Consistency* (SC) [251] or *Total Store Ordering* (TSO) [305, 306] that captures the promised guarantees. Since for most memory models, checking consistency is NP-complete [180], finding efficient consistency algorithms highly benefits from the fine-grained perspective that parameterized complexity offers.

We provide a framework for (provably optimal) consistency algorithms. It can be instantiated by various memory models and each consistency algorithm obtained from the framework automatically runs in time

$$O^*(2^k),$$

where k is the number of write accesses in the given test case. For several other parameters, testing consistency is intractable [180, 183, 270].

Our framework is based on on a *universal consistency problem* MM-Cons that takes a memory model MM and a test case and asks whether the test case can be executed under MM. We show that solving MM-Cons amounts to finding a total order that linearizes the write accesses in the given test case. The latter can be decided by applying a dynamic programming on so-called *snapshot orders*, total orders on particular subsets of write accesses.

We apply our framework to the memory models SC, TSO, PSO, and RMO and obtain an $O^*(2^k)$-time consistency algorithm for each of the models. Moreover, for the former three models, the algorithms are provably optimal. We provide an ETH-based lower bound showing that consistency under SC, TSO, and PSO cannot be checked in time $2^{o(k)}$.

1.4 Outlook

The thesis is structured into three parts and an appendix. In Part I, we give an introduction to parameterized and fine-grained complexity. The part begins with Chapter 2, where a first example and the basic notion of a *parameterized problem* lay the foundations. In Chapter 3 we consider fixed-parameter tractable problems and related algorithmic techniques that are essential to our algorithms. The aforementioned hardness theory within parameterized complexity is the matter of Chapter 4. Finally, Chapter 5 summarizes lower-bound techniques common in fine-grained analyses.

Part II of the thesis elaborates on our contribution. It begins with Chapter 6 which focuses on *Bounded Context Switching* and its variants *Round Robin* and *Bounded Stage Reachability*. In Chapter 7, we present our algorithms for the liveness verification problems around broadcast networks. Subsequently, in Chapter 8, we consider safety and liveness verification of leader contributor systems. Our framework for consistency algorithms is given in Chapter 9.

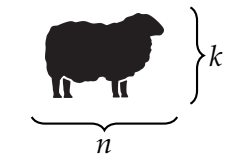
The results obtained in this thesis are discussed in Part III. This includes a comparison with additional related work that has not been considered in earlier chapters. It is given in Chapter 10. Chapter 11 lists some directions and ideas for future research. Chapter 12 concludes the thesis.

Part I

Parameterized and Fine-Grained Complexity

2. The Importance of Parameters

Measuring the complexity of a decision problem is typically achieved by grouping the problem into one of several *robust complexity classes*. The most famous among these is probably the nondeterministic class NP. A problem that is known to be NP-hard is commonly believed to be *unlikely to admit a deterministic polynomial-time algorithm*. Instead, it is assumed that a deterministic algorithm requires exponential running time to solve the problem at hand.



While NP-hardness denies a deterministic polynomial running time, it leaves several questions unanswered that become important when designing algorithms for practical purpose. For instance, we do not obtain a statement on the precise shape of an algorithm's running time. It could be of the form n^n, 2^n or $2^{\sqrt{n}}$ where n is the input size. From a performance point of view, knowing the precise exponential dependence is essential.

Moreover, classical NP-hardness proofs do not take into account *parameters* of the problem. The proofs solely rely on the input size, a measure that is quite rough. A *parameter* is a finer measure describing the structure of an instance in more detail. Compared to the input size, a parameter is usually much smaller. However, it may still have significant impact on the complexity. The problem may be solvable in time $2^k \cdot p(n)$ where k is the parameter and $p(n)$ is only a polynomial in the input size n. This results in much better performance. On the other hand, a parameter may already be enough to cause hardness, resulting in intractability whenever the parameter is large.

Parameterized Complexity measures the complexity in terms of parameters, it resolves questions of the latter type. *Fine-Grained Complexity* answers questions on the precise shape of an algorithm's running time. Together, the theories clarify *how far away* we are from a polynomial-time algorithm. In Section 2.1, we give an introductory example into the theories. In Section 2.2 we define *parameterized problems*, the foundation of parameterized complexity, and we consider two important parameterized problems that are needed throughout the thesis. In the following, we assume familiarity with classical complexity theory. For standard literature, we refer to [18, 190, 303, 324].

2.1 Introductory Example

Parameterized Complexity is a theory for measuring the space and time requirements of a problem in finer detail than classical complexity theory does.

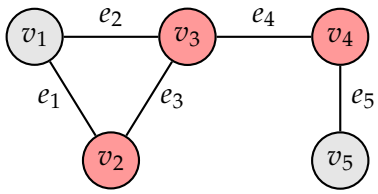


Figure 2.1: A graph $G = (V, E)$ with vertices $V = \{v_1, \ldots, v_5\}$ and edges $E = \{e_1, \ldots, e_5\}$. The red marked vertices form a vertex cover $C = \{v_2, v_3, v_4\}$ of size 3. Note that a vertex cover of smaller size does not exist for the graph.

Instead of only considering the size of the input as a basis for the measurement, parameterized complexity is capable of showing the precise influence of several parameters on the problem's complexity. The theory evolved in the late 1980s and 1990s mainly driven by Downey and Fellows in cooperation with other authors [5, 135, 136, 140, 157, 158]. Since its discovery, parameterized complexity has evolved to a major field within complexity theory. The new field has led to the development of novel algorithmic techniques, provided new insights about classical decision problems, and found many applications in various fields of computer science as well as mathematics.

We give an example which shows the importance of measuring the influence of parameters on a problem's complexity. The problem Vertex Cover is one of the most studied problems in parameterized complexity. Listed in Karp's 21 NP-complete problems in 1972 [229], Vertex Cover has experienced a hunt for the most efficient algorithm since the discovery of parameterized complexity [87, 140]. Formally, the problem asks for a *vertex cover* of a certain size. A *vertex cover* C of an undirected graph $G = (V, E)$ is a subset $C \subseteq V$ of vertices such that each edge in E has one of its endpoints in C. An illustration is given in Figure 2.1. The problem Vertex Cover takes a graph G and an integer k and asks whether G has a vertex cover of size at most k.

Vertex Cover

Input: A graph G and an integer $k \in \mathbb{N}$.

Question: Is there a vertex cover of size at most k?

Consider the graph in Figure 2.1. With the parameter $k = 3$, it forms a *yes*-instance of Vertex Cover. A corresponding cover of size 3 is shown. Note that there is no cover of size 2 or less, resulting in *no*-instances if $k \leq 2$. The latter can be verified by testing each subset of vertices of size at most 2 for being a vertex cover. For all these subsets there is at least one edge which has both endpoints outside the set. Hence, no vertex cover of size at most 2 exists.

To obtain an algorithm for Vertex Cover, we can follow the same idea.

Given a graph $G = (V, E)$ and an integer k, the existence of a vertex cover of size at most k can be decided by iterating over each subset $C \subseteq V$ of size at most k and by testing whether each edge is *covered*. The latter means that each edge has an endpoint in C. As soon as we find such a set, we can safely reply *yes*. If the iteration fails to find a suitable set, we return *no*.

Let us measure the time complexity of the algorithm. A naive implementation runs in time $O(n^k \cdot m)$, where $n = |V|$ is the number of vertices and $m = |E|$ is the number of edges. Already for moderately-sized instances, running the algorithm is hardly feasible. The reason for this inefficiency is that the parameter k appears in the exponent of n. Even if we assume k to be a constant, say $k = 8$, the algorithm runs in polynomial time but the polynomial being $n^8 \cdot m$. Hence, the degree of the polynomial depends on k, resulting in practically useless polynomial-time algorithms for larger values of k.

From an efficiency point of view, it is essential to avoid such a dependence. Phrased differently, we need to find an algorithm for Vertex Cover running in time $O(f(k) \cdot p(n, m))$, where f is a function only depending on k and p is a fixed polynomial depending on n and m. If we assume k to be constant in this setting, we obtain an algorithm running in polynomial time $O(p(n, m))$. Different from the algorithm above, the algorithm at hand has a fixed degree for n and hence, might have the potential for being efficient. Finding an algorithm of the described form is a main task of parameterized complexity.

While the factor $f(k)$ vanishes in theoretical considerations like the one above, it typically has a great impact on the running time of the algorithm. Note that f cannot be a polynomial as this would contradict the NP-hardness of the problem Vertex Cover. Therefore, it is *at least exponential*. Hence, when looking for an algorithm running in time $O(f(k) \cdot p(n, m))$, one should keep the function f as small as possible. But even if f is deemed to be exponential, there are still tractable functions like k^k, 2^k or even $2^{\sqrt{k}}$ that one could try to obtain. Finding the precise function f that is required to solve the problem at hand is referred to as fine-grained complexity.

It is known how to construct an algorithm for Vertex Cover that satisfies the above conditions [112, 275]. It runs in time $O(f(k) \cdot p(n, m))$ and keeps the function f small. Let $(G = (V, E), k)$ be an instance of Vertex Cover. By definition, in a vertex cover C of G, each edge $e \in E$ has one of its endpoints in C. So if we consider an edge $e = \{v, w\}$ with $v, w \in V$, one of the vertices v or w, must be in C. This allows for the design of a *branching algorithm*. It constructs a vertex cover C step by step, starting from the empty set. When it encounters an edge $e = \{v, w\}$, it will have two branches: one for adding v to C and one for adding w to C. Consider the first branch. Then, our potential vertex cover is $C = \{v\}$. This already covers all edges where v is an endpoint. Hence, it remains to find at most $k - 1$ vertices that cover those edges not containing v. If there is no such edge, C is already a proper cover. Otherwise, we pick such an edge and start the branching again. If the choice of v does

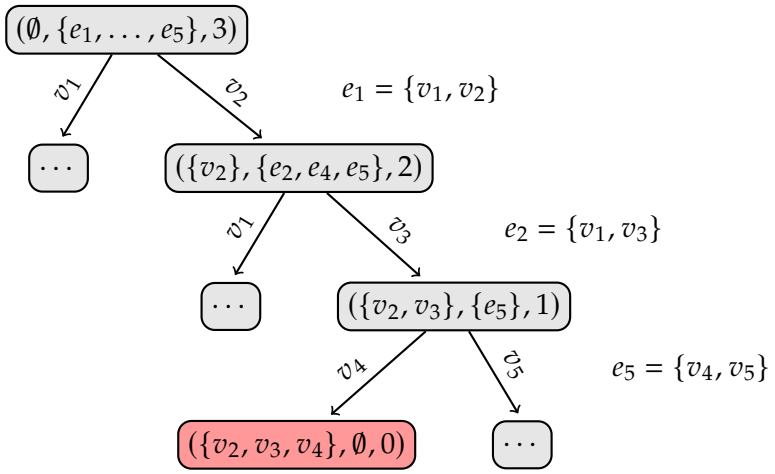


Figure 2.2: A part of the bounded search tree computed for the graph given in Figure 2.1. The accepting leaf corresponds to the vertex cover $\{v_2, v_3, v_4\}$ of the graph. The leaf in the tree is marked red.

not lead to a vertex cover in the end, the algorithm returns to the other branch of $e = \{v, w\}$ and adds w to the vertex cover instead of v.

Intuitively, the algorithm is a depth-first search on a binary tree. A part of the tree computed for the graph from Figure 2.1 is shown in Figure 2.2. Each node consists of three arguments (C, R, i). Here, $C \subseteq V$ denotes a partial vertex cover that we have constructed, $R \subseteq E$ is the set of edges not covered by C, and i is the number of vertices that we can still add to the cover. The root is given by $(\emptyset, E, k)$. The children of a node (C, R, i) are obtained by updating the arguments according to the chosen endpoint of the currently considered edge. Each branch eventually reaches a leaf, either if $R = \emptyset$ or if $R \neq \emptyset$ but $i = 0$. The former constitutes an *accepting* leaf. A vertex cover of size at most k exists if and only if there is an accepting leaf in the tree.

It is left to determine the time complexity of the algorithm. Note that the tree is of height at most k, and the arguments for each node can be updated in time $O(n + m)$. Hence, the algorithm takes a total running time of $O(2^k \cdot (n + m))$. This matches our initial goal of achieving a running time of the form $O(f(k) \cdot p(n, m))$ with a moderately sized function $f(k)$.

The tree-approach employed above is an algorithmic technique called *Bounded Search Tree* [112, 170]. It is applicable to other decision problems as well [112] and originates in the branching technique of Davis and Putnam [116]. In contrast to the algorithm for Vertex Cover which runs in time $O(n^k \cdot m)$, bounded search tree provides a significant speed-up which allows for finding solutions to instances larger than before. Taking a step back, knowing that Vertex Cover is NP-complete is only a rough classification. When designing efficient deterministic algorithms, the algorithmic

techniques provided by parameterized and fine-grained complexity are beneficial. *The right technique in the right place can make all the difference*[1].

2.2 Parameterized Problems

We introduce *parameterized problem*, the basic notion of a problem in parameterized complexity. It formalizes the handling of parameters and is key to expressing results like the one above uniformly. Roughly, a parameterized problem is defined like a classical decision problem with an additional input: the *parameter*. A parameter describes an instance of a problem or the expected solution in more detail. Usually, it is an upper or lower bound on certain aspects of one of them. We define it as a natural number. Let Σ be a finite alphabet and let Σ^* denote the set of finite words over Σ.

Definition 2.1. A *parameterized problem* L is a subset of $\Sigma^* \times \mathbb{N}$. An *instance* of L is a pair (x, k) where $x \in \Sigma^*$ is a classical instance and k is the *parameter*.

Since the parameter is part of the input, we can define the *size* of an instance (x, k) to be $|(x, k)| = |x| + k$, where k is given in unary. The size becomes important once we define *tractable* parameterized problems. For some problems, we examine multiple parameters at the same time. Definition 2.1 can be extended appropriately. We introduce the basic definition for a single parameter only to prevent technical difficulties. We focus on two examples of (parameterized) problems that are essential throughout the thesis.

Clique A fundamental parameterized problem is that of finding a *clique* of a certain size in a given graph. The problem known as Clique is one of Karp's 21 NP-complete problems [229]. Due to its intrinsic hardness, that we will elaborate on later, Clique is commonly used to derive lower bounds for other parameterized problems [85, 86, 112, 138, 140, 141]. To state the problem formally, recall that a *clique* of an undirected graph $G = (V, E)$ is a subset $C \subseteq V$ such that for each two vertices $v, w \in C$, there is an edge $\{v, w\} \in E$. Two vertices in C are always adjacent. The *size* of a clique is the number $|C|$ of its vertices. Given an undirected graph G and an integer k, the problem Clique asks whether G contains a clique of size exactly k.

Clique(k)

Input:	A graph G and an integer $k \in \mathbb{N}$.
Parameter:	k.
Question:	Does there exist a clique of size k in G?

[1] Originally: *the right man in the wrong place can make all the difference in the world*. By *G-Man* `https://en.wikipedia.org/wiki/G-Man_(Half-Life)` in *Half-Life 2*.

Note that the parameterized problem Clique(k) explicitly mentions the parameter k in the problem name. The notation is used to separate the parameterized problem from the (unparameterized) decision problem Clique. We call Clique(k) the *parameterization* of Clique *by* k. Parameterizations indicate the currently considered parameters of a problem.

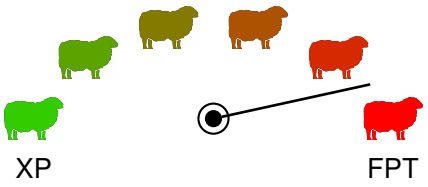
Set Cover A further problem that we use throughout the thesis is Set Cover. The problem was proven NP-complete by Karp [229] and, like Clique, serves as a lower bound in parameterized complexity [110, 112, 140, 141]. In Set Cover, we are given a family $\mathcal{F} \subseteq \mathcal{P}(U)$ of sets over a *universe* U and an integer $r \in \mathbb{N}$. The task is to find r sets $S_1, \ldots, S_r$ in $\mathcal{F}$ that form a *set cover* of U. This means that each element in U is contained in an S_i, $U = \bigcup_{i \in [1..r]} S_i$.

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 $U = \bigcup_{i \in [1..r]} S_i$?

We stated Set Cover as a classical decision problem. The canonical parameterization is Set Cover(r), by the size r of the cover. In Chapter 4, we will see that Set Cover(r) is of particularly high complexity. Therefore, another parameterization turns out to be more useful for our purpose. We usually consider Set Cover(n), where n is the size of the universe U.

3. Fixed-Parameter Tractability



We formalize the notion of a *tractable* parameterized problem and introduce techniques that allow for proving tractability. Intuitively, a parameterized problem is tractable if it can be solved by an algorithm the running time of which can be separated into two factors: one depending only on the considered parameter, the other one being a polynomial in the input size. Problems that admit such an algorithm are called *fixed-parameter tractable* (FPT) since fixing the parameter to a constant value only leaves the polynomial part with a degree independent of the parameter.

Fixed-parameter tractability can sometimes be tricky to obtain. Typically, it needs an FPT-*technique*, an algorithmic gimmick or strategy that has proven to be useful in obtaining algorithms that meet the requirements formulated by fixed-parameter tractability. FPT-techniques are the backbone of many algorithms in parameterized complexity and there is whole zoo of such techniques available [112, 170]. Usually, it requires some experience to find the right one and to apply it correctly to the problem of interest. We present a selection of FPT-techniques that are needed throughout this thesis.

In Section 3.1, we formally introduce the complexity class FPT of fixed-parameter tractable problems as well as the class XP, a more general complexity class of so-called *slicewise polynomial* problems. The remaining sections in this chapter are devoted to FPT-techniques. We focus on three techniques: *dynamic programming* — Section 3.2, *subset convolution* — Section 3.3, and *kernelizations* — Section 3.4. The idea behind *dynamic programming* is storing intermediary results, *subset convolution* is an algebraic technique that uses *discrete transforms* to compute in a domain of choice, and *kernelizations* are a form of *polynomial preprocessing*. We introduce the basic notions around the techniques and give examples for their application.

3.1 Basic Parameterized Complexity Classes

In Section 2.1, we have seen an algorithm for Vertex Cover running in time $O(2^k \cdot (n+m))$. As discussed above, we consider the problem *tractable* since a *fixed parameter* k would leave a polynomial of constant degree 1. The point is that the running time is of the form $O(f(k) \cdot p(n,m))$, where f is a function only depending on k and p is a polynomial. This prevents an expo-

nential dependence between k and n, m. In general, we consider a problem tractable if it admits an algorithm with a running time of this form.

Definition 3.1. A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *fixed-parameter tractable* (FPT) if there is an algorithm that, given an instance (x, k), decides whether $(x, k) \in L$ and runs in time $f(k) \cdot |(x, k)|^d$, where f is a computable function only depending on k and $d \in \mathbb{N}$ is a constant.

The notation FPT is also used for the *complexity class* comprising all problems that are fixed-parameter tractable. Hence, a problem is FPT if it lies in the class FPT. Given a problem from this class, the corresponding algorithm that shows membership is called an FPT*-algorithm*. The running time $f(k) \cdot n^d$, where n is the size of the input, of an FPT-algorithm is sometimes denoted by $O^*(f(k))$. This O^**-notation* suppresses the polynomial factor and focuses on the dominant part of the running time: the function $f(k)$.

The discovery of FPT goes back to Downey and Fellows' early work with other authors [5, 136, 157, 158]. The definition, as simple as it seems, was fundamental to parameterized complexity. In fact, one of the major tasks of parameterized complexity is to examine the difference between problems that are FPT and those that are *intractable*. Subsequently, the development of a whole theory around intractability [135, 136, 138, 140, 141] followed, providing tools for separating tractable from intractable problems. We will consider the intractability theory in Chapter 4.

We have already seen that Vertex Cover(k) is fixed-parameter tractable with an algorithm running in $O^*(2^k)$. In the following, we provide two further examples of tractable problems that will repeatedly appear in the thesis.

Example 3.2. The following parameterizations are fixed-parameter tractable.

a) We consider Set Cover(m), where $m = |\mathcal{F}|$ is the size of the given family $\mathcal{F}$ of sets. Since we are looking for r sets within the family $\mathcal{F}$ that cover U, we can just iterate over all subfamilies $\mathcal{F}'$ of $\mathcal{F}$ of size r and test whether the sets in $\mathcal{F}'$ cover U. Once we have fixed $\mathcal{F}'$, the test is simple and can be done in polynomial time. However, we need to iterate over $\binom{m}{r} \leq 2^m$ possible choices for $\mathcal{F}'$. Hence, Set Cover can be solved in time $O^*(2^m)$ which entails that Set Cover(m) is FPT. Note that the size m of the family $\mathcal{F}$ is a rather large parameter in general. Therefore the presented algorithm is not very efficient on practical instances although it is an FPT-algorithm. We obtain a more detailed picture of the parameterized complexity of Set Cover in Sections 3.2, 3.3, 4.4, and 5.3. This includes more efficient algorithms as well as intractability results.

b) The well-known problem SAT [105, 229] asks for the satisfiability of a Boolean formula. By iterating over all possible evaluations of the n variables, we obtain an algorithm for SAT running in time $O^*(2^n)$. This shows

that SAT(n) is FPT. Although this algorithm is very simple, it is believed that we cannot improve on the exponential dependence on n any further. In fact, this is formulated as a widely believed hypothesis that is used to obtain similar lower bounds for other parameterized problems. We consider the hypothesis and similar assumptions in Chapter 5.

One may ask whether there is a simple FPT-algorithm for the problem Clique(k). To this end, let us consider a brute force approach. We iterate over all k-tuples of vertices of the given graph and test whether each two vertices of a tuple are distinct and adjacent. The algorithm runs in time $O^*(n^k)$, where n is the number of vertices. Obviously, this is not an FPT-algorithm but no better algorithm for Clique(k) is known. In particular, no FPT-algorithm is known. We will see in Chapter 4 that the existence of such an algorithm is rather unlikely since Clique(k) is complete for W[1], a class of intractable problems. The brute force algorithm only shows that for a fixed parameter k, the problem Clique(k) can be solved in polynomial time. When we think of each subproblem with a fixed value for k as a *slice*, we may call Clique(k) *slicewise polynomial*. The following definition formalizes the notion.

Definition 3.3. A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called *slicewise polynomial* (XP) if there is an algorithm that, given an instance (x, k), decides membership in L in time $|(x, k)|^{g(k)}$, where g is a computable function. The *complexity class* of slicewise polynomial problems is also denoted by XP.

The class XP is more general than FPT. Instead of the constant exponent for the polynomial factor, that we require in an FPT-algorithm, XP allows for having a whole function in the parameter. This immediately yields that each fixed-parameter tractable problem is also slicewise polynomial, constituting the first inclusion in the landscape of parameterized complexity classes:

$$\mathsf{FPT} \subseteq \mathsf{XP}.$$

We will refine the inclusion in Chapter 4 when we consider intractable problems in more detail. Roughly, the outcome will be that the class XP is much larger than FPT: in fact, it is so much larger that a whole hierarchy of intractability classes fits between FPT and XP. But before we turn to intractability, we focus on techniques for showing membership in FPT.

3.2 Dynamic Programming

Dynamic programming was discovered as an algorithmic technique by Bellman in the 1950s [33]. Since then, it evolved to one of the major techniques for problems that admit a recursive formulation. Dynamic programming found applications in different fields of computer science and mathematics like graph algorithms [34, 163, 173], string algorithms [106, 207, 284, 322],

context-free languages and parsing [144, 212, 335], amongst others [106]. In parameterized complexity, the technique has proven to be a reliable tool providing fast FPT-algorithms. It was applied to a whole variety of important problems in the field like Set Cover [171], Hamiltonian Path, Hamiltonian Cycle and Traveling Salesperson [35, 204], Steiner Tree [142], or Longest Path [13]. Moreover, dynamic programming often builds the basis of more refined FPT-techniques. This includes *fast transforms*, *subset convolution* [48], and further algebraic methods [112], *Cut & Count* [113], and randomized techniques like *Color Coding* [13]. The downside of dynamic programming is its memory consumption. In order to improve the running time, memory requirements can be high — seemingly a necessary trade-off. But there are even techniques [261] to limit the space usage of dynamic programming algorithms. We will later apply dynamic programming in the context of program verification tasks. More precise, the technique builds the basis for our algorithms on safety and liveness in leader contributor systems, see Chapter 8, as well as for checking consistency in shared memories, see Chapter 9.

The rough idea of dynamic programming is to decompose a problem into subproblems, solve these recursively, and combine their solutions to obtain a solution for the whole problem. When decomposing subproblems further in this process, it might be the case that the same *subsubproblem* occurs in the decomposition of different subproblems. Phrased differently, subproblems can *overlap* [106]. The strength of dynamic programming comes from the fact that it solves each of these *susubproblems* just once and stores the solution in a table. Then, whenever the solution to a *subsubproblem* is needed again, it can be looked up in the table rather than being computed again.

Given a (parameterized) problem, we usually apply dynamic programming the following way. First, we identify a formulation of the problem that allows for a decomposition into nested subproblems. For problems in the context of parameterized complexity, this is typically achieved by considering a suitable subset lattice. Then, we build a table along the structure and find a recurrence among its entries that allows for computing the solution to larger subproblems from smaller (subsubproblems) ones. The actual algorithm is a bottom-up table-filling, starting from base cases that do not depend on further subproblems. Once the table is filled, we can look up the entry corresponding to the solution of the original problem.

We work out two examples of dynamic programming applied to problems important for parameterized complexity, namely Hamiltonian Cycle and Set Cover. For further examples and a more detailed introduction to dynamic programming, we refer to Cormen, Leiserson, Rivest, and Stein [106].

3.2.1 Dynamic Programming for Hamiltonian Cycle

In the problem Hamiltonian Cycle we are given a graph and we have to decide the existence of a cycle going through each vertex exactly once. The

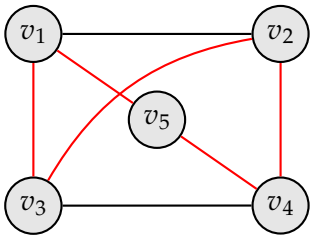


Figure 3.1: A graph $G = (V, E)$ with vertices $V = \{v_1, \ldots, v_5\}$. The red marked edges form a Hamiltonian cycle $v_1.v_3.v_2.v_4.v_5$ of G.

problem, like the related Hamiltonian Path, is a classical problem [60, 133] of graph theory and known to be NP-complete [229]. As an example, consider the graph given in Figure 3.1. The red marked edges form a Hamiltonian cycle — each vertex along the cycle is visited exactly once.

To formalize Hamiltonian Cycle, let $G = (V, E)$ be an undirected graph. A *simple path* in G is a path where no vertex occurs twice, a sequence of vertices $v_1.v_2 \ldots v_k$ such that for each $i \in [1..k-1]$ there is an edge $\{v_i, v_{i+1}\} \in E$ and for $i \neq j$ we have $v_i \neq v_j$. A simple path involving all vertices of V is called a *Hamiltonian path*. If a Hamiltonian path $C = v_1.v_2 \ldots v_n$ forms a cycle, which means there is an edge $\{v_n, v_1\} \in E$, we call C a *Hamiltonian cycle*. The decision problem Hamiltonian Cycle asks for the existence of such a cycle.

Hamiltonian Cycle

Input: A graph G.

Question: Is there a Hamiltonian cycle in G?

Note that in the formulation of the problem the input graph G is undirected. There is also a variant of Hamiltonian Cycle where G is assumed to be directed. The definition of a *Hamiltonian cycle* carries over to this case and the corresponding decision problem is still NP-complete [229].

We analyze the parameterized complexity of Hamiltonian Cycle in terms of the parameter $n = |V|$, the number of vertices. To this end, we first consider a simple brute force approach to the problem. It iterates over all sequences of vertices of length n and checks if the sequence is indeed a Hamiltonian cycle. The latter takes polynomial time but we have to iterate over n^n many different sequences in the worst case. Hence, the algorithm runs in time $O^*(n^n)$. Our goal is to show that dynamic programming can be used to speed things up. In fact, we can solve Hamiltonian Cycle in time $O^*(2^n)$ which is a much more efficient FPT-algorithm than the brute force approach. Besides that, the currently fastest algorithm for Hamiltonian Cycle, as it is stated here, is the $O^*(1.66^n)$-time algorithm by Björklund [44] based on algebraic techniques.

The dynamic programming approach for Hamiltonian Cycle goes back to Bellman, Held, and Karp [35, 204] and is also known as *Bellman-Held-Karp algorithm*. Formally, the algorithm proves the following result.

Theorem 3.4. Hamiltonian Cycle *can be solved in time* $O(2^n \cdot n \cdot d)$.

The parameter d is the maximal *degree* of a vertex. It is defined as the maximal number of vertices adjacent to a single vertex: $d = \max_{v \in V}(\deg(v))$, where $\deg(v)$ is the size of the set $\{v' \in V \mid \{v, v'\} \in E\}$.

The dynamic programming for Hamiltonian Cycle relies on the idea of building up a simple path step by step until it visits each vertex and can be completed to a cycle [112]. But instead of remembering the precise path taken, we only store the beginning of the path, the end, and the set of vertices that are involved. Knowing this information suffices for establishing a recursive formulation: if there is a simple path on a set of vertices with a particular end t, we can extend the path by appending a neighboring vertex of t that has not been visited yet. In the end, when all vertices have been visited, we need to check that the last vertex of the simple path has an edge to the intended beginning which constitutes a Hamiltonian cycle.

We define a table T that stores the information required for turning the idea into an algorithm. First, fix a vertex $s \in V$. This is where we assume a Hamiltonian cycle to start. Table T has an entry $T[X, t]$ for each subset $X \subseteq V$ that contains s and for each vertex $t \in V \setminus X$. The entry will be 1 if there there is a simple path that starts in s, visits all the vertices of X, and ends in t. If such a path does not exist, the entry of the table is 0. Formally, we have

$$T[X, t] = \begin{cases} 1, \text{ if there is a simple path on } X \cup \{t\} \text{ starting in } s, \text{ ending in } t, \\ 0, \text{ otherwise.} \end{cases}$$

Assume we have already filled the table T. Then, we can check the existence of a Hamiltonian cycle. A cycle exists if and only if there is a neighboring vertex t of s and a simple path from s to t visiting all vertices in $V \setminus \{t\}$. Since we can test the latter by reading the entry $T[V \setminus \{t\}, t]$, we obtain the following lemma formulating the correctness of the approach.

Lemma 3.5. *There exists a Hamiltonian cycle in the input graph* $G = (V, E)$ *if and only if the following Boolean expression evaluates to true:*

$$\bigvee_{\substack{t \in V, \\ \{t,s\} \in E}} T[V \setminus \{t\}, t].$$

It is left to explain how the table is filled. We consider the base cases and then establish a recurrence among the entries of T that allow for a bottom-up filling. The base cases are simple. We set $T[\{s\}, t] = 1$ for each $t \in V$ that has an edge $\{s, t\} \in E$. For other choices of t, the corresponding entry is 0.

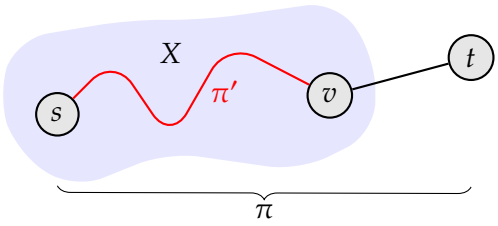


Figure 3.2: Any simple path π from s to t visiting X, the blue highlighted area, can be decomposed into a simple (sub)path π', marked red, from s to v, and a single edge $\{v, t\}$. The path π' visits the vertices $X \setminus \{v\}$ along the way.

For the recurrence, we rely on the above idea of extending a simple path. However, our formulation will be the other way around: instead of prolonging such a path, we will take away a vertex and shorten it. The reason is that we want to obtain a top-down recurrence that evaluates entries of longer paths from entries of shorter ones. The key observation is the following: there exists a simple path π in G from s to t visiting the vertices X if and only if there is a vertex $v \in X$ with $\{v, t\} \in E$ along with a simple path π' from s to v that visits $X \setminus \{v\}$. We provide an illustration in Figure 3.2.

Lemma 3.6. *Table T admits the following recurrence for entries with $\{s\} \subsetneq X$:*

$$T[X, t] = \bigvee_{\substack{v \in X, \\ \{v,t\} \in E}} T[X \setminus \{v\}, v].$$

The algorithm for Hamiltonian Cycle evaluates the table T along the base cases and the recurrence given in Lemma 3.6. Once T is filled, the criterion from Lemma 3.5 is checked. We determine the time complexity of the approach. Table T has at most $2^n \cdot n$ many entries that are evaluated. Evaluating a single entry $T[X, t]$ with the recurrence takes time $O(d)$ since we have to iterate over each neighboring vertex v of t and read-off the value stored in $T[X \setminus \{v\}, v]$. The latter takes constant time. Hence, T can be filled in time $O(2^n \cdot n \cdot d)$. The estimate also covers the time needed to evaluate the expression of Lemma 3.5. This completes the proof of Theorem 3.4.

3.2.2 Dynamic Programming for Set Cover

Recall that in the problem Set Cover we are given a family $\mathcal{F}$ of sets over a universe U and an integer $r \in \mathbb{N}$. The question to answer is whether there are r sets in $\mathcal{F}$ that cover U. We have already argued in Example 3.2 that Set Cover(m) is FPT, where $m = |\mathcal{F}|$ is the size of the family. However, m is usually a rather large parameter, much larger than $n = |U|$, the size of the

universe. Instead of considering m as a parameter, we employ a dynamic programming to show that Set Cover(n) is fixed-parameter tractable. Moreover, the obtained FPT-algorithm is efficient: it solves the problem in time $O^*(2^n)$.

Theorem 3.7. *Set Cover can be solved in time $O(2^n \cdot n \cdot m)$.*

We consider the dynamic programming for Set Cover given in [171]. The main idea is to consider covers for subsets of the universe U and to make sets from the family $\mathcal{F}$ available step by step. To this end, assume that $\mathcal{F}$ is of the form $\mathcal{F} = \{F_1, \ldots, F_m\}$. We define a table T that captures the recursive structure of the problem. It has an entry $T[X, j]$ for each subset $X \subseteq U$ and $j \in [0..m]$. The entry stores the minimal size of a *cover* C of X which only contains sets from the subfamily $\{F_1, \ldots, F_j\}$. Here, *cover* means that $X \subseteq \bigcup_{S \in C} S$. Formally, we define the entry $T[X, j]$ as follows:

$$T[X, j] = \min_{\substack{C \subseteq \{F_1, \ldots, F_j\}, \\ C \text{ covers } X}} |C|.$$

If there is no cover $C \subseteq \{F_1, \ldots, F_j\}$ of the set X, we set $T[X, j] = \infty$. We are interested in the entry $T[U, m]$. It stores the minimal size of a set cover of U taken from the family $\mathcal{F}$. If the entry is strictly larger than the given integer r, there will not be a set cover of size r for U. Otherwise, there is a set cover of size at most r. Such a cover can always be extended to one of size exactly r by adding further sets. We obtain the correctness of the approach.

Lemma 3.8. *There is a set cover $C \subseteq \mathcal{F}$ of U with $|C| = r$ if and only if $T[U, m] \leq r$.*

We set up a recurrence among the entries of the table. The rough idea is the following. Entry $T[X, j]$ is defined over covers within the subfamily $\{F_1, \ldots, F_j\}$. To compute it, we recurse on entries of T that are defined over covers in $\{F_1, \ldots, F_{j-1}\}$, entries of the form $T[X', j-1]$. To this end, note that each cover $C \subseteq \{F_1, \ldots, F_j\}$ of X falls in one of two categories: either C contains F_j or it does not. In the latter case, C is already a cover of X in $\{F_1, \ldots, F_{j-1}\}$ and hence appears in the definition of $T[X, j-1]$. In the former case, $F_j \in C$. Then, the set $X \setminus F_j$ is covered by $C \setminus \{F_j\}$ which lies in $\{F_1, \ldots, F_{j-1}\}$ and appears in $T[X \setminus F_j, j-1]$. From this observation, we can extract a recurrence that ties the entries together. It is formalized in the following lemma, the proof of which can be found in Appendix A.1.1.

Lemma 3.9. *The table T admits the following recurrence:*

$$T[X, j] = \min(T[X, j-1], 1 + T[X \setminus F_j, j-1]).$$

The algorithm for Set Cover is a bottom-up table-filling that employs the recurrence proven in Lemma 3.9 and the correctness criterion given in Lemma 3.8. It starts from the following base cases:

$$T[\emptyset, 0] = 0 \text{ and } T[X, 0] = \infty \text{ for each } X \neq \emptyset.$$

The algorithm needs to compute all $2^n \cdot (m+1)$ entries of T. Computing each $T[X, j]$ takes time $O(n)$. To see this, note that a minimal set cover of X has size at most $|X| \leq n$. According to Lemma 3.9, we only need to compute the minimum of two integers that are bounded by n. Altogether, we obtain a running time of the form $O(2^n \cdot n \cdot m)$, as stated in Theorem 3.7.

Note that the size m of the family appears as a linear factor in the running time. Like mentioned before, m might be rather large in some applications and related problems [51, 69, 168, 170]. In fact, it can be exponential in n. It is therefore essential to avoid m even as a linear factor in the running time. In Section 3.3 we obtain a corresponding algorithm.

3.3 Subset Convolution

Many classical NP-complete decision problems arise in the context of finding partitions. These problems ask to decompose an underlying structure like a set or a graph while satisfying certain problem-specific properties. The latter are described by a family of sets, like in Set Cover, or more generally by a characteristic function that indicates valid members of a partition by mapping them to 1 and others to 0. *Fast subset convolution* [48] is a recently developed technique that allows for quickly reasoning about all partitions while applying (characteristic) functions to their members. Fast subset convolution was used to solve a variety of decision problems and is incorporated into further state of the art techniques in the field. Formally, *fast subset convolution* computes the *subset convolution*, an operator well-known in functional analysis [297].

Definition 3.10. Let U be a finite set and $f, g : \mathcal{P}(U) \to \mathbb{Z}$ two functions, mapping subsets of U to integers. The *subset convolution* of f and g is the function $f * g : \mathcal{P}(U) \to \mathbb{Z}$ mapping each subset X of U to the sum

$$(f * g)(X) = \sum_{Y \subseteq X} f(Y) \cdot g(X \setminus Y).$$

The following equivalent formulation highlights that the subset convolution takes all partitions with two components into account.

$$(f * g)(X) = \sum_{\substack{A \cup B = X \\ A \cap B = \emptyset}} f(A) \cdot g(B).$$

Subset convolution is associative and commutative. Consequently, applying it to more than two functions yields a function that allows for reasoning about partitions involving three or more components. The following lemma states the form of the function. The proof follows from the definition.

Lemma 3.11. *Let U be a finite set and $f_1, \ldots, f_k : \mathcal{P}(U) \to \mathbb{Z}$ functions. The subset convolution $f_1 * \cdots * f_k : \mathcal{P}(U) \to \mathbb{Z}$ takes the following form:*

$$(f_1 * \cdots * f_k)(X) = \sum_{\substack{A_1 \cup \ldots \cup A_k = X \\ A_i \cap A_j = \emptyset, i \neq j}} f_1(A_1) \cdots f_k(A_k).$$

The sum on the right hand side ranges over all subsets $A_1, \ldots, A_k \subseteq X$ partitioning X. Hence, the convolution sums up all *(ordered) k-partitions*, all k-tuples $(A_1, \ldots, A_k)$ with $A_i \subseteq X$, $\bigcup_{i \in [1..k]} A_i = X$, and $A_i \cap A_j = \emptyset$ for $i \neq j$.

The form given in Lemma 3.11 makes subset convolution a flexible tool with many degrees of freedom. Many algorithms for prominent problems rely on it. For instance, fast subset convolution was applied to Steiner Tree [48] to obtain the currently fastest algorithm for the problem, it is incorporated into the *Cut & Count*-technique [113] which solves connectivity problems on graphs, and it was applied to variants of Dominating Set [223, 319]. The technique is further capable of counting partitions and covers of sets, *colorings* and *spanning forests* of graphs [48, 170], and of computing the *Tutte polynomial* [49]. As we will see in Chapter 6, we employ fast subset convolution in an algorithm for *bounded context switching*. To the best of our knowledge, this is the first application in the context of automata theory and verification.

The wide landscape of applications raises the question on how to compute the complex expression given in Lemma 3.11. We start with computing the convolution of two functions. Let $f, g : \mathcal{P}(U) \to \mathbb{Z}$ be functions and let $n = |U|$ denote the size of the underlying set. By *computing* the convolution $f * g$ we mean computing all of its 2^n values, assuming that the values of f and g are already given. After the computation, we want to have a table that stores for each input $X \subseteq U$ the integer $(f * g)(X)$. An algorithm for computing the convolution is measured in the number of *arithmetic operations* it needs.

A simple algorithm can be obtained by following Definition 3.10. It evaluates $f * g$ in each subset X separately. For $X \subseteq U$, the algorithm sums up the terms $f(Y) \cdot g(X \setminus Y)$ for each subset $Y \subseteq X$ and stores the result. Hence, if $k = |X|$, it takes $O(2^k)$ arithmetic operations to evaluate $(f * g)(X)$. Since there are $\binom{n}{k}$ subsets of U of size k, the number of arithmetic operations for computing $f * g$ can be estimated, up to a constant factor by

$$\sum_{k \in [0..n]} \binom{n}{k} \cdot 2^k = 3^n.$$

The equality follows from the binomial theorem. It shows that the simple algorithm requires $O(3^n)$ arithmetic operations. Björklund, Husfeldt, Kaski, and Koivisto [48] were the first ones to improve upon the simple approach. The authors developed the fast subset convolution, an algorithm which computes the convolution of two or more functions in only $O^*(2^n)$ arithmetic operations. This paved the way for fast convolution-based algorithms and techniques which are nowadays standard in parameterized complexity.

Theorem 3.12. *Let U be a set of size n and $f, g : \mathcal{P}(U) \to \mathbb{Z}$ two functions. The subset convolution $f * g$ can be computed in $O(2^n \cdot n^2)$ arithmetic operations.*

In the theorem, we assume the values of f and g to be given. If we have more information on the images of both functions, we can estimate the worst-case time complexity rather than the number of arithmetic operations. To state the corresponding result, we need the time requirements of arithmetic. The costly operation is multiplication. Let $mult(b)$ be the time of multiplying two b-bit integers. Precise time bounds can be obtained by the Schönhage-Strassen multiplication [300], where $mult(b) = O(b \cdot \log b \cdot \log\log b)$, or by the newer result $mult(b) = b \cdot \log b \cdot 2^{O(\log^* b)}$ of Fürer [181]. Addition, subtraction, and comparison can be performed in time $O(b)$.

Corollary 3.13. *Let U be a set of size n, $M \in \mathbb{N}$ an integer, and $f, g : \mathcal{P}(U) \to \mathbb{Z}$ two functions mapping into the set $\{-M, -M+1, \ldots, M-1, M\}$. The subset convolution $f * g$ can be computed in time $O(2^n \cdot n^2 \cdot mult(n \cdot \log M))$.*

In the following sections, we will present fast subset convolution and provide proofs for Theorem 3.12 as well as Corollary 3.13. We mainly follow the literature [49, 112, 170]. Proofs in this section are taken from these works. To showcase the applicability, we apply fast subset convolution to count partitions and colorings and we will consider a variant of subset convolution required to solve the problem Set Cover.

3.3.1 Fast Transforms

Fast subset convolution requires a notion of *transforms*. These are operators on functions that help representing the subset convolution in terms of a *simpler* product operation. In this setting, *simple* means that computing this product requires less arithmetic operations than directly evaluating the convolution. The algorithm fast subset convolution is a clever way of reducing the computation of the subset convolution to the simpler product by applying and inverting a particular transform that can be evaluated efficiently.

Assume we have already constructed an operator T that turns the subset convolution $*$ into a simpler product $\circledast$ by satisfying the following equation for all functions f and g:

$$T(f * g) = Tf \circledast Tg.$$

Moreover, assume that T is bijective with inverse operator T^{-1}. When applying T^{-1} to the above equation, we obtain

$$f * g = T^{-1}(T(f * g)) = T^{-1}(Tf \circledast Tg).$$

Hence, instead of computing the convolution, we can also compute Tf and Tg, evaluate $Tf \circledast Tg$, and then apply the inverse T^{-1}. An illustration is given

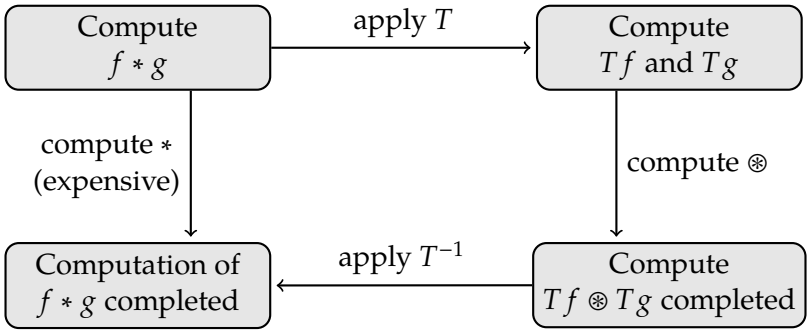


Figure 3.3: Directly computing $f * g$ is expensive. Instead, we first apply the transform T to obtain Tf and Tg, compute $Tf \circledast Tg$, and then apply T^{-1} to obtain the convolution $f * g$. Each step takes $O^*(2^n)$ arithmetic operations.

in Figure 3.3. We will see that each of the three steps takes $O^*(2^n)$ arithmetic operations. Therefore, chaining the steps results in a faster algorithm for computing $f * g$ than the direct evaluation.

The trick of turning a *convolution* into a simpler operation is well-known in the theory of transforms and sometimes referred to as the *convolution theorem* [66]. We prove the corresponding theorem for the subset convolution by employing a transform that is related to other famous transforms like the *Fourier transform* [66, 257] or the *Walsh-Hadamard transform* [8]. It is called *zeta transform* and is not to be confused with the *Z-transform* [66] for processing discrete-time signals. For the definition and the remaining section, we fix U to be a finite set of size n.

Definition 3.14. Let $f : \mathcal{P}(U) \to \mathbb{Z}$ be a function. The *zeta transform* ζ of f is a function $\zeta f : \mathcal{P}(U) \to \mathbb{Z}$ mapping a subset X of U to the integer

$$(\zeta f)(X) = \sum_{Y \subseteq X} f(Y).$$

Once we have applied the zeta transform to a function, we can regain the original function by the so-called *Möbius inversion formula*. This means that the zeta transform is actually bijective and we know the inverse operator. Before we define the inverse formally, we first prove the inversion formula.

Lemma 3.15. *For a function $f : \mathcal{P}(U) \to \mathbb{Z}$, we have*

$$f(X) = \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot (\zeta f)(Y).$$

Proof. We expand the right-hand side of the equation and reorder the terms:

$$\begin{aligned}\sum_{Y\subseteq X}(-1)^{|X\setminus Y|}\cdot(\zeta f)(Y) &= \sum_{Y\subseteq X}(-1)^{|X\setminus Y|}\cdot\sum_{Z\subseteq Y}f(Z)\\ &= \sum_{Z\subseteq X}f(Z)\cdot\sum_{Z\subseteq Y\subseteq X}(-1)^{|X\setminus Y|}\\ &= f(X)+\sum_{Z\subsetneq X}f(Z)\cdot\sum_{Z\subseteq Y\subseteq X}(-1)^{|X\setminus Y|}.\end{aligned}$$

It remains to show that the latter sum evaluates to 0. First note that it ranges over all subsets of the non-empty set $X \setminus Z$. Hence, if $|X \setminus Z| = k$, we can reformulate the sum as the simpler expression:

$$\sum_{Z\subseteq Y\subseteq X}(-1)^{|X\setminus Y|}=\sum_{i=0}^{k}(-1)^i\cdot\binom{k}{i}=0.$$

Since each non-empty set has an equal amount of subsets of even size and odd size, the expression vanishes. This completes the proof. □

From Lemma 3.15, we can derive the inverse operator of the zeta transform. It is named after the Möbius inversion formula and will be a central tool in the development of the fast subset convolution.

Definition 3.16. The *Möbius transform* μ of a function $f : \mathcal{P}(U) \to \mathbb{Z}$ is the function $\mu f : \mathcal{P}(U) \to \mathbb{Z}$ that maps a subset $X \subseteq U$ to the integer

$$(\mu f)(X)=\sum_{Y\subseteq X}(-1)^{|X\setminus Y|}\cdot f(Y).$$

Lemma 3.15 shows that applying the Möbius transform after the zeta transform results in the identity. When composing both operators, we obtain the identity operator *id* mapping each function $f : \mathcal{P}(U) \to \mathbb{Z}$ to itself:

$$\mu\circ\zeta = id.$$

The other direction also holds true. First applying Möbius transform and then zeta transform results in the identity. The proof is similar. We have

$$\zeta\circ\mu = id.$$

The above shows that ζ and μ are inverse operators. Hence, the transforms meet the first condition on the way to the fast subset convolution, as shown in Figure 3.3. The second condition the transforms have to fulfill is that they must be computable in $O^*(2^n)$ arithmetic operations.

In the subsequent theorem, we describe the desired algorithms for computing the zeta and Möbius transforms. These are also called *fast zeta* and *fast Möbius transform*. Both algorithms rely on dynamic programming and can be dated back to the work of Yates in 1937 [334].

Theorem 3.17. *Given the values of a function $f : \mathcal{P}(U) \to \mathbb{Z}$, the zeta transform ζf and the Möbius transform μf can be computed in $O(2^n \cdot n)$ arithmetic operations.*

Proof. We may assume without loss of generality that $U = \{1, \ldots, n\}$. Since f maps from the subsets of U, we can also consider it as a function on bitvectors $f : \{0,1\}^n \to \mathbb{Z}$. Note that each subset $X \subseteq U$ corresponds to its characteristic vector $(x_1, \ldots, x_n) \in \{0,1\}^n$ where $x_j = 1$ if and only if $j \in X$. Then, the value of $f(x_1, \ldots, x_n)$ is given by $f(X)$.

We begin with the algorithm for the zeta transform. By interpreting f as a function on vectors, the transform can be rephrased as follows:

$$(\zeta f)(x_1, \ldots, x_n) = \sum_{y_1, \ldots, y_n \in \{0,1\}} [y_1 \leq x_1 \wedge \cdots \wedge y_n \leq x_n] \cdot f(y_1, \ldots, y_n).$$

The bracket $[y_1 \leq x_1 \wedge \cdots \wedge y_n \leq x_n]$ is meant to evaluate to 1 if the condition inside is met. Otherwise, it evaluates to 0. The bracket represents the fact that the set corresponding to $(y_1, \ldots, y_n)$ is a subset of the set corresponding to $(x_1, \ldots, x_n)$. Hence, the summation is over all subsets of the given set.

For computing the transform, we dynamically build up a table. It has an entry $\zeta[(x_1, \ldots, x_n), j]$ for each vector $(x_1, \ldots, x_n) \in \{0,1\}^n$ and each integer $j \in [0..n]$. The index j fixes certain bits in the transform. This allows for constructing a recurrence later. Formally, we define the entry $\zeta[(x_1, \ldots, x_n), j]$ with $j \in [1..n]$ to store the sum

$$\sum_{y_1, \ldots, y_j \in \{0,1\}} [y_1 \leq x_1 \wedge \cdots \wedge y_j \leq x_j] \cdot f(y_1, \ldots, y_j, x_{j+1}, \ldots, x_n).$$

Moreover, we set $\zeta[(x_1, \ldots, x_n), 0] = f(x_1, \ldots, x_n)$. The definition implies that $\zeta[(x_1, \ldots, x_n), n] = (\zeta f)(x_1, \ldots, x_n)$.

It is left to evaluate the table. To this end, we employ the following recurrence which holds for each integer $j \geq 1$:

$$\zeta[(x_1, \ldots, x_n), j] = \begin{cases} \zeta[(x_1, \ldots, x_n), j-1] & \text{if } x_j = 0, \\ \zeta[(x_1, \ldots, x_{j-1}, 0, x_{j+1}, \ldots, x_n), j-1] \\ \quad + \zeta[(x_1, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_n), j-1] & \text{if } x_j = 1. \end{cases}$$

For the correctness note that $x_j = 0$ implies that $y_j = x_j = 0$. Hence, there is no need to sum over y_j anymore and we can remove the condition $y_j \leq x_j$ from the bracket. This results in the entry $\zeta[(x_1, \ldots, x_n), j-1]$. If $x_j = 1$, there are two cases: $y_j = 0$ and $y_j = 1$. Setting the j-th coordinate to 0 or 1 results in the sum of the two entries as described in the recurrence.

The table has $O(2^n \cdot n)$ many entries. With the recurrence above, we can compute them all in $O(2^n \cdot n)$ arithmetic operations and hence obtain ζf. This completes the proof for the zeta transform. For the Möbius transform, a similar trick can be applied to obtain a corresponding algorithm. □

Now that we have fast algorithms for the zeta and the Möbius transform, we would like to proceed with the last step according to Figure 3.3, namely showing that the transforms can be used to represent the convolution in terms of a simpler product operation. Formally, we would like to have that $\zeta(f * g) = (\zeta f) \circledast (\zeta g)$, where $\circledast$ is some simple product. For the subset convolution, the approach does not work immediately. We will have to make some *adjustments* to the transforms to achieve this goal. Luckily, the adjustments will not spoil the work that we have invested into the first two steps. We still obtain an inversion formula and fast algorithms for computing the adjusted transforms. Moreover, we will see in Section 3.3.4 that the original zeta and Möbius transforms work as expected for the so-called *cover product*, a more liberal form the convolution that reasons over all covers of a set.

We adjust the zeta and Möbius transforms by introducing *ranks*. These allow for a controlled summation over subsets of a certain size.

Definition 3.18. Let $f : \mathcal{P}(U) \to \mathbb{Z}$ be a function. The *ranked zeta transform* $\hat{\zeta}$ of f is the function $\hat{\zeta} f : [0..n] \times \mathcal{P}(U) \to \mathbb{Z}$ defined by

$$(\hat{\zeta} f)(k, X) = \sum_{Y \subseteq X, |Y|=k} f(Y),$$

where $k \in [0..n]$ is an integer and X is a subset of U.

For a function $g : [0..n] \times \mathcal{P}(U) \to \mathbb{Z}$, the *ranked Möbius transform* $\hat{\mu}$ of g is the function $\hat{\mu} g : \mathcal{P}(U) \to \mathbb{Z}$ mapping a subset $X \subseteq U$ to the integer

$$(\hat{\mu} g)(X) = \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot g(|X|, Y).$$

Similar to the original zeta and Möbius transforms, we obtain an inversion formula which is compatible with $\hat{\zeta}$ and $\hat{\mu}$. Hence, the first condition on our way to fast subset convolution is met by the ranked transforms.

Lemma 3.19. *We have $\hat{\mu} \circ \hat{\zeta} = id$, or phrased differently*

$$f(X) = \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot (\hat{\zeta} f)(|X|, Y),$$

for each function $f : \mathcal{P}(U) \to \mathbb{Z}$.

Proof. Expanding the right-hand side of the equation yields:

$$\sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot (\hat{\zeta} f)(|X|, Y) = \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot \sum_{Z \subseteq Y, |Z|=|X|} f(Z) = f(X).$$

The last equality holds since the only subset Z of X with $|Z| = |X|$ is X. □

The second condition on our way to fast subset convolution is also satisfied by the ranked transforms. Like for the zeta and Möbius transforms, we can derive algorithms for computing $\hat{\zeta}$ and $\hat{\mu}$ that require $O^*(2^n)$ arithmetic operations. We may call these algorithms *fast ranked zeta/Möbius transform*.

Theorem 3.20. *Let $f : \mathcal{P}(U) \to \mathbb{Z}$ and $g : [0..n] \times \mathcal{P}(U) \to \mathbb{Z}$ be two functions.*

a) Given all the values of f, the ranked zeta transform $\hat{\zeta}f$ can be computed in $O(2^n \cdot n^2)$ arithmetic operations.

b) Given all the values of g, the ranked Möbius transform $\hat{\mu}g$ can be computed in $O(2^n \cdot n^2)$ arithmetic operations.

The proof resembles the idea of Theorem 3.17. We interpret the given function over bitvectors and then apply a dynamic programming to fill a table that expresses the transform. Details can be found in Appendix A.1.2.

Now that we have an inversion formula and fast algorithms for computing the ranked transforms, it is left to show that the convolution can be represented by a simpler product. We will define the needed product in the upcoming section and finalize the fast subset convolution.

3.3.2 Fast Subset Convolution

The simpler operation that we use to represent the subset convolution is the so-called *ranked convolution.* It is defined over ranked zeta transforms.

Definition 3.21. Let $f, g : \mathcal{P}(U) \to \mathbb{Z}$ be two functions and $\hat{\zeta}f$, $\hat{\zeta}g$ their ranked zeta transforms. The *ranked convolution* of $\hat{\zeta}f$ and $\hat{\zeta}g$ is the function $\hat{\zeta}f \circledast \hat{\zeta}g : [0..n] \times \mathcal{P}(U) \to \mathbb{Z}$, defined by

$$(\hat{\zeta}f \circledast \hat{\zeta}g)(k, X) = \sum_{j=0}^{k} (\hat{\zeta}f)(j, X) \cdot (\hat{\zeta}g)(k - j, X),$$

where $k \in [0..n]$ is an integer and X is a subset of U.

Note that the summation in the ranked convolution is over the rank k instead of the partitions of X, like in the subset convolution. Therefore, it can be computed fast once the zeta transforms $\hat{\zeta}f$ and $\hat{\zeta}g$ are provided. By evaluating the ranked convolution separately for each $k \in [0..n]$ and $X \subseteq U$, we obtain all of its values via $O(2^n \cdot n^2)$ arithmetic operations. This makes the ranked convolution *simpler* than the subset convolution.

With the help of the ranked transforms, we can represent the subset convolution in terms of the ranked convolution. The following theorem formalizes the statement and is the last step to the fast subset convolution.

Theorem 3.22. *Let $f, g : \mathcal{P}(U) \to \mathbb{Z}$ be two functions. The subset convolution can be represented by $f * g = \hat{\mu}(\hat{\zeta} f \circledast \hat{\zeta} g)$. Phrased differently, for $X \subseteq U$ we have*

$$(f * g)(X) = \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot (\hat{\zeta} f \circledast \hat{\zeta} g)(|X|, Y).$$

Proof. First, we expand the right-hand side of the equation by plugging in the definition of the ranked convolution and the ranked zeta transform:

$$\begin{aligned}
&\sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot (\hat{\zeta} f \circledast \hat{\zeta} g)(|X|, Y) \\
&= \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot \sum_{j=0}^{|X|} (\hat{\zeta} f)(j, Y) \cdot (\hat{\zeta} g)(|X| - j, Y) \\
&= \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot \sum_{j=0}^{|X|} \Big(\sum_{A \subseteq Y, |A| = j} f(A) \Big) \cdot \Big(\sum_{B \subseteq Y, |B| = |X| - j} g(B) \Big) \\
&= \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot \sum_{j=0}^{|X|} \sum_{\substack{A, B \subseteq Y \\ |A| = j, |B| = |X| - j}} f(A) \cdot g(B). \qquad (3.1)
\end{aligned}$$

To simplify the resulting sum, we consider the coefficient of each summand. Let the pair (A, B) be fixed. The product $f(A) \cdot g(B)$ occurs in the sum whenever $|A| + |B| = |X|$. If so, it occurs once for each subset Y of X such that $A \cup B \subseteq Y$ with coefficient $(-1)^{|X \setminus Y|}$. Summing up, we obtain that the coefficient of $f(A) \cdot g(B)$ in the sum is given by

$$\sum_{Y \,:\, A \cup B \subseteq Y \subseteq X} (-1)^{|X \setminus Y|} = \sum_{i = |A \cup B|}^{|X|} (-1)^{|X| - i} \cdot \binom{|X| - |A \cup B|}{i - |A \cup B|}.$$

For the equation, we employed the index i as the size of Y. For the binomial coefficient, note that the sets Y with $A \cup B \subseteq Y \subseteq X$ are bijective to the subsets Y' of $X \setminus (A \cup B)$. We further reformulate the sum. First, by shifting the index i and then by invoking the binomial theorem. We obtain

$$\begin{aligned}
\sum_{i = |A \cup B|}^{|X|} (-1)^{|X| - i} \cdot \binom{|X| - |A \cup B|}{i - |A \cup B|} &= \sum_{i=0}^{|X|} (-1)^{|X| - |A \cup B| - i} \cdot \binom{|X| - |A \cup B|}{i} \\
&= (1 - 1)^{|X| - |A \cup B|} \\
&= \begin{cases} 1, \text{ if } X = A \cup B, \\ 0, \text{ otherwise.} \end{cases}
\end{aligned}$$

Hence, the coefficient of $f(A) \cdot g(B)$ for a pair (A, B) of subsets of X is 1 if and only if $|A| + |B| = |X|$ and $A \cup B = X$. But this means that A and B

must be disjoint and thus form a partition of X. We have $A \cup B = X$ and $A \cap B = \emptyset$. If A and B are not a partition of X, the corresponding coefficient is 0. Combining this information with (3.1) finally yields:

$$\sum_{Y \subseteq X} (-1)^{|X \setminus Y|} \cdot (\hat{\zeta} f \circledast \hat{\zeta} g)(|X|, Y) = \sum_{\substack{A,B \subseteq X \\ A \cup B = X \\ A \cap B = \emptyset}} f(A) \cdot g(B).$$

By Definition 3.10, the sum on the right-hand side is the subset convolution $f * g$ applied to X. This completes the proof. □

Theorem 3.22 offers a way of computing the subset convolution according to Figure 3.3. For given functions f and g, we first employ the fast ranked zeta transform to obtain $\hat{\zeta} f$ and $\hat{\zeta} g$. Then, we compute the ranked convolution $\hat{\zeta} f \circledast \hat{\zeta} g$. Finally, we apply the fast ranked Möbius transform and get $\hat{\mu}(\hat{\zeta} f \circledast \hat{\zeta} g)$ which is equal to $f * g$. Since each of these three steps requires $O(2^n \cdot n^2)$ arithmetic operations, the subset convolution can be computed in $O(2^n \cdot n^2)$ arithmetic operations as well. We refer to the resulting algorithm as *fast subset convolution*. This formally proves Theorem 3.12.

To obtain Corollary 3.13, assume that the functions f and g map into a set $\{-M, -M+1, \ldots, M-1, M\}$ where M is some integer bound. Then, the intermediary results that occur when computing the subset convolution $f * g$ are $O(n \cdot \log M)$-bit integers. Since fast subset convolution requires a total of $O(2^n \cdot n^2)$ ring operations over these, the result follows.

3.3.3 Counting Partitions and Colorings

We present some applications of fast subset convolution in the context of *counting problems*. Instead of usual decision problems that ask for the existence of a solution, counting problems require to compute the exact number of solutions. This makes these problems algorithmically harder to solve than their decision variants. For instance, instead of asking for the existence of a clique, the corresponding counting problem would ask for the number of cliques in a given graph. Fast subset convolution comes into picture when we need to count partitions of an underlying structure.

We state and prove a meta theorem which forms the basis of several algorithms for counting problems that incorporate fast subset convolution. For its general formulation, we need a new notion. Let U be a finite set and $\mathcal{F} \subseteq \mathcal{P}(U)$ a family of subsets over U. A *k-partition of U into sets of $\mathcal{F}$* is a k-partition $(A_1, \ldots, A_k)$ of U such that each set A_i belongs to $\mathcal{F}$. The meta theorem states that we can count the k-partitions of U into sets of $\mathcal{F}$ in time $O^*(2^n)$, where n is the size of U. We will see that many different counting problems can be traced back to counting such partitions. Hence, these problems can be solved by instantiating the meta theorem.

Theorem 3.23. *Let U be a finite set of size n and $\mathcal{F} \subseteq \mathcal{P}(U)$ a family of subsets over U. Moreover, let $k \geq 1$ be an integer. The number of k-partitions of U into sets of $\mathcal{F}$ can be computed in time $O(2^n \cdot n^2 \cdot \log k \cdot mult(n))$.*

Proof. We can express the number of k-partitions of U into sets of $\mathcal{F}$ in terms of a subset convolution. Let $\chi_{\mathcal{F}} : \mathcal{P}(U) \to \{0, 1\}$ be the characteristic function of $\mathcal{F}$. For each subset X of U, it guarantees that $\chi_{\mathcal{F}}(X) = 1$ if and only if $X \in \mathcal{F}$. Then, the number of k-partitions can be described as

$$\sum_{\substack{A_1 \cup \ldots \cup A_k = U \\ A_i \cap A_j = \emptyset, i \neq j}} \chi_{\mathcal{F}}(A_1) \cdots \chi_{\mathcal{F}}(A_k).$$

Note that the sum ranges over all k-partitions of U but counts only those the sets of which are all in $\mathcal{F}$. Hence, it provides the correct number.

With the subset convolution at hand, we can now apply Lemma 3.11 to the characteristic function $\chi_{\mathcal{F}}$ and obtain that

$$(*^k \chi_{\mathcal{F}})(U) = \sum_{\substack{A_1 \cup \ldots \cup A_k = U \\ A_i \cap A_j = \emptyset, i \neq j}} \chi_{\mathcal{F}}(A_1) \cdots \chi_{\mathcal{F}}(A_k).$$

The function $*^k \chi_{\mathcal{F}}$ is the convolution of k copies of the characteristic function: $*^k \chi_{\mathcal{F}} = \chi_{\mathcal{F}} * \cdots * \chi_{\mathcal{F}}$. We can compute it by iterating fast subset convolution. First, we compute $\chi_{\mathcal{F}} * \chi_{\mathcal{F}} = *^2 \chi_{\mathcal{F}}$. Then, we apply fast subset convolution again and obtain $*^3 \chi_{\mathcal{F}}$. This process can be iterated until $*^k \chi_{\mathcal{F}}$ is computed. Summing up, we apply fast subset convolution $k - 1$ times in total. According to Corollary 3.13, this requires $O(2^n \cdot n^2 \cdot k \cdot mult(n))$ time. To get the factor $\log k$ instead of k, we use *repeated squaring*. Formally, we compute the powers $*^{2^i} \chi_{\mathcal{F}}$ and do not need to iterate up to k. This requires only $O(\log k)$ applications of fast subset convolution and proves the time complexity as stated above. □

Counting Colorings We show how Theorem 3.23 can be employed to count colorings of graphs. This yields algorithms for the NP-complete problem of deciding whether a coloring exists for a given graph and for determining the chromatic number. But before we dive into the details, we introduce some notations around colorings. Let $G = (V, E)$ be an undirected graph and $k \in \mathbb{N}$ an integer. A *k-coloring* of G is a map $c : V \to [1..k]$ that assigns to each vertex $v \in V$ a *color* $c(v) \in [1..k]$ and that satisfies for all vertices $v, u \in V$:

$$\text{if there is an edge } \{v, u\} \in E \text{ then } c(v) \neq c(u).$$

Hence, a k-coloring assigns one out of k colors to each vertex of the graph and neighboring vertices are assigned distinct colors. An example is given in Figure 3.4. Given a graph, deciding the existence of a k-coloring is a well-known decision problem, shown to be NP-complete by Karp [229]. In his work,

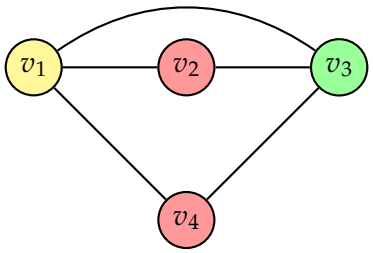


Figure 3.4: A 3-coloring of the underlying graph. Note that the three colors are needed as there is no coloring with two or less colors. The 3-coloring of the graph corresponds to a partition of the set of vertices into the three independent sets $\{v_1\}$ (yellow), $\{v_2, v_3\}$ (red), and $\{v_3\}$ (green).

the problem is called *Chromatic Number*. We use the name Colorable to avoid confusion with the problem that actually requires computing the chromatic number of a graph. We formalize the problem below.

Colorable
Input: A graph G and an integer $k \in \mathbb{N}$.
Question: Is there a k-coloring of G?

The following theorem shows the power of fast subset convolution. It is not only capable of solving Colorable, and by the way providing an FPT-algorithm for the parameterization Colorable(n), it can moreover count the precise number of possible k-colorings of all induced subgraphs at once.

Theorem 3.24. *Let G be a graph with n vertices and let $k \in \mathbb{N}$ be an integer. The number of k-colorings of all induced subgraphs of G can be computed in time $O^*(2^n)$.*

For the proof of Theorem 3.24, we need to represent colorings in terms of partitions. To this end, we need the notion of an *independent set*, the complement of a clique. Let $G = (V, E)$ be a graph. A subset $X \subseteq V$ of vertices is called *independent set* if no two vertices in X are adjacent. Formally this means that for each two vertices $v, u \in X$ there is no edge $\{v, u\} \in E$.

Observe that a k-coloring of G corresponds to a k-partition of V into independent sets, as indicated in Figure 3.4. In fact, given a k-coloring $c : V \to [1..k]$, we can derive such a partition by grouping those vertices together that are colored by the same color. Formally, we define $A_i = c^{-1}(i)$ for each $i \in [1..k]$. The vertices in A_i are not adjacent since neighboring vertices have distinct colors and hence, we obtain a k-partition $(A_1, \ldots, A_k)$ of V into independent sets. When starting with such a partition $(A_1, \ldots, A_k)$, we construct a k-coloring by coloring the vertices according to membership

in a set A_i. We define $c : V \to [1..k]$ with $c(v) = i$ if $v \in A_i$. Then, c is a coloring of G. Altogether, we obtain that there is a one-to-one correspondence between the k-colorings of G and the k-partitions of V into independent sets. This also holds for subsets of V. We can derive the following lemma.

Lemma 3.25. *Let $G = (V, E)$ be a graph, $X \subseteq V$, and $k \in \mathbb{N}$. The number of k-colorings of $G[X]$ is equal to the number of k-partitions of X into independent sets.*

Hence, counting colorings amounts to counting partitions into independent sets. We can achieve the latter by an application of Theorem 3.23. To this end, let $\mathcal{F}$ be the family of independent sets in V and $\chi_{\mathcal{F}}$ its characteristic function. It satisfies $\chi_{\mathcal{F}}(X) = 1$ if and only if X is an independent set. Note that the function can be computed in time $O^*(2^n)$. Theorem 3.23 computes all values of $*^k\chi_{\mathcal{F}}$ in time $O^*(2^n)$. Recall that

$$(*^k\chi_{\mathcal{F}})(X) = \sum_{\substack{A_1 \cup \ldots \cup A_k = X \\ A_i \cap A_j = \emptyset, i \neq j}} \chi_{\mathcal{F}}(A_1) \cdots \chi_{\mathcal{F}}(A_k),$$

which amounts to the number of k-partitions of X into independent sets. Hence, according to Lemma 3.25, the value $(*^k\chi_{\mathcal{F}})(X)$ is the number of k-colorings of $G[X]$. This completes the proof of Theorem 3.24.

In Appendix A.1.3, we show further applications of fast subset convolution. The technique can be used to determine *chromatic* and *domatic number* and to count *spanning forests* [170] of a graph, a #P-complete problem [219].

3.3.4 Cover Product

There are multiple variations of subset convolution that were designed to broaden the spectrum of applications. One change that offers applications in the realm of optimization problems is to consider the subset convolution over functions that map into the *integer min-sum semiring* or the *integer max-sum semiring* instead of $\mathbb{Z}$ with multiplication and addition. A famous example of this approach is the application to Steiner Tree [48]. But we can also vary the convolution operation itself. As we have seen in Set Cover, we are not always interested in a partition of the underlying structure but only in a cover. In such a scenario, subset convolution cannot be applied immediately to the problem. However, we can change the convolution operation to fit the application.

Definition 3.26. Let U be a finite set and $f, g : \mathcal{P}(U) \to \mathbb{Z}$ two functions, mapping subsets of U to integers. The *cover product* of f and g is a function $f *_c g : \mathcal{P}(U) \to \mathbb{Z}$ that maps each subset X of U to the sum

$$(f *_c g)(X) = \sum_{A \cup B = X} f(A) \cdot g(B).$$

The summation in the cover product is over all covers of the given set instead of all partitions. Hence, we cannot compute the cover product via fast subset convolution but we can borrow the main idea of the algorithm: use transforms to represent the cover product in terms of a simpler operation, according to Figure 3.3. In fact, the transforms that we need are the (ordinary) zeta and Möbius transform. The simpler operation is the *pointwise product* of functions. For the remaining section, we fix a finite set U with n elements.

Definition 3.27. Let $f, g : \mathcal{P}(U) \to \mathbb{Z}$ be two functions. The *pointwise product* of f and g is the function $f \cdot g : \mathcal{P}(U) \to \mathbb{Z}$ that maps subsets $X \subseteq U$ to

$$(f \cdot g)(X) = f(X) \cdot g(X).$$

The pointwise product of two functions can be computed fast, assuming the functions are provided. It only takes $O(2^n)$ arithmetic operations. Moreover, it can be used to represent the cover product as follows.

Lemma 3.28. *Let $f, g : \mathcal{P}(U) \to \mathbb{Z}$ be two functions. The cover product can be represented by the pointwise product. We have $f *_c g = \mu\big((\zeta f) \cdot (\zeta g)\big)$.*

A proof is given in Appendix A.1.4. The lemma yields an algorithm for computing the cover product. Given f and g, we compute ζf and ζg with the fast zeta transform from Theorem 3.17. Afterwards, we compute the pointwise product $(\zeta f) \cdot (\zeta g)$ and apply the fast Möbius transform to obtain $f *_c g$. Altogether, the algorithm needs $O(2^n \cdot n)$ arithmetic operations.

Theorem 3.29. *Let U be a set of size n and $f, g : \mathcal{P}(U) \to \mathbb{Z}$ two functions. The cover product $f *_c g$ can be computed in $O(2^n \cdot n)$ arithmetic operations.*

Similarly to the subset convolution, we can also obtain an upper bound on the worst-case time complexity for computing the cover product.

Corollary 3.30. *Let U be a set of size n, $M \in \mathbb{N}$ an integer, and $f, g : \mathcal{P}(U) \to \mathbb{Z}$ two functions mapping into the set $\{-M, -M+1, \ldots, M-1, M\}$. The cover product $f *_c g$ can be computed in time $O(2^n \cdot n \cdot mult(n \cdot \log M))$.*

As an application of the cover product, we present a faster algorithm for the problem Set Cover. Let $(U, \mathcal{F}, r)$ be an instance. In Section 3.2.2, we have obtained an algorithm for the problem based on dynamic programming. It has a worst-case time complexity of $O(2^n \cdot n \cdot m)$, where n is the size of the universe U and m is the size of the family $\mathcal{F}$. Like mentioned before, the running time depends on the factor m which might be quite large. An algorithm based on the cover product avoids such a dependence. Recall that r denotes the size of the cover that we are looking for. The result is as follows.

Theorem 3.31. Set Cover *can be solved in time $O(2^n \cdot n \cdot \log r \cdot mult(n))$.*

Proof. First, we compute the characteristic function $\chi_{\mathcal{F}}$ of the family $\mathcal{F}$. This takes $O(2^n)$ time. Then, we use the repeated squaring trick and run the algorithm from Theorem 3.29 $\log r$ many times to obtain the function $*_c^r \chi_{\mathcal{F}}$. Like for the subset convolution, this is defined via

$$*_c^r \chi_{\mathcal{F}} = \underbrace{\chi_{\mathcal{F}} *_c \cdots *_c \chi_{\mathcal{F}}}_{r \text{ times}}.$$

According to the complexity stated in Corollary 3.30, computing $*_c^r \chi_{\mathcal{F}}$ takes $O(2^n \cdot n \cdot \log r \cdot \mathit{mult}(n))$ time. We are interested in the value

$$(*_c^r \chi_{\mathcal{F}})(U) = \sum_{A_1 \cup \ldots \cup A_r = U} \chi_{\mathcal{F}}(A_1) \cdots \chi_{\mathcal{F}}(A_r).$$

Note that $(*_c^r \chi_{\mathcal{F}})(U) > 0$ if and only if there is a set cover of U into r sets of the family $\mathcal{F}$. Hence, we only need to look up the value of the function. This completes the proof with the complexity estimation as stated above. □

3.4 Kernelizations

Preprocessing is a central technique in the design of algorithms tailored to large-scaled practical instances. The goal is to reduce a given instance to its difficult part, called the *kernel*. The kernel is algorithmically expensive to solve but its size is usually much smaller than the size of the entire instance. To achieve such a reduction, a preprocessing uses light-weight, often polynomial-time, algorithms to erase the *simple parts* of an instance. Then, only the kernel is left and the application of an expensive decision procedure can be restricted to this much smaller part of the instance.

The idea of preprocessing has already appeared quite early in the literature [295] and is nowadays often built into tools that can handle large instances like Z3 [118] or CPLEX [215]. Despite its early discovery, a comprehensive theory being capable of formulating what a preprocessing is mathematically and when it is effective was missing. In fact, classical complexity theory did not provide sufficient tools to express or classify preprocessings [260]. This only changed with the beginning of parameterized complexity [140].

Taking a parameter for complexity measurements into account allows for a clear definition of what a preprocessing is. Namely, a process that runs in polynomial time and that transforms a given instance into a *smaller* instance the size of which is bounded by (a function in) the parameter. Furthermore, it has to be ensured that making the instance smaller does not change the result: the original is a yes-instance if and only if the smaller one is a yes-instance. A process satisfying both conditions is called a *kernelization*. The notion matches the intuition behind preprocessings: the *kernel* is the transformed instance and its bounded size is expressed via the parameter.

Kernelizations have evolved to a major tool in parameterized complexity. But their popularity is not only due to their nature of preprocessing. In fact, it turned our that kernelizations are a rather powerful notion of algorithms that allow for an alternative characterization of FPT: a problem has a kernelization if and only if it is fixed-parameter tractable [73]. Consequently, finding kernelizations is equivalent to finding FPT-algorithms, a central goal of parameterized complexity. This led to a whole zoo of results on kernelizations for particular problems. Famous examples are kernelizations for the problems Vertex Cover [27, 68], MAXSAT [258], Edge Clique Cover [194], and Hitting Set [6]. For more applications of the technique, we refer to the surveys on kernelization [196, 260], the standard books in parameterized complexity [112, 140, 167, 141] and the new book [172] solely on kernelization. In our applications in chapters 6 and 8, we mostly prove lower bounds for kernelizations showing that preprocessing has only a limited effect.

We formally define kernelizations. Like mentioned before, these are processes or algorithms that transform instances into *smaller* ones in polynomial time. Here, *smaller* means that the new instance is bounded by a parameter.

Definition 3.32. Let $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem. A *kernelization* or *kernel* of Q is an algorithm that, given an instance $(I, k) \in \Sigma^* \times \mathbb{N}$, computes in polynomial time an instance $(I', k') \in \Sigma^* \times \mathbb{N}$ such that

$$(I, k) \in Q \text{ if and only if } (I', k') \in Q,$$

and we have $|I'| + k' \leq h(k)$ for some computable function $h : \mathbb{N} \to \mathbb{N}$.

The function h bounds the size of the instances obtained from the kernelization. Therefore, we also call h the *size* of the kernel. Note that it only depends on the parameter k. This prevents a dependence of $|I'| + k'$ on the size of the input instance $|I|$. In order to reduce the size of the new instance as much as possible, it is important to find kernels which allow for a *small* function h. This constitutes the major quality feature of kernels. Accordingly, we consider kernels as good if h is a polynomial. In this case we also say that the problem *admits a polynomial kernel* or a *kernel of polynomial size*.

The definition of a kernel does not require that the parameter of the new instance is smaller than the original one: $k' \leq k$. Although the requirement seems plausible intuitively and many kernels linked to practical applications satisfy it, it would be too strict for theoretic considerations like the following. In its above form, kernelizations can be used to obtain FPT-algorithms and in fact, are strong enough to capture every problem in the class. Hence, kernelizations offer a new perspective on fixed-parameter tractability.

Lemma 3.33. *Let $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem. Then, Q is fixed-parameter tractable if and only if Q is decidable and admits some kernelization.*

Proof. Let Q be a decidable problem that admits a kernelization. Then, Q can be solved by an algorithm $\mathcal{A}$ that runs in time $f(n)$, where n is the size of an instance. Let the kernelization be called $\mathcal{K}$ and assume it transforms instances (I, k) to instances (I', k') such that $|I'| + k' \leq h(k)$.

We can design an FPT-algorithm for Q as follows. Given an instance (I, k), we run $\mathcal{K}$ and obtain, in polynomial time, an instance (I', k') the size of which is bounded by $h(k)$ and that satisfies: $(I, k) \in Q$ if and only if $(I', k') \in Q$. Now we decide the latter by applying $\mathcal{A}$ to (I', k'). This takes time

$$f(|I'| + k') \leq f(h(k)).$$

Altogether, the algorithm runs in time $O^*(f(h(k)))$. Since this is a function only dependent on the parameter k, the parameterized problem Q is FPT.

Now assume that Q is fixed-parameter tractable. Then, there is an algorithm $\mathcal{A}$ deciding whether an instance (I, k) belongs to Q in time $f(k) \cdot |I|^c$, where $c \in \mathbb{N}$ is a constant and f is a computable function.

We construct a kernelization for Q as follows. Given an instance (I, k), we first run $\mathcal{A}$ for at most $|I|^{c+1}$ many steps. If the algorithm yields an answer within that time bound, we can return the same. Otherwise, we return the unmodified instance (I, k) as a result of the kernelization. Since in the latter case $\mathcal{A}$ did not terminate within $|I|^{c+1}$ many steps, we obtain that

$$|I|^{c+1} < f(k) \cdot |I|^c.$$

Hence, the size of the instance can be bounded: $|I| + k \leq f(k) + k$. When setting $h(k) = f(k) + k$, we obtain a proper kernelization of size h. □

In the following, we consider kernelizations for two different problems: Vertex Cover and MAXSAT. Both kernelizations are based on simple reduction rules. But like for finding FPT-algorithms, there are various subtle techniques of algebraic, combinatoric, or even randomized nature to derive kernelizations. Applying these typically results in smaller and more efficient kernels. For further kernelization-related techniques, we refer to [172].

3.4.1 A Kernelization for Vertex Cover

A kernelization for Vertex Cover can be constructed by considering the degree of vertices in more detail. The idea goes back to Buss and Goldsmith [68] and is also called *Buss kernelization* in the literature [112]. It relies on two *reduction rules*. These can be understood as algorithms that reduce a given instance by discarding certain vertices due to their degree. Both rules can be implemented in polynomial time and the kernelization will be an exhaustive application of both. For their formal description, let (G, k) be an instance to Vertex Cover with graph $G = (V, E)$ and parameter k.

The first rule, called (VC-isolated), removes *isolated* vertices from G. A vertex $v \in V$ is *isolated* if it is not adjacent to any other vertex, $\deg(v) = 0$.

Such a vertex does not contribute to a vertex cover and hence, a vertex cover of G does not need to contain an isolated vertex. Therefore, removing v from the graph does not change whether we have a yes- or a no-instance. Phrased differently, (G, k) is a yes-instance if and only if $(G[V \setminus \{v\}], k)$ is one.

(VC-isolated) : Let (G, k) be given with $G = (V, E)$. If $v \in V$ is a vertex with $\deg(v) = 0$, output the new instance $(G[V \setminus \{v\}], k)$.

The observation behind the second rule, called (VC-maxdeg), is the following. Assume in the instance (G, k) there is a vertex $v \in V$ with $\deg(v) \geq k + 1$. Then, v is part of each vertex cover of size at most k. Otherwise, we would need at least $k + 1$ vertices, namely the neighborhood of v, to cover all edges outgoing of v. Since we cannot afford the latter, v is in the vertex cover. Hence, (G, k) is a yes-instance if and only if $(G[V \setminus \{v\}], k - 1)$ is one.

(VC-maxdeg) : Let (G, k) be given with $G = (V, E)$. If $v \in V$ is a vertex with $\deg(v) \geq k + 1$, output the new instance $(G[V \setminus \{v\}], k - 1)$.

Exhaustively applying the rules (VC-isolated) and (VC-maxdeg) removes all vertices of G the degree of which is 0 or at least $k + 1$. Hence, the degree of all vertices in the obtained instance lies in the interval $[1..k]$. In this case we can prove that each yes-instance is of bounded size. Then, no-instances can be identified by testing the size of the instance against the bound.

Lemma 3.34. *Let $G = (V, E)$ be a graph and $k \in \mathbb{N}$ be an integer such that $\deg(v) \in [1..k]$ for each $v \in V$. If G has a vertex cover of size at most k, then*

$$|V| \leq k^2 + k \text{ and } |E| \leq k^2.$$

Proof. Assume that $C \subseteq V$ is a vertex cover of G and $|C| \leq k$. Since there are no isolated vertices in V and C is a vertex cover, each $v \in V \setminus C$ is adjacent to a vertex from C. All vertices have degree bounded by k. Consequently, the vertices of C can be adjacent to at most $k \cdot |C|$ many vertices. Hence,

$$|V| - |C| = |V \setminus C| \leq k \cdot |C|.$$

Since C is of size at most k, we obtain $|V| \leq k^2 + k$.

The bound on the number of edges follows since each edge is covered by a vertex of C. But such a vertex can cover at most k edges: its degree is bounded by k. Hence, C can cover at most $k \cdot |C| \leq k^2$ edges. □

With Lemma 3.34, we can derive a kernelization for Vertex Cover. Given an instance, we exhaustively apply the rules (VC-isolated) and (VC-maxdeg). Note that both can be carried out in polynomial time. Let the resulting instance be (G, k) with $G = (V, E)$. Note that G and k satisfy $\deg(v) \in [1..k]$

for each $v \in V$. Then we test whether $|V| > k^2 + k$ or whether $|E| > k^2$. In either of the two cases, we can safely report a no-instance due to Lemma 3.34. Otherwise, we output the instance (G, k) of bounded size.

We obtain a kernel of quadratic size. The result is summarized in the following theorem, being more precise on the size of the kernel.

Theorem 3.35. *Vertex Cover admits a kernel with $k^2 + k$ vertices and k^2 edges.*

3.4.2 A Kernelization for Maximum Satisfiability

We consider MAXSAT, a generalization of SAT [229]. Given a formula φ in conjunctive normalform (CNF) and an integer $k \in \mathbb{N}$, MAXSAT asks whether there exists an assignment that satisfies at least k clauses of φ. The problem should not be confused with its optimization variant which determines the maximal number of satisfiable clauses. We state MAXSAT formally.

MAXSAT

Input: A formula φ in CNF and an integer $k \in \mathbb{N}$.
Question: Is there an assignment satisfying at least k clauses of φ?

Clearly, SAT is a special case of MAXSAT where all clauses have to be satisfied. This implies that MAXSAT is NP-complete. Consequently, the problem has been extensively studied from the viewpoint of parameterized complexity [31, 108, 258, 265, 285, 286]. We present a simple kernelizations of MAXSAT [285] which shows that MAXSAT(k) is fixed-parameter tractable.

Like for Vertex Cover above, we use two reduction rules to describe the kernelization. Let an instance (φ, k) of MAXSAT be given. The first rule removes all *trivial* clauses of φ. A clause is called *trivial* if it contains both literals, x and $\neg x$, of some variable x. Trivial clauses are satisfied by any assignment to the variables, making them obsolete when searching for one. Hence, they can be removed, as stated in the rule (MSAT-trivial). We use the notation $\varphi \setminus \{C\}$ to remove a clause C from the formula φ.

(MSAT-trivial) : Let (φ, k) be given and $C_1, \ldots, C_t$ the trivial clauses of φ. Output (φ_n, k_n), where $\varphi_n = \varphi \setminus \{C_1, \ldots, C_t\}$ and $k_n = k - t$.

Note that the rule (MSAT-trivial) safely reduces the instance. We have that the tuple (φ, k) is a yes-instance if and only if (φ_n, k_n) is a yes-instance.

Before we give the second rule of the kernelization, we need a lemma providing a sufficient criterion for yes-instances: if the number of *long* clauses in φ_n is too large, we can conclude that (φ_n, k_n) is a yes-instance. Here, a clause of φ_n is called *long* if it contains at least k_n literals.

Lemma 3.36. *If φ_n has at least k_n long clauses, (φ_n, k_n) is a yes-instance.*

Proof. Let $C_1, \ldots, C_\ell$ with $\ell \geq k_n$ be the long clauses of φ_n. We construct an assignment that satisfies at least k_n of the C_i and hence, (φ_n, k_n) is a yes-instance of MAXSAT. Each clause C_i is non-trivial and therefore contains at least k_n literals of distinct variables. We construct the aforementioned assignment by setting a literal in C_1 to true. Then, we go on with some literal in C_2 that belongs to a yet unset variable. The process is repeated with C_3 and so on. Note that in each step up to clause C_{k_n}, we will find an unset variable/literal. After reaching C_{k_n}, we set the remaining variables randomly. The assignment at least satisfies the clauses $C_1, \ldots, C_{k_n}$. □

With Lemma 3.36 in place, we can assume that φ_n has less than k_n long clauses. The next rule, called (MSAT-long), removes them completely.

(MSAT-long) : Let (φ_n, k_n) be given and $C_1, \ldots, C_\ell$ the long clauses of φ_n. Output (φ_s, k_s), where $\varphi_s = \varphi_n \setminus \{C_1, \ldots, C_\ell\}$ and $k_s = k_n - \ell$.

For the correctness of rule (MSAT-long), note the following. Any assignment to the variables that satisfies at least k_n clauses of φ_n also satisfies at least $k_n - \ell = k_s$ clauses of φ_s. For the other direction, assume an assignment is given which satisfies at least k_s clauses of φ_s. Such an assignment requires at most $k_n - \ell = k_s$ set variables. The remaining ones can be changed without altering the result. This means we have at least ℓ variables the value of which we can adjust. Now we can proceed as in Lemma 3.36 and construct an assignment which, in addition to the clauses of φ_s, also satisfies the ℓ long clauses of φ_n. Hence, the reduction rule is safe and we obtain that the instance $(\varphi_n, k_n) \in$ MAXSAT if and only if the instance $(\varphi_s, k_s) \in$ MAXSAT.

After removing all long clauses from the formula and reducing the instance further to (φ_s, k_s), we employ a second criterion which suffices to identify yes-instances. It is formulated in the following lemma.

Lemma 3.37. *If φ_s has at least $2 \cdot k_s$ clauses, (φ_s, k_s) is a yes-instance.*

Proof. Let v be any assignment to the variables. If v already satisfies at least k_s variables, we are done. Otherwise, let $k_s - d$ with $d > 0$ be the number of clauses satisfied by v. We consider the complement assignment $\bar{v}$. It flips the valuation of each variable. Then, $\bar{v}$ satisfies at least $2 \cdot k_s - (k_s - d) = k_s + d$ clauses of φ_s. Hence, (φ_s, k_s) is a yes-instance which finishes the proof. □

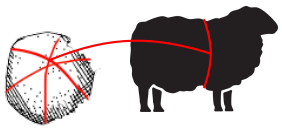
Finally we have all the tools to formulate the kernelization of MAXSAT. Given an instance (φ, k), we first apply the rule (MSAT-trivial) and obtain (φ_n, k_n). If the formula φ_n has more than k_n long clauses, we can report a yes-instance according to Lemma 3.36. Otherwise, we remove the long

clauses by applying (MSAT-long). The resulting instance is (φ_s, k_s). If φ_s contains at least $2 \cdot k_s$ clauses, we can again report a yes-instance, see Lemma 3.37. If not, we output (φ_s, k_s) as result of the kernelization.

Clearly, each step in the kernelization takes polynomial time. Moreover, the size of the output (φ_s, k_s) is quadratic. To see this, note that the formula φ_s has at most $2 \cdot k_s \leq 2 \cdot k$ clauses. Since none of these clauses is long, there are at most $2 \cdot k \cdot k_n \leq 2 \cdot k^2$ variables. We summarize the result.

Theorem 3.38. *MAXSAT admits a kernel with* $2 \cdot k$ *clauses and* $2 \cdot k^2$ *variables.*

4. A Theory of Hardness



Finding a fixed-parameter tractable algorithm for a given problem can be a challenging and sometimes also frustrating task. It may require lots of attempts and might consume plenty of hours. Moreover, comprehensive knowledge of various techniques for obtaining FPT-algorithms, like the ones presented in Chapter 3, is needed. There might be an FPT-algorithm for the problem relying on a more elaborate technique that the researcher just does not know about yet. However, despite all the techniques and the theory developed around it, occasionally it happens that an FPT-algorithm for a particular problem cannot be found at all. The parameterized problem Clique(k) is a prime example. Despite extensive effort put into the problem, none of the known FPT-techniques was able to significantly improve upon the simple brute force approach which runs in time $O^*(n^k)$. Consequently, no fixed-parameter tractable algorithm for Clique(k) is known until today.

All this fruitless effort raises the question on whether Clique(k) is fixed-parameter tractable at all. Maybe the problem is *intractable* and should be separated from the FPT-problems. In classical complexity theory, this is done all the time: when proving that a problem is NP-hard, it is separated from the tractable problems in P. Hence, when translating to parameterized complexity theory, we would like to have a class of intractable problems, similar to NP, that are provably not FPT. Then we only need to show that Clique(k) is hard for the intractability class.

Although the plan seems intuitive, there are three major concerns. (1) Unlike in classical complexity theory, one class of intractable problems will not suffice. In fact, considering the precise parameter-dependent complexity will give us many different degrees of intractability that will result in different intractability classes. (2) We cannot provide proofs for intractable problems that show non-membership in FPT. This is already known from classical complexity theory, where NP-hard problems are *unlikely* to be in P but NP $\neq$ P remains an open question. Hence, we need to establish a relative hardness theory which shows that intractable problems are unlikely to be FPT. (3) For a theory of (relative) hardness, we need a suitable notion of reductions. We cannot use polynomial-time computable reductions like in classical complexity theory. The reason is that we wish to separate NP-complete problems from each other. For instance, we would like to separate the intractable problem Clique(k) from the FPT-problem Vertex Cover(k). But among the classi-

cal decision problems, Clique and Vertex Cover, there are polynomial-time reductions in both directions. Hence, we cannot use the classical kind of reductions — we need a new notion of reductions. It should allow for reductions within FPT-problems and at the same time make it nearly impossible to find reductions from intractable problems to tractable ones.

All three concerns were resolved in the hardness theory developed by Downey and Fellows in the early days of parameterized complexity. They found a suitable notion of *parameterized reductions* [135, 136, 137] which is capable of handling parameters and which can separate parameterized problems. Moreover, they came up with a whole hierarchy of intractability classes, the so-called W-*hierarchy* [138, 139]. Nowadays the theory is standard in parameterized complexity and the complexity of many parameterized problems outside FPT has been classified accordingly. We use the hardness theory in chapters 6, 8, and 9 to classify the intractability of program verification tasks.

In the following, we give an introduction into the theory of Downey and Fellows by loosely following [112] and [141]. We start with *parameterized reductions* in Section 4.1. These form the basis of the relative hardness theory. Then, in Section 4.2, we will define the W-hierarchy, based on a suitable notion of *circuits*. Finally, we consider the first two levels of the hierarchy, the classes W[1] and W[2], in Section 4.3 and Section 4.4. Most of the natural intractable problems like Clique(k) appear within these two classes.

4.1 Parameterized Reductions

For classifying parameterized problems according to a hardness theory, we need a notion of reductions tailored to parameterized complexity. As mentioned above, classical polynomial-time reductions are not suited for this task. However, their central idea and properties are still important and lead the way to the new notion of reductions. Recall that a *polynomial-time computable reduction* is an algorithm that takes an instance I of a problem L, runs in polynomial time, and outputs an instance I' of a problem L' such that I is a yes-instance if and only if I' is a yes-instance. Polynomial-time reductions satisfy two rather useful properties. Namely, the class P is closed under this kind of reductions: if $L' \in \mathsf{P}$, we can deduce that L lies in P as well. Secondly, the reductions can be concatenated and form a transitive relation among problems.

We would like to define the new notion of reductions in such a way that both properties take over to the parameterized world: the class FPT should be closed under the new reductions and they should form a transitive relation among parameterized problems. To this end, we need to change the definition of a polynomial-time reduction in some aspects. The new reductions must handle parameterized problems, so a given instance (I, k) is now reduced to an instance (I', k'). Here, we have to be careful: the parameter k' should *remain a parameter* after the reduction. This means it is not allowed to depend

on the size of I but only on k. Further, we can be more liberal with the time to compute the instance (I', k'). It can now require any FPT-*time*.

Definition 4.1. Let $Q, Q' \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. A *parameterized reduction* from Q to Q' is an algorithm that, given an instance $(I, k) \in \Sigma^* \times \mathbb{N}$, computes an instance $(I', k') \in \Sigma^* \times \mathbb{N}$ such that

$$(I, k) \in Q \text{ if and only if } (I', k') \in Q',$$

and there are computable functions $f, g : \mathbb{N} \to \mathbb{N}$ with $k' \leq g(k)$ and the running time is bounded by $f(k) \cdot |I|^d$ where $d \in \mathbb{N}$ is a constant.

If there is a parameterized reduction from Q to Q', we also say that Q is *reducible* to Q' or that we can *reduce* Q to Q'. This should not be confused with polynomial-time reducible problems. Note that a classical polynomial-time reduction is not necessarily a parameterized reduction: it may run in FPT-time but there is no need to satisfy the bound on the parameter k'. Hence, some of the classical polynomial-time reductions among NP-complete problems are not parameterized reductions. Vice versa, a parameterized reduction may require more than polynomial time. Both notions are incomparable.

4.1.1 Features of Parameterized Reductions

Our next step is to show that parameterized reductions satisfy both properties that we hoped for: the class FPT is closed under the reductions and they are transitive. We begin with the former property, formalized in the upcoming lemma. As we can observe in the proof, it is crucial that a parameterized reduction requires a bound on the output parameter.

Lemma 4.2. *Let $Q, Q' \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems and assume there is a parameterized reduction from Q to Q'. If Q' is* FPT*, then Q is also* FPT*.*

Proof. The reduction takes an instance (I, k) of Q and transforms it into an instance (I', k') of Q' such that (I, k) is a yes-instance of Q if and only if (I', k') is a yes-instance of Q'. Moreover, we have that $k' \leq g(k)$ where g is a computable function and the reduction takes time at most $f(k) \cdot |I|^d$ for a computable function f and a constant $d \in \mathbb{N}$. Note that this also yields a bound on the size of the constructed instance I'. In fact we have that $|I'| \leq f(k) \cdot |I|^d$, as the size of the output is clearly bounded by the time an algorithm takes to compute the output. Since there is an FPT-algorithm for Q', we may assume it to run in time $h(k) \cdot n^c$ where h is a computable function, $c \in \mathbb{N}$ is a constant, and n describes the size of the given instance.

We obtain a fixed-parameter tractable algorithm for Q if we first run the reduction and then apply the FPT-algorithm for Q'. Clearly, the algorithm obtained this way is sound and complete — it works correctly. We only need

to reason about the time it requires to answer that an instance (I, k) is a yes-instance of Q or not. Since we first compute the instance (I', k') of Q' and then run the FPT-algorithm for Q' we obtain that the total running time is

$$\begin{aligned} f(k) \cdot |I|^d + h(k') \cdot |I'|^c &\leq f(k) \cdot |I|^d + h(g(k)) \cdot (f(k) \cdot |I|^d)^c \\ &\leq \big(f(k) + h(g(k)) \cdot f(k)\big) \cdot |I|^{d \cdot c}. \end{aligned}$$

Note that, formally, the first inequality requires that h is a monotone function. However, we can always assume this since h is an upper bound for the time complexity of an FPT-algorithm. The function $f(k) + h(g(k)) \cdot f(k)$ is computable and $d \cdot c$ is a constant. Hence, the described algorithm is fixed-parameter tractable and therefore, Q is FPT. □

We prove that parameterized reductions are transitive. In general, transitivity of reductions is important to set up a proper theory of hard and complete problems. Assume that Q is a problem that is hard (or complete) for a class of intractable problems. This means that each of the problems in the class admits a reduction to Q. If we want to show that a new problem Q' is hard for the class and our notion of reductions is transitive, it is enough to find a reduction from Q to Q'. By transitivity, each problem in the class then reduces to Q' and consequently, Q' is hard for the class as well.

Lemma 4.3. *Let $Q, R, S \subseteq \Sigma^* \times \mathbb{N}$ be parameterized problems. Assume there are parameterized reductions from Q to R and from R to S, then there is one from Q to S.*

Proof. Let $\mathcal{A}_1$ be reduction from Q to R and $\mathcal{A}_2$ be the reduction from R to S. By definition, algorithm $\mathcal{A}_1$ transforms an instance (I, k) to an instance (I', k') such that $(I, k) \in Q$ if and only if $(I', k') \in R$. Moreover, $\mathcal{A}_1$ runs in time $f_1(k) \cdot |I|^{d_1}$ and $k' \leq g_1(k)$, where f_1, g_1 are computable functions and $d_1 \in \mathbb{N}$ is a constant. Note that the size of the computed instance can be bounded by the running time of the reduction. We have $|I'| \leq f_1(k) \cdot |I|^{d_1}$. Similarly, $\mathcal{A}_2$ takes an instance (J, t) and reduces to an instance (J', t') such that $(J, t) \in R$ if and only if $(J', t') \in S$. The running time of $\mathcal{A}_2$ is bounded by $f_2(t) \cdot |J|^{d_2}$ and $t' \leq g_2(t)$ for some computable functions f_2, g_2 and a constant $d_2 \in \mathbb{N}$.

To obtain a parameterized reduction from Q to S, we concatenate both reductions, $\mathcal{A}_1$ and $\mathcal{A}_2$. Given an instance (I, k) of Q, we first apply $\mathcal{A}_1$ and obtain an instance (I', k') of R. Then we run $\mathcal{A}_2$ on this input and obtain an instance (I'', k'') of S. We immediately obtain the correctness of the construction: $(I, k) \in Q$ if and only if $(I'', k'') \in S$. Moreover, we can give a bound on the parameter k''. Note that $g_2(g_1(k))$ is computable. We have:

$$k'' \leq g_2(k') \leq g_2(g_1(k)).$$

It is left to determine the running time of the constructed reduction. Since we first run $\mathcal{A}_1$ followed by $\mathcal{A}_2$, the total running time can be bounded by

$$\begin{aligned} f_1(k) \cdot |I|^{d_1} + f_2(k') \cdot |I'|^{d_2} &\leq f_1(k) \cdot |I|^{d_1} + f_2(g_1(k)) \cdot |I'|^{d_2} \\ &\leq f_1(k) \cdot |I|^{d_1} + f_2(g_1(k)) \cdot (f_1(k) \cdot |I|^{d_1})^{d_2} \\ &\leq \big(f_1(k) + f_2(g_1(k)) \cdot f_1(k)\big) \cdot |I|^{d_1 \cdot d_2}. \end{aligned}$$

The function $f_1(k) + f_2(g_1(k)) \cdot f_1(k)$ is computable and $d_1 \cdot d_2 \in \mathbb{N}$ is a constant. Hence, the constructed algorithm runs in the required time and constitutes a proper parameterized reduction from Q to S. □

4.1.2 Reductions among Parameterized Problems

We consider some examples of parameterized reductions. These will become crucial in Sections 4.3 and 4.4 when we prove the completeness of certain intractable problems. The examples include reductions among problems that we already consider as intractable as well as a new *generic intractable problem*, called Short TM Acceptance which plays a central role in the hardness theory. Moreover, we give an example of a classical polynomial-time reduction that turns out not to be a parameterized reduction.

Clique and Independent Set To start with, we consider two simple examples: parameterized reductions between the problems Clique(k) and Independent Set(k). Recall that in an undirected graph $G = (V, E)$ a subset of the vertices $X \subseteq V$ is called an *independent set* if no two vertices of X are adjacent to each other. The *size* of an independent set X is its cardinality $|X|$. Given a graph and an integer $k \in \mathbb{N}$, the classical decision problem Independent Set asks for an independent set of size k. We state the problem.

Independent Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exists an independent of size k in G?

For reducing Clique(k) to Independent Set(k), we need the notion of *complement graphs*. Let $G = (V, E)$ be an undirected graph. The *complement graph* $\bar{G}$ of G takes the vertices V and complements the edges among them. This means it has an edge between two vertices if and only if the graph G has none. Formally, we have $\bar{G} = (V, \bar{E})$ with edges

$$\bar{E} = \{\{u, v\} \mid \{u, v\} \notin E\}.$$

Having the complement graph at hand, the following observation is simple. The graph G has a clique of size k if and only if the complement graph $\bar{G}$

has an independent set of size k. To see this, note that an independent set is the complement of a clique. From the observation we can construct a parameterized reduction. In fact, given an instance (G, k) of Clique, we construct the instance $(\bar{G}, k)$ of Independent Set. This takes polynomial time and the output parameter is just k and therefore bounded by k. Hence, all conditions of a parameterized reduction are met. Similarly, we can construct a reduction from Independent Set(k) to Clique(k). Note that by Lemma 4.2 this means that Clique(k) is FPT if and only if Independent Set(k) is FPT.

From Clique to Short Turing Machine Acceptance In the beginning of Chapter 4 we already conjectured that the problem Clique(k) seems to be intractable and that it is probably not FPT. However, the problem itself is of graph theoretical nature which makes it easy to understand and hard to imagine that Clique(k) can capture intractable or complex computations. We define a problem where this seems more plausible. To this end, we need to go a step back to classical complexity theory.

In complexity theory, the robust complexity classes were originally defined in terms of Turing Machines. In particular, the class NP is the family of all problems that can be solved by a nondeterministic Turing Machine which runs in polynomial time. We can extract a generic NP-hard problem from the definition, namely the corresponding bounded halting problem: given a Turing Machine M which takes at most polynomially many steps and an input x, decide whether M accepts x. What makes the problem so hard is that Turing Machines can capture all kinds of complex and intractable computations. Especially those which appear in the *hardest* problems of NP. In fact, Turing Machines were designed for this task.

So far, we did not use Turing Machines in the context of parameterized complexity. All considered problems were either of combinatorial or graph theoretical nature. However, Turing Machines are a good provider of hard problems, so we should use them to find a generic intractable problem of which we believe that it is not FPT. Considering the above generic NP-hard problem, we can consider a parameterized variant of it which bounds the number of steps. The problem is called Short TM Acceptance:

Short TM Acceptance

Input: A Turing Machine M, an input x, and an integer $k \in \mathbb{N}$.

Question: Does M accept x in at most k steps?

The problem was formulated by Cai, Chen, Downey, and Fellows in [74] and became more important for hardness theory later, due to Cesati [78, 79]. Because of the involved Turing Machine, the problem is capable of capturing intractable computations, like finding cliques, which makes it a generic

intractable problem in parameterized complexity. Note that a brute force approach for the problem simulates the Turing Machine for k steps and hence runs in time $O^*(n^k)$ where n is the size of the input. The reason is that a computation of the Turing Machine can branch into n different states each step. It seems impossible to improve upon the simple approach. In fact, an improvement would mean that we found a new algorithmic technique which can simulate Turing Machines faster — something which seems not plausible. Hence, it is unlikely that Short TM Acceptance is fixed-parameter tractable.

To support the claim, we show that Clique(k) can be reduced to Short TM Acceptance(k). Hence, the problem is *harder* than that of finding cliques, something that we already conjectured to be intractable. Note that clearly, Turing Machines can find cliques. However, in this reduction we need to ensure that the parameter k is preserved to obtain a proper parameterized reduction.

Theorem 4.4. *The problem Clique(k) reduces to Short TM Acceptance(k).*

Proof. Let (G, k) be an instance of Clique with $G = (V, E)$. We construct a Turing Machine M in polynomial time that reaches an accepting state within $g(k)$ steps, where g is computable, if and only if G contains a clique of size k.

The machine M proceeds as follows. Initially, its tape is empty, so the input x is ε. Then it runs in two phases. In the first phase, it nondeterministically guesses k vertices of V and writes these onto the tape, into the first k cells. Let these be $v_1, \ldots, v_k \in V$. This takes $O(k)$ many steps. The selected vertices are a candidate for the clique. In the second phase, M verifies that the guessed vertices indeed form a clique in G. To this end, it performs $\binom{k}{2}$ many checks. Each of the checks tests whether two guessed vertices are actually adjacent in the graph. If M performs the test between v_i and v_j, it takes $O(k)$ many steps and can only proceed if there is an edge $\{v_i, v_j\} \in E$. In this case, a corresponding transition exists. Otherwise, the computation gets stuck. As soon as all checks were successful, M enters an accepting state.

By construction, we have that G contains a clique of size k if and only if M guesses the correct vertices and all checks are successful. Since the latter takes $O(k + k \cdot \binom{k}{2})$ many steps, we can derive that G has a k-clique if and only if M accepts in at most $O(k + k \cdot \binom{k}{2})$ many steps. Note that M can be constructed in polynomial time and that the steps can be bounded by a function in k. Therefore, we have constructed a parameterized reduction. □

Dominating Set and Set Cover Recall that in the problem Set Cover, we are given a family $\mathcal{F}$ of subsets over a universe U and an integer $r \in \mathbb{N}$. The problem asks whether there are r sets in $\mathcal{F}$ that cover U. While we have already seen in Sections 3.2.2 and 3.3.4 that Set Cover(n) is FPT, where $n = |U|$, it is not clear at this point whether this also holds for the parameterization Set Cover(r). In the following, we do not resolve the question, the reader has

to be patient until Section 4.4, but we identify a different problem from graph theory that is equally hard. Namely, finding a dominating set.

Given an undirected graph $G = (V, E)$, recall that a dominating set is a subset X of V such that each vertex outside of X is adjacent to a vertex in X. The *size* of a dominating set X is its cardinality $|X|$. Examples can be found in Figure A.1. The search for a dominating set of a given size in a given graph is formalized in the problem Dominating Set. We state it below.

Dominating Set
Input: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.
Question: Does there exist a dominating set of G of size at most k?

We show that Set Cover(r) and Dominating Set(k) are equally hard problems. To this end, we establish parameterized reductions in either directions. First, we go from Dominating Set(k) to Set Cover(r).

Theorem 4.5. *The problem Dominating Set(k) reduces to Set Cover(r).*

Proof. Let (G, k) be an instance of Dominating Set with $G = (V, E)$. The parameterized reduction is based on the following observation: a subset $X \subseteq V$ is dominating if and only if each vertex in V either belongs to X or lies in the neighborhood of a vertex in X. The latter fits the nature of Set Cover since we can formulate it as a cover problem over the universe V.

To construct a corresponding instance $(\mathcal{F}, r)$ of Set Cover over the universe U, we first set $U = V$ and $r = k$. For each vertex $v \in V$, we add the set $S_v = N(v) \cup \{v\}$, the union of the neighborhood of v and v, to the family $\mathcal{F}$. Clearly the construction takes polynomial time and the parameter r is bounded by k. Hence, it is left to show that the reduction is correct.

Suppose there is a dominating set $X \subseteq V$ in G with $|X| \leq k$. Following the above observation, the sets S_v with $v \in X$ cover the universe U. Indeed we have $U = \bigcup_{v \in X} S_v$. The cover consists of at most $k = r$ sets of $\mathcal{F}$. The other direction is similar. If we have a cover of at most $k = r$ sets $S_{v_1}, \ldots, S_{v_\ell}$, we collect the vertices $v_1, \ldots, v_\ell$ in a set X. Then, X is a dominating set in G. □

The second parameterized reduction takes the inverse direction, from Set Cover(r) to Dominating Set(k). The following theorem formalizes the reduction. A detailed proof can be found in Appendix A.2.1.

Theorem 4.6. *The problem Set Cover(r) reduces to Dominating Set(k).*

In Section 4.4 it turns out that both considered problems, Set Cover(r) and Dominating Set(k), are actually intractable. In fact, they are even *more intractable* than Clique(k) or Short TM Acceptance(k).

A Negative Example A good start to find parameterized reductions is to consider classical polynomial-time reductions from complexity theory. In fact, many of these reductions are actually parameterized and we do not have to provide a new parameterized reduction among the involved problems. But this is not always the case. We show a negative example, a classical polynomial-time reduction that is not parameterized and which could not be replaced by a corresponding parameterized reduction until today. To this end, we revisit the problems Independent Set and Vertex Cover.

Let $G = (V, E)$ be a graph with n vertices and $k \in \mathbb{N}$ an integer. Observe the following: G has an independent set of size k if and only if G has a vertex cover of size $n - k$. Indeed, if G has an independent set X of size k, consider the complement $V \setminus X$. It is actually a vertex cover of size $n - k$. Vice versa, if G has a vertex cover of size $n - k$, the complement is an independent set of size k. Hence, by transforming an instance (G, k) of Independent Set into an instance $(G, n - k)$ of Vertex Cover, we obtain a polynomial-time reduction.

However, the constructed reduction is not parameterized. Note that the constructed parameter is $n-k$, a value which cannot be bounded by a function only dependent on k. Until today, there is no parameterized reduction from Independent Set(k) to Vertex Cover(k) known. In fact, it is believed that the existence of such a reduction is highly unlikely since Independent Set(k) is considered intractable while Vertex Cover(k) is FPT.

4.2 The W-hierarchy

In classical complexity theory, all NP-complete problems are equally hard. Due to the definition of completeness, there are polynomial-time reductions between each two of them. When employing parameterized reductions, this is no longer true. For instance, there is a trivial parameterized reduction from Vertex Cover(k) to Clique(k)[2] but no reduction into the other direction is known. Hence, there is a separation of the parameterized problems into those problems that are FPT and those that are intractable and hence, unlikely to be FPT. The latter problems could be grouped together into a single intractability class. However, a single class for intractable problems does not suffice for the same reason: the parameterized reductions are too fine. For instance, there is a parameterized reduction from the intractable problem Independent Set(k) to the intractable problem Dominating Set(k) [112] but none into the reverse direction. This means that we have to separate the intractable problems into the *level one intractable* problems and the *level two intractable* ones. We would group Independent Set(k) and the equally hard problem Clique(k) into the first level and Dominating Set(k) and the equally

[2]Note that in fact, each problem in FPT has a parameterized reduction to Clique(k) since the reduction itself can solve the problem and then just output a trivial yes- or no-instance.

hard Set Cover(k) into the second level. This process continues and one can construct parameterized problems for each level of intractability.

When developing a hardness theory, Downey and Fellows were well-aware of this behavior. To model it, they came up with the W-hierarchy, a hierarchy of intractability classes with one class per level. It seems that they found the right tool to classify the parameterized complexity of problems: almost all intractable problems considered until today can be grouped into a level of the hierarchy [141]. The definition of the hierarchy is based on *circuits*. In fact, the levels are defined by a generic complete problem, namely the *weighted satisfiability problem over circuits*. Different levels of intractability are then obtained by allowing the circuits to admit certain shapes.

4.2.1 Circuits

In order to define the W-hierarchy, we need the notion of *Boolean circuits*, a generalization of Boolean formulas. It should be remarked at this point that we do not give a thorough introduction into *Circuit Complexity* in the upcoming section. For an introduction to the topic, we refer the reader to [18, 303]. However, all needed notions around circuits are introduced and knowledge in circuit complexity is not needed. In fact, after a few basic definitions we quickly return to parameterized complexity.

Circuits are defined in terms of graphs. But not every term from graph theory is transferred to circuits. Some notions are changed so that they suit the nature of circuits more properly. In the following, a *node* or *gate* are just different names for a vertex. The *fan-in* of a node/gate denotes its indegree, the number of incoming edges. Similarly, its *fan-out* is the outdegree.

Definition 4.7. A *Boolean circuit* is a directed acyclic graph $C = (V, E)$ with a labeling $\lambda : V \to \{in, out, \vee, \wedge, \neg\}$ such that

(1) if $v \in V$ has fan-in 0 then $\lambda(v) = in$. These nodes are called *input nodes*,

(2) if $v \in V$ has fan-in 1 then $\lambda(v) = \neg$. These nodes are called *negation gates*,

(3) if $v \in V$ has fan-in at least 2 then $\lambda(v)$ is either $\vee$ or $\wedge$. The node is then called *or gate* or *and gate*, corresponding to its label,

(4) there is exactly one node $v \in V$ with fan-out 0 that, at the same time, is labeled by $\lambda(v) = out$. It is called the *output node*.

Given a circuit, its *depth* is the maximal length of a path from an input node to the output node. We give an illustration of a circuit in Figure 4.1.

To each input node of a circuit C, we can assign the value 0 or 1, like for variables in a Boolean formula. This results in an *assignment* of C. The circuit can then be evaluated under the assignment. To this end, we propagate values according to the labeling of the gates: an *and* gate v for instance requires that

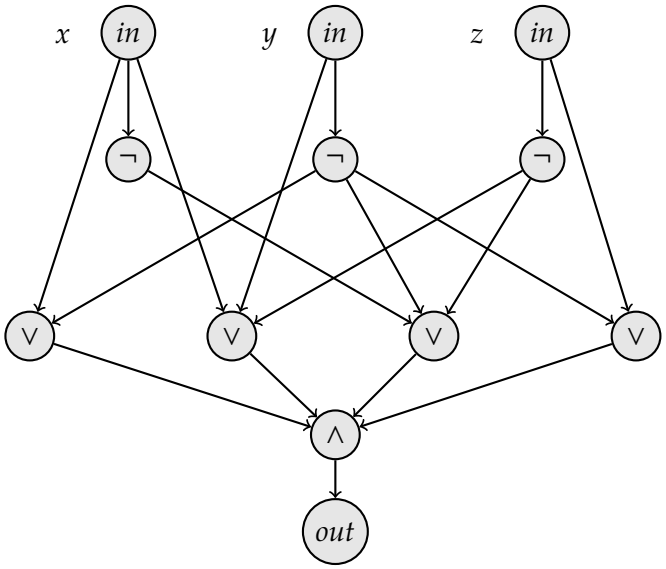


Figure 4.1: A circuit C with three inputs, seen as variables x, y, z. The labeling of C is written into each node/gate, its depth is 4. The circuit represents the Boolean formula $(x \vee \neg y) \wedge (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg y \vee z)$.

the current value of all nodes/gates pointing to v is 1. Only then it will output 1. If the value of the output node is 1, we have a *satisfying assignment*.

Clearly, given a circuit and an assignment, evaluating the former can be done in polynomial time. Hence, checking whether the assignment is satisfying is actually a problem in P. However, it is much more difficult to find a satisfying assignment when only the circuit is given. We formulate the task in the decision problem Circuit SAT, very similar to the classical problem SAT. Note that instead of a Boolean formula, the input is a circuit.

Circuit SAT

Input: A Boolean circuit C.

Question: Does there exists a satisfying assignment for C?

The problem Circuit SAT is NP-complete. Membership is clear. For hardness, note that Boolean formulas are circuits. This allows for a polynomial-time reduction from SAT to Circuit SAT with a construction like in Figure 4.1.

4.2.2 Classes of Intractability

The problem Circuit SAT plays a central role in the definition of complexity classes for each level of intractability. But for comparing the complexity

of parameterized problems with that of Circuit SAT, we need a reasonable parameterized variant of the latter. To this end, we consider assignments of a certain *weight*. Here, a *weight* of an assignment is the number of input nodes that are assigned the value 1. Now it is possible to ask for a satisfying assignment of weight k, where $k \in \mathbb{N}$ is a given integer. The corresponding decision problem is called *Weighted Circuit Satisfiability* or short WCS.

WCS	
Input:	A Boolean circuit C and an integer $k \in \mathbb{N}$.
Question:	Is there a satisfying assignment for C of weight k?

The canonical parameterization that we consider is WCS(k). When comparing its complexity with that of other parameterized problems, we notice that WCS(k) seems to be among the hardest or most intractable problems that we have considered. In fact, the level one intractable problem Clique(k) and the level two intractable problem Set Cover(r) both reduce to WCS(k) revealing a level of intractability that we have not seen so far.

Lemma 4.8. *The problems Clique(k) and Set Cover(r) reduce to WCS(k).*

Proof. We begin with the reduction from Clique(k) to WCS(k). Let (G, k) be an instance of the former with graph $G = (V, E)$. We can formulate a circuit that mimics the conditions required for a clique. In fact, a clique in G only contains adjacent vertices, it is not allowed that between two vertices there is no edge. This can be encoded into a circuit. Assume that v and w are vertices in V that are not adjacent, then we should not select both: $\neg(v \wedge w) = \neg v \vee \neg w$.

To construct a proper circuit C from the idea, we have to understand the vertices as inputs to C. Then, we add *negation* gates below each input to obtain the terms $\neg v$ for each $v \in V$. For each non-edge, for each $\{v, w\} \notin E$, we add an *or* gate with inputs from $\neg v$ and $\neg w$. This models the fact that v and w should not be contained together in a clique. The output of these is then led into one large *and* gate and finally to the output node.

An illustration of the construction is given in Figure 4.2. We immediately obtain that G contains a clique of size k if and only if C has a satisfying assignment of weight k. Moreover, constructing the circuit C takes polynomial time, hence we have a proper parameterized reduction.

We reduce Set Cover(r) to WCS(k). Let the tuple $(\mathcal{F}, r)$ be an instance of Set Cover where $\mathcal{F}$ is a family of sets over the universe U. The goal is to cover U by r sets from $\mathcal{F}$. This can be modeled by the following circuit C.

For each set S from $\mathcal{F}$, we have an input node in C. In an assignment we will then choose a candidate for a set cover. To test whether the chosen candidate is indeed a cover, there is an *or* gate for each element $u \in U$ on a second layer. The gate receives an input from each input node S with $u \in S$.

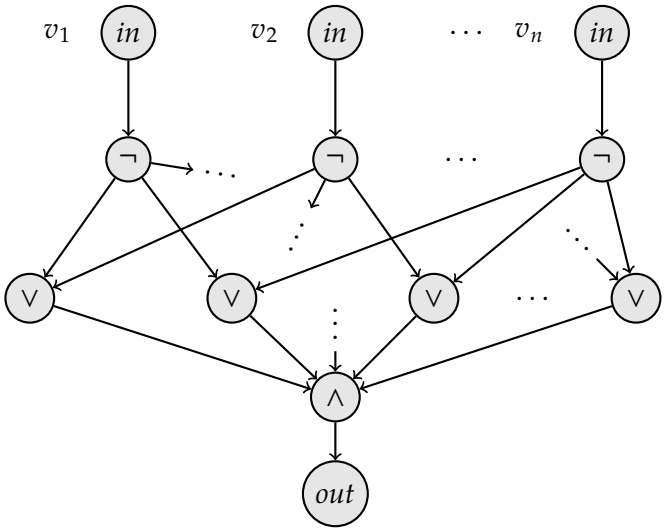


Figure 4.2: The circuit C for modeling cliques in the graph G. For each vertex in the set $V = \{v_1, \ldots, v_n\}$ it has an input node. The input is negated and for each non-edge we add an *or* gate. In the figure, this means that there are for instance no edges of the form $\{v_1, v_2\}$, $\{v_1, v_n\}$, and $\{v_2, v_n\}$. Note that the *and* gate in C is *larger* than the other gates. While the fan-in of each other gate is bounded by a constant, namely 2, the *and* gate gets its inputs from each *or* gate. Hence, its fan-in may grow with the size of the input.

This means the gate ensures that u is covered by the chosen candidate. In the end, there is a large *and* gate obtaining inputs from each of the *or* gates. It checks whether all elements from U are covered.

The construction is illustrated in Figure 4.3. It clearly detects proper set covers. We obtain that there is a set cover of U into r sets of $\mathcal{F}$ if and only if C has a satisfying assignment of weight r. Since the construction of C takes polynomial time, we have a parameterized reduction. □

The lemma shows that WCS(k) is not convenient when searching for a generic complete problem for each level of intractability. In fact, it seems that WCS(k) lies above each level — it is too hard. Hence, we need to find a way to restrict its intractability. At best in some parameter t such that the t-restriction of WCS(k) is complete for the level t intractable problems.

One way to make WCS easier is to restrict the input to a certain class of circuits. A popular class in circuit complexity is to consider circuits of bounded depth. However, for us this will not work. Consider the reductions from Clique(k) and Set Cover(r) to WCS(k) given in Lemma 4.8. Both construct circuits that are of constant depth, see Figure 4.2 and 4.3. Hence, restricting WCS to circuits of constant depth does not affect the intractability.

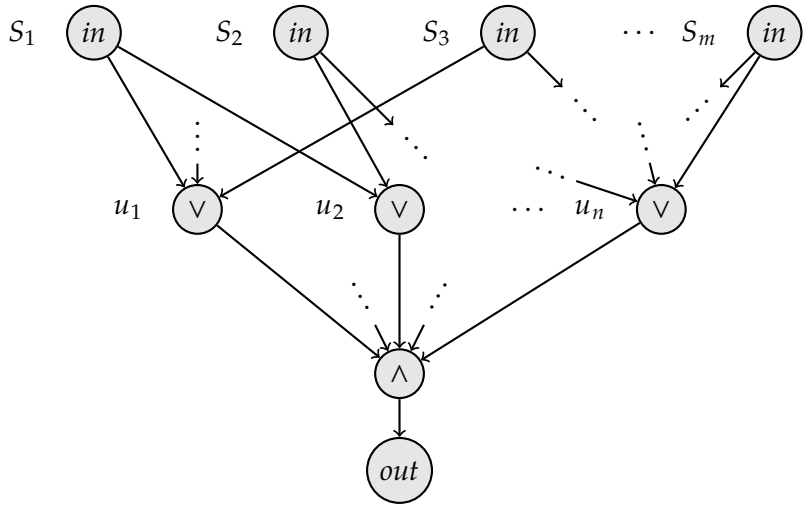


Figure 4.3: The circuit C detects set covers of U into r sets of $\mathcal{F}$. It has an input node for each set in $\mathcal{F} = \{S_1, \ldots, S_m\}$. For each element in $U = \{u_1, \ldots, u_n\}$, the circuit has an *or* gate. There is an edge from S_i to u_j if $u_j \in S_i$. From the figure, we can for instance read off that $u_1 \in S_1$ and $u_1 \in S_3$. At the bottom, there is an *and* gate receiving inputs from each *or* gate. Note that neither the fan-in of the *or* gates nor the fan-in of the *and* gate are bounded by 2. Both depend on parts of the input, namely on m and n.

The restriction that we need to employ is trickier. When reconsidering the reduction from Clique(k) to WCS(k) once again, see Figure 4.2, something in the structure of the constructed circuit stands out. Each gate has a fan-in bounded by 2, except for the *and* gate. The gate is *larger* than the others and its fan-in can only be bounded by the size of the input. When going from an input node to the output node we pass exactly one large gate. We can make a similar observation in the reduction from Set Cover(r) to WCS(k), see Figure 4.3. Here, we only have large gates: the *or* gates as well as the *and* gate do not have a fan-in bounded by 2. When we traverse from an input node to the output node in this circuit, we will see at most two large gates. This constitutes a difference between the reductions. When counting large gates seen along an input-output path, the former reduction is simpler compared to the latter one. The observation is formalized in the notion of *weft*.

Definition 4.9. Let C be a Boolean circuit. A gate of C is called *large* if its fan-in is larger than 2. Consequently, a *small* gate is one which has fan-in at most 2. The *weft* of a circuit C is the maximal number of large gates seen on a path from an input node to the output node of C.

To give an example, consider the circuits shown in Figure 4.2 and 4.3. The former is of weft 1, the latter of weft 2. It is important to note that the constant

2 chosen as bound for the fan-in of small gates is not crucial for our purpose. In fact, we show later that it could be replaced by any other constant.

The definition of weft approves our observation from above. It seems that level one intractable problems like Clique(k) require circuits of weft 1 when we reduce them to WCS(k). Similarly, for reducing from level two intractable problems like Set Cover(r) we need circuits of weft 2. So if we consider WCS(k) over circuits of a certain weft, we obtain generic problems for each level of intractability. To this end, let $\mathcal{C}(t, d)$ denote the family of Boolean circuits that are of weft at most t and depth at most d, where $t, d \in \mathbb{N}$ are constants. The problem WCS[$\mathcal{C}(t, d)$] is the restriction of WCS to inputs from $\mathcal{C}(t, d)$. We can now define the W-hierarchy of intractability classes.

Definition 4.10. Let $t \geq 1$ be an integer. The complexity class $\mathsf{W}[t]$ contains all problems which can be reduced, by a parameterized reduction, to the weighted circuit satisfiability problem WCS[$\mathcal{C}(t, d)$] for a $d \in \mathbb{N}$ [3].

From the definition and Lemma 4.8 it immediately follows that the problem Clique(k) lies in the class $\mathsf{W}[1]$ and Set Cover(r) is a member of the class $\mathsf{W}[2]$. Moreover, we obtain inclusions among the classes. We have

$$\mathsf{W}[t] \subseteq \mathsf{W}[t+1]$$

for each integer $t \geq 1$. Note that a circuit of weft t is also one of weft $t + 1$.

The W-hierarchy is defined in terms of the weft. Recall that the latter was defined by counting large gates with fan-in greater than 2. Small gates have fan-in at most 2. For some applications it would be handy if we could use another constant than 2 and allow small gates to have a fan-in bounded by some fixed constant. At first sight this might change the definition of the W-hierarchy. But the following lemma shows that we can replace 2 by any other constant of choice. We provide a proof in Appendix A.2.2.

Lemma 4.11. *Let $t \geq 1$ be an integer. The definition of $\mathsf{W}[t]$ is independent of the chosen constant which bounds the fan-in of small gates.*

The restrictions of WCS to circuits of bounded weft yields a class for each level of intractability. But also the problem itself can be used to define an intractability class. We have seen that WCS(k) is among the most intractable problems considered so far. Hence, its corresponding complexity class should reflect this behavior and lie above the whole W-hierarchy.

Definition 4.12. The complexity class $\mathsf{W}[P]$ contains all problems which can be reduced, by a parameterized reduction, to the problem WCS(k).

[3] Technically, we require a parameterized reduction to the parameterized problem WCS[$\mathcal{C}(t, d)$](k). Note that we dropped the parameter k for better readability.

Clearly, each level of the W-hierarchy is contained in W[P]. The class itself is contained in XP. To see the inclusion, note that WCS can be solved in time $O^*(n^k)$ where n is the size of the circuit. We iterate over all assignments of weight k and evaluate the circuit correspondingly. This implies that WCS(k) is slicewise polynomial, it lies in the class XP. Hence, W[P] $\subseteq$ XP. The complete hierarchy, in its full beauty, is given below:

$$\mathsf{FPT} \subseteq \mathsf{W}[1] \subseteq \mathsf{W}[2] \subseteq \cdots \subseteq \mathsf{W}[t] \subseteq \cdots \subseteq \mathsf{W}[P] \subseteq \mathsf{XP}.$$

There are many more intractability classes and a lot of structure theory around them. But at this point we do not investigate further structural results anymore. Instead, we turn to the first two levels of the W-hierarchy and refer to [141] for the reader interested in more structure theory. The reason is that in our research the main goal is to find FPT-algorithms. However, sometimes the existence of such an algorithm seems unlikely. In this case we can provide evidence for the intractability of a problem by proving hardness for a class W[t] of the hierarchy. Since most *natural* intractable problems are hard either for W[1] or W[2], taking a closer look at these two classes will provide us with a set of problems sufficient to prove hardness whenever we need it.

4.3 The First Level

We consider the first level of the W-hierarchy, the class W[1]. Our goal is to identify intractable problems that are complete for the class. To this end, we first state a parameterized variant of the Cook-Levin theorem [105] showing that W[1] behaves quite similar to NP. The FPT *versus* W[1] *question* can then be seen as the parameterized P *versus* NP *question*. In fact, the theorem will give us a characterization of W[1] in terms of Turing machines and shows that suitable variants of SAT are complete for the class. With the theorem at hand we can then provide more complete problems in a second step.

4.3.1 Parameterized Cook-Levin

The famous theorem of Cook and Levin shows that SAT is an NP-complete problem. While developing the hardness theory for parameterized complexity, Downey and Fellows [141] found a variant of the theorem which adapts to the first level of intractability. To state it, we first need to consider a suitable parameterized version of SAT which fits the nature of W[1].

Recall that the class W[1] is defined in terms of the weighted circuit satisfiability problem WCS, restricted to circuits of weft 1. The input to plain SAT is a Boolean formula in CNF, a special form of a circuit. Indeed, a formula in ℓ-CNF where each clause has at most ℓ literals, is a simple weft-1 circuit: there are inputs for each variable, *negation* gates, an *or* gate for each clause with inputs from its literals, and one large *and* gate. Hence, the weighted

satisfiability problem for CNF-formulas is actually a restriction of the WCS-problem which defines W[1] and therefore a good candidate for a suitable parameterization of SAT. Let $\ell \geq 2$, we formalize the problem below:

WEIGHTED ℓ-SAT
Input: A formula φ in ℓ-CNF and an integer $k \in \mathbb{N}$.
Question: Is there a satisfying assignment for φ of weight k?

We consider the parameterization WEIGHTED ℓ-SAT(k). The problem clearly lies in the class W[1] since, as explained above, each instance to WEIGHTED ℓ-SAT(k) is actually a simple weft-1 circuit. The more complicated part is to show that the problem is W[1]-hard. Technically, we would need to prove that an arbitrary circuit of weft 1 can be reduced to the special form of an ℓ-CNF. Downey and Fellows [141] came up with some non-trivial structural results on circuits of weft 1 that prove essentially this. We skip these results and proofs due to their length and complexity and rely on one of their main consequences: WEIGHTED ℓ-SAT(k) is indeed W[1]-hard.

Downey and Fellows even went one step further. They restricted the problem WEIGHTED ℓ-SAT to less expressive formulas without losing the hardness for W[1]. This will become rather useful in Section 4.3.2 when we prove the W[1]-hardness of some other problems. To state the restriction, let φ be a Boolean formula. We say that φ is in *antimonotone* ℓ-CNF if it is in ℓ-CNF but each occurring literal is the negation of a variable. Hence, clauses in φ consist only of negated variables. Note that φ is still a weft-1 circuit, where each input node is followed by *negation* gate. The corresponding weighted satisfiability problem for formulas in antimonotone CNF is stated below:

ANTIMONOTONE WEIGHTED ℓ-SAT
Input: A formula φ in antimonotone ℓ-CNF and an $k \in \mathbb{N}$.
Question: Is there a satisfying assignment for φ of weight k?

The classical proof of the Cook-Levin theorem by Cook [105] constructs a reduction from the halting problem of a nondeterministic polynomially-time bounded Turing machine to the problem SAT. In order to obtain such a reduction in the parameterized setting, we need a Turing machine characterization of W[1] since the class was defined in terms of circuits. In Section 4.1.2, we have already discussed that SHORT TM ACCEPTANCE(k) is a parameterized variant of the halting problem which defines NP. Hence, it is a prime candidate for the characterization of W[1]. However, the proof of its W[1]-completeness is again non-trivial and rather lengthy. We skip it and refer to [141] for further

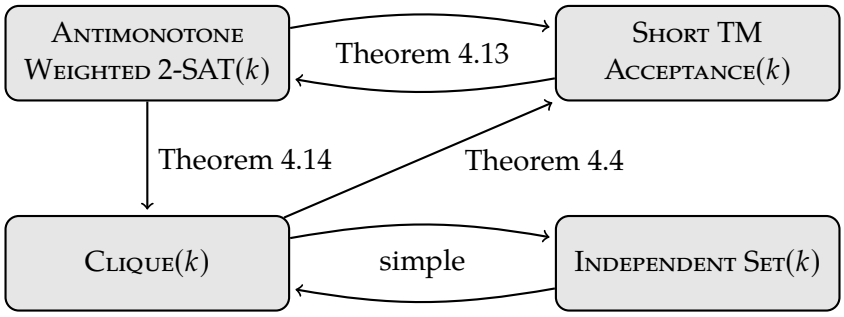


Figure 4.4: All W[1]-complete problems that we encounter in Section 4.3.2. The arrows correspond to reductions between the problems.

details[4]. Finally, we can state the parameterized variant of the Cook-Levin theorem which makes precise the similarity between NP and W[1].

Theorem 4.13. *The following problems are* W[1]*-complete:*

a) Short TM Acceptance(k)*,*

b) (Antimonotone) Weighted ℓ*-SAT*(k) *for each* $\ell \geq 2$*.*

4.3.2 Complete Problems

We prove that Clique(k) and Independent Set(k) are both W[1]-complete. In Figure 4.4 we give a summary of the considered W[1]-complete problems and the corresponding reductions among them.

We have already seen that Clique(k) is a member of W[1], either by the reduction to Short TM Acceptance(k) in Theorem 4.4 or by the explicit construction of a weft-1 circuit in Lemma 4.8. Hence, for the completeness of Clique(k) it remains to prove hardness. This is achieved by giving a parameterized reduction from Antimonotone Weighted 2-SAT(k), a problem which is known to be W[1]-hard by Theorem 4.13.

Theorem 4.14. *Antimonotone Weighted* 2*-SAT*(k) *reduces to Clique*(k)*.*

Proof. Let (φ, k) be an instance of Antimonotone Weighted 2-SAT where φ is a Boolean formula in antimonotone 2-CNF. We construct a graph $G = (V, E)$ such that G has a clique of size k if and only if φ has a satisfying assignment of weight k. The vertices V of G are given by the variables *Var* of φ. Choosing a

[4]In [141], the W[1]-completeness of Short TM Acceptance(k) relies on the W[1]-completeness of Clique(k). The latter is proven by reducing from Antimonotone Weighted 2-SAT(k). We follow a different order: we take the completeness of Short TM Acceptance(k) and Antimonotone Weighted 2-SAT(k) as given and prove the completeness of Clique(k) later. This does not generate a cyclic dependence, all problems are known to be W[1]-complete.

vertex to be in the clique represents setting the corresponding variable to 1 in an assignment. To simulate the clauses, note the following: each clause is of the form $\neg x \vee \neg y = \neg(x \wedge y)$ for $x, y \in \mathit{Var}$. Hence, in a satisfying assignment, x and y do not both evaluate to 1. This is represented in G by a missing edge between x and y. Then in each clique, x and y cannot appear together.

Formally, we have an edge $\{x, y\}$ in G if x and y do not appear together in a clause. For the correctness of the construction, assume that v is a satisfying assignment and let $C = \{x_1, \ldots, x_k\}$ be the variables set to 1. Then, C is a clique in G. For each two vertices x_i, x_j, there is no clause of the form $\neg x_i \wedge \neg x_j$ since v is a satisfying assignment. Hence, by construction of G there is an edge $\{x_i, x_j\}$. For the other direction, assume that C is a clique of size k in G. Construct as assignment v with $v(x) = 1$ if $x \in C$ and $v(x) = 0$ otherwise. Then v is of weight k and satisfies all clauses of φ. To prove the latter, let $c = \neg x \vee \neg y$ be a clause. The variables x and y are not both in C since there is no edge $\{x, y\}$ in G. Without loss of generality, assume that $x \notin C$. Then we have that $v(x) = 0$ and hence $v(c) = 1$. Consequently, v is satisfying. □

We have already argued that Clique(k) is in W[1]. Finally, we obtain the completeness confirming our intuition on the intractability of the problem.

Corollary 4.15. *The problem* Clique(k) *is* W[1]*-complete.*

The W[1]-completeness takes over from Clique(k) to Independent Set(k). Recall that both problems can be reduced to each other by constructing the complement of the input graph, see Section 4.1.2.

Corollary 4.16. *The problem* Independent Set(k) *is* W[1]*-complete.*

4.4 The Second Level

In the preceding section we have seen that antimonotone formulas in CNF are powerful enough to capture weft-1 circuits and therefore provide a characterization of the first level of intractability. Now we consider a similar result for the second level W[2], the so-called *normalization theorem* [141]. Roughly, it characterizes the class W[2] in terms of *monotone* CNF-formulas. We will then apply the result to obtain W[2]-complete problems, among them the already considered Set Cover(r) as well as Dominating Set(k).

4.4.1 Normalization Theorem

The parameterized variant of the Cook-Levin theorem shows that formulas in ℓ-CNF for a fixed integer $\ell \geq 2$ are sufficient to express weft-1 circuits. Their only large gate is the *and* gate which forms the conjunction of all the clauses. We relax the notion by allowing the formula to have a second large gate, namely we allow for a large *or* gate in each clause. This corresponds to

a more general CNF were the clauses are not streamlined to a fixed number of literals, each clause can now have an arbitrary number. Seen as a circuit, the formula has weft 2. Hence, the resulting weighted satisfiability problem is in W[2]. We state the problem below. Note that we explicitly demand for a formula in CNF in the input. Unlike an ℓ-CNF, this amounts to a general CNF with no restriction on the number of literals in the clauses.

Weighted CNF-SAT
Input: A formula φ in CNF and an $k \in \mathbb{N}$.
Question: Is there a satisfying assignment for φ of weight k?

Downey and Fellows have shown [136, 137, 138] that formulas in general CNF are enough to characterize W[2]. One can even assume that the formulas are *monotone*. This means that in the CNF, none of the literals is a negation of a variable. The corresponding decision problem is defined as follows:

Monotone Weighted CNF-SAT
Input: A formula φ in monotone CNF and an $k \in \mathbb{N}$.
Question: Is there a satisfying assignment for φ of weight k?

We state the result by Downey and Fellows. It is referred to as the *normalization theorem* and shows that both problems, (Monotone) Weighted CNF-SAT(k) are complete for the second level of the W-hierarchy. This makes the theorem a quite useful tool when identifying further complete problems for the class. We skip the proof due to its length and refer to [141] for details.

Theorem 4.17. *(Monotone) Weighted CNF-SAT(k) is* W[2]*-complete.*

In the literature, Theorem 4.17 is typically stated in a more general form. Downey and Fellows actually provided a characterization for each class W[t] where $t \geq 2$ via so-called *t-normalized* formulas. This also explains the name *normalization theorem*. However, in this thesis we are only interested in W[2] and since 2-normalized formulas correspond to general CNF-formulas, our instance of the normalization theorem is correct.

4.4.2 Complete Problems

We utilize the normalization theorem to identify some W[2]-complete problems. This includes Set Cover(r) and Dominating Set(k), two problems that we conjectured as *more intractable* than Clique(k). The proof of their completeness for the second level of intractability confirms this. An overview of all considered W[2]-complete problems is given in Figure 4.5.

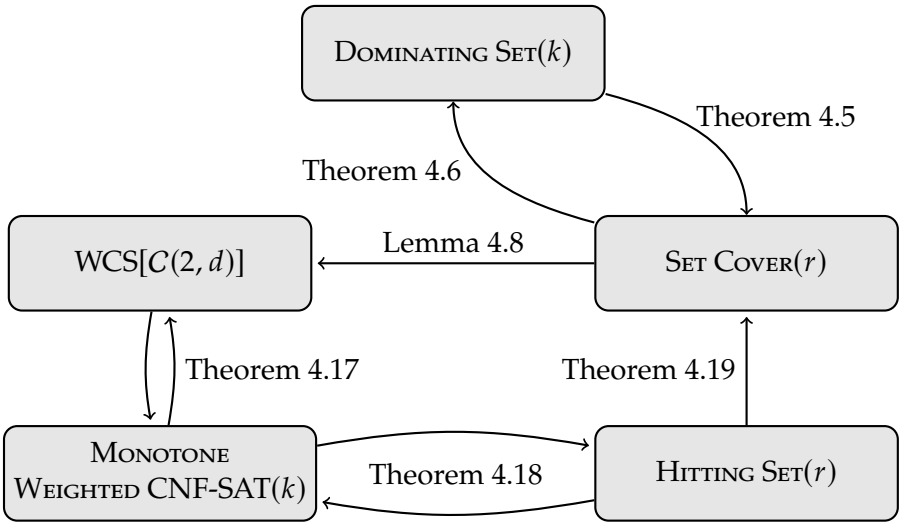


Figure 4.5: All W[2]-complete problems that we consider in Section 4.4.2. The arrows correspond to reductions between the problems.

Hitting Set The completeness of Set Cover(r) and Dominating Set(k) rely on the W[2]-completeness of the problem Hitting Set(r). Its unparameterized form Hitting Set is one of Karp's 21 NP-complete problems [229]. Like in Set Cover, we are given a family $\mathcal{G} \subseteq \mathcal{P}(U)$ of subsets over some universe U and some integer $r \in \mathbb{N}$. But instead of a cover of U the goal is to find a *hitting set* of $\mathcal{G}$ of size at most r. Formally, this is a subset $H \subseteq U$ with $|H| \leq r$ that intersects with each set of $\mathcal{G}$, we have $H \cap G \neq \emptyset$ for each $G \in \mathcal{G}$.

Hitting Set
Input: A family of sets $\mathcal{G} \subseteq \mathcal{P}(U)$ over a universe U and $r \in \mathbb{N}$.
Question: Is there an $H \subseteq U : |H| \leq r$ and $H \cap G \neq \emptyset$ for all $G \in \mathcal{G}$?

To show that Hitting Set(r) is complete for W[2], we argue that there are parameterized reductions from and to Monotone Weighted CNF-SAT(k). In fact, let (φ, k) be an instance of Monotone Weighted CNF-SAT. Then an assignment v of weight k satisfies φ if and only if v sets at least one variable in each clause to true. Here, we already use that φ is monotone — the absence of negation makes the problem simpler. The latter characterization can be phrased in terms of Hitting Set. Construct an instance $(\mathcal{G}, r)$, where $r = k$ and $\mathcal{G}$ is over the universe $U = \mathit{Var}$, the variables of φ. The family $\mathcal{G}$ contains the clauses: for each clause C, the family contains a set consisting of the variables present in C. Then there is a hitting set H of size (at most) r intersecting each set of $\mathcal{G}$ if and only if φ has a satisfying assignment of weight k. In fact, H

contains the variables that are evaluated to 1 by an assignment.

A reduction into the other direction is established by turning an instance $(\mathcal{G}, r)$ of Hitting Set into a monotone formula. To this end, we let the universe U be the variables and construct φ in such a way that it has a clause for each set in $\mathcal{G}$. The clause is a disjunction of all the elements contained in the corresponding set. Then φ is in monotone CNF. Like above we get that there is a satisfying assignment of φ of weight $k = r$ if and only if there is a hitting set of size (at most) r. Along with the above reduction and the W[2]-completeness of Monotone Weighted CNF-SAT(k) which holds due to Theorem 4.17, we obtain the completeness of Hitting Set(r).

Theorem 4.18. *Hitting Set(r) is* W[2]*-complete.*

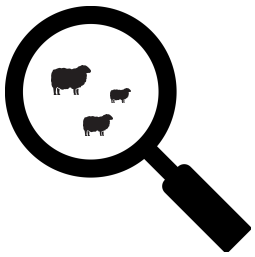
Set Cover and Dominating Set We already know that Set Cover(r) lies in the class W[2] by a reduction to the weighted circuit satisfiability problem for circuits of weft 2 from Lemma 4.8. Hence to obtain completeness, we only need to show hardness. This is achieved by a reduction from Hitting Set(r).

Theorem 4.19. *The problem Hitting Set(r) reduces to Set Cover(r).*

Details of the proof can be found in Appendix A.2.3. We can derive that Set Cover(r) is indeed W[2]-complete. Since Dominating Set(k) is equivalently hard under parameterized reductions by Theorem 4.6 and 4.5, it is W[2]-complete as well. We summarize the results:

Corollary 4.20. *Both, Set Cover(r) and Dominating Set(k) are* W[2]*-complete.*

5. Fine-Grained Analyses



When classifying the complexity of a parameterized problem, the most urgent question is typically whether it is fixed-parameter tractable or not. In Chapters 3 and 4 we have tackled this question and provided tools for going both ways: proving membership in FPT or proving it unlikely. However, while classifying a problem as fixed-parameter tractable might be sufficient from a theoretic point of view, it is usually not when we are concerned with the precise running time of an algorithm due to its practical applicability. Indeed, the plain definition of FPT allows for an arbitrarily large function $f(k)$ in the running time of a problem where k is the parameter. Hence, a problem might be FPT but only with an algorithm that takes $O^*(2^{2^k})$ time. There are even examples where $f(k)$ is not even elementary [112]. Except for very small values of k, we would not consider such an algorithm as efficient. But there might be an efficient algorithm for the problem, one that utilizes a different FPT-technique and that runs in time $O^*(k^k)$ or even $O^*(2^k)$, revealing a huge gap to the $O^*(2^{2^k})$-algorithm. It is therefore important to find the precise function $f(k)$ that is required to solve the problem at hand. Questions like this can be answered by *Fine-Grained Complexity*.

With a fine-grained (complexity) analysis we can determine an *optimal* function $f(k)$, on the one hand by providing an algorithm that actually runs in time $O^*(f(k))$ and on the other hand by proving that a faster algorithm is unlikely. We already discussed how to find efficient algorithms by applying FPT-techniques in Chapter 3. In this section, we are concerned with the latter, namely finding *lower bounds* for the function $f(k)$.

As usual in complexity theory, these lower bounds are relative, obtained from assumptions that we consider *hard to break* like $\mathsf{P} \neq \mathsf{NP}$. The appropriate assumption in parameterized complexity would be $\mathsf{FPT} \neq \mathsf{W}[1]$, something which seems intuitive but turns out to be too weak to provide lower bounds for problems within FPT. Instead, Impagliazzo, Paturi, and Zane [217, 218] suggested to rely on the intrinsic hardness of 3-SAT. In fact, since the problem's discovery there have been many algorithmic advances in solving 3-SAT but none of these algorithms actually runs in *subexponential* time. This means that there seems to be a natural lower bound of the form $2^{o(n)}$ for solving 3-SAT-instances with n variables, forbidding algorithms that run in time $O^*(2^{n/g(n)})$ where $g(n)$ is an unbounded monotonously increasing function. The three

authors assumed that breaking this lower bound is highly unlikely as 50 years of algorithmic improvements were not able to. Their assumption is formulated in the so-called *Exponential Time Hypothesis* (ETH) asserting that there is no algorithm solving 3-SAT in time $2^{o(n)}$.

The ETH was the starting point for obtaining lower bounds in parameterized complexity. Nowadays the assumption is widely believed and evolved to a standard tool in fine-grained analyses. It has been employed to derive lower bounds of different forms like $2^{o(k)}$, $2^{o(k \cdot \log(k))}$, or $2^{2^{o(k)}}$ for a whole family of problems [112, 141], for the first time providing proofs for the optimality of various algorithms. Applying the ETH typically amounts to finding an *appropriate* reduction from 3-SAT to the problem of choice. Here, *appropriate* means that the parameter n needs to be handled with a lot of care. This makes it sometimes tricky to find such reductions.

Lower bounds based on the ETH typically prevent that the running time of a problem can be improved in the exponent. However, the ETH is not capable of providing similar bounds for the basis. This is where two alternative hardness assumptions enter the picture. These can be utilized to prohibit algorithms running in time $O^*((c - \varepsilon)^k)$ for some $\varepsilon > 0$. The first is the *Strong Exponential Time Hypothesis* (SETH) [217, 218]. It assumes that general SAT, where the input is a formula in CNF with n variables, cannot be solved in time $O^*((2 - \varepsilon)^n)$ for an $\varepsilon > 0$. While there are exceptions like 3-SAT where such an algorithm is known, it would at least be considered rather surprising if such an algorithm would be found for general SAT. However, although SETH is not as widely believed as ETH, the assumption was used in numerous problems to obtain lower bounds and is considered a standard tool. The second hardness assumption is the so-called *Set Cover Conjecture* (SCON). This relatively new conjecture [110, 111] relies on the fact that despite extensive effort, no algorithm for Set Cover running in time $O^*((2-\varepsilon)^n)$ for an $\varepsilon > 0$ was found so far. In fact, the fastest known algorithms for the problem run in time $O^*(2^n)$. Like SETH, the conjecture is disputed but provides lower bounds for an increasing number of problems.

A further task of a fine-grained analysis is to determine the size of the problem kernel. In Section 3.4 we have seen how to give an upper bound on this size. Here we take a look at the contrary approach, namely proving that a kernel of polynomial size is unlikely to exist. In fact, there is a whole framework devoted to this task [57, 59], constituting the primary source for kernel lower bounds nowadays. As before, the obtained lower bounds rely on some hardness assumption. While formulating the assumption needs some more technical details, breaking the assumption would cause a highly unlikely fact: a collapse of the polynomial hierarchy to the third level [333].

We employ the lower bound assumptions and the framework for kernel lower bounds in chapters 6, 8, and 9 for proving that our verification algorithms are optimal. In order to lay the foundations, we introduce the

exponential time hypothesis in Section 5.1. Then, in Section 5.2, we consider the SETH and in Section 5.3 the SCON. Finally, we give an overview of the framework for obtaining kernel lower bounds in Section 5.4.

5.1 The Exponential Time Hypothesis

The problem 3-SAT is one of the most-studied NP-complete problems and has experienced an extensive algorithmic development in the past 50 years. Its practical applicability and mathematical beauty make it appealing from an algorithmic point of view. Starting from a brute force algorithm running in time $O^*(2^n)$, where n is the number of variables, there has been an ongoing hunt for the fastest algorithm solving 3-SAT, culminating in an $O^*(1.308^n)$-time algorithm by Hertli [206] and the currently fastest $O^*(1.307^n)$-time algorithm by Hansen, Kaplan, Zamir, and Zwick [201]. However, none of these algorithms reports a subexponential running time of the form $2^{o(n)}$. In fact, the improvement is typically on the constant present in the basis. Consequently, the barrier $2^{o(n)}$ for algorithms solving 3-SAT persists until today.

This observation was first formulated in the *exponential time hypothesis* (ETH) by Impagliazzo, Paturi, and Zane in their work [217, 218]. While the ETH is often referred to as the assumption that 3-SAT cannot be solved in time $2^{o(n)}$, its precise definition is more technical. In fact, the mentioned assumption is only a consequence of the *real* ETH but it is more intuitive when proving lower bounds for other problems. We state the precise definition of the ETH and provide a proof of the consequence. To this end, quickly recall that an instance to 3-SAT is any formula in 3-CNF.

To formulate the hypothesis, we consider the following constant δ_3, defined to be the infimum among all factors appearing in the exponent of the running time of an algorithm solving 3-SAT. Formally, we set

$$\delta_3 = \inf\{c \in \mathbb{R} \mid \text{There is an algorithm solving 3-SAT in time } O^*(2^{c \cdot n})\}.$$

Definition 5.1. The *exponential time hypothesis* is the assumption $\delta_3 > 0$.

As already mentioned above, the ETH implies that 3-SAT cannot be solved in time $2^{o(n)}$. It is not clear yet whether the inverse implication also holds. But, as we will see, the latter formulation is much more applicable in the algorithmic setting when transferring lower bounds from the ETH to other problems. Due to this, we typically identify the ETH with its consequence. A proof is provided in Appendix A.3.1.

Lemma 5.2. *Unless ETH fails, 3-SAT cannot be solved in time* $2^{o(n)}$.

The key when applying the ETH for deriving lower bounds are *appropriate* reductions from 3-SAT. But before we consider their definition and examples in Section 5.1.2, we briefly need to discuss a feature of SAT which allows for a less strict definition of what an *appropriate* reduction actually is.

5.1.1 Sparsification

When reducing from 3-SAT to a problem of choice, the size of the constructed instance often depends on the number of clauses of the given formula. Since a formula in 3-CNF may have up to $m = n^3$ clauses, the reduction may cause a polynomial blow-up in the number of variables n. We will see later that such blow-ups negatively affect the lower bounds that we obtain from the ETH. Hence, it is important to avoid them. One possible way would be to avoid a dependence of the reduction on m, the number of clauses. However, this is not always practicable and would make finding appropriate reductions rather difficult. Fortunately, SAT has a property which allows for restricting to *sparse* instances where the number of clauses and the number of variables coincide up to a linear factor, $m = O(n)$. When we reduce from a sparse instance, the reduction may depend on m without causing a polynomial blow up.

The property of SAT was first formalized in [218] and is used to broaden the spectrum of applications of ETH. The corresponding result, known as the *Sparsification Lemma*, is considered standard nowadays and its consequences were used to obtain various lower bounds in parameterized complexity.

Theorem 5.3. *Let $\ell \in \mathbb{N}$. For each $\varepsilon > 0$, there exists a constant $C \in \mathbb{N}$ such that any formula φ in ℓ-CNF with n variables admits an equivalent representation*

$$\varphi = \bigvee_{i=1}^{t} \varphi_i,$$

where $t \leq 2^{\varepsilon \cdot n}$ and φ_i is in ℓ-CNF containing at most $C \cdot n$ clauses. Moreover, there is an algorithm computing the formulas φ_i in $O^*(2^{\varepsilon \cdot n})$ time.

The proof given in [218] shows a similar sparsification lemma for Set Cover and derives Theorem 5.3 as a corollary. Due to its complexity and length we skip it and rather focus on an important consequence.

Note that the formulas φ_i obtained from Theorem 5.3 are sparse. Hence, our goal is to restrict to those. But this does not work immediately as computing the φ_i takes exponential time $O^*(2^{\varepsilon \cdot n})$. However, we have control over the parameter ε and if we chose it small enough, we can actually assume sparseness and refine the lower bound for 3-SAT. The corresponding result is the key for obtaining lower bounds for other problems beyond 3-SAT.

Theorem 5.4. *Unless ETH fails, 3-SAT cannot be solved in time $2^{o(n+m)}$.*

Proof. We assume the contrary and show how to derive a contradiction. Hence, let $\mathcal{A}$ be an algorithm solving 3-SAT in time $2^{o(n+m)}$. We use $\mathcal{A}$ to construct an algorithm for 3-SAT that contradicts the exponential time hypothesis. To this end, let φ be a formula in 3-CNF with n variables and recall that by ETH we have $\delta_3 > 0$. Set $\varepsilon = \frac{1}{4} \cdot \delta_3$. Then we have $\varepsilon > 0$ and hence we can apply Theorem 5.3. We obtain a constant $C \in \mathbb{N}$ and an algorithm $\mathcal{D}$

that runs in time $O^*(2^{\varepsilon \cdot n})$ and that computes a disjunction $\varphi = \bigvee_{i=1}^{t} \varphi_i$, where $t \leq 2^{\varepsilon \cdot n}$ and the φ_i are in 3-CNF with at most $C \cdot n$ clauses.

Now we can describe the new algorithm for 3-SAT. Given a formula φ, we first apply algorithm $\mathcal{D}$ to compute the disjunction $\varphi = \bigvee_{i=1}^{t} \varphi_i$. Then we apply algorithm $\mathcal{A}$ on each of the formulas φ_i. If $\mathcal{A}$ finds a satisfiable φ_i, we output *yes*, otherwise *no*. Note that the described algorithm is correct since φ is satisfiable if and only if one of the φ_i is.

It is left to analyze the time consumption. There are two major steps in the described algorithm, the application of $\mathcal{D}$ and the application of $\mathcal{A}$ to the sparse formulas φ_i. The former runs in time

$$O^*(2^{\varepsilon \cdot n}) = O^*(2^{\frac{1}{4} \cdot \delta_3 \cdot n}).$$

To determine the time consumed by the second step of the algorithm, we need to take one step back. Recall that algorithm $\mathcal{A}$ runs in time $2^{o(n+m)}$. This means there is an unbounded and monotonously increasing function $g(x)$ such that the running time of $\mathcal{A}$ can be bounded by

$$O^*(2^{\frac{n+m}{g(n+m)}}).$$

Since $\delta_3 > 0$, there is an $x_0 \in \mathbb{N}$ such that for all $x \geq x_0$ we have the inequality $g(x) \geq \frac{4 \cdot (C+1)}{\delta_3}$. In particular for $n, m \in \mathbb{N}$ with $n + m \geq x_0$, we have

$$g(n+m) \geq \frac{4 \cdot (C+1)}{\delta_3}.$$

Hence, we can bound the asymptotic running time of $\mathcal{A}$ by

$$O^*(2^{\frac{n+m}{g(n+m)}}) = O^*(2^{\frac{\delta_3 \cdot (n+m)}{4 \cdot (C+1)}}).$$

Finally we can determine the time needed by the second step of the above algorithm. We apply $\mathcal{A}$ to $t \leq 2^{\varepsilon \cdot n}$ formulas φ_i. These have at most n variables and $C \cdot n$ clauses. Hence, the step of the algorithm takes time at most

$$O^*(\underbrace{2^{\frac{\delta_3 \cdot (n + C \cdot n)}{4 \cdot (C+1)}}}_{\mathcal{A} \text{ on } \varphi_i} \cdot \underbrace{2^{\frac{\delta_3 \cdot n}{4}}}_{\text{nr. of } \varphi_i}) = O^*(2^{\frac{\delta_3 \cdot n \cdot (C+1)}{4 \cdot (C+1)} + \frac{\delta_3 \cdot n}{4}}) = O^*(2^{\frac{1}{2} \cdot \delta_3 \cdot n}).$$

Hence, the complete algorithm runs in time $O^*(2^{\frac{1}{2} \cdot \delta_3 \cdot n})$. By the definition of δ_3 we get $0 \leq \delta_3 \leq \frac{1}{2} \cdot \delta_3$ and thus $\delta_3 = 0$. This contradicts the ETH. □

Note that Theorem 5.4, in contrast to the above Lemma 5.2, is a major step. In fact, the lower bound for 3-SAT gets lifted from $2^{o(n)}$ to $2^{o(n+m)}$. This will significantly ease the transfer of lower bounds from 3-SAT to other problems, as we will see in the upcoming sections.

5.1.2 Linear Reductions

We utilize the lower bound of 3-SAT to obtain similar lower bounds for other decision problems. To this end, we need to establish a reduction from 3-SAT to the problem of choice. However, it is not enough to find just *any* polynomial-time reduction. In fact, we need to ensure that the parameter of the target problem does not undergo a polynomial blow-up, it has to be kept linear. The appropriate notion of reductions are *linear reductions*.

Definition 5.5. Let Q be a parameterized problem with parameter k. A *linear reduction* from 3-SAT to Q is a polynomial-time algorithm that, given a formula φ with n variables and m clauses, computes an instance (I, k) of Q where $k = O(n + m)$ such that φ is satisfiable if and only if $(I, k) \in Q$.

Once a linear reduction to the problem Q is established, the lower bound of 3-SAT can be transferred. Indeed, assume that Q can be solved in time $2^{o(k)}$. Then we can construct a new algorithm for 3-SAT. For a formula φ, we first apply the linear reduction and obtain an instance (I_φ, k) of Q with parameter $k = O(n + m)$. Then we run the $2^{o(k)}$-time algorithm for Q on (I_φ, k). If it reports a yes-instance of Q, we can return that φ is satisfiable. Otherwise, φ is unsatisfiable. Since the reduction takes polynomial time and k is linearly bounded in $n + m$, the new algorithm for 3-SAT runs in time

$$2^{o(k)} = 2^{o(n+m)}.$$

This contradicts the ETH and therefore, Q does not admit an $2^{o(k)}$-time algorithm unless ETH fails. We summarize the result in the following lemma.

Lemma 5.6. *If there is a linear reduction from 3-SAT to a parameterized problem Q with parameter k, then Q cannot be solved in time $2^{o(k)}$ unless ETH fails.*

Lemma 5.6 is the starting point for deriving lower bounds. The only thing that we need to establish is a linear reduction from 3-SAT to the problem of choice. Note that due to the sparsification lemma and its consequence, Theorem 5.4, we can allow for a dependence of the parameter k on $n + m$ in such a reduction. Without sparsification, a linear reduction would need to be more strict and only allow for a dependence of k on n. Hence, sparsification significantly simplifies the task of finding linear reductions.

Note that Lemma 5.6 relies on a fact that also holds in a more general setting. To formulate it, let Q' and Q be two parameterized problems with parameters k' and k. Assume we have a polynomial-time reduction from Q' to Q where k is not kept linear but $k \leq p(k')$ for some function $p(k')$. Then, an algorithm for Q running in time $2^{o(f(k))}$ would yield an algorithm for Q' running in time $2^{o(f(p(k')))}$: we only need to *append* the $2^{o(f(k))}$-time algorithm for Q to the reduction. This has two important consequences.

The first consequence is a generalization of Lemma 5.6 which allows to transfer lower bounds based on the ETH from other problems than 3-SAT, provided we have already proven a lower bound for the former.

Lemma 5.7. *Let Q', Q be par. problems with parameters k', k and let there be a polynomial-time reduction from Q' to Q with $k \leq p(k')$. If Q' cannot be solved in $2^{o(f(p(k')))}$ unless ETH fails, then Q cannot be solved in $2^{o(f(k))}$ unless ETH fails.*

The second consequence explains why we cannot afford (polynomial) blow-ups in the parameter. It can be considered as the special case of Lemma 5.7 where Q' is 3-SAT. Recall that n denotes the number of variables and m the number of clauses of an 3-SAT-instance.

Lemma 5.8. *Let Q be a par. problem with parameter k and assume there is a polynomial-time reduction from 3-SAT to Q such that $k \leq p(n+m)$. Then, Q cannot be solved in time $2^{o(f(k))}$ unless ETH fails if $p(x)$ is the inverse of $f(x)$.*

Proof. If $p(x)$ is the inverse of $f(x)$, we have that $f(p(n+m)) = n+m$ and thus the result follows by an application of Lemma 5.7 and Theorem 5.4. □

Assume we have found a polynomial-time reduction from 3-SAT to some problem Q but it causes a quadratic blow-up, $k \leq p(n+m) = (n+m)^2$. Then, by Lemma 5.8, we can only derive a weak lower bound for Q of the form

$$2^{o(\sqrt{k})}.$$

We can conclude that a polynomial blow-up of the parameter in the reduction leads to a weaker lower bound and should thus be avoided where possible. In the following, we give some examples of proper linear reductions that provide tight lower bounds according to Lemma 5.6. In fact, many classical reductions from 3-SAT to other NP-complete problems turned out to be linear [112]. Hence, the following examples might be familiar.

A Lower Bound for Vertex Cover We begin with a lower bound for Vertex Cover. In Section 2.1 we have seen that the problem can be solved by a bounded search tree-algorithm in time $O^*(2^k)$. However, this does not rule out a possibly faster algorithm which solves Vertex Cover in subexponential time $2^{o(k)}$. We show that, assuming the ETH, one can exclude this possibility. The key for obtaining the lower bound is a well-known reduction from 3-SAT to Vertex Cover which matches the conditions of a linear reduction.

Theorem 5.9. *Unless ETH fails, Vertex Cover cannot be solved in time $2^{o(k)}$.*

Proof. We describe the aforementioned linear reduction from 3-SAT to Vertex Cover. Then an application of Lemma 5.6 provides the lower bound for the latter. To this end, let φ be an instance of 3-SAT, a formula in 3-CNF containing n variables and m clauses. We show how to construct a graph G and

a parameter $k = O(n + m)$ in polynomial time such that φ has a satisfying assignment if and only if G has a vertex cover of size at most k.

For each clause in φ, we construct a so-called *clause gadget*. This means we add a triangle to G: three vertices, one for each literal in the clause, and an edge between each two of them. An illustration is given in Figure 5.1. For each variable x in φ, we construct a *variable gadget*, consisting of two vertices x and $\neg x$, and an edge between the two. Finally, the gadgets are connected. To this end, we add edges from x and $\neg x$ in the variable gadget to the corresponding literals in the clause gadgets.

The constructed graph G has a vertex cover of size $k = n + 2 \cdot m$ if and only if φ is satisfiable. The idea is as follows. If a satisfying assignment of formula φ is given, at least one literal in each clause is satisfied. We add the two remaining literals to the cover in each clause gadget. Moreover, we select one of the vertices in each variable gadget, according to the evaluation given in the assignment. In total, we selected k vertices and they cover each edge of G. If a vertex cover of G is given, the assignment that can be drawn out of the variable gadgets actually satisfies φ. Since k depends only linearly on n and m, the shown reduction is indeed a linear one. □

Due to Theorem 5.9 we can conclude that the bounded search tree-algorithm for Vertex Cover from Section 2.1 is *optimal* regarding the exponent of the running time. However, there is still space for improvement. The lower bound does not exclude algorithms providing a speed-up in the basis of the running time. An example is the currently fastest algorithm for Vertex Cover. Established in 2010 by Chen, Kanj, and Xia [87], it runs in time $O^*(1.2738^k)$.

A Lower Bound for Hamiltonian Cycle We employ the ETH to derive a lower bound for Hamiltonian Cycle. It shows that the dynamic programming from Section 3.2.1 is optimal in the fine-grained sense. To achieve our goal, we follow a sequence of two reductions. First, a reduction from 3-SAT to Directed Ham. Cycle is given. As the name suggests, the problem asks for a Hamiltonian cycle in a given directed graph. Then, we reduce Directed Ham. Cycle to its undirected counterpart Hamiltonian Cycle. Both reductions are kept linear and hence, the lower bound from 3-SAT takes over.

Before we give details of the reductions, we quickly remark on the problem Directed Ham. Cycle. Note that notions like (simple) paths, Hamiltonian paths, and Hamiltonian cycles clearly carry over to the directed setting. Hence, the definition of the problem is similar to its undirected version.

Directed Ham. Cycle

Input: A directed graph G.
Question: Is there a Hamiltonian cycle in G?

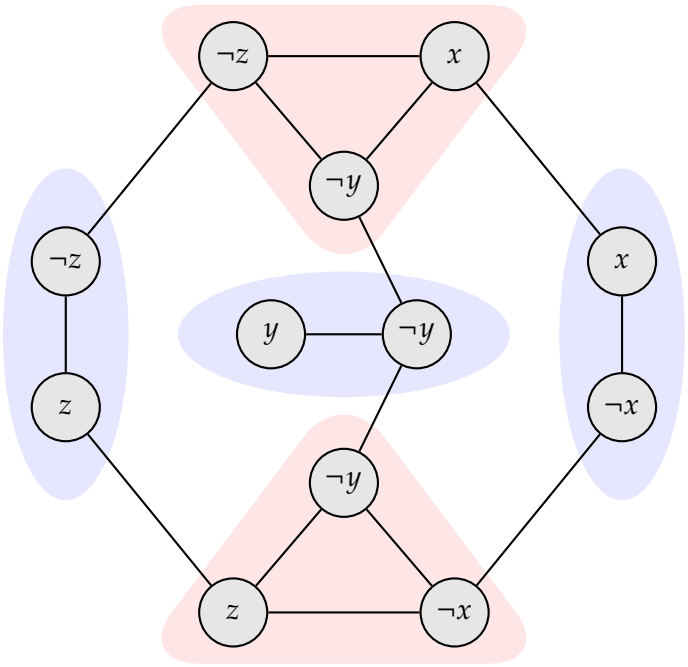


Figure 5.1: The graph G constructed from the formula φ with variables x, y, z and the two clauses $\neg z \vee x \vee \neg y$ and $z \vee \neg x \vee \neg y$. The clause gadgets are the triangles that are marked red. They have a vertex for each literal in the corresponding clause. The variable gadgets are marked blue.

Like Hamiltonian Cycle, the problem is NP-complete [229]. Moreover, the dynamic programming from Section 3.2.1 also applies here and solves the problem in $O^*(2^N)$ time, where N is the number of vertices of the given graph. By using a well-known reduction from 3-SAT to Directed Ham. Cycle, we show that the algorithm is optimal: Directed Ham. Cycle cannot be solved in time $2^{o(N)}$ unless the ETH fails. The following theorem states the reduction. We provide a proof in Appendix A.3.2. This constitutes the first step towards deriving a lower bound for the undirected version Hamiltonian Cycle.

Theorem 5.10. *Unless ETH fails, Directed Ham. Cycle cannot be solved in* $2^{o(N)}$.

With Theorem 5.10 at hand, we can derive the desired lower bound for (undirected) Hamiltonian Cycle. The key is a reduction from the problem Directed Ham. Cycle to Hamiltonian Cycle that simulates a directed graph via an undirected graph but blows up the size of the new graph only linearly. Since Directed Ham. Cycle cannot be solved in time $2^{o(N)}$ unless ETH fails, Lemma 5.7 applies and yields the same result for Hamiltonian Cycle.

Theorem 5.11. *Unless ETH fails, Hamiltonian Cycle cannot be solved in* $2^{o(N)}$.

Proof. We give a reduction from Directed Ham. Cycle to Hamiltonian Cycle that keeps the parameter N linear. By Lemma 5.7, the lower bound of Directed Ham. Cycle carries over to Hamiltonian Cycle.

Let $G = (V, E)$ be an instance of Directed Ham. Cycle, a directed graph. We construct an undirected graph $G^{un} = (V^{un}, E^{un})$ as follows. Each vertex $v \in V$ is simulated by three copies in the new graph, namely by v^{in}, v^{mid}, and $v^{out} \in V^{un}$. These copies are tied together, there are (undirected) edges $\{v^{in}, v^{mid}\}$ and $\{v^{mid}, v^{out}\}$. A directed edge $e = (v, u) \in E$ is simulated by an (undirected) edge $\{v^{out}, u^{in}\}$ connecting the copies of v and u. The undirected graph G^{un} has $3 \cdot N$ vertices and can clearly be constructed in polynomial time.

It is left to show the correctness of the construction: G has a Hamiltonian cycle if and only if G^{un} has one. To this end, let a Hamiltonian cycle $C = v_1.v_2 \dots v_n$ of G be given. Then, we get a Hamiltonian cycle in G^{un}:

$$v_1^{in}.v_1^{mid}.v_1^{out}.v_2^{in}.v_2^{mid}.v_2^{out} \dots v_n^{in}.v_n^{mid}.v_n^{out}.$$

Note that the directed edges (v_i, v_{i+1}) of C induce the edges $\{v_i^{out}, v_{i+1}^{in}\}$.

For the other direction, let a Hamiltonian cycle of G^{un} be given. By construction of the graph, the cycle must be of the following form:

$$v_1^{in}.v_1^{mid}.v_1^{out}.v_2^{in}.v_2^{mid}.v_2^{out} \dots v_n^{in}.v_n^{mid}.v_n^{out}.$$

Hence, the sequence $v_1.v_2 \dots v_n$ is a Hamiltonian cycle of G. □

A Lower Bound for 3-Coloring In Section 3.3.3 we gave a convolution-based algorithm to solve Colorable. Recall that in the problem we are given a graph G and an integer k and have to decide the existence of a k-coloring of G. Now we focus on a special case of the problem where the existence of a 3-coloring has to be decided. The problem known as 3-Coloring is still NP-complete [262, 308] and will become a handy tool when we want to obtain ETH-based lower bounds for more involved decision problems.

3-Coloring

Input: A graph G.

Question: Is there a 3-coloring of G?

There is a classical reduction from 3-SAT to 3-Coloring which is indeed linear. Hence, unless ETH fails, we obtain that 3-Coloring cannot be solved in time $2^{o(N)}$ where N is the number of vertices of the given graph. Together with the $O^*(2^N)$-time algorithm from Section 3.3.3, upper and lower bound match. We state the corresponding result and recall the reduction in Appendix A.3.3.

Theorem 5.12. *Unless ETH fails,* 3-Coloring *cannot be solved in time* $2^{o(N)}$.

5.1.3 Slightly Superexponential Lower Bounds

The lower bounds obtained by employing the ETH in Section 5.1.2 forbid algorithms that run in subexponential time. While this is sufficient to prove the optimality of algorithms for a series of parameterized problems, there are some problems that seem to have a stronger lower bound. The problem Closest String is an example. Given strings $s_1, \ldots, s_n \in \Sigma^\ell$ over some alphabet Σ, and an integer $d \in \mathbb{N}$, the problem asks for a string $s \in \Sigma^\ell$ that differs to each s_i in at most d positions. The latter is expressed via the *Hamming distance*: $d(s, s_i)$ is the number of positions where s and s_i differ.

Closest String

Input: Strings $s_1, \ldots, s_n \in \Sigma^\ell$ and an integer $d \in \mathbb{N}$.

Question: Is there a string $s \in \Sigma^\ell$ with $d(s, s_i) \leq d$ for each $i \in [1..n]$?

The problem is NP-complete [177]. From the viewpoint of parameterized complexity, Closest String(d) is known to be fixed-parameter tractable: Gramm, Niedermeier, and Rossmanith [195] gave a bounded search tree algorithm running in *slightly superexponential* time $O^*(d^d)$. Since then, there was no further algorithmic improvement on Closest String(d) in fine-grained terms. No algorithm running in single exponential time $O^*(c^d)$, where $c \in \mathbb{N}$ is a constant, was found. This raised the question whether such an algorithm does actually exists or whether there is a natural lower bound for Closest String(d) that forbids a single exponential-time algorithm.

The question was answered in 2011 by Lokshtanov, Marx, and Saurabh in their work [259]. They provided an ETH-based lower bound for Closest String that proves the existence of an $2^{o(d \cdot \log(d))}$-time algorithm highly unlikely. Therefore, a single exponential-time algorithm for the problem can be excluded proving the above $O^*(d^d)$-time algorithm optimal. But the three authors did not only provide a lower bound for Closest String. In fact they set up a whole framework for obtaining slightly superexponential lower bounds based on the ETH in general. They applied their framework to other problems like Distortion [156] and Disjoint Paths [299] as well, thereby closing gaps between known lower and upper bounds. We observed that the framework also applies in the context of program verification and obtained slightly superexponential lower bounds that we will see in Chapters 6 and 8. Due to the complexity of the framework, we give a brief introduction and skip a major part. However, this will already be sufficient for our purpose.

At the core of the framework is a modification of the problem Clique, called $k \times k$-Clique. Instead of searching a clique in any given graph, we assume the input graph to be of a particular form. Let $k \in \mathbb{N}$ be an integer. A graph $G = (V, E)$ is called *$k \times k$-graph* if its set of vertices is $V = [1..k] \times [1..k]$. Hence, vertices are arranged in a matrix and each vertex is an entry. We use

the usual matrix notation (i, j) to denote the vertex in the i-th row and j-th column. In $k \times k$-Clique, we are given a $k \times k$-graph and need to decide whether there is a clique of size k with exactly one vertex from each row.

$k \times k$-Clique

Input: A $k \times k$-graph G.
Question: Is there a clique of size k in G with a vertex in each row?

The problem can be solved by a brute force branching algorithm in time $O^*(k^k)$: just pick a vertex per row and afterwards check whether the chosen vertices form a clique. Hence, at each of the k rows, we need to branch into k possibilities, resulting in the running time $O^*(k^k)$. This is the stereotype of branching algorithms that require slightly superexponential time: we branch over k stages and each stage has a branching degree of k. In such a situation it seems not possible to find a faster algorithm. And indeed, for $k \times k$-Clique we can prove that, unless the ETH fails, it cannot be solved in time $2^{o(k \cdot \log(k))}$, implying the optimality of the brute force algorithm.

Theorem 5.13. *Unless ETH fails, $k \times k$-Clique cannot be solved in time $2^{o(k \cdot \log(k))}$.*

Proof. We reduce from 3-Coloring. According to Theorem 5.12 recall that the problem cannot be solved in time $2^{o(N)}$ unless ETH fails, where N is the number of vertices of the input graph. However, to obtain the slightly superexponential lower bound, we cannot give a linear reduction like we did before. Due to Lemma 5.7 we need to establish a reduction such that

$$k = O(\frac{N}{\log(N)}).$$

In fact, such a polynomial-time reduction along with a $2^{o(k \cdot \log(k))}$-time algorithm for $k \times k$-Clique would lead to an algorithm for 3-Coloring running in time $2^{o(N)}$. To see this, note that $k \cdot \log(k) = O(N)$. Hence, constructing the aforementioned reduction is key to obtain the lower bound for $k \times k$-Clique.

Let $G = (V, E)$ be a graph with N vertices, an instance of 3-Coloring. Set

$$k = \left\lceil \frac{2 \cdot N}{\log_3(N)} \right\rceil.$$

Due to the definition of k we may assume that $3 \cdot \sqrt{N} \leq k$. If this is not the case, we can conclude that $\log_3(N) > \frac{2}{3} \cdot \sqrt{N}$, an inequality that is only valid for bounded values of N. Then $N \leq c$ for some constant $c \in \mathbb{N}$ and we can solve the given instance G by brute force in polynomial time. Hence, for the remaining proof we might assume that $3 \cdot \sqrt{N} \leq k$ holds.

Now, with the parameter at hand, arbitrarily partition the vertex set V into k disjoint parts $V_1, \ldots, V_k$, where each part holds at most

$$|V_i| \leq \left\lceil \frac{\log_3(N)}{2} \right\rceil$$

many vertices. Note that finding such a partition is possible since

$$k \cdot \left\lceil \frac{\log_3(N)}{2} \right\rceil \geq N.$$

Then, for each part V_i, we list all possible 3-colorings of the induced subgraph $G[V_i]$. There is a bounded number of such 3-colorings, at most

$$3^{|V_i|} \leq 3^{\lceil \frac{\log_3(N)}{2} \rceil} \leq 3^{\frac{\log_3(N)}{2}+1} = 3 \cdot 3^{\log_3(N) \cdot \frac{1}{2}} = 3 \cdot \sqrt{N} \leq k.$$

Hence, we can easily list these colorings in time linear in N. If we detect that a subgraph $G[V_i]$ does not have a proper 3-coloring, we can report that G is a no-instance. Let $c_i^1, \ldots, c_i^k$ be the listed 3-colorings of $G[V_i]$. We assume that there are exactly k colorings. If not, we can copy some until we reach k.

Now we construct the instance $G^\square$ of $k \times k$-Clique. To this end, create a vertex (i, j) for each $i, j \in [1..k]$. We obtain an $k \times k$-graph. We add an edge between the vertices (i, j) and (i', j') if $i \neq i'$ and if $c_i^j \cup c_{i'}^{j'}$ is a 3-coloring of the subgraph $G[V_i \cup V_{i'}]$. Hence, edges indicate that colorings are compatible.

It is left to prove the correctness: G has a 3-coloring if and only if $G^\square$ has a clique of size k with one vertex from each row. Let us assume the former, then there is a 3-coloring c of G. For each $i \in [1..k]$ consider the restriction $c_{|V_i}$ of c to V_i. It is a proper 3-coloring of $G[V_i]$. Hence, there exists an index $j_i \in [1..k]$ such that $c_{|V_i} = c_i^{j_i}$. Now we show that the vertices (i, j_i) for $i \in [1..k]$ form a clique in $G^\square$. To this end, let (i, j_i) and $(i', j_{i'})$ be two of these vertices located in different rows, $i \neq i'$. Then we have that

$$c_i^{j_i} = c_{|V_i} \text{ and } c_{i'}^{j_{i'}} = c_{|V_{i'}}$$

are the restrictions of c to V_i and $V_{i'}$. Since c is a 3-coloring of the whole graph G, the union $c_i^{j_i} \cup c_{i'}^{j_{i'}} = c_{|V_i \cup V_{i'}}$ is a 3-coloring of $G[V_i \cup V_{i'}]$. By construction of $G^\square$ there is an edge between (i, j_i) and $(i', j_{i'})$.

For the other direction, assume that we have a clique in $G^\square$ on some vertices (i, j_i) with $i \in [1..k]$. Construct $c = \bigcup_{i \in [1..k]} c_i^{j_i}$. We show that it is a 3-coloring of G. First of all note that c assigns a color to each vertex in V since it is a union of colorings for each part V_i. Now let $\{v, w\} \in E$ be an edge in G. We need to show that their colors are distinct, $c(v) \neq c(w)$.

If v, w are in V_i for an $i \in [1..k]$, we get $c(v) = c_{|V_i}(v) = c_i^{j_i}(v)$ and similarly $c(w) = c_i^{j_i}(w)$. Since $c_i^{j_i}$ is a 3-coloring of $G[V_i]$, we obtain $c(v) \neq c(w)$.

If $v \in V_i$ and $w \in V_{i'}$ for $i, i' \in [1..k]$ with $i \neq i'$, we get that

$$c(v) = c_{|V_i \cup V_{i'}}(v) = (c_i^{j_i} \cup c_{i'}^{j_{i'}})(v)$$

and similarly $c(w) = (c_i^{j_i} \cup c_{i'}^{j_{i'}})(w)$. Since there is an edge between the vertices (i, j_i) and $(i', j_{i'})$, the coloring $c_i^{j_i} \cup c_{i'}^{j_{i'}}$ properly 3-colors $G[V_i \cup V_{i'}]$. Hence, we get $c(v) \neq c(w)$. This completes the proof. □

Although $k \times k$-Clique seems artificial, the problem is an important source for slightly superexponential lower bounds. The reason is that it is simple to reduce from and allows for liberal reductions. In our applications we often consider parameterizations of problems with two parameters at the same time, N and K. The lower bound that we want to obtain is typically of the form $2^{o(N \cdot \log(K))}$, so that it matches an algorithm running in time $O^*(N^K)$. When reducing from $k \times k$-Clique to transfer the lower bound, we need to ensure that N depends only linearly on k. However, parameter K may depend polynomially on k without negatively affecting the lower bound.

Lemma 5.14. *Let Q be a par. problem with parameters N and K and assume there is a polynomial-time reduction from $k \times k$-Clique to Q with $N \in O(k)$ and $K \leq p(k)$ for a polynomial $p(x)$. Unless ETH fails, Q cannot be solved in time $2^{o(N \cdot \log(K))}$.*

Proof. Assume the contrary, namely that there is an algorithm solving Q in time $2^{o(N \cdot \log(K))}$. Since $K \leq p(k)$ for some polynomial $p(x)$, we get that $K \in O(k^c)$ where c is the largest degree occurring in $p(x)$. Hence, by applying the polynomial-time reduction followed by the assumed algorithm for Q, we get an algorithm for $k \times k$-Clique running in time

$$2^{o(N \cdot \log(K))} = 2^{o(k \cdot \log(k^c))} = 2^{o(k \cdot c \cdot \log(k))} = 2^{o(k \cdot \log(k))}.$$

Hence, by Theorem 5.13, this contradicts the ETH. □

Other than $k \times k$-Clique, the framework set up by Lokshtanov, Marx, and Saurabh [259] provides many more artificial and natural problems to reduce from for obtaining slightly superexponential lower bounds. However, Theorem 5.13 and Lemma 5.14 are the results that we essentially needed for our applications. Due to this, we stop with the introduction of the framework at this point and refer the reader to [112, 259] for more details.

5.1.4 The ETH for Intractable Problems

So far we have employed the ETH to derive lower bounds for problems that are known to be fixed-parameter tractable. These lower bounds solved our initial question on how to prove the optimality of an algorithm. Of course, the same question applies in the context of intractable problems as well. Although a problem is marked *intractable*, it could still be solved in a reasonable time.

For instance Clique: we can find cliques by brute force in time $O^*(n^k)$. But there might be an algorithm solving the problem in much lesser time $O^*(n^{\sqrt{k}})$, or more generally in time $O^*(n^{g(k)})$ where $g(k)$ is a small function compared to k. However, we need to disappoint the reader once more. Joint work of the authors Chen, Chor, Fellows, Huang, Juedes, Kanj, and Xia [85, 86] shows that Clique is unlikely to be solved in time $f(k) \cdot n^{o(k)}$ for any computable function $f(k)$. This means that, no matter how large $f(k)$ might be, we cannot solve the problem with n depending only subexponentially on the parameter k. This shows a further facet of the intrinsic hardness of Clique and demonstrates once more why the problem is considered *intractable*.

The lower bound is based on the ETH. To derive it, we can follow a reduction [112] similar to that of Theorem 5.13. However, we skip the details and focus on the consequences of the result. But first we need to state it.

Theorem 5.15. *Unless ETH fails, Clique cannot be solved by an algorithm running in time $f(k) \cdot n^{o(k)}$ where $f(k)$ is some computable function.*

Like before, the consequent question after establishing the *first* lower bound in a new setting is how to transfer it to other problems. For lower bounds relying on 3-SAT and $k \times k$-Clique we used a kind of polynomial-time reductions that keep the parameter (or at least one parameter) *linear*, see Lemma 5.6 and Lemma 5.14. Here, we can be more liberal in the running time: we still need the linear dependence on the parameter but can allow for the reduction to take FPT-*time* like a parameterized reduction does.

Definition 5.16. Let Q be a parameterized problem with parameter t. A *linear parameterized reduction* from Clique to Q is a parameterized reduction from Clique(k) to Q that outputs instances of Q with parameter $t = O(k)$.

Establishing a linear parameterized reduction from Clique to a problem of choice lets us transfer the lower bound. We can afford more than polynomial time since the FPT-*time* that a parameterized reduction takes will vanish in comparison with the time it takes to solve an intractable problem.

Lemma 5.17. *If there is a linear parameterized reduction from Clique to a parameterized problem Q with parameter t, then Q cannot be solved by an algorithm running in time $f(t) \cdot n^{o(t)}$, for any computable function $f(t)$, unless ETH fails.*

The proof is straight forward — it can be found in Appendix A.3.4. The lemma is our main tool to obtain lower bounds for other intractable problems. Since the concatenation of linear parameterized reductions [5] is again a linear parameterized reduction, we can apply the lemma in a transitive fashion. This means that once we have established a linear parameterized reduction

[5] Generally, *linear parameterized reductions* are parameterized reductions that keep the parameter linear. The source problem does not necessarily have to be Clique.

from a problem Q to some problem Q', with parameter k', and a linear parameterized reduction from Clique to Q is already known, we can exclude an $f(k') \cdot n^{o(k')}$-time algorithm for Q' against the ETH.

Many of the parameterized reductions that we considered before, especially in Section 4.3.2, are actually linear parameterized reductions. This immediately yields lower bounds for the following W[1]-complete problems.

Corollary 5.18. *Unless ETH fails, (Multi) Clique and (Multi) Independent Set cannot be solved in time $f(k) \cdot n^{o(k)}$ for any computable function $f(k)$.*

A Lower Bound for Subgraph Isomorphism We want to obtain a lower bound for a further intractable problem called Subgraph Isomorphism. The problem is a generalization of Clique that attracted quite some attention in classical and parameterized complexity theory [13, 167, 267, 272, 273]. For an overview from the viewpoint of parameterized complexity, we refer to the work of Marx and Pilipczuk [268]. In Chapter 6, we will employ the problem to obtain a lower bound for bounded context switching.

In Subgraph Isomorphism we are given two graphs $H = (V(H), E(H))$ and $G = (V(G), E(G))$. We need to decide whether H *embeds* into G, namely if there is an injective map $\varphi : V(H) \to V(G)$ such that for each edge $\{u, v\} \in E(H)$ of H, the image $\{\varphi(u), \varphi(v)\}$ is actually an edge in G. In this case, the graph H is indeed isomorphic to some subgraph of G.

Subgraph Isomorphism
Input: Graphs H and G.
Question: Does H embed into G?

Clearly, the problem generalizes Clique. An instance (G, k) of Clique can be transformed to an instance (H, G) of Subgraph Isomorphism where H is a clique of size k. The question whether H embeds into G is then equivalent to asking whether G contains a clique of size k. This can be seen as a polynomial-time reduction and hence shows that Subgraph Isomorphism is NP-complete.

For obtaining fine-grained lower bounds, we consider two parameterizations of the problem: Subgraph Isomorphism(k), where $k = |V(H)|$ is the number of vertices of H and Subgraph Isomorphism(e), where $e = |E(H)|$ is the number of edges of H. Note that the above reduction is a linear parameterized reduction from Clique to Subgraph Isomorphism(k). Hence, by the above Lemma 5.17 we get that Subgraph Isomorphism cannot be solved in time $f(k) \cdot n^{o(k)}$ for any computable function $f(k)$, unless the ETH fails.

Corollary 5.19. *Unless ETH fails, Subgraph Isomorphism cannot be solved by an algorithm running in time $f(k) \cdot n^{o(k)}$ where $f(k)$ is some computable function.*

The lower bound matches the running time of the brute force approach for Subgraph Isomorphism. That is, iterating over all maps $\varphi : V(H) \to V(G)$ from vertices of H to vertices of G and testing whether each edge of H is mapped to an edge of G. The approach runs in time $O^*(n^k)$.

In order to obtain a lower bound in the alternative parameter e, we consider the second parameterization of the problem. Again, the above reduction is a parameterized reduction, from Clique(k) to Subgraph Isomorphism(e). However, it is not linear anymore: the constructed graph H is a clique of size k and therefore has $e = O(k^2)$ edges. Due to this quadratic blow-up we can only exclude algorithms for Subgraph Isomorphism that run in time

$$f(e) \cdot n^{o(\sqrt{e})}.$$

The lower bound does not match the corresponding brute force algorithm running in $O^*(n^e)$. It is obtained by mapping the edges of H into the edges of G. Hence, a gap between current lower and upper bound remains.

The quadratic blow-up that weakens the above lower bound does not only occur in the context of Subgraph Isomorphism. In fact, it seems to be a common phenomena when reducing from Clique [112, 267]. This *square problem* typically appears when we try to select the edges of a clique of size k, a task that intuitively requires k^2 edge-selection gadgets. It is an open question how the square problem can be avoided in general and whether the gap for Subgraph Isomorphism can be closed completely. But a striking result by Marx from 2007 shed some light on the problem: he was able to improve on the lower bound for Subgraph Isomorphism leaving only a small gap [267].

Theorem 5.20. *Unless ETH fails, Subgraph Isomorphism cannot be solved by an algorithm running in time $f(e) \cdot n^{o(e/\log(e))}$ where $f(e)$ is some computable function.*

Like before, the lower bound of Theorem 5.20 can be transported by establishing an appropriate linear parameterized reduction from Subgraph Isomorphism(e) to a problem of choice. The proof of the theorem follows from a more general result by Marx on the complexity of the so-called *Constraint Satisfaction Problem*, a generalization of SAT. Due to its length and technical complexity, we skip it here. For details, we refer to [267].

5.2 The Strong Exponential Time Hypothesis

We have seen that the ETH forbids algorithms that run in subexponential time $2^{o(k)}$ or in less than slightly superexponential time $2^{o(k \cdot \log(k))}$. Hence, it determines a lower bound on the exponent of a running time. However, the ETH is too weak to impose such a bound on the basis: although there are ETH-based lower bounds for the problems Vertex Cover and Hamiltonian Cycle preventing $2^{o(k)}$-time algorithms and $2^{o(N)}$-time algorithms respectively, there is

an $O^*(1.2738^k)$-time algorithm for the former problem [87] and an $O^*(1.657^N)$-time algorithm for the latter [44]. In order to exclude such algorithms, we need a stronger assumption — the *strong exponential time hypothesis* (SETH) formulated by Impagliazzo, Paturi, and Zane [217, 218].

Like the ETH, the SETH relies on the hardness of the satisfiability problem but emphasizes on a different aspect of it. Although there are $O^*((2-\varepsilon)^n)$-time algorithms with $\varepsilon > 0$ for different variants of the problem, like 3-SAT on n variables, there is no such algorithm for the general problem SAT where the input is a formula in unconstrained CNF. Typically the SETH is referred to as the assumption that such an algorithm does not exist. However, its original definition is more technical and the above assumption is actually implied by the SETH. To state the SETH properly, recall that an input to the problem q-SAT for a $q \in \mathbb{N}$ is a formula in q-CNF where each clause contains exactly q literals. We define constants δ_q similar to δ_3 from the definition of the ETH.

$$\delta_q = \inf\{c \in \mathbb{R} \mid \text{There is an algorithm solving } q\text{-SAT in time } O^*(2^{c \cdot n})\}.$$

Note that for each of these values we have $\delta_q \leq 1$ since we can always solve q-SAT by a brute force approach which runs in time $O^*(2^n)$. Moreover, the values are growing, we have $\delta_q \leq \delta_{q+1}$. The latter is true since any algorithm solving $(q+1)$-SAT can also solve q-SAT within the same running time. To this end, note that a formula in q-CNF can be transformed to $(q+1)$-CNF by copying literals. Consequently, the sequence $(\delta_q)_{q \in \mathbb{N}}$ is monotonously increasing and bounded from above, hence convergent. The SETH is an assumption on the limit of this sequence.

Definition 5.21. The *strong exponential time hypothesis* is the assumption

$$\lim_{q \to \infty} \delta_q = 1.$$

The definition yields that, as the number q of literals in a clause grows, an $O^*(2^{c \cdot n})$-time algorithm for q-SAT with $c < 1$ becomes more unlikely. This assumption is indeed stronger than the ETH and consequently more disputed. However, breaking SETH would at least be considered very surprising and would mark a rather big result in (parameterized) complexity theory. To show that SETH is the stronger assumption, we prove that it implies ETH. The following theorem states the result. A proof is given in Appendix A.3.5.

Theorem 5.22. *SETH implies ETH.*

As mentioned before, the SETH implies an $O^*((2-\varepsilon)^n)$-time lower bound for SAT. Like for the ETH, we identify the SETH with this consequence as it is more intuitive when proving lower bounds for further problems.

Lemma 5.23. *Unless SETH fails, SAT cannot be solved in $O^*((2-\varepsilon)^n)$ for $\varepsilon > 0$.*

A proof of the lemma is provided in Appendix A.3.6. The next question to consider is how we can transport the lower bound of SAT to other problems. Again, we need some kind of linear reductions but this time the definition has to be rather strict. We cannot even allow for a linear blow-up on the parameter, all we can afford is to add a constant. Moreover, there is no sparsification lemma for general SAT. This means that the parameter of the target problem should solely depend on the number of variables of the formula.

Definition 5.24. Let Q be a par. problem with parameter k. A *strong linear reduction* from SAT to Q is a polynomial-time algorithm that, given a formula φ in CNF with n variables, computes an instance (I, k) of Q such that $k = n + c$, where $c \in \mathbb{N}$ is a constant, and φ is satisfiable if and only if $(I, k) \in Q$.

We prove that establishing a strong linear reduction from SAT to the problem of choice indeed transports the lower bound. However, it should be noted that finding such a reduction in practice is typically a quite challenging task. Therefore we will restrict to a single example after proving the lemma.

Lemma 5.25. *If there is a strong linear reduction from SAT to a par. problem Q with parameter k, then Q cannot be solved in $O^*((2-\varepsilon)^k)$ for $\varepsilon > 0$ unless SETH fails.*

Proof. Assume that Q can be solved by an algorithm running in $O^*((2-\varepsilon)^k)$ time for some $\varepsilon > 0$. Since there is a strong linear reduction from SAT to Q, we get that $k = n + c$ for a constant $c \in \mathbb{N}$. Hence, by appending the above algorithm to the reduction, we get an algorithm for SAT running in time

$$O^*((2-\varepsilon)^{n+c}) = O^*((2-\varepsilon)^n).$$

Note that the part $(2-\varepsilon)^c$ vanishes since it is a constant. This algorithm clearly contradicts Lemma 5.23 which completes the proof. □

A Lower Bound for NAE-SAT We apply Lemma 5.25 to obtain a lower bound for a special variant of SAT, called *not-all-equal satisfiability*, or NAE-SAT for short. In the problem we are given a formula φ in CNF and we need to decide whether there is an *NAE-assignment*. That is an assignment such that in each clause of φ at least one literal is satisfied and at least one is not. Hence, the literals of a clause should not all be evaluated to an equal value.

NAE-SAT

Input: A formula φ in CNF.

Question: Is there an NAE-assignment for φ?

Clearly, we can solve NAE-SAT in time $O^*(2^n)$ by brute force, where n is the number of variables of the input formula. Assuming the SETH, this is

actually the best we can hope for. In fact, we will see that there is a strong linear reduction from SAT to NAE-SAT that allows for transporting the lower bound. However, note that the reduction is quite simple compared to other strong linear reductions. For more examples, we therefore refer to [112].

Theorem 5.26. *Unless SETH fails, NAE-SAT cannot be solved in* $O^*((2-\varepsilon)^n)$.

Proof. We present a strong linear reduction from SAT to NAE-SAT. To this end, let φ be an instance of SAT, a formula in CNF. We construct an instance φ' of NAE-SAT as follows. First, we add a fresh variable s to φ'. Then, for each clause $C = \ell_1 \vee \ldots \vee \ell_q$ of φ, we add $C' = \ell_1 \vee \ldots \vee \ell_q \vee s$ to φ'.

We prove that φ has a satisfying assignment if and only if φ' has an NAE-assignment. Let v be a satisfying assignment of φ. To construct an NAE-assignment v' for φ', we take over the assignment v and add the evaluation $v'(s) = 0$. Since v is satisfying, each clause $C = \ell_1 \vee \ldots \vee \ell_q$ of φ has a literal ℓ_i evaluating to 1. Since s evaluates to 0 under v', the clause $C' = \ell_1 \vee \ldots \vee \ell_q \vee s$ has one literal evaluating to 1 and one literal evaluating to 0 under v'. Hence, v' is a proper NAE-assignment for the formula φ'.

Now assume that φ' has an NAE-assignment v'. If $v'(s) = 1$, then we consider the complement assignment $\bar{v}'$. Note that $\bar{v}'$ is still an NAE-assignment of φ' which evaluates s to 0. Hence, we can assume that $v'(s) = 0$. To construct a satisfying assignment v for φ, we take the values of v' but cut away the evaluation of s. Let $C = \ell_1 \vee \ldots \vee \ell_q$ be a clause of φ. Then, $C' = \ell_1 \vee \ldots \vee \ell_q \vee s$ is a clause in φ'. Since $v'(s) = 0$, there has to be a literal ℓ_i that is evaluated to 1 by v'. Hence, v evaluates ℓ_i to 1 as well and thereby satisfies the clause C. Altogether, v is a satisfying assignment for φ.

The reduction is correct and indeed strongly linear: the number of variables present in φ' is just $n + 1$. This completes the proof. □

5.3 The Set Cover Conjecture

A further assumption for obtaining tight lower bounds for the basis of an algorithm's running time is the *set cover conjecture* (SCON)[6]. It was first formulated in the work of Cygan, Dell, Lokshtanov, Marx, Nederlof, Okamoto, Paturi, Saurabh, and Wahlström [110] and is nowadays considered a standard assumption for proving the optimality of algorithms. Roughly, the SCON assumes that there is no algorithm for Set Cover running in time $O^*((2-\varepsilon)^n)$, where n is the size of the universe and $\varepsilon > 0$. Although there are algorithms for the problem running in time $O^*(2^n)$, no algorithm improving on the basis of the running time was found until today. One might see the reason for this as a sheer lack of algorithmic improvement and believe that the SCON can be refuted at any time. However, there are three reasons against this.

[6] SCON is not to be confused with a *Scone* `https://en.wikipedia.org/wiki/Scone`

There is a growing number of techniques, often inspired by algebraic methods, that allow for breaking the typical 2^k-barrier. These techniques were successful for many problems [44, 45, 46, 47, 50, 52, 54, 87] and it is quite surprising that none applied to Set Cover, given that the problem plays a central role in parameterized complexity. It seems that there is an intrinsic lower bound, prohibiting algorithms running in time $O^*((2-\varepsilon)^n)$.

An increasing number of problems rely on the lower bound provided by the SCON [53, 91, 96, 110]. Interestingly, these are typically problems that do not admit a lower bound via the SETH since they lack a suitable reduction from SAT. The reason may lie in their algorithmic nature. Problems that obtain their lower bound from the SETH usually match it by a simple brute force algorithm, like SAT does. For problems that obtain their lower bound from the SCON this is different: these can only match their lower bound by a non-trivial dynamic programming algorithm, like in the case of Set Cover. Hence, it seems that SETH is a provider of lower bounds for problems that are more of a *branching-nature*, like SAT, whereas SCON provides lower bounds for problems that are more of a *dynamic programming-nature*, like Set Cover. Given that the latter captures a chunk of problems, for which the SETH cannot provide a lower bound, the SCON actually seems a reasonable assumption.

There is some evidence on the connection between SETH and SCON. Although it is open whether one of the assumptions implies the other, it is known that an $O^*((2-\varepsilon)^n)$-time algorithm for counting the number of set covers modulo 2 contradicts the $\oplus$-SETH [110]. Roughly, this is the assumption that the number of satisfying assignments modulo 2 cannot be computed in $O^*((2-\varepsilon)^n)$. Hence, there is a relation among the assumptions on a counting level but it remains to be lifted to the general assumptions.

We proceed by defining the SCON formally. To this end, we consider the problem q-Set Cover for any $q \in \mathbb{N}$. An instance to q-Set Cover is of the form $(U, \mathcal{F}, r)$, like for Set Cover, but each set in the family $\mathcal{F}$ has at most q many elements. Similarly to SETH, we investigate the exponents appearing in the running times of algorithms solving q-Set Cover:

$$\lambda_q = \inf\{c \in \mathbb{R} \mid \text{There is an algorithm solving } q\text{-Set Cover in time } O^*(2^{c\cdot n})\}.$$

Note that the sequence $(\lambda_q)_{q\in\mathbb{N}}$ is convergent. Indeed, each element λ_q is bounded by 1 since there are algorithms for Set Cover, and consequently for q-Set Cover, that run in time $O^*(2^n)$. For instance, the dynamic programming from Section 3.2.2 or the convolution-based algorithm from Section 3.3.4. Moreover, the sequence is monotonously increasing, we have $\lambda_q \leq \lambda_{q+1}$, since an algorithm for $(q+1)$-Set Cover also solves the problem q-Set Cover. Therefore, $(\lambda_q)_{q\in\mathbb{N}}$ is bounded and increasing and thus convergent. The SCON now assumes to know the limit of the sequence.

Definition 5.27. The *set cover conjecture* is the assumption

$$\lim_{q\to\infty} \lambda_q = 1.$$

On an intuitive level, the conjecture states that, with growing $q \in \mathbb{N}$, we cannot avoid an $O^*(2^n)$-time algorithm for q-Set Cover, leaving no space for improvement on the basis of the running time. Like mentioned above, this is disputed. But like for the SETH, it would be rather surprising if the SCON was actually refuted. This would clearly be a quite impressive result and probably one of the most significant steps in parameterized complexity in recent years.

Often, the SCON is identified with a lower bound for the general problem Set Cover which is of the form $O^*((2-\varepsilon)^n)$. But formally, the latter is only a consequence of the conjecture. In the following lemma, we state the lower bound. A proof is provided in Appendix A.3.7.

Lemma 5.28. *Unless SCON fails, Set Cover cannot be solved in $O^*((2-\varepsilon)^n)$.*

In order to transport the lower bound of Lemma 5.28 from Set Cover to other problems we could proceed as for the SETH and consider strong linear reductions. However, as we have seen, the conditions imposed on these reductions are rather strict, making it hard to find a suitable reduction in practice. Therefore, we follow a result from [110] which allows for a relaxed notion of linear reductions that still transport the lower bound of Set Cover.

The mentioned result strengthens the statement of Lemma 5.28. Indeed, the SCON does not only exclude algorithms for Set Cover running in time $O^*((2-\varepsilon)^n)$ but also those running in time $O^*((2-\varepsilon)^{n+r})$. Note that now the size r of the desired set cover may appear in the exponent, resulting in a more liberal running time. This clearly prohibits a larger group of algorithms than before, therefore strengthening the lower bound. The key to the result is the so-called *powering technique*. Roughly, this is a polynomial-time reduction from Set Cover to itself that, given an instance $(U, \mathcal{F}, r)$, increases the size of $\mathcal{F}$ but decreases the size of r. We formalize it below.

Lemma 5.29. *Let $q \in \mathbb{N}$. There is a poly.-time reduction transforming instances $(U, \mathcal{F}, r)$ of q-Set Cover to instances $(U, \hat{\mathcal{F}}, \hat{r})$ of q^2-Set Cover with $\hat{r} \leq \frac{1}{q} \cdot n$.*

A proof is given in Appendix A.3.8. With the powering technique available we can now improve the lower bound of Lemma 5.28.

Theorem 5.30. *Unless SCON fails, Set Cover cannot be solved in $O^*((2-\varepsilon)^{n+r})$.*

Due to its length, we also moved the proof of Theorem 5.30 to Appendix A.3.9. We proceed by defining the reductions needed to prove lower bounds based on the SCON. Note that these may depend on both parameters, n and r. Therefore, they form a more relaxed notion than the strong linear reductions used in the context of the SETH, making it simpler to find them.

Definition 5.31. Let Q be a parameterized problem with parameter k. A *tight linear reduction* from Set Cover to Q is a polynomial-time algorithm that, given an instance $(U, \mathcal{F}, r)$ of Set Cover, computes an instance (I, k) of Q with

$$k = n + r + c,$$

where $c \in \mathbb{N}$ is a constant, and $(U, \mathcal{F}, r)$ is a yes-instance if and only if (I, k) is.

Tight linear reductions are capable of transporting the lower bound of Set Cover to a problem of choice, provided we have found such a reduction. The reasoning is as before, an algorithm breaking the lower bound of the latter problem contradicts the lower bound of Set Cover.

Lemma 5.32. *If there is a tight linear reduction from Set Cover to a par. problem Q with parameter k, then Q cannot be solved in $O^*((2-\varepsilon)^k)$ for $\varepsilon > 0$ unless SCON fails.*

Proof. For the sake of a contradiction assume that there is an algorithm solving Q in time $O^*((2-\varepsilon)^k)$ for an $\varepsilon > 0$. By putting the tight linear reduction from Set Cover to Q before the mentioned algorithm, we actually obtain an algorithm solving Set Cover in estimated time

$$O^*((2-\varepsilon)^{n+r+c}) = O^*((2-\varepsilon)^{n+r}),$$

which contradicts Theorem 5.30 and hence, the SCON. □

A Lower Bound for Steiner Tree As an example for the applicability of the SCON, we consider a lower bound for the problem Steiner Tree [110]. The problem is a classic [142] in parameterized complexity and has received considerable attention from an algorithmic point of view [48, 169, 179, 283]. To define it, we need some preceding notions. Let $G = (V, E)$ be a graph and $T \subseteq V$ a subset of vertices called the *terminals*. A *Steiner tree* is a subset $X \subseteq V$ that contains the terminals T and such that the induced subgraph $G[X]$ is connected. The *size* of X is its cardinality $|X|$. It may seem odd at first sight to call a set of vertices a *tree*. However, finding a Steiner tree corresponds to finding a spanning tree of a connected subgraph that covers the terminals.

The problem Steiner Tree asks whether for a given graph and set of terminals, there exists a Steiner tree of a certain size. We state it below.

Steiner Tree

Input: A graph $G = (V, E)$, terminals $T \subseteq V$, an integer $t \in \mathbb{N}$.
Question: Is there a Steiner tree of size at most t?

We are interested in the parameter t. The fastest algorithms for Steiner Tree(t) run in time $O^*(2^t)$ [48, 283]. Notably, these are not obtained by brute

force, they rely on some non-trivial dynamic programming technique like fast subset convolution. We prove that, assuming the SCON, the algorithms are optimal: Steiner Tree cannot be solved in $O^*((2-\varepsilon)^t)$ for an $\varepsilon > 0$. To this end, we consider a tight linear reduction from Set Cover to Steiner Tree. By an application of Lemma 5.32, the lower bound follows.

Theorem 5.33. *Unless SCON fails, Steiner Tree cannot be solved in $O^*((2-\varepsilon)^t)$.*

Proof. We describe a tight linear reduction from Set Cover to Steiner Tree. Let $(U, \mathcal{F}, r)$ be an instance of the former. We construct an instance (G, T, t) of Steiner Tree with parameter $t = n + r + 1$. Roughly, the graph $G = (V, E)$ is the incidence graph of U and $\mathcal{F}$. This means that V contains a vertex for each element in U and each set in $\mathcal{F}$ and an edge among them to display the membership relation. Formally, we set $V = U \cup \mathcal{F} \cup \{s\}$. Note that s is a fresh vertex, solely needed for connectivity requirements. The edges are given by $\{u, S\}$ for each $u \in U$ and $S \in \mathcal{F}$ that satisfy $u \in S$. Moreover, s has an edge to all vertices of $\mathcal{F}$. The terminals are given by $T = U \cup \{s\} \subseteq V$. Clearly, the instance (G, T, t) can be constructed in polynomial time.

It is left to prove the correctness of the construction. To this end, first assume that $S_1, \ldots, S_r \in \mathcal{F}$ is a set cover of U. Consider the Steiner tree $X = U \cup \{s\} \cup \{S_1, \ldots, S_r\}$ of size at most t. We need to argue that $G[X]$ is connected. Note that two vertices S_i and S_j are connected through s. Let $u \in U$ be a vertex. Then, u and S_i are connected since there exists an index $j \in [1..r]$ such that $u \in S_j$. Consequently, there is an edge $\{u, S_j\}$ and since S_j and S_i are connected, u and S_i are as well.

For the other direction, assume there is a Steiner tree X of size at most t. Then, X contains the terminals $T = U \cup \{s\}$ and therefore intersects $\mathcal{F}$ in at most r sets $S_1, \ldots, S_r$. These sets form a cover of U. To prove it, consider $u \in U$. Since $u \in T$ and the graph $G[X]$ is connected, there has to be an edge to a vertex S_j, $j \in [1..r]$. Phrased differently, we have $u \in S_j$.

Since the described reduction is indeed a tight linear reduction from Set Cover to Steiner Tree, the lower bound follows from Lemma 5.32. □

5.4 Lower Bounds for Kernels

With the hardness assumptions in the preceding sections we have considered tools for proving the optimality of algorithms. The need for such algorithms is one of the major concerns of fine-grained complexity. Like for algorithms, there is a similar need for *optimal kernelizations*. As kernelizations are preprocessing algorithms that compress a given instance to a smaller one, the kernel, they play an important role in applications that handle large-scaled instances. For these, an optimal kernelization would output a kernel being as small as possible. However, we are currently missing tools to make the latter more precise. While we can potentially design kernelization algorithms

yielding kernels of small size, we do not have techniques to obtain lower bounds on the same. This makes it impossible to determine whether an efficient kernelization for a problem does not exist or was just not found yet.

In 2008, this problem was resolved by the lower bound framework of Bodlaender, Downey, Fellows, Fortnow, Hermelin, and Santhanam [56, 57, 175]. The framework is capable of proving that a parameterized problem does not admit a polynomial-sized kernel, showing that all potential kernelizations still output rather large instances. Hence, for the first time it was possible to distinguish problems that admit an efficient, a polynomial-sized, kernelization from those that do not. The framework was further generalized by Bodlaender, Jansen, and Kratsch in [58, 59] and is nowadays the primary source for kernel lower bounds. It has been applied to exclude polynomial kernels for various problems in parameterized complexity, including Longest Path [110], Steiner Tree, Hitting Set, Set Cover [134], and Set Splitting [114]. We apply the framework to verification tasks, see Chapter 6, and 8.

As typical in (fine-grained) complexity theory, the lower bound framework relies on some kind of hardness assumption. Unlike the assumptions considered in preceding sections, the new assumption does not take its hardness from a particular decision problem like SAT or Set Cover. In fact, it is based on a rather unlikely inclusion from classical complexity theory: it is not believed that the class NP is contained in *non-uniform* coNP. Formally, $\mathsf{NP} \not\subseteq \mathsf{coNP/poly}$. Although the precise relation between the two classes remains unclear, it is known that $\mathsf{NP} \subseteq \mathsf{coNP/poly}$ would cause a collapse of the polynomial hierarchy to the third level, a result that is referred to as Yap's theorem [333]. Since such a collapse is commonly considered unlikely, it constitutes the reliability of the non-inclusion assumption.

In the upcoming sections, we introduce the basics of the framework and show how the above non-inclusion assumption is linked to polynomial-sized kernels. It should be noted that the framework can be further extended to even exclude kernels of a precise polynomial size. However, we skip this part as we are only interested in disproving the existence of *any* polynomial-sized kernel. For further details, we refer the reader to the book [110].

5.4.1 Non-uniform Polynomial-Time

As mentioned above, the basis of the lower bound framework involves the non-uniform complexity class coNP/poly. Before we can define the class properly, we need to give a short introduction into non-uniform polynomial-time classes. Like for other complexity classes, the definition relies on a variant of Turing machines. This time, the machines have access to an *advice string* that may help deciding whether a given input is accepted or not.

Definition 5.34. A *polynomial-time Turing machine with advice* is defined to be a tuple $(M, (\alpha_n)_{n\in\mathbb{N}})$ consisting of a Turing machine M that runs in polynomial

time and a sequence of so-called *advice strings* $(\alpha_n)_{n\in\mathbb{N}}$ such that, on an input x of length n, the machine M has read access to advice α_n.

Note that there is only one advice α_n for all inputs of length n. The advice is therefore universal to these and helps accepting or rejecting those inputs. Formally, $(M, (\alpha_n)_{n\in\mathbb{N}})$ accepts an input x if M accepts x *under* advice $\alpha_{|x|}$. This means that M runs on x but can additionally read the string $\alpha_{|x|}$ whenever needed. The language $L(M, (\alpha_n)_{n\in\mathbb{N}})$ of the machine with advice $(M, (\alpha_n)_{n\in\mathbb{N}})$ is then defined as usual, as set of all accepted words.

In the above definition, we did not specify whether polynomial-time Turing machines with advice are deterministic or not. But like for classical Turing machines, we can distinguish between deterministic, nondeterministic, and co-nondeterministic[7] machines, depending on M. Hence, if M is deterministic, we may call the tuple $(M, (\alpha_n)_{n\in\mathbb{N}})$ a P-*machine with advice*. Note that the class P is mentioned here since it is defined in terms of deterministic polynomial-time Turing machines. Similarly, if M is (co-)nondeterministic, we call the tuple $(M, (\alpha_n)_{n\in\mathbb{N}})$ a $(\mathsf{co})\mathsf{NP}$-*machine with advice*.

We are now ready to define the non-uniform polynomial-time complexity classes. These contain all languages of Turing machines with advice where the length of each advice string is bounded by some predefined function.

Definition 5.35. Let $s : \mathbb{N} \to \mathbb{N}$ be a function and $C \in \{\mathsf{P}, \mathsf{NP}, \mathsf{coNP}\}$ a complexity class. The *non-uniform complexity class* $C/s(n)$ is defined by

$$C/s(n) = \left\{ L \;\middle|\; \begin{array}{l} \text{There is a } C\text{-machine with advice } (M, (\alpha_n)_{n\in\mathbb{N}}) \text{ such} \\ \text{that } L = L(M, (\alpha_n)_{n\in\mathbb{N}}) \text{ and } |\alpha_n| \leq s(n) \text{ for each } n \in \mathbb{N}. \end{array} \right\}$$

Let us discuss what a reasonable bound $s(n)$ on the length of each advice would be. To this end, assume our inputs are words over the alphabet $\{0, 1\}$. Then there are 2^n possible inputs of length n. If our C-machines have access to advice strings that are of exponential length 2^n, they accept any language $L \subseteq \{0, 1\}^*$: an advice α_n could encode the membership relation in L for all inputs of length n. Hence, a machine only needs to read α_n and can then output the correct answer. Note that we are typically not able to compute the advice α_n but the definition only demands for the existence of such.

Consequently, advice strings of exponential length are too powerful as they can encode the *answer* for all inputs of a certain length. It is therefore reasonable to bound the length of each advice by a polynomial in the input size. The bound yields three non-uniform complexity classes:

$$\mathsf{P/poly} = \bigcup_{c\in\mathbb{N}} \mathsf{P}/n^c, \quad \mathsf{NP/poly} = \bigcup_{c\in\mathbb{N}} \mathsf{NP}/n^c, \quad \mathsf{coNP/poly} = \bigcup_{c\in\mathbb{N}} \mathsf{coNP}/n^c.$$

[7] A co-nondeterministic machine is an alternating Turing machine without alternations and only universal states. Phrased differently, this is a machine that can guess, like a nondeterministic machine, but only accepts an input if *each* computation on it is accepting.

However, note that having polynomial-sized advice still allows for accepting undecidable problems. In fact, even 1 bit of advice already suffices. For instance, consider the undecidable language $L = \{x \mid N_{\text{bin}(|x|)} \text{ accepts } \varepsilon\}$ where $\text{bin}(|x|)$ is the binary representation of the length of x and $N_{\text{bin}(|x|)}$ is the Turing machine encoded by it. We define the following advice bit:

$$\alpha_n = \begin{cases} 1, \text{ if } N_n \text{ accepts } \varepsilon, \\ 0, \text{ otherwise.} \end{cases}$$

A machine with advice for L would then just output the bit $\alpha_{|x|}$ on an input x and therefore, provide the correct answer. Hence, we obtain that $L \in \mathsf{P/poly}$.

For the hardness assumption used in the lower bound framework for kernels, we need to consider the connection between the non-uniform classes above with standard (uniform) complexity classes. To this end, we focus on two results in this context, namely the Karp-Lipton(-Sipser)[8] theorem [230] and Yap's theorem [333]. Both results reveal an intriguing connection between non-uniform classes and polynomial hierarchy. To state the results, recall that the polynomial hierarchy consists of different levels. Level Σ_i^{P} contains all languages of alternating Turing machines that start in an existential state and have at most $i-1$ alternations. Similarly, Π_i^{P} consists of those languages accepted by alternating Turing machines that start in an universal state and have at most $i-1$ alternations. The polynomial hierarchy is the union:

$$\mathsf{PH} = \bigcup_{i \in \mathbb{N}} \Sigma_i^{\mathsf{P}} = \bigcup_{i \in \mathbb{N}} \Pi_i^{\mathsf{P}}.$$

The Karp-Lipton theorem states that, if NP is contained in P/poly, then the polynomial hierarchy collapses to the second level. It is shown as Part a) in the following theorem. Yap's theorem shows a similar statement for the relation between NP and coNP/poly. If NP is contained in the non-uniform class, the PH collapses to the third level. The result is shown below as Part b).

Theorem 5.36. *The following two connections between non-uniform complexity classes and polynomial hierarchy are known:*

a) $\mathsf{NP} \subseteq \mathsf{P/poly}$ *implies* $\mathsf{PH} = \Sigma_2^{\mathsf{P}} = \Pi_2^{\mathsf{P}}$.

b) $\mathsf{NP} \subseteq \mathsf{coNP/poly}$ *implies* $\mathsf{PH} = \Sigma_3^{\mathsf{P}} = \Pi_3^{\mathsf{P}}$.

As a collapse of the polynomial hierarchy is considered rather unlikely, neither the inclusion $\mathsf{NP} \subseteq \mathsf{P/poly}$ nor the inclusion $\mathsf{NP} \subseteq \mathsf{coNP/poly}$ are believed to hold. We employ the latter as a hardness assumption for kernel lower bounds. The former is typically used to obtain circuit lower bounds: $\mathsf{NP} \not\subseteq \mathsf{P/poly}$ means that SAT cannot be solved by circuits of polynomial size.

[8] Typically, the result is referred to as the Karp-Lipton theorem. But due to the authors, see [230], Sipser actually improved the result to its present form.

5.4.2 OR-Distillations

The key idea of the lower bound framework for kernels is to employ the theory of so-called *OR-distillations*. Roughly, these are polynomial-time compression algorithms that take a sequence of instances of a source problem and reduce them to a single instance of a target problem. Then the constructed instance is a yes-instance of the target if and only if one of the original instances is a yes-instance of the source. Phrased differently, an OR-distillation constructs the logical operator OR of the given instances.

The importance of OR-distillations goes back to a result by Fortnow and Santhanam [175]. In 2008, the authors revealed the connection between OR-distillations and the class coNP/poly. More precise, the result forbids certain OR-distillations, based on the assumption that NP $\subseteq$ coNP/poly does not hold. It therefore links the existence of a compression algorithm to the inclusion. Although kernelizations and OR-distillations are quite different, the former are compression algorithms as well. Hence, the result can be seen as a significant first step in the development of the lower bound framework.

We continue with the definition of an OR-distillation. Note that, unlike a kernelization, an OR-distillation is employed among ordinary, unparameterized problems. This constitutes a further difference between the two notions which needs to be resolved later. In the following, let Σ be a finite alphabet.

Definition 5.37. Let S and R be two languages over Σ. An *OR-distillation of S into R* is an algorithm $\mathcal{A}$ that takes a sequence of instances $s_1, \ldots, s_t \in \Sigma^*$ and outputs a single instance $w \in \Sigma^*$ such that

(1) the instance $w \in R$ if and only if there is an $i \in [1..t]$ with $s_i \in S$,

(2) the size of w is bounded: $|w| \leq p(\max_{i \in [1..t]} |s_i|)$ for a polynomial $p(x)$,

(3) $\mathcal{A}$ runs in time $q(\sum_{i=1}^{t} |s_i|)$ for a polynomial $q(x)$.

According to the definition, OR-distillations compute, in polynomial time, the logical operator OR of the given instances. Moreover, note that the size constraint on the constructed instance w is rather strict: it is not bounded by a polynomial in the input size $\sum_{i=1}^{t} |s_i|$. Instead, the size of w has to be bounded polynomially in the largest $|s_i|$. This means that the number t of given instances is not allowed to appear in $|w|$, a fact that weakens the applicability of OR-distillations at first sight. We can however mitigate this requirement as we will see in the upcoming Section 5.4.3.

In the following, we state the theorem of Fortnow and Santhanam [175] and an important corollary of it. Both results show a quite surprising relation between OR-distillations and the non-uniform class coNP/poly and constitute a major step towards obtaining kernel lower bounds.

Theorem 5.38. *Let S and R be any two languages over the alphabet Σ. If there is an OR-distillation of S into R then we have $S \in$ coNP/poly.*

Note that there is no requirement at all on the language R. Hence, it can even be undecidable and still, an OR-distillation from S into the language suffices to show membership of S in the class coNP/poly. This means that R has no influence and the sheer fact that we are able to compress information via an OR-distillation already implies the existence of some algorithm and corresponding advice strings that show membership in coNP/poly.

As a consequence of Theorem 5.38, we are now able to link the existence of certain OR-distillations with the unlikely inclusion NP $\subseteq$ coNP/poly.

Corollary 5.39. *Let S be an* NP*-hard language and R any language. If there exists an OR-distillation from S into R then we have* NP $\subseteq$ coNP/poly*.*

Proof. Let L be any language in NP. Then there exists a polynomial-time reduction from L to S. If we are given several instances of the language L, we can first apply the mentioned reduction and then append the OR-distillation of S into R. This way we actually obtain an OR-distillation of L into R. Hence, the result follows by applying Theorem 5.38. □

Due to the unlikeliness of NP $\subseteq$ coNP/poly to hold, we may assume that an OR-distillation from an NP-hard problem does not exist. However, this is still a result on unparameterized languages. If we want to employ it to exclude certain kernelizations, it remains to be lifted to the parameterized world.

5.4.3 Cross-Compositions

Like mentioned before, one of the major drawbacks of OR-distillations is the fact that they only work on ordinary, unparameterized, problems. Therefore, in their present form, they are not capable of excluding polynomial kernels, self-reductions among parameterized problems. Hence, we are in dire need of a theory making OR-distillations and parameterized problems compatible.

Such a theory was developed by Bodlaender, Downey, Fellows, Hermelin, Jansen, and Kratsch [56, 57, 58, 59]. Their main idea is to squeeze a parameterized problem into an OR-distillation by breaking the latter apart into two parts, as shown in Figure 5.2. The first part, called *cross-composition*, constructs the logical OR of several instances of the source problem by mapping them to a single instance of an intermediary parameterized problem. The second part is a more general form of a kernelization, called *polynomial compression*. It maps from the parameterized problem to the ordinary (unparameterized) target problem. By appending polynomial compression and cross-composition, we actually obtain an OR-distillation. Hence, once we have established a cross-composition from an NP-hard problem to a parameterized problem of choice, we can exclude the existence of a polynomial compression for the latter. Otherwise we would obtain an OR-distillation starting in an NP-hard problem and due to Corollary 5.39 contradict the assumption NP $\not\subseteq$ coNP/poly.

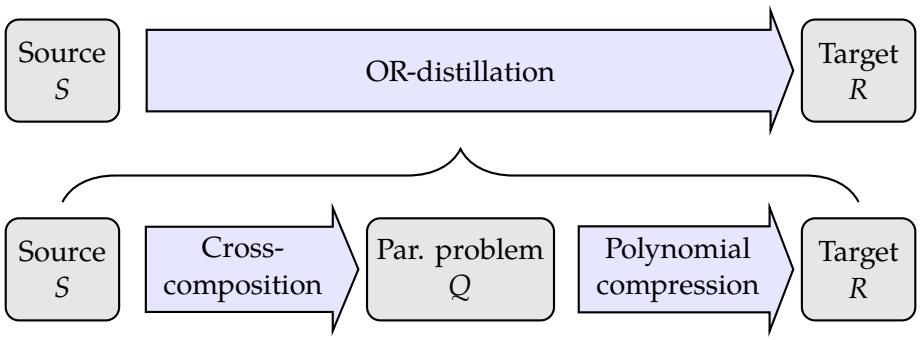


Figure 5.2: Let problems S, R be unparameterized. A parameterized problem Q can be squeezed into an OR-distillation of S into R if we consider it as two separate parts: a *cross-composition* and a *polynomial compression*. The former maps from S to Q, the latter from Q to R. By appending both, we obtain an OR-distillation from S into R. If S is NP-hard and we have a cross-composition from S into Q, a polynomial compression from Q is unlikely to exist.

We define both notions, cross-composition and polynomial compression, and provide a proof that appending both yields an OR-distillation. Let us focus on the former. Roughly, a cross-composition mimics the behavior of an OR-distillation on a parameterized level. It takes several instances of an unparameterized source problem and maps into a parameterized target problem while constructing the logical OR of the instances. However, cross-compositions are designed to be more applicable than OR-distillations. One major feature is that the given instances can be streamlined along an equivalence relation. This means that, when we are given several formulas as input, we can for instance assume the formulas to have the same number of variables and the same number of clauses. Assumptions like this often ease the construction of a suitable cross-composition. The only conditions that the employed equivalence relation has to meet are that it has polynomially many equivalence classes and it must be decidable in polynomial time whether two given instances are actually equivalent. Let Σ be a finite alphabet. For each $n \in \mathbb{N}$ let $\Sigma^{\leq n}$ denote the set of words of length at most n.

Definition 5.40. A *polynomial equivalence relation* is an equivalence relation $\mathcal{R} \subseteq \Sigma^* \times \Sigma^*$ over the words of Σ that satisfies the following:

(1) there is an algorithm that takes words $v, w \in \Sigma^*$ and decides in time $q(|v| + |w|)$ whether $(v, w) \in \mathcal{R}$, where $q(x)$ is some polynomial.

(2) for each $n \in \mathbb{N}$, the restriction of $\mathcal{R}$ to words $\Sigma^{\leq n}$ has at most $p(n)$ many equivalence classes, where $p(x)$ is some polynomial.

We consider two examples of polynomial equivalence relations that will be important later on. The first example shows a simple but powerful equivalence relation on CNF-formulas. The second one is a relation on graphs.

Example 5.41. The following are polynomial equivalence relations.

a) Assume we have an encoding of Boolean CNF-formulas over the alphabet Σ. Let $w \in \Sigma^*$, by φ_w we denote the formula that is encoded by w, by $Var(\varphi_w)$ its set of variables and by $Clause(\varphi_w)$ its set of clauses. Define the following relation: $w \sim_{\mathcal{R}} v$ if and only if w and v do not encode proper formulas, or w and v do encode proper formulas φ_w, φ_v and $|Var(\varphi_w)| = |Var(\varphi_v)|$ and $|Clause(\varphi_w)| = |Clause(\varphi_v)|$. Clearly, $\mathcal{R}$ is an equivalence relation that puts all words not encoding a formula into one class and those that do are equivalent when they have the same number of variables and clauses. This yields a polynomial equivalence relation.

   Note that (1) is clearly satisfied since we can simply count the number of variables and clauses. Regarding (2), note that a word w of length at most n encodes a formula having at most n variables and n clauses. Hence, $\mathcal{R}$ restricted to $\Sigma^{\leq n}$ has at most $n^2 + 1$ many classes: one class for each combination of $i \in [1..n]$ variables and $j \in [1..n]$ clauses, and an additional class for all words not encoding a formula.

b) Now assume that an encoding of pairs (G, k) over Σ is given, where G is an undirected graph and k is some additional parameter. For $w \in \Sigma^*$, let G_w be the graph encoded by w, $V(G_w)$ its vertices and $E(G_w)$ its edges. Moreover, let k_w be the encoded parameter. We define a relation: $w \sim_{\mathcal{R}} v$ if and only if w, v do not encode proper pairs, or w and v do encode proper pairs $(G_w, k_w), (G_v, k_v)$ with $|V(G_w)| = |V(G_v)|$, $|E(G_w)| = |E(G_v)|$, and $k_w = k_v$. This means that the number of vertices, edges, and the parameter need to coincide. Again, $\mathcal{R}$ is a polynomial equivalence relation.

   Condition (1) is met by counting vertices and edges. For (2), let $w \in \Sigma^{\leq n}$. Then, w encodes a graph with at most n vertices, n edges, and $k_w \leq n$. Hence, there are $n^3 + 1$ many classes when $\mathcal{R}$ is restricted to $\Sigma^{\leq n}$: one for each combination of $i \in [1..n]$ vertices, $j \in [1..n]$ edges, and $k_w \in [1..n]$, as well as a class combining all words not encoding proper pairs.

We are now ready to formally define cross-compositions. Note that these assume the given instances to be equivalent under some polynomial equivalence relation. This feature is not supported by OR-distillations, making cross-compositions a more versatile tool for applications.

Definition 5.42. Let $\mathcal{R}$ be a polynomial equivalence relation, $S \subseteq \Sigma^*$ a language, and $Q \subseteq \Sigma^* \times \mathbb{N}$ a parameterized language. A *cross-composition of S into Q* is an algorithm C that takes a sequence of instances $s_1, \ldots, s_t \in \Sigma^*$, all equivalent under $\mathcal{R}$, and outputs a single instance $(y, k) \in \Sigma^* \times \mathbb{N}$ such that

(1) the instance $(y, k) \in Q$ if and only if there is an $i \in [1..t]$ with $s_i \in S$,

(2) the parameter is bounded: $k \leq p(\max_{i \in [1..t]} |s_i| + \log(t))$ for a poly. $p(x)$,

(3) C runs in time $q(\sum_{i=1}^{t} |s_i|)$ for a polynomial $q(x)$.

Since the definition requires the existence of a polynomial equivalence relation, we may say that S *cross-composes into* Q if there exists such a relation along with a corresponding cross-composition of S into Q.

Let us compare the definition of cross-compositions with that of OR-distillations, see Definition 5.37. Besides the obvious differences, namely the involved polynomial equivalence relation and the parameterized problem, there is one subtle change: OR-distillations require that the size of the constructed instance is bounded polynomially in $\max_{i \in [1..t]} |s_i|$. The number of instances t is not allow to appear in that bound. However, cross-compositions only bound the parameter of the constructed instance by a polynomial which may as well depend on $\log(t)$. Consequently, the size of the instance y may depend polynomially on t and the parameter k may depend polylogarithmically on t. Although the change seems rather small, we often need to heavily rely on this dependence when constructing cross-compositions.

Assume we have constructed a cross-composition from an NP-hard problem to a parameterized problem of choice. To refute the existence of a polynomial kernel for the latter, we need to show that a polynomial kernelization appended to the cross-composition actually yields an OR-distillation. However, we do not directly consider kernelizations. Instead, we examine a slight generalization called polynomial compression and refute the existence of those, which also yields the desired result for kernels. A polynomial compression takes an instance of a parameterized problem and outputs an instance of any problem. Like for a polynomial kernelization, the size of the constructed instance is bounded polynomially only in the parameter.

Definition 5.43. Let $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized language and $R \subseteq \Sigma^*$ any language. A *polynomial compression of* Q *into* R is an algorithm $\mathcal{B}$ that takes an instance $(y, k) \in \Sigma^* \times \mathbb{N}$ and outputs an instance $w \in \Sigma^*$ such that

(1) the instance $w \in R$ if and only if $(y, k) \in Q$,

(2) the size of w is bounded: $|w| \leq p(k)$ for a polynomial $p(x)$,

(3) $\mathcal{B}$ runs in time $q(|y| + k)$ for a polynomial $q(x)$.

Note that a polynomial kernelization is a polynomial compression. Although a kernelization algorithm maps from a parameterized problem Q to itself, we can consider Q, as target of the map, in unparameterized form. Consequently, the next result does not only refute the existence of a polynomial compression but also that of a polynomial kernelization.

Theorem 5.44. *Let S be* NP*-hard and Q a par. problem. If S cross-composes into Q, then Q does not admit a polynomial compression unless* NP $\subseteq$ coNP/poly*.*

A proof is given in Appendix A.3.10. The result shows that the existence of a polynomial-sized kernel is unlikely as soon as we have established a cross-composition. Hence, finding suitable cross-compositions is key for obtaining kernel lower bounds. However, they can be quite difficult to find. Often they are rather technical and demand for deep knowledge about the considered problem and what it is computationally capable of. In the following we present an example involving a simple cross-composition.

A Kernel Lower Bound for Longest Path In Section 3.4, we have seen examples of parameterized problems that do admit kernelizations of polynomial size. Now, we consider an example that does not admit such a kernel. The problem, known as Longest Path or also k-Path, is a basic problem of parameterized complexity and closely related to Hamiltonian Path. To state it, recall that a simple path in an undirected graph is a path without repetition of a vertex. Its length is the number of involved vertices.

Longest Path

Input: A graph G and an integer $k \in \mathbb{N}$.

Question: Is there a simple path of length at least k in G?

Note that the problem Hamiltonian Path is the special case of Longest Path where the parameter k equals the number of vertices. Both problems are NP-complete and especially Longest Path has seen quite a competition for the fastest FPT-algorithm [13, 50, 88, 240, 279, 328]. But although there is an $O^*(1.66^k)$-time algorithm, it is unlikely that Longest Path(k) admits a polynomial kernel. Indeed, below we show a cross-composition of Hamiltonian Path into Longest Path(k). Since the former is NP-hard, we can invoke Theorem 5.44 and exclude a polynomial kernel unless NP $\subseteq$ coNP/poly.

Theorem 5.45. *Unless the inclusion* NP $\subseteq$ coNP/poly *holds, the problem Longest Path does not admit a kernelization of polynomial size.*

Proof. We describe a cross-composition of Hamiltonian Path into Longest Path(k). To this end, we first need to fix a polynomial equivalence relation $\mathcal{R}$. We employ the relation that identifies two encodings $u, v \in \Sigma^*$ of graphs over some alphabet Σ as equivalent if and only if u, v do not encode proper graphs or u, v do encode proper graphs G_u and G_v such that the number of vertices coincides: $|V(G_u)| = |V(G_v)|$. This is a simpler variant of the relation given in Example 5.41 and indeed a polynomial equivalence relation.

Let $G_1, \ldots, G_t$ be instances of Hamiltonian Path, equivalent under $\mathcal{R}$. Then, all G_i have the same number of vertices, say n many. We construct an

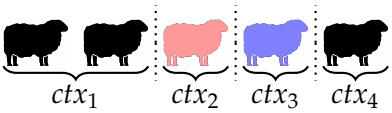
instance (G, k) of Longest Path by defining G to be the disjoint union of the G_i and by setting $k = n$. This already describes a cross-composition.

Indeed, we have that (G, k) has a simple path of length at least $k = n$ if and only if one of the G_i has a simple path of length n, a Hamiltonian path. Moreover, the parameter k is clearly bounded by the maximal size of the instances G_i. It does not depend on t at all. Lastly, the construction can be carried out in polynomial time. By Theorem 5.44, the result follows. □

Part II

Fine-Grained Complexity Analyses of Verification Tasks

6. Bounded Context Switching and Variants



Bounded context switching (BCS) is a popular under-approximate method for finding violations to safety properties in shared-memory concurrent programs. Instead of considering all computations of a program, BCS restricts the search for bugs to those that switch the processor only a bounded number of times among the threads. In contrast to complete safety verification of (finite-state) shared-memory concurrent programs, this drastically reduces the worst-case complexity, from PSPACE to NP. Moreover, verification tools that employ BCS do not only find violations to safety properties reliably, but also rather efficiently in practice [292, 293]. Both reasons make a fine-grained analysis of this under-approximation appealing.

We conducted a comprehensive fine-grained complexity analysis of BCS. It provides new and provably optimal verification algorithms, further lower bounds, and a classification into tractable and intractable parameterizations of the problem. Moreover, we conducted similar analyses for variants and generalizations of BCS, including *Round Robin* and the *Bounded Stage Restriction*. In the following we present our results. Parts of the text appeared in our publications [91, 93, 96]. This text is an extension to these works.

6.1 Popular Under-Approximations

Proving safety in shared-memory concurrent programs is complexity-wise rather hard. The corresponding decision problem is PSPACE-complete in the case of finite-state programs [241]. The high complexity can be understood when considering the case that an unsafe state is not reachable: we need to explore the whole space of computations to exclude a potential computation leading to this state. Since this is intractable for large applications, the current trend for shared-memory concurrent programs is bug-hunting: algorithms that detect safety violations in an under-approximation of the computations. The corresponding (non-)reachability problem becomes easier when we need to explore less computations. The downside is that the approach is no longer complete. There might be a computation of the program that our under-approximation does not contain. Hence, the right choice of under-approximation is essential. It should be large enough to capture most safety

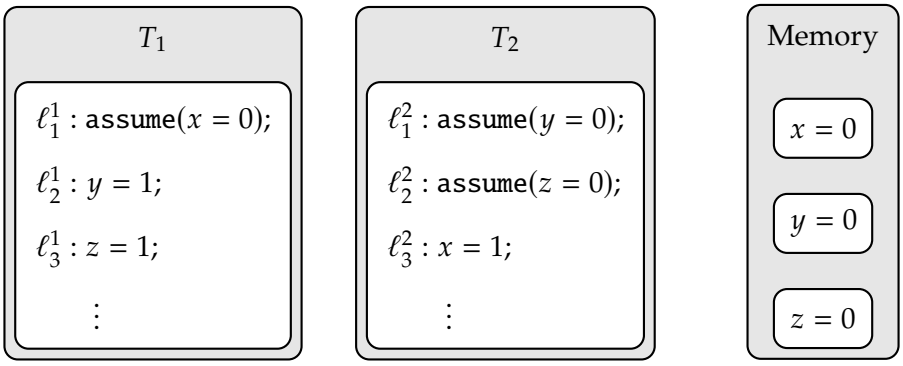


Figure 6.1: A shared-memory concurrent program with threads T_1, T_2 on Boolean variables x, y, z. Note that each instruction has a unique label ℓ_i^j. The memory stores the current evaluaton of the variables. The state $x = y = z = 1$ where all variables are evaluated to 1 is considered unsafe.

violations but still efficient from a practical and algorithmic point of view.

6.1.1 Bounded Context Switching

The most prominent method in the under-approximate verification of shared-memory concurrent programs is *bounded context switching* (BCS). It was first formulated by Qadeer, Rehof, and Wu [292, 293] around 2004 for the verification of multi-threaded C programs. A *context switch* occurs when a thread of the program leaves the processor for another thread to be scheduled. The idea of BCS is then to limit the number of context switches. This constitutes an under-approximation where we only consider computations that switch the processor a bounded number of times. Note that, effectively, this limits the amount of information that can be exchanged among the threads during a computation, but does not put a bound on how long a thread may run.

We illustrate the idea behind BCS via an example. Consider the shared-memory concurrent program depicted in Figure 6.1. It consists of two threads T_1 and T_2 that manipulate the Boolean variables x, y, z. The memory manages the evaluation of the variables. When a thread executes the instruction assume($x = 0$), it can only proceed if the current evaluation of x in the memory is 0. The write $y = 1$ changes the evaluation of y to 1. Each instruction has a unique label ℓ_i^j. Assume that it might be unsafe for the memory to arrive at the state $x = y = z = 1$. For instance, there could be two threads concurrently entering their critical sections in this case although these sections are supposed to run mutually exclusive. To verify that the above program is safe, we would need to show that none of its computations — no interleaving of the executions of T_1 and T_2 — leads to the unsafe state. However, the unsafe

state is reachable via the following sequence of instructions:

$$\rho = \ell_1^1 \,.\, \ell_1^2\, \ell_2^2\, \ell_3^2 \,.\, \ell_2^1\, \ell_3^1 \,.$$

Computation ρ is a safety violation with two context switches, marked by the dots in ρ. This splits the computation into three *contexts*, three parts of ρ that capture when the threads T_1 and T_2 are running. Note that a computation involving only two contexts or equivalently, one context switch, does not suffice to reach the unsafe state. Hence, finding the safety violation in this example requires a computation with two context switches.

Although the above example seems artificial, the number of context switches needed to detect the safety violation is quite realistic. Indeed, numerous experiments demonstrate that already few context switches suffice to find bugs in a program [280]. Phrased differently, BCS satisfies the first requirement on a useful under-approximation: it is precise enough to find most safety violations. But BCS is also appealing from an algorithmic point of view. Technically, the under-approximation corresponds to a reachability problem that is known to be NP-complete. Hence, it improves on the PSPACE-completeness of other verification methods in the case of finite-state programs, thereby satisfying the second requirement on under-approximations: BCS has a reasonable worst-case complexity.

These two factors significantly contributed to the success that BCS has seen in verification. The method was employed for different models and verification questions. Qadeer and Wu [293] originally used it to sequentialize multi-threaded C programs in order to apply the model-checker SLAM [29]. This led to further research regarding the question on how concurrent programs running under a context bound, or a related bound, can be sequentialized efficiently [61, 231, 244, 245, 248]. Bounded context switching has also been applied in the context of (multi-stack) pushdown systems and recursive programs [19, 20, 22, 23, 153, 242, 247, 248], concurrent queue systems [243] and concurrent programs with dynamic thread creation [21]. Moreover, it is implemented in several model checkers such as KISS [293], SLAM [29], CHESS [281], jMoped [310], and more. These tool perform well even on industrial instances therefore standing in contrast to the NP-completeness of BCS. Even though we consider the latter a reasonable worst-case complexity for a verification problem, it is not sufficient to explain the success that these tools see from a theoretical point of view. Phrased differently, NP seems too rough a measure for the precise complexity of BCS and so far, there is no finer complexity analysis available for the problem.

Contribution We contribute a fine-grained analysis of the bounded context switching under-approximation. The most important parameter for BCS is the number of context switches *cs*. As mentioned before, introducing *cs* to the problem was essential to improve upon the PSPACE-completeness and

Problem	Upper Bound	Lower Bounds	
BCS(cs)	in W[1]	W[1]-hard	
BCS(cs, m)	$O^*(m^{cs} \cdot 2^{cs})$	$m^{o(cs/\log(cs))}$	No polynomial kernel
Shuffle Mem(k)	$O^*(2^k)$	$O^*((2-\varepsilon)^k)$ for $\varepsilon > 0$	

Table 6.1: Results of the fine-grained complexity analysis of BCS. Main results are the algorithms and lower bounds for BCS(cs, m) and Shuffle Mem(k).

to arrive at NP instead. Moreover, the parameter is typically rather small in practical examples [280] making it an ideal candidate for a parameterized complexity analysis. The hope is to find an FPT-algorithm for BCS, non-polynomial only in cs. But surprisingly, as we shall see, parameterizing only in the number of context switches does not suffice. We show that BCS(cs) is actually W[1]-complete and therefore rather unlikely to be FPT. It requires a second parameter to obtain the desired result: the size of the memory m. Since it is often the case that shared-memory communication happens via signaling, by setting flags and semaphores, memory requirements are moderate [263]. Hence, also m is a prime candidate for a parameterization and indeed, we show that there is an FPT-algorithm for BCS(cs, m).

The complete set of results of our fine-grained analysis is summarized in Table 6.1 with the main findings highlighted in gray. One of these main results is the upper bound for BCS(cs, m). We show that the problem can be solved in time $O^*(m^{cs} \cdot 2^{cs})$. The idea of the corresponding algorithm is to enumerate the sequences of memory states at which the threads could potentially switch the context. Since m bounds the size of the memory and cs the number of context switches, there are m^{cs} such sequences. Given a sequence, the next step is to check a problem called Shuffle Mem. It tests whether the threads actually have proper computations that justify the sequence. We employ fast subset convolution to solve the problem in time $O^*(2^{cs})$. We remark that Shuffle Mem is a problem that might be interesting in its own right. It is an under-approximation that still leaves freedom for the local computations of the threads. Indeed, related ideas are common in testing [77, 98, 155, 180, 183]. It is an interesting task to determine the fine-grained complexity of testing problems and to see how subset convolution and other FPT-techniques can be applied in this context. We already point to Chapter 9, where we employed a fine-grained analysis of the testing problem for weak memory models.

The lower bound for BCS(cs, m) is established via a reduction from Subgraph Isomorphism. It shows that BCS cannot be solved in time $m^{o(cs/\log(cs))}$ unless the ETH fails. This *almost* matches the upper bound and we argue that the remaining gap is not easy to close. A matching algorithm for BCS would resolve the question on whether Subgraph Isomorphism can be solved in time

$n^{e/\log(e)}$ which has remained open since 2007 [267]. It should also be noted that a reduction from $k \times k$-Clique to BCS, which is the typical way to obtain slightly superexponential lower bounds, causes the well-known squaring problem, explained in Section 5.1.4. Therefore, $k \times k$-Clique is not suitable to provide a lower bound for BCS. For the problem Shuffle Mem, we do not leave a gap. By assuming the SCON, and setting up a reduction from Set Cover, we obtain a matching lower bound: Shuffle Mem cannot be solved in time $O^*((2-\varepsilon)^k)$ for an $\varepsilon > 0$. Like other problems that rely on the SCON, Shuffle Mem is more of a *dynamic programming-nature* like Set Cover and less of a *branching-nature* like SAT. To the best of our knowledge, it is the first verification problem that obtains a lower bound via the SCON. To investigate the applicability of preprocessings for bounded context switching, we prove that BCS(cs, m) does not admit a polynomial kernel. The result is obtained by establishing a cross-composition from 3-SAT to the problem and shows that polynomial-time preprocessings have only limited success.

6.1.2 Round Robin

Round Robin (Round Rob) is a variant of BCS where the threads act along a cyclic schedule. This means that computations of a program can be split into different rounds in which the threads consecutively take turns. Round robin assumes that the number of these rounds is bounded. Like BCS, the method constitutes a proper under-approximation and verification under round robin has received quite some attention in the literature [61, 248, 280].

Contribution To tackle the complexity of Round Rob we contribute a fine-grained complexity analysis of a *local* variant of BCS. As we will see, round robin is a special case of this variant and the time complexity of Round Rob follows from the more general analysis. Instead of sharing a number of context switches among all threads, the local variant of BCS gives a budget of cs context switches to each thread. This may increase the number of total context switches available for a computation but our focus here is on scheduling. We associate with each computation a so-called *scheduling graph* that dictates how the threads take turns. The complexity of such a graph is determined by the *scheduling dimension*, a new measure that is closely related to the carving width [301]. Restricting the computations to a given graph of bounded scheduling dimension then amounts to limiting the complexity of the scheduling. Like for BCS, this does not induce a bound on the running time of the threads. Round robin is then the special case where the scheduling graph is a cycle of scheduling dimension cs.

Table 6.2 shows the results of our fine-grained analysis in more detail. The problem LBCS is the local variant of BCS where the threads act along a given graph of scheduling dimension at most s. We construct an algorithm solving the problem in time $O^*((2m)^{4s})$. As a corollary, we obtain an algorithm for

Problem	Upper Bound	Lower Bound
LBCS(m, s)	$O^*((2m)^{4s})$	$2^{o(s \cdot \log(m))}$
Round Rob(m, cs)	$O^*(m^{4cs})$	$2^{o(cs \cdot \log(m))}$

Table 6.2: Results of the fine-grained analysis of local BCS. The two main results are on the complexity of LBCS. The entries are highlightd in gray.

Round Rob, running in time $O^*(m^{4cs})$. The upper bound is complemented by a matching lower bound of the form $2^{o(cs \cdot \log(m))}$ that also applies to LBCS. It is obtained by a suitable reduction from $k \times k$-Clique to Round Rob.

6.1.3 Bounded Stage

A further under-approximation that we consider is *Bounded Stage Reachability* (BSR). The method was introduced in 2014 [20] and can be understood as a generalization of BCS. Instead of considering computations with a bounded number of contexts, BSR focuses on *stages*. These are parts of a computation where only one thread is allowed to write to the memory while the others can still read from it. Hence, unlike for contexts, threads can still run concurrently during a stage, only writing is restricted. A stage might contain several contexts and is therefore more general. The idea of BSR is to limit the number of stages that occur in a computation. Consequently, BSR explores more computations than BCS does but, surprisingly, maintains the same complexity: BSR on finite-state programs is still NP-complete [20]. Hence, classical complexity theory does not distinguish between the problems BCS and BSR although the latter is a strict generalization. It requires a fine-grained complexity analysis to separate the complexity of both problems.

Contribution We conducted a fine-grained complexity analysis of BSR. The outcome is summarized in Table 6.3. Roughly, the results show that BSR is a much harder problem than BCS, even when restricting to a simpler model. For the latter, assume that the threads read from and write to a single memory cell. The cell is allowed to hold values from a memory domain of size d. Like for BCS, one would suppose that the number of stages k is the most important parameter of BSR. However, we were able to show that parameterizing by k and by d yields a problem that is not even in XP unless P = NP. Therefore, this parameterization of BSR can be considered intractable, showcasing the power of the under-approximation. To get an FPT-result, it requires a stricter parameterization: when we parameterize by the number of threads t and the maximal size of a thread p, we obtain an $O^*(p^{2t})$-time upper bound for the problem. But there is not much hope for improvement: we show that there is a lower bound of the form $2^{o(t \cdot \log(p))}$. Even more, there is a kernel lower

Problem	Upper Bound	Lower Bounds	
BSR(k, d)	Not in XP unless P = NP		
BSR(p, t)	$O^*(p^{2t})$	$2^{o(t \cdot \log(p))}$	No polynomial kernel

Table 6.3: Results of the fine-grained analysis of BSR. The main result is the kernel lower bound for BSR(p, t), highlighted in gray.

bound for the already rather restricted parameterization BSR(p, t). The corresponding cross-composition is technically quite involved but demonstrates once more what BSR is capable of.

6.2 Fine-Grained Complexity Analysis of Bounded Context Switching

We showcase the results of our fine-grained analysis of bounded context switching. The presentation is split into four parts. First, we introduce shared-memory concurrent programs and formally state BCS as a decision problem in Section 6.2.1. Then, in Section 6.2.2, we focus on upper bounds and elaborate on the algorithms for BCS and Shuffle Mem. Both upper bounds are complemented by corresponding lower bounds in Section 6.2.3. Finally, in Section 6.2.4, we prove that the parameterization of BCS only in the number of context switches *cs* is rather unlikely to be fixed-parameter tractable.

6.2.1 Shared-Memory Concurrent Programs and BCS

Bounded context switching is a safety verification problem of shared-memory concurrent programs. To obtain precise complexity results, it is common to assume the number of involved threads and the data domain to be finite. This also allows for modeling programs as interacting finite automata. Since checking safety corresponds to reachability, we express the latter problem via a language-theoretic formulation that will form the basis of our study.

Shared Memory We begin by formally defining shared memories of programs. These manage the current evaluation over the data domain and are the backbone of the communication among a program's threads.

Definition 6.1. A *shared memory* is a (nondeterministic) finite automaton $M = (Q, \Sigma, \delta_M, q_0, q_f)$. The *size* of M, denoted m, is its number of states $|Q|$.

The formulation of a shared memory in terms of a (nondeterministic) finite automaton $M = (Q, \Sigma, \delta_M, q_0, q_f)$ serves several purposes. The states Q correspond to the finite data domain of the program, the set of values

that the memory can be in. The initial state $q_0 \in Q$ is some fixed value that computations always start in. The final state $q_f \in Q$ reflects the reachability problem. Arriving at q_f via a computation corresponds to reaching an unsafe state of the memory. Operations of the program are modeled by the finite alphabet Σ. Note that operations may change the memory valuation. This is formalized by the corresponding transition relation $\delta_M \subseteq Q \times \Sigma \times Q$.

The semantics of a shared memory are as expected, inherited from the finite automaton. Nonetheless, we provide the needed definitions. We generalize the transition relation δ_M to words $w \in \Sigma^*$ in the obvious way. The following language contains all words, all sequences of operations in Σ^*, that lead from a state q to another state q':

$$L(M(q, q')) = \{w \in \Sigma^* \mid q' \in \delta_M(q, w)\}.$$

We typically refer to a sequence of operations as a *computation*. Then, the *language of* M is defined to contain all computations leading from the initial state q_0 to the unsafe state q_f. Formally, we have $L(M) = L(M(q_0, q_f))$.

Threads Threads are defined in terms of automata as well. In fact, we identify a thread of a program via its (labeled) control flow graph.

Definition 6.2. A *thread* is defined as a (nondeterministic) finite automaton $A_{id} = (P, \Sigma \times \{id\}, \delta_{id}, p_0, p_f)$. Here, *id* is called *thread identifier*.

The thread identifier links the definition of A_{id} to the *real* thread *id* of a program. The automaton is the labeled control flow graph of *id*. Its states P and its transition relation δ_{id} form the vertices and edges. The latter are labeled by operations in Σ, the alphabet that is also used by the shared memory M, but indexed by the thread identifier. This will become important when operations are associated with certain threads, something that we rely on when formally defining context switches. The initial state p_0 marks the initial instruction of *id*, the final state p_f reflects the reachability problem.

The *language of a thread* A_{id} is the set of computations that *id* could potentially execute to reach the final state p_f. Formally, it is defined by

$$L(A_{id}) = L(A_{id}(p_0, p_f)).$$

As the language does not take into account the effect of the operations on the shared memory M, not all of these computations are actually feasible in the program. Indeed, the thread A_{id} may issue a command $write(x, 1)$ followed by $read(x, 0)$, which the shared memory will reject. The computations of *id* that are compatible with M are given by the language intersection $L(M) \cap L(A_{id})$. To ease the notation at this point, we assume that the intersection projects away the thread identifier from the thread alphabet $\Sigma \times \{id\}$.

Shared-Memory Concurrent Programs A shared-memory concurrent program consists of multiple threads that influence each other by accessing the same shared memory. The formal definition is as follows.

Definition 6.3. Let M be a shared memory and $A_1, \ldots, A_t$ threads. A *shared-memory concurrent program* (SMCP) is a tuple $S = (\Sigma, M, (A_i)_{i \in [1..t]})$.

Note that, although not explicitly stated in the definition, the memory M acts on the alphabet Σ of operations and the threads A_i are defined over $\Sigma \times \{i\}$. To define the semantics of an SMCP, we need to mimic the influence that the threads have on each other via a language theoretic operator. The tool of choice is the so-called *shuffle* operator. It interleaves given languages and simulates the behavior of threads taking turns in a computation.

Definition 6.4. Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ be languages over disjoint alphabets $\Sigma_1 \cap \Sigma_2 = \emptyset$ and let π_i be the projection $\Sigma_1 \cup \Sigma_2 \to \Sigma_i$ preserving only the letters of Σ_i. The *shuffle* of L_1 and L_2 is the language containing all words over $\Sigma_1 \cup \Sigma_2$ the projections of which belong to the operand languages:

$$L_1 \amalg L_2 = \{w \in (\Sigma_1 \cup \Sigma_2)^* \mid \pi_i(w) \in L_i \cup \{\varepsilon\}, i = 1, 2\}.$$

Intuitively, the shuffle of two languages L_1 and L_2 contains all words that can be derived by interleaving words from L_1 with those from L_2. As a simple example, consider the singleton languages $L_1 = \{ab\}$ and $L_2 = \{c\}$ over the disjoint alphabets $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{c\}$. We have

$$L_1 \amalg L_2 = \{\varepsilon, ab, c, abc, acb, cab\}.$$

Now we are ready to define the language of a shared-memory concurrent program. It is the shuffle of all thread languages, synchronized with the language of the shared memory. The former models that the threads interleave during a computation, the latter ensures that the computations are actually compatible with the memory and thus feasible.

Definition 6.5. Let $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ be an SMCP. The *language of* S is

$$L(S) = L(M) \cap (\amalg_{i \in [1..t]} L(A_i)).$$

Note that in the definition, we again assume the intersection to project away the thread identifiers from the alphabets of the A_i. Hence, $L(S) \subseteq \Sigma^*$. To illustrate the semantics of an SMCP, we provide an example.

Example 6.6. Reconsider the simple program given in Figure 6.1. We model it as an SMCP. The threads can manipulate the evaluation of the variables by writing and they can read the currently stored evaluation from the memory. This is reflected by the set of operations Σ. We have two operations, $write(var, v)$ and $read(var, v)$, for each variable var and value v:

$$\Sigma = \{write(var, v), read(var, v) \mid var \in \{x, y, z\}, v \in \{0, 1\}\}.$$

The threads T_1 and T_2 of the program are turned into finite automata A_1 and A_2 over Σ. These are the labeled control flow graphs. The construction is illustrated in Figure 6.2. Note that A_i is supposed to be an automaton over the indexed alphabet $\Sigma \times \{i\}$. However, we omit the thread identifier for better readability. The languages of the threads are given by

$$L(A_1) = \{read(x,0).write(y,1).write(z,1)\},$$
$$L(A_2) = \{read(y,0).read(z,0).write(x,1)\}.$$

The shared memory M is an automaton managing the evaluation of the variables. It has a state for each evaluation of the triple (x, y, z), in total 2^3 states. The final state is $(1, 1, 1)$, reflecting the unsafe state of the memory. Transitions in M are defined as follows: If there is a write operation $write(x, v)$, the automaton needs to change the evaluation of variable x to value v. To this end, it contains transitions of the form

$$(v_x, v_y, v_z) \xrightarrow{write(x,v)} (v, v_y, v_z),$$

where $v_x, v_y, v_z \in \{0, 1\}$. If a read operation $read(x, v)$ appears, a corresponding transition can only occur if the current value of x matches v. We have

$$(v, v_y, v_z) \xrightarrow{read(x,v)} (v, v_y, v_z).$$

The construction of M is shown in Figure 6.2. Its language $L(M)$ contains all computations that take the memory from the initial state $(0, 0, 0)$ to the unsafe state $(1, 1, 1)$. Hence, we have constructed the complete SMCP $S = (\Sigma, M, A_1, A_2)$ modeling the behavior of the above program.

We have already considered a computation of the program leading to the unsafe state of the memory. We can translate it to a word over Σ:

$$w = read(x,0).read(y,0).read(z,0).write(x,1).write(y,1).write(z,1).$$

The word w clearly lies in the shuffle of the languages $L(A_1)$ and $L(A_2)$. Moreover, w is contained in the language of M as it compatible with the memory. Hence, we have that $w \in L(S)$ and the word corresponds to a computation witnessing that S can reach an unsafe state.

The observation made in the example is also true in general: the words in the language of an SMCP correspond to those computations of the program that witness non-safety. Hence, given a shared-memory concurrent program S, it is essential to determine whether its language $L(S)$ is non-empty. Note that an empty language means that S is safe, no unsafe state is reachable. On the other hand, a non-empty language shows that there is at least one safety violation. We are interested in measuring the complexity of testing whether $L(S)$ is non-empty. To this end, we define a corresponding decision

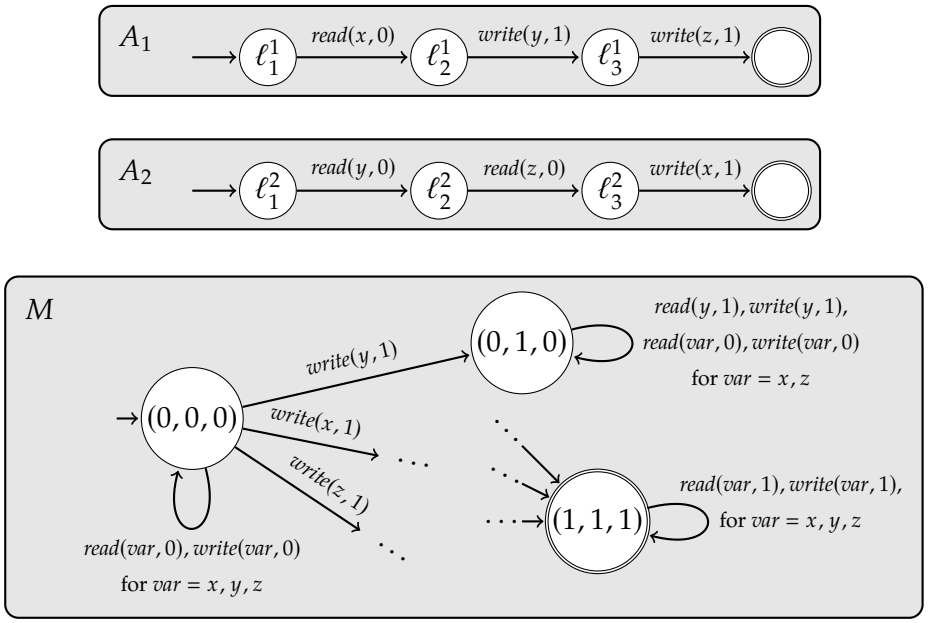


Figure 6.2: The SMCP $S = (\Sigma, M, A_1, A_2)$ according to Example 6.6. It consists of the threads A_1, A_2 and the shared memory M. Note that we omitted the thread identifiers in the alphabets of the threads A_i.

problem for this task. The problem is called *safety verification problem for SMCPs*, abbreviated by SMCP Safety. We state the problem below.

SMCP Safety

Input: An SMCP $S = (\Sigma, M, (A_i)_{i \in [1..t]})$.

Question: Is $L(S) \neq \emptyset$?

SMCP Safety is known to be PSPACE-complete [241]. Therefore, the problem in its current form is less practical for large instances of SMCPs. However, we still need it as a first step to formally capture BCS.

Bounded Context Switching To define the bounded context switching problem of interest [292], we need to formalize the notion of context switches.

Definition 6.7. Let Σ be an alphabet, $t \in \mathbb{N}$ an integer, and $w \in (\Sigma \times [1..t])^*$. Then, w has a unique decomposition into maximal infixes, where each is generated by a particular thread. Formally, we have

$$w = w_1 \dots w_{cs+1}$$

along with a function $\varphi : [1..cs+1] \to [1..t]$ that satisfies two conditions: $w_i \in (\Sigma \times \{\varphi(i)\})^+$ for $i \in [1..cs+1]$ and $\varphi(i) \neq \varphi(i+1)$ for $i \in [1..cs]$.

We refer to an infix w_i as *context* and to a thread change between two contexts w_i and w_{i+1} as *context switch*. Note that then, the word w has $cs+1$ many contexts and cs many context switches in total.

Intuitively, the contexts of a word show how the thread took turns in a computation of the program. Since we are interested in computations where the threads switch only a bounded number of times, we define the language $\text{Context}(\Sigma, t, cs)$ of all words over $\Sigma \times [1..t]$ that can be decomposed into at most $cs+1$ many contexts, stemming from at most t different threads. Formally,

$$\text{Context}(\Sigma, t, cs) = \left\{ w = w_1 \dots w_\ell \;\middle|\; \begin{array}{l} \ell \leq cs+1 \text{ and } \exists\, \varphi : [1..\ell] \to [1..t] : \\ w_i \in (\Sigma \times \{\varphi(i)\})^+ \wedge \varphi(i) \neq \varphi(i+1). \end{array} \right\}$$

The *bounded context switching* under-approximation limits safety verification for SMCPs to the language $\text{Context}(\Sigma, t, cs)$. Note that in the problem, we again assume the intersection to project away the thread identifiers.

BCS

Input: An SMCP $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ and a bound $cs \in \mathbb{N}$.
Question: Is $L(S) \cap \text{Context}(\Sigma, t, cs) \neq \emptyset$?

BCS is NP-complete, even in the case of a unary alphabet [153]. To understand which instances of the problem can be solved efficiently and, in turn, what makes an instance hard, we examined a fine-grained complexity analysis. In the following sections, we present the results.

6.2.2 Upper Bounds

We focus on the upper bounds found by the fine-grained complexity analysis. To this end, we consider the parameterization BCS(cs, m) by the number of context switches cs and the size of the memory m. Our main result is an algorithm showing that the parameterization is FPT. The idea is to decompose an instance of BCS into exponentially many instances of the complexity-wise simpler problem *shuffle membership*, denoted by Shuffle Mem. The latter is then solved via fast subset convolution.

To state the precise complexity of the algorithm, let (S, cs) be an instance of BCS, consisting of an SMCP $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ and a bound $cs \in \mathbb{N}$. To each thread A_i, our algorithm associates a new automaton B_i of size polynomial in A_i. Let $b = \max_{i \in [1..t]} |B_i|$ be the maximal size of such an automaton. Moreover, let the following denote the complexity of solving Shuffle Mem:

$$\mathsf{Shuff}(b, k, t) = O(2^k \cdot t \cdot k \cdot (b^2 + k \cdot \mathit{mult}(k))).$$

Recall that $mult(k)$ is the time that is needed to multiply two k-bit integers. The factor appears within $\mathsf{Shuff}(b, k, t)$ as a result of employing the fast subset convolution. We refer to Section 3.3 for details. The precise time complexity of solving the problem BCS can now be estimated as follows.

Theorem 6.8. *BCS can be solved in* $O(m^{cs+1} \cdot \mathsf{Shuff}(b, cs+1, t) + t \cdot m^3 \cdot b^3)$.

Interface Sequences To tackle BCS, we decompose the problem along so-called *interface sequences*. These are sequences over pairs of states of the memory that mark where a context switch may happen during a computation.

Definition 6.9. Let $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ be an SMCP with shared memory $M = (Q, \Sigma, \delta_M, q_0, q_f)$. An *interface sequence of* S is a word

$$\sigma = (q_1, q_1') \dots (q_k, q_k') \in (Q \times Q)^*.$$

The *length* of σ is k. We may call an interface sequence σ *valid* if $q_1 = q_0$ is the initial state, if $q_k' = q_f$ is the final state, and if $q_i' = q_{i+1}$ for each $i \in [1..k-1]$.

A valid interface sequence σ makes precise where the context switches in a potential computation of S should happen. Indeed, σ starts in the initial state, ends in the final state, and there is no gap between q_i' and q_{i+1} as the states coincide. Note that we can write it as $\sigma = (q_0, q_1)(q_1, q_2) \dots (q_{k-1}, q_k)$ with $q_k = q_f$. The idea is that a tuple (q_i, q_{i+1}) displays the state change that is seen on the memory M while a particular thread A_{id} is running. Hence, the state q_i shows where A_{id} takes over from another thread. Phrased differently, q_i is the state of M that appears upon the i-th context switch of a computation.

To verify that a valid interface sequence stems from an actual computation of the program, we still need to check that the threads can support a computation of S that follows the pattern of context switches dictated by the interface sequence. To this end, we need to consider *induced* interface sequences.

Definition 6.10. Let $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ be an SMCP with shared memory $M = (Q, \Sigma, \delta_M, q_0, q_f)$ and let $w \in L(S)$ be a word with its decomposition into contexts: $w = w_1 \dots w_k$. A valid interface sequence $\sigma = (q_0, q_1) \dots (q_{k-1}, q_k)$ is *induced by* w if there is an accepting run of M on w such that for all $i \in [1..k]$, q_i is the state reached by M upon reading the prefix $w_1 \dots w_i$. We further define the *language of induced interface sequences of* S by

$$IIF(S) = \{\sigma \in (Q \times Q)^* \mid \text{There is a } w \in L(S) \text{ that induces } \sigma\}.$$

The induced interface sequences are those that can be derived from a real computation of the program S. Hence, the sequences are capable of witnessing non-emptiness of $L(S)$. We obtain the following:

Lemma 6.11. *Let* $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ *be an SMCP. We have that*

$$L(S) \neq \emptyset \text{ if and only if } IIF(S) \neq \emptyset.$$

Proof. Clearly, each word in $L(S)$ induces an interface sequence. Moreover, each induced interface sequence requires a word in $L(S)$ that induces it. □

Since the number of interface sequences is not bounded, Lemma 6.11 is not well-suited to be turned into a test for non-emptiness of $L(S)$. However, note that a word with k many context switches induces an interface sequence of length precisely $k + 1$. Since we assume the number of context switches to be bounded by cs, we can thus restrict to sequences of length at most $cs + 1$.

Lemma 6.12. *Let $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ be an SMCP and $cs \in \mathbb{N}$ a given upper bound on the number of context switches. We have that*

$$L(S) \cap \text{Context}(\Sigma, t, cs) \neq \emptyset \text{ if and only if } IIF(S) \cap (Q \times Q)^{\leq cs+1} \neq \emptyset.$$

The lemma already suggests an algorithm for deciding BCS. The idea is to iterate over all valid sequences in $(Q \times Q)^{\leq cs+1}$ and test each of them for being an induced interface sequence, a member of $IIF(S)$. Since we require the sequences to be valid, there are at most $O(m^{cs+1})$ sequences to test. This constitutes the first factor in the complexity estimation of Theorem 6.8.

Interface Languages It remains to decide whether a valid interface sequence is actually induced by a word in the language of the program. To this end, we do a preprocessing step that compiles away the dependence on the memory: for each thread, we compute a language making visible the state changes on the memory that the contexts of this thread may induce.

Definition 6.13. Let $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ be an SMCP and A_{id} a thread. The *interface language of A_{id}*, denoted by $IF(A_{id})$, consists of all interface sequences

$$\sigma = (q_1, q_1') \dots (q_k, q_k'),$$

that satisfy $L(A_{id}) \cap (\, L(M(q_1, q_1')) \dots L(M(q_k, q_k'))\,) \neq \emptyset$.

An interface sequence $\sigma \in IF(A_{id})$ shows the state changes of M that the thread A_{id} could induce while running. Note that these sequences do not have to be valid as the thread may be well interrupted by others. One advantage of considering interface languages is that we can efficiently compute a representation of them: the languages are regular. This will become handy when we test whether a valid interface sequence is actually induced.

Lemma 6.14. *Let $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ be an SMCP. We can compute in time $O(|A_{id}|^3 \cdot m^3)$ a finite automaton B_{id} with $L(B_{id}) = IF(A_{id})$.*

Proof. Let the shared memory of S be $M = (Q, \Sigma, \delta_M, q_0, q_f)$. We define B_{id} over the alphabet $Q \times Q$. The states are given by the states of A_{id}. We add a transition from p to p' in B_{id} that is labeled by (q, q') if

$$L(A_{id}(p, p')) \cap L(M(q, q')) \neq \emptyset.$$

Then, $L(B_{id}) = IF(A_{id})$. Computing whether all $L(A_{id}(p, p')) \cap L(M(q, q'))$ are non-empty and constructing B_{id} can be done in time $O(|A_{id}|^3 \cdot m^3)$. □

In our algorithm for solving BCS, we employ Lemma 6.14 as a preprocessing: we construct the automaton B_i for each thread A_i. Note that this takes time at most $O(t \cdot m^3 \cdot b^3)$ as stated in Theorem 6.8.

The next lemma connects the interface languages of the threads with the induced interface sequences of the shared-memory concurrent program.

Lemma 6.15. *Let $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ be an SMCP. We have*

$$IIF(S) = \text{Ш}_{i\in[1..t]} IF(A_i) \cap \{\sigma \in (Q \times Q)^* \mid \sigma \text{ is valid}\}.$$

Proof. We show two inclusions. For the first one, let $\sigma = (q_0, q_1) \dots (q_{k-1}, q_k)$ be an induced interface sequence, an element of $IIF(S)$. There is a word $w \in L(S)$ that induces σ. This means that we can decompose w into contexts $w = w_1 \dots w_k$ and there is an accepting run ρ of M on w of the form:

$$\rho = q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots \xrightarrow{w_{k-1}} q_{k-1} \xrightarrow{w_k} q_k = q_f.$$

Since w also lies in the shuffle of the $L(A_i)$, there are subwords $w^1, \dots, w^t$, partitioning w such that $w^i \in L(A_i)$. Note that the subword w^i appends all contexts of w that are due to thread A_i. Following run ρ, every w^i leads to an (possibly non-valid) interface sequence $\sigma^i = (q_{i_1}, q_{i_2}) \dots (q_{i_\ell}, q_{i_{\ell+1}})$. These partition σ and by construction, we get that $\sigma^i \in IF(A_i)$. Thus, we obtain that $\sigma \in \text{Ш}_{i\in[1..t]} IF(A_i)$ and σ is valid by definition.

For the converse inclusion, let σ be a valid sequence in $\text{Ш}_{i\in[1..t]} IF(A_i)$. Then there are subsequences $\sigma^i \in IF(A_i)$, forming a partition of σ. By the definition of $IF(A_i)$, for each σ^i there is a word $w^i \in L(A_i)$ that follows the state changes in M depicted by σ^i. We compose (shuffle) the w^i in the same order as the σ^i compose to σ. Hence, we get a word w in the shuffle of the $L(A_i)$. The word w follows the states in M as dictated by σ and since σ is valid, we get $w \in L(M)$. Hence, $w \in L(S)$ and it induces σ. □

Solving Shuffle Membership Recall that due to Lemma 6.12, the missing bit to solve BCS is to decide whether a valid interface sequence of length at most $cs + 1$ is induced. With Lemmas 6.14 and 6.15, we are now able to design an efficient test. In fact, it remains to check whether a valid sequence $\sigma \in (Q \times Q)^{\leq cs+1}$ is included in the language $\text{Ш}_{i\in[1..t]} L(B_i)$. Phrased differently, this means we address the following problem, called *shuffle membership*:

Shuffle Mem

Input: Automata $(B_i)_{i\in[1..t]}$ over an alphabet Γ and a word $u \in \Gamma^k$.
Question: Is $u \in \text{Ш}_{i\in[1..t]} L(B_i)$?

The problem Shuffle Mem can be seen as a testing problem. The given word u is a potential computation that leads to an unsafe state and it remains to test whether the threads can actually execute u. If so, u is a valid computation of the program showing that there is a safety violation.

We present the following upper bound for Shuffle Mem, where b is defined like in Theorem 6.8, namely as the maximal size of an automaton B_i.

Theorem 6.16. *Shuffle Mem can be solved in time* $O(2^k \cdot t \cdot k \cdot (b^2 + k \cdot mult(k)))$.

The key to solve Shuffle Mem is fast subset convolution. In order to apply the technique, we give a characterization of the problem in terms of partitions. The following observation constitutes the basis of this characterization. Let $((B_i)_{i\in[1..t]}, u)$ be an instance of Shuffle Mem with $u \in \Gamma^k$. Then, the word u lies in the shuffle of the languages $L(B_i)$ if and only if there are non-overlapping, *scattered* subwords $u_1, \ldots, u_t$ of u that decompose u completely and that satisfy $u_i \in L(B_i) \cup \{\varepsilon\}$ for all $i \in [1..t]$. By scattered, we mean that the subwords do not have to form an infix of u. Note that a subword u_i is the contribution of automaton B_i to the word u.

A decomposition of u into t subwords like above induces a t-partition of the set of positions of u and vice versa. In order to formally prove this result, we need to introduce some more notation:

Definition 6.17. Let $u \in \Gamma^k$ be a word. The *set of positions of* u is given by $Pos(u) = [1..k]$. For a subset $P \subseteq Pos(u)$, we denote by $u[P] \in \Gamma^*$ the subword obtained from u by projecting to the positions in P.

Assume $u \in \text{Ш}_{i\in[1..t]} L(B_i)$ and we are given a decomposition of u into subwords $u_1, \ldots, u_t$ as above. Then we obtain a t-partition $(P_1, \ldots, P_t)$ of $Pos(u)$, where each P_i holds exactly the positions of u_i. Formally, $u[P_i] = u_i$ and moreover we have $u[P_i] \in L(B_i) \cup \{\varepsilon\}$. In turn, when we are given a t-partition $(P_1, \ldots, P_t)$ of $Pos(u)$ where $u[P_i] \in L(B_i) \cup \{\varepsilon\}$, we immediately get a decomposition of u into the subwords $u_i = u[P_i]$ and thus, $u \in \text{Ш}_{i\in[1..t]} L(B_i)$. The discussion proves the following lemma:

Lemma 6.18. *Let* $u \in \Gamma^k$ *be a word and* $(B_i)_{i\in[1..t]}$ *automata over* Γ. *We have* $u \in \text{Ш}_{i\in[1..t]} L(B_i)$ *if and only if there exists a* t*-partition* $(P_1, \ldots, P_t)$ *of the set* $Pos(u)$ *such that* $u[P_i] \in L(B_i) \cup \{\varepsilon\}$ *for all* $i \in [1..t]$.

The lemma yields that deciding Shuffle Mem amounts to finding a t-partition that respects membership in the languages of the given automata. However, the plain subset convolution iterates over all t-partitions. Hence, we need to exclude those that violate the language constraints. This can be achieved by expressing the constraints in terms of characteristic functions.

Definition 6.19. Let $u \in \Gamma^k$ be a word and $(B_i)_{i\in[1..t]}$ automata over Γ. For each $i \in [1..t]$, we define the *characteristic function*

$$f_i : \mathcal{P}(Pos(u)) \to \{0,1\}$$
$$P \mapsto \begin{cases} 1, \text{ if } u[P] \in L(B_i) \cup \{\varepsilon\}, \\ 0, \text{ otherwise.} \end{cases}$$

Note that f_i evaluates to 1 at P if and only if the subword given by the positions in P belongs to the language $L(B_i) \cup \{\varepsilon\}$. By the definition of the subset convolution, we refer to Lemma 3.11, we get that $(f_1 * \cdots * f_t)(Pos(u)) > 0$ if and only if there is a t-partition $(P_1, \ldots, P_t)$ of $Pos(u)$ such that $f_i(P_i) = 1$ for $i \in [1..t]$. With Lemma 6.18, we have proven the desired characterization:

Lemma 6.20. *Let $u \in \Gamma^k$ and $(B_i)_{i\in[1..t]}$ automata over Γ. We have*

$$u \in \text{⧢}_{i\in[1..t]} L(B_i) \text{ if and only if } (f_1 * \cdots * f_t)(Pos(u)) > 0.$$

We can now formulate our algorithm for Shuffle Mem. Given an instance $((B_i)_{i\in[1..t]}, u)$, it first computes the characteristic functions f_i for each $i \in [1..t]$. Then it applies fast subset convolution $t-1$ times to obtain the function $f_1 * \cdots * f_t$. As a last step, it evaluates the function at the set of positions $Pos(u)$ and returns *yes*, if the obtained value is strictly larger than 0. Clearly, otherwise it returns *no*. Correctness immediately follows by Lemma 6.20.

We estimate the complexity of the algorithm. Computing and storing a value $f_i(P)$ for a subset $P \subseteq Pos(u)$ takes time $O(k \cdot b^2)$ since we have to test membership of a word of length at most k in B_i. Hence, computing all f_i takes time $O(2^k \cdot t \cdot k \cdot b^2)$. The functions f_i map into the set $\{0,1\}$. By Corollary 3.13, the $t-1$ convolutions can then be computed in time $O(2^k \cdot k^2 \cdot (t-1) \cdot mult(k))$. Altogether, this concludes the proof of Theorem 6.16.

Algorithm for Bounded Context Switching We summarize the algorithm of BCS. It is stated as Algorithm 6.1. Given an instance (S, cs) with an SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ and a bound $cs \in \mathbb{N}$, the first step is to compute the automata $(B_i)_{i\in[1..t]}$ according to Lemma 6.14. This is performed once and takes time $O(t \cdot m^3 \cdot b^3)$. Then, the algorithm iterates over all valid interface sequences of length at most $cs+1$ and tests each of them for being induced. The latter is decided by applying the algorithm for Shuffle Mem. Since there are $O(m^{cs+1})$ many sequences and the test can be performed in time $\mathsf{Shuff}(b, cs+1, t)$, the proof of Theorem 6.8 follows.

6.2.3 Lower Bounds

We present the lower bounds of the fine-grained analysis of BCS and Shuffle Mem. This includes lower bounds on the running times for both problems

Algorithm 6.1 Bounded Context Switching

Input: An SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ and a bound $cs \in \mathbb{N}$.
Output: *True*, if $L(S) \cap \text{Context}(\Sigma, t, cs) \neq \emptyset$. *False* otherwise.

```
1: compute for each A_i the automaton B_i following Lemma 6.14
2: for each valid σ ∈ (Q × Q)^{≤cs+1} do
3:   if σ ∈ Ш_{i∈[1..t]} L(B_i) then  // Apply algorithm for Shuffle Mem.
4:     return true
5:   end if
6: end for
7: return false
```

which (almost) match the running times of the algorithms obtained in Section 6.2.2 as well as a kernel lower bound for BCS(cs, m), showing that the problem only admits a non-polynomial kernel.

Lower Bound for Bounded Context Switching We begin with the lower bound for BCS. Ideally, we would like to obtain a bound of the form $2^{o(cs\cdot\log(m))}$ matching the running time of Algorithm 6.1. The tool of choice would be a reduction from $k\times k$-Clique to BCS that keeps the parameter linear according to Lemma 5.14. However, when setting up such a reduction, one faces the typical *square problem*: checking whether k vertices are connected requires $O(k^2)$ communications among the chosen vertices. This translates to $O(k^2)$ context switches. Hence, we would only obtain $m^{o(\sqrt{cs})}$ as a lower bound.

To overcome this, we follow the result of Marx, stated as Theorem 5.20. Recall that it shows that Subgraph Isomorphism cannot be solved in time $f(e)\cdot n^{o(e/\log(e))}$ for any computable function $f(e)$, unless the ETH fails. The parameter e is the number of edges of the smaller graph that gets embedded. We show that one can establish a reduction from Subgraph Isomorphism to BCS mapping the number of edges linearly to the number of context switches. This excludes an algorithm for BCS running in time $g(cs)\cdot n^{o(cs/\log(cs))}$, for any computable function $g(cs)$. The reduction is shown in the following theorem.

Theorem 6.21. *Unless ETH fails, BCS cannot be solved by an algorithm running in time $g(cs)\cdot n^{o(cs/\log(cs))}$ where $g(cs)$ is some computable function.*

Proof. Let (H, G) be an instance of Subgraph Isomorphism and $e = |E(H)|$ the number of edges of H. The idea is to construct an SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ such that $L(S)$ contains a word with at most $cs = 2e$ context switches if and only if H can be embedded into the graph G.

Let $V(G) = \{w_1, \ldots, w_\ell\}$, $k = |V(H)|$, and assume that H does not contain isolated vertices since these are not relevant for the complexity of Subgraph Isomorphism. Hence, we have that $k \leq 2e$. To embed H into G, we need to

find an injective map from $V(H)$ into $V(G)$ preserving edges. We can express such a map over the alphabet $\Sigma = V(H) \times V(G)$. Intuitively, a word in Σ shows how the individual vertices of H are mapped. Roughly, the idea is then to use the shared memory M to output all injective maps and the threads A_i to ensure that the edges of H are also present in G.

For capturing all injective maps, we let M accept the following language:

$$\{(v_1, w_{i_1})^{d_1} \dots (v_k, w_{i_k})^{d_k} \mid v_i \in V(H),\ i_1 < \dots < i_k, \text{ and } \textstyle\sum_{i \in [1..k]} d_i = 2e\}.$$

The order among the w_{i_j} is important to avoid that various vertices of H are mapped to the same vertex of G. Hence, $L(M)$ indeed describes injective but potentially partial maps. The exponents d_i are needed to verify that the chosen map preserves the edges of H. This will become clear in a moment.

For each edge $e_i = \{v_s, v_t\}$ of H, we construct a thread A_i. It ensures that the edge is safely transported to G by the suggested map of M. To this end, A_i accepts all words of length 2 describing a successful translation of e_i:

$$\bigcup_{\{w_{i_s}, w_{i_t}\} \in E(G)} \begin{cases} (v_s, w_{i_s}).(v_t, w_{i_t}), \text{ if } i_s < i_t \\ (v_t, w_{i_t}).(v_s, w_{i_s}), \text{ otherwise.} \end{cases}$$

Note that a word in $L(M)$ has length exactly $2e$. Hence, each thread $A_1, \dots, A_e$ has to contribute exactly 2 times when verifying that the chosen map is compatible with the edges. Once for the start and once for the end vertex. This has two consequences. First, since none of the vertices of H is isolated, each is adjacent to at least one edge and will therefore appear in the contribution of an A_i. Therefore, each vertex is present in the map suggested by M. This shows that the map is indeed complete and not partial. Second, the computation requires at most $2e$ context switches.

Altogether, we obtain the desired reduction. Correctness and the fact that this is indeed a polynomial-time reduction are proven in Appendix B.1.1. □

Since the size of the shared memory m is smaller than the input size n, Theorem 6.21 implies the desired lower bound: BCS cannot be solved in time $m^{o(cs/\log(cs))}$, unless the ETH fails. Note that the lower bound almost matches the running time of the presented algorithm, only the $\log(cs)$-gap remains.

Corollary 6.22. *Unless ETH fails, BCS cannot be solved in time $m^{o(cs/\log(cs))}$.*

Kernel Lower Bound for Bounded Context Switching We provide a kernel lower bound for the parameterization BCS(cs, m). It shows that the existence of a polynomial kernel for the problem is rather unlikely. Hence, polynomial-time preprocessing techniques for BCS might only be applicable with limited success. Even if they can reduce the total number of instances, in the end a large problem kernel of difficult instances remains.

We follow the framework presented in Section 5.4 and set up a cross-composition from 3-SAT into BCS(cs, m). Recall that, by Theorem 5.44, this implies the kernel lower bound for the latter problem under the assumption that the inclusion NP $\subseteq$ coNP/poly does not hold.

Theorem 6.23. *BCS(cs, m) does not admit a poly. kernel, unless* NP $\subseteq$ coNP/poly.

Proof. First, we need a suitable polynomial equivalence relation. Assume some standard encoding of 3-SAT-instances over a finite alphabet Γ. Then, we can employ the polynomial equivalence relation $\mathcal{R}$ of Example 5.41. It identifies two formulas φ and ψ as equivalent if both are proper 3-SAT-instances and have the same number of clauses and variables.

We describe the cross-composition. Let $\varphi_1, \ldots, \varphi_t$ be instances of 3-SAT, all equivalent with respect to $\mathcal{R}$. Then, each φ_i has exactly ℓ clauses and k variables. We may assume the set of variables to be $\{x_1, \ldots, x_k\}$. To handle their evaluation, we introduce the threads $A_i, i \in [1..k]$, each storing the current evaluation of variable x_i. We further construct a thread B. It picks one of the t formulas φ_j and tries to satisfy it. To this end, B iterates through each of the ℓ clauses of formula φ_j, chooses one out of its three variables, and requests the corresponding value to satisfy the clause.

The request of B is synchronized with the shared memory M. Assume B wants variable x_i to be evaluated to v. After the sent request, M either ensures that x_i actually has the requested value v or immediately stops the computation. Checking that x_i evaluates to v is achieved by a synchronization with the corresponding thread A_i, keeping the evaluation of x_i.

If the computation succeeds without stopping, B managed to satisfy φ_j. Phrased differently, this means that the constructed SMCP implements the logical OR of the given 3-SAT-instances. Note that the number of context switches is $O(\ell)$ since there are ℓ requests by B. Moreover, the size of the memory is bounded by $O(k)$. This ensures that the parameters cs and m of the constructed BCS-instance are bounded by the maximal size of an φ_j and independent of their number t. Since the above construction takes polynomial time, all conditions on a cross-composition are met. We provide a more detailed construction and a proof of correctness in Appendix B.1.2. □

Lower Bound for Shuffle Membership We prove it unlikely that the problem Shuffle Mem can be solved in time $O^*((2-\varepsilon)^k)$ for an $\varepsilon > 0$. Therefore, the convolution-based algorithm presented in Theorem 6.16 is indeed optimal. For the lower bound, we employ the SCON. The key is to set up a tight linear reduction from Set Cover to Shuffle Mem.

To the best of our knowledge, Shuffle Mem is the first problem in the context of verification that admits a lower bound based on the SCON. However, like other verification problems [96], Shuffle Mem shows the *typical* attributes making it suitable for such a lower bound. In fact, Shuffle Mem admits an

$O^*(2^k)$-time algorithm only via a non-trivial dynamic programming algorithm and it does not (easily) permit a strong linear reduction from SAT.

Before stating the result, recall that an instance $(U, \mathcal{F}, r)$ of Set Cover consists of an universe U of size $n = |U|$, a family of sets $\mathcal{F} \subseteq \mathcal{P}(U)$, and an integer r. The parameter k of Shuffle Mem refers to the length of the given word. We provide a tight linear reduction, a polynomial-time reduction from Set Cover to Shuffle Mem, that ensures $k = n+r+c$, where $c \in \mathbb{N}$ is a constant. By Lemma 5.32, this implies the lower bound for Shuffle Mem.

Theorem 6.24. *Unless SCON fails, Shuffle Mem cannot be solved in $O^*((2-\varepsilon)^k)$.*

Proof. Let $(U, \mathcal{F}, r)$ be an instance of Set Cover with universe $U = \{u_1, \ldots, u_n\}$. We construct an instance $((B_S)_{S\in\mathcal{F}}, w)$ of Shuffle Mem such that $k = |w| = n+r$ and $w \in \text{⧢}_{S\in\mathcal{F}} L(B_S)$ if and only if U can be covered by r sets of $\mathcal{F}$.

The underlying alphabet of the instance is $\Gamma = U \cup [1..t]$. We introduce an automaton B_S for each set S in the given family $\mathcal{F}$. The automaton consists of two states and it accepts the following language:

$$L(B_S) = \{u^*.j \mid u \in S, j \in [1..t]\}.$$

We further define the word $w = w_U.w_r \in \Gamma^*$ to be the concatenation of $w_U = u_1 \ldots u_n$ and $w_r = 1 \ldots r$. The prefix w_U ensures that each element of U gets covered while w_r ensures that we do not use more than r sets. The length of w is $n+r$. Hence, we constructed an instance $((B_S)_{S\in\mathcal{F}}, w)$ of Shuffle Mem with $k = n + r$. Note that the construction takes polynomial time.

For the correctness, first assume that U can be covered by r sets $S_1, \ldots, S_r$ of $\mathcal{F}$. We use an interleaving of the automata B_{S_i} with $i \in [1..r]$ to read w. First, each B_{S_i} reads those letters u_j of w_U with $u_j \in S_i$. Note that an element u_j can lie in more than one of the S_i. In this case, u_j is read by one of the corresponding B_{S_i}. After reading w_U, automaton B_{S_1} reads the letter 1, followed by B_{S_2} reading 2. The process is continued until r is read by B_{S_r} and all involved automata reach their final state. This shows that $w \in \text{⧢}_{S\in\mathcal{F}} L(B_S)$.

Now let $w \in \text{⧢}_{S\in\mathcal{F}} L(B_S)$. Since $w_t = 1 \ldots t$, we get that exactly t of the automata B_S are involved in reading the word w. We may assume that these are $B_{S_1}, \ldots, B_{S_t}$. Then the prefix w_U is read by interleaving the finite automata B_{S_i}. This means that each $u \in U$ lies in (at least) one of the sets S_i and hence, $S_1, \ldots, S_t \in \mathcal{F}$ forms a cover of the universe U. □

6.2.4 Intractability Results

The search for FPT-algorithms around BCS revealed that some parameterizations of the problem are actually intractable. In fact, parameterizing only in the number of context switches *cs* does not suffice to obtain a tractability result: BCS(*cs*) turned out to be W[1]-complete. Even adding the number of threads t as a parameter does not change this complexity. The following theorem summarizes the intractability of BCS in the two parameters.

Theorem 6.25. *BCS(cs) and BCS(cs, t) are both* W[1]*-complete.*

We prove the theorem by setting up a chain of reductions. To this end, recall that the problem Short TM Acceptance(k) is W[1]-complete, we refer to Theorem 4.13. Moreover, note that the parameterized problem Subgraph Isomorphism(k) is W[1]-hard since it is a generalization of Clique(k) and the latter is W[1]-hard by Corollary 4.15. The following shows the mentioned chain. Each arrow marks one parameterized reduction:

$$\text{Subgraph Iso}(k) \to \text{BCS}(cs) \to \text{BCS}(cs, t) \to \text{Short TM Acceptance}(k).$$

The chain implies Theorem 6.25. Note that the first reduction, from Subgraph Isomorphism(k) to BCS(cs), follows from Theorem 6.21. The theorem presents a reduction from Subgraph Isomorphism(e) to BCS(cs) such that $cs = O(e)$. Since the number of edges e is always bounded by k^2, where k is the number of vertices, the presented reduction is also a parameterized reduction from Subgraph Isomorphism(k) to BCS(cs). The second reduction in the chain is proven in the following lemma.

Lemma 6.26. *BCS(cs) reduces to BCS(cs, t).*

Proof. Let (S, cs) be an instance of BCS with SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$. We construct an equivalent instance $(S', 2cs)$ of BCS where S' contains $cs+1$ many threads $B_i, i \in [1..cs+1]$. Consequently, the number of threads is bounded by a function in cs and thus, we obtain the desired parameterized reduction.

During a computation of S with cs many context switches, at most $cs+1$ many different threads A_j can interleave. The idea is to construct S' in such a way that its threads B_i can choose to simulate a particular A_j. To this end, S' contains $cs+1$ copies of the same thread B. The automaton is the union of all A_j, preceded by a new initial state. In this state, B may choose to simulate an A_j by reading a special letter $\#_j$ or it chooses to simulate none of the threads and immediately accepts by reading the letter #. Its language is given by

$$L(B) = \# \cup \bigcup_{j\in[1..t]} \#_j.L(A_j).$$

A computation of the memory M' of S' is split into two phases. In the first phase, M' activates at most $cs+1$ many B_i by reading a word from

$$L(M'_1) = \{\#_{i_1}.\#^{n_1}.\#_{i_2}.\#^{n_2} \dots \#_{i_\ell}.\#^{n_\ell} \mid \ell \geq 1, \ell + \textstyle\sum n_i = cs+1, i_1 < \dots < i_\ell\}.$$

The B_i synchronize with M' on the read word and choose to simulate the corresponding threads A_j or are deactivated by reading #. Note that the symbols $\#_j$ appear in increasing order. This ensures that none of the threads A_j of S is simulated by more than one automaton B_i.

In the second phase, when the threads to be simulated are chosen, M' simulates M. Then, M' and the remaining active B_i can simulate a computation of S that involves the chosen threads A_j. Formally,

$$L(M') = L(M'_1).L(M).$$

Note that the first phase requires cs many context switches. Since we simulate a computation of S with cs many context switches in the second phase, we have a total of $2cs$ many context switches. The proof of correctness and a more detailed construction are provided in Appendix B.1.3. □

It remains to show the last parameterized reduction, from BCS(cs, t) to Short TM Acceptance(k). Recall that the latter problem asks whether a given Turing Machine accepts a given input in at most k steps.

Lemma 6.27. *BCS(cs, t) reduces to Short TM Acceptance(k).*

Proof. We elaborate on the idea, a detailed proof is given in Appendix B.1.4. Let (S, cs) be an instance of BCS with SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$. We construct a Turing Machine N and a word w such that N accepts w in at most $t \cdot (cs + 3)$ steps if and only if (S, cs) is a yes-instance.

Machine N works on t cells of its tape. The i-th cell holds the current state of thread A_i. The states of the memory M and a context counter are stored within the states of N. The idea is that the machine simulates a whole context of a computation of S in a single transition. Assume we want to simulate a context where the memory M interacts with a thread A_i. Machine N detects that M is in state q and A_i is in state p. The former is stored in the states of N, the latter is the content of the i-th cell. Machine N can then write a state p' into the cell and move to a state q' if $L(M(q, q')) \cap L(A_i(p, p')) \neq \emptyset$. The non-emptiness of the intersection ensures that there is a corresponding context. Note that we can precompute this information for all states (q, q', p, p') in polynomial time. Initially, N is given the input word $w = q_1^0 \dots q_t^0$. It stores the initial state q_i^0 of thread A_i in cell i.

After a simulation step, the counter for contexts is increased. If it reaches $cs + 1$, it is checked whether M and the participating threads all reached their final state. This amounts to a successful computation of S with at most cs many context switches. Note that for the complete simulation of $cs + 1$ contexts, N requires at most $t \cdot (cs + 1)$ many steps. The machine has to move left and right on a tape of length t to pick different threads. In the end, when the simulation has stopped, N requires another $2 \cdot t$ steps to check whether each participating thread accepts. This is checked by moving once to the left end of the tape and then perform a scan to the right end. Hence, the exact number of steps is $t \cdot (cs + 3)$. It is thus bounded by the parameters cs and t. Consequently, the construction describes a parameterized reduction. □

6.3 Local Variant of Bounded Context Switching

We present the results of the fine-grained complexity analysis of LBCS, a local variant of BCS. In contrast to BCS, where the budget of context switches is shared among all threads globally and communication is not restricted to a certain pattern, LBCS assumes that each thread has its own budget of context switches and communication happens along a given *scheduling graph*. The complexity of such graphs is measured in terms of the *scheduling dimension*. We formally define the scheduling dimension and the problem LBCS in Section 6.3.1. Subsequently, in Section 6.3.2, we present an algorithm for LBCS. As an application of it, we obtain an upper bound for the problem Round Rob in Section 6.3.3. The upper bound is complemented by a matching lower bound which also closes the gap for LBCS.

6.3.1 Scheduling Dimension and LBCS

The analysis presented in Section 6.2 shows that the number of context switches plays a central role in the fine-grained complexity of BCS. Although communication among the threads happens freely, the number of context switches limits the amount of exchanged information. In this section, we follow a different approach to restrict the communication. Instead of putting a bound on how often threads can communicate with each other, we limit the complexity of their scheduling. This requires a formal definition of schedules and a new measure capturing their complexity.

Scheduling Graphs A schedule of a computation is represented by a suitable graph. It reflects the order in which the threads take turns. To formally define it, recall that a multigraph may contain multiple edges between vertices. Moreover, let t again denote the number of threads.

Definition 6.28. Let $w \in \text{Context}(\Sigma, t, cs)$ be a word along with its decomposition into contexts $w = w_1 \dots w_k$. The *scheduling graph of* w is a directed multigraph $G(w) = (V, E)$, where $V \subseteq [1..t]$ contains one vertex for each thread participating during the contexts of w:

$$V = \{i \in [1..t] \mid \exists \ell \in [1..k] : w_\ell \in (\Sigma \times \{i\})^*\}.$$

The edge weights $E : V \times V \to \mathbb{N}$ are defined as follows. Value $E(i, j)$ is the number of times the context switches from thread i to thread j in w. Formally,

$$E(i, j) = |\{\ell \in [1..k] \mid w_\ell \in (\Sigma \times \{i\})^* \text{ and } w_{\ell+1} \in (\Sigma \times \{j\})^*\}|.$$

A scheduling graph does not contain loops — we have $E(i, i) = 0$. Hence, in the following we refer to directed multigraphs without loops as graphs. Before we turn to properties of scheduling graphs, we consider an example.

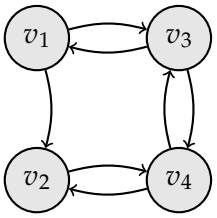


Figure 6.3: The scheduling graph $G(w) = (V, E)$ of the word described in Example 6.29. There is one vertex v_i for each thread A_i. The edges are obtained from the order in which the threads take turns. Note that the initial vertex is v_1, the final vertex is v_2. The degree of G is $\deg(G) = 2$.

Example 6.29. Assume we are given an SMCP S with four threads A_1, A_2, A_3, and A_4. Moreover, let $w \in \text{Ш}_{i\in[1..4]} L(A_i)$ be a word the contexts of which are due to the threads taking turns in the following order:

$$\sigma = A_1 \,.\, A_3 \,.\, A_4 \,.\, A_2 \,.\, A_4 \,.\, A_3 \,.\, A_1 \,.\, A_2.$$

This means that the first context of w is due to A_1, the second is due to A_3, and so on. We constructed the scheduling graph of w in Figure 6.3.

Note that A_1 has the processor two times during the computation of w. In the graph, this is reflected by the degree of vertex v_1, the maximum number of incoming or outgoing edges. This also holds for the other threads.

The observation made in the example is true in general. The degree of a vertex corresponds to the number of times the thread has the processor. To formalize it, we recall the definition of the degree of a vertex/graph.

Definition 6.30. Let $G = (V, E)$ be a graph and $v \in V$ a vertex. The *indegree of* v and the *outdegree of* v are defined to be the number of incoming edges, and the number of outgoing edges respectively. Formally, we have

$$\text{indeg}(v) = \sum_{u\in V} E(u, v) \quad \text{and} \quad \text{outdeg}(v) = \sum_{u\in V} E(v, u).$$

The *degree of* v is the maximum over indegree and outdegree. We denote it by $\deg(v) = \max\{\text{indeg}(v), \text{outdeg}(v)\}$. The *degree of* G is the maximum among the degrees of all vertices, $\deg(G) = \max\{\deg(v) \mid v \in V\}$.

To see the correspondence between how often a thread contributes during a computation and the degree, observe that a scheduling graph can only have three kinds of vertices. The *initial vertex* is the only vertex v where

$$\text{indeg}(v) = \text{outdeg}(v) - 1.$$

The corresponding thread has the processor $\mathrm{outdeg}(v)$ many times. The *final vertex* is the only vertex v in the graph that satisfies

$$\mathrm{outdeg}(v) = \mathrm{indeg}(v) - 1.$$

This means that the corresponding thread computes for $\mathrm{indeg}(v)$ many contexts. For all other (usual) vertices, indegree and outdegree coincide. An example is given in Figure 6.3. Note that any scheduling graph either has one initial, one final, and only usual vertices or only consists of usual vertices. The latter holds if the computation starts and ends in the same thread.

Scheduling Dimension Our goal is to measure the complexity of schedules. Intuitively, a schedule is simple if the threads take turns following some pattern, like in round robin where they are scheduled in a cyclic way. In such a pattern, heavily interacting threads are scheduled adjacent to each other. Since a large part of the communication happens between these threads, the idea is to join them and to see them as a new thread. The new thread hides the interior communication of the joined ones and eases the scheduling.

We formalize the idea. On scheduling graphs, joining two threads amounts to *contracting* two vertices. This means we join both vertices into a single vertex and add up the weights for incoming and outgoing edges.

Definition 6.31. Let $G = (V, E)$ be a graph and $v_1, v_2 \in V$ two vertices. The *contraction of v_1 and v_2* into a new vertex $v \notin V$ is defined to be the graph $G[v_1, v_2 \mapsto v] = (V', E')$, where $V' = (V \setminus \{v_1, v_2\}) \cup \{v\}$ and E' is defined by:

$$E'(u, v) = E(u, v_1) + E(u, v_2) \quad \text{and} \quad E'(v, u) = E(v_1, u) + E(v_2, u)$$

for $u \in V \setminus \{v_1, v_2\}$. For all other vertices $u, w \in V \setminus \{v_1, v_2\}$, the edge weights are similar to that of G, we have $E'(u, w) = E(u, w)$.

To determine the complexity of a scheduling, we repeatedly contract scheduling graphs to a single vertex and measure the degree of the intermediary appearing graphs. The idea is that a simple schedule can be contracted without increasing the degree by much since we contract only heavily interacting threads. To formalize repeated contraction, we use a notion of *contraction processes*. Along these processes we can then define our measure for the complexity of a schedule, the *scheduling dimension*.

Definition 6.32. Let $G = (V, E)$ be a graph. A *contraction process of G* is a sequence of graphs $\pi = G_1, \ldots, G_{|V|}$ such that $G_1 = G$, $G_{|V|}$ consists of only a single vertex, and for $i \in [1..|V| - 1]$ we have

$$G_{i+1} = G_i[v_1, v_2 \mapsto v],$$

for some $v_1, v_2 \in V(G_i)$ and $v \notin V(G_i)$. The *degree of π* is the maximal degree of a graph occurring in the process: $\deg(\pi) = \max_{i \in [1..|V|]} \deg(G_i)$.

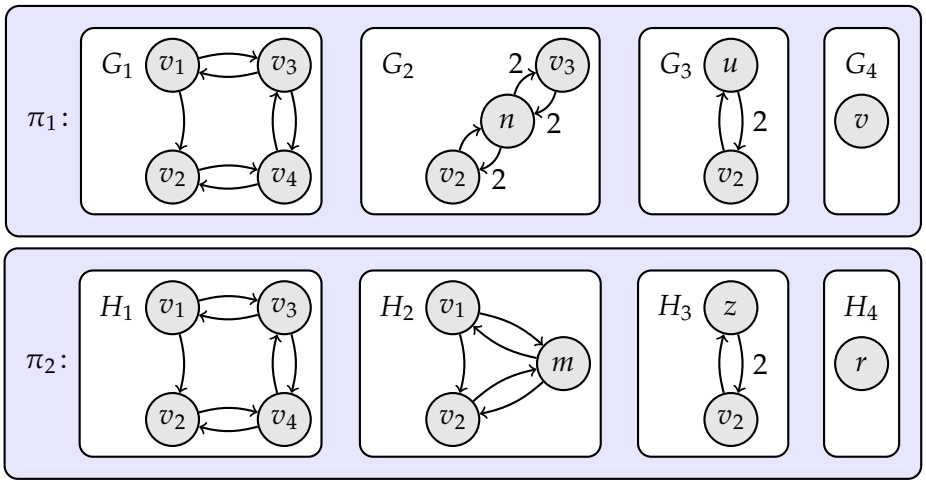


Figure 6.4: The upper part shows the contraction process π_1, given by $G_1 = G$, $G_2 = G_1[v_1, v_4 \mapsto n]$, $G_3 = G_2[v_3, n \mapsto u]$, and $G_4 = G_3[v_2, u \mapsto v]$. The lower part shows the contraction process π_2, given by $H_1 = G$, $H_2 = H_1[v_3, v_4 \mapsto m]$, $H_3 = H_2[v_1, m \mapsto z]$, and $H_4 = H_3[v_2, z \mapsto r]$.

The *scheduling dimension of* G is the minimal degree of a contraction process. Formally, $sdim(G) = \min\{\deg(\pi) \mid \pi \text{ is a contraction process of } G\}$.

The scheduling dimension measures the minimal communication demands in a scheduling graph. For actual contraction processes, these demands may vary: different processes may not have the same degree. This is the reason why the scheduling dimension is defined as the minimum over all contraction processes. The upcoming example illustrates the discussion.

Example 6.33. Consider the scheduling graph $G = G(w)$ from Figure 6.3. We construct two different contraction processes for G that vary in their degree. The first process π_1 is given by the following sequence:

$$G_1 = G,\ G_2 = G_1[v_1, v_4 \mapsto n],\ G_3 = G_2[v_3, n \mapsto u],\ G_4 = G_3[v_2, u \mapsto v].$$

The contraction process π_1 is illustrated in the upper part of Figure 6.4. Note that $\deg(\pi_1) = \deg(G_2) = 4$ since the vertex $n \in V(G_2)$ has an outdegree of 4. The first graph G_1 has degree 2. Hence, π_1 significantly increases the degree. The reason is that in its first step, it contracts v_1 and v_4 — two vertices that do not interact. This stands in contrast to the idea of an efficient contraction process only contracting vertices that heavily interact.

The second contraction process π_2 follows the idea. It is given by the below sequence and illustrated in the lower part of Figure 6.4.

$$H_1 = G,\ H_2 = H_1[v_3, v_4 \mapsto m],\ H_3 = H_2[v_1, m \mapsto z],\ H_4 = H_3[v_2, z \mapsto r].$$

The degree of π_2 is 2. Therefore, the process does not increase the degree of the originally given graph G. Note that this implies $sdim(G) = 2$.

The scheduling dimension is closely related to the *carving width*. This measure was introduced by Seymour and Thomas in [301] and has received some attention in parameterized complexity [40, 170, 315]. The idea of carving width is to measures the complexity of a communication graph. These are undirected multigraphs where each edge weight represents a number of communication demands (calls) between two vertices, or locations, quite similar to scheduling graphs. To route these calls efficiently, one is interested in finding a routing tree that minimizes the needed bandwidth. The carving width is the minimal required bandwidth among all such routing trees.

To relate carving width and scheduling dimension, we need to turn a directed multigraph $G = (V, E)$ without loops into an undirected multigraph $G' = (V, E')$ the following way. We keep the vertices V and assign the weights $E'(u, v) = \max\{E(u, v), E(v, u)\}$ for all $u, v \in V$. The following holds.

Lemma 6.34. *For any directed multigraph G without loops, we have*

$$sdim(G) \leq cw(G') \leq 2 \cdot sdim(G).$$

Note that $cw(G')$ denotes the carving width of G'. For a formal definition of carving width and a proof of Lemma 6.34, we refer to Appendix B.1.5.

Although scheduling dimension and carving width are closely related, we suggest the former as a measure for the complexity of schedules. The carving width is the natural measure for undirected communication calls. However, a scheduling leads to a notion of directed graphs accentuating the need for a measure for directed communication calls. If threads are tightly coupled, they should be grouped together to one thread. This leads to a contraction process rather than a routing tree and consequently to the scheduling dimension.

Local Variant of Bounded Context Switching With the scheduling dimension at hand, we can bound the complexity of schedules that are allowed to occur in a program. The first natural question to ask might be the following: given an SMCP S and a bound $s \in \mathbb{N}$, does $L(S)$ contains a word w such that $sdim(G(w)) \leq s$? This is a variant of BCS focusing on schedules of bounded dimension. We provide an analysis of the problem in Appendix B.1.6.

However, often we know about the precise scheduling of a program, for instance in round robin. Therefore we consider a variant of BCS that fixes the scheduling to a given graph of bounded scheduling dimension and asks whether there is a computation following the fixed schedule. We formalize the problem. Let $t \in \mathbb{N}$ and $G = (V, E)$ a graph with $V = [1..t]$. We define

$$Sched(\Sigma, t, G) = \{w \in (\Sigma \times [1..t])^* \mid G(w) = G\}.$$

The *local bounded context switching* problem takes a shared-memory concurrent program S, a graph G, and a contraction process of degree s and asks whether there exists a word in the intersection $L(S) \cap \mathit{Sched}(\Sigma, t, G)$.

LBCS

Input: An SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$, a graph G, and a contraction process π of G with $\deg(\pi) = s$.

Question: Is $L(S) \cap \mathit{Sched}(\Sigma, t, G) \neq \emptyset$?

6.3.2 An Algorithm for LBCS

We present an algorithm for LBCS showing that parameterizing by the size of the memory m and by the bound on the scheduling dimension s yields a problem in FPT. As in the algorithm for BCS, presented in Section 6.2.2, the idea is to use interface sequences instead of explicit words. In the beginning, each vertex or thread in the given scheduling graph is represented by its interface language. Then, the key is to mimic the given contraction process on interface sequences. This requires a suitable contraction operator on the latter. With the operator at hand, it is only left to check whether mimicking the complete contraction process yields an interface sequences describing a computation of the memory from the initial to the final state.

Theorem 6.35. *LBCS can be solved in time* $O((2m)^{4s} \cdot s \cdot t^2)$.

Contraction Operator We start by developing the needed contraction operator for interface sequences. To this end, let us fix an SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ with shared memory given by $M = (Q, \Sigma, \delta_M, q_0, q_f)$.

Definition 6.36. Let $\sigma \in (Q\times Q)^k$ and $\tau \in (Q\times Q)^\ell$ be two interface sequences. Moreover, let $\rho \in \sigma \amalg \tau \in (Q \times Q)^{k+\ell}$. An *out-contraction of* ρ is a subsequence

$$(q_1, q_1').(q_2, q_2')$$

of ρ such that (q_1, q_1') belongs to σ, (q_2, q_2') belongs to τ, and $q_1' = q_2$. An out-contraction marks a position where a computation of the memory switches from σ to τ. Similarly, an *in-contraction of* ρ is a subsequence $(q_1, q_1').(q_2, q_2')$ of ρ where (q_1, q_1') belongs to τ, (q_2, q_2') belongs to σ, and $q_1' = q_2$.

An out/in-contraction $(q_1, q_1').(q_2, q_2')$ of ρ can be *contracted* to (q_1, q_2'). This yields a new interface sequence ρ' with the out/in-contraction replaced by (q_1, q_2'). We may also contract sequences of out/in-contractions. Let

$$\bar{\rho} = \sigma_1.\tau_1 \ldots \sigma_r.\tau_r$$

be a subsequence of ρ with $\sigma_i, \tau_i \in Q \times Q$, $\sigma_i.\tau_i$ an out-contraction, and $\tau_i.\sigma_{i+1}$ an in-contraction. Moreover, let $\sigma_1 = (q_1, q_1')$ and $\tau_r = (q_{2r}, q_{2r}')$. The *contraction of* $\bar{\rho}$ is (q_1, q_{2r}'). This results in the new interface sequence ρ', where $\bar{\rho}$ is replaced by (q_1, q_{2r}'). Note that we can similarly contract sequences starting with an in-contraction and ending with an out/in-contraction.

Intuitively, an out-contraction of ρ corresponds to an edge in the scheduling graph from the thread A_σ inducing σ to the thread A_τ inducing τ. It marks where the context switches from A_σ to A_τ. Similarly, an in-contraction is an edge in the reverse direction. When the given contraction process joins the vertices of A_σ and A_τ, we need to contract their interface sequences. This can only happen at out- and in-contractions. However, we cannot just contract each appearing out/in-contraction. In fact, the scheduling graph makes explicit their precise numbers: assume there are i edges from A_σ to A_τ and j edges from A_τ to A_σ. Then we must contract exactly i out-contractions and j in-contractions. The following operator takes these numbers into account.

Definition 6.37. Let $\sigma \in (Q \times Q)^k$ and $\tau \in (Q \times Q)^\ell$ be two interface sequences and $\rho \in \sigma \text{⧢} \tau \in (Q \times Q)^{k+\ell}$. Moreover, let $i, j \in \mathbb{N}$ be two integers. An *i-j-contraction of* ρ is an interface sequence $\rho' \in (Q \times Q)^{k+\ell-(i+j)}$ obtained by contracting subsequences of ρ, in total consisting of exactly i out-contractions and j in-contractions. The *language of i-j-contractions of* ρ is denoted by

$$con_{(i,j)}(\rho) = \{\rho' \in (Q \times Q)^{k+\ell-(i+j)} \mid \rho' \text{ is an } i\text{-}j\text{-contraction of } \rho\}.$$

The *directed contraction product of σ and τ* is the operator defined by

$$\sigma \odot_{(i,j)} \tau = \bigcup_{\rho \in \sigma \text{⧢} \tau} con_{(i,j)}(\rho).$$

The contraction product will be an essential part of our algorithm for the problem LBCS. We will later consider in more detail how the product is computed. But first we illustrate its definition with an example.

Example 6.38. Consider the two interface sequences $\sigma = (q_1, q_2).(p_1, p_2)$ and $\tau = (q_2, p_1).(p_2, q_1)$ in $(Q \times Q)^2$. We compute the product $\sigma \odot_{(1,1)} \tau$. According to the definition, we consider all sequences $\rho \in \sigma \text{⧢} \tau$ and determine the language $con_{(1,1)}(\rho)$. The directed contraction product is the union of these. The first element $\rho_1 \in \sigma \text{⧢} \tau$ that we consider is given by

$$\rho_1 = (q_1, q_2).(p_1, p_2).(q_2, p_1).(p_2, q_1).$$

Note that ρ_1 neither contains an out- nor an in-contraction. Consequently, we obtain that $con_{(1,1)}(\rho_1) = \emptyset$. For the same reason, this also holds for all the other sequences in the shuffle $\sigma \text{⧢} \tau$, except for the following:

$$\rho_2 = (q_1, q_2).(q_2, p_1).(p_1, p_2).(p_2, q_1).$$

The sequence ρ_2 has two out-contractions, namely $(q_1, q_2).(q_2, p_1)$ and $(p_1, p_2).(p_2, q_1)$ and one in-contraction $(q_2, p_1).(p_1, p_2)$. Since we need to contract exactly one out-contraction but two are available, we need to distinguish two cases: either we contract $(q_1, q_2).(q_2, p_1)$ or we contract $(p_1, p_2).(p_2, q_1)$. For the in-contraction there is no choice. Hence, $con_{(1,1)}(\rho_2)$ is given by

$$con_{(1,1)}(\rho_2) = \{(q_1, p_2).(p_2, q_1), (q_1, q_2).(q_2, q_1)\}.$$

Finally, the directed contraction product is $\sigma \odot_{(1,1)} \tau = con_{(1,1)}(\rho_2)$.

We will mostly employ the directed contraction product on sets of interface sequences. Note that the operator can be easily extended. For two sets $\Delta \subseteq (Q \times Q)^k$ and $\Gamma \subseteq (Q \times Q)^\ell$, the *directed contraction product* is the set

$$\Delta \odot_{(i,j)} \Gamma = \{\sigma \odot_{(i,j)} \tau \mid \sigma \in \Delta, \tau \in \Gamma\}.$$

Iteration Algorithm We elaborate on the algorithm for LBCS, proving Theorem 6.35. Let (S, G, π) be an instance of LBCS with SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$, graph G, and contraction process π of degree s. The idea is to use the directed contraction product along the order depicted by π. However, note that this order does not reflect the order in which the threads take turns in a computation of S. In particular the first vertices contracted by π do not need to be the first threads acting in a computation. It is up to us to find the first thread.

Recall that a scheduling graph either has a designated initial and a final vertex or only consists of usual vertices. Hence, in the former case, initial and final vertex are given. We develop an algorithm handling this case. In the latter case, initial and final vertex coincide. We iterate through all vertices, designate any to be initial (and final), and run the algorithm for the first case.

To describe the algorithm, recall that $G = (V, E)$ contains exactly one vertex for each thread of S, $V = [1..t]$. Let the initial vertex of G be v_0 and the final vertex be v_f. Moreover, let the contraction process π be given by the sequence

$$\pi = G_1, \ldots, G_t,$$

where $G_1 = G$ and G_t is a single vertex. Note that π has length exactly t.

The algorithm is an iteration along π. We summarize it as Algorithm 6.2. It starts by assigning to each vertex $v \in V \setminus \{v_0, v_f\}$ its interface language of length $\deg(v)$. Recall that $\deg(v)$ is the number of times that thread v contributes during a computation according to G. We assign the set

$$S_v = IF(A_v) \cap (Q \times Q)^{\deg(v)}.$$

For v_0 and v_f, we assign the sets S_{v_0} and S_{v_f}, putting additional requirements on the interface sequences. In S_{v_0}, the first component of the first pair

needs to be q_0 — the initial state of M. Similarly, S_{v_f} requires the second component of the last pair to be q_f — the final state of M. Formally, we assign

$$S_{v_0} = IF(A_{v_0}) \cap \{(q_0, q).w \mid q \in Q, w \in (Q \times Q)^{\deg(v_0)-1}\}.$$
$$S_{v_f} = IF(A_{v_f}) \cap \{w.(q, q_f) \mid q \in Q, w \in (Q \times Q)^{\deg(v_f)-1}\}.$$

Algorithm 6.2 Local Bounded Context Switching

Input: An SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$, a graph $G = (V, E)$ with set of vertices $V = [1..t]$, initial vertex v_0, and final vertex v_f, and a contraction process $\pi = G_1, \ldots, G_t$ of G of degree s.
Output: *True*, if $L(S) \cap Sched(\Sigma, t, G) \neq \emptyset$. *False* otherwise.

1: assign to each $v \in V \setminus \{v_0, v_f\}$ the set S_v
2: assign to v_0 and v_f the sets S_{v_0} and S_{v_f}
3: **for** $\ell \in [1..t-1]$ **do**
4: let $G_{\ell+1} = G_\ell[v_1, v_2 \mapsto v']$ and $G_\ell = (V_\ell, E_\ell)$
5: set $i = E_\ell(v_1, v_2)$ and $j = E_\ell(v_2, v_1)$
6: assign to v' the set $S_{v'} = S_{v_1} \odot_{(i,j)} S_{v_2}$
7: **end for**
8: let $G_t = (\{u\}, \emptyset)$
9: return $((q_0, q_f) \in S_u)$

After the initial assignment, the algorithm iterates along π. Assume, we are in iteration $\ell \in [1..t-1]$. Let $G_{\ell+1} = G_\ell[v_1, v_2 \mapsto v']$ be the ℓ-th contraction in process π. Moreover, let $G_\ell = (V_\ell, E_\ell)$. We assign to the vertex v' the set

$$S_{v'} = S_{v_1} \odot_{(i,j)} S_{v_2},$$

where $i = E_\ell(v_1, v_2)$ is the number of edges from vertex v_1 to v_2 in G_ℓ and $j = E_\ell(v_2, v_1)$ is the number of edges from v_2 to v_1. Since the product contracts exactly i out-contractions and j in-contractions, we obtain $S_{v'} \subseteq (Q \times Q)^{\deg(v')}$, where $\deg(v')$ denotes the degree of v' in $G_{\ell+1}$. Indeed, note that

$$\deg(v') = \deg(v_1) + \deg(v_2) - (i + j).$$

The iteration terminates after $t-1$ steps. Let $V(G_t) = \{u\}$ be the last graph occurring in π. In the last iteration, we computed the set S_u. It contains all interface sequences obtained from contracting the interface languages along π. The algorithm reports that (S, G, π) is a yes-instance if and only if the sequence consisting of the single pair (q_0, q_f) is in S_v. The correctness is stated in the following lemma. A proof can be found in Appendix B.1.7.

Lemma 6.39. *We have $L(S) \cap Sched(\Sigma, t, G) \neq \emptyset$ if and only if $(q_0, q_f) \in S_u$.*

It is left to determine the complexity of Algorithm 6.2. In total, it takes $O(t)$ steps to assign all the sets S_v. The crux is the computation of the product $S_{v'} = S_{v_1} \odot_{(i,j)} S_{v_2}$. To estimate its complexity, recall that $S_{v_i} \subseteq (Q \times Q)^{\deg(v_i)}$. We have $\deg(v_i) \leq s$ since π is a contraction process of degree s. Hence for $S_{v'}$, we need to compute the product $\odot_{(i,j)}$ of at most

$$m^{2s} \cdot m^{2s} = m^{4s}$$

many pairs of interface sequences. The time that is needed to compute the directed contraction product of such a pair is given in the following lemma.

Lemma 6.40. *Let $k, \ell \leq s$. For interface sequences $\sigma \in (Q \times Q)^k$ and $\tau \in (Q \times Q)^\ell$, the directed contraction product $\sigma \odot_{(i,j)} \tau$ can be computed in time $O(2^{4s} \cdot s)$.*

Proof. For computing the directed contraction product, we need to form all interface sequences $\rho \in \sigma \text{Ш} \tau$, compute $con_{(i,j)}(\rho)$, and then unify in the end.

A shuffle of σ and τ corresponds to a binary string of length $k + \ell$ with exactly ℓ positions set to 1. Intuitively, the positions with a 1 are those that belong to τ, the others belong to σ. Hence, the number of shuffles is

$$\binom{k+\ell}{\ell} \leq 2^{k+\ell} \leq 2^{2s}.$$

Forming all shuffles can be done in $O(2^{2s} \cdot s)$ by iterating over all binary strings of length $k + \ell$ and checking whether it contains ℓ positions with a 1.

Given $\rho \in \sigma \text{Ш} \tau$, we can compute all sequences in $con_{(i,j)}(\rho)$ by choosing exactly i out-contractions and j in-contractions that we contract. This amounts to choosing $i + j$ positions out of $k + \ell = |\rho|$. Hence, $|con_{(i,j)}(\rho)|$ is bounded by

$$\binom{k+\ell}{i+j} \leq 2^{k+\ell} \leq 2^{2s}.$$

Thus, forming $con_{(i,j)}(\rho)$ takes time $O(2^{2s} \cdot s)$. Since we need to form the set for each $\rho \in \sigma \text{Ш} \tau$, the directed contraction product can be computed in

$$O(2^{2s} \cdot s + 2^{2s} \cdot 2^{2s} \cdot s) = O(2^{4s} \cdot s)$$

time. Note that the first summand comes from computing all the possible shuffles. The estimation completes the proof. □

In total, Algorithm 6.2 needs time $O(m^{4s} \cdot 2^{4s} \cdot s \cdot t^2)$ which constitutes the complexity estimation of Theorem 6.35. Note that one factor t is due to the number of iterations. The other one is due to the application of the algorithm in the case where we need to choose an initial vertex in the scheduling graph.

6.3.3 Round Robin

We consider the complexity of the *Round Robin* under-approximation. This local variant of BCS assumes that the threads take turns in a cyclic way and each thread has a budget of exactly $cs \in \mathbb{N}$ many contexts. To formalize the problem, let $t \in \mathbb{N}$ be the number of threads. We define the set of all words induced by threads taking turns in a cyclic fashion:

$$Cyclic(\Sigma, t, cs) = \{w \in (\Sigma \times [1..t])^* \mid \pi_{[1..t]}(w) = (1.2 \dots t)^{cs}\}.$$

Here, $\pi_{[1..t]}(w)$ projects away the Σ-component from the letters of w.

The problem Round Rob takes an SMCP S over Σ consisting of t threads and a bound on the number of contexts cs. It asks whether there is a word in the language of S the contexts of which switch accordingly to $Cyclic(\Sigma, t, cs)$.

Round Rob
Input: An SMCP $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ and a bound $cs \in \mathbb{N}$.
Question: Is $L(S) \cap Cyclic(\Sigma, t, cs) \neq \emptyset$?

We present the results of our fine-grained analysis of Round Rob. This includes an upper bound and a matching lower bound for the problem. The latter also applies to LBCS and shows that the algorithm presented in Section 6.3.2 is optimal in the fine-grained sense. Note that the given integer cs in Round Rob is a bound on the number of contexts, not on the number of context switches. However, the upcoming complexity results also hold if a bound on the latter is given since both numbers only differ by 1.

Upper Bound To obtain an upper bound for Round Rob, we want to apply the algorithm for LBCS. To this end, we need to turn an instance of Round Rob into an equivalent instance of the latter problem and then run the algorithm. By definition, Round Rob restricts the scheduling to a cycle where each thread contributes cs many times. We capture it by a suitable graph and also provide a corresponding contraction process.

Definition 6.41. Let $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ be an SMCP and $cs \in \mathbb{N}$. Define the graph $G(S, cs) = (V, E)$ with $V = [1..t]$ and edges $E(i, i+1) = cs$ for $i \in [1..t-1]$ and $E(t, 1) = cs - 1$. This means that thread i switches to $i+1$ exactly cs many times and t switches to 1 exactly $cs - 1$ many times.

The constructed graph $G(S, cs)$ works as expected. The following lemma states the correctness. Its proof immediately follows from the definition.

Lemma 6.42. *Let $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ be an SMCP and $cs \in \mathbb{N}$. We have*

$$L(S) \cap Cyclic(\Sigma, t, cs) \neq \emptyset \text{ if and only if } L(S) \cap Sched(\Sigma, t, G(S, cs)) \neq \emptyset.$$

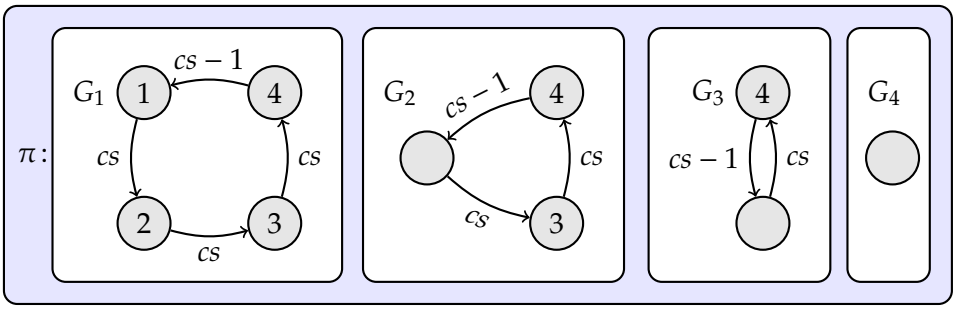


Figure 6.5: The contraction process π for the case of four threads. Note that $G_1 = G(S, cs)$. We contract in the order given by the threads. Except from the last vertex in G_4, each vertex v in the process has degree exactly cs.

To obtain an instance of LBCS, we still need to construct a contraction process π of the graph $G(S, cs)$. We can easily describe one. First, we contract the vertices 1 and 2, then the result with vertex 3 and we repeat up to vertex t. An illustration is given in Figure 6.5. Note that $\deg(\pi) = cs$.

We are now ready to describe the algorithm for Round Rob in more detail. For a given instance (S, cs), first construct the equivalent instance $(S, G(S, cs), \pi)$ of LBCS. This takes time at most $O(cs \cdot t)$. Then, apply Algorithm 6.2 to the constructed instance. Correctness follows from Lemma 6.42.

For the complexity estimation it is important to note that an application of Algorithm 6.2 in this case takes only $O(m^{4cs} \cdot cs \cdot t)$ time. The algorithm performs faster than for general LBCS since the given graph $G(S, cs)$ has a quite simple form. It allows for a computation of the directed contraction product in linear time $O(cs)$ and avoids the factor $O^*(2^{4cs})$ which we would get according to Lemma 6.40. We provide details in Appendix B.1.8. The complexity estimation is summarized below.

Corollary 6.43. *Round Rob can be solved in time $O(m^{4cs} \cdot cs \cdot t)$.*

Lower Bound We focus on the lower bound for Round Rob. It matches the presented upper bound and proves the optimality of our algorithm. The key is a reduction from $k \times k$-Clique. In fact, we show that a $k \times k$-graph G can be translated into an instance (S, cs) of Round Rob such that $m = O(k^3)$ and $cs = O(k)$. According to Lemma 5.14, an algorithm for Round Rob running in time $2^{o(cs \cdot \log(m))}$ would then contradict the ETH.

Theorem 6.44. *Unless ETH fails, Round Rob cannot be solved in time $2^{o(cs \cdot \log(m))}$.*

Proof. We elaborate on the idea of the reduction, a formal proof can be found in Appendix B.1.9. Let G be a $k \times k$-graph with vertices $V = [1..k] \times [1..k]$.

We need to construct an SMCP S. The idea is to proceed in two phases. In the first phase, the memory suggests a clique candidate and the threads store

it. The second phase then verifies that it is indeed a clique by enumerating all the needed edges. For the communication among threads and memory, the underlying alphabet Σ contains three different kinds of symbols: the vertices $(i, j) \in V$, symbols for a trivial context $(i, \#)$ with $i \in [1..k]$, and edge symbols $e((i', j'), (i, j))$ for each edge $\{(i, j), (i', j')\}$ in the graph G.

We construct a thread A_i for each row $i \in [1..k]$ capable of storing a particular vertex (i, j) of the row. After storing (i, j), the thread can repeat the stored vertex, perform a trivial context $(i, \#)$, or read an edge symbol $e((i', j'), (i, j))$ where (i', j') is an adjacent vertex in a smaller row $i' < i$.

The memory M organizes the computation along the aforementioned phases. In the first phase, it suggests a clique candidate by reading a word of the form $(1, j_1) \dots (k, j_k)$. From each row, M chooses a vertex and synchronizes with the corresponding thread A_i which stores the choice. Note that the communication happens along the cycle $A_1.A_2 \dots A_k$.

It remains to verify that the chosen vertices form a clique. To this end, M enters the second phase and performs $k - 1$ verification rounds. We describe the i-th round. In the beginning, M synchronizes with the threads A_ℓ, where $\ell < i$, on the trivial context $(\ell, \#)$. These threads do not contribute in this round and the synchronization is only to ensure a cyclic scheduling. Then, M synchronizes with A_i on the chosen vertex (i, j_i) and temporarily stores the vertex in its states until the end of the round. Note that the synchronization is possible since A_i stored the vertex. Subsequently, memory and remaining threads check whether there are edges from (i, j_i) to all chosen vertices (ℓ, j_ℓ) in larger rows $\ell > i$. To this end, M synchronizes with the A_ℓ on the symbols $\{(i, j_i), (\ell, j_\ell)\}$. The synchronization can only proceed if the edges exist.

After completing the second phase, all needed edges are enumerated and the chosen candidate $(1, j_1), \dots, (k, j_k)$ is indeed a clique. Communication in this phase again follows the above cycle. In total, each thread requires exactly k contexts, we get $cs = k$. The size of the memory is $O(k^3)$ since we need to store a chosen vertex $k - 1$ many times in the second phase. □

The reduction described in Theorem 6.44 can also be seen as a reduction from $k \times k$-Clique to LBCS. To obtain an instance of the latter, we only need to construct the corresponding scheduling graph — the cycle of weight k, as in Definition 6.41, and the corresponding contraction process of degree $s = k$. Consequently, we obtain the more general lower bound for LBCS which proves optimality of our algorithm for the problem.

Corollary 6.45. *Unless ETH fails, LBCS cannot be solved in time* $2^{o(s \cdot \log(m))}$.

6.4 The Complexity of Bounded Stage

We consider the complexity of the *bounded stage reachability* problem (BSR), a generalization of BCS that does not bound the number of context switches

but the number of *stages*. A stage is a collection of contexts where one thread has the write permission on the shared memory, the others can only read. A new stage begins once the write permission changes to another thread. For making write and read accesses visible, we first need to introduce the model of *read-write SMCPs* in Section 6.4.1. Then we can define the problem BSR on the new model. In Section 6.4.2, we focus on upper and lower bound for a tractable parameterization of BSR. Our main result is the kernel lower bound presented in Section 6.4.3. It demonstrates the computational power of the under-approximation. Intractability results are given in Section 6.4.4.

6.4.1 Read-Write SMCPs and BSR

To properly define the bounded stage under-approximation, we need to make visible the write and read accesses of the threads on the shared memory. To this end, we alter the definition of shared-memory concurrent programs. Like before, a program consists of finitely many threads that access a shared memory. However in the new model, the memory is not an automaton but a cell, holding a single value at a time. Threads can read this value or write a new value via corresponding *read* and *write* operations. We formalize below.

Shared Memory and Operations The shared memory is a cell over an underlying finite data domain D. At the beginning of a computation, it holds a particular value $a_0 \in D$ which we refer to as the *initial value*. For reading and manipulating the content of the memory, we employ so-called *operations*.

Definition 6.46. Let D be a finite data domain. The set of operations is

$$OP(D) = \{!a, ?a \mid a \in D\}.$$

The set contains two different kinds of operations. The *write operations* $WR(D) = \{!a \mid a \in D\}$ are used to change the content of the memory, they represent write accesses. The *read operations* $RD(D) = \{?a \mid a \in D\}$ access the memory without changing its value. These model read accesses.

Threads As for an SMCP, threads are labeled control flow graphs of real threads in a program. But this time we are more specific on a thread's interaction with the shared memory. To this end, we assume that the control flow graphs are labeled by read and write operations.

Definition 6.47. Let D be a finite data domain. A *thread* is a (nondeterministic) finite automaton $A_{id} = (P_{id}, OP(D) \times \{id\}, \delta_{id}, p_0, p_f)$.

The semantics of a thread is as expected. A computation of A_{id} corresponds to a word in $(OP(D) \times \{id\})^*$ following the transition relation δ_{id}.

Read-Write SMCPs We are ready to define the new model of so-called *read-write SMCPs*. These consists of a memory cell over an underlying data domain and several threads that interact with the cell.

Definition 6.48. Let D be a finite data domain, $a_0 \in D$ an initial value, and $A_1, \ldots, A_t$ threads over D. A *read-write shared-memory concurrent program* (RW-SMCP) is defined to be a tuple $S = (D, a_0, (A_i)_{i\in[1..t]})$.

We define the semantics of an RW-SMCP S in terms of labeled transitions between *configurations*. These are snapshots of a computation at a certain point in time. Note that we could also define the semantics of S in terms of language theoretic operators like we did for SMCPs. However, the definition via configurations emphasizes more on the interaction with the memory cell.

Definition 6.49. Let $S = (D, a_0, (A_i)_{i\in[1..t]})$ be an RW-SMCP. A *configuration of* S is a tuple (pc, a) consisting of a *program counter* $pc \in P_1 \times \cdots \times P_t$ and some value $a \in D$. The program counter holds the current state $pc(i) \in P_i$ of each thread A_i. The value a shows the current value stored in the memory cell. We define the *set of configurations of* S as the product

$$Conf(S) = P_1 \times \cdots \times P_t \times D.$$

The *initial configuration* $c_0 \in Conf(S)$ is given by $c_0 = (pc_0, a_0)$, where $pc_0(i)$ is the initial state of thread A_i for all $i \in [1..t]$ and a_0 is the initial value. A configuration $c = (pc, a)$ is *final* or *accepting* if $pc(i)$ is the final state of A_i for each $i \in [1..t]$ and $a \in D$. We denote by $Fin(S)$ the set of final configurations.

Computations of an RW-SMCP may change the memory value and the states of the threads. If, for instance, a thread writs a new value into the memory cell, this should be reflected by changing the current configuration accordingly, by updating the state of the acting thread and the memory value. This leads to the definition of a transition relation among configurations.

Before we define the relation, we introduce some new notation for updating entries of a program counter. Let pc, pc' be program counters of an RW-SMCP. We write $pc' = pc[i = p]$ if pc' coincides with pc except for the i-th component, where $pc'(i) = p$ updates the state of the i-th thread to be p.

Definition 6.50. Let $S = (D, a_0, (A_i)_{i\in[1..t]})$ be an RW-SMCP. The *transition relation of* S, denoted by $\rightarrow_S$, is a relation on configurations:

$$\rightarrow_S \subseteq Conf(S) \times (OP(D) \cup \varepsilon) \times Conf(S).$$

The relation contains read, write, and ε-transitions, each of which is induced by the threads. We begin with the definition of the former. If thread A_i has a read transition, this implies transitions of $\rightarrow_S$ in the following sense. For each $pc, pc_1[i = p_i]$, and $pc_2[i = p_i']$, we have:

$$p_i \xrightarrow{(?a,i)}_{\delta_i} p_i' \text{ implies } (pc_1, a) \xrightarrow{?a}_S (pc_2, a).$$

Note that the memory is required to hold the desired value a. Otherwise, the transition cannot be performed since the value is not available to be read.

Similarly, if A_i has a write transition, this implies corresponding write transitions of $\rightarrow_S$. These change the current memory value:

$$p_i \xrightarrow{(!b,i)}_{\delta_i} p_i' \text{ implies } (pc_1, a) \xrightarrow{!b}_S (pc_2, b).$$

Finally, an interior ε-transition of thread A_i induces transitions of $\rightarrow_S$ that change the current state of A_i but do not affect the memory value:

$$p_i \xrightarrow{(\varepsilon,i)}_{\delta_i} p_i' \text{ implies } (pc_1, a) \xrightarrow{\varepsilon}_S (pc_2, a).$$

We can generalize the transition relation $\rightarrow_S$ to words of operations. Let $w \in OP(D)^*$ be such a word. Then we write $c \xrightarrow{w}_S c'$ for a corresponding sequence of transitions and call it a *computation of* S. Where appropriate, we omit the index and only write $c \xrightarrow{w} c'$ for a computation.

With the notion of a computation we can now summarize the semantics of an RW-SMCP in its language. It consists of all words seen during a computation leading from the initial to an accepting configuration.

Definition 6.51. Let $S = (D, a_0, (A_i)_{i\in[1..t]})$ be an RW-SMCP and c_0 the initial configuration. The *language of* S is defined by

$$L(S) = \{w \in OP(D)^* \mid c_0 \xrightarrow{w}_S c_f, \text{ where } c_f \in Fin(S)\}.$$

Note that $L(S) \subseteq OP(D)^*$. Thread identifiers are projected away by the transition relation $\rightarrow_S$. Before we define the bounded stage reachability problem of interest, we illustrate the semantics with an example.

Example 6.52. Consider the RW-SMCP $S = (D, a_0, A_1, A_2)$ over the domain $D = \{a_0, a, b\}$ with threads as shown in Figure 6.6. We compute the language $L(S)$. To this end, we consider all computations from the initial to an accepting configuration. The first computation is the following:

$$\sigma_1 = (p_1, q_1, a_0) \xrightarrow{!a} (p_1, q_3, a) \xrightarrow{?a} (p_3, q_3, a).$$

The last configuration visited by σ_1 is accepting. Hence, the word $w_1 = !a.?a$ is in the language $L(S)$. The second possible computation of S is:

$$\sigma_2 = (p_1, q_1, a_0) \xrightarrow{!b} (p_1, q_2, b) \xrightarrow{?b} (p_2, q_2, b) \xrightarrow{!c} (p_2, q_3, c) \xrightarrow{?c} (p_3, q_3, c).$$

Hence, $w_2 = !b.?b.!c.?c$ is the second word of $L(S)$. Note that $L(S) = \{w_1, w_2\}$, there are no further computations of the RW-SMCP.

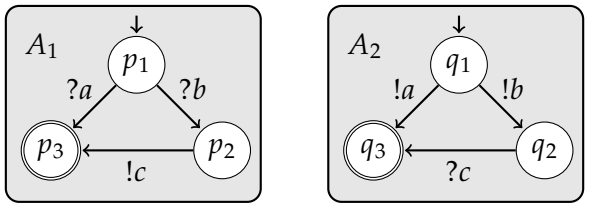


Figure 6.6: The RW-SMCP of Example 6.52. The data domain is given by $D = \{a_0, a, b\}$. Note that we omitted the thread identifiers of A_1 and A_2.

Bounded Stage Reachability BSR limits the verification of RW-SMCPs to bounded-stage computations. A *stage* is a generalization of a context. Instead of only allowing for one thread to act, a stage allows for concurrent behavior as long as only one thread is writing and the others are restricted to reading. Once the write permission is passed to another thread, a further stage begins.

Definition 6.53. Let D be a finite domain, $t \in \mathbb{N}$, and $w \in (OP(D) \times [1..t])^*$. The word w admits a unique decomposition into maximal infixes

$$w = w_1 \dots w_k$$

together with a function $\varphi : [1..k] \to [1..t]$ that satisfies the following two conditions. For each $i \in [1..k-1]$, we have $\varphi(i) \neq \varphi(i+1)$. Moreover, for $i \in [1..k]$, the writes in w_i are only from the single thread $\varphi(i)$:

$$\pi_W(w_i) \in (WR(D) \times \{\varphi(i)\})^+.$$

Note that the projection $\pi_W : OP(D) \times [1..t] \to WR(D) \times [1..t]$ is employed to map away the reads of a given word.

Example 6.52 shows two words with a different numbers of stages. In fact, the word w_1 has one stage since only the thread A_2 is writing. The word w_2 has two stages — first A_2 is writing, then A_1.

Now we can restrict to computations with a bounded number of stages. To this end, we define the language of words over $OP(D)$ with at most k stages that can be associated with t threads. The definition is as follows:

$$Stage(D, t, k) = \left\{ w = w_1 \dots w_\ell \;\middle|\; \begin{array}{c} \ell \leq k \text{ and } \exists\, \varphi : [1..\ell] \to [1..t] : \\ \pi_W(w_i) \in (WR(D) \times \{\varphi(i)\})^+, \\ \varphi(i) \neq \varphi(i+1). \end{array} \right\}$$

The *bounded stage reachability* problem takes an RW-SMCP S and a bound $k \in \mathbb{N}$ as input. It asks whether the language $L(S)$ contains a word with at most k stages. Although the problem can be seen as a generalization of BCS (on RW-SMCPs), it maintains the same complexity: it is still NP-complete [20].

BSR

Input: An RW-SMCP $S = (D, a_0, (A_i)_{i\in[1..t]})$ and a bound $k \in \mathbb{N}$.

Question: Is $L(S) \cap \mathit{Stage}(D, t, k) \neq \emptyset$?

Classical complexity theory does not distinguish between BCS and BSR. Only a fine-grained complexity analysis shows that the latter problem is actually much harder. For this analysis, we consider various parameters. These are the size d of the data domain D, the number of threads t, and the bound k on the number of stages. Moreover, we consider the parameter p, the maximal number of states of a thread: $p = \max_{i\in[1..t]} |A_i|$.

6.4.2 Upper and Lower Bound

We first focus on a parameterization of the problem BSR by p and t. Although the parameterization is quite strict, it is in fact the only way to obtain a fixed-parameter tractable algorithm. Other parameterizations are indeed intractable, as we will see in Section 6.4.4. Especially the number of stages k does not have much impact on the fine-grained complexity of BSR. We present an upper and a matching lower bound for BSR(p, t). The former relies on a suitable product construction, the latter is based on a reduction from $k \times k$-Clique and shows that we cannot avoid building the product.

Upper Bound We show that BSR(p, t) is fixed-parameter tractable. To this end, we reduce the problem to a reachability problem an a product automaton. The automaton stores the current configuration, the thread that is currently writing, and the number of stages. This allows for mimicking stage-bounded computations of the given RW-SMCP by runs of the automaton.

Theorem 6.54. *BSR can be solved in time* $O(p^{2t} \cdot d^2 \cdot t^2 \cdot k^2)$.

Proof. Let (S, k) be an instance of BSR with an RW-SMCP $S = (D, a_0, (A_i)_{i\in[1..t]})$. The idea behind building the product automaton $P(S)$ is simple. We can afford a state for each configuration of S and moreover, store the currently writing thread and the number of stages. To this end, $P(S)$ works on the states

$$\mathit{Conf}(S) \times [0..t] \times [0..k].$$

Now, any transition of S can be mimicked by $P(S)$. For instance a read transition $c \xrightarrow{?a}_S c'$ on configurations c, c' is simulated by the transitions $(c, i, \ell) \xrightarrow{?a} (c', i, \ell)$, where $i \in [1..t]$ and $\ell \in [1..k]$. Note that neither the current writer i nor the number of stages ℓ is changed.

For a write transition, this is different. Assume S performs $c \xrightarrow{!a}_S c'$ and the write transition is due to thread i'. Then, $P(S)$ simulates it by

$(c, i, \ell) \xrightarrow{!a} (c', i', \ell')$. Here, $\ell' = \ell + 1$ if $i' \neq i$. This means, if the currently writing thread changes from i to i', we have to increase the number of stages. If the writing thread remains the same, $i' = i$, we have $\ell' = \ell$.

The initial state of $P(S)$ is the tuple $(c_0, 0, 0)$ where c_0 is the initial configuration. The final states of $P(S)$ are those that contain a final configuration. We have that S contains a word with at most k stages if and only if $L(P(S)) \neq \emptyset$. Hence, it remains to check the latter. Since $P(S)$ has $O^*(p^t)$ states, checking non-emptiness of its language takes time $O^*(p^{2t})$. For a formal construction of $P(S)$ and a detailed complexity estimation, we refer to Appendix B.1.10. □

Lower Bound We show that the algorithm from Theorem 6.54 is optimal. To this end, we give a reduction from $k \times k$-Clique to BSR keeping the parameters small. In fact, given a $k \times k$–graph G, we construct an RW-SMCP S with $p = O(k^2)$ and $t = O(k)$. Moreover, G has the desired clique if and only if $L(S)$ contains a word with only one stage. By Lemma 5.14, we obtain that an $2^{o(t \cdot \log(p))}$-time algorithm for BSR would contradict the ETH.

Theorem 6.55. *Unless ETH fails, BSR cannot be solved in time* $2^{o(t \cdot \log(p))}$.

Proof. We elaborate on the idea of the construction. A formal proof is given in Appendix B.1.11. Let G be an instance of $k \times k$-Clique with set of vertices $V = [1..k] \times [1..k]$. We first define $D = V \cup \{a_0\}$ to be the data domain. In fact, we want the threads to communicate on the vertices of G.

For each row $i \in [1..k]$, we introduce thread P_i responsible for storing a particular vertex of the row. We also add the thread P that is used to steer the communication between the P_i. Our RW-SMCP S is then given by the tuple $S = (D, a_0, (P_i)_{i \in [1..k]}, P)$. Note that the number of threads is $t = k + 1$.

Intuitively, S proceeds in two phases. In the first phase, each P_i nondeterministically chooses a vertex from the i-th row and stores it in its states. To this end, P_i has a transition reading the initial value a_0 and branching into a state representing the chosen vertex. After this phase, the P_i are storing a clique candidate $(1, j_1), \ldots, (k, j_k) \in V$. Consequently, it is left to verify that there are edges among each two of these vertices.

In the second phase, we ensure that the needed edges exist. Thread P starts to write a random vertex $(1, j_1')$ of the first row to the memory cell. The first thread P_1 reads $(1, j_1')$ from the memory and verifies that the read vertex is actually the stored 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 can only proceed if 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 deadlocks. Otherwise, if no deadlock occurs, we have verified that $(1, j_1)$ is adjacent to each vertex in the clique candidate. After this step, P writes a vertex $(2, j_2')$ from the second row to the memory, and the verification steps repeat. In the end, after k repetitions of the procedure, we can ensure that the guessed candidate has all the needed edges and is indeed a clique.

Note that P has only $O(k)$ states since it only needs to store the rows. The P_i have $O(k^2)$ states each. These threads need to remember the chosen column $j \in [1..k]$ and for each of these, count that they have seen one adjacent vertex from each row. Moreover, P is the only thread that is writing. Hence, G has the desired clique if and only if $L(S)$ contains a 1-stage word. □

6.4.3 Absence of a Polynomial Kernel

We present a kernel lower bound for the problem BSR(p, t). Along with the lower bound on the running time proven in Section 6.4.2, we get a clear picture regarding the parameterized complexity of BSR(p, t). Although the parameterization is already rather restricted, we can neither hope for a fast algorithm nor for a kernel of polynomial size. Altogether, the results demonstrate the computational power of BSR as an under-approximation.

To prove the kernel lower bound, the idea is to cross-compose the problem 3-SAT into BSR(p, t). However, setting up the cross-composition is not straight forward as we face a challenging technical problem. Assume we are given I instances of 3-SAT, I formulas for which we need to check whether one of them is satisfiable. Then, the two significant parameters p and t are not allowed to depend polynomially on I. Otherwise we would contradict the requirements on a cross-composition, see Definition 5.42. In fact, we need to ensure that p and t depend at most logarithmically on I.

This restricted dependence immediately excludes any idea for the cross-composition which involves a thread choosing a particular 3-SAT-instance by branching into I different states. It would cause a polynomial dependence of p on I. Furthermore, it is not possible to construct a thread for each instance as this would cause such a dependence of t on I. To circumvent the problem, we need to find a way of choosing an instance without having I many states or threads. As we will see, the solution is to assign a binary number of length $\log(I)$ to each given instance and being able to compare these numbers bit by bit. For each bit we will need one small thread, resulting in parameters p and t depending only logarithmically on I. The result is as follows.

Theorem 6.56. *BSR(p, t) does not admit a poly. kernel, unless* NP ⊆ coNP/poly.

Proof. We construct the required cross-composition. Let $\varphi_1, \ldots, \varphi_I$ be the given 3-SAT-instances, where each two are equivalent under $\mathcal{R}$, the polynomial equivalence relation of Example 5.41. Recall that two formulas are equivalent under $\mathcal{R}$ if they have the same number of clauses and variables. Hence we can assume each formula φ_ℓ with $\ell \in [1..I]$ to have m clauses, namely $\varphi_\ell = C_1^\ell \wedge \ldots \wedge C_m^\ell$, and the n variables $\{x_1, \ldots, x_n\}$.

In the RW-SMCP that we construct, 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 C_j^ℓ of φ_ℓ can be satisfied by variable x_i with valuation v.

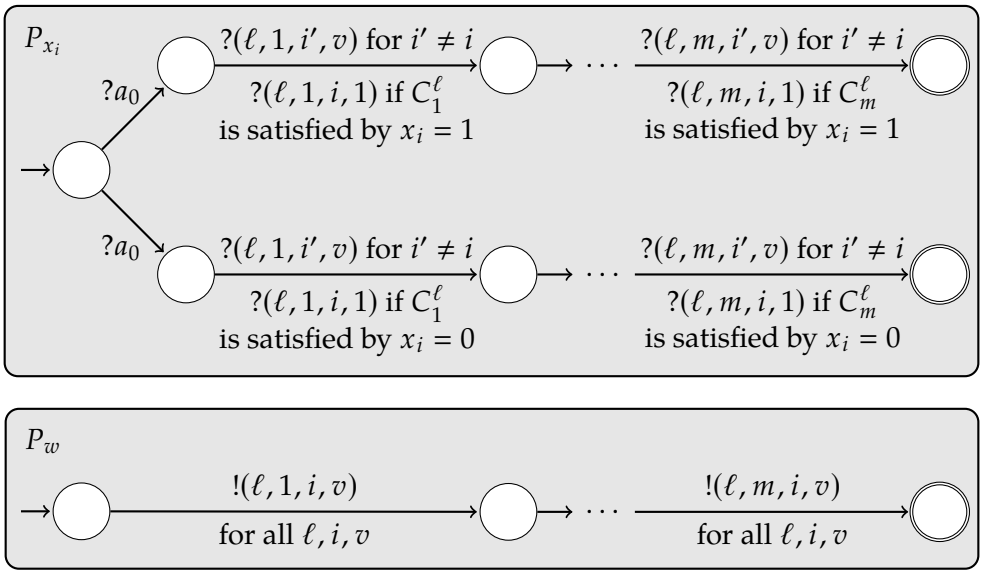


Figure 6.7: The above thread P_{x_i} chooses and stores a valuation of x_i. The upper branch corresponds to $x_i = 1$, the lower branch to $x_i = 0$. The thread P_{x_i} can only read those tuples that do not affect x_i or those that encode a clause which can be satisfied by the sored valuation. Thread P_w suggests m clauses that need to be satisfied by the valuation stored in the P_{x_i}.

Hence, the data domain of the program is given by

$$D = ([1..I] \times [1..m] \times [1..n] \times \{0, 1\}) \cup \{a_0\}.$$

For choosing and storing an valuation of the variables, we introduce a thread P_{x_i} for each variable x_i. In the beginning, each P_{x_i} non-deterministically chooses an valuation for x_i and stores it in its state space.

We further introduce a writing thread 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 the threads P_{x_i} then start to read the tuple. If P_{x_i} with $i \neq i_j$ reads it, the thread will simply 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 stored value for x_i that P_{x_i} guessed initially and moreover, if x_i with valuation v_j satisfies the clause $C_j^{\ell_j}$. Subsequently, P_w proceeds to the next tuple. The construction of the P_{x_i} and P_w is illustrated in Figure 6.7.

Now suppose P_w has written m tuples in exactly m steps and each P_{x_i} has performed exactly $m+1$ reads. Then this proves the satisfiability of m clauses by the chosen valuation. But these clauses can be part of different instances.

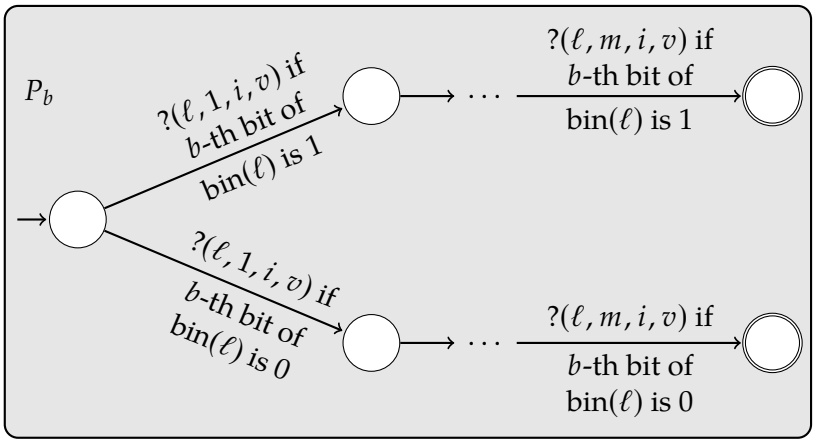


Figure 6.8: The bit checkers P_b for $b \in [1..\log(I)]$. After reading a tuple $(\ell, 1, i, v)$, these threads store the b-th bit of bin(ℓ). They can only reach a final state if the b-th bit does not change throughout all suggested tuples.

It is not ensured that the clauses were chosen from one formula φ_ℓ. The major technical 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 bin(ℓ), 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 bin(ℓ_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 bits of bin(ℓ_1) and bin(ℓ_2) coincide and only proceeds if they are equal. Otherwise it will deadlock. This will be repeated any time P_w writes a new tuple to the memory. We illustrate the construction of the bit checkers in Figure 6.8.

Assume the computation does not deadlock in any of the P_b. Then we can ensure that the b-th bit of bin(ℓ_j) with $j \in [1..m]$ never changed during the computation. This means that bin$(\ell_1) = \cdots =$ bin(ℓ_m). Hence, P_w has chosen clauses of a single instance φ_ℓ. Moreover, the valuation stored in the P_{x_i} satisfies the formula. This implies the correctness of the construction: there is an $\ell \in [1..I]$such that φ_ℓ is satisfiable if and only if the constructed RW-SMCP contains a word with one stage. Note that we are only interested in 1-stage words since P_w is the only thread that writes to the memory.

The parameters are $p \in O(m)$ and $t \in O(n + \log(I))$. Hence, they depend at most logarithmically on I and therefore the construction constitutes a proper cross-composition. We provide a proof of correctness in Appendix B.1.12. □

6.4.4 Intractability

As a last result on the fine-grained complexity of BSR, we show that parameterizations involving the number of stages k and the size of the data domain d are intractable. It justifies our choice of parameters in Section 6.4.2 and shows once more that fixed-parameter tractability can only be obtained by a strict parameterization of BSR. Our goal is to show the following theorem.

Theorem 6.57. *BSR*(k, d) *is not in* XP, *unless* P = NP.

The theorem has a surprising aspect. The number of stages k is a powerful parameter. Introducing the bound in a simultaneous reachability problem like BSR lets the complexity drop from PSPACE to NP. However, its impact is not enough to guarantee an FPT-algorithm. Even worse, we do not even get membership in XP. This means there is no algorithm for BSR depending exponentially only on the parameters k and d.

In order to prove the theorem, we show that BSR is NP-hard even in the case when both, k and d, are constants. Note that this indeed implies Theorem 6.57. If BSR(k, d) would be a member of XP, then there would be an algorithm for the problem running in time $f(k, d) \cdot n^{g(k,d)}$, for some computable functions f and g. The algorithm could then also be applied to ConstBSR, the variant of BSR, where k and d are constant. In this case, it would solve ConstBSR in polynomial time which contradicts the NP-hardness of the problem. Hence, BSR(k, d) cannot be in XP. It is left to prove the following lemma.

Lemma 6.58. *BSR is* NP*-hard even if k and d are constants.*

Proof. We give a reduction from 3-SAT to BSR keeping both parameters constant. Let φ be a 3-SAT-instance with m clauses and variables $x_1, \ldots, x_n$. We construct an RW-SMCP $S = (D, a_0, (P_{x_i})_{i \in [1..n]}, P_v)$ with $d = 4$ symbols in the data domain and $L(S)$ only containing 1-stage words.

First, we clarify the communication among the threads of S. Like in other related reductions we would like to base it on the literals of φ. However, we cannot write and read literals directly, as this would cause a blow-up in parameter d. Instead, we need to use a suitable binary encoding to keep the data domain small. Let ℓ be a literal in φ. It consists of a variable x_i and a valuation $v \in \{0, 1\}$. We employ the *padded binary encoding of* i:

$$\mathrm{bin}_{\#}(i) \in (\{0, 1\}.\#)^{\log(n)+1}.$$

It is defined as the usual binary encoding of i but each bit is separated by the padding symbol #. The *encoding* of literal ℓ is then defined to be the string $\mathrm{enc}(\ell) = v.\#.\,\mathrm{bin}_{\#}(i)$. It encodes that variable x_i has valuation v.

The RW-SMCP S communicates by reading and writing messages of the form $\mathrm{enc}(\ell)$. To this end, we need the data domain $D = \{a_0, \#, 0, 1\}$. Note that the padding symbol # is required to prevent the threads from stuttering,

reading the same symbol more than once. In fact we will make sure that each thread in S has to read an exact number of symbols to reach a final state.

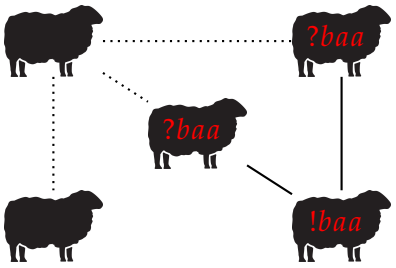
We have a thread P_{x_i} for each variable x_i. Initially, these choose a valuation for the variables and store it. To this end, it can branch on reading a_0 and choose whether it assigns 0 or 1 to x_i. Then another thread, called P_v, starts to iterate over the clauses of φ. For each clause C, it nondeterministically picks a literal ℓ in C and writes $\text{enc}(\ell)$ to the memory. Each P_{x_i} reads $\text{enc}(\ell)$. Note that reading and writing the encoding requires a sequence of transitions.

It is the task of P_{x_i} to check whether the chosen literal ℓ is conform with the stored valuation of x_i. If ℓ involves a variable x_j with $j \neq i$, thread P_{x_i} continues its computation by reading the whole string $\text{enc}(\ell)$. If ℓ involves x_i, then P_{x_i} can only continue its computation if the first bit in $\text{enc}(\ell)$ shows the stored valuation. Formally, there is only an outgoing path of transitions on $\text{enc}(x_i)$ if P_{x_i} stored 1 and on $\text{enc}(\neg x_i)$ if it stored 0.

Note that each time P_v picks a literal ℓ, all P_{x_i} read $\text{enc}(\ell)$, even if the literal involves a different variable. This means that each P_{x_i} reads exactly m encodings of literals, corresponding to a word of fixed length. This is important for correctness as the threads will only reach a final state if they did not miss a single symbol. Phrased differently, there is no loss in communication.

Now assume P_v iterated through all m clauses and none of the P_{x_i} got stuck. Then each of them read exactly m encodings without running into a deadlock. This means that the picked literals were all conform with the valuations chosen by the P_{x_i} and that a satisfying assignment for φ was found. Note that P_v is the only thread that is writing to the memory. Hence, each word in $L(S)$ consists of a single stage, proving that also parameter k is a constant. For a proof of correctness, we refer to Appendix B.1.13. □

7. Liveness Verification in Broadcast Networks



Broadcast networks are a model that was originally introduced for the verification of ad hoc networks and cache-coherence protocols. Such networks and protocols typically have to manage an arbitrarily large number of identical devices or processors. To model such applications, broadcast networks are *parameterized*: they consist of an arbitrary number of identical clients that is not known a priori[9]. The communication among the clients is based on sending broadcast messages. Proving the correctness of a broadcast network amounts to proving the correctness for each potential number of clients. While the complexity of most safety verification questions in this context has already been solved, liveness verification of broadcast networks is not fully understood. In particular, the complexity of the basic *broadcast network liveness verification problem* (BNL) has been left open. The problem asks whether there exists a computation in the network such that one client visits a final state infinitely often. BNL was shown to be P-hard [123] but only in EXPSPACE [124].

By a fine-grained complexity analysis of BNL, we found that the problem admits a polynomial-time algorithm. The algorithm relies on a characterization of live computations in terms of paths in a suitable abstraction. The latter might be of interest for other parameterized models as well. We also considered the complexity of the *fair liveness verification problem* (FBNL). It asks for a computation where all participating clients visit a final state infinitely often. We adjusted the algorithm of BNL to also solve FBNL in polynomial time. The results were originally obtained in our publication [95]. The upcoming text is an extension of the paper.

7.1 Verification of Broadcast Networks

Broadcast networks were introduced by Emerson and Namjoshi [147] for the verification of *parameterized systems*. Such a system consists of an arbitrary number of threads or devices that run an identical protocol or program. We understand those threads and devices as *anonymous clients* that are not

[9] Parameterized models/systems are not to be confused with parameterized complexity.

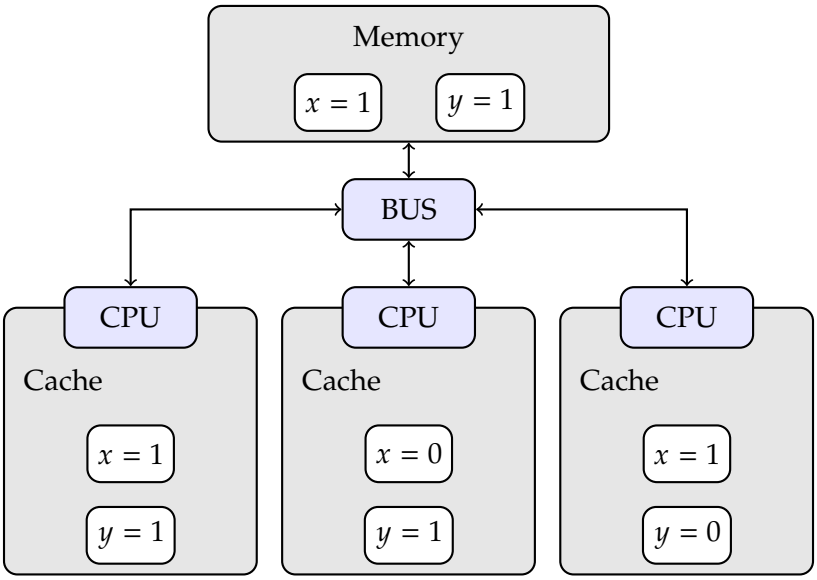


Figure 7.1: A multiprocessor system consisting of three CPUs with caches and a memory over the variables x and y. Communication is via a bus.

aware of their identity. Clients typically communicate via some mechanism like shared memory, rendezvous, or broadcast messages [17, 146, 147, 182]. Parameterized systems appear in various applications like distributed algorithms [237], ad hoc networks [2, 124, 224, 236, 298, 302], depth-bounded systems [276, 277, 327, 337], and cache-coherence protocols [9, 121, 147, 151]. Verifying parameterized systems is a challenging task. Correctness has to be proven not only for a single instance of the system but for every number of participating clients. This has led to substantial effort concerning the verification of parameterized systems, resulting in the development of a new research field known as *parameterized verification* [4, 55].

In parameterized verification, broadcast networks are a well-established model to represent parameterized systems. The clients of a broadcast network are identical finite-state automata reflecting the interaction of a single device, thread, or processor with its environment. A broadcast network consists of an arbitrary number of clients that communicate via broadcast messages.

To demonstrate the applicability of the model, we consider an example: a broadcast network for the cache-coherence protocol MSI [304]. Figure 7.1 illustrates the setting. It shows a schematic multiprocessor system which contains a shared memory and three CPUs, each having a dedicated cache. CPUs and memory communicate over a bus. Assume the memory maintains the two variables x and y. When a CPU writes to or reads from a variable, it does not immediately access the memory as such accesses are expensive and degrade performance in practice. Instead, the CPU prefers to access its

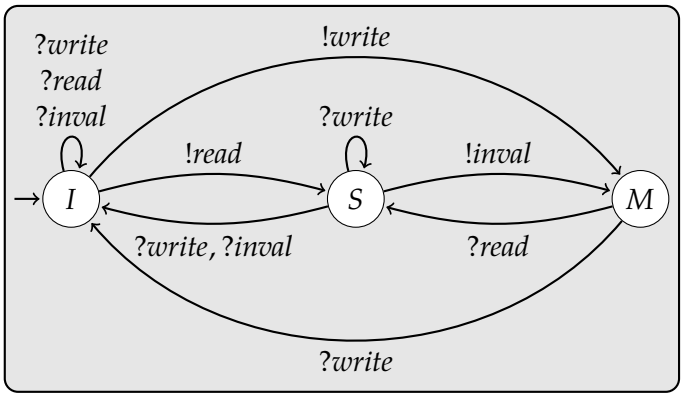


Figure 7.2: The communication scheme of the cache-coherence protocol MSI written as a finite automaton over the domain of messages $\{write, read, inval\}$.

own cache, an operation that is rather fast. To this end, the cache holds a local copy of x and y. Without regulation this may lead to different values of x and y in each of the caches and the memory. Hence, when a variable is assigned a new value by some CPU, other CPUs that access the same variable may not have the corresponding up-to-date value in their caches and instead work with their own value — a behavior that leads to faulty executions. We call such a system *incoherent*. It is the task of a *cache-coherence protocol* to define a communication scheme that ensures coherence of the system while keeping the performance as high as possible.

A basic cache-coherence protocol is MSI [304]. To each local copy of x (and similarly for y) in a cache, it assigns one of the following three states:

- M (*modified*): The CPU has written to x, other CPUs did not see the change. The up-to-date value of x is private to the writing CPU.
- S (*shared*): The up-to-date value of x is shared among at least two caches.
- I (*invalid*): No value is yet assigned to x or the assigned value is outdated.

MSI employs the bus to communicate messages among the CPUs that may change the state in a cache. There are three types of messages. A *write request* (*write*) formulates that a CPU wants to write to x. Similarly, a *read request* (*read*) represents the fact that a CPU wants to update its outdated copy of x. Upon receiving an *invalidation message* (*inval*), a CPU has to invalidate its local copy of x — it is outdated now. The specification of MSI yields the finite automaton shown in Figure 7.2. A CPU can send ($!a$) and receive ($?a$) a message a. We do not elaborate on each of the state changes, instead we pick an example. If a CPU has an up-to-date copy of x and wants to write to it, it sends an invalidation message $!inval$ over the BUS to the other CPUs

and changes its state from S to M. All other CPUs receive ?*inval* and either change their state from S to I or stay in I if they have already been there.

MSI promises coherence for systems with arbitrarily many processors. To verify this, we model the protocol as a broadcast network. The specification given in Figure 7.2 reflects the code executed on the clients of the network. The broadcast network consists of arbitrarily many copies of these clients. An example computation involving three clients is given below:

$$(I, I, I) \xrightarrow{!write/?write/?write} (M, I, I) \xrightarrow{?read/!read/?read} (S, S, I).$$

Note that communication in the network is based on sending broadcast messages that each client, except from the sending client, has to receive.

To ensure coherence, MSI satisfies safety requirements. These can be checked on the corresponding broadcast network for the protocol. For instance, one of the requirements formulates that the state M can only be assigned to one CPU or cache at a time. We can rephrase it as a reachability problem in the broadcast network: if there is a computation that leads to a point where M is assigned to two or more clients, we have found a bug in the protocol. One can verify that such a computation does not exist [151].

An assumption that we made for MSI is that, once a message is sent, it is received by all other CPUs. The *communication topology* is perfect. However, when a fixed infrastructure is absent, like in ad hoc networks that are based on wireless communication, we may have message loss. In this case, the communication topology is *reconfigurable*. It may change during a computation, resulting in different listening clients for each sent message. In fact, a client ready to receive a message may ignore it, and it may be the case that no client receives it at all. Assume the communication in MSI is reconfigurable. Then, the protocol does not guarantee coherence since it breaks the above safety requirement. The following computation puts two caches into state M. Note that the sent write requests are not received by any CPU.

$$(I, I) \xrightarrow{!write/-} (M, I) \xrightarrow{-/!write} (M, M).$$

Broadcast networks on various communication topologies have attracted considerable attention in the literature. This includes work on reconfigurable topologies [95, 123, 124, 176], locally changing topologies [26], fixed topologies [2], a perfect topology like the one above [151], and topologies with a bounded diameter [125]. Moreover, there are variants and generalizations of broadcast networks which involve local strategies for clients [38], communication failures [126], and probabilistic elements [37, 65]. In the following, we focus on the verification of broadcast networks with reconfigurable topology.

7.1.1 Safety Verification

The wide spectrum of applications of reconfigurable broadcast networks, especially for ad hoc networks and cache-coherence protocols, has led to the formulation of two basic safety verification problems [124]. In the *broadcast network coverability problem* (BNC), the question is whether at least one client, participating in the current computation, can reach an unsafe or final state. While the problem is undecidable for a static topology [124], it has a surprisingly low complexity in the reconfigurable case. In fact, BNC is P-complete [123]. The second problem is the *broadcast network synchronization problem* (BNS). The problem asks whether all participating clients can reach a final state at the same time. Although seemingly harder than BNC, also BNS turned out to be P-complete [123, 176] for reconfigurable networks.

The results show that reconfigurable broadcast networks are a model for efficient safety verification. However, the results do not yield tight complexity bounds for liveness verification and model-checking although these problems were originally considered in the context of broadcast networks [147, 151].

7.1.2 Liveness Verification

Complexity of liveness verification for reconfigurable broadcast networks was first studied in [124]. The authors considered the *broadcast network liveness verification problem* (BNL)[10], a generalization of the coverability problem BNC. BNL asks whether there exists a *live computation* where at least one participating client can visit a final state infinitely many times. By a reduction to repeated coverability in Petri nets [154], BNL was shown to be solvable in EXPSPACE [124]. However, the only known lower bound is P-hardness [123], leaving a rather large gap which remained to be closed.

Contribution We contribute an algorithm solving BNL in polynomial time. It closes the aforementioned gap. The algorithm relies on a representation of live computations in terms of paths in an appropriate abstraction graph. Since the graph is of exponential size one can neither construct it nor immediately apply a path finding algorithm. Both would only yield an exponential-time procedure for detecting live computations. Instead, we show that a path exists if and only if there is a path in some particular normalform. Paths in normalform can then be found efficiently by a fixed-point iteration which terminates after polynomially many steps. Technically, the algorithm runs in time $O(p^4 \cdot t^2)$ where p is the number of states of the client in the given broadcast network and t is the size of its transition relation.

Although our result is a pure polynomial-time algorithm, we consider it a contribution to fine-grained complexity. This has two reasons. First, we give a precise upper bound in the parameters p and t for BNL which only

[10] The problem is also known as *repeated coverability*.

turned out to be of polynomial shape. In fact, the bound was only achieved while considering the parameter p in more detail. The second reason is that the polynomial-time algorithm for BNL solves questions on the fine-grained complexity of the *LTL model-checking problem* [290] of broadcast networks. Indeed, with our algorithm, the time needed to solve the model-checking problem depends polynomially on the size of the broadcast network and exponentially only on the size of the LTL formula [320]. In [97], we elaborate on the complexity in more detail. Here, we focus on the algorithm for BNL.

By a discussion with one of the authors of [37], we found that membership in P of BNL can also be deduced from a result given in their work. The idea is to construct a suitable vector addition systems with states (VASS) to simulate broadcast networks and to employ an algorithm of Kosaraju and Sullivan [239] for finding cycles. However, the approach requires combining non-trivial machinery like VASS and linear programming [239]. Our algorithm is comparatively simple and tailored to the problem. In fact, our fixed-point iteration is easy to implement and has a better time complexity. Moreover, our abstraction via paths and the normalform might be of independent interest in the verification of further parameterized models.

7.1.3 Fair Liveness

As we have seen, the liveness verification problem BNL generalizes the coverability problem BNC. For the related synchronization problem BNS, a generalization to liveness verification is missing. We introduce a variant of BNL that incorporates a notion of fairness which lifts the requirement of BNS to live computations. The *fair liveness verification problem* (FBNL) requires that all clients which participate infinitely often during a computation also see a final state infinitely often, a requirement also known as *compassion* [291].

Contribution We show that FBNL, like BNL, can be solved in polynomial time. The key is an instrumentation that compiles away the compassion requirement in polynomial time and reduces FBNL to finding cycles in broadcast networks. For the latter we can then apply the algorithm developed for BNL. By our results, safety and liveness verification for reconfigurable broadcast networks have the same time complexity. This phenomenon has been observed for other (parameterized) models as well [143, 152, 198, 199], even on the level of fine-grained complexity [93, 94, 96].

7.2 Complexity of Liveness in Broadcast Networks

We present the polynomial-time algorithm for BNL. To this end, we formally introduce the model of reconfigurable broadcast networks and the liveness verification problem in Section 7.2.1. In Section 7.2.2, we proceed by elaborating on the abstraction graph and its properties. We also introduce paths

in normalform and show that they characterize live computations. Finally, in Section 7.2.3, we present the fixed-point iteration which finds paths in normalform and show how it is applied to detect a live computation.

7.2.1 Broadcast Networks and BNL

A broadcast network is a concurrent system consisting of an arbitrary but finite number of identical clients that communicate by passing messages. To model such a network via finite-state systems, we assume that the messages are chosen from a finite *message domain* D. A message $a \in D$ can either be sent, denoted by $!a$, or received, $?a$. The communication operations that a client can then perform are summarized in the *set of operations*

$$OP(D) = \{!a, ?a \mid a \in D\}.$$

Clients When modeling the clients of a broadcast network, we abstract away the internal behavior and focus on the communication with other clients via the available communication operations from $OP(D)$. With this, a client is then simply given as a finite automaton over the set of operations.

Definition 7.1. A *client* is a finite automaton $P = (Q, OP(D), \delta, I)$.

Here, Q is the set of states of the client, $I \subseteq Q$ the set of initial states, and $\delta \subseteq Q \times OP(D) \times Q$ the transition relation. As usual, we extend δ to $OP(D)^*$ and write $q \xrightarrow{w} q'$ instead of $(q, w, q') \in \delta$. We do not specify final states of the client as they are not required to define broadcast networks. We will add them later as an input to the liveness verification problems of interest.

Broadcast Networks Broadcast networks fix an underlying message domain and a layout for the clients. The syntax is defined as follows.

Definition 7.2. A *(reconfigurable) broadcast network* is a tuple $N = (D, P)$ where D is a finite message domain and P is a client over $OP(D)$.

To define the semantics of a broadcast network, we need to make explicit how the communication among the clients in the network proceeds. This is typically achieved by defining the reconfigurable communication topology in terms of graphs that change over time [26, 123, 124]. However, our formulation avoids the usage of graphs. Instead, we tend to understand the reconfigurable communication topology as a shared memory where the clients can write to and read from. This moves broadcast networks closer to the related model of *leader contributor systems* which we analyze in Chapter 8.

During a communication phase in a broadcast network, one client sends a message and a number of other clients receive it. This induces a change of the current state in each client participating in the communication. We use

configurations to display the current states of all clients present in the network and a corresponding *transition relation on configurations* to model the state changes. We begin by defining the former. To this end, note that a broadcast network contains arbitrarily but finitely many clients.

Definition 7.3. Let the pair $N = (D, P)$ be a broadcast network with client $P = (Q, OP(D), \delta, I)$. A *configuration* of N is a tuple $c = (q_1, \ldots, q_k) \in Q^k$ where $k \in \mathbb{N}$ is the number of involved clients. The *set of configurations of* N is

$$Conf(N) = \bigcup_{k \in \mathbb{N}} Q^k.$$

A configuration c is called *initial* if each client is in an initial state. Consequently, the *set of initial configurations* is given by $Conf_I(N) = \bigcup_{k \in \mathbb{N}} I^k$.

For a configuration $c = (q_1, \ldots, q_k) \in Q^k$, we use $Set(c) = \{q_1, \ldots, q_k\} \subseteq Q$ to list the client states which occur within c. Moreover, we use the notation $c(i) = q_i$ to access the individual components of the configuration.

With configurations, we can model the communication in the broadcast network. Each communication phase induces a state change in the participating clients which is reflected by the following transition relation.

Definition 7.4. Let the tuple $N = (D, P)$ be a broadcast network with client $P = (Q, OP(D), \delta, I)$. The *transition relation of* N is a relation on configurations:

$$\rightarrow_N \subseteq Conf(N) \times D \times Conf(N).$$

It is defined as follows. Let $c = (q_1, \ldots, q_k)$ and $c' = (q'_1, \ldots, q'_k)$ be two configurations involving k clients and let $a \in D$ be a message. Then, there is a transition $c \xrightarrow{a}_N c'$ if the following three conditions are satisfied.

(1) There is a *sender*, a client $i \in [1..k]$ with transition $q_i \xrightarrow{!a} q'_i$.

(2) There is a number of *receivers*, a subset of clients $R \subseteq [1..k] \setminus \{i\}$ without the sender such that $q_j \xrightarrow{?a} q'_j$ for each $j \in R$.

(3) All other clients stay idle: for all $j \notin R \cup \{i\}$ we have $q_j = q'_j$.

To denote the indices of clients that contributed to the transition, we use the notation $idx(c \xrightarrow{a}_N c') = R \cup \{i\}$. Note that it may well be the case that $R = \emptyset$. This means that all clients, except from the sender i, ignore the message. Moreover, sender and receivers may change for each transition. This reflects the reconfigurable communication topology.

The transition relation $\rightarrow_N$ can be extended to words $w \in D^*$. We write $c \xrightarrow{w}_N c'$ for a sequence of consecutive transitions and call it a *computation* of the broadcast network N. For a computation $c \xrightarrow{w} c'$, we omit the index N

where it is appropriate. Moreover, we write $c \to_N^* c'$ if a corresponding word $w \in D^*$ with $c \xrightarrow{w}_N c'$ exists. If $|w| \geq 1$, we also use $c \to_N^+ c'$. The definition of *idx* can be extended to computations by listing all clients that contribute to one of the transitions. Note that $\to_N$ is only defined among configurations with the same number of clients. Hence, the number of clients does not change during a computation but is fixed a priori. This separates the model from others that feature dynamic thread creation [21].

We illustrate the notion of a computation with an example. To this end, reconsider the broadcast network modeling the cache-coherence protocol MSI.

Example 7.5. As we have seen in Section 7.1, the broadcast network $N = (D, P)$ for modeling MSI relies on the messages *write*, *read*, and *inval*. We abbreviate these messages and capture them in the domain $D = \{w, r, i\}$. The client P of N is given in Figure 7.3. It is the finite automaton from Figure 7.2 but over the simplified domain and without self loops in the states. In fact, the latter can be ignored since we have a reconfigurable communication topology where clients can ignore messages and remain in their current state.

We consider a computation of N. Assume we have two clients. Due to the reconfigurable communication topology, we can bring both into the state q_M:

$$(q_I, q_I) \xrightarrow{w} (q_M, q_I) \xrightarrow{w} (q_M, q_M).$$

Note that $idx((q_I, q_I) \xrightarrow{w} (q_M, q_I)) = \{1\}$ since the first client is the sender and the second ignores the message. Similarly, $idx((q_M, q_I) \xrightarrow{w} (q_M, q_M)) = \{2\}$. As argued in Section 7.1, this breaks a safety requirement of the MSI protocol in the case of a reconfigurable communication topology.

Consider the following computation, again involving two clients:

$$(q_I, q_I) \xrightarrow{w} (q_M, q_I) \xrightarrow{r} (q_S, q_S) \xrightarrow{i} (q_M, q_I).$$

Since the configuration (q_M, q_I) repeats, we can actually add a loop to the computation and repeat it an arbitrary number of times or even infinitely:

$$(q_I, q_I) \xrightarrow{w} (q_M, q_I) \xrightarrow{r} (q_S, q_S) \xrightarrow{i} (q_M, q_I) \xrightarrow{r} (q_S, q_S) \xrightarrow{i} (q_M, q_I) \xrightarrow{r} \dots$$

This shows that there is a computation of the broadcast network in which the first client visits the state q_M infinitely often. Intuitively this means, the client can infinitely often modify the local variable stored in its cache.

Liveness Verification It is the task of the liveness verification problem BNL to decide the existence of infinite computations like the one depicted in Example 7.5. Let $N = (D, P)$ be a reconfigurable broadcast network with client $P = (Q, OP(D), \delta, I)$. Formally, an *infinite computation* is a sequence

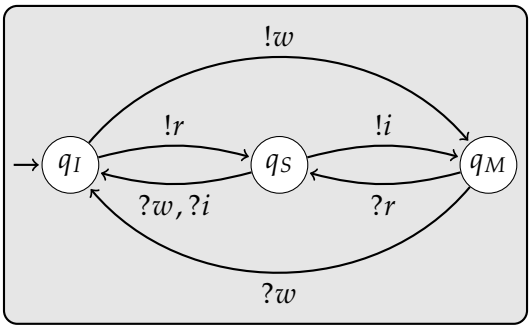


Figure 7.3: The client P of the broadcast network $N = (D, P)$ with message domain $D = \{w, r, i\}$. It is a modified variant of the client modeling the MSI cache-coherence protocol, see Figure 7.2 for a comparison.

$\sigma = c_0 \to c_1 \to \dots$ of infinitely many consecutive transitions. Such a computation is called *initialized* if $c_0 \in Conf_I(N)$. To detect the infinite occurrence of certain states within σ, we fix a set $Q_F \subseteq Q$ of final states. Let

$$Fin(\sigma) = \{i \in \mathbb{N} \mid \exists^\infty j : c_j(i) \in Q_F\}$$

denote the set of clients that visit a final state infinitely often.

The *broadcast network liveness verification problem* (BNL) takes a broadcast network $N = (D, P)$ with client $P = (Q, OP(D), \delta, I)$ and a set of final states $Q_F \subseteq Q$. It asks whether there exists an infinite initialized computation σ of N in which at least one client visits a final state infinitely often, $Fin(\sigma) \neq \emptyset$.

BNL

Input: A broadcast network $N = (D, P)$, a set of final states Q_F.
Question: Is there an initialized infinite σ with $Fin(\sigma) \neq \emptyset$?

The problem BNL was introduced as *Repeated Coverability* in [124]. In the same work, it was shown to be solvable in EXPSPACE. However, the only known lower bound is P-hardness [123], leaving a rather large gap to its upper bound. We present an algorithm for BNL which runs in time $O(p^4 \cdot t^2)$, where p is the number of states of the client and t the number of transitions.

Theorem 7.6. *The problem BNL can be solved in time $O(p^4 \cdot t^2)$.*

Combining Theorem 7.6 with the known hardness result for BNL closes the gap and shows that the problem is complete for the class P.

Corollary 7.7. *The problem BNL is P-complete.*

7.2.2 Graph Abstraction

The first step to our polynomial-time algorithm for BNL is a characterization of infinite computations in terms of paths on a finite abstraction graph. In order to prove it, we need to relate the existence of an infinite computation to the existence of a finite one. As common in the handling of infinite computations on finite automata, this is achieved by splitting such a computation into a prefix and a suitable cycle that can be iterated. Fix a broadcast network $N = (D, P)$ with client $P = (Q, OP(D), \delta, I)$ and a set of final states $Q_F \subseteq Q$. The following lemma makes the splitting more precise.

Lemma 7.8. *There is an infinite computation $\sigma = c_0 \to c_1 \to \dots$ with $Fin(\sigma) \neq \emptyset$ if and only if there is a computation of the form $c_0 \to^* c \to^+ c$ with $Set(c) \cap Q_F \neq \emptyset$.*

Proof. Assume there is a computation $c_0 \to^* c \to^+ c$ with $Set(c) \cap Q_F \neq \emptyset$. Then $c \to^+ c$ can be iterated infinitely often to obtain an infinite computation visiting Q_F infinitely often. In turn, in any infinite sequence from Q^k one can find a repeating configuration by the pigeon hole principle. This in particular holds for the infinite sequence of configurations containing final states. □

Abstraction Graph By Lemma 7.8, BNL can be solved by finding a reachable configuration c that contains a final state and that can be iterated. However, broadcast networks are parameterized, meaning that the number of involved clients in c is not fixed. Hence, we need to deal with an infinite number of configurations and cannot immediately search for c. To overcome this we need to abstract configurations and their transition relation in a suitable way. In fact, we will devise a finite abstraction graph the cycles of which simulate cycles on configurations. Then we only need to search for a cycle in the graph.

The basic idea of the abstraction graph is to represent configurations by sets. In fact, a vertex of the graph keeps track of those sets of states $S \subseteq Q$ that are acquired by the clients. This means that a state $s \in S$ is present in a vertex, if the corresponding configuration contains at least one client in s. The abstraction is inspired by the powerset construction for finite automata [296], where we keep the set of states in which the automaton might be in. However, different from this construction, keeping a state $s \in S$ in our abstraction means that there is not at most one client in s but an arbitrary number of clients greater than one. As a consequence, an edge from s to s' may have two effects. Some of the clients that are currently in s change their state to s' while others stay in s. In that case, the set of acquired states is updated to $S' = S \cup \{s'\}$. Alternatively, all clients may change their state to s', in which case we obtain $S' = (S \setminus \{s\}) \cup \{s'\}$. Hence, the edges of our abstraction graph need to respect that certain states get *killed* while others get *generated* when transitioning to a new vertex. The following definition takes this into account.

Definition 7.9. The *abstraction graph* of the broadcast network N is a directed graph $\mathcal{G} = (V, \rightarrow_{\mathcal{G}})$. The vertices $V = \bigcup_{k \leq p} \mathcal{P}(Q)^k$ are k-tuples of sets of states where $k \leq p = |Q|$. For the edges $\rightarrow_{\mathcal{G}}$, we need some preliminary notation.

Let $S \subseteq Q$ and $a \in D$. The *successor states of S (under receiving a)* are those states that can be reached from S via a $?a$-transition in the client:

$$succ_{?a}(S) = \{q' \in Q \mid \exists q \in S : q \xrightarrow{?a} q'\}.$$

The states within the set S that have an outgoing $?a$-transition in the client, are captured in the corresponding set of *enabling states*:

$$enabled_{?a}(S) = \{q \in S \mid \exists q' \in Q : q \xrightarrow{?a} q'\}.$$

Let $V_1 = (S_1, \ldots, S_k)$ and $V_2 = (S'_1, \ldots, S'_k)$ be two vertices in V. There is an edge $V_1 \rightarrow_{\mathcal{G}} V_2$ in the graph if the following three conditions are satisfied.

(1) There is a sender, an index $i \in [1..k]$, states $s \in S_i$ and $s' \in S'_i$, as well as an element $a \in D$ such that $s \xrightarrow{!a} s'$ is a send transition of the client.

(2) There are clients that move upon receiving a. For each $j \in [1..k]$ there are sets of states $Gen_j \subseteq succ_{?a}(S_j)$ and $Kill_j \subseteq enabled_{?a}(S_j)$ such that

$$S'_j = \begin{cases} (S_j \setminus Kill_j) \cup Gen_j, & \text{for } j \neq i, \\ (U_j \setminus Kill_j) \cup Gen_j \cup \{s'\}, & \text{for } j = i, \end{cases}$$

where U_j is either S_j or $S_j \setminus \{s\}$.

(3) Killed states are properly replaced by generated ones. For each index $j \in [1..k]$ and state $q \in Kill_j$, we have that $succ_{?a}(q) \cap Gen_j$ is non-empty.

Intuitively, an edge in the abstraction graph $\mathcal{G}$ mimics a transition in the broadcast network N without making explicit the precise configurations. Condition (1) requires a sender, a component i that is capable of sending a particular message $a \in D$. Clients receiving the message a are represented in (2). The subset $Gen_j \subseteq succ_{?a}(S_j)$ holds those states that are generated, reached by clients performing a corresponding receive transition. These states are added to the set S_j. As mentioned above, states can also get killed. If, during a receive transition, all clients in a particular state move to a newly generated one, the original state will not be present anymore in a configuration. We capture killed states in the set $Kill_j \subseteq enabled_{?a}(S_j)$ and remove them from S_j. Note that for the sender, the i-th component, we add the state s' due to the send transition. Moreover, we need to distinguish whether s gets killed or not. This makes up the definition of the set U_i. Finally, Condition (3) guarantees that each killed state is properly replaced by a generated state.

We illustrate the definition of the abstraction graph with an example. It shows that cyclic computations can be mimicked by cycles in the graph.

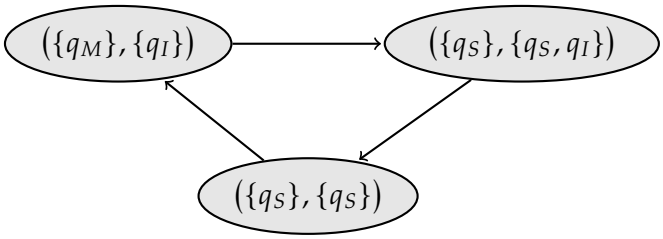


Figure 7.4: A cycle in the abstraction graph $\mathcal{G}$ mimicking the cyclic computation from Example 7.10. The vertices involve two components of sets. The first component represents the first client, the second component represents the second and the third client of the computation.

Example 7.10. Reconsider the broadcast network N from Example 7.5. As argued above, it admits a cyclic computation involving two clients. Now we consider a cyclic computation of the network which involves three clients:

$$(q_M, q_I, q_I) \xrightarrow{r} (q_S, q_S, q_I) \xrightarrow{r} (q_S, q_S, q_S) \xrightarrow{i} (q_M, q_I, q_I).$$

The cyclic computation can be mimicked by a cycle in the abstraction graph $\mathcal{G}$ of N. In fact, consider the cycle given in Figure 7.4. It is induced by the computation but we need to argue that it is a proper cycle in $\mathcal{G}$. To this end, we need to check whether the edges satisfy Conditions (1), (2), and (3) from Definition 7.9. We show it for the edge $(\{q_M\}, \{q_I\}) \rightarrow_{\mathcal{G}} (\{q_S\}, \{q_S, q_I\})$.

Condition (1) is satisfied since there is a corresponding send transition $q_I \xrightarrow{!r} q_S$ in the second component. To see that Condition (2) is satisfied, note that $Gen_1 = \{q_s\}$ and $Kill_1 = \{q_M\}$. In the component of the sender, we have $Gen_2 = Kill_2 = \emptyset$ and since one client remains in q_I, we set $U_2 = \{q_I\}$. Note that q_S is automatically added to the component since it is the target state of the send transition. For Condition (3), we have that $succ_{?r}(q_M) \cap Gen_1 = \{q_S\}$ is not empty. The reasoning for the remaining edges is similar.

The observation made in the example is true in general. Any cyclic computation implies a cycle in the abstraction graph. Moreover, the converse direction is also true: a cycle in the abstraction graph entails a cyclic computation of the broadcast network. The following lemma states the equivalence. It is a major step in the development of our algorithm for BNL.

Lemma 7.11. *There is a cycle* $(\{s_1\}, \ldots, \{s_k\}) \rightarrow_{\mathcal{G}}^{+} (\{s_1\}, \ldots, \{s_k\})$ *in* $\mathcal{G}$ *if and only if there is a configuration* c *with* $Set(c) = \{s_1, \ldots s_k\}$ *and* $c \rightarrow_N^{+} c$.

The lemma also explains the restriction of the vertices in the abstraction graph to k-tuples of sets of states, with $k \leq p$. In fact, we are searching for a cycle in a configuration and the latter can have at most p different states.

Note that we need to keep the sets of states separately. This is to ensure that, for every state s_i, the corresponding clients starting in s_i perform a cyclic computation and all go back to s_i. The proof of Lemma 7.11 is rather technical and lengthy. We omit it here and refer to Appendix B.2.1.

A first idea to find a cyclic and thus infinite computation of the broadcast network would be to search for a cycle in the abstraction graph like the one depicted in Lemma 7.11. However, the graph is of exponential size and the search would result in an exponential-time procedure. Since our goal is a polynomial-time procedure, we cannot afford running a cycle finding algorithm on the abstraction graph immediately. Instead, we show that if a cycle in the graph exists, then there is also one in a certain normalform. Then it suffices to search for cycles in normalform. In Section 7.2.3, we will see that these can be found in polynomial time by a fixed-point iteration.

Normalform We introduce the normalform — not only for cycles but for general paths. The idea is to split a path into prefix and suffix. In the prefix, we can only have edges that increase the sets stored in the components of the current vertex. In the suffix, all edges decrease the stored sets. To define the normalform, let $V_1 = (S_1, \ldots, S_k)$ and $V_2 = (S'_1, \ldots, S'_k)$ be two vertices of the abstraction graph. By $V_1 \sqsubseteq V_2$, we denote the *componentwise inclusion* of the vertices. Recall that this means $S_j \subseteq S'_j$ for each component $j \in [1..k]$.

Definition 7.12. Let $V_1 \to_{\mathcal{G}} V_2 \to_{\mathcal{G}} \cdots \to_{\mathcal{G}} V_n$ be a path in the abstraction graph $\mathcal{G}$. It is said to be in *normalform*, if it takes the shape

$$V_1 \to^*_{\mathcal{G}} V_m \to^*_{\mathcal{G}} V_n$$

such that the following conditions hold. In the prefix $V_1 \to^*_{\mathcal{G}} V_m$, the sets of states increase monotonically, $V_i \sqsubseteq V_{i+1}$ for all $i \in [1..m-1]$. In the suffix $V_m \to^*_{\mathcal{G}} V_n$, the sets decrease monotonically, $V_i \sqsupseteq V_{i+1}$ for all $i \in [m..n-1]$.

Note that an arbitrary path in $\mathcal{G}$ does not need to respect the order given by the normalform. It may start with an edge that decreases a component of the vertex and continue with an increasing edge. Moreover, general edges can increase and decrease several components of a vertex at the same time. The following lemma shows that, nevertheless, we can assume the path to be in normalform. In fact we show that, if there is a path, there is also one in normalform. This will ease the search for cycles in $\mathcal{G}$ significantly.

Lemma 7.13. *Let V_1, V_2 be vertices of the abstraction graph $\mathcal{G}$. There exists a path from V_1 to V_2 if and only if there exists a path in normalform from V_1 to V_2.*

Proof. If $V_1 \to^*_{\mathcal{G}} V_2$ is a path in normalform, there is nothing to prove. For the other direction, let $\sigma = V_1 \to^*_{\mathcal{G}} V_2$ be an arbitrary path. To get a path in normalform, we first simulate the edges of σ in such a way that no states are

deleted. In a second step, we erase the states that should have been deleted. We have to respect a particular deletion order to obtain a valid path.

Let $\sigma = U_1 \to_{\mathcal{G}} U_2 \to_{\mathcal{G}} \cdots \to_{\mathcal{G}} U_\ell$ with $U_1 = V_1$ and $U_\ell = V_2$. We inductively construct an increasing path $\sigma_{inc} = U'_1 \to_{\mathcal{G}} \cdots \to_{\mathcal{G}} U'_\ell$ such that $U_i \sqsubseteq U'_j$ for all $i \leq j$. For the base case, we set $U'_1 = U_1$. Now assume σ_{inc} has already been constructed up to vertex U'_j. In the path σ, there is an edge $e = U_j \to_{\mathcal{G}} U_{j+1}$. Since $U'_j \sqsupseteq U_j$, we can simulate e on U'_j: all states needed for the execution are present in U'_j. Moreover, we mimic e in such a way that no states get deleted. This is achieved by setting the corresponding *Kill* sets to be empty. Hence, we get $U'_j \to_{\mathcal{G}} U'_{j+1}$ with $U'_{j+1} \sqsupseteq U'_j$ since we do not delete any states and $U'_{j+1} \sqsupseteq U_{j+1}$ since we simulate the edge e.

Let $V'_2 = U'_\ell$. The states in V'_2 that are not in V_2 are those that were deleted along σ. We construct a decreasing path $\sigma_{dec} = V'_2 \to^*_{\mathcal{G}} V_2$ which deletes all these states. To this end, let $V'_2 = (T_1, \ldots, T_k)$ and $V_2 = (S_1, \ldots, S_k)$. An edge in σ deletes sets of states in each component $i \in [1..k]$. Hence, to mimic the deletion in the decreasing path, we need to consider subsets of

$$Del = \bigcup_{i \in [1..k]} (T_i \setminus S_i) \times \{i\}.$$

Note that the index i in a tuple (s, i) displays the component the state s is in.

We define an equivalence relation $\sim$ over *Del*: we have $(x, i) \sim (y, t)$ if and only if the last occurrence of x in component i and y in component t in the path σ coincide. Hence, two elements are equivalent if they get deleted at the same time and do not appear again in σ. Note that the definition indeed yields an equivalence relation. Now we introduce an order on the equivalence classes. We have $[(x, i)]_\sim < [(y, t)]_\sim$ if and only if the last occurrence of (x, i) was before the last occurrence of (y, t). Since the order is total, we get a partition of *Del* into equivalence classes $P_1, \ldots, P_n$ such that $P_j < P_{j+1}$.

Finally, we are able to construct $\sigma_{dec} = K_0 \to_{\mathcal{G}} \cdots \to_{\mathcal{G}} K_n$ with $K_0 = V'_2$ and $K_n = V_2$. During each edge $K_{j-1} \to_{\mathcal{G}} K_j$, we delete precisely the elements in P_j and do not add further states. Deleting the states P_j in σ is due to an edge $e = U_k \to_{\mathcal{G}} U_{k+1}$. We mimic e in such a way that no state gets added by setting the corresponding *Gen* sets to be empty. Since we respect the order $<$ with the deletions, the simulation of e from K_{j-1} is possible. Suppose, we need a state s in component t to simulate e but the state is not available in component t of K_{j-1}. Then it was deleted before, we actually have $(s, t) \in P_1 \cup \ldots \cup P_{j-1}$. But this contradicts the fact that s is present in U_k. Hence, all the needed states are available and we can perform the deletions. Since after the last edge of σ_{dec} we have deleted all states within *Del*, we get that $K_n = V_2$. □

Before we show how cycles in normalform can be found algorithmically, we consider an example which illustrates the above proof.

Example 7.14. Recall the cycle in the abstraction graph from Example 7.10:

$$\sigma = (\{q_M\}, \{q_I\}) \to_{\mathcal{G}} (\{q_S\}, \{q_S, q_I\}) \to_{\mathcal{G}} (\{q_S\}, \{q_S\}) \to_{\mathcal{G}} (\{q_M\}, \{q_I\}).$$

It clearly violates the normalform. To obtain a corresponding cycle in normalform, we perform the construction from Lemma 7.13. First, we construct an increasing path σ_{inc} that simulates only the generations of σ:

$$\sigma_{inc} = (\{q_M\}, \{q_I\}) \to_{\mathcal{G}} (\{q_M, q_S\}, \{q_S, q_I\}).$$

Note that σ_{inc} actually has two more edges that just loop in the last vertex. We omit these. The deletions of σ are simulated by the path σ_{dec}. It deletes the state q_S from both components. The states q_M and q_I are not deleted since they still occur in the last vertex of σ. We have:

$$\sigma_{dec} = (\{q_M, q_S\}, \{q_S, q_I\}) \to_{\mathcal{G}} (\{q_M\}, \{q_I\}).$$

The cycle $\sigma_{inc}.\sigma_{dec}$ is in normalform and like σ, starts and ends in $(\{q_M\}, \{q_I\})$.

7.2.3 Algorithm

We present the second step to our algorithm for BNL, a fixed-point iteration that finds normalform-cycles in polynomial time. More precise, the iteration decides the existence of a cycle $(\{s_1\}, \ldots, \{s_k\}) \to_{\mathcal{G}}^{+} (\{s_1\}, \ldots, \{s_k\})$ in normalform. According to the results presented in the previous section, this witnesses a cyclic computation of the broadcast network. Subsequently, we show how the cycle detection can be used to decide the existence of an infinite computation that visits a final state infinitely often and thus to solve BNL. For the remaining section, we fix a broadcast network $N = (D, P)$ with client $P = (Q, OP(D), \delta, I)$ and a set of final states $Q_F \subseteq Q$.

Operators In order to formulate the existence of a cycle in terms of a fixed point, we require some suitable operators. The idea is to mimic the monotonically increasing prefix of a cycle in normalform by a *post operator* and the monotonically decreasing suffix by a *pre operator*. Both operators are applied to a vertex $(\{s_1\}, \ldots, \{s_k\})$ and the result is intersected. This way we hope to find an intermediary vertex that is reachable from $(\{s_1\}, \ldots, \{s_k\})$ and that can reach $(\{s_1\}, \ldots, \{s_k\})$ again to close the cycle.

The challenge in designing the post operator is to ensure that receive transitions are actually enabled by sends that lead to states in the intersection. The same has to be ensured for the pre operator. We solve this by using a fixed-point iteration. Indeed, in the first step of the iteration, we determine the *post* of $(\{s_1\}, \ldots, \{s_k\})$ and intersect it with the *pre* of the vertex. In the next step, we constrain the *post* and the *pre* to visiting only states within the previous intersection. The results are intersected again, which may remove further

states. Hence, the computation is repeated relative to the new intersection. We continue until the intersection stabilizes and the fixed-point is found.

In order to constrain the operators as mentioned above, we do not work with standard post and pre operators. Instead, we require both to have an additional input, besides the vertex, which can be used to restrict the underlying broadcast network to a certain set of states.

Definition 7.15. Let $k \leq p$ be an integer and $C = (C_1, \ldots, C_k) \in \mathcal{P}(Q)^k$ a k-tuple of sets of states. The *post operator*, denoted by $post_C$, is a mapping over the domain $\mathcal{P}(Q)^k$. It is defined by $post_C(X_1, \ldots, X_k) = (X'_1, \ldots, X'_k)$, where

$$\begin{aligned} X'_i &= \{q' \in Q \mid \exists q \in X_i : q \xrightarrow{!a}_{P\downarrow_{C_i}} q'\} \\ &\quad \cup \{q' \in Q \mid \exists q_1, q_2 \in X_\ell : \exists q \in X_i : q_1 \xrightarrow{!a}_{P\downarrow_{C_\ell}} q_2 \wedge q \xrightarrow{?a}_{P\downarrow_{C_i}} q'\}. \end{aligned}$$

Here, $P\downarrow_{C_i}$ denotes the automaton obtained from the client P by restricting it to the set of states C_i. Similarly, we define the *pre operator*, denoted by pre_C. We set $pre_C(X_1, \ldots, X_k) = (X'_1, \ldots, X'_k)$ with

$$\begin{aligned} X'_i &= \{q \in Q \mid \exists q' \in X_i : q \xrightarrow{!a}_{P\downarrow_{C_i}} q'\} \\ &\quad \cup \{q \in Q \mid \exists q_1, q_2 \in X_\ell : \exists q' \in X_i : q_1 \xrightarrow{!a}_{P\downarrow_{C_\ell}} q_2 \wedge q \xrightarrow{?a}_{P\downarrow_{C_i}} q'\}. \end{aligned}$$

Let us consider the definition of the post operator in more detail. In the i-th component, it collects all states that are either reachable via a send transition starting in X_i or a receive transition starting in X_i. The receive, however, has to be enabled by a corresponding send which may be performed among states in a different component X_ℓ. Additionally, the used transitions have to respect the restriction imposed by C. Each transition used in the i-th component must be available in the automaton $P\downarrow_{C_i}$. The pre operator is similar. The only difference is that we now collect predecessors of states instead of successors.

Post and pre operator are both monotone in C. This means, if $C \sqsubseteq C'$, we have $post_C(X_1, \ldots, X_k) \sqsubseteq post_{C'}(X_1, \ldots, X_k)$ for each vector $(X_1, \ldots, X_k)$. Similarly for *pre*. Monotonicity is essential when we want to apply a fixed-point iteration involving the operators. The second important property is that we can compute their (reflexive) transitive closures in polynomial time.

Lemma 7.16. *Let $C = (C_1, \ldots, C_k) \in \mathcal{P}(Q)^k$ be a k-tuple of sets of states. The closures $post^+_C(X_1, \ldots, X_k)$ and $pre^*_C(X_1, \ldots, X_k)$ can be computed in time $\mathcal{O}(p^2 \cdot t^2)$.*

Proof. Both closures can be computed by a saturation. For $post^+_C(X_1, \ldots, X_k)$, we keep k sets $R_1, \ldots, R_k \subseteq Q$, each collecting the successors of a particular component. Initially, we set $R_i = X_i$. The defining equation of X'_i in $post^+_C(X_1, \ldots, X_k)$ already gives the saturation. One substitutes X_i by R_i and X_ℓ by R_ℓ on the right side. The resulting set of states is added to the current R_i. This process is applied to each component and repeated until the R_i

do not change anymore, namely until the fixed point is reached. There is a subtlety here: since we want to compute $post^+_C(X_1, \ldots, X_k)$, we do not add the newly computed states to R_i in the first iteration. Instead, we replace R_i by these states to ensure that at least one transition was taken.

Let us analyze the time requirement of the saturation. After updating the R_i in each component, we either already terminated or added at least one new state to a set R_i. Since there are $k \leq p$ of these sets and each one is a subset of Q, we need to update the sets R_i at most p^2 many times. For a single update, the dominant time factor comes from finding an appropriate send transition to a receive transition. This can be achieved in time $O(t^2)$. Altogether, we can compute the closure $post^+_C(X_1, \ldots, X_k)$ within the time bound stated above.

Computing the closure $pre^*_C(X_1, \ldots, X_k)$ is similar. One can apply the above saturation and only needs to reverse the transitions in the client. □

With post and pre operator at hand, we can now turn the above discussion into an actual fixed-point iteration that finds cycles in normalform. The following lemma shows the correctness of the approach and, moreover, that the fixed-point can be computed in polynomial time.

Lemma 7.17. *There is a cycle* $(\{s_1\}, \ldots, \{s_k\}) \to^+_{\mathcal{G}} (\{s_1\}, \ldots, \{s_k\})$ *in the abstraction graph if and only if there is a non-trivial solution to the equation*

$$C = post^+_C(\{s_1\}, \ldots, \{s_k\}) \cap pre^*_C(\{s_1\}, \ldots, \{s_k\}).$$

Moreover, such a solution can be found in time $O(p^4 \cdot t^2)$.

Proof. We use a Kleene iteration [330] to compute the greatest solution. Initially we set $C = (Q, \ldots, Q)$ to be the largest element in the underlying domain $(\mathcal{P}(Q)^k, \sqsubseteq)$. The iteration invokes Lemma 7.16 as a subroutine to compute $post^+_C(\{s_1\}, \ldots, \{s_k\})$ and $pre^*_C(\{s_1\}, \ldots, \{s_k\})$ each step and intersects the two resulting vectors in each component. We take the intersection as a new vector C and repeat the process. Since both involved operators are monotone in C, the process finds the greatest non-trivial solution if it exists.

Every step of the iteration reduces the number of states in C by at least one. Hence, we are guaranteed to terminate after $O(p^2)$ many iterations. Since computing the new C each iteration takes time $O(p^2 \cdot t^2)$ by Lemma 7.16, the total time that we need to find a solution is at most $O(p^4 \cdot t^2)$.

It is left to prove the correctness of the approach. To this end, let a cycle $(\{s_1\}, \ldots, \{s_k\}) \to^+_{\mathcal{G}} (\{s_1\}, \ldots, \{s_k\})$ in $\mathcal{G}$ be given. By Lemma 7.13, we can assume it to be in normalform. Hence, there is an increasing part $(\{s_1\}, \ldots, \{s_k\}) \to^*_{\mathcal{G}} C$ and a decreasing part $C \to^*_{\mathcal{G}} (\{s_1\}, \ldots, \{s_k\})$ of the cycle. Then, C is a non-trivial solution to the above equation.

For the other direction, let a solution C be given. Then we clearly have the inclusion $C \sqsubseteq post^+_C(\{s_1\}, \ldots, \{s_k\})$. Since $post^+_C(\{s_1\}, \ldots, \{s_k\}) \sqsubseteq C$ holds by definition, we obtain equality. Hence, we can construct a monotonically

increasing path $(\{s_1\}, \ldots, \{s_k\}) \rightarrow^*_{\mathcal{G}} C$. Similarly, we can construct a monotonically decreasing path $C \rightarrow^*_{\mathcal{G}} (\{s_1\}, \ldots, \{s_k\})$ and get the desired cycle. □

Fixed-Point Algorithm With the fixed-point iteration, we can decide the existence of a cyclic computation $c \rightarrow^+_N c$ with $Set(c) = \{s_1, \ldots, s_k\}$. However, it is yet open on which states $s_1, \ldots, s_k$ we have to perform the search for a cycle in order to find a live computation that visits the final states infinitely often. After all, we need that the corresponding configuration c can be reached from an initial configuration. The idea is to use the set of all states that are reachable from an initial state. In fact, there is a live computation if and only if there is one involving all those states. If a state is not active during the cycle, the corresponding clients will just stop moving after the initial phase.

The states that are reachable from an initial state can be computed in polynomial time [123]. We compute these as a first step in our algorithm for BNL and then apply the above fixed-point iteration. Algorithm 7.1 summarizes our approach. The dominant time factor comes from the involved fixed-point iteration. Hence, the time estimation of Theorem 7.6 follows. Correctness is due to Lemmas 7.17, 7.11, and 7.8.

Algorithm 7.1 Broadcast Network Liveness Verification

Input: A broadcast network $N = (D, P)$ and a set of final states $Q_F \subseteq Q$.
Output: *True*, if there is an initialized σ with $Fin(\sigma) \neq \emptyset$. *False* otherwise.

1: use algorithm of [123] to compute reachable states $\{s_1, \ldots, s_k\}$ in P
2: **if** $\{s_1, \ldots, s_k\} \cap Q_F = \emptyset$ **then**
3: return *false* // Check is needed according to Lemma 7.8.
4: **end if**
5: set $S = (\{s_1\}, \ldots, \{s_k\})$, $C = Q^k$, and $C' = \emptyset$.
6: **while** $C \neq C'$ **do** // Computation of greatest fixed point.
7: set $C' = C$
8: compute $C = post^+_{C'}(S) \cap pre^*_{C'}(S)$ with algorithm from Lemma 7.16
9: **end while**
10: **if** $C = \emptyset$ **then**
11: return *false* // Exclude trivial solution.
12: **end if**
13: return *true*

7.3 Complexity of Fair Liveness

The problem BNL does not take fairness into account. A client may contribute to the live computation and help the distinguished client reach a final state without ever making progress towards its own final state. We formulate a

variant of BNL that incorporates a notion of fairness in that each participating client needs to visit a final state infinitely often. In Section 7.3.1, we introduce the problem formally. Subsequently, in Section 7.3.2, we solve it by a reduction to BNL. Hence, also the fair variant of BNL can be solved in polynomial time.

7.3.1 Fair Computations and FBNL

The notion of fairness dates back to Pnueli and Sa'ar [291] and strengthens the requirement that we have on a computation. Indeed, instead of only one client visiting a final state infinitely often, we now require that all clients contributing infinitely often to the computation also see a final state infinitely often. The search for this kind of computations seems harder than the problem BNL. But we show that they can also be found in polynomial time.

Fair Computations To formalize fairness in our setting, fix a broadcast network $N = (D, P)$ with client $P = (Q, OP(D), \delta, I)$ and final states $Q_F \subseteq Q$. Moreover, let $\sigma = c_0 \to c_1 \to \dots$ be an initialized infinite computation of N. Recall that the set of clients visiting a final state infinitely often during σ is denoted by $Fin(\sigma)$. To capture all clients participating infinitely often in the computation — without the requirement of visiting a final state — we use

$$Inf(\sigma) = \{i \in \mathbb{N} \mid \exists^\infty j : i \in idx(c_j \to c_{j+1})\}.$$

Definition 7.18. A *fair computation* of the broadcast network N is an initialized infinite computation σ that satisfies the inclusion $Inf(\sigma) \subseteq Fin(\sigma)$.

Before we formulate the search for fair computations as a decision problem, let us consider an example of a fair and an unfair computation.

Example 7.19. Recall the broadcast network from Example 7.5. As we have seen, it admits the following infinite computation:

$$\sigma = (q_I, q_I) \xrightarrow{w} (q_M, q_I) \xrightarrow{r} (q_S, q_S) \xrightarrow{i} (q_M, q_I) \xrightarrow{r} (q_S, q_S) \xrightarrow{i} (q_M, q_I) \xrightarrow{r} \dots$$

When we set $Q_F = \{q_M\}$, then σ is not fair according to Definition 7.18. In fact, we have $Inf(\sigma) = \{1, 2\}$ since both clients contribute infinitely often to the computation. But $Fin(\sigma) = \{1\}$ since only the first client visits q_M infinitely often. Consequently, we do not have the required inclusion.

The broadcast network also has a fair computation. Indeed, the clients can take turns visiting the final state q_M as follows:

$$\rho = (q_I, q_I) \xrightarrow{w} (q_M, q_I) \xrightarrow{w} (q_I, q_M) \xrightarrow{w} (q_M, q_I) \xrightarrow{w} \dots$$

For this computation, we have $Inf(\rho) = Fin(\rho) = \{1, 2\}$. Hence, ρ is fair.

Fair Liveness Verification We are ready to formalize the search for live computations in the *fair liveness verification problem* (FBNL). Similar to BNL, the problem takes a broadcast network $N = (D, P)$ with client $P = (Q, OP(D), \delta, I)$ and a set of final states $Q_F \subseteq Q$ as input and asks whether N has a fair computation, a computation σ with $Inf(\sigma) \subseteq Fin(\sigma)$.

FBNL

Input: A broadcast network $N = (D, P)$, a set of final states Q_F.
Question: Is there an initialized infinite σ with $Inf(\sigma) \subseteq Fin(\sigma)$?

Concerning the complexity of the problem, only a lower bound is known. In fact, P-hardness of FBNL follows from [123]. But so far, no work has been devoted to finding an upper bound. We prove that FBNL can be solved within the same polynomial time bound as BNL. The result is surprising since the fairness requirement seems to make FBNL harder than BNL. However, we show that it can be compiled away without creating much overhead.

Theorem 7.20. *The problem FBNL can be solved in time* $O(p^4 \cdot t^2)$.

Like for BNL, we can deduce that FBNL is P-complete. With the result, we have shown that both liveness verification problems have the same complexity as their corresponding safety verification counterparts. Summing up, this means that safety and liveness verification for reconfigurable broadcast networks do not differ in terms of time complexity.

Corollary 7.21. *The problem FBNL is* P*-complete.*

7.3.2 Reduction to BNL

We present the upper bound for FBNL. To find fair computations of a broadcast network, the idea is to apply the algorithm for BNL to an instrumentation of the network that compiles away the fairness requirement. For constructing the instrumentation, we first prove that fair computations can be decomposed, similarly to live computations, into a prefix and a *good cycle*. In a good cycle, each participating client visits a final state. The mentioned instrumentation is then constructed in such a way that it contains a plain cycle if and only if the original network contains a good cycle. Hence, fairness — in terms of good cycles — gets compiled away. For finding cycles in the instrumentation, we can then employ an adjusted version of the fixed-point iteration for BNL.

Good Cycles A *good cycle*, or more general a *good computation*, ensures that all clients contributing to the computation visit a final state at least once. The notion allows for a simpler representation of the fairness requirement which

we can handle algorithmically. For the remaining section, we fix a broadcast network $N = (D, P)$ with $P = (Q, OP(D), \delta, I)$ and final states $Q_F \subseteq Q$.

Definition 7.22. A finite computation $\sigma = c_0 \to c_1 \to \cdots \to c_n$ of N is called *good for* Q_F, denoted by $c_1 \Rightarrow_{Q_F} c_n$ if each client contributing to σ visits a final state. Formally, if $i \in idx(\sigma)$ then there is a $j \in [1..n]$ such that $c_j(i) \in Q_F$.

Note that a good computation does not need to be initialized. Instead, it may start at any configuration. If a good computation forms a cycle, we also call it a *good cycle*. The following lemma decomposes a fair computation into a prefix and a good cycle. It can be seen as a stronger variant of Lemma 7.8.

Lemma 7.23. *There is a fair computation $\sigma = c_0 \to c_1 \to \ldots$ of the broadcast network if and only if there is a computation of the form $c_0 \to^* c \Rightarrow_{Q_F} c$.*

Proof. If there is a computation of the form $c_0 \to^* c \Rightarrow_{Q_F} c$, then the good cycle $c \Rightarrow_{Q_F} c$ can be iterated infinitely often to obtain a fair computation.

For the other direction, let a fair computation $\sigma = c_0 \to c_1 \to \ldots$ of the network be given. Assume the configurations visited by σ are in Q^k for some integer $k \in \mathbb{N}$. Then, there is a configuration $c \in Q^k$ that repeats infinitely often in σ. Let $c_0 \to^* c$ be the prefix of the computation σ leading to c.

Let $Inf(\sigma) = \{i_1, \ldots, i_n\}$ be the set of clients that participate infinitely often in σ. By the definition of a fair computation, we have $Inf(\sigma) \subseteq Fin(\sigma)$. Phrased differently, this means that each of the clients in $\{i_1, \ldots, i_n\}$ visits a state from Q_F infinitely often along σ. Hence, for each $j \in [1..n]$ we can find a subcomputation $\sigma_j = c \to^+ c$ of σ, in which client i_j visits a state from Q_F. Concatenating all σ_j yields the desired good cycle $c \Rightarrow_{Q_F} c$. □

Instrumentation Good cycles characterize fair computations. Hence, for solving FBNL we need an algorithm detecting the former. Our algorithm for this task takes the broadcast network N and constructs an instrumentation N_{Q_F} which ensures that (plain) cycles over N_{Q_F} correspond to good cycles over N. Cycles in N_{Q_F} can be found by the fixed-point iteration for BNL.

The idea behind the construction of N_{Q_F} is to let the clients compute in three phases during a cycle. For each of these phases, there is a copy of the state space Q. We denote these by Q, $\hat{Q}$, and $\tilde{Q}$. In a cycle, the initial phase is over the states of Q. As soon as a client participates, it has to move to the second phase $\hat{Q}$. From there, it can only go to the third phase $\tilde{Q}$ upon seeing a final state. Finally, the client can return to Q. Hence, the instrumentation detects whether the clients that participate in a cycle also see a final state during their participation. We provide a formal definition.

Definition 7.24. The *instrumentation of* N is defined to be the broadcast network $N_{Q_F} = (P', D')$ with domain $D' = D \cup \{n\}$, where $n \notin D$ is a fresh

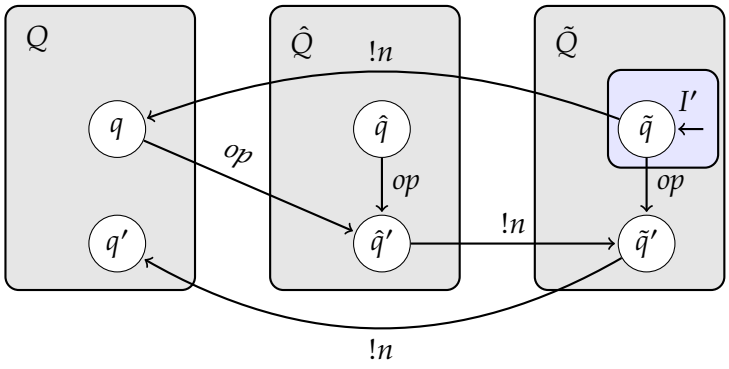


Figure 7.5: Structure of the instrumentation N_{Q_F} of the underlying broadcast network N. Let $I = \{q\}$, then $I' = \{\tilde{q}\}$. Assume that N has a transition $q \xrightarrow{op} q'$. Then we get the above transitions within $\tilde{Q}$, $\hat{Q}$ and from Q to $\hat{Q}$. Moreover, let $q' \in Q_F$. Then we obtain the transition $\hat{q}' \xrightarrow{!n} \tilde{q}'$. The remaining transitions lead from each state of $\tilde{Q}$ via $!n$ to the corresponding copy in Q.

symbol, and client $P' = (Q', OP(D'), \delta', I')$ is defined as follows. The states Q' of the client are given by three copies of Q that we refer to as *phases*:

$$Q' = Q \cup \hat{Q} \cup \tilde{Q}.$$

To define initial states and transition relation, we use a notation that distinguishes states of Q' by their phase. We write q, $\hat{q}$, or $\tilde{q}$ depending on the phase in which the considered state lies. The initial states are then given by $I' = \{\tilde{q} \mid q \in I\}$. The transition relation δ' is given as follows. For every transition $q \xrightarrow{op} q'$ in δ, we have the three transitions

$$q \xrightarrow{op} \hat{q}', \;\; \hat{q} \xrightarrow{op} \hat{q}', \;\; \tilde{q} \xrightarrow{op} \tilde{q}' \in \delta'.$$

Moreover, for each final state $q \in Q_F$, a client can pass from $\hat{Q}$ to $\tilde{Q}$ via $\hat{q} \xrightarrow{!n} \tilde{q} \in \delta'$. Finally, for every state $q \in Q$, we have $\tilde{q} \xrightarrow{!n} q \in \delta'$.

We illustrate the structure of the instrumentation in Figure 7.5. Note that within the states of Q, there are no internal transitions. We are interested in cycles leading from Q to Q. To leave Q, a client has to pass through $\hat{Q}$ and can only arrive at $\tilde{Q}$ upon seeing a final state. Then, it can go back to Q. But this means that each client participating in such a cycle sees a final state on its way. Hence, cycles from Q to Q have to be good. The observation is crucial for the following lemma. It characterizes computations of N with good cycles in terms of computations of N_{Q_F} involving (plain) cycles.

Lemma 7.25. *There is a computation of the form $c_0 \to^* c \Rightarrow_{Q_F} c$ in N if and only if there is a computation of the form $\tilde{c_0} \to^* c \to^+ c$ in the instrumentation N_{Q_F}.*

The configuration $\tilde{c_0}$ in the lemma is the copy of c_0 over the states of $\tilde{Q}$. This is where the computation has to start. Note that an initial prefix of N can always be mimicked by a computation within $\tilde{Q}$. The configuration c of N_{Q_F} has only states in Q. Hence, $c \to^+ c$ is indeed a cycle from Q to Q.

Proof. We elaborate on the idea, a formal proof is given in Appendix B.2.2. The reasoning for prefixes is simple. By adding $!n$-transitions to the end of a prefix $c_0 \to^* c$ of N, we can turn it into a prefix $\tilde{c_0} \to^* c$ of N_{Q_F}. Vice versa, a prefix $\tilde{c_0} \to^* c$ of N_{Q_F} has to end with some $!n$-transitions. By removing these, we can turn the prefix into a prefix $c_0 \to^* c$ of the original network N.

Now let a cycle $c \to^+ c$ in N_{Q_F} be given. Then, all clients are in a state from phase Q. As soon as a client participates in the cycle, it will immediately move from Q to $\hat{Q}$. To return to Q, the client has to go through $\tilde{Q}$ since there is no direct edge leading from $\hat{Q}$ back to Q. But by the definition of the instrumentation N_{Q_F}, the client can only transition to $\tilde{Q}$ when it sees a state from Q_F. This means that each participating client in the cycle sees a final state on the way, resulting in a good cycle $c \Rightarrow_{Q_F} c$ of N.

For the other direction, let $c \Rightarrow_{Q_F} c$ be a good cycle of N. If a client participates in it, we can simulate its computation on N_{Q_F}. Assume it starts in the state $q \in Q$. Upon its first transition, it moves from Q to $\hat{Q}$ in N_{Q_F}. The client stays within the states of $\hat{Q}$ and continues its computation until it sees a state from Q_F. Note that the assumed cycle is good, hence the client will definitely see such a state at some point. After visiting Q_F, the client moves to $\tilde{Q}$ via taking the corresponding $!n$-transition. It continues its computation in the phase $\tilde{Q}$ until it reaches the state $\tilde{q}$. Then, it moves to q by taking an $!n$-transition. Altogether, this constitutes a cycle $c \to^+ c$ in N_{Q_F}. □

Algorithm For solving FBNL, it is left to argue that computations of the form $\tilde{c_0} \to^* c \to^+ c$ in the instrumentation can be found with the fixed-point iteration from Section 7.2.3. To this end, we use a similar trick as in the detection of cycles over N. Note that there is a cycle from phase Q to phase Q in N_{Q_F} if and only if there is such a cycle involving all states that are reachable in Q. The clients of those states that are not needed during the cycle will not participate and just preserve their current state.

This allows for employing the fixed-point iteration as in Algorithm 7.1, with only slight variations. First, we compute the instrumentation N_{Q_F} and the states of N_{Q_F} that are reachable from I'. Again we employ the algorithm from [123] for this task. Then we intersect these states with the phase Q to obtain all reachable states within Q. Let the intersection be $\{s_1, \ldots, s_k\}$. Finally, we apply the fixed-point iteration from Lemma 7.17 to $\{s_1, \ldots, s_k\}$. We summarized the approach in Algorithm 7.2.

The correctness of Algorithm 7.2 follows from Lemmas 7.17, 7.11, and 7.25. Recall that the fixed-point iteration witnesses the existence of a cycle over a configuration c of N_{Q_F} with $Set(c) = \{s_1, \ldots, s_k\}$. Since all these states are

Algorithm 7.2 Fair Liveness Verification

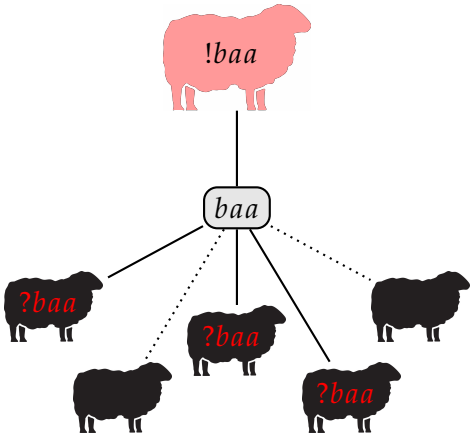
Input: A broadcast network $N = (D, P)$ and a set of final states $Q_F \subseteq Q$.
Output: *True*, if there is an initialized σ with $Fin(\sigma) \neq \emptyset$. *False* otherwise.

compute N_{Q_F}, the instrumentation of N
use algorithm of [123] to compute R, the set of reachable states of N_{Q_F}
compute $\{s_1, \ldots, s_k\} = R \cap Q$ // Reachable states in phase Q.
set $S = (\{s_1\}, \ldots, \{s_k\})$, $C = Q^k$, and $C' = \emptyset$.
while $C \neq C'$ **do** // Computation of greatest fixed point.
  set $C' = C$
  compute $C = post^+_{C'}(S) \cap pre^*_{C'}(S)$ with algorithm from Lemma 7.16
end while
if $C = \emptyset$ **then**
  return *false* // Exclude trivial solution.
end if
return *true*

reachable in Q, there is a corresponding prefix $\tilde{c_0} \rightarrow^* c$. Altogether, we have a computation $\tilde{c_0} \rightarrow^* c \rightarrow^+ c$ entailing a computation of N with a good cycle. There is a subtlety: the set of reachable states $\{s_1, \ldots, s_k\}$ in Q can be reached by a corresponding prefix. In fact, a state $\tilde{q} \in \tilde{Q}$ can always reach its corresponding copy $q \in Q$ simply by sending $!n$. This means we perform the corresponding prefix in $\tilde{Q}$ and then, at the end, move to the copies in Q.

The dominant time factor of Algorithm 7.2 is again caused by the fixed-point iteration. To formulate it, note that the size of the instrumentation N_{Q_F} is linear in the size of N. Indeed, we have $O(p)$ many states and $O(t + p)$ many transitions. We may assume that the given broadcast network N has no isolated states since these can be removed. In this case, we obtain that $p \in O(t)$ and $O(t + p) = O(t)$. According to Lemma 7.17, the algorithm then runs in time $O(p^4 \cdot t^2)$. This completes the proof of Theorem 7.20.

8. Safety and Liveness in Leader Contributor Systems



Leader contributor systems are a parameterized model consisting of a designated leader process and an arbitrary number of identical contributors. Communication is performed via a shared memory. Leader contributor systems are employed for the verification of client-server applications and protocols on wireless sensor networks — essentially concurrent programs that rely on a master/slave architecture and that have an arbitrary number of slave threads. Safety verification in a leader contributor system is typically formulated as the so-called *leader contributor reachability problem* (LCR). The problem asks whether the leader process can reach an unsafe state when interacting with a certain number of contributors. In the finite-state case, LCR is known to be NP-complete [152]. Liveness verification is modeled by the *leader contributor liveness problem* (LCL). The generalization of LCR asks whether the leader can reach a particular set of states infinitely often. Similar to its reachability counterpart, the problem is known to be NP-complete [143]. Since these results have been obtained, no further algorithmic progress for both problems has been made. This may include a more detailed picture of the complexity or optimal verification algorithms.

In order to provide both, we conducted a fine-grained complexity analysis of LCR and LCL. For LCR, the analysis identifies two tractable parameterizations. We show that each admits a provably optimal safety verification algorithm. Moreover, we present lower bounds on the corresponding kernels and for identifying intractable parameterizations. The same parameterizations can be used for LCL. Lower bounds from LCR naturally carry over to LCL. Surprisingly, our results show that the upper bounds also carry over, resulting in two optimal liveness verification algorithms. This means that from a fine-grained point of view, the complexities of LCR and LCL differ only by a polynomial factor, although LCL seems to be harder. The upcoming text is an extended version of the original presentation of the results in [93, 94, 96].

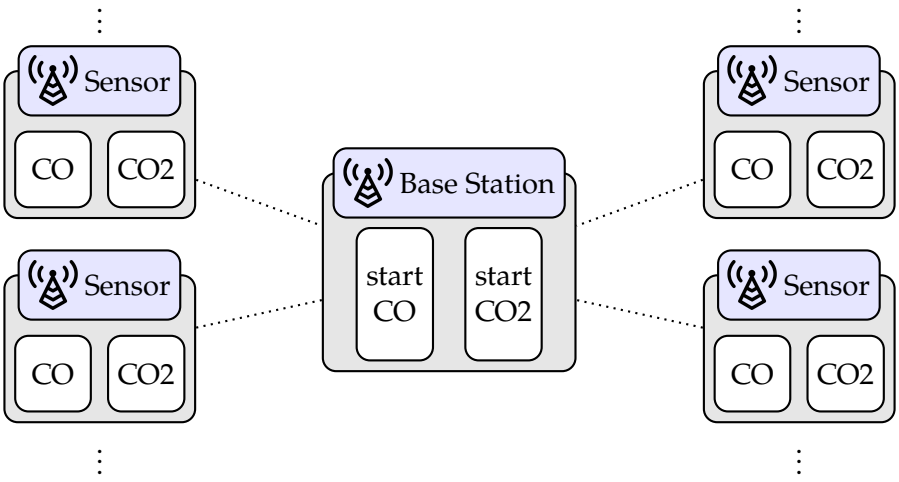


Figure 8.1: Simple wireless sensor network for measuring the CO and the CO2 concentration in the air. The base station initiates measurements, the sensors perform them and send the received data back to the base station.

8.1 Verification of Leader Contributor Systems

A parameterized concurrent program combines the challenge of concurrent reasoning with that of having an arbitrarily large number of threads. Problems appearing in the verification of such programs can be undecidable [17, 311]. In the case of a shared-memory communication, decidability can often be recovered [21, 28, 103, 122, 152, 198, 227, 245]. A particularly successful model for the verification of parameterized concurrent programs that communicate via a shared memory are so-called leader contributor systems [93, 94, 96, 143, 152, 174, 198, 246]. These assume to have a distinguished leader thread that interacts with an arbitrary but finite number of identical contributor threads. Leader contributor systems first appeared in the work of German and Sistla [182] and later in the work of Hague [198].

The popularity of these systems is due to two aspects. From a modeling perspective, a variety of systems can be formulated as anonymous entities interacting with a central authority. This includes client-server applications, resource management systems, and in particular protocols on wireless sensor networks (WSN) [228]. From an algorithmic point of view, the model has been shown to be tractable. Safety verification is PSPACE-complete even when the components of the system are pushdown automata [152, 198], an assumption that usually leads to undecidability [202]. In fact, decidability of safety verification is preserved even when the components are rather powerful: for instance for Petri nets and lossy channel systems [246]. In the case of finite-state components, safety verification is only NP-complete [152]. Surpris-

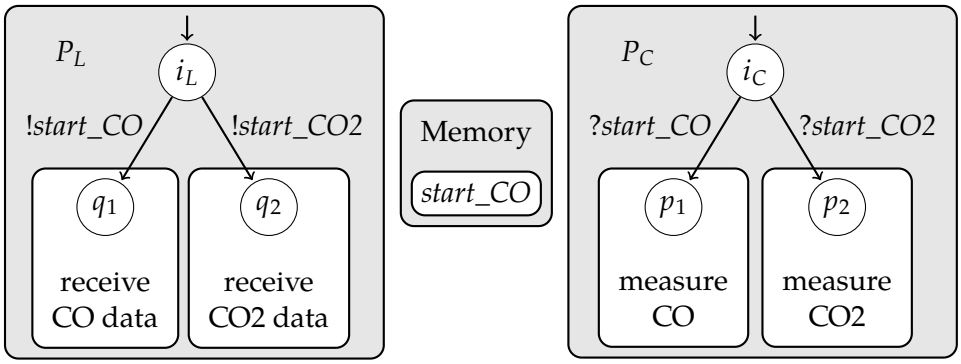


Figure 8.2: Leader contributor system consisting of a leader thread P_L, a memory cell, an a contributor thread P_C. The system models a mutual exclusion among the subroutines of P_C. Either all contributors enter the subroutine *measure CO* or all enter the subroutine *measure CO2*.

ingly, the complexity remains the same for liveness verification. The problem is NP-complete [143] for finite-state components and PSPACE-complete for pushdown automata [143, 174]. Checking (some) LTL-definable properties for the model turned out to be NEXP-complete [174].

We illustrate the idea of leader contributor systems with an example. Assume we are given a wireless sensor network which measures the air pollution. The network consists of a base station and a large amount of sensors at different locations. We illustrate its structure in Figure 8.1. The sensors can either measure the current carbon monoxide (CO) or carbon dioxide concentration (CO2) in the air. The base station accumulates and steers the measurements. For receiving coherent data, the sensors must agree on which measurement to perform next: CO or CO2. Phrased differently, no sensors should measure the CO concentration while others currently measure the CO2 concentration. This can be realized by a mutual exclusion protocol: either all sensors in the network enter their subroutine for measuring CO or all sensors enter the subroutine for measuring CO2.

With leader contributor systems, we can model the wireless sensor network and verify the mutual exclusion. We show the corresponding system in Figure 8.2. The base station is represented by the leader thread P_L — it is the only distinguishable component of the network. Sensors are modeled as contributors threads P_C. There is an arbitrarily large number of sensors and they all run the same procedure. The wireless communication is modeled in terms of a shared memory. Leader and contributors may write $!a$ to or read $?a$ from the memory. Since wireless communication may be unstable and sensors can be mobile, messages in the system may get lost. We assume a reconfigurable communication topology, like for broadcast networks.

The mutual exclusion protocol works as follows. The leader P_L either

writes *start_CO* or *start_CO2* to the memory, depending on which measurement the thread wants to initiate. A contributor can read the value and branch into the corresponding subroutine. The following example computation with two contributors initiates a measurement of the CO concentration. Note that one of the two contributors actually starts the measurement, the other one did not receive the message due to lossy communication:

$$(i_L, i_C, i_C) \xrightarrow{!start\_CO} (q_1, i_C, i_C) \xrightarrow{?start\_CO} (q_1, p_1, i_C).$$

Like in the computation, it may be the case that contributors do not start their measurement. But it cannot happen that some contributor decides to measure CO2 instead of CO. Hence, the simple communication pattern already guarantees mutually exclusion among the subroutines of the contributor P_C. Although the example is simple, it reveals a rather powerful mechanism of leader contributor systems. In fact, we will later use this kind of mutual exclusion to encode SAT into the model.

Like in the example, we focus on leader contributor systems with finite-state components. Our goal is to understand the complexity of safety and liveness verification in more detail and go beyond NP-completeness. This includes fine-grained upper bounds as well as matching lower bounds. In the following, we state our contribution to both problems.

8.1.1 Safety Verification

Safety verification for leader contributor systems is typically phrased in terms of a reachability problem for the leader thread. The *leader contributor reachability problem* (LCR) asks whether the leader can reach a final state in the presence of a certain number of contributors. The complexity of the problem was first considered by Hague [198] and later by Esparza, Ganty, and Majumdar [152]. The studies pinpoint the complexity of LCR in various cases, including the PSPACE-completeness for pushdown components and the NP-completeness for finite-state components. We are interested in the latter.

Contribution We contribute a fine-grained complexity analysis of LCR in the finite-state case. It is performed in three canonical parameters: the size d of the data domain of the shared memory, the size of the leader thread l, and the size of the contributor thread c. The size of these parameters depends on the modeled application. Intuitively, in a wireless sensor network, the sensors are comparatively simple. Hence, c tends to be small in this case. We would then prefer an FPT-algorithm which depends solely on c in the exponential part of its running time. The complexity analysis provides exactly this: we show that the parameterization LCR(c) is fixed-parameter tractable. If c tends to be large when modeling other applications, we suggest a parameterization in d and l: in fact, LCR(d, l) is FPT as well. Note that both parameters are

Problem	Upper Bound	Lower Bounds	
LCR(d, l)	$(d+l)^{O(d+l)}$	$2^{o((d+l)\cdot\log(d+l))}$	No polynomial kernel
LCR(c)	$O^*(2^c)$	$O^*((2-\varepsilon)^c)$ for $\varepsilon > 0$	No polynomial kernel
LCR(d)	-	W[1]-hard	
LCR(l)	-	W[1]-hard	

Table 8.1: Fine-grained complexity of LCR. Main results are the algorithms for LCR(d, l) and LCR(c) as well as the lower bound for the former.

required in this case. Parameterizing by d or l alone leads to W[1]-hardness. We summarize the results of our fine-grained analysis in Table 8.1.

Our analysis does not only yield tractable parameterizations of LCR, we also give provably optimal verification algorithms. One of our main contributions is the corresponding algorithm for LCR(d, l) which runs in time $(d+l)^{O(d+l)}$. The algorithm bases on the notion of a *witness*, a sketch of a computation of the underlying leader contributor system. A witness is valid if there is an actual computation corresponding to it. Hence, instead of searching for a computation, we can seek for a valid witness. The first algorithmic idea one would probably have, is to iterate over all witnesses and perform a validity check on each. Although this leads to membership in FPT [93], the resulting algorithm is rather slow and can be accelerated significantly. Instead of iterating over all witnesses, we show that valid witnesses can be build up inductively from small witnesses. To this end, we interleave validity checks with compression phases. Our algorithm is then a dynamic programming on small witnesses, exploiting the inductive approach.

The algorithm is complemented by a matching lower bound. We set up a reduction from $k \times k$-Clique to LCR which yields that the latter cannot be solved in time $2^{o((d+l)\cdot\log(d+l))}$ unless the ETH fails. The reduction is technically challenging since we have to keep the size of the data domain d linear. This restricts the amount of information that can be exchanged among leader and contributors and requires a communication pattern to transfer more complex messages. The idea is related to reductions known from (weak) memory testing [77, 98, 180, 183]. We also provide a second lower bound for LCR(d, l): the problem does not admit a polynomial kernel.

When considering c as a parameter, we obtain an algorithm running in time $O^*(2^c)$. To this end, we first characterize computations of the leader contributor systems in terms of paths on a compressed graph. The characterization relies on a saturation argument which is inspired by thread-modular reasoning [161, 162, 193, 211]. The problem LCR can then be phrased as a reachability problem on the graph. Constructing the graph immediately and

applying a path finding algorithm however, results in an $O^*(4^c)$-time algorithm [93]. To obtain the desired running time, we show that the reachability problem on the graph can be solved without constructing the whole graph at once. In fact, we can decompose the graph into so-called *slices* and for each solve a smaller reachability problem in polynomial time. By employing dynamic programming, we can then accumulate the information acquired for the slices to decide reachability on the complete graph.

The algorithm is optimal in the fine-grained sense. We provide a lower bound which prohibits an $O^*((2-\varepsilon)^c)$-time algorithm for any $\varepsilon > 0$. The lower bound relies on the SCON and a corresponding reduction from the problem Set Cover to LCR. Note that intuitively, also LCR fits into the category of problems admitting a lower bound via the SCON. Indeed, LCR requires a non-trivial dynamic programming algorithm to be solved efficiently and the problem seems to be less related to SAT. Finally, we prove that the parameterization LCR(c) does not admit a polynomial kernel.

8.1.2 Liveness Verification

Like for safety verification, liveness for leader contributor systems is phrased in terms of the leader thread. The most common decision problem is the *leader contributor liveness problem* (LCL). It asks whether the leader can reach a set of final states repeatedly and hence, infinitely often. The problem was first studied by Durand-Gasselin, Esparza, Ganty, and Majumdar in [143] and has been shown to admit the same complexity as reachability: LCL is NP-complete for finite-state components and PSPACE-complete for pushdowns. We consider the former complexity result in more detail.

Contribution In order to obtain a more detailed picture of its complexity, we conducted a fine-grained analysis of LCL in the case of finite-state components. To this end, we considered the same parameterizations as for the safety verification problem LCR. The results are given in Table 8.2. Our analysis shows that, even from a fine-grained point of view, the complexities of LCR and LCL differ only by a polynomial factor. Indeed, when parameterized by d and l, we obtain an $(d+l)^{O(d+l)}$-time algorithm for LCL. When parameterized by c, we get an algorithm running in time $O^*(2^c)$.

We explain the results in more detail. The first step to derive algorithms for LCL is to decompose a live computation of the leader contributor system into a prefix and a cycle. This decomposition is guided along so-called *interfaces* which match corresponding prefixes and cycles. With the notion of interfaces, we can then split LCL into two separate problems: finding a prefix and finding a cycle. Finding prefixes is a matter of reachability and algorithms for LCR can be applied. We show how to combine these algorithms with a cycle detection to obtain algorithms that find live computations. The latter will run in time

Problem	Upper Bound	Lower Bounds
LCL(d, l)	$(d+l)^{O(d+l)}$	$2^{o((d+l)\cdot\log(d+l))}$
LCL(c)	$O^*(2^c)$	$O^*((2-\varepsilon)^c)$ for $\varepsilon > 0$
CYC	$O(d^2 \cdot (c^2 + l^2 \cdot d^2))$	-

Table 8.2: Results of the fine-grained analysis of the liveness verification problem LCL. The main result is the polynomial-time algorithm for CYC which entails the FPT-algorithms for both parameterizations of LCL.

$O(Reach(d, l, c) \cdot Cycle(d, l, c))$ where $Reach(d, l, c)$ denotes the running time of the invoked reachability algorithm and $Cycle(d, l, c)$ that of the cycle detection.

We denote the problem of finding cycles by CYC. Our main contribution is an algorithm for CYC which runs in polynomial time $O(d^2 \cdot (c^2 + l^2 \cdot d^2))$. This shows that $Cycle(d, l, c)$ is only a polynomial factor and consequently, liveness has the same complexity as reachability — up to a polynomial factor. Phrased differently, the running times of the algorithms for LCR carry over to LCL which results in the two FPT-algorithms stated in Table 8.2. Technically, the algorithm for CYC relies on a characterization of cycles via (certain) SCC decompositions of the contributor. To find these decompositions in polynomial time, we employ a fixed-point iteration which repeatedly invokes Tarjan's algorithm [313] for finding general SCC decompositions.

Note that lower bounds from LCR carry over to LCL immediately. Hence, our analysis provides two provably optimal verification algorithms for LCL.

8.2 Safety Verification Algorithms

We present the FPT-algorithms for the safety verification problem LCR. To this end, we begin by formally introducing leader contributor systems and the problem LCR in Section 8.2.1. Then, the presentation is split into two parts according to the two tractable parameterizations of the problem. In Section 8.2.2, we focus on the parameterization LCR(d, l) and present the corresponding $(d+l)^{O(d+l)}$-time algorithm. Consequently, in Section 8.2.3, we parameterize by the size of the contributor c and develop the $O^*(2^c)$-time algorithm for LCR. Lower bounds are postponed to Section 8.3.

8.2.1 Leader Contributor Systems and LCR

A *leader contributor system* is a parameterized system consisting of a designated leader thread and an arbitrary but finite number of identical contributor threads. Communication is handled via a shared memory cell which can

hold one value at a time from some underlying data domain. We formalize these systems and introduce the corresponding reachability problem LCR.

Shared Memory and Operations Let D be some fixed data domain. The leader and the contributors interact with a memory cell holding a value of D. In fact, both kinds of threads can write $!a$ a value $a \in D$ to the memory or read $?a$ a value $a \in D$ from the memory. We formally model this interaction by the alphabet of *operations* over the data domain D:

$$OP(D) = \{!a, ?a \mid a \in D\}.$$

To prevent the memory cell from being empty, it holds a particular value $a^0 \in D$ in the beginning of a computation. We refer to a^0 as the *initial value*.

Leader and Contributor The leader thread is an abstraction of a master thread which focuses on making visible the interaction with the memory. To this end, it is defined in terms of a finite automaton over the operations of D.

Definition 8.1. Let D be some finite data domain. The *leader (thread)* is a (nondeterministic) finite automaton $P_L = (Q_L, OP(D), \delta_L, q_L^0)$.

Note that the leader comes without final states. We incorporate these later as an input to the reachability problem. The semantics of P_L is that of a usual finite automaton, defined in terms of its transition relation δ_L. We extend it to words of operations $w \in OP(D)^*$ and denote it by $q \xrightarrow{w}_L q'$ if $(q, w, q') \in \delta_L$.

Contributors model the interaction of identical slave threads with the memory. We employ a definition, similar to that of the leader.

Definition 8.2. Let D be some finite data domain. A *contributor (thread)* is a (nondeterministic) finite automaton $P_C = (Q_C, OP(D), \delta_C, p_C^0)$.

The semantics of a single contributor is as expected. Like for the leader, we write $q \xrightarrow{w}_C q'$ if $(q, w, q') \in \delta_C$ and $w \in OP(D)^*$. However, contributors are the parameterized part of a leader contributor systems. This means they appear in arbitrary but finite numbers and cannot be distinguished. The impact that this has on a computation is discussed in a minute when we formalize the semantics of a leader contributor system.

Leader Contributor Systems We combine the above ingredients in the definition of a *leader contributor system*. The syntax is as follows.

Definition 8.3. A *leader contributor system* (LCS) is defined to be a tuple of the form $S = (D, a^0, P_L, P_C)$ where D is a finite data domain, $a^0 \in D$ is an initial memory value, P_L is the leader thread, and P_C is a contributor.

Computations of a leader contributor system are defined in terms of sequences of *configurations*. A configuration carries all information that leader or contributors require in order to interact with their environment. An interaction may depend on the current memory value and on the internal state of the considered thread. To track this, a configuration contains the current states of all contributors and the leader as well as the current memory value. Since there might be an arbitrary number of contributors involved in a computation, the number of configurations cannot be finite.

Definition 8.4. Fix a leader contributor system $S = (D, a^0, P_L, P_C)$. A *configuration of* S is a tuple $(q, a, pc) \in Conf(S)^t = Q_L \times D \times Q_C^t$. It consists of the current state q of the leader, the current memory value a, and the *program counter* pc, holding the current value of each of the $t \in \mathbb{N}$ participating contributors. The *set of configurations of* S is given by

$$Conf(S) = \bigcup_{t \in \mathbb{N}} Conf(S)^t.$$

A configuration is called *initial* if it is of the form $c^0 = (q_L^0, a^0, pc^0)$, where $pc^0(i) = q_C^0$ is the initial state of the contributor for each component i.

It will be handy to access the different components of a configuration. We use several projections for this task. Let π_L and π_D denote the maps that project a configuration to the state of the leader, respectively the memory content. Phrased differently: $\pi_L(q, a, pc) = q$ and $\pi_D(q, a, pc) = a$. Moreover, let π_C be the map that projects a configuration to the set of contributor states that are present in the vector pc. We have $\pi_C(q, a, pc) = \{pc(i) \mid i \in [1..t]\}$ where t is the number of currently involved contributors.

The current configuration of a leader contributor system may change over time due to interaction with the memory or an internal transition of the leader or one of the involved contributors. These changes are captured as usual — by a transition relation on configurations. We formalize below.

Definition 8.5. Let $S = (D, a^0, P_L, P_C)$ be a leader contributor system. We define the *transition relation of* S to be a relation among configurations:

$$\rightarrow_S \subseteq Conf(S) \times (OP(D) \cup \varepsilon) \times Conf(S).$$

The relation contains transitions that are either induced by the leader or by one of the contributors. We focus on the former. If there is a write transition of P_L, then it induces a corresponding transition of $\rightarrow_S$:

$$q \xrightarrow{!b}_L q' \text{ implies } (q, a, pc) \xrightarrow{!b}_S (q', b, pc).$$

Note that the leader changes its state to q' and the memory changes its value to b. The states pc of the contributors are not affected by the transition.

If the leader has a read transition, it carries over to a transition of $\rightarrow_S$ if the read value is currently available in the memory. We have:

$$q \xrightarrow{?a}_L q' \text{ implies } (q, a, pc) \xrightarrow{?a}_S (q', a, pc).$$

An internal transition of P_L induces a transition of $\rightarrow_S$ independent of the current memory value. It only changes the current state of the leader:

$$q \xrightarrow{\varepsilon}_L q' \text{ implies } (q, a, pc) \xrightarrow{\varepsilon}_S (q', a, pc).$$

Transitions induced by the contributors are defined similarly. Let pc be a program counter with $pc(i) = p$ and $pc' = pc[i = p']$. Recall that pc and pc' coincide except for the i-th component, where $pc'(i) = p'$. We have:

$$p \xrightarrow{!b/?a/\varepsilon}_C p' \text{ implies } (q, a, pc) \xrightarrow{!b/?a/\varepsilon}_S (q, b/a, pc')$$

like for the leader. Note that the transitions do not change the state of P_L.

To define computations, we generalize the transition relation $\rightarrow_S$ to words over $OP(D)$ in the obvious way. We denote the generalization by $c \xrightarrow{w}_S c'$ where $w \in OP(D)^*$ and call it a *computation of* S. Sometimes we also write $c \rightarrow_S^* c'$ if a word w with $c \xrightarrow{w}_S c'$ exists, and if $|w| \geq 1$ we write $c \rightarrow_S^+ c'$. A computation $\sigma = c^0 \rightarrow_S c^1 \rightarrow_S \cdots \rightarrow_S c^k$ of the leader contributor system is called *initialized* if c^0 is an initial configuration.

Note that $\rightarrow_S$ is only defined among configurations involving the same number of contributors. This means that contributor threads cannot be dynamically created once a computation has begun. Moreover, it is convenient to assume that, during a computation, the leader thread P_L never writes $!a$ and immediately reads $?a$ the same value a again. This can be achieved via a slight modification of the leader. Assume we have two transitions $\sigma_1 = q \xrightarrow{!a}_L q'$ and $\sigma_2 = q' \xrightarrow{?a}_L \hat{q}$. Since P_L can provide the requested a when it passes from q to $\hat{q}$ via $\sigma_1.\sigma_2$, we can add the transition $\sigma_3 = q \xrightarrow{!a}_L \hat{q}$. When the leader now takes $\sigma_1.\sigma_2$ during a computation, we can assume it to take the transition σ_3 instead without changing what is visible to the contributors on the memory.

Before we formalize LCR, the reachability problem of interest, we illustrate computations of leader contributor systems with an example.

Example 8.6. We reconsider the leader contributor system from Section 8.1 which models a mutual exclusion. We extend the system so that it checks whether the mutual exclusion works as expected. The corresponding system $S = (D, a^0, P_L, P_C)$ is given in Figure 8.3. Consider the following computation involving two contributors. It chooses the upper branch of the system:

$$(q^0, a_0, p^0, p^0) \xrightarrow{!a}_S (q^1, a, p^0, p^0) \xrightarrow{?a}_S (q^1, a, p^1, p^0) \xrightarrow{?a}_S (q^1, a, p^1, p^1).$$

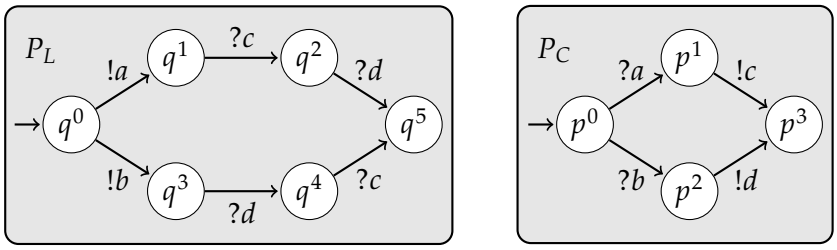


Figure 8.3: Leader contributor system $S = (D, a^0, P_L, P_C)$ modeling a mutual exclusion with sanity check. The domain is $D = \{a^0, a, b, c, d\}$. The state q^5 of P_L cannot be reached, no matter how many contributors are active.

As we can see in the computation, the leader has the power of choosing which branch to follow. To this end, it either writes $!a$ or $!b$. With the operation, the leader forces all active contributors to read the corresponding symbol a or b. Hence, all P_C either go to state p^1 or all go to state p^2. The contributors can confirm arriving the states by writing c or d. Since both states can never be arrived at the same time, only one message c or d can be read by the leader. But this means that state q^5 is not reachable in P_L, independent of the number of contributors. The mutual exclusion works as expected.

In general, reaching a particular state may well depend on the number of contributors. Consider the leader contributor system $T = (D, a^0, P_L, P_C)$ depicted in Figure 8.4. We consider a computation which lets the leader P_L arrive at the state q^4. The computation involves two contributors:

$$\begin{aligned}(q^0, a^0, p^0, p^0) &\xrightarrow{!a}_T (q^0, a, p^1, p^0) \xrightarrow{?a}_T (q^1, a, p^1, p^0) \xrightarrow{!b}_T (q^2, b, p^1, p^0) \\ &\xrightarrow{?b}_T (q^2, b, p^1, p^2) \xrightarrow{!c}_T (q^2, c, p^1, p^2) \xrightarrow{?c}_T (q^3, c, p^1, p^2) \\ &\xrightarrow{!a}_T (q^3, a, p^1, p^2) \xrightarrow{?a}_T (q^4, a, p^1, p^2).\end{aligned}$$

In fact, to reach q^4, we need at least two contributors that participate. Having only one contributor does not suffice: it cannot provide a and c when the leader needs to read it from the memory. Hence, a larger number of contributors may enable reachability of a particular state.

Leader Contributor Reachability In the example we have seen that a large number of contributors may lead to reachability of an unsafe state. When modeling an actual parameterized application, this means that once the number of slave threads is too large, the system does not satisfy its safety requirements anymore. This is not what we expect to be a correct system. Instead, safety requirements should hold, independent of the number of involved threads. This means that we need to verify leader contributor systems for

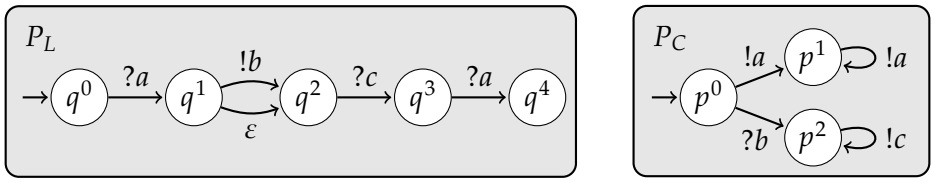


Figure 8.4: Leader contributor system $T = (D, a^0, P_L, P_C)$ over the domain $D = \{a^0, a, b, c\}$. Two contributors are required to reach the state q^4 in P_L.

each potentially occurring number of contributors. To this end, the *leader contributor reachability problem* (LCR) does not put a bound on their number. The problem asks whether in a given leader contributor system, a final state of the leader can be reached when interacting with some number of contributors.

LCR

Input: An LCS $S = (D, a^0, P_L, P_C)$ and final states $Q_F \subseteq Q_L$.
Question: Is there an initialized $c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$?

As mentioned above, the problem LCR with finite-state components, as it is stated here, is NP-complete [143]. For our fine-grained analysis, we consider three parameters. The size d of the underlying data domain D, the number of states l of the leader P_L and the number of states c of the contributor P_C.

8.2.2 Parameterization by Domain and Leader

The first step to our fine-grained analysis is an algorithm for the parameterization LCR(d, l). Note that both parameters are required since dropping one already results in intractability, see Section 8.3.3. Our algorithm runs in time $(d + l)^{O(d+l)}$ and shows that the parameterization is FPT. Moreover, the algorithm is optimal in the fine-grained sense. We present the corresponding lower bound in Section 8.3.1. Formally, we prove the following theorem.

Theorem 8.7. *The problem LCR can be solved in time* $(d + l)^{O(d+l)}$.

Our algorithm relies on a notion of *witnesses*. These are sketches of computations that can witness reachability of a final state. A witness is *valid* if it represents an actual computation. Hence, LCR can be solved by finding a valid witness leading to a final state. To obtain an FPT algorithm, one can therefore iterate over all witnesses and test each for being valid[11]. The latter takes polynomial time. However, this results in an algorithm running in time

[11] The number of witnesses, as they are defined here, is infinite. In [93] we used a slightly different variant of witnesses which only yields a finite number.

proportional to $(d \cdot l)^{O(d \cdot l)}$, the number of witnesses [93]. The key to improve the upper bound to $(d+l)^{O(d+l)}$ is the fact that we can restrict to so-called *short witnesses*. Intuitively, these are sketches of loop-free computations. We show that validity of witnesses can be checked inductively from validity of short witnesses. The inductive approach is then used within a dynamic programming which admits the desired running time and proves Theorem 8.7.

Witnesses A witness is a compact way of representing a computation of a leader contributor system. From a computation, a witness only stores the operations performed by the leader and the positions where memory symbols were written by a contributor for the first time. These are called *first-write positions*. They are crucial since from such a position on, we can assume an unbounded supply of the corresponding symbol. We formalize:

Definition 8.8. Let $S = (D, a^0, P_L, P_C)$ be a leader contributor system, ρ some computation of S, and $a \in D$ an arbitrary data value. Let $c \xrightarrow{!a}_S c'$ be the transition (if it exists) of ρ where a gets written for the first time by a contributor. Then we may call the transition *the first write (of a)*.

Once a first write of a symbol a has been performed, we can assume that an arbitrary number of contributors is waiting to provide a whenever we need it. The reasoning is as follows. Assume the first write is carried out by contributor P_C^1 which takes the transition $p \xrightarrow{!a}_C p'$. Then we can assume that there is a contributor P_C^2 that mimicked the behavior of P_C^1 during the computation. This means whenever P_C^1 did a transition, P_C^2 copycatted it right away. Hence, P_C^2 also arrives in state p and is ready to provide a. The argument similarly works for arbitrarily many copies of P_C^1, providing a along a computation whenever it is required. The idea goes back to the *copycat lemma* stated in [152]. We illustrate the notion of a first write via an example.

Example 8.9. Recall the leader contributor system $S = (D, a^0, P_L, P_C)$ given in Figure 8.3. We consider the following computation of S with one contributor:

$$\rho = (q^0, a^0, p^0) \xrightarrow{!a}_S (q^1, a, p^0) \xrightarrow{?a}_S (q^1, a, p^1) \xrightarrow{!c}_S (q^1, c, p^3).$$

The transition that is marked red in ρ is the first write of c. Note that the first transition, the write of a, is not a first write since the symbol is written by the leader and not by a contributor. We can add any number of copies of the involved contributor to ρ which all arrive in state p^1 and wait to provide c. The corresponding computation is the following:

$$\begin{aligned}(q^0, a^0, p^0, \ldots, p^0) &\xrightarrow{!a}_S (q^1, a, p^0, \ldots, p^0) \xrightarrow{?a}_S (q^1, a, p^1, p^0, \ldots, p^0) \xrightarrow{?a}_S \ldots \\ &\ldots \xrightarrow{?a}_S (q^1, a, p^1, \ldots, p^1) \xrightarrow{!c}_S (q^1, c, p^3, p^1, \ldots, p^1).\end{aligned}$$

Again, the first write is marked red. Note that, after the write, the introduced copies are all waiting to provide c whenever it is needed.

The trick with first writes is that we do not need to store subsequent writes of the contributors in a witness. We only need to store the position where the first write is happening. From then on, we can assume that there is a contributor providing the symbol when we need it. Note that the number of first writes is bounded by d. Clearly, each symbol from the domain D can only be written once for the first time. This means that the number of first-write positions that a witness needs to store is bounded by d.

Definition 8.10. Let $S = (D, a^0, P_L, P_C)$ be a leader contributor system. A *witness* is a triple $x = (w, q, \sigma)$ consisting of the following ingredients.

- The word $w = (q_1, a_1).(q_2, a_2)\dots(q_n, a_n) \in (Q_L \times (D \cup \{\bot\}))^*$ represents a run of the leader P_L. It contains states of P_L followed either by a memory value from the domain D or the symbol $\bot \notin D$.
- The state $q \in Q_L$ is the target state of the encoded run of P_L.
- Let $k \leq d$. The monotonically increasing map $\sigma : [1..k] \to [1..n]$ specifies the positions where first writes happen, the so-called *first-write positions*. Note that σ does not specify any symbols.

The number of first-write positions k is called the *order of* x. We denote it by $ord(x) = k$. Moreover, let $init(x) = q_1$ denote the first state appearing in w. We call the witness x *initialized* if $init(x) = q_L^0$ is the initial state of the leader P_L. The *set of all witnesses* is denoted by *Wit*.

Each computation of a leader contributor system can be compressed into a witness. We illustrate this compression with an example.

Example 8.11. We reconsider the computation of the leader contributor system $T = (D, a^0, P_L, P_C)$ from Example 8.6 and Figure 8.4. It is given below:

$$\begin{aligned}(q^0, a^0, p^0, p^0) \xrightarrow{!a}_C (q^0, a, p^1, p^0) \xrightarrow{?a}_L (q^1, a, p^1, p^0) \xrightarrow{!b}_L (q^2, b, p^1, p^0)\\ \xrightarrow{?b}_C (q^2, b, p^1, p^2) \xrightarrow{!c}_C (q^2, c, p^1, p^2) \xrightarrow{?c}_L (q^3, c, p^1, p^2)\\ \xrightarrow{!a}_C (q^3, a, p^1, p^2) \xrightarrow{?a}_L (q^4, a, p^1, p^2).\end{aligned}$$

Note that we marked the first writes red and that the transitions are labeled by L and C, depending on whether the leader or a contributor moved. We construct a witness $x = (w, q, \sigma)$ out of the computation. The word w encodes the leader run. For its construction, we only store transitions of the leader and we ignore those symbols that were read by the leader. We obtain:

$$w = (q^0, \bot).(q^1, b).(q^2, \bot).(q^3, \bot).$$

Moreover, we have $q = q^4$ as it is the target state of the run on the leader P_L. We may understand w as follows. The leader moves from q^0 to q^1 by reading a particular symbol. Then, it moves from q^1 to q^2 by writing b, and so on.

The reason why we can omit the symbols read by P_L is that, once P_L reads a symbol $a \in D$, it has to be a symbol that already appeared as a first write. Indeed, in the above case P_L reads a when passing from q^0 to q^1. But a has already been written by a contributor for the first time. When passing from q^2 to q^3, the leader P_L reads c. But at this point, also c has seen its first write.

To finish the construction of the witness x, it is left to determine σ. The map shows the first write positions. In our case, it is given by

$$\begin{aligned} \sigma : [1,2] &\to [1..4] \\ 1 &\mapsto 1 \\ 2 &\mapsto 3 \end{aligned}$$

Intuitively, the map encodes that a first write is happening at position 1 of the word w. This is when P_L passes from q^0 to q^1. Note that this is indeed true, since the first write of a happens at that position. According to σ, a first write also appears at position 3 of w. This is also true as the first write of c appears while the leader traverses from state q^2 to q^3.

Altogether, the witness $x = (w, q, \sigma)$ is a compressed representation of the above computation. But the compression looses some information. For instance, we do not recall the actual symbols that appeared at the first write positions. Moreover, we did not store any transition of the contributors. As we will see, we can reconstruct the latter out of the information given by a witness. The former, however, is something that we need to provide.

Validity Some witnesses correspond to an actual computation of the system, some do not. To get a reliable test for correspondence, we formalize two requirements for witnesses: *leader validity* and *contributor validity*. If both are satisfied by a witness, we can guarantee that there is a computation corresponding to the witness at hand. Moreover, each witness that stems from an actual computation satisfies the requirements. Intuitively, leader validity means that the witness at hand encodes a proper run of the leader. Contributor validity ensures that the first writes along this run can be provided by the contributors. However, the definition of a witness only specifies first-write positions but not the actual values of the first writes that appear along the run. To obtain these, we need the notion of *first-write sequences*. These will allow for defining leader and contributor validity.

Definition 8.12. Let $S = (D, a^0, P_L, P_C)$ be a leader contributor system. A *first-write sequence* β is a sequence of distinct data values. Formally:

$$\beta = \beta_1 \dots \beta_k \in D^{\leq d}$$

with $\beta_i \neq \beta_j$ for $i \neq j$. The *set of first write sequences* is denoted by FW.

Fix a leader contributor system $S = (D, a^0, P_L, P_C)$ for the remaining section. We formalize leader validity. Intuitively, the requirement ensures that a witness encodes a proper run of the leader that only writes as depicted by the witness. Reading is restricted to so-called *available first writes* — data values that already appeared in first writes up to the current position of the run. To this end, leader validity is defined along first write sequences. The length of the sequence needs to match the order of the considered witness.

Definition 8.13. Let $x = (w, q, \sigma) \in \mathit{Wit}$ with $w = (q_1, a_1) \dots (q_n, a_n)$ and $\beta \in \mathit{FW}$ a first-write sequence with $|\beta| = \mathit{ord}(x)$. For an index $i \in [1..n]$, let

$$S_\beta(i) = \{\beta_\ell \mid \sigma(\ell) \leq i\}$$

be a set containing all those symbols of β that occurred as first writes up to position i. We call $S_\beta(i)$ the set of *available first writes*. The witness x is called *leader valid (along β)* if for all $i \in [1..n]$, the following holds:

- if $a_i \neq \bot$, the leader P_L has a write transition $q_i \xrightarrow{!a_i}_L q_{i+1}$.
- if $a_i = \bot$, either $q_i = q_{i+1}$, there is a transition $q_i \xrightarrow{\varepsilon}_L q_{i+1}$, or there is a transition $q_i \xrightarrow{?b}_L q_{i+1}$ with $b \in S_\beta(i)$ an available first write.

Note that we assume $q_{n+1} = q$ in the definition. We use the predicate $\mathrm{LValid}_\beta(x)$ to denote that the witness x is leader valid along β.

We apply the technical notion to our running witness example. It illustrates the requirements that are encoded into leader validity.

Example 8.14. Reconsider the witness $x = (w, q, \sigma)$ from Example 8.11 with

$$w = (q^0, \bot).(q^1, b).(q^2, \bot).(q^3, \bot),$$

target state $q = q^4$, and σ defined by $\sigma(1) = 1$ and $\sigma(2) = 3$. It compresses a computation of leader contributor system $T = (D, a^0, P_L, P_C)$ from Figure 8.4. Witness x is leader valid along the first-write sequence $\beta = a.c$. Consider for instance position 3. We have $a_3 = \bot$ and there is a read transition $q^2 \xrightarrow{?c}_L q^3$ of $c \in \{a, c\} = S_\beta(3)$. For position 2, we have $a_2 = b$ and a write transition $q^1 \xrightarrow{!b}_L q^2$. The reasoning for the remaining positions is similar.

We formalize our second requirement — contributor validity. It ensures that first writes can be provided by the contributors in the positions determined by the witness at hand. To provide the i-th first write β_i, at least one contributor needs to reach a state $p \in Q_C$ with an outgoing transition

$$p \xrightarrow{!\beta_i}_C p'.$$

However, on its path to p, reading is restricted to *available writes*, symbols that already appeared as first writes and symbols that the leader can write.

Definition 8.15. Let $x = (w, q, \sigma) \in \mathit{Wit}$ with $w = (q_1, a_1) \dots (q_n, a_n)$. To define the available writes, we need some notation: for $s \in Q_L$ and $\Gamma \subseteq D$, let

$$\mathit{Loop}_L(s, \Gamma) = \{a \mid s \xrightarrow{u.!a.v}_L s \wedge \pi_{RD}(u.!a.v) \in \Gamma^*\},$$

where π_{RD} is the homomorphism that maps a read $?b$ to b and projects away writes, $\pi_{RD}(!b) = \varepsilon$. Intuitively, $\mathit{Loop}_L(s, \Gamma)$ is the set of writes that the leader can provide in a loop over s while reading is restricted to Γ.

Now let $\alpha \in FW$ be a first-write sequence of length $k < \mathit{ord}(x)$. Assume a contributor needs to provide the $(k+1)$-st first write. The symbols that are available to the contributor for reading either stem from the leader P_L or are first writes from fellow contributors. Let $j = \sigma(k+1)$ be the positions of the $(k+1)$-st first write. We define the (regular) *language of available writes* by

$$\mathit{LAW}(x, \alpha) = \Gamma_1^*\{a_1, \varepsilon\}\Gamma_2^*\{a_2, \varepsilon\} \dots \Gamma_j^*,$$

where $\Gamma_i = \mathit{Loop}(q_i, S_\alpha(i)) \cup S_\alpha(i)$. Note that if $a_i = \bot$, we interpret it as $a_i = \varepsilon$.

The language $\mathit{LAW}(x, \alpha)$ is built in such a way that a symbol becomes available for reading once it is written for the first time. Moreover, a first write can potentially enable the leader to access a new loop and thus provide a fresh symbol for reading. Since the leader has only a single instance, unlike the contributors, the copycat argument does not work at this point. In fact, we cannot just provide the writes of P_L as a set to the contributors. Instead, we have to follow a run of P_L which rather leads to the definition of a language.

We are now ready to define contributor validity. We might see it as a reachability problem along the language of available writes.

Definition 8.16. Let $x = (w, q, \sigma)$ be a witness and $\beta \in FW$ a first-write sequence with $|\beta| = \mathit{ord}(x)$. Moreover, let $i \in [1..\mathit{ord}(x)]$ and let

$$Q_i = \{p \in Q_C \mid \exists p' : p \xrightarrow{!\beta_i}_C p'\}$$

be the states of the contributor that can provide the i-th first write β_i. The set $\mathit{Trace}_C(Q_i) = \{w \mid \exists p \in Q_i : p_C^0 \xrightarrow{w}_C p\}$ contains all words leading to Q_i.

The witness x is called *valid for the i-th first write (of β)* if

$$\mathit{LAW}(x, \beta_1 \dots \beta_{i-1}) \cap \pi_{RD}(\mathit{Trace}_C(Q_i)) \neq \emptyset.$$

We use the predicate $\mathrm{CValid}_\beta^i(x)$ to indicate non-emptiness of the intersection. If x is valid for all first writes of β, we call it *contributor valid (along β)* and indicate it with the predicate $\mathrm{CValid}_\beta(x)$. Formally, we have

$$\mathrm{CValid}_\beta(x) = \bigwedge_{i \in [1..\mathit{ord}(x)]} \mathrm{CValid}_\beta^i(x).$$

Note that a witness is valid for the i-th first write if there is a trace leading to the set of states Q_i. During this trace, reading is restricted to available writes that has been seen up to first-write position $\sigma(i)$. We give an example.

Example 8.17. Consider the witness $x = (w, q, \sigma)$ from Example 8.11 where

$$w = (q^0, \bot).(q^1, b).(q^2, \bot).(q^3, \bot),$$

$q = q^4$, and σ is given by $\sigma(1) = 1$ and $\sigma(2) = 3$. We show that x is valid for the second first write of $\beta = a.c$. To this end, recall the structure of the underlying leader contributor system $T = (D, a^0, P_L, P_C)$, shown in Figure 8.4.

The first step is to determine $Q_2 = \{p \in Q_C \mid \exists p' : p \xrightarrow{!c}_C p'\} = \{p^2\}$. The traces leading to Q_2 are given by $Trace_C(\{p^2\}) = ?b.(!c)^*$. Now we have to determine the language of available writes. Since $\sigma(2) = 3$, we have

$$LAW(x, \beta_1) = LAW(x, a) = \Gamma_1^*.\Gamma_2^*.\{b, \varepsilon\}.\Gamma_3^*,$$

with $\Gamma_1 = Loop(q^0, S_a(1)) \cup S_a(1) = \{a\}$ since there are no loops in P_L and we have $S_a(1) = \{a\}$. Similarly, we obtain $\Gamma_2 = \Gamma_3 = \{a\}$. Altogether, $LAW(x, \beta_1) = a^*.(b \cup \varepsilon).a^*$. The intersection of interest is then given by:

$$LAW(x, \beta_1) \cap \pi_{RD}(Trace_C(Q_2)) = a^*.(b \cup \varepsilon).a^* \cap b = b.$$

This means, the contributor which provides the first write of c, can take the trace $?b$ to get to the state p^2. From the state, it can write the symbol c.

With the notions in place, we can now define a witness to be *valid* if it is both, leader valid and contributor valid. As we will see, valid witnesses characterize computations of leader contributor systems. Moreover, it is simple to test whether a given witness is actually valid.

Definition 8.18. Let $x \in Wit$ and $\beta \in FW$ a first-write sequence of length $|\beta| = ord(x)$. If x is leader valid along β and contributor valid along β, we call it *valid (along β)*. Again, we use a predicate: $\text{Valid}_\beta(x) = \text{LValid}_\beta(x) \wedge \text{CValid}_\beta(x)$.

The following lemma shows that validity of a witness along a first-write sequence can be checked in polynomial time. Phrased differently, the corresponding predicate can be evaluated rather efficiently.

Lemma 8.19. *For a given witness $x \in Wit$ and first-write sequence $\beta \in FW$, the predicate* $\text{Valid}_\beta(x)$ *can be evaluated in polynomial time.*

Proof. The predicate $\text{LValid}_\beta(x)$ can be evaluated in polynomial time since we only need to look for appropriate transitions in P_L along x. For contributor validity, consider the predicate $\text{CValid}^i_\beta(x)$. We can clearly construct a finite automaton A for the language $LAW(x, \beta_1 \dots \beta_{i-1})$ in polynomial time. Moreover, the set $\pi_{RD}(Trace_C(Q_i))$ admits a representation via some finite

automaton B. Hence, to check validity for the i-th first write, we construct the cross product of A and B and check the resulting automaton for non-emptiness. This takes polynomial time. By definition, also the predicate $\mathrm{CValid}_\beta(x)$ and thus $\mathrm{Valid}_\beta(x)$ can then be evaluated in polynomial time. □

Note that the lemma implies that all predicates, $\mathrm{LValid}_\beta(x)$, $\mathrm{CValid}^i_\beta(x)$, $\mathrm{CValid}_\beta(x)$, and $\mathrm{Valid}_\beta(x)$, can be evaluated in polynomial time. Moreover, we dropped the assumption $|\beta| = ord(x)$. If order of witness and length of first-write sequence do not match, the predicates all evaluate to false.

Validity ensures that a witness corresponds to an actual computation of the system and vice versa. The following lemma states the desired characterization. We omit the proof. Details can be found in Appendix B.3.1.

Lemma 8.20. *There is an initialized computation $c^0 \to_S^* c$ with $\pi_L(c) = q$ if and only if there is an initialized $x = (w, q, \sigma) \in Wit$ and a $\beta \in FW$ with $\mathrm{Valid}_\beta(x)$.*

Note that the number of witnesses, as they are stated here, is infinite. Hence, we cannot just iterate over all witnesses and test each for validity. When using an adjusted form of witnesses, this is possible. We followed the approach in [93]. However, the obtained algorithm runs in time $(d \cdot l)^{O(d \cdot l)}$ only. Our goal is to speed this up by using a short variant of witnesses.

Short Witnesses Intuitively, a *short witness* encodes a run of the leader without loops. This limits their length and hence constitutes a suitable foundation for an algorithmic approach. The idea is as follows. It suffices to consider (long) witnesses x that can be decomposed into a concatenation of short witnesses: $x = x_1 \times \cdots \times x_k$. An x_i then encodes that part of the leader run of x around the i-th first write. Leader validity and contributor validity of x along a first-write sequence can then be checked inductively along the short witnesses x_i. We exploit the inductive structure by a dynamic programming which involves only short witnesses and runs in time proportional to their number. Since there are at most $(d + l)^{O(d+l)}$ many, we obtain the desired running time. Before we can formulate the corresponding algorithm, we need to introduce short witnesses and the concatenation operator mentioned above.

Definition 8.21. A *short witness* is a witness $x = (w, q, \sigma) \in Wit$ where the leader states in $w = (q_1, a_1) \dots (q_n, a_n)$ are distinct. We have $q_i \neq q_j$ for $i \neq j$. We use Wit^{sh} to denote the *set of all short witnesses*. Moreover for $k \in [1..d]$, let $Ord(k)$ denote the set of those short witnesses x with $ord(x) = k$.

The concatenation operator can be applied to all witnesses instead of only short ones. Intuitively, it concatenates the leader runs of the given operands and glues together the first-write positions accordingly. To make this a reasonable definition, we assume that the target state of the first operand matches the initial state encoded in the second operand.

Definition 8.22. Let $x = (w, q, \sigma)$ and $y = (v, p, \tau)$ be two witnesses with $\mathit{init}(y) = q$. The *concatenation of x and y* is the witness

$$x \times y = (w.v, p, \rho).$$

The map $\rho = \sigma.\tau$ is the concatenation of σ and τ. If $\sigma : [1..k] \to [1..n]$ and $\tau : [1..\ell] \to [1..m]$, then $\rho : [1..k + \ell] \to [1..n + m]$ is defined by

$$\rho(i) = \begin{cases} \sigma(i), \text{ if } i \leq k, \\ \tau(i - k) + n, \text{ otherwise.} \end{cases}$$

Note that ρ is monotonically increasing since σ and τ are. Hence, $x \times y$ is indeed a witness. Moreover, we have $\mathit{ord}(x \times y) = \mathit{ord}(x) + \mathit{ord}(y)$.

We will later use the concatenation of witnesses to increase the order step by step. However, note that the concatenation does not yield a short witness. Combining loop-free runs of the leader may well lead to repetition of states and therefore spawn loops. We define an operator that shrinks a given witness to a short one by cutting out these loops.

Definition 8.23. Let (w, q, σ) be a witness with $w = (q_1, a_1) \dots (q_n, a_n)$. Let $i \in [1..n]$ be the least index such that there exists an $i' > i$ with $q_i = q_{i'}$. Fix j to be the minimal among the indices i'. We define the operator

$$\begin{aligned} \mathit{Shrink} : \mathit{Wit} &\to \mathit{Wit} \\ (w, q, \sigma) &\mapsto (w', q, \sigma'), \end{aligned}$$

where $w' = (q_1, a_1) \dots (q_{i-1}, a_{i-1}).(q_j, a_j) \dots (q_n, a_n)$ and the first-write positions σ' are defined by cutting out $j - i$ positions:

$$\sigma'(\ell) = \begin{cases} \sigma(\ell), \text{ if } \sigma(\ell) < i, \\ i, \text{ if } i \leq \sigma(\ell) \leq j, \\ \sigma(\ell) - j + i, \text{ otherwise.} \end{cases}$$

If the input is a short witness, *Shrink* is the identity. To obtain a short witness from a long one, we apply the operator repeatedly. Let Shrink^* denote the repeated application of *Shrink* until a fixed point is reached — until all loops are cut out. We obtain $\mathit{Shrink}^* : \mathit{Wit} \to \mathit{Wit}^{sh}$.

Note that Shrink^* does not change the order of a given witness. It preserves all first-write positions. We employ the operator along with the above concatenation to obtain a product that keeps short witnesses short.

Definition 8.24. Let $x = (w, q, \sigma) \in \mathit{Ord}(k)$ and $y = (v, p, \tau) \in \mathit{Ord}(\ell)$ with $\mathit{init}(y) = q$. The *short concatenation of x and y* is defined to be the short witness

$$x \otimes y = \mathit{Shrink}^*(x \times y) \in \mathit{Ord}(k + \ell).$$

With these tools at hand, we can describe how validity of a witness can be checked inductively. The idea is to build up a valid witness step by step. We start from a single short witness. In each step, we concatenate the current witness at hand with a new short witness, responsible for providing the next first write. Hence, each step increases the order by exactly one. We stop the process as soon as all first writes have been provided.

During the process, we have to take care of two properties. First, we only concatenate with short witnesses preserving validity. Second, to obtain an algorithm with the desired running time from the process, we cannot afford working with long witnesses. We need to mix the concatenation with shrinking to obtain a short witness in each step. The discussion leads to an inductive definition of validity for short witnesses — depending on the order.

Definition 8.25. Let $z \in \mathit{Wit}^{sh}$ be a short witness and $\beta \in \mathit{FW}$ a first-write sequence such that $|\beta| = \mathit{ord}(z)$. We call z *short valid (along β)* if the predicate $\mathrm{Valid}^{sh}_{\beta}(z)$ evaluates to true. The predicate is defined depending on $\mathit{ord}(z)$.

- If $\mathit{ord}(z) = 0$, then $\beta = \varepsilon$. There are no first-write positions and hence, only leader validity is important. In this case, we have:

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

- If $\mathit{ord}(z) = k + 1$ for a $k \in \mathbb{N}$, then $\beta = \beta_1 \dots \beta_{k+1}$. We have:

$$\mathrm{Valid}^{sh}_{\beta}(z) = \bigvee_{\substack{x \in \mathit{Ord}(k) \\ y \in \mathit{Ord}(1)}} \left(\begin{array}{l} [z = x \otimes y] \wedge \mathrm{LValid}_{\beta}(x \times y) \wedge \\ \mathrm{CValid}^{k+1}_{\beta}(x \times y) \wedge \mathrm{Valid}^{sh}_{\beta'}(x) \end{array} \right),$$

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

We elaborate on the recursive structure of the predicate $\mathrm{Valid}^{sh}_{\beta}(z)$ in more detail. The main idea is to cut off the last first write β_{k+1}, check z's validity with β_{k+1} separately, and then continue on the remaining part. To this end, z is decomposed into two short witnesses $x \in \mathit{Ord}(k)$ and $y \in \mathit{Ord}(1)$. The bracket $[z = x \otimes y]$ evaluates to true if and only if z is the short concatenation of x and y. Intuitively, y is the short witness responsible for providing the last first write β_{k+1} and x is the remaining part. The short witness x is already known to be valid along β' since $\mathrm{Valid}^{sh}_{\beta'}(x)$ holds. With the remaining predicates, we test whether x and y can be composed while maintaining validity. In fact, $\mathrm{LValid}_{\beta}(x \times y)$ guarantees leader validity and $\mathrm{CValid}^{k+1}_{\beta}(x \times y)$ ensures that y provides the first write β_{k+1}, and thus contributor validity holds.

The upcoming lemma shows that (usual) validity can be replaced by short validity. Both notions are capable of witnessing computations leading to some final state in the leader contributor system. This has two implications. We can work with short witnesses instead of long ones and, algorithmically, we can check (short) validity of a witness inductively along Definition 8.25.

Lemma 8.26. *Let $\beta \in FW$ be some first-write sequence. There is a (long) witness $x = (w, q, \sigma) \in Wit$ with $\mathrm{Valid}_\beta(x)$ and $init(x) = p$ if and only if there is a short witness $z = (v, q, \tau) \in Wit^{sh}$ with $\mathrm{Valid}^{sh}_\beta(z)$ and $init(z) = p$.*

We omit the proof of Lemma 8.26 and refer to Appendix B.3.2 for technical details. Along with Lemma 8.20 we have now shown that LCR reduces to finding an initialized short witness $z \in Wit^{sh}$ and a first-write sequence $\beta \in FW$ such that the predicate $\mathrm{Valid}^{sh}_\beta(z)$ evaluates to true.

Algorithm We present the algorithm that finds valid short witnesses and decides LCR. The idea is to turn the recursive definition of short validity into a dynamic programming. In Algorithm 8.1, we give a detailed description.

The main idea is as follows. For each first-write sequence $\beta \in FW$ and each short witness $z \in Wit^{sh}$, we compute $\mathrm{Valid}^{sh}_\beta(z)$ until we find a valid z that is initialized and the target state of which is in Q_F. Note that in the latter case, we witness a computation of the leader contributor system in which the leader visits a final state. To store already computed validity information, we maintain a table T which has an entry $T[\beta, z]$ for each $\beta \in FW$ and each $z \in Wit^{sh}$ with $ord(z) = |\beta|$. It stores the validity predicate: $T[\beta, z] = \mathrm{Valid}^{sh}_\beta(z)$.

To compute the table T, we elaborate on how a single entry $T[\beta, z]$ is computed. According to Definition 8.25, for $ord(z) = |\beta| = 0$ we have

$$T[\varepsilon, z] = \mathrm{LValid}_\varepsilon(z).$$

If $ord(z) = |\beta| = k \geq 1$, we employ the recursive structure of $\mathrm{LValid}_\beta(z)$. We iterate over all short witnesses $x \in Ord(k-1)$ and $y \in Ord(1)$ and check whether $z = x \otimes y$ holds. If so, we compute the predicates $\mathrm{LValid}_\beta(x \times y)$ and $\mathrm{CValid}^k_\beta(x \times y)$ and look up $\mathrm{Valid}^{sh}_{\beta'}(x) = T[\beta', x]$ in the table, where $\beta' = \beta_1 \dots \beta_{k-1}$ is the trimmed first-write sequence. The entry is given by:

$$T[\beta, z] = \mathrm{LValid}_\beta(x \times y) \wedge \mathrm{CValid}^k_\beta(x \times y) \wedge T[\beta', x].$$

We stop the computation of T as soon as we find a $\beta \in FW$ and a $z \in Wit^{sh}$ with target state q and $init(z) = q^0_L$ such that $T[\beta, z] = true$. In this case, we can return *true*. If we do not find such a β and z, we return *false*. The correctness of Algorithm 8.1 follows from Lemma 8.20 and Lemma 8.26.

It is left to determine the complexity of the algorithm. We provide a complete analysis in Appendix B.3.3. Here, we elaborate on the idea. The dominant time factor comes from the number of entries of T and the time that is required to compute an entry. The former is proportional to the number of short witnesses: $(d + l)^{O(d+l)}$. Since (short) concatenation and the validity predicates $\mathrm{LValid}_\beta(w)$ and $\mathrm{CValid}^k_\beta(w)$ can be computed in polynomial time, the time required to compute an entry of T is also bounded by $(d + l)^{O(d+l)}$. Hence, the running time is $(d + l)^{O(d+l)}$, as stated in Theorem 8.7.

Lemma 8.27. *Algorithm 8.1 runs in time $(d + l)^{O(d+l)}$.*

Algorithm 8.1 Leader Contributor Reachability

Input: An LCS $S = (D, a^0, P_L, P_C)$, a set of final states $Q_F \subseteq Q_L$.
Output: *True*, if $\exists$ initialized $c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$. *False* otherwise.

let T be a table with an entry $T[\beta, z]$ for $\beta \in FW$, $z \in Wit^{sh}$.
for each $z \in Ord(0)$ **do**
  $T[\varepsilon, z] = \text{LValid}_\varepsilon(z)$ // Order 0 — no first writes.
end for
for $k = 1, \ldots, d$ **do**
  for each $\beta \in FW$ with $|\beta| = k$ **do**
    for each $z \in Ord(k)$ **do**
      set $T[\beta, z] = false$ // Initialize entry with false.
      for each $x \in Ord(k-1)$ and $y \in Ord(1)$ **do**
        if $z = x \otimes y$ **then** // Decomposition of z found.
          compute witness $w = x \times y$
          compute Boolean $b_{val} = \text{LValid}_\beta(w) \wedge \text{CValid}_\beta^k(w)$
          look up value $b_{rec} = T[\beta_1 \ldots \beta_{k-1}, x]$
          if $b_{val} \wedge b_{rec}$ **then**
            set $T[\beta, z] = true$ // Valid according to Definition 8.25.
            if target state of z is q_L^0 and $init(z) \in Q_F$ **then**
              return *true* // Valid short witness leading to Q_F found.
            else
              goto next witness $z \in Ord(k)$
            end if
          end if
        end if
      end for
    end for
  end for
end for
return *false*

8.2.3 Parameterization by Contributor

The second algorithm of our fine-grained complexity analysis of LCR focuses on the parameterization in the size c of the contributor. We show that the parameter has substantial influence on the complexity of the problem. In fact, our algorithm solves LCR in time $O^*(2^c)$ only. A matching lower bound for the algorithm is provided later, in Section 8.3.1.

Our algorithm is based on a characterization of computations in terms of paths in a so-called *saturation graph*. This reduces LCR to reachability in that graph. The saturation graph represents an arbitrary number of contributors by only a single set. Intuitively, it simulates a computation of the leader contributor system while saturating the set of states that can be reached by the contributors. With a dynamic programming over the subsets of the contributor states, we can then efficiently decide reachability in the saturation graph. This solves LCR. The corresponding result is as follows.

Theorem 8.28. *The problem LCR can be solved in time* $O(2^c \cdot c^4 \cdot d^2 \cdot l^2)$.

In the following, we prove Theorem 8.28. To this end, we fix a leader contributor system $S = (D, a^0, P_L, P_C)$ and a set of final states $Q_F \subseteq Q_L$.

Saturation Graph The reason d'être of the saturation graph is to reduce from the infinite set of configurations to a manageable search space. Key to this reduction is the observation that keeping one set of states for all contributors suffices. Indeed, assume that the contributors can reach a particular set of states $S \subseteq Q_C$ during a computation. Then we can assume that for each $p \in S$, there is an arbitrary number of contributors that are currently in state p. The reason is the *copycat lemma* [152]: we can always mimic the behavior of a contributor by an arbitrary number of other contributors. Hence, if one contributor can reach state $p \in S$, an arbitrary number of contributors can. This means that we do not have to distinguish between different contributor instances anymore. We formally define the graph of interest.

Definition 8.29. The *saturation graph* $\mathcal{G} = (V, E)$ is a directed graph over

$$V = Q_L \times D \times \mathcal{P}(Q_C).$$

Hence, a vertex $v \in V$ is not an explicit configuration but a tuple $v = (q, a, S)$, where $q \in Q_L$, $a \in D$, and $S \subseteq Q_C$. Between the vertices, we define the edges E. They mimic the transition relation among configurations.

Write and read transitions of the leader change the current state of P_L and potentially the memory value. Let $b \in D$ and $S \subseteq Q_C$, we have:

$$q \xrightarrow{!a}_L q' \text{ implies } (q, b, S) \to_E (q', a, S),$$
$$q \xrightarrow{?b/\varepsilon}_L q' \text{ implies } (q, b, S) \to_E (q', b, S).$$

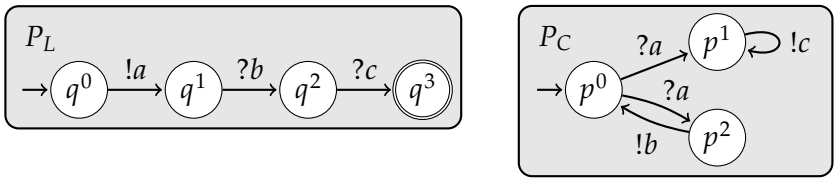


Figure 8.5: Leader contributor system $S = (D, a^0, P_L, P_C)$ over the domain $D = \{a^0, a, b, c\}$. The only final state is given by $Q_F = \{q^3\}$.

Similarly, we obtain edges that mimic the behavior of the contributors. These edges also change the memory value but only saturate the set S instead of precisely tracking the set of states that the contributors reached. Let $q \in Q_L$ and $b \in D$. Like for the leader, we have two cases:

$$p \xrightarrow{!a}_C p' \text{ implies } (q, b, S) \to_E (q, a, S \cup \{p'\}),$$
$$p \xrightarrow{?b/\varepsilon}_C p' \text{ implies } (q, b, S) \to_E (q, b, S \cup \{p'\}).$$

Computations start in the initial configuration. The counterpart in the saturation graph $\mathcal{G}$ is the *initial vertex* $v^0 = (q_L^0, a^0, \{q_C^0\})$.

The saturation graph is built to simulate computations of the leader contributor system. But before we elaborate on the corresponding characterization of computations, we illustrate the graph with an example.

Example 8.30. Consider the leader contributor system given in Figure 8.5. It contains a final state $Q_F = \{q^3\}$. We construct the saturation graph $\mathcal{G} = (V, E)$ associated to the system. The vertices are given by

$$V = Q_L \times D \times \mathcal{P}(\{p^0, p^1, p^2\}).$$

The edges are constructed along P_L and P_C. For instance, there is an edge $(q^1, a, \{p^0\}) \to_E (q^1, a, \{p^0, p^1\})$ since P_C has a read transition from p^0 to p^1 on symbol a. Intuitively, the edge describes that currently, the leader is in q^1, the memory holds a, and an arbitrary number of contributors is in p^0. Then, some of these read a and move to p^1, some stay in p^0. Hence, we might assume an arbitrary number of contributors in both states now, p^0 and p^1.

We constructed the saturation graph $\mathcal{G}$ in Figure 8.6. Note that it is a collection of subgraphs $\mathcal{G}_S = \mathcal{G}[Q_L \times D \times \{S\}]$. In the graph, the vertex (q^3, c, Q_C) involving the final state of the leader is reachable from the initial vertex $v^0 = (q^0, a^0, \{p^0\})$. A path is for instance given by

$$\begin{aligned} v^0 &\to_E (q^1, a, \{p^0\}) \to_E (q^1, a, \{p^0, p^1\}) \to_E (q^1, a, Q_C) \\ &\to_E (q^1, b, Q_C) \to_E (q^2, b, Q_C) \to_E (q^2, c, Q_C) \to_E (q^3, c, Q_C). \end{aligned}$$

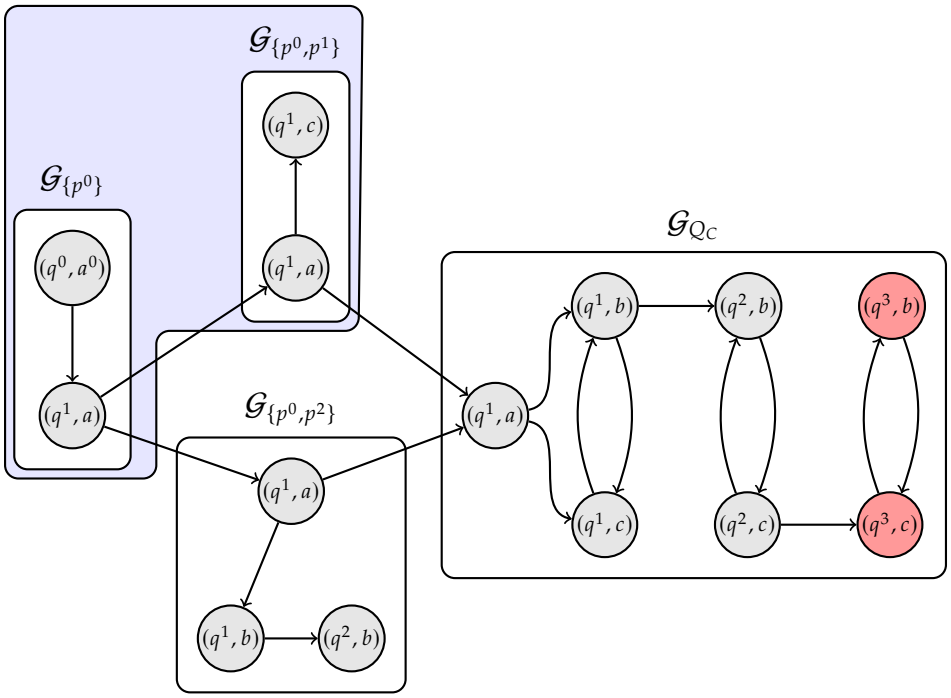


Figure 8.6: Saturation graph $\mathcal{G}$ of the LCS given in Figure 8.5. Self-loops and vertices not reachable from $v^0 = (q^0, a^0, \{p^0\})$ are omitted. We use $\mathcal{G}_S$ for the induced subgraph $\mathcal{G}[Q_L \times D \times \{S\}]$. Since it is clear from the context, we also omit the third component of the vertices. Red vertices involve the final state q^3 of the leader. The blue highlighted area shows the slice $\mathcal{G}_{\{p^0\},\{p^0,p^1\}}$.

The path encodes a computation of the underlying leader contributor system. In fact, it starts in c^0 and the leader first writes a to the memory, the contributors read the symbol and move to the states p^1, p^2, and one stays in p^0. Hence, the contributors reached the set Q_C of states. Then one contributor provides b, another one provides c, so that the leader can arrive at q^3.

The example shows that paths in the saturation graph correspond to computations of the system. The observation is true in general: for each computation we can construct a corresponding path and vice versa. This means that we can reduce LCR to the problem of finding an appropriate path in the saturation graph. The following lemma states the desired characterization.

Lemma 8.31. *There is an initialized computation $c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$ if and only if there is a path $v^0 \to_E^* v$ in $\mathcal{G}$ with vertex $v \in Q_F \times D \times \mathcal{P}(Q_C)$.*

We omit the proof of Lemma 8.31 and refer to Appendix B.3.4 for technical details. The lemma immediately yields an algorithmic idea for solving LCR: construct the saturation graph $\mathcal{G}$ and apply a suitable path finding algorithm.

We followed the approach in [93]. However, the resulting algorithm runs in time $O^*(4^c)$ and is therefore too slow to match the desired running time stated in Theorem 8.28. In fact, the expensive part is the construction of the saturation graph $\mathcal{G}$. Consequently, we need to solve the reachability problem on the graph without explicitly constructing it at once.

Slicewise Reachability To avoid the construction of the saturation graph, we restrict to so-called *slices*. These are polynomially-sized subgraphs of $\mathcal{G}$ where reachability queries can be decided efficiently. The general reachability problem on $\mathcal{G}$ can then be solved by accumulating reachability information of the slices. To this end, we employ a table T storing this information. Our algorithm then fills the table with a dynamic programming. We define T.

Definition 8.32. Table T has for each set $S \subseteq Q_C$ an entry $T[S]$, defined by

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

Hence, $T[S]$ contains all vertices in the subgraph $\mathcal{G}_S = \mathcal{G}[Q_L \times D \times \{S\}]$, where the third component is fixed to be S, that are reachable from v^0.

Once computed, table T stores sufficient information to solve the considered problem LCR. Indeed, any entry of T that contains a final state of the leader clearly witnesses a computation to the same. We formally state the correspondence. Note that the proof follows from Lemma 8.31.

Lemma 8.33. *There is an initialized computation $c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$ if and only if there exists a subset $S \subseteq Q_C$ such that $T[S] \cap (Q_F \times D) \neq \emptyset$.*

It remains to compute the table. To this end, we show that T admits a recurrence relation which can be exploited by a dynamic programming to fill T bottom-up. The relation relies on the slices of $\mathcal{G}$. Intuitively, these subgraphs divide the reachability problem on $\mathcal{G}$ along the subsets of states that the contributors can reach. We can then consider the problem slicewise.

Definition 8.34. Let $W \subseteq Q_C$ be a subset and $p \in Q_C \setminus W$ be a state. Set $S = W \cup \{p\}$. The *slice* $\mathcal{G}_{W,S}$ is the induced subgraph

$$\mathcal{G}_{W,S} = \mathcal{G}[Q_L \times D \times \{W, S\}].$$

We denote the edges of $\mathcal{G}_{W,S}$ by $E_{W,S}$.

A slice $\mathcal{G}_{W,S}$ fixes the set of contributor states to be W and S. Phrased differently, it focuses on the interface between the subgraphs $\mathcal{G}_W$ and $\mathcal{G}_S$ in the saturation graph. We illustrate the notion with an example.

Example 8.35. Consider the saturation graph shown in Figure 8.6. The blue highlighted subgraph is the slice $\mathcal{G}_{\{p^0\},\{p^0,p^1\}}$. It contains the subgraphs $\mathcal{G}_{\{p^0\}}$ and $\mathcal{G}_{\{p^0,p^1\}}$ as well as the connecting edge $(q^1, a, \{p^0\}) \rightarrow_E (q^1, a, \{p^0, p^1\})$. Other slices of the saturation graph can be found similarly.

Note that each path from v^0 to (q^3, c, Q_F) either leads through the two slices $\mathcal{G}_{\{p^0\},\{p^0,p^1\}}$ and $\mathcal{G}_{\{p^0,p^1\},Q_C}$ or the two slices $\mathcal{G}_{\{p^0\},\{p^0,p^2\}}$ and $\mathcal{G}_{\{p^0,p^2\},Q_C}$.

With the notion of slices at hand, we can formulate the desired recurrence relation for the table T. The main idea is to reduce the reachability problem of $\mathcal{G}$ to its slices. Let ρ be a path in $\mathcal{G}$. Then, the set of contributor states seen along ρ gets saturated over time. If we cut ρ each time a new contributor state gets added, we obtain subpaths which belong to different slices of $\mathcal{G}$. For instance, if a state $p \in Q_C$ gets added to the currently visited set $W \subseteq Q_C$ of contributor states, the corresponding subpath leads from $\mathcal{G}_W$ to $\mathcal{G}_S$, where S is defined by $S = W \cup \{p\}$. Hence, it is a path in the slice $\mathcal{G}_{W,S}$.

The observation shows that any path in the saturation graph $\mathcal{G}$ is a concatenation of paths in the slices of $\mathcal{G}$. This means that we can identify the reachable vertices of $\mathcal{G}$ slice by slice. To this end, we need the following notion. It makes the reachability problem within a single slice more precise.

Definition 8.36. Let $W \subseteq Q_C$ be a subset of contributor states and $p \in Q_C \setminus W$. Set $S = W \cup \{p\}$. The set $R(W, S) \subseteq Q_L \times D$ contains all vertices of $\mathcal{G}_S$ that are reachable from $T[W]$ within the slice $\mathcal{G}_{W,S}$. Formally, we have

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

The following recurrence relation on the entries of T is the foundation of our algorithm for filling the table. It reflects the fact that a vertex (q, a, S) in $\mathcal{G}$ is reachable from v^0 if it can be reached from some vertex $(q', a', S \setminus \{p\})$ which is itself reachable from v^0. The state p can be any state in S. The proof of the recurrence relation immediately follows from the definitions.

Lemma 8.37. *Table T admits the recurrence relation $T[S] = \bigcup_{p \in S} R(S \setminus \{p\}, S)$.*

Before we state our algorithm for computing the table T, we illustrate the introduced notions and the recurrence relation with an example.

Example 8.38. Reconsider the saturation graph given in Figure 8.6. The table T has eight entries in this case, one for each subset of $Q_C = \{p^0, p^1, p^2\}$. But there are only four entries that are non-empty. These can be drawn out of Figure 8.6. Each of the shown subgraphs contains exactly those vertices that are reachable from the initial vertex v^0. For instance, we have

$$T[\{p^0\}] = \{(q^0, a^0), (q^1, a)\} \text{ and}$$
$$T[\{p^0, p^1\}] = \{(q^1, a), (q^1, c)\}.$$

For $W = \{p^0\}$ and $S = \{p^0, p^1\}$, the set $R(W, S)$ contains those vertices in $\mathcal{G}_S$ that are reachable from $T[W] = T[\{p^0\}]$. According to the graph, these are (q^1, a, S) and (q^1, c, S) and hence we get $T[S] = R(W, S)$.

Algorithm We apply the recurrence relation of Lemma 8.37 in a bottom-up dynamic programming to fill the table T. Let $S \subseteq Q_C$ be a subset and assume we have already computed the entries $T[S \setminus \{p\}]$, for each $p \in S$. To compute $T[S] = \bigcup_{p \in S} R(S \setminus \{p\}, S)$, we first need to determine the sets $R(S \setminus \{p\}, S)$. For a fixed p, we can compute $R(S \setminus \{p\}, S)$ by a simple fixed-point iteration on the slice $\mathcal{G}_{S \setminus \{p\}, S}$ which finds the reachable vertices. Then, we construct the union of all sets $R(S \setminus \{p\}, S)$ and obtain the new entry $T[S]$.

We realized the approach in Algorithm 8.2, stated below. Note that the algorithm computes the entry $T[\{p_C^0\}] = \{(q, a) \mid v^0 \to_E^* (q, a, \{p_C^0\})\}$ by a fixed-point iteration on the subgraph $\mathcal{G}_{\{p_C^0\}}$. Moreover, it applies the aforementioned steps for computing the remaining entries $T[S]$. The correctness of Algorithm 8.2 follows from Lemma 8.37 and Lemma 8.33.

Algorithm 8.2 Leader Contributor Reachability

Input: An LCS $S = (D, a^0, P_L, P_C)$, a set of final states $Q_F \subseteq Q_L$.
Output: *True*, if $\exists$ initialized $c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$. *False* otherwise.

```
1:  let T be a table with entry T[S] = ∅ for each S ⊆ Q_C.   // Initialize table.
2:  compute T[{p_C^0}]   // By a fixed-point iteration in G_{{p_C^0}}.
3:  for each {p^0} ⊊ S ⊆ Q_C do   // Some increasing order on subsets of Q_C.
4:     for each p ∈ S do
5:        compute R = R(S \ {p}, S)   // By a fixed-point iteration in G_{S\{p},S}.
6:        set T[S] = T[S] ∪ R
7:     end for
8:     if T[S] ∩ (Q_F × D) ≠ ∅ then
9:        return true   // Computation exists due to Lemma 8.33.
10:    end if
11: end for
12: return false
```

It is left to determine the complexity of Algorithm 8.2. Note that , in the worst case, the algorithm computes all 2^c many entries of the table T. Hence, we need to determine the time needed to compute a single entry. According to the recurrence relation given in Lemma 8.37, an entry $T[S]$ is given by the union of the sets $R(S \setminus \{p\}, S)$ with $p \in S$. The following lemma shows that we can compute these efficiently — by a fixed-point iteration on the slice $\mathcal{G}_{S \setminus \{p\}, S}$.

Lemma 8.39. *Let $S \subseteq Q_C$ and $p \in S$. Assume we have already computed the entry $T[S \setminus \{p\}]$. Then, we can compute the set $R(S \setminus \{p\}, S)$ in time $O(c^3 \cdot d^2 \cdot l^2)$.*

Proof. We elaborate on the idea. A detailed proof is given in Appendix B.3.5. First, we construct the slice $\mathcal{G}_{S\setminus\{p\},S}$. This takes time $O(c^3 \cdot d^2 \cdot l^2)$. Then we apply a fixed-point iteration which identifies all vertices (q, a, S) in the slice that are reachable from some vertex $(q', a', S \setminus \{p\})$ with $(q', a') \in T[S \setminus \{p\}]$. The iteration runs in time $O(c^2 \cdot l^2)$ since the slice contains $O(c \cdot l)$ many vertices. Hence, the dominant time factor is due to the construction of $\mathcal{G}_{S\setminus\{p\},S}$. □

Wrapping up, an entry $T[S]$ can be computed in time $O(c^4 \cdot d^2 \cdot l^2)$ since it is the union of at most $|S| \leq c$ many sets $R(S \setminus \{p\}, S)$. Note that the complexity estimation also holds for $T[\{p_C^0\}]$, which is computed by a fixed-point iteration on the subgraph $\mathcal{G}_{\{p_C^0\}}$. Hence, Algorithm 8.2 runs in time $O(2^c \cdot c^4 \cdot d^2 \cdot l^2)$.

8.3 Lower Bounds for Safety Verification

We present the lower bounds found by our fine-grained complexity analysis of the problem LCR. In Section 8.3.1, we provide two lower bounds, one for each parameterization LCR(d, l) and LCR(c). These prove that the safety verification algorithms from Section 8.2 are optimal in the fine-grained sense. In Section 8.3.2, we show that both parameterizations are unlikely to admit a polynomial kernel. At last, in Section 8.3.3, we prove that the parameterizations LCR(d) and LCR(l) are W[1]-hard and therefore intractable.

8.3.1 Optimality of Safety Verification Algorithms

We prove that both tractable parameterizations of LCR, namely LCR(d, l) and LCR(c), admit fine-grained lower bounds. These show that the presented FPT-algorithms for the parameterizations are optimal in the fine-grained sense.

For the former parameterization LCR(d, l) we rely on the ETH as a lower bound provider. Technically, we prove that the problem cannot be solved in time $2^{o((d+l)\cdot\log(d+l))}$. This matches the running time $(d+l)^{O(d+l)}$ of the algorithm for the parameterization presented in Section 8.2.2. For LCR(c), we obtain a lower bound from the SCON. It yields that LCR cannot be solved in time $O^*((2-\varepsilon)^c)$ for any $\varepsilon > 0$. Again, this matches the upper bound for the parameterization: the algorithm from Section 8.2.3 runs in time $O^*(2^c)$.

Parameterization by Domain and Leader We prove that LCR cannot be solved in time $2^{o((d+l)\cdot\log(d+l))}$ unless the ETHfails. To achieve the lower bound, we give a reduction from the problem $k\times k$-Clique to LCR. Technically, given a $k\times k$–graph G, where the vertices are arranged in a $k\times k$-matrix, we construct a leader contributor system S that satisfies the following: the data domain and the leader are of size linear in k. Formally, $d = O(k)$ and $l = O(k)$. Moreover, there is a computation of S where the leader can reach a final state if and only if G contains a clique with one vertex from each of its rows.

The reduction implies the desired lower bound. Indeed, assume there is some algorithm for LCR which solves the problem in time $2^{o((d+l)\cdot\log(d+l))}$. Combined with the above reduction, which takes polynomial time, we obtain an algorithm that solves $k \times k$-Clique and runs in time:

$$2^{o((d+l)\cdot\log(d+l))} = 2^{o((k+k)\cdot\log(k+k))} = 2^{o(k\cdot\log(k))}.$$

By Theorem 5.13, this contradicts the exponential time hypothesis. Hence, such an algorithm for LCR cannot exist. We elaborate on the reduction.

Theorem 8.40. *Unless ETH fails, LCR cannot be solved in time* $2^{o((d+l)\cdot\log(d+l))}$.

Proof. Let G be an instance of $k \times k$-Clique. Recall that a vertex of G is a tuple $(i, j) \in V = [1..k] \times [1..k]$, where i is the row and j is the column. In the reduction to LCR, we need that the leader and the contributors can communicate on these vertices in order to find the desired clique. However, we cannot store tuples (i, j) in the memory immediately. This would cause a quadratic blow-up $d = O(k^2)$. Instead, we communicate a vertex (i, j) as a string $\mathrm{row}(i).\,\mathrm{col}(j)$. Formally, the data domain of the leader contributor system $S = (D, a^0, P_L, P_C)$, that we are about to construct, is then given by

$$D = \{\mathrm{row}(i), \mathrm{col}(i), \#_i \mid i \in [1..k]\} \cup \{a^0\}.$$

The symbols $\#_i$ will become clear in a moment, the symbol a^0 is the initial memory value. Note that d is now linear in k. Moreover, note that we need to explicitly distinguish between row and column symbols to avoids stuttering, the repeated reading of the same symbol. In fact, without the distinction it could happen that a contributor or the leader reads a row symbol twice and takes it for a column. With the chosen data domain, we circumvent this.

A computation of the system S starts with each involved contributor choosing a vertex of G to store. Let us for simplicity denote a contributor storing the vertex $(i, j) \in V$ by $P_{(i,j)}$. Note that there might be several copies of $P_{(i,j)}$. Since there are arbitrarily many contributors, the initially chosen vertices are only a superset of the clique which 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. Other contributors either store the guessed vertex or a vertex from a different row. The former can continue their computation, the latter ones need to verify that their stored vertex is connected to the guessed one.

Technically, the leader contributor system S performs for each $i \in [1..k]$ the following steps. If (i, j) is the vertex of interest, P_L writes the symbol $\mathrm{row}(i)$ to the memory. Each contributor that is still active reads the symbol and moves on for one state. Then P_L communicates the chosen column by writing $\mathrm{col}(j)$. The structure of P_L is illustrated in Figure 8.7.

Upon reading the symbol $\mathrm{col}(j)$, a contributor $P_{(i',j')}$ reacts in one of the following three ways. (1) If $i' \neq i$, then $P_{(i',j')}$ stores a vertex of a different

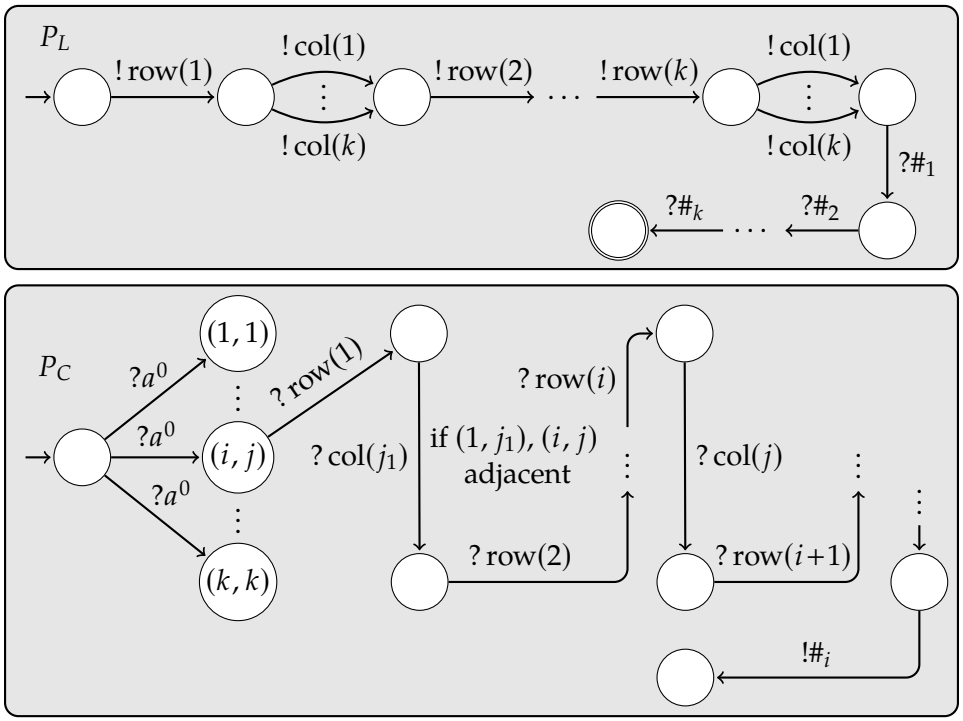


Figure 8.7: The leader P_L shown above guesses k vertices, one from each row. After guessing, it waits for the confirmatiion $\#_1 \ldots \#_k$. A contributor P_C chooses a vertex (i, j) to store. It can only provide $\#_i$ if the leader guesses vertices that are connected to (i, j) and, for the i-th row, exactly guesses (i, j).

row. The computation in $P_{(i',j')}$ can only continue if (i', j') is adjacent to (i, j) in the given graph G. Otherwise it will stop. (2) If $i' = i$ and $j' = j$, 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 $j' \neq j$, contributor $P_{(i',j')}$ stores a different vertex from the i-th row. The computation in this contributor immediately stops. We constructed a contributor thread in Figure 8.7.

After k rounds of the above communication pattern, there are only contributors left that store vertices guessed by the leader P_L. Moreover, each two of these vertices are adjacent and hence, they form a clique. To transmit this information to the leader P_L, each $P_{(i,j)}$ writes the symbol $\#_i$ to the memory, a special confirmation symbol for row i. After P_L has read the string $\#_1 \ldots \#_k$, it moves to its final state. We obtain that there exists a computation of S where P_L visits its final state if and only if G contains a clique of size k with exactly one vertex from each row. A formal construction of the system S and a proof of correctness are given in Appendix B.3.6. □

Parameterization by Contributor We prove that an algorithm for LCR running in time $O^*((2-\varepsilon)^c)$ for an $\varepsilon > 0$ is unlikely to exist. To this end, we rely on the SCON and establish a tight linear reduction from Set Cover to LCR. More precise, we turn an instance $(U, \mathcal{F}, r)$ of Set Cover with $n = |U|$ into a leader contributor system S such that the size of the contributor satisfies $c = n + t$ where $t \in \mathbb{N}$ is some constant. Moreover, we show that there are r sets in $\mathcal{F}$ that cover U if and only if there is a computation of S where the leader visits a final state. By Lemma 5.32, the desired lower bound for LCR follows.

Theorem 8.41. *Unless SCON fails, LCR cannot be solved in time $O^*((2-\varepsilon)^c)$.*

Proof. We describe the mentioned reduction from Set Cover to LCR in more detail. Let $(U, \mathcal{F}, r)$ be an instance of Set Cover. We want to construct a leader contributor system $S = (D, a^0, P_L, P_C)$. The main idea is as follows. 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 set cover by querying whether all elements of U have been stored by the contributors. To this end, communication is based on the elements of U. Formally, the data domain is given by

$$D = \{u, u^\# \mid u \in U\} \cup \{a^0\}.$$

For guessing r sets from the family $\mathcal{F}$, the leader P_L consists of r similar phases. Each phase starts with P_L choosing an internal transition to some set $T \in \mathcal{F}$. Once T is chosen, the leader writes a sequence of all elements $u \in T$ to the memory. Figure 8.8 illustrates the structure of P_L.

A contributor P_C of S consists of $c = n + 1$ many states: one initial state and a state to store each element $u \in U$. It is shown in Figure 8.8. When P_L writes an element $u \in T$ to the memory, there is a contributor storing this element in its states by reading u. Hence, each element that got covered by the chosen set T 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 proper set cover of U. To this end, it checks 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 set cover of U was chosen. For a formal construction and a proof of correctness we refer to Appendix B.3.7. □

8.3.2 Kernel Lower Bounds

In order to emphasize on the hardness of the problem LCR, we show that both tractable parameterizations, LCR(d, l) and LCR(c), do not admit a polynomial kernel. Hence, despite their tractability, problem kernels of these

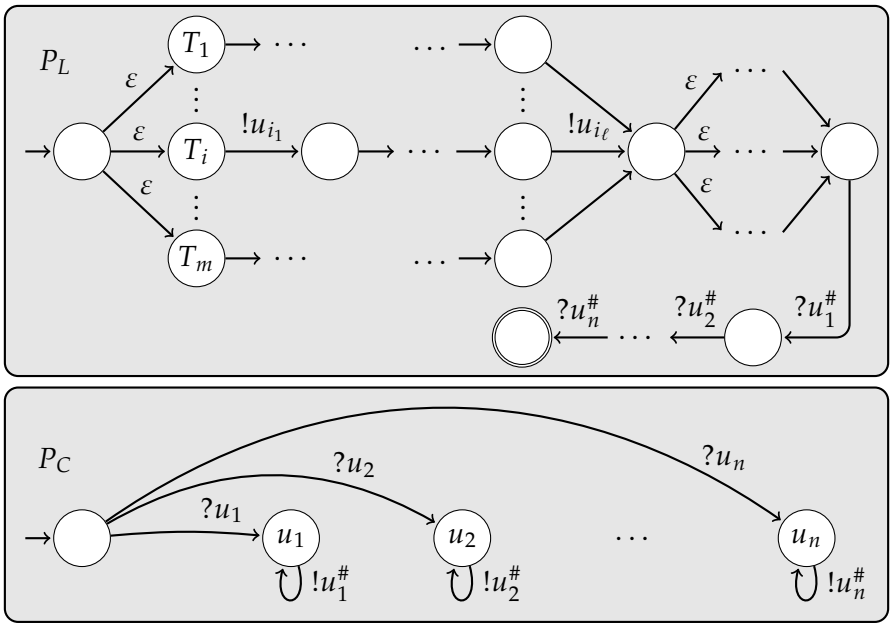


Figure 8.8: The leader P_L chooses r sets from the family $\mathcal{F} = \{T_1, \ldots, T_m\}$. If it chooses $T_i = \{u_{i_1}, \ldots, u_{i_\ell}\}$, it writes all elements of the set to the memory. The process is repeated r times. After this, P_L waits for the confirmation $u_1^\# \ldots u_n^\#$ to reach its final state. The contributor P_C reads and stores one element u_i of $U = \{u_1, \ldots, u_n\}$. It can confirm that it has stored u_i by writing $u_i^\#$.

parameterizations remain large after a preprocessing. To obtain the kernel lower bounds, we rely on the cross-composition framework from Section 5.4. Recall that a cross-composition from a problem Q into a parameterization of LCR is a polynomial-time reduction that acts as a logical OR. Given several instances of Q, it can detect whether one of these is a yes-instance. Moreover, a cross-composition keeps the parameter(s) of LCR small. If we are given t instances of Q, the parameter(s) of LCR are not allowed to depend polynomially on t. Establishing a cross-composition into LCR(d, l) and LCR(c) yields that polynomial kernels for the parameterizations are unlikely to exist.

Parameterization by Domain and Leader The kernel lower bound for the parameterization LCR(d, l) is obtained by a cross-composition from 3-SAT. The difficulty in coming up with such a cross-composition is the restriction on the size of the parameters that it needs to satisfy. In this case, this affects d and l. Both parameters are not allowed to depend polynomially on t, the number of given 3-SAT-instances. We resolve a potential polynomial dependence on t by encoding the choice of a particular instance into the contributors via a binary tree. The result is as follows.

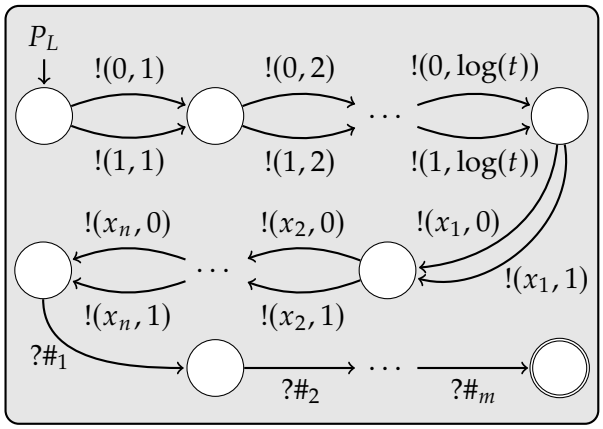


Figure 8.9: The leader P_L guesses a binary representation of an index $\ell \in [1..t]$ to pick a 3-SAT-instance φ_ℓ. Then it guesses an assignment for the variables $x_1, \ldots, x_n$. The contributors confirm that the chosen assignment satisfies the j-th clause of φ_ℓ by writing $\#_j$. The leader waits to read $\#_1 \ldots \#_m$.

Theorem 8.42. *LCR(d, l) does not admit a poly. kernel, unless* NP $\subseteq$ coNP/poly.

Proof. Before we describe the details of the cross-composition from 3-SAT into LCR(d, l), we need a polynomial equivalence relation on Boolean formulas. To this end, we employ the relation $\mathcal{R}$ from Example 5.41. Roughly, two formulas are equivalent under $\mathcal{R}$ if both encode 3-SAT-instances and if both formulas have the same number of variables and clauses.

Let $\varphi_1, \ldots, \varphi_t$ be given 3-SAT-instances, each two of them equivalent under the equivalence relation $\mathcal{R}$. This means that all φ_ℓ have the same number of clauses m and use the same set of variables $\{x_1, \ldots, x_n\}$. Hence, we may assume that φ_ℓ is of the form $\varphi_\ell = C_1^\ell \wedge \ldots \wedge C_m^\ell$.

We construct a leader contributor system $S = (D, a^0, P_L, P_C)$ and a set of final states Q_F such that S has an initialized computation where the leader reaches a final state if and only if an φ_ℓ has a satisfying assignment. Computations of S proceed in three phases. In the first phase, S chooses an instance φ_ℓ out of the given 3-SAT-instances. In the second phase, the system guesses an assignment for all variables and in the third phase, S verifies that the chosen assignment indeed satisfies φ_ℓ. While the second and the third phase do not cause a dependence of the parameters d and l on t, the first phase does. Guessing a number $\ell \in [1..t]$ and communicating it in one shot requires a data domain of size $O(t)$. This causes a polynomial dependence of d on t.

To implement the first phase of S without causing the polynomial dependence, we transmit the indices $\ell \in [1..t]$ of the 3-SAT-instances in binary. In fact, the leader P_L guesses and writes a sequence of tuples

$$(u_1, 1), \ldots, (u_{\log(t)}, \log(t))$$

with $u_i \in \{0, 1\}$ to the shared memory. This amounts to choosing an instance φ_ℓ with the binary representation $\text{bin}(\ell) = u_1 \dots u_{\log(t)}$. We illustrate the construction of the leader thread P_L in Figure 8.9.

It is the contributors' task to store the choice. Each time the leader writes (u_i, i) to the memory, 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 t many leaves, each leaf storing the index of a 3-SAT-instance φ_ℓ. Since we did not assume that t is a power of 2, there might be computations arriving at leaves that do not represent proper indices. In this case, the computation of S deadlocks immediately. Note that the number of symbols which are required to transmit the binary strings is $O(\log(t))$. The construction of a contributor thread P_C is shown in Figure 8.10.

In the second phase, S guesses an assignment for the variables. To this end, the system communicates on tuples of the form (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. A participating contributor is free to read one of the tuples. After it has read (x_i, v), it stores that variable x_i evaluates to v.

In the third phase, S performs the satisfiability check. Each contributor that is still active has stored the chosen instance φ_ℓ, a variable x_i, and its evaluation v. Assume that x_i when evaluated to v, satisfies C_j^ℓ, 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 whole string $\#_1 \dots \#_m$. If P_L can succeed, we are sure that the m clauses of φ_ℓ were satisfied by the chosen assignment. Thus, φ_ℓ is satisfiable and P_L moves to its final state.

Wrapping up, S has the required property of detecting whether one φ_ℓ is satisfiable. Moreover, we have that $d = l = O(\log(t) + n + m)$. This matches the requirements of a cross-composition. For details on the construction of S and a formal proof of correctness, we refer to Appendix B.3.8. □

Parameterization by Contributor We cross-compose the problem 3-SAT into LCR(c). This shows that the parameterization is unlikely to admit a kernel of polynomial size. The result is the following.

Theorem 8.43. *LCR(c) does not admit a polynomial kernel, unless* NP ⊆ coNP/poly.

Proof. Let $\varphi_1, \dots, \varphi_t$ be instances of 3-SAT such that each two are equivalent under the polynomial equivalence relation $\mathcal{R}$ from Example 5.41. Then, all φ_ℓ have the same number of clauses m and are defined over the same set of variables $x_1, \dots, x_n$. We may assume that $\varphi_\ell = C_1^\ell \wedge \dots \wedge C_m^\ell$.

We construct a leader contributor system $S = (D, a^0, P_L, P_C)$ that has a computation where P_L visits a final state if and only if there is an $\ell \in [1..t]$ such that φ_ℓ is satisfiable. The basic idea behind S is the following. First, the leader P_L guesses the instance φ_ℓ as well as an evaluation for the variables.

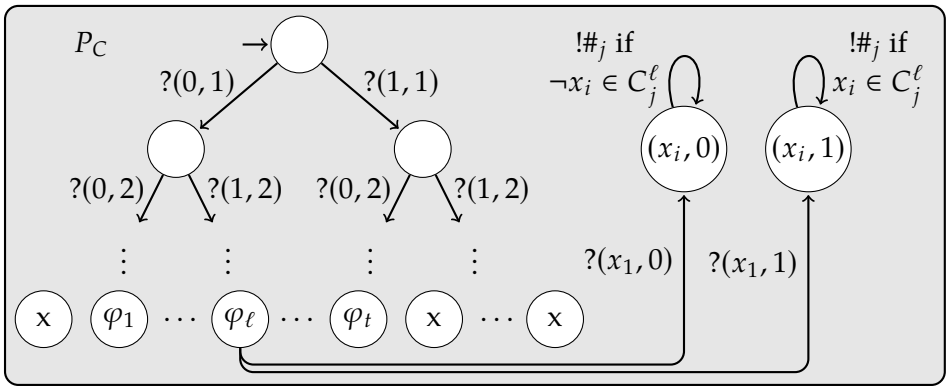


Figure 8.10: The contributor P_C first acts like a binary tree. Leaves of the tree either encode indices of $\varphi_1, \ldots, \varphi_\ell$ or binary numbers not belonging to an instance (x). Then P_C reads a variable x_i with a valuation and stores it. It can write $\#_j$ if the variable with the assignment satisfies the j-th clause of φ_ℓ.

The contributors store the latter. Then, leader and contributors verify that the chosen evaluation indeed satisfies the formula φ_ℓ.

For guessing φ_ℓ, the leader has a separate branch for each instance. Note that we can afford the size of the leader l to depend on the number of given instances t. The cross-composition only restricts the parameter c. Hence, we do not face the problem that we had in Theorem 8.42.

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

Note that the parameter c is of size $O(n)$ and does not depend on t at all. Hence, the restrictions of a cross-composition are met. We omit a formal construction of S since it is quite similar to that of Theorem 8.42. □

8.3.3 Intractability

The parameterization LCR(d, l) is FPT. This raises the question on whether already one of the parameters, d or l, suffices to obtain tractability. We answer it in the negative. In fact, both parameterizations LCR(d) and LCR(l) are W[1]-hard. This justifies our choice of parameterizations in Section 8.2 and resolves the last bit of the fine-grained complexity of LCR.

We obtain the hardness for W[1] by constructing parameterized reductions from Clique(k) to LCR(d) and LCR(l). Recall that the problem Clique(k) is

W[1]-hard by Corollary 4.15. The reductions transport the hardness to the parameterizations of LCR. We summarize in the following theorem.

Theorem 8.44. *LCR(d) and LCR(l) are both* W[1]*-hard.*

Proof. We first reduce Clique(k) to LCR(l). More precisely, we construct from an instance (G, k) of Clique in polynomial time a leader contributor system $S = (D, a^0, P_L, P_C)$ with $l = O(k)$. The system has a computation where P_L visits a final state if and only if G has a clique of size k. This meets the requirements of a parameterized reduction.

Let $G = (V, E)$. The system S operates in three phases. In the first phase, the leader chooses a clique candidate, k vertices of the graph. To this end, P_L consecutively writes symbols $(v_1, 1), (v_2, 2), \ldots, (v_k, k)$ with $v_i \in V$ to the memory. 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 P_L again writes a sequence of vertices to the memory. But this time it uses different symbols: $(w_1^\#, 1), (w_2^\#, 2), \ldots, (w_k^\#, k)$. Note that the vertices $w_i \in V$ do not have to coincide with the vertices v_i from the first phase. It is the contributor's task to verify that the new sequence coincides with the one guesses in the first phase and moreover, that it constitutes a proper clique. To this end, for each $i \in [1..k]$, the contributors P_C operate as follows. If a contributor storing (v_i, i) reads the symbol $(w_i^\#, i)$, 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 in G.

Finally, in the third phase, we need to ensure that there was at least one contributor storing (v_i, i) and that the checks from the second phase all succeeded. 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 $\#_1 \ldots \#_k$. This ensures the selection of k different vertices and that each two of them are actually adjacent.

Note that the construction of the leader requires only $l = O(k)$ many states as it only guesses $2 \cdot k$ many vertices and waits to read k symbols. Hence, we obtain the desired parameterized reduction to LCR(l). For a formal construction and a proof of correctness, we refer to Appendix B.3.9.

For proving W[1]-hardness of the parameterization LCR(d), we slightly change the above reduction. Note that in the reduction, the size of the data domain D is $|V| \cdot k$. Hence, it is not a parameterized reduction for the parameter d. The factor $|V|$ appears since leader and contributors communicate on the vertices of G. The main idea of the new reduction is to decrease the size of D by transmitting vertices in binary. To this end, fix some binary encoding $\mathrm{bin} : V \to \{0,1\}^t$ with $t = \log(|V|)$. Then, instead of a single tuple (v, i) with $v \in V$ and $i \in [1..k]$, the leader P_L writes a sequence

$$\#.(b_1, i)\#.(b_2, i)\# \ldots \#.(b_t, i).\#$$

to the memory with $\text{bin}(v) = b_1 \dots b_t$. The special padding symbol # is needed to prevent the contributors from reading a symbol (b_j, i) multiple times. Note that the new data domain only needs $O(k)$ many symbols.

The reduction then proceeds as above. The only difference is that the contributors now need to decode the binary encoding of a vertex. This is achieved by adding a binary branching tree of the contributors like in Theorem 8.42. We omit the details and refer to Appendix B.3.9. □

8.4 Fine-Grained Complexity of Liveness Verification

The *leader contributor liveness problem* (LCL) is a generalization of LCR. It asks whether the leader can reach a final state infinitely often instead of only once. We conducted a fine-grained complexity analysis of LCL and in the following, we present our results. In Section 8.4.1, we formally introduce the liveness problem. Then, in Section 8.4.2, we show how LCL can be divided into the reachability problem LCR and a cycle detection problem which we refer to as CYC. Since we have already established algorithms for LCR in Section 8.2, it is left to solve CYC. In Section 8.4.3, we formulate a polynomial-time algorithm for the problem. Finally, we compose the algorithms for LCR and the algorithm for CYC in Section 8.4.4. The composed algorithms solve LCL within the same running times as for LCR.

Note that the lower bounds for LCL naturally take over from LCR. Hence, we only need to provide upper bounds for the liveness verification problem.

8.4.1 The Problem LCL

The problem LCL is the task of deciding whether the leader satisfies a liveness specification while interacting with a certain number of contributors. Unlike LCR, this is modeled as a repeated reachability problem on the given leader contributor system: is there a computation such that the leader visits a set of final states infinitely often? So far, we only considered finite computations of leader contributor systems. To define LCL, we need to consider infinite ones.

Definition 8.45. Let $S = (D, a^0, P_L, P_C)$ be a leader contributor system with leader thread $P_L = (Q_L, OP(D), \delta_L, q_L^0)$. An *infinite computation of* S is a sequence $\sigma = c^0 \to_S c^1 \to_S \dots$ of infinitely many transitions.

Let $Q_F \subseteq Q_L$ be a set of final states. Since σ involves infinitely many configurations but the set Q_L is finite, there are states of the leader that occur infinitely often along the computation. The following set captures all final states of the leader that occur infinitely often. Note that it might be empty.

$$Fin(\sigma) = \{q \in Q_F \mid \exists^\infty i : q = \pi_L(c^i)\}$$

Given a leader contributor system $S = (D, a^0, P_L, P_C)$ and a set of final states Q_F for the leader, the *leader contributor liveness problem* (LCL) asks

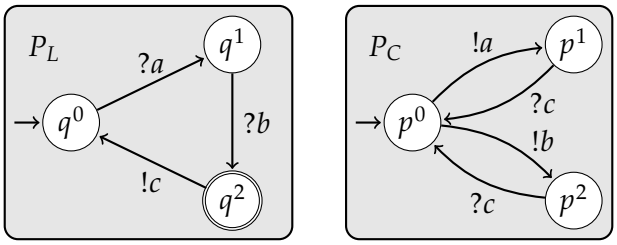


Figure 8.11: Leader contributor system $S = (D, a^0, P_L, P_C)$ over the data domain $D = \{a^0, a, b, c\}$. The leader thread P_L is given on the left hand side of the figure, the contributor P_C on the right hand side.

whether there is an initialized infinite computation σ such that the leader visits Q_F infinitely often along σ. Phrased differently, it asks whether $Fin(\sigma) \neq \emptyset$. If this is the case, we also call σ a *live computation*.

LCL
Input: An LCS $S = (D, a^0, P_L, P_C)$ and final states $Q_F \subseteq Q_L$.
Question: Is there an initialized infinite σ with $Fin(\sigma) \neq \emptyset$?

Before we consider the fine-grained complexity of LCL we illustrate the problem of finding an infinite computation with an example.

Example 8.46. Consider the leader contributor system $S = (D, a^0, P_L, P_C)$ given in Figure 8.11. Note that the leader P_L contains a final state $Q_F = \{q^2\}$. The following computation with two contributors is a live computation:

$$(q^0, a^0, p^0, p^0) \xrightarrow{!a}_S (q^0, a, p^1, p^0) \xrightarrow{?a}_S (q^1, a, p^1, p^0)$$
$$\xrightarrow{!b}_S (q^1, b, p^1, p^2) \xrightarrow{?b}_S (q^2, b, p^1, p^2) \xrightarrow{!c}_S (q^0, c, p^1, p^2)$$
$$\xrightarrow{?c}_S (q^0, c, p^0, p^2) \xrightarrow{?c}_S (q^0, c, p^0, p^0) \rightarrow_S^* (q^0, c, p^0, p^0) \rightarrow_S \dots$$

Indeed, P_L visits the final state q^2 infinitely often. Note that the computation splits into a prefix, leading to (q^0, c, p^0, p^0), and a cycle in this configuration. We will later show that each live computation can actually be decomposed in such a way and that we can exploit this fact algorithmically.

Like LCR, the problem LCL is NP-complete [143]. For a fine-grained complexity analysis of LCL, we consider the same parameters as for the reachability counterpart: the size of the data domain d, the number of states of the leader thread l, and the number of states of the contributor thread c. Since lower bounds from LCR trivially carry over to LCL, all results established

in Section 8.3 also hold for LCL. This includes lower bounds based on the ETH and on the SCON, kernel lower bounds, and intractability results. We summarize the derived lower bounds for LCL in the following corollary.

Corollary 8.47. *The problem LCL admits the following lower bounds.*

a) LCL(d) and LCL(l) are both W[1]*-hard.*

b) Unless ETH fails, LCL cannot be solved in time $2^{o((d+l)\cdot\log(d+l))}$.

c) Unless SCON fails, LCL cannot be solved in time $O^*((2-\varepsilon)^c)$ *for an* $\varepsilon > 0$.

d) Neither LCL(d, l) nor LCL(c) admit a polynomial kernel, unless NP $\subseteq$ coNP/poly.

Upper bounds do not easily transfer from LCR to LCL. Hence, to match the lower bounds on the running time for LCL stated above, we need to come up with two new algorithms. We develop these algorithms in the subsequent sections. The rough idea is to decompose LCL into the reachability problem LCR and a cycle detection problem and show that the latter can actually be solved in polynomial time. Then, the running times of the algorithms for LCR are the dominant time factors. Since LCR can be solved in time $(d+l)^{O(d+l)}$, we obtain the same running time for liveness verification.

Theorem 8.48. *The problem LCL can be solved in time* $(d+l)^{O(d+l)}$.

Similarly, since LCR can be solved in time $O^*(2^c)$, the running time carries over to LCL. The following theorem states the precise running time.

Theorem 8.49. *The problem LCL can be solved in time* $O(2^c \cdot d^2 \cdot l \cdot (l \cdot c^4 + d \cdot c^2 + d^3 \cdot l^2))$.

8.4.2 Interfaces

A live computation naturally decomposes into a prefix and a cycle. Hence to solve LCL, we need to find both, a prefix computation and a cyclic computation. However, we need to satisfy two requirements. First, cycles should be *saturated*. This means that no new contributor states are encountered along a cycle. The property is required to find cycles efficiently, by a fixed-point iteration in polynomial time. Second, prefix and cycle need to match: the prefix should lead to a configuration that the cycle loops on. Since there are infinitely many configurations, we introduce the finite domain of *interfaces*. These are capable of matching prefix and cyclic computations. Then, LCL amounts to finding a prefix and a saturated cycle that match at some interface.

Saturated Cycles We first focus on the decomposition of a live computation into a prefix and a saturated cycle. To this end, we fix some leader contributor system $S = (D, a^0, P_L, P_C)$ and a set of final states $Q_F \subseteq Q_L$. In a saturated cycle, the initial configuration already contains all contributor states that will be encountered along the cycle. We define the notion below.

Definition 8.50. Let $\tau : c \to^* c$ be a cyclic computation of S on some configuration $c \in \mathit{Conf}(S)$. Then, τ is called *saturated* if $\pi_C(c') \subseteq \pi_C(c)$ for each configuration c' visited by τ. We write $c \to^*_{sat} c$ for a saturated cycle.

The following lemma decomposes a live computation into a prefix and a saturated cycle. Its proof once again relies on the *copycat lemma* [152].

Lemma 8.51. *There is an initialized infinite computation σ with $\mathit{Fin}(\sigma) \neq \emptyset$ if and only if there is an initialized finite computation $c^0 \to^* c \to^+_{sat} c$ with $\pi_L(c) \in Q_F$.*

Proof. Given a computation of the form $c^0 \to^* c \to^+_{sat} c$ such that $\pi_L(c) \in Q_F$, we can iterate the cyclic part to obtain a computation that visits the final states Q_F infinitely often. For the other direction, let σ be an initialized infinite computation with $\mathit{Fin}(\sigma) \neq \emptyset$ which involves t contributors. Then, σ visits infinitely many configurations that contain a state from Q_F. Since t is fixed, there are only finitely many such configurations. Hence, there is a repeating configuration c with $\pi_L(c) \in Q_F$ and $c^0 \to^* c \to^+ c$.

It is left to show that we can assume the cycle to be saturated. Suppose $c \to^+ c$ is not saturated. Then there is a state $p \in Q_C$ which does not occur in c but is encountered in a configuration c' along the cycle. Let P denote the contributor that visits p in c'. We add a new contributor P^c to the computation that mimics the behavior of P. Each time P takes a transition, P^c copycats it immediately. Once P^c reaches p, it does not move any further and stays in p. We apply the procedure for each new state occurring in the cycle. After having iterated through the cycle once, we have collected all of these states and there is a contributor staying in each of them. Now we can run the cycle without discovering new states. This yields a computation of the form $d^0 \to^* d \to^+_{sat} d$ with $\pi_L(d) \in Q_F$ as required. □

Interfaces Lemma 8.51 is a first step to decompose LCL into finding a prefix and a cycle. But so far, we cannot just search for both separately. We need to ensure that the found computations can be linked at a configuration. To avoid handling explicit configurations, we introduce the notion of *interfaces*. An interface abstracts away from a configuration to its leader state, memory value, and set of contributor states. Hence, an interface can be seen as a summary of those configurations that are suitable for linking prefix and cycle.

Definition 8.52. An *interface* is a triple $I = (q, a, T) \in Q_L \times D \times \mathcal{P}(Q_C)$ consisting of a state of the leader q, a memory value a, and a set of contributor states T.

A configuration $c \in \mathit{Conf}(S)$ *matches* the interface I if $\pi_L(c) = q$, $\pi_D(c) = a$, and $\pi_C(c) = T$. We write $I(c)$ if c matches I, interpreting the interface as a predicate. The set of all interfaces is denoted by IF.

The next lemma shows that the notion of interfaces finally allows for decomposing LCL. In fact, we can search for prefix and cyclic computation separately as long as they both match the same interface. We skip the proof of the lemma and refer to Appendix B.3.10 for details.

Lemma 8.53. *Let $I \in \mathit{IF}$. There is a computation $c^0 \to^* c \to^+_{sat} c$ with $I(c)$ if and only if there are computations $d^0 \to^* d$ and $f \to^+_{sat} f$ with $I(d) \wedge I(f)$.*

Since finding prefix computations is a matter of LCR, we can apply the algorithms from Section 8.2 for this task. To solve LCL, it is therefore left to detect cycles. We formalize the problem. It takes an interface and asks for a saturated cycle on a configuration that matches the given interface.

CYC

Input: An LCS $S = (D, a^0, P_L, P_C)$ and an interface $I \in \mathit{IF}$.

Question: Is there a computation $c \to^+_{sat} c$ with $I(c)$?

We present a fixed-point iteration that solves the problem CYC in polynomial time. The precise running time is stated in the below theorem.

Theorem 8.54. *CYC can be solved in time $O(d^2 \cdot (c^2 + d^2 \cdot l^2))$.*

With Theorem 8.54, we can elaborate on the algorithmic idea to solve LCL in more detail. First, we start one of the two safety verification algorithms solving LCR, on those final states that the live computation should visit. After a slight modification, these algorithms output all interfaces witnessing prefix computations to those states. Then, we can iterate over the obtained interfaces and pass each to the algorithm solving CYC. If it finds a cycle, a live computation exists. Since CYC can be solved in polynomial time, the running times of LCR carry over to LCL — up to a polynomial factor.

8.4.3 Finding Cycles

We present an algorithm which solves CYC in time $O(d^2 \cdot (c^2 + d^2 \cdot l^2))$. This proves Theorem 8.54. The algorithm relies on a characterization of saturated cycles in terms of *stable SCC decompositions*. These are decompositions of the contributor thread into strongly connected subgraphs that are stable in the sense that they write exactly the symbols they intend to read. By setting up a suitable operator, we show that stable SCC decompositions can be found by a fixed-point iteration in the mentioned time. Technically, the iteration is

simple. It repeatedly calls Tarjan's algorithm [313] for computing SCC decompositions. Hence, the algorithm is easy to implement and shows that stable SCC decompositions are an ideal structure for detecting saturated cycles.

We also discovered that cycles can be detected by a non-trivial polynomial-time reduction to the problem of finding cycles in dynamic graphs. Although the latter can be solved in polynomial time [239], the obtained algorithm for CYC does not admit an efficient polynomial-time complexity. The reason is that the algorithm in [239] repeatedly solves linear programs that grow large due to the reduction. Compared to this method, our algorithm is more efficient and technically simpler due to being tailored to the actual problem.

Strongly Connectedness We focus on the characterization of saturated cycles in terms of stable SCC decompositions. The first step is to define the correct notion of strongly connectedness. Intuitively, a stable SCC decomposition is a decomposition of the contributor thread that can provide itself with all the symbols that a cycle along this structure may read. Hence, we link such a decomposition to a set $\Gamma \subseteq D$ of reads that are enabled. Then, we can define strongly connectedness depending on the set Γ.

To illustrate the upcoming definitions, we start with a fixed saturated cycle and generalize its properties. To this end, let $I = (q, a, T) \in IF$ be an interface and $\tau = c \rightarrow_{sat}^{+} c$ be a saturated cycle with $I(c)$. We assume that the set

$$Writes(\tau) = \{b \in D \mid d \xrightarrow{!b}_S d' \in \tau\}$$

is non-empty — τ contains at least one write. If τ contains only reads, then either a contributor or the leader run in an $?a$-loop which is easy to detect.

Let P be some contributor which moves during the cycle τ. Then, P changes its current state over time. Assume the contributor starts in a state $p \in Q_C$ and visits a state $p' \in Q_C$ during τ. Since P runs along a cycle, the contributor will eventually move from p' back to p again. This means that in the underlying contributor thread P_C, there is a path from p to p' and vice versa. Phrased differently, p and p' are strongly connected.

We generalize the observation from the fixed cycle τ to a set of enabled reads $\Gamma \subseteq D$. To this end, we first define a suitable subgraph of the contributor thread P_C, where reads are restricted to Γ. The notion of strongly connectedness that we pursue is then defined on this graph.

Definition 8.55. Let $\Gamma \subseteq D$. The *graph over T with enabled reads* Γ is the directed graph $\mathcal{G}_T(\Gamma) = (T, E(\Gamma))$. It consists of the vertices $T \subseteq Q_C$ and the set of edges $E(\Gamma)$. The latter contains an edge for each transition of P_C between states in T that is either a read of a symbol in Γ or a write of an arbitrary symbol. Formally, for two states $p, p' \in T$ we have

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

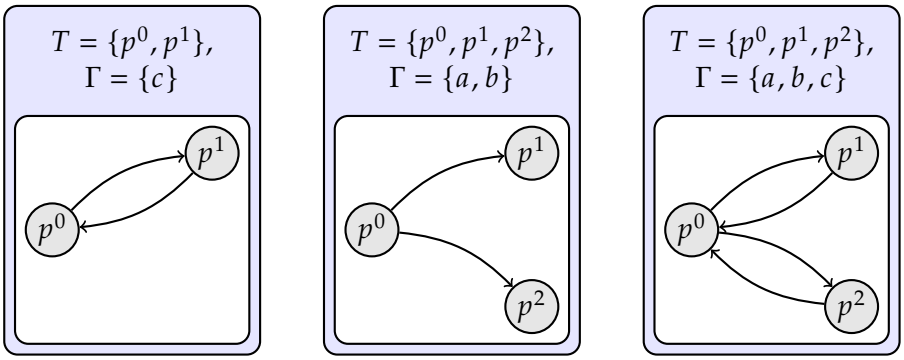


Figure 8.12: The graph $\mathcal{G}_T(\Gamma)$ for three instances of $T \subseteq Q_C$ and $\Gamma \subseteq D$. Then, $SCCdcmp_T(\Gamma)$ is the SCC decomposition of the corresponding $\mathcal{G}_T(\Gamma)$.

Let $p, p' \in T$ be two states — and thus vertices of $\mathcal{G}_T(\Gamma)$. We say that p and p' are *strongly* Γ*-connected* if p and p' are strongly connected in $\mathcal{G}_T(\Gamma)$.

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

The fixed cycle $\tau = c \rightarrow_{sat}^{+} c$ induces the graph $\mathcal{G}_T(\Gamma)$ with $\Gamma = Writes(\tau)$. The cycle runs along the SCC decomposition of this graph. Indeed, following a particular contributor P of τ, we collect all states visited along τ in a set $T_P \subseteq T$. Then T_P is strongly Γ-connected since reads performed during τ must be from the set $\Gamma = Writes(\tau)$. But this means that T_P is contained in an inclusion maximal strongly Γ-connected set, an SCC of $\mathcal{G}_T(\Gamma)$. Hence, the contributors in τ stay within the SCCs of the graph. We generalize the observation to an arbitrary set of enabled reads.

Definition 8.56. Let $\Gamma \subseteq D$ be a set of enabled reads and $V \subseteq T$ strongly Γ-connected. We call V a *strongly* Γ*-connected component* (an Γ-SCC) if it is inclusion maximal. The latter means that for each $V \subseteq V'$ with V' strongly Γ-connected, we already have $V = V'$. Note that a Γ-SCC is an SCC of $\mathcal{G}_T(\Gamma)$.

We consider the unique partition $(T_1, \ldots, T_\ell)$ of T into Γ-SCCs $T_i \subseteq T$. It is called the Γ-*SCC decomposition of* T and we denote it by $SCCdcmp_T(\Gamma)$. Note that the order of the partition is not important and that it is indeed unique. In fact, $SCCdcmp_T(\Gamma)$ is the (usual) SCC decomposition of the graph $\mathcal{G}_T(\Gamma)$.

The Γ-SCC decomposition of T can be computed by an application of Tarjan's algorithm [313] to the graph $\mathcal{G}_T(\Gamma)$. This becomes important when we compute Γ-SCC decompositions in our algorithm for solving CYC. We illustrate the notion of Γ-SCC decompositions with an example.

Example 8.57. Reconsider the simple leader contributor system illustrated in Figure 8.11. We constructed three graphs $\mathcal{G}_T(\Gamma)$ of the system, for different

instances of T and Γ. They are given in Figure 8.12. From the graphs we can determine the corresponding Γ-SCC decomposition of T. We list them below:

For $T = \{p^0, p^1\}$ and $\Gamma = \{c\}$: $SCCdcmp_T(\Gamma) = \{p^0, p^1\}$.
For $T = \{p^0, p^1, p^2\}$ and $\Gamma = \{a, b\}$: $SCCdcmp_T(\Gamma) = (\{p^0\}, \{p^1\}, \{p^2\})$.
For $T = \{p^0, p^1, p^2\}$ and $\Gamma = \{a, b, c\}$: $SCCdcmp_T(\Gamma) = \{p^0, p^1, p^2\}$.

Stability The second step to the characterization of saturated cycles in terms of stable SCC decompositions is the correct definition of stability. Intuitively, an Γ-SCC decomposition is stable if it can provide the enabled reads Γ as writes. Phrased differently, a cyclic computation on the strongly connected components can generate all the writes that it needs and thus, keep itself live. This requires a formal definition of the writes that an Γ-SCC decomposition can provide. To this end, recall that we fixed an interface $I = (q, a, T) \in IF$.

Definition 8.58. Let $\Gamma \subseteq D$ and $SCCdcmp_T(\Gamma) = (T_1, \dots, T_\ell)$. The *writes of the* Γ*-SCC decomposition* $(T_1, \dots, T_\ell)$, denoted by $Writes(T_1, \dots, T_\ell)$, is the set of all symbols that occur as writes either between the states of an T_i or in a cycle $q \rightarrow_L^* q$ of the leader while preserving the memory content a. Formally, we set $Writes(T_1, \dots, T_\ell) = Writes_C(T_1, \dots, T_\ell) \cup Writes_L(T_1, \dots, T_\ell)$ where

$$Writes_C(T_1, \dots, T_\ell) = \{b \in D \mid p \xrightarrow{!b}_C p' \text{ with } p, p' \in T_i\} \text{ and}$$
$$Writes_L(T_1, \dots, T_\ell) = \{b \in D \mid \exists u, v \in OP(D)^* : (q, a) \xrightarrow{u.!b.v}_{L'} (q, a)\}.$$

Here, $\rightarrow_{L'}$ denotes the transition relation of the automaton $P_{L'}$, a restriction of the leader P_L to reads within $Writes_C(T_1, \dots, T_\ell)$. The automaton also keeps track of the memory content. Formally, we define it to be the tuple $P_{L'} = (Q_L \times D, OP(D), \delta_{L'}, (q_L^0, a^0))$ with transitions

$$(s, b) \xrightarrow{!b'}_{L'} (s', b') \qquad \text{if } s \xrightarrow{!b'}_L s',$$
$$(s, b) \xrightarrow{?b}_{L'} (s', b) \qquad \text{if } s \xrightarrow{?b}_L s' \text{ and } b \in Writes_C(T_1, \dots, T_\ell),$$
$$(s, b) \xrightarrow{\varepsilon}_{L'} (s, b') \qquad \text{if } b' \in Writes_C(T_1, \dots, T_\ell).$$

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

Before we elaborate on properties concerning the writes of an Γ-SCC decomposition, we illustrate the notion with an example.

Example 8.59. We determine the writes of the Γ-SCC decompositions of Example 8.57. Recall that the underlying leader contributor system is given in Figure 8.11. Fix an interface $I = (q^0, c, T)$ with $T = \{p^0, p^1, p^2\}$. We computed two Γ-SCC decompositions of T, for varying Γ. For $\Gamma = \{a, b\}$ we have

$SCCdcmp_T(\Gamma) = (\{p^0\}, \{p^1\}, \{p^2\})$. Therefore, we obtain

$$Writes_C(SCCdcmp_T(\Gamma)) = \emptyset \text{ and}$$
$$Writes_L(SCCdcmp_T(\Gamma)) = \emptyset.$$

Note that each component in the Γ-SCC decomposition consists of only a single states and there is no write transition in P_C that loops on such a state. There are no writes that the contributors could provide. Moreover, since $?a$ and $?b$ are not enabled in P_L, there is no write transition that occurs on a loop in q^0. This means that also the leader cannot provide any writes.

For $\Gamma' = \{a, b, c\}$ we have that $SCCdcmp_T(\Gamma') = \{p^0, p^1, p^2\}$ consists of only a single component. Consequently, we obtain

$$Writes_C(SCCdcmp_T(\Gamma')) = \{a, b\} \text{ and}$$
$$Writes_L(SCCdcmp_T(\Gamma')) = \{c\}.$$

In the chosen example, the writes of the decompositions behave monotonically. We have $\Gamma \subseteq \Gamma'$ and in fact, $\emptyset = Writes(SCCdcmp_T(\Gamma))$ is contained in the set $Writes(SCCdcmp_T(\Gamma')) = \{a, b, c\}$. As we will see, this is true in general.

The next lemma makes the monotonicity of writes more precise. This fact will become important when we formulate an operator for finding stable SCC decompositions. We skip the technical proof and refer to Appendix B.3.11.

Lemma 8.60. *Let $\Gamma \subseteq \Gamma' \subseteq D$ be two sets of enabled reads. We have*

$$Writes(SCCdcmp_T(\Gamma)) \subseteq Writes(SCCdcmp_T(\Gamma')).$$

When we go back to the saturated cycle $\tau = c \to^+_{sat} c$, reads in the cycle are always preceded by corresponding writes. Phrased differently, the writes $Writes(\tau)$ performed along τ provide all symbols that are needed for reading. The following generalizes this property to Γ-SCC decompositions.

Definition 8.61. Let $\Gamma \subseteq D$. The Γ-SCC decomposition $SCCdcmp_T(\Gamma)$ of T is called *stable* if it provides Γ as its writes, meaning $Writes(SCCdcmp_T(\Gamma)) = \Gamma$.

Note that the definition asks for an equality instead of an inclusion. The reason is that we can express stability as a fixed point of a suitable operator. We give an example of stable and non-stable decompositions.

Example 8.62. Recall the SCC decomposition and their writes from Example 8.59. For $\Gamma = \{a, b\}$, we have $Writes(SCCdcmp_T(\Gamma)) = \emptyset$. Hence, the decomposition $SCCdcmp_T(\Gamma)$ is clearly not stable. For $\Gamma' = \{a, b, c\}$, we have $Writes(SCCdcmp_T(\Gamma')) = \{a, b, c\}$ which shows stability of $SCCdcmp_T(\Gamma')$.

Characterization We proceed with the desired characterization. There is a saturated cycle if and only if there exists a stable SCC decomposition. This is a major step towards the polynomial-time algorithm for the problem CYC. As we will see, the existence of a stable SCC decomposition can be determined by a fixed-point iteration. Recall that we fixed the interface $I = (q, a, T) \in IF$.

Lemma 8.63. *There is a saturated cycle $\tau = c \to_{sat}^{+} c$ with $I(c)$ if and only if there exists a non-empty subset $\Gamma \subseteq D$ such that $SCCdcmp_T(\Gamma)$ is stable.*

Proof. Assume the existence of a saturated cycle τ. Our candidate set of enabled reads is $\Gamma = Writes(\tau)$. We have that $\Gamma \subseteq Writes(SCCdcmp_T(\Gamma))$ since each write occurring in τ is either carried out by a contributor within a single component or by the leader which performs the loop $q \to_L^* q$. If equality $\Gamma = Writes(SCCdcmp_T(\Gamma))$ holds, then $SCCdcmp_T(\Gamma)$ is stable and Γ is the desired set. Otherwise, we have $\Gamma \subsetneq Writes(SCCdcmp_T(\Gamma))$.

In the latter case, we consider $\Gamma' = Writes(SCCdcmp_T(\Gamma))$ instead of Γ. Since $\Gamma \subseteq \Gamma'$, we can apply Lemma 8.60 and obtain that

$$\Gamma' = Writes(SCCdcmp_T(\Gamma)) \subseteq Writes(SCCdcmp_T(\Gamma')).$$

Iterating this process yields a sequence of sets $(\Gamma_i)_{i\in\mathbb{N}}$ which is defined by $\Gamma_{i+1} = Writes(SCCdcmp_T(\Gamma_i))$ and which is strictly increasing, $\Gamma_i \subsetneq \Gamma_{i+1}$. The sequence is finite since $\Gamma_i \subseteq D$ for all i. Hence, there is a last set Γ_d which necessarily fulfills $\Gamma_d = \Gamma_{d+1}$. But this means $\Gamma_d = Writes(SCCdcmp_T(\Gamma_d))$.

For the other direction, we give the idea behind the proof. A formal construction is provided in Appendix B.3.12. Let $\Gamma \subseteq D$ be a set of enabled reads such that $SCCdcmp_T(\Gamma)$ is stable. We do not immediately construct a saturated cycle, but a *balanced* computation $\rho = c \to^{+} d$ where d and c coincide up to the order of the contributor states. Phrased differently, d is a permutation of c. Moreover, ρ is saturated in the above sense. There are now new contributor states discovered along ρ. Since d contains the same contributor states as c, ρ can also be started in the configuration d. This yields $c \to^{+} d \to^{+} d'$ where d' is a new permutation of c. Since there are only finitely many such permutations, we eventually get a computation of the form $c \to^{*} e \to_{sat}^{+} e$ and hence, the desired saturated cycle.

Let $SCCdcmp_T(\Gamma) = (T_1, \dots, T_\ell)$. To construct ρ, we first fix the behavior of the leader. We pick a run ρ_L of $P_{L'}$ from (q, a) to (q, a) that, on its way, writes all the symbols in $Writes_L(T_1, \dots, T_\ell)$. Note that such a run exists. We let t denote its length. To execute ρ_L properly, we have to provide the symbols that it reads on the way. Since these are from the set $Writes_C(S_1, \dots, S_\ell)$, we construct supporting runs of the contributors providing the required symbols.

Let $b \in Writes_C(T_1, \dots, T_\ell)$. Then there is a write transition $p \xrightarrow{!b}_C p'$ with states $p, p' \in T_i$. The idea is to keep enough copies of contributors in the source state p to provide b whenever the leader needs it. However, to obtain a balanced computation, we have to transfer the amount of contributors that

moved from p to p' back to p. Since T_i is strongly Γ-connected, there is a path $p' \to^* p$ in $\mathcal{G}_T(\Gamma)$. Hence, there is a run on P_C from p' back to p reading only symbols from Γ. With the above transition and the mentioned run of P_C, we get a cyclic run on p. We refer to it as $\mathit{cycle}(p)$.

In the configuration c, we keep for each symbol $b \in \mathit{Writes}_C(T_1, \ldots, T_\ell)$ with source state p_b exactly $t+1$ copies of all states occurring in $\mathit{cycle}(p_b)$. We assume the contributors in c are grouped into blocks $B_b(i)$ for $i \in [1..(t+1)]$. Each block $B_b(i)$ simulates the cyclic run $\mathit{cycle}(p_b)$.

When the leader starts to move along ρ_L, it might need to read a symbol b. Then, there is a block $B_b(i)$ which can provide b. One contributor executes the corresponding write transition of $\mathit{cycle}(p_b)$. To balance the block again, the remaining contributors execute the remaining transitions of $\mathit{cycle}(p_b)$. The idea is that each remaining contributor moves one state further in the cycle. Executing write transitions is simple. These can be just be ignored by other participants. Read transitions in the block are handled in two different ways.

Reads within the set $\mathit{Writes}_C(T_1, \ldots, T_\ell)$ are executed in a special initial phase. To this end, we let the $(t+1)$-st copies of the cycles move. These provide the corresponding symbol and all other blocks read.

Reads within the set $\mathit{Writes}_L(T_1, \ldots, T_\ell)$ are provided by the leader on ρ_L. Since the leader traverses through all symbols in $\mathit{Writes}_L(T_1, \ldots, T_\ell)$ at least once, there is a transition which writes the required symbol for the first time. This write is used to synchronize with all blocks simultaneously. The described computation is indeed balanced. □

Operator To solve CYC it is left to check the existence of a set $\Gamma \subseteq D$ with a stable SCC decomposition. Following the definition of stability, we can express Γ as a fixed point which can be computed by a Kleene iteration [330] in polynomial time. We define the suitable operator.

Definition 8.64. The operator $\mathit{Writes}_{SCC} : \mathcal{P}(D) \to \mathcal{P}(D)$ maps a given set X to the writes of the X-SCC decomposition of T. Formally,

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

We have already shown that the writes of SCC decompositions behave monotonically. Moreover, we can compute SCC decompositions efficiently by applying Tarjan's algorithm [313]. These properties carry over to the operator, it is monotone and can be evaluated in polynomial time.

Lemma 8.65. *For $X \subseteq X'$ subsets of D, we have $\mathit{Writes}_{SCC}(X) \subseteq \mathit{Writes}_{SCC}(X')$. Moreover, $\mathit{Writes}_{SCC}(X)$ can be evaluated in time $O(d \cdot (c^2 + d^2 \cdot l^2))$.*

Proof. Monotonicity follows immediately from Lemma 8.60. For the evaluation, let X be given. We apply Tarjan's algorithm to the graph $\mathcal{G}_T(X)$ in order to compute the X-SCC decomposition $\mathit{SCCdcmp}_T(X)$. It is left to compute the

writes of the decomposition. This requires a simple automata construction. For details we refer to Appendix B.3.13. □

The following lemma summarizes our reasoning. It states that the non-trivial fixed points of $Writes_{SCC}$ are precisely the sets with a stable SCC decomposition. Hence, searching for a saturated cycle amounts to searching for a non-trivial fixed point. The proof follows from the definition of the operator.

Lemma 8.66. *Let $\Gamma \subseteq D$ be a non-empty subset. Then, we have*

$$\Gamma = Writes_{SCC}(\Gamma) \text{ if and only if } SCCdcmp_T(\Gamma) \text{ is stable.}$$

Fixed-Point Algorithm We describe the fixed-point iteration for finding a stable SCC decomposition and summarize the algorithm for solving CYC. It is stated as Algorithm 8.3 below. To find a set Γ with a stable SCC decomposition, the algorithm employs a Kleene iteration that computes the greatest fixed point of $Writes_{SCC}$. It starts from $\Gamma = D$, the largest element of the underlying lattice $\mathcal{P}(D)$. At each step, the iteration evaluates $Writes_{SCC}(\Gamma)$ by invoking the algorithm from Lemma 8.65 as a subroutine.

Algorithm 8.3 Saturated Cycle Finding

Input: An LCS $S = (D, a^0, P_L, P_C)$ and an interface $I \in IF$.
Output: *True*, if there is a cycle $c \rightarrow_{sat}^{+} c$ with $I(c)$. *False* otherwise.

1: Set $\Gamma = D$ and $\Gamma' = \emptyset$.
2: **while** $\Gamma \neq \Gamma'$ **do** // Computation of greatest fixed point.
3: Set $\Gamma' = \Gamma$
4: Compute $\Gamma = Writes_{SCC}(\Gamma)$ with algorithm from Lemma 8.65
5: **end while**
6: **if** $\Gamma = \emptyset$ **then**
7: return *false* // Exclude trivial solution.
8: **end if**
9: return *true*

Algorithm 8.3 terminates after at most d iterations of its loop since at least one element is removed from the set Γ each iteration. Since evaluating $Writes_{SCC}(\Gamma)$ once takes time $O(d \cdot (c^2 + d^2 \cdot l^2))$, the algorithm has a total running time of $O(d^2 \cdot (c^2 + d^2 \cdot l^2))$ which matches Theorem 8.54. Note that correctness of Algorithm 8.3 follows from Lemma 8.63 and Lemma 8.66.

8.4.4 Liveness Verification Algorithms

Recall that Lemma 8.53 uses interfaces to decompose the problem LCL into finding a prefix and finding a cycle. We already solved the latter problem with the algorithm for CYC. For the former, we would like to apply the algorithms

for LCR from Section 8.2. But there is a subtlety. While the two algorithms can determine whether the leader can reach a final state, they do not output proper interfaces that witness reachability. However, the algorithm for CYC requires these interfaces as its input. Hence, we need to adjust the two algorithms for LCR, so that they output the corresponding interfaces.

Fix a leader contributor system $S = (D, a^0, P_L, P_C)$ and a set of final states $Q_F \subseteq Q_L$. We define the interfaces that are reachable via a corresponding prefix computation. Once these interfaces are computed by an algorithm solving LCR, we can pass them to the algorithm for CYC. The definition limits to *increasing* prefix computations that do not lose contributor states along the way. Note that we can always assume an increasing prefix when solving LCL. We provide a proof of this fact in Appendix B.3.14.

Definition 8.67. An initialized computation $c^0 \to_S c^1 \to_S \cdots \to_S c^\ell$ is called *increasing* if $\pi_C(c^i) \subseteq \pi_C(c^{i+1})$ for each i. We denote it by $c^0 \to_{inc} c^\ell$. An interface $I = (q, a, T) \in IF$ is called *reachable* if there exists an initialized increasing computation $c^0 \to^*_{inc} c$ such that c matches I, formally $I(c)$.

Assume we have computed the set of reachable interfaces. Then we can solve LCL. On each interface $I = (q, a, T)$ with $q \in Q_F$ that is contained in the set, we start the algorithm for CYC. If it detects a cycle, we found a live computation where the leader loops on q. Otherwise, we go to the next reachable interface containing a final state of the leader.

Algorithm 8.4 Leader Contributor Liveness

Input: An LCS $S = (D, a^0, P_L, P_C)$, a set of final states $Q_F \subseteq Q_L$.
Output: *True*, if $\exists$ initialized σ with $Fin(\sigma) \neq \emptyset$. *False* otherwise.

```
1: Compute set 𝓘 of reachable interfaces with an algorithm from Lemma 8.68
2: for I = (q, a, T) ∈ 𝓘 with q ∈ Q_F do  // Iterate over reachable interfaces.
3:    Run Algorithm 8.3 on input (S, I).  // Algorithm for CYC.
4:    if result is true then
5:       return true  // Prefix and cycle found.
6:    end if
7: end for
8: return true  // No interface admits a cycle.
```

Hence, it is left to compute the set of reachable interfaces. We can modify the two algorithms for LCR to output the set. This will not increase the running times of these algorithms. The next lemma states the modified algorithms. We provide a proof in Appendix B.3.15.

Lemma 8.68. *The set of reachable interfaces can be computed in time*

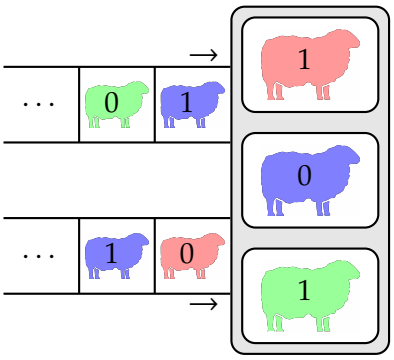
$$(d + l)^{O(d+l)} \text{ or } O(2^c \cdot c^4 \cdot d^2 \cdot l^2).$$

With Lemma 8.68 at hand, we can finally state the complete liveness verification algorithm solving LCL. It is given in Algorithm 8.4 and follows the above idea. We iterate over all reachable interfaces that contain a final state of the leader and pass each of them to the algorithm for solving CYC until we find a cycle. Correctness of the approach is proven in Appendix B.3.16.

It is left to determine the running time. It depends on which algorithm we use to compute the set $\mathcal{I}$ of reachable interfaces. If we decide for the $(d+l)^{O(d+l)}$-time algorithm from Lemma 8.68 then the size of $\mathcal{I}$ is bounded by this running time. This means that the iteration over all interfaces $I \in \mathcal{I}$ takes at most $(d+l)^{O(d+l)}$ many steps. Each step calls the algorithm for CYC which runs in polynomial time. Hence, altogether we obtain that Algorithm 8.4 takes time $(d+l)^{O(d+l)}$. This proves Theorem 8.48.

Now assume we employ the $O(2^c \cdot c^4 \cdot d^2 \cdot l^2)$-time algorithm for computing the set $\mathcal{I}$. Since $\mathcal{I} \subseteq IF$, it is of size at most $2^c \cdot d \cdot l$. This means that the iteration over all $I \in \mathcal{I}$ takes at most $2^c \cdot d \cdot l$ steps. Like above, each step calls the algorithm for CYC which runs in time $O(d^2 \cdot (c^2 + d^2 \cdot l^2))$. Summing up, we obtain a total running time of the form $O(2^c \cdot d^2 \cdot l \cdot (l \cdot c^4 + d \cdot c^2 + d^3 \cdot l^2))$ which matches the upper bound stated in Theorem 8.49.

9. Memory Consistency



Consistency algorithms are a central tool in the verification of shared-memory implementations. Given an execution of a concurrent program over a shared memory, such an algorithm checks whether the execution respects the *consistency model* of the memory. A *memory (consistency) model* defines when changes in the memory performed by one thread are visible to other threads. Prominent examples are *Sequential Consistency* (SC) and *Total Store Order* (TSO). The given execution consists of multiple sequences, one for each thread, of read and write events. Formally, a consistency algorithm then checks whether the events in the execution can be arranged in an interleaving that satisfies the axioms of the memory model. For most models, checking consistency is NP-complete and hence, consistency algorithms require exponential running time. This makes the problem appealing for more detailed complexity analyses. While such analyses have been conducted, most of them focus on proving NP-hardness or membership in P under the assumption that some parameters of the problem are constant. But none of these analyses yields FPT-algorithms.

We developed a framework which provides provably optimal consistency algorithms for various memory models. The framework is based on a universal consistency problem which can be instantiated by different models. We construct an algorithm for the problem running in time $O^*(2^k)$, where k is the number of write events in the given execution. Each instance of the framework then admits an $O^*(2^k)$-time consistency algorithm. By applying the framework, we obtain corresponding algorithms for SC, TSO, PSO, and RMO. Moreover, we show that the obtained algorithms for SC, TSO, and PSO are optimal. Parts of the upcoming text have been published in our work [98]. This chapter is an extension of the work.

9.1 Testing Consistency of Shared Memories

An implementation of a shared memory typically promises consistency guarantees to programmers. These guarantees describe the intended behavior of the shared memory and, for instance, ensure that write accesses are con-

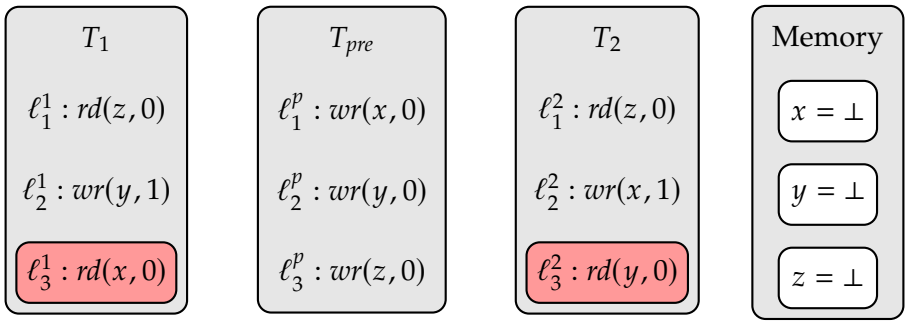


Figure 9.1: Execution of threads T_{pre}, T_1, T_2 over the variables x, y, z. The operation $rd(z,0)$ is a read access on variable z which loads the value 0. Similarly, $wr(y,1)$ writes 1 to y. The red marked operations are critical sections. Both threads should not enter them simultaneously.

sumed in the issued order. Programmers developing software over the shared memory can rely on the consistency guarantees and tune their software accordingly. Providing correct consistency guarantees is difficult. One needs to represent the memory by simple guarantees that help the programmer to use specific features of the memory. Moreover, due to the complex and performance-oriented design, implementing shared memories is prone to errors and implementations may actually fail to provide the promised guarantees. Consistency algorithms help to overcome both obstacles. They take an execution of a concurrent program over the shared memory and check whether the execution respects the guarantees promised by the memory.

Consistency guarantees are described in the form of memory models. These formulate axioms on how accesses to the memory can be arranged or reordered. Two important memory models are the basic model *Sequential Consistency* (SC) by Lamport [251] and the model *Total Store Order* (TSO) by Sparc [305, 306] which is often applied to simulate the x86-architecture. An execution of a concurrent program over the shared memory is given by sequences of read and write accesses — one for each thread. Checking consistency then amounts to checking whether a given execution can be arranged in an interleaving that satisfies the axioms of a particular memory model.

We consider an example. Figure 9.1 shows an execution of a program involving the three threads T_1, T_2, and T_{pre} over the variables x, y, and z. An operation $\ell_2^1 : wr(y,1)$ consists of the label ℓ_2^1 and the memory access $wr(y,1)$. In this case, it is a write of the value 1 to the variable y. Similarly, $rd(y,0)$ tries to read the value 0 from y. Let us assume that the red-marked operations $\ell_3^1 : rd(x,0)$ and $\ell_3^2 : rd(y,0)$ are critical sections of T_1 and T_2. The threads should not enter them at the same time.

At first sight, this seems impossible. Assuming that the thread T_{pre} starts

the execution, it sets all variables to 0. Indeed, its execution

$$wr(x,0).wr(y,0).wr(z,0)$$

yields the memory content $x = 0$, $y = 0$, and $z = 0$. Note that T_{pre} must run before the two other threads can even start since both require to read $z = 0$ initially. If after T_{pre} has completed the thread T_1 proceeds, it sets variable y to 1 with its write $\ell_2^1 : wr(y,1)$. This already prevents T_2 from entering its critical section since it cannot execute $\ell_3^2 : rd(y,0)$ anymore and there is no write on y that could set it back to 0 again. Vice versa, if T_2 proceeds after T_{pre}, the thread sets x to 1 which prevents T_1 from entering its critical section. Hence, the two writes $\ell_2^1 : wr(y,1)$ and $\ell_2^2 : wr(x,1)$ enforce mutual exclusion and avoid that both threads enter their critical section simultaneously. This construction is also referred to as Dekker's mutex [132].

In the example we assumed that accesses to the memory are atomic. If a write $\ell_2^1 : wr(y,1)$ is issued, it gets pushed to the memory and is executed in one atomic step. Similarly, each read immediately loads the currently stored value of the desired variable from the memory. No other operations can interfere. This behavior is formalized in the memory model SC. Hence the example shows that, under SC, mutual exclusion works as expected. While SC is a rather intuitive way of imagining the behavior of a memory, it is not what is typically implemented on an x86-architecture. The reason is that atomic accesses to the memory are quite expensive and a high-performance system cannot afford to execute them each time a programmer expects the system to do so. Instead, atomic memory accesses should be relaxed to more liberal accesses that allow for faster interaction with the shared memory.

A relaxed memory model is TSO. Instead of considering writes and reads atomic actions, TSO assumes that threads have a FIFO buffer into the memory, as depicted in Figure 9.2. Once a thread arrives at a write, the write gets pushed into the buffer. Subsequent writes are appended in FIFO-manner. At some point, the memory decides that a certain amount of writes from a buffer gets flushed. It changes the stored values of the affected variables accordingly. Read accesses of a thread either load their value from the memory or, if a write to the same variable is present in the buffer, perform an *early read*. The latter loads the value from the last write to the variable in the buffer.

Let us consider the execution from Figure 9.1 under the relaxed memory model TSO. Before the threads T_1 and T_2 can begin, T_{pre} first needs to set the variable z to 0. Hence, T_{pre} needs to start. The thread performs three writes. Under TSO, each gets pushed into the buffer in FIFO-order. This is illustrated on the left hand side of Figure 9.2. Once the memory flushes the buffer and executes all three writes, we have the memory valuation $x = 0$, $y = 0$, and $z = 0$. Now the threads T_1 and T_2 can start their execution. Assume T_1 proceeds. Its read $\ell_1^1 : rd(z,0)$ is loaded from the memory. Then the write $\ell_2^1 : wr(y,1)$ gets pushed into the buffer. Finally, the read $\ell_3^1 : rd(x,0)$

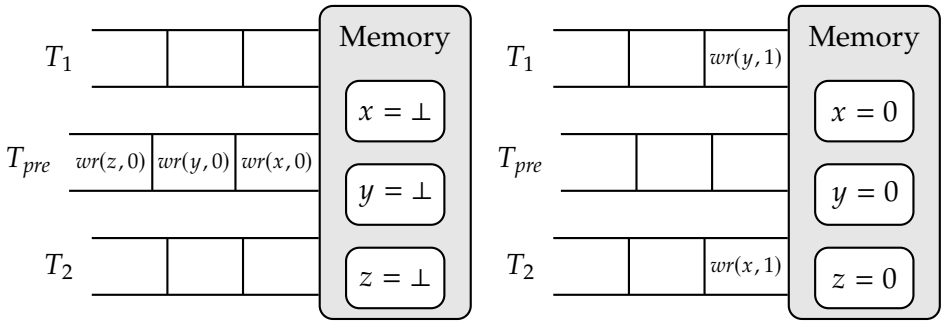


Figure 9.2: Two configurations of the memory with buffers for each thread. On the left hand side, T_{pre} pushed all its writes into the buffer. On the right hand side, the memory has completely flushed the buffer of T_{pre}. The threads T_1 and T_2 pushed their writes into the corresponding buffers.

is loaded from the memory since the buffer of T_1 does not contain a write to x. The thread T_1 reaches its critical section. Now it is T_2's turn. The thread performs its read $\ell_1^2 : rd(z, 0)$ by loading the value from the memory and subsequently pushes its write $\ell_2^2 : wr(x, 1)$ into the buffer. The resulting situation is illustrated on the right hand side of Figure 9.2. Note that the memory still holds the values $x = 0$, $y = 0$, and $z = 0$ since it did not flush the buffer of T_1 yet. This means that T_2 can perform its read $\ell_3^2 : rd(y, 0)$ by loading from the memory. Note that T_2 does not access the buffer of T_1. Hence, T_2 cannot perform an early read and loads the value of y from the memory which finally lets T_2 enter its critical section.

The example shows that mutual exclusion breaks under TSO. Hence, relaxing the notion of consistency may lead to unexpected behavior. In general one has to find a trade-off between performance and programmability. If the latter is too strict, the performance will drop but if we relax the guarantees by too much, the shared memory may imply unwanted errors. To avoid these errors, it is important to test whether the consistency guarantees, or the memory model, promised by a shared memory are actually fulfilled.

The problem of checking consistency has received considerable attention in the literature [62, 77, 98, 180, 183, 326, 336]. We are interested in its complexity. It mostly depends on the considered memory model and for most models, the problem is NP-complete. We give a more detailed overview.

9.1.1 Classical Complexity Results

Gibbons and Korach were the first to study the complexity of consistency checking [183]. They focused on the memory model SC [251] and proved that checking consistency in this setting is NP-complete. Moreover, the authors considered several restrictions of the problem and showed that even with a

constant number of threads, the problem remains NP-complete. The complexity of checking consistency under the SPARC memory models [305, 306] TSO, *Partial Store Order* (PSO), and *Relaxed Memory Order* (RMO) was investigated by Cantin, Lipasti, and Smith in [77]. The authors showed that, like for SC, checking consistency for these models is NP-complete. Besides the SPARC models, the work also states NP-completeness of consistency checking under *Processor Consistency* [7, 192] and *Alpha* [104]. Furbach, Meyer, Schneider, and Senftleben [180] extended the NP-completeness to almost all models appearing in the Steinke-Nutt hierarchy [307], a hierarchy for comparing memory models based on the behavior that they allow for. This yields NP-completeness results for memory models like *Causal Consistency* (CC) [250], *Pipelined RAM* (PRAM) [256], and *Cache Consistency* [192]. The complexity of checking consistency under CC was further investigated by Bouajjani, Enea, Guerraoui, and Hamza [62]. The authors found that checking consistency under variants of CC is NP-complete as well.

But there are also polynomial-time consistency algorithms. For the memory model LOCAL [203], consistency can be checked in polynomial time [180]. Moreover, also *Cache Consistency* and PRAM allow for polynomial-time consistency checks if certain parameters of the problem are assumed to be constant [180]. A further assumption that sometimes allows for checking consistency in polynomial time is *data independence* [43, 62, 336]. In fact, the behavior of a shared-memory implementation or a database should not depend on precise values used in applications [3, 332]. One may therefore assume that in a given execution, a value is written to a variable at most once. The assumption leads to two polynomial-time consistency algorithms, namely for CC [62] and for PRAM [326]. However, for the models SC, TSO, and PSO, the NP-hardness carries over to the data-independent case [180, 183].

Related to the problem of checking consistency is checking *robustness*. In its full generality, this is the question whether the semantics of a program changes when certain aspects like the memory model are altered. This can for instance mean that a program running under garbage collection shows the same behavior when run under safe memory reclamation [331]. In the context of memory models, robustness is often checked according to a particular model. Then, instead of only a single execution, one needs to verify that all executions are consistent under the model. For SC and CC, this is undecidable [16, 62]. Under data-independence, it becomes decidable for CC [62]. A decidability result [63] was also obtained for verifying *Eventual Consistency* [314]. Robustness can also be checked *against* a particular model. Checking robustness against TSO refers to checking whether the behavior of a program under TSO and the behavior under SC coincide. The problem is known to be PSPACE-complete [64]. A similar result is known for checking robustness against POWER [131] and Partitioned Global Address Spaces [76].

Problem	Upper Bound	Lower Bound
MM-Cons(k)	$O^*(2^k)$	-
SC-Cons(k)	$O^*(2^k)$	$2^{o(k)}$
TSO-Cons(k)	$O^*(2^k)$	$2^{o(k)}$
PSO-Cons(k)	$O^*(2^k)$	$2^{o(k)}$
RMO-Cons(k)	$O^*(2^k)$	-

Table 9.1: Fine-grained complexity of checking consistency parameterized by the number of write operations k. Main results are the algorithm for MM-Cons and the lower bounds for checking consistency under SC, TSO, PSO.

9.1.2 Framework for Consistency Algorithms

While the classical complexity of checking consistency is well understood, only little is known about the fine-grained complexity of the problem. We take a first step towards the fine-grained complexity in the data-independent case. Let SC-Cons denote the consistency problem for SC under the assumption of data independence. Like mentioned above, SC-Cons is NP-complete. The problem offers various parameters for an analysis and for some of them, intractability is already known. For instance, parameterizing in the number of threads leads to W[1]-hardness [270], taking the maximum number of operations per thread as a parameter is not in XP [183], and parameterizing by the size of the data domain is not in XP as well, as we will see later.

We focus on the parameter k — the number of write operations in the given execution. This is not only due to the intractability of other parameters but also due to polynomial preprocessings [336] that manage to reduce the number of write operations in instances. Performing such a reduction beforehand limits the size of k and speeds-up a potential FPT-algorithm in the parameter.

Contribution We provide a framework which yields provably optimal consistency algorithms for various memory models. The obtained algorithms run in time $O^*(2^k)$. We demonstrate the applicability of the framework by giving the corresponding algorithms for the models SC, TSO, PSO, and RMO.

We summarized our contribution in Table 9.1. Our framework is based on a universal consistency problem that can be instantiated by a memory model of choice. We denote it as MM-Cons. Our main result is an algorithm for MM-Cons which runs in time $O^*(2^k)$. Then, any instance by a memory model automatically admits an $O^*(2^k)$-time consistency algorithm. For the formulation of the problem and the framework, we rely on the axiomatic language for specifying consistency CAT, developed by Alglave [11] and Alglave,

Maranget, and Tautschnig [12]. Roughly, CAT formulates memory models in terms of relations and acyclicity constraints among them. Then, checking consistency amounts to finding a particular *store order* [336] on the write operations that satisfies the acyclicity constraints of the considered model.

For solving MM-Cons, we show that instead of a store order we can also find a total order on the write operations that satisfies similar acyclicity constraints. The advantage is that total orders are algorithmically simpler to find. Then, we develop a notion of *snapshot orders* that mimics total orders on subsets of write events. This allows for shifting from the relation-based domain of the problem MM-Cons to the subset lattice of write operations. On this lattice, we can perform a dynamic programming which builds up the desired total order step by step along the subsets. This avoids an explicit iteration over all total orders which would result in an $O^*(k^k)$-time algorithm. Keeping track of the acyclicity constraints is achieved by so-called *coherence graphs*. Acyclicity of these graphs ensures that we build up the total order consistently with the considered memory model. In total, the dynamic programming runs in time $O^*(2^k)$. This constitutes the time-complexity of solving MM-Cons.

We applied our framework to the memory models SC, TSO, PSO, and RMO. To this end, we followed the CAT formulation of these memory models given in [11, 12] and instantiated the universal consistency problem MM-Cons by them. The result are $O(2^k)$-time algorithms for the corresponding consistency problems SC-Cons, TSO-Cons, PSO-Cons, and RMO-Cons. Note that these algorithms significantly improve upon already existing $O^*(k^k)$-time algorithms for the problems [336]. Those are usually based on an iteration over all store orders which we avoid by a dynamic programming.

For proving that our framework also yields optimal algorithms, we show that SC-Cons, TSO-Cons, and PSO-Cons cannot be solved in time $2^{o(k)}$. To this end, we rely on the ETH and construct a linear reduction from 3-SAT to each of the three problems. As we will see, the assumption of data independence makes these constructions non-trivial. Since writing the same value twice to a certain variable is not allowed, we need to find other sources of hardness that allow for encoding SAT into the corresponding consistency problem.

9.2 Framework

We present our framework for consistency algorithms. Given a memory model, the framework provides an (optimal) algorithm for the corresponding consistency problem. To this end, we first introduce some basic notions around memory models in Section 9.2.1. These are needed to state the universal consistency problem MM-Cons which is at the heart of our framework. We formally introduce the problem in Section 9.2.2. Finally, in Section 9.2.3, we solve MM-Cons with a dynamic programming algorithm running in time $O^*(2^k)$. Then, each instance of MM-Cons by a concrete

memory model admits an $O^*(2^k)$-time consistency algorithm. Note that the instantiations with SC, TSO, PSO, and RMO are shown in Section 9.3.

9.2.1 Memory Models

Describing memory models uniformly was a long-standing open problem. The task is difficult since some models like IBM's POWER [216] simply lack a formal definition while others, like the SPARC models [305, 306], do provide a formal definition within their specification. The pioneering work by Alglave [11] and Alglave, Maranget, and Tautschnig [12] solved the task and unified the approaches for describing memory models. The authors presented the language CAT which describes memory models in terms of relations and acyclicity constraints. Many memory models have been formulated in CAT up to now. We base our definition of a memory model on this language. However, note that our framework cannot be applied to any memory model formulated in CAT. We require that the model and the corresponding consistency check admit a particular form. This will become more clear in Section 9.2.2. We begin by introducing the notion of an *event* and we recall basic operations on relations. We mainly follow [11, 12, 62, 336].

Events and Relations To define the problem of checking consistency for a given execution, we first need a description of what an execution actually is. Formally, it consists of sequences of so-called *events* that model write and read accesses to the shared memory. Before we define them, fix some finite set of variables Var, a finite data domain Val, and a finite set of labels Lab.

Definition 9.1. A *write event* is defined by $w : wr(x, v)$, where $w \in Lab$ is a label, $x \in Var$ is a variable, and $v \in Val$ is a value. It models a write access which stores value v in variable x. The set of all write events is defined by

$$WR = \{w : wr(x, v) \mid w \in Lab, x \in Var, v \in Val\}.$$

A *read event* is given by $r : rd(x, v)$ with $r \in Lab$, $x \in Var$, and $v \in Val$. It models a read access loading v from variable x. The set of read events is

$$RD = \{r : rd(x, v) \mid r \in Lab, x \in Var, v \in Val\}.$$

The *set of all events* is given by $E = WR \cup RD$. For a subset $O \subseteq E$, we denote by $WR(O)$ and $RD(O)$, the set of write and read events in O. For a particular event $o \in E$, we often omit the label since it is clear from the context. Moreover, we use $var(o) \in Var$ to access the variable of o.

A real execution of a program imposes dependencies among events. For instance, a thread executes events in some particular order or a read event gets its value from some particular write event. To make these dependencies visible, we employ the notion of strict orders and relations.

Definition 9.2. Let $O \subseteq E$ be a subset of events. A *strict partial order on* O is an irreflexive, transitive relation over O. Note that it is not a usual partial order since the relation has to be irreflexive. A strict partial order that is total on the events of O is called a *strict total order on* O. We often refer to these orders without explicitly mentioning that they are strict.

Before we formally describe executions, we quickly fix the notation for some basic operations on relations. Let $O \subseteq E$ be a subset of events and $rel, rel' \subseteq O \times O$ two arbitrary relations over the set. We denote by $rel \circ rel'$ the composition of the relations, by rel^+ the transitive closure, and by rel^{-1} the inverse relation. For a fixed variable $x \in Var$, we denote by rel_x the restriction of relation rel to events that contain variable x. Formally, we set

$$rel_x = \{(o, o') \in rel \mid var(o) = var(o') = x\}.$$

Histories Now we have all the tools at hand to formally describe executions. They are modeled by so-called *histories*. A history consist of several sequences of events, each describing the execution of one thread. Dependencies among the events are described by the *program order* and the *reads-from relation*.

Definition 9.3. A *history* is a tuple $h = \langle O, po, rf \rangle$, where $O \subseteq E$ is a subset of events executed by the threads of the underlying program. The *program order* $po \subseteq O \times O$ is a partial order which orders the events of a thread according to its execution. Typically, po is a union of total orders, one for each thread. The relation $rf \subseteq WR(O) \times RD(O)$ is called *reads-from relation*. For each read event, it specifies the write event providing the needed value. Formally, for each read event $r \in RD(O)$ there is a write event $w \in WR(O)$ such that $(w, r) \in rf$ and if $(w, r) \in rf$, both events contain the same variable and value.

If a history $h = \langle O, po, rf \rangle$ is fixed, we typically focus on the subset of events O. To this end, we abuse notation and write WR and RD instead of $WR(O)$ and $RD(O)$. For a fixed variable $x \in Var$, we denote by $WR(x)$ the set of write events in h that write some value to variable x:

$$WR(x) = \{w \in WR \mid var(w) = x\}.$$

Since we need it later, we also define the relation *po-loc*. It is the restriction of the program order po to events on the same variable:

$$po\text{-}loc = \{(o, o') \in po \mid var(o) = var(o')\}.$$

To illustrate the definition and the notations, we continue with an example. To this end, we model the execution considered in Section 9.1 by a history.

Example 9.4. The history $h = \langle O, po, rf \rangle$ in Figure 9.3 formalizes the execution of Figure 9.1. Like the execution, h consists of the three treads T_1, T_2, and T_{pre}

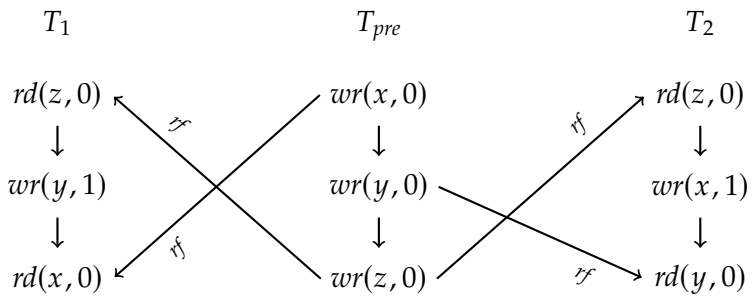


Figure 9.3: A history consisting of the threads T_1, T_2, and T_{pre}. The program order *po* is given by the unlabeled arrows. It follows the execution order (top to bottom) given by each thread. Formally, it is a union of the resulting three total orders. Arrows labeled by *rf* show the reads-from relation.

that communicate via the variables x, y, z over the data domain $\{0, 1\}$. The set of events O is given by the events listed in the figure.

Each thread executes from top to bottom. This is simulated by the program order *po*, given by the transitive closure of the unlabeled edges in h. Altogether, *po* is the union of three total orders, one for each thread. The reads-from relation is given by the arrows labeled by *rf*. Each read event in h is linked to its corresponding write event by the relation. For instance, we have that both read events $rd(z, 0)$ are linked to the write event $wr(z, 0)$. This means that in an actual execution, the write needs to happen before the two reads. Consequently, the threads T_1 and T_2 have to wait until T_{pre} finishes and executes $wr(z, 0)$ since the correct value for z is not available for reading earlier. The relation *po-loc* is empty in the example. Indeed, the program order does not relate two events involving the same variable.

Note that in the definition of a history $h = \langle O, po, rf \rangle$, we assume the reads-from relation *rf* to be given. This is due to the assumption of data independence [2, 43, 62, 288, 332, 336] that we make. In fact, implementations of shared memories and databases do not always depend on actual data values in practice. This means that in an execution, we may assume that a specific value is written to a variable at most once. From such an execution, we can simply read off the relation *rf*. Note that without having *rf* at hand, analyzing histories is more involved. A history without reads-from relation induces many histories with specific *rf*, one for each possible relation among writes and reads that explains how the read values are obtained.

Memory Models We define our notion of *memory models*. Intuitively, such a model is an abstraction of the concrete behavior of a shared memory. It formulates axioms that the program order and the reads-from relations of

histories must satisfy. The upcoming definition goes back to the work of Alglave [11]. Compared to the full language CAT [12], the definition can be seen as an earlier and simpler version of the language that, for instance, does not handle memory fences. While this limits the expressiveness of our framework, it offers certain structures that we can exploit algorithmically.

Definition 9.5. A *memory model* MM is a tuple MM = (*po-mm*, *rf-mm*) consisting of the two relations *po-mm* and *rf-mm*. The *preserved program order* *po-mm* is a subrelation of *po* that describes which pairs in program order are maintained by MM. The latter relation, *rf-mm*, is a subrelation of *rf*. It shows which write events are visible globally under the memory model.

As a teaser, we phrase the memory model *Sequential Consistency* (SC) in terms of the above definition. Note that SC's original definition by Lamport [251] is informal but Alglave [11] showed that it is actually an instance of Definition 9.5. The formal definitions of other models like TSO and PSO are postponed to Section 9.3 when we apply our algorithmic framework.

Definition 9.6. The memory model *Sequential Consistency* (SC) is defined by

$$\text{SC} = (\textit{po-sc}, \textit{rf-sc}).$$

In the tuple, the preserved program order *po-sc* = *po* is the complete program order and similarly, *rf-sc* = *rf* is the complete reads-from relation.

The memory model SC preserves the complete program order. This means it does not allow for reordering events within the same thread but only for interleaving events from different threads. Moreover, under SC each issued write event gets pushed immediately to the memory and is therefore visible to each other thread. Consequently, the memory model also preserves the complete reads-from relation. As we will see later, other more relaxed memory models, do not preserve *po* and *rf* completely. Hence, we may see sequential consistency as the strictest of all memory models [307].

9.2.2 Universal Consistency

We formulate the foundation of our framework. It is based on a consistency problem that we call MM-Cons. The term MM in the name refers to an arbitrary memory model. The idea is that MM-Cons can be instantiated to simulate a particular model of choice. But before we can define MM-Cons, we first need to make the notion of consistency more precise. We chose a formulation that allows for an efficient algorithmic check but deviates from the literature [11, 12, 62, 336] at first sight. However, as we will prove later in Section 9.3, our notion and the standard notion from literature coincide.

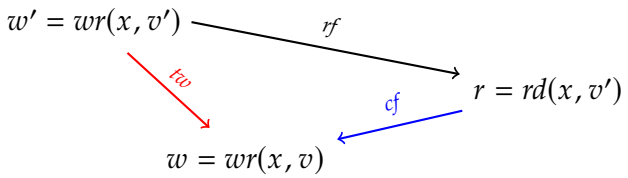


Figure 9.4: The conflict relation cf. The write w succeeds w' in tw-order (red) but the read r obtains its value from w'. We get an cf-edge (blue) from r to w indicating that r needs to happen before w in an execution.

Consistency We elaborate on our notion of consistency. Intuitively, a history is consistent under a memory model if the events can be scheduled in such a way that no dependency cycles occur. These cycles clash with the program order and the reads-from relation preserved by the memory model and cannot be scheduled properly. Following [11, 12], finding a schedule amounts to finding a particular *store order* on the write events of the history which satisfies certain acyclicity requirements. We replace the store order by a total order and focus on two particular acyclicity requirements.

Definition 9.7. Let $h = \langle O, po, rf \rangle$ be a history and MM $= (po\text{-}mm, rf\text{-}mm)$ an arbitrary memory model. Then h is called MM-*consistent* or *consistent under* MM if there exists a strict total order $tw \subseteq WR \times WR$ such that the two graphs

$$\mathcal{G}_{loc} = (O, po\text{-}loc \cup rf \cup tw \cup cf) \text{ and } \mathcal{G}_{mm} = (O, po\text{-}mm \cup rf\text{-}mm \cup tw \cup cf)$$

are both acyclic. In the graphs, the *conflict relation* $cf \subseteq RD \times WR$ is defined by

$$cf = rf^{-1} \circ \bigcup_{x \in Var} tw_x.$$

In the definition, the relation cf makes visible the conflicts between read and write events that stem from fixing the total order tw. We have $(r, w) \in cf$ if r is a read event on a variable x, w is a write event on x, and there is a write event w' on x such that $(w', r) \in rf$ and $(w', w) \in tw$. Hence, in an execution, the read r gets its value from w' and since w succeeds w' in tw-order, we need that r happens before the write w. Otherwise, the read value gets overwritten by w too early. This is reflected by the single edge $(r, w) \in cf$ in conflict relation. We illustrate the situation in Figure 9.4.

The acyclicity of the graph $\mathcal{G}_{loc}$ is called *uniprocessor* requirement [11] or *memory coherence* for each location [77]. Roughly it demands that the program order restricted to writes on the same variables is respected by the total order tw. The second acyclicity requirement in the definition resembles the memory model MM. If the graph $\mathcal{G}_{mm}$ is acyclic, the given history can be scheduled without violating the preserved program order and reads-from relation. We consider an example that clarifies the acyclicity requirements.

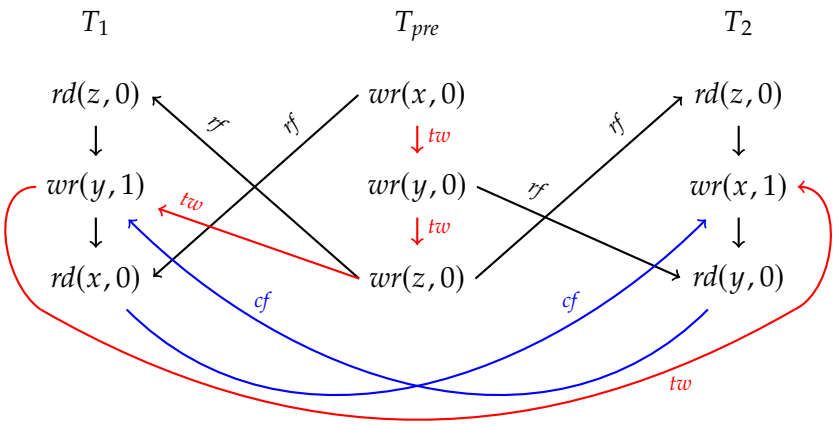


Figure 9.5: The graph $\mathcal{G}_{sc}$. The total order tw is the transitive closure of the red edges. The blue edges are the conflict relation cf. Note that the graph contains a cycle showing that tw is not a witness for SC-consistency of h.

Example 9.8. Consider the history $h = \langle O, po, rf \rangle$ from Figure 9.3. In Section 9.1, we have already seen that h cannot be executed under SC. Hence, the history should not be SC-consistent. We argue that this is indeed true.

By Definition 9.6, SC is given by the tuple SC $= (po\text{-}sc, rf\text{-}sc) = (po, rf)$. To disprove SC-consistency of h, we would need to show that for each total order tw on the write events, one of the two graphs $\mathcal{G}_{loc}$ or $\mathcal{G}_{sc}$ contains a cycle. But instead of testing it for each tw, we focus on a particular one. Other orders can be tested similarly. Figure 9.5 illustrates the graph

$$\mathcal{G}_{sc} = (O, po \cup rf \cup tw \cup cf)$$

for the total order tw that is given by the transitive closure of the red edges in the graph. For the construction of $\mathcal{G}_{sc}$, note that the conflict relation cf consists of two edges, namely $rd(y,0) \xrightarrow{cf} wr(y,1)$ and $rd(x,0) \xrightarrow{cf} wr(x,1)$. These are marked blue in Figure 9.5. The former edge stems from the inverted rf-edge $rd(y,0) \rightarrow wr(y,0)$ composed with the tw-edge $wr(y,0) \rightarrow wr(y,1)$. The other edge of the conflict relation is obtained similarly.

Obviously, $\mathcal{G}_{sc}$ contains several cycles. One of these is given by the edges

$$rd(y,0) \xrightarrow{cf} wr(y,1) \xrightarrow{tw} wr(x,1) \xrightarrow{po} rd(y,0).$$

It shows that the chosen total order tw is not a witness for consistency of h under SC. In fact, similarly to our example, one can show that any total order tw will cause such a cycle. This implies that h is not SC-consistent.

As we already mentioned above, our definition of consistency deviates from the literature [11, 12]. We make the differences more precise. The

largest deviation is that we demand a total order *tw* instead of a *store order*. The latter is a partial order that is total on write events to the same variable. In Section 9.3 we will show that we can replace one by another without changing the resulting notion of consistency. A further difference is that we do not explicitly test for *out of thin air values* [266]. For the majority of memory models considered in this thesis, the test is not required as it is already implied by the acyclicity of $\mathcal{G}_{loc}$ and $\mathcal{G}_{mm}$. But when the test is needed, like in the case of RMO, it can easily be added. Lastly, we do not need a particular test for *well-formedness*. In fact, it is already implied by our definition of histories and the fact that we search for a total order on all write events.

Checking Consistency With the correct notion of consistency at hand, we are ready to define the universal consistency problem MM-Cons. To this end, let MM be some fixed memory model. Given a history h as input, MM-Cons asks whether h is actually consistent under MM.

MM-Cons
Input: A history $h = \langle O, po, rf \rangle$.
Question: Is h MM-consistent?

As discussed earlier, the classical complexity of the problem is well-understood. Instantiations by memory models like SC, TSO, or PSO are typically NP-complete [180, 183]. Concerning the fine-grained complexity, many parameterizations of MM-Cons are intractable. For instance parameterizing by the number of threads is W[1]-hard [270], the parameterization in the number of events per thread is not even in XP [183], and as we will see later, the parameterization in the size of the data domain is not in XP as well. Therefore, we conduct a fine-grained complexity analysis in the parameter $k = |WR|$, the number of write events in the given history. The choice of k as a parameter is also motivated by practice. In fact, the number seems to be small in practical instances after an application of the polynomial preprocessing described in [336], to which our framework can be adapted to.

One of the main findings of our fine-grained complexity analysis is an algorithm for MM-Cons which runs in time $O^*(2^k)$. We will later instantiate the problem with concrete memory models and obtain an $O^*(2^k)$-time consistency algorithm for each. Moreover, we will provide lower bounds that prove their optimality. Let $n = |O|$ denote the number of events in the given history h. The following theorem formally states the upper bound.

Theorem 9.9. *The problem MM-Cons can be solved in time* $O(2^k \cdot k^2 \cdot n^2)$.

Note that MM-Cons(k) is quickly seen to be fixed-parameter tractable. In fact, the problem can be solved by a simple algorithm running in time $O^*(k^k)$.

It iterates over all total orders of WR and checks acyclicity of $\mathcal{G}_{loc}$ and $\mathcal{G}_{mm}$ in polynomial time. Since the number of total orders is bounded by k^k, we obtain the stated running time. Our goal is to improve on this algorithm. As we cannot afford to iterate over all total orders in $O^*(2^k)$, we need an alternative approach. We summarize our development in the upcoming section.

9.2.3 Upper Bound

We present the algorithm for the universal consistency problem MM-Cons. The main idea is to switch from the domain of total orders, where the problem is defined, to the subset lattice of write events. Over the lattice, we can then employ a dynamic programming which runs in time $O^*(2^k)$. The crux in changing the domain of the problem is that for a particular subset of write events, we do not need to remember a precise total order. As we will see, we only need to store that it can be ordered by a so-called *snapshot order*. These orders approximate total orders on subsets of writes. But not having a precise order at hand yields a significant disadvantage. We cannot just check acyclicity requirements in the end like in the simple iteration algorithm above. Instead, we need to perform several acyclicity tests on-the-fly. To this end, we employ the notion of *coherence graphs*. These graphs carry enough information to ensure acyclicity as it is required by MM-Cons. For the remaining section, we fix an instance of MM-Cons, a history $h = \langle O, po, rf \rangle$.

Snapshot Orders We begin our technical development by introducing *snapshot orders*. Intuitively, they express that a certain subset of write events has already been ordered totally while the complement of the subset did not admit an order yet. To this end, a snapshot order consists of two parts: a total order on the particular subset and a partial order expressing that the complement precedes the subset but is yet unordered.

Definition 9.10. Let $V \subseteq WR$ be a subset. A *snapshot order on* V is a union

$$tw[V] = t[V] \cup r[V].$$

It consists of some strict total order $t[V]$ on V and a fixed relation $r[V]$ which expresses that the elements of the complement $\overline{V} = WR \setminus V$ are smaller than the elements of V, but which leaves $\overline{V}$ unordered. It is defined by

$$r[V] = \{(\overline{v}, v) \mid \overline{v} \in \overline{V}, v \in V\}.$$

Note that, like the name already suggests, a snapshot order is indeed a strict partial order. Moreover, when the underlying subset V is the set of all write events, so if we have $V = WR$, any snapshot order $tw[WR]$ is actually a total order on WR. Note that $r[WR] = \emptyset$ in this case. But this means that consistency under MM can be checked by finding a snapshot order $tw[WR]$

that satisfies both acyclicity constraints. This formulation has one significant advantage over the formulation in terms of plain total orders: snapshot orders can be constructed from other snapshot orders on smaller subsets. Technically, this is the foundation of our dynamic programming algorithm and the reason why we can shift from total orders to subsets of write events.

Since we replace total orders by snapshot orders, we need to generalize the acyclicity constrains of MM-consistency to the latter. With this adjustment, we can then phrase the problem MM-Cons over the subset lattice $\mathcal{P}(WR)$.

Definition 9.11. Let $V \subseteq WR$ be a subset of write events and $tw[V]$ some snapshot order on V. We define the following two graphs:

$$\mathcal{G}_{loc}(tw[V]) = (O, po\text{-}loc \cup rf \cup tw[V] \cup cf[V]),$$
$$\mathcal{G}_{mm}(tw[V]) = (O, po\text{-}mm \cup rf\text{-}mm \cup tw[V] \cup cf[V]).$$

Similar to the definition of MM-consistency given above, the *conflict relation* $cf[V] \subseteq RD \times WR$ in both graphs is defined by $cf[V] = rf^{-1} \circ \bigcup_{x \in Var} tw[V]_x$.

Note that Definition 9.11 is obtained by exchanging the total order tw in the graphs $\mathcal{G}_{loc}$ and $\mathcal{G}_{mm}$ by a snapshot order on some subset $V \subseteq WR$. Consequently, for a snapshot order $tw[WR]$ on the complete set of write events, the resulting graphs $\mathcal{G}_{loc}(tw[WR])$ and $\mathcal{G}_{mm}(tw[WR])$ are exactly $\mathcal{G}_{loc}$ and $\mathcal{G}_{mm}$ as they appear in the definition of MM-consistency. To illustrate the newly defined graphs, we consider an example.

Example 9.12. In Example 9.8, we have analyzed the consistency of history $h = \langle O, po, rf \rangle$ from Figure 9.3 under SC. To this end, we constructed $\mathcal{G}_{sc}$ for a particular total order tw on the write events of h. Now we construct the graph $\mathcal{G}_{sc}(tw[V])$ along a snapshot order $tw[V]$. It is shown in Figure 9.6.

First, we fix a subset. Let $V = \{wr(y,1), wr(x,1)\}$. It is shown by the gray highlighted events in the graph. As a snapshot order we chose

$$tw[V] = t[V] \cup r[V],$$

where $t[V]$ consists of only one edge, namely $wr(y,1) \to wr(x,1)$. This already constitutes a total order on V. The edge is marked red in the graph. The relation $r[V]$ is fixed by definition. It contains an edge from each write event in $\overline{V}$ to each write in V. The corresponding edges are marked green.

To complete the construction of $\mathcal{G}_{sc}(tw[V])$, it is left to determine the conflict relation $cf[V]$. It consists of the two edges

$$rd(y,0) \xrightarrow{cf[V]} wr(y,1) \text{ and } rd(x,0) \xrightarrow{cf[V]} wr(x,1).$$

Both edges are highlighted in blue. Note that the former one exists since there is an rf-edge $rd(y,0) \to wr(y,0)$ and an $tw[V]$-edge $wr(y,0) \to wr(y,1)$. The reasoning for the second edge of the conflict relation is similar.

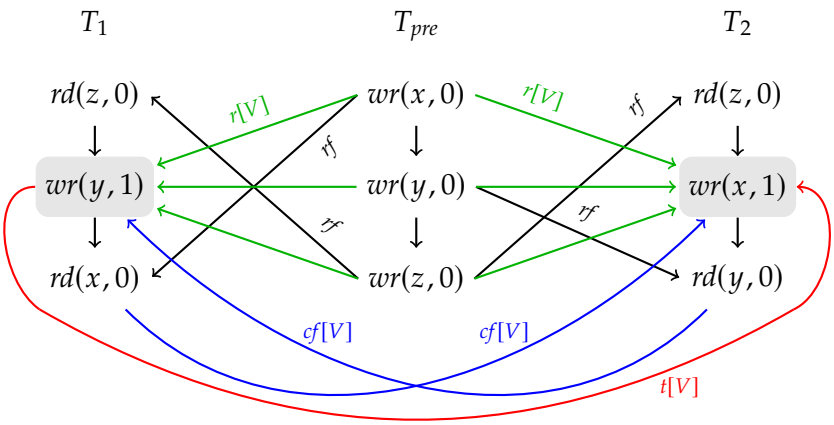


Figure 9.6: The graph $\mathcal{G}_{sc}(tw[V]) = (O, po \cup rf \cup tw[V] \cup cf[V])$ with set $V = \{wr(y,1), wr(x,1)\}$ highlighted in gray. The snapshot order $tw[V]$ is the union of the total order $t[V]$, marked red, and the partial order $r[V]$, marked green. The conflict relation $cf[V]$ consists of the two blue edges.

The graph $\mathcal{G}_{sc}(tw[V])$ already contains a cycle. Hence, we do not need to construct a total order on the complete set of write events to obtain one, the snapshot order is sufficient. In fact, each total order tw that contains $tw[V]$ induces a cycle in its graph $\mathcal{G}_{sc}$ and is hence not able to witness SC-consistency of h. Note that the total order constructed in Example 9.8 is such an order.

With snapshot orders and the above defined graphs we can now formulate MM-Cons over subsets of write events. This reflects the desired switch away from the domain of total orders to a domain that has an algorithmically appealing structure. Formally, we state MM-consistency in terms of a table T with a Boolean entry $T[V]$ for each subset $V \subseteq WR$. The entry $T[V]$ will evaluate to 1, if there is a snapshot order on V that does not induce a dependency cycle. Otherwise, $T[V]$ will evaluate to 0. We give a formal definition.

Definition 9.13. Table T has for each set $V \subseteq WR$ an entry $T[V]$, defined by

$$T[V] = \begin{cases} 1, \text{ if } \exists \text{ snapshot ord. } tw[V] : \mathcal{G}_{loc}(tw[V]) \text{ and } \mathcal{G}_{mm}(tw[V]) \text{ acyclic,} \\ 0, \text{ otherwise.} \end{cases}$$

Note that the entry $T[WR]$ of the table actually stores whether h is MM-consistent. In fact, any snapshot order $tw[WR]$ on the complete set of writes is a total order. Moreover, the corresponding graphs $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V])$ coincide with $\mathcal{G}_{loc}$ and $\mathcal{G}_{mm}$, those graphs which are required to be acyclic. This means that T stores sufficient information to solve MM-Cons. The following lemma summarizes the relation between problem and table.

Lemma 9.14. *History h is MM-consistent if and only if $T[WR] = 1$.*

Coherence Graphs Lemma 9.14 leaves us with the problem of evaluating the entry $T[WR]$. To obtain $T[WR]$, our approach is to set up a recurrence relation among the entries of T and then to fill the table with a bottom-up dynamic programming. For the formulation of the recurrence relation, we add one write event after another. This means we show how an entry $T[V]$ can be used to compute an entry $T[V \cup \{v\}]$ for a write event $v \in \overline{V}$.

When passing from $T[V]$ to $T[V \cup \{v\}]$, we need to provide a snapshot order on the larger set $V \cup \{v\}$ which satisfies the stated acyclicity requirements. Note that a snapshot order on V can always be extended to a snapshot order on $V \cup \{v\}$: we only need to insert v as new minimal element in the contained total order. But this is not sufficient to evaluate $T[V \cup \{v\}]$. We also need to keep track of whether the acyclicity requirements are compatible with the new minimal element v. To this end, we perform acyclicity tests on so-called *coherence graphs*. These graphs do not depend on a particular snapshot order and solely rely on the fact that v is the minimal element. This will later allow for a recurrence relation on T that avoids touching precise orders.

Definition 9.15. Let $V \subseteq WR$ be a subset of write events and v an element in $\overline{V}$. The *coherence graphs of V and v* are defined as follows:

$$\mathcal{G}_{loc}[V, v] = (O, \mathit{po\text{-}loc} \cup \mathit{rf} \cup r[V, v] \cup \mathit{cf}[V, v]),$$
$$\mathcal{G}_{mm}[V, v] = (O, \mathit{po\text{-}mm} \cup \mathit{rf\text{-}mm} \cup r[V, v] \cup \mathit{cf}[V, v]).$$

The relation $r[V, v]$ expresses that the set $\overline{V \cup \{v\}}$ precedes $V \cup \{v\}$ and that the write event v is the minimal element in $V \cup \{v\}$. It is defined by

$$r[V, v] = r[V \cup \{v\}] \cup \{(v, w) \mid w \in V\},$$

Note that $r[V \cup \{v\}]$ is the fixed order from Definition 9.10. The *conflict relation* is defined as expected, by $\mathit{cf}[V, v] = \mathit{rf}^{-1} \circ \bigcup_{x \in Var} r[V, v]_x$.

Coherence graphs provide less information than the graphs $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V])$ that depend on a particular snapshot order. We show a comparison in Figure 9.7. In fact, a coherence graph $\mathcal{G}_{mm}[V, v]$ like on the left hand side of the figure does not know about a total order within the elements of V. It only carries information that can be derived from v and V immediately. Namely that v is minimal in $V' = V \cup \{v\}$ and that $\overline{V'}$ precedes V. This means we can construct the coherence graphs without knowing a precise snapshot order. This is crucial for the recurrence relation that we aim for since it prevents us from iterating over possible snapshot orders.

Assume we know that $T[V] = 1$ and we want to evaluate $T[V']$. Since the entry of V evaluates to 1, we know that there is a snapshot order $tw[V]$ such that $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V])$ are both acyclic. Now we construct a snapshot order $tw[V']$ on V' as above — by inserting v as the minimal element of V'. Now the question is whether the graphs $\mathcal{G}_{loc}(tw[V'])$ and $\mathcal{G}_{mm}(tw[V'])$

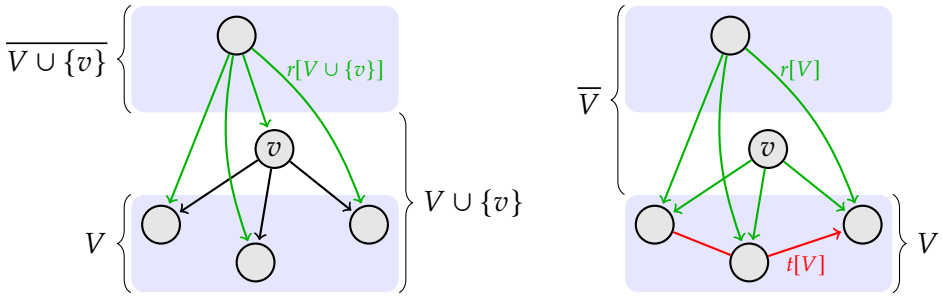


Figure 9.7: Schematic illustration of a coherence graph $\mathcal{G}_{mm}[V, v]$ on the left hand side. Vertices are write events of the history. The figure focuses on the relation $r[V, v] = r[V \cup \{v\}] \cup \{(v, w) \mid w \in V\}$. Edges of $r[V \cup \{v\}]$ are marked green, others are marked black. On the right hand side, we illustrate $\mathcal{G}_{mm}(tw[V])$ for some snapshot order $tw[V] = t[V] \cup r[V]$ on V. The total order $t[V]$ is marked red, the relation $r[V]$ is marked green.

are both acyclic. Instead of answering it directly, we show that each cycle which occurs in one of the two graphs already occurs in one of the coherence graph $\mathcal{G}_{loc}[V, v]$ or $\mathcal{G}_{mm}[V, v]$. Hence, if both coherence graphs are known to be acyclic, we obtain that $T[V'] = 1$ as well. This means that in the recurrence relation, we need to test two things to determine $T[V']$. Namely whether $T[V] = 1$ and whether the corresponding coherence graphs are both acyclic.

In the subsequent lemma, we make the recurrence relation more precise. Note that it is a top-down formulation that only refers to non-empty subsets of write events. Evaluating the base case $V = \emptyset$ is trivial. The only snapshot order on the empty set is the empty order. Hence, $T[\emptyset] = 1$ if and only if the graphs $\mathcal{G}_{loc}(\emptyset) = (O, po\text{-}loc \cup rf)$ and $\mathcal{G}_{mm}(\emptyset) = (O, po\text{-}mm \cup rf\text{-}mm)$ are acyclic.

Lemma 9.16. *Let $V \subseteq WR$ be a non-empty subset of write events. The table T admits the following recurrence relation among its entries:*

$$T[V] = \bigvee_{v \in V} \big(\mathcal{G}_{loc}[V \setminus \{v\}, v]\ acyclic\big) \wedge \big(\mathcal{G}_{mm}[V \setminus \{v\}, v]\ acyclic\big) \wedge T[V \setminus \{v\}].$$

We interpret the expression $\big(\mathcal{G}_{loc}[V \setminus \{v\}, v]\ \text{acyclic}\big)$ as a predicate that evaluates to 1 if and only if the corresponding graph is acyclic. Hence, the recurrence relation requires the existence of a write event $v \in V$ such that both coherence graphs are acyclic and the entry $T[V \setminus \{v\}]$ evaluates to 1. A proof of Lemma 9.16 is provided in Appendix B.4.1.

Algorithm With the recurrence relation at hand we can evaluate the table T by a dynamic programming. To this end, we store already computed entries and look them up when required. We formulated the approach as Algorithm 9.1 below. Initially, we evaluate $T[\emptyset]$. As mentioned above, this

Algorithm 9.1 MM-Consistency

Input: A history $h = \langle O, po, rf \rangle$.
Output: *True*, if h is MM-consistent. *False* otherwise.

```
1: let T be a table with entry T[V] = 0 for any V ⊆ WR.   // Initialize table.
2: compute T[∅] = ((O, po-loc ∪ rf) acyclic) ∧ ((O, po-mm ∪ rf-mm) acyclic)
3: for each ∅ ≠ V ⊆ WR do   // Some increasing order on subsets of WR.
4:   for each v ∈ V do
5:     set V' = V \ {v}   // Test recurrence for v ∈ V.
6:     compute T[V] = T[V'] ∧ (G_loc[V', v] acyclic) ∧ (G_mm[V', v] acyclic)
7:     if T[V] = 1 then
8:       goto next subset of WR
9:     end if
10:  end for
11: end for
12: return T[WR]
```

amounts to testing whether $(O, po\text{-}loc \cup rf)$ and $(O, po\text{-}mm \cup rf\text{-}mm)$ are acyclic. An entry $T[V]$ with $V \neq \emptyset$ is evaluated as follows. We branch over all write events $v \in V$, test whether the two coherence graphs $\mathcal{G}_{loc}[V \setminus \{v\}, v]$ and $\mathcal{G}_{mm}[V \setminus \{v\}, v]$ are acyclic, and look up whether $T[V \setminus \{v\}] = 1$. If all three queries are positive, we store $T[V] = 1$. If such a write $v \in V$ does not exist, we obtain $T[V] = 0$. Correctness of Algorithm 9.1 follows from the recurrence relation, Lemma 9.16, and the above characterization, Lemma 9.14.

To prove Theorem 9.9, it is left to elaborate on the complexity of Algorithm 9.1. The table T has 2^k many entries that we need to compute. This constitutes the exponential factor in the estimation of Theorem 9.9. For each entry $T[V]$, we branch over at most k many write events $v \in V$. Looking up the value of $T[V \setminus \{v\}]$ can be done in constant time because we have already computed it. As we will see in the subsequent lemma, constructing the coherence graphs and testing them for acyclicity can be performed in time $O(k \cdot n^2)$. Hence, a single entry can be computed in time $O(k^2 \cdot n^2)$. Consequently, we can estimate the time complexity of Algorithm 9.1 by $O(2^k \cdot k^2 \cdot n^2)$.

Lemma 9.17. *Let $V \subseteq WR$ and $v \in \overline{V}$. Constructing the coherence graphs $\mathcal{G}_{loc}[V, v]$ and $\mathcal{G}_{mm}[V, v]$ and testing both for acyclicity can be done in time $O(k \cdot n^2)$.*

Proof. The construction of the coherence graphs is straight forward. Acyclicity is checked by employing Kahn's algorithm [226] for finding a topological sorting. We provide details in Appendix B.4.2. □

9.3 Instantiating the Framework

We show the applicability of our framework and obtain consistency algorithms for the memory models SC, TSO, PSO, and RMO. To this end, we first need to prove that our notion of consistency and the notion of consistency used in the literature actually coincide. We refer to the latter as *validity*. In Section 9.3.1, we show that consistency and validity are indeed equal. This ensures that consistency algorithms obtained from our framework really solve the correct problem. In Section 9.3.2 we then apply our framework to the mentioned memory models. Applying it to SC, TSO, and PSO is straight forward. For RMO, we need to make a slight adjustment.

9.3.1 Validity versus Consistency

Consistency, as it is considered in the literature, is sometimes also referred to as *validity* [11, 12]. We use this name to avoid confusion with consistency as we defined it. In Section 9.2.2, we elaborated on the differences between consistency and validity. Now we show that although the definitions vary, both notions are actually equal. To this end, we first give a formal definition of validity. It relies on so-called *store orders* [11, 12, 336] that are also known as *coherence orders*. These are orders that are total on write events to the same variable but do not relate write events to different variables.

Definition 9.18. Let $h = \langle O, po, rf \rangle$ be a history. A *store order* is a strict partial order $ww \subseteq WR \times WR$ such that ww_x is a strict total order on $WR(x)$ for each variable $x \in Var$ and such that it decomposes into

$$ww = \bigcup_{x \in Var} ww_x.$$

Note that in contrast to total orders on WR, store orders provide less information. In fact, we may see a total order as an enriched store order where edges between write events to distinct variables have been added.

With store orders at hand, we can now define the notion of validity. Similar to consistency, it requires that certain dependency graphs are acyclic. But instead of a total order, it only requires a store order.

Definition 9.19. Let $h = \langle O, po, rf \rangle$ be a history and MM = $(po\text{-}mm, rf\text{-}mm)$ some arbitrary memory model. History h is called MM-*valid* if there exists a store order $ww \subseteq WR \times WR$ such that the two graphs

$$\mathcal{G}_{loc}^{ww} = (O, po\text{-}loc \cup rf \cup ww \cup fr) \text{ and } \mathcal{G}_{mm}^{ww} = (O, po\text{-}mm \cup rf\text{-}mm \cup ww \cup fr)$$

are acyclic. The *from-read relation* $fr \subseteq RD \times WR$ is defined by $fr = rf^{-1} \circ ww$.

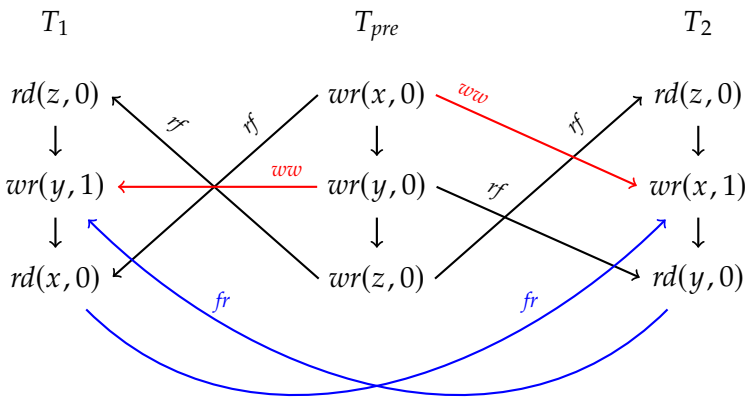


Figure 9.8: The graph $\mathcal{G}_{sc}^{ww}$. The corresponding store order ww is given by the red edges, the from-read relation fr is shown in blue. Note that the graph contains a cycle. Hence, ww is not a witness for SC-validity of h.

Like in the definition of consistency, we do not explicitly check for out of thin air values. Whenever the test is required, we can add it since it does not depend on a particular store order. Moreover, well-formedness again follows automatically from the definition of histories and the fact that we are searching for a store order. We illustrate validity with our running example.

Example 9.20. We reconsider history $h = \langle O, po, rf \rangle$ of Figure 9.3. In Example 9.8 we have seen that h is not SC-consistent. Since we are about to how that consistency and validity are equal notions, h should also not be SC-valid. This is indeed true. Like for consistency, for validity we would have to show that each store order ww induces a cycle in one of the graphs $\mathcal{G}_{loc}^{ww}$ or $\mathcal{G}_{sc}^{ww}$. Again, we focus on a single ww. In Figure 9.8, we illustrate the graph

$$\mathcal{G}_{sc}^{ww} = (O, po \cup rf \cup ww \cup fr)$$

for the store order ww given by the red edges. Note that it is indeed a store order. By the definition of the from-read relation, we get the two edges

$$rd(y,0) \xrightarrow{fr} wr(y,1) \text{ and } rd(x,0) \xrightarrow{fr} wr(x,1).$$

The from-read-edges induce a cycle in the graph. Hence, ww does not witness SC-validity of h and indeed no store order can. One can show that for each other store order, a dependency cycle occurs.

We show the equivalence of validity and consistency. To this end, we need to prove that the store order in the definition of validity can be replaced by a total order on the write events that preserves acyclicity. A first step is provided by the following lemma. It shows that a store order ww in $\mathcal{G}_{loc}^{ww}$

can be replaced by any linearization of ww without affecting the acyclicity. Phrased differently, any total order tw on the write events that contains ww can be inserted into $\mathcal{G}_{loc}^{ww}$ - it will still be acyclic. The reasoning for the graph $\mathcal{G}_{mm}^{ww}$ will be simpler. We state the lemma, a proof is given in Appendix B.4.3.

Lemma 9.21. *Let $h = \langle O, po, rf \rangle$ be a history, ww a store order, and tw a total order on WR such that $ww \subseteq tw$. If $\mathcal{G}_{loc}^{ww}$ is acyclic, then so is the graph*

$$\mathcal{G}_{loc}^{tw} = (O, po\text{-}loc \cup rf \cup tw \cup fr).$$

Now we are ready to prove the desired result: validity and consistency actually refer to the same notion. The result is crucial for the applicability of our framework. We formalize it in the upcoming lemma.

Lemma 9.22. *A history is MM-valid if and only if it is MM-consistent.*

Proof. Fix a history $h = \langle O, po, rf \rangle$ and assume that h is MM-valid. Then there exists a store order ww such that $\mathcal{G}_{loc}^{ww}$ and $\mathcal{G}_{mm}^{ww}$ are both acyclic. We consider the edges of the latter graph in more detail. They form a relation

$$ord\text{-}mm = po\text{-}mm \cup rf\text{-}mm \cup ww \cup fr.$$

Since the graph $\mathcal{G}_{mm}^{ww}$ is acyclic, the transitive closure $ord\text{-}mm^+$ does not contain reflexive elements. Hence, it is a strict partial order on the events O. But this means that there is a linear extension of $ord\text{-}mm^+$, a strict total order $L \subseteq O \times O$ such that $ord\text{-}mm^+ \subseteq L$. From L, we can construct the required total order on the set of write events. We define it by

$$tw = L \cap WR \times WR.$$

Then, tw is a total order on WR and we have $ww \subseteq L \cap WR \times WR = tw$. To show MM-consistency of h, we need to prove that $\mathcal{G}_{loc}$ and $\mathcal{G}_{mm}$ are acyclic. Note that the latter refer to the graphs from Definition 9.7:

$$\mathcal{G}_{loc} = (O, po\text{-}loc \cup rf \cup tw \cup cf) \text{ and } \mathcal{G}_{mm} = (O, po\text{-}mm \cup rf\text{-}mm \cup tw \cup cf).$$

The store order ww is contained in tw. Hence, we obtain that $ww_x \subseteq tw_x$ for each variable $x \in Var$. This implies that $ww_x = tw_x$ since ww_x is total on $WR(x)$, the write events to x. We can therefore deduce that

$$ww = \bigcup_{x \in Var} ww_x = \bigcup_{x \in Var} tw_x$$

and consequently, by the definition of the conflict relation cf, that

$$cf = rf^{-1} \circ \bigcup_{x \in Var} tw_x = rf^{-1} \circ ww = fr.$$

Since $fr = cf$ and $ww \subseteq tw$, we obtain the acyclicity of $\mathcal{G}_{loc} = \mathcal{G}_{loc}^{tw}$ from Lemma 9.21. The acyclicity of the graph $\mathcal{G}_{mm}$ follows since its edges

$$po\text{-}mm \cup rf\text{-}mm \cup tw \cup cf$$

form a subrelation of L. A cycle would mean that L has a reflexive element, but L is a strict order. Hence, h is MM-consistent.

For the other direction, assume that h is MM-consistent. By definition, there is a total order tw on WR such that $\mathcal{G}_{loc}$ and $\mathcal{G}_{mm}$ are both acyclic. To show that h is also MM-valid, we construct the store order

$$ww = \bigcup_{x \in Var} tw_x.$$

Note that, since tw_x is total on $WR(x)$, the order ww is indeed a proper store order and we clearly have the inclusion $ww \subseteq tw$.

We show that $\mathcal{G}_{loc}^{ww}$ and $\mathcal{G}_{mm}^{ww}$ are acyclic graphs. In fact, we have that

$$fr = rf^{-1} \circ ww = cf.$$

This implies that $\mathcal{G}_{loc}^{ww}$ and $\mathcal{G}_{mm}^{ww}$ are subgraphs of $\mathcal{G}_{loc}$ and $\mathcal{G}_{mm}$, respectively. Hence, the two graphs are acyclic and h is MM-valid. □

9.3.2 Instances

We apply our framework to the memory models SC, TSO, PSO, and RMO and obtain $O^*(2^k)$-time consistency algorithms for each. The result from Section 9.3.1 ensures that the algorithms are indeed correct. Note that for the application of the framework, we rely on the formal description of the mentioned memory models due to Alglave, Maranget, and Tautschnig [11, 12].

Sequential Consistency We have already considered the memory model *Sequential Consistency* (SC) [251] in Definition 9.6. Recall that it is given by

$$\text{SC} = (po\text{-}sc, rf\text{-}sc) = (po, rf).$$

This means that SC preserves the complete program order and reads-from relation. This makes the uniprocessor test, the acyclicity of the graph $\mathcal{G}_{loc}$, obsolete since it is a subgraph of $\mathcal{G}_{sc}$. However, our framework still applies in this case. Following Theorem 9.9, it yields an algorithm for SC-consistency which runs in time $O(2^k \cdot k^2 \cdot n^2)$. In Section 9.4, we show that SC-consistency cannot be solved in time $2^{o(k)}$ unless the ETH fails. Hence, our framework yields a provable optimal consistency algorithm for SC.

Corollary 9.23. *The problem SC-consistency can be solved in time* $O(2^k \cdot k^2 \cdot n^2)$.

Total Store Order The memory model *Total Store Order* (TSO) [305, 306] by SPARC resembles a relaxed memory behavior. Instead of directly accessing the memory with each write or read event like in SC, each thread has an own FIFO buffer which delays the memory access. In fact, issued write events of a thread are pushed into the buffer and are only visible to the owning thread but not to others. At some nondeterministic point in time, the memory decides to flush the buffer and to update the variables accordingly. Only then, the writes are visible to other threads as well. If the owner of the buffer reads a certain variable, it tries to perform an *early read*. This means the thread first looks through its buffer and reads the latest issued write on that variable. It only loads the value of the variable from the memory if there is no write on that variable in the buffer. We give a formal definition.

Definition 9.24. The memory model *Total Store Order* (TSO) is defined by

$$\text{TSO} = (po\text{-}tso, rf\text{-}tso).$$

The preserved program order *po-tso* is a relaxation of the program order which contains no write-read pairs. Formally, it is given by

$$po\text{-}tso = po \setminus WR \times RD.$$

The relation $rf\text{-}tso = rf_e$ is a restriction of the reads-from relation rf to write-read pairs stemming from different threads:

$$rf_e = \{(w, r) \in rf \mid (w, r) \notin po \wedge (r, w) \notin po\}.$$

Unlike SC, the model TSO relaxes the program order and the reads-from relation. The relaxation of the former does not contain write-read pairs. This means TSO allows for reordering consecutive write and read events. Intuitively, the reason is as follows. Once the thread sees a write event, it pushes the same to the buffer. But this does not mean that the event gets flushed into the memory immediately, the write is pending. A consecutive read event is performed by the thread immediately, either by an early read or by accessing the memory. Flushing the write event into the memory and performing the corresponding changes to a variable can be delayed until after the read. Hence, there is no need for a program order edge between the two events. The relaxation of rf represents that writes are only visible to other threads once they were flushed into the memory. Writes that are read within a thread do not have to be visible to other threads due to early reads. To illustrate the relaxed notion of memory behavior, we consider an example.

Example 9.25. We show that the history $h = \langle O, po, rf \rangle$ from Figure 9.3 is TSO-consistent. Recall that we have already seen in Example 9.8 that it is not SC-consistent. This illustrates that TSO is the more relaxed notion of memory.

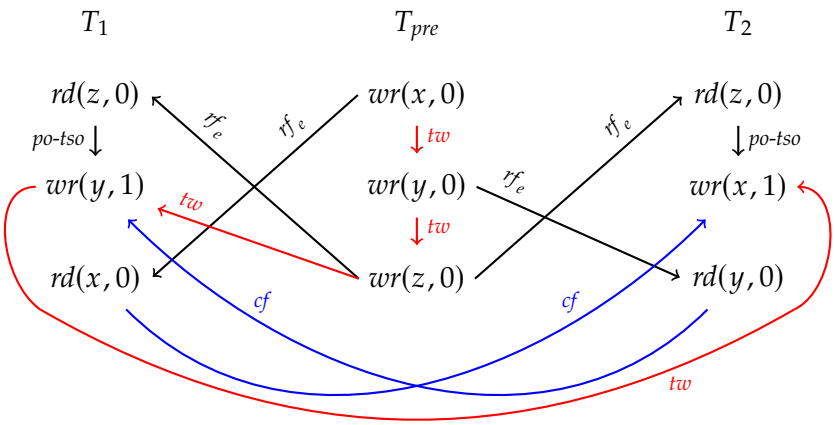


Figure 9.9: The graph $\mathcal{G}_{tso} = (O, po\text{-}tso \cup rf_e \cup tw \cup cf)$. Note that *po-tso* does not contain the edges $wr(y,1) \to rd(x,0)$ and $wr(x,1) \to rd(y,0)$. The relation rf_e is the complete reads-from relation. The graph is acyclic.

To show consistency, we need a total order tw on the writes events of h such that $\mathcal{G}_{loc}$ and $\mathcal{G}_{tso}$ are acyclic. We constructed the latter graph in Figure 9.9. It involves the preserved program order *po-tso* which does not contain write-read pairs and the relaxed reads-from relation rf_e which coincides with rf in this case. The corresponding total order tw is given by the transitive closure of the red edges. Note that the graph is actually acyclic. Moreover, the other graph $\mathcal{G}_{loc} = (O, po\text{-}loc \cup rf \cup tw \cup cf)$ is a subgraph of $\mathcal{G}_{tso}$ in this example. Hence, it is acyclic as well and consequently, h is TSO-consistent.

The observation from the example is true in general. The model TSO allows for strictly more behavior than SC does [307]. In fact, each execution that is consistent under SC is also consistent under TSO but there are examples that are TSO-consistent and not SC-consistent.

We apply our framework to TSO. Unlike for SC, the uniprocessor test is really needed this time. The graph $\mathcal{G}_{loc}$ is not necessarily a subgraph of $\mathcal{G}_{tso}$. The application yields an algorithm for TSO-consistency running in time $O(2^k \cdot k^2 \cdot n^2)$. The optimality of the algorithm is shown in Section 9.4.

Corollary 9.26. *The problem TSO-consistency can be solved in time* $O(2^k \cdot k^2 \cdot n^2)$.

Partial Store Order The second memory model by SPARC that we consider is called *Partial Store Order* (PSO) [305, 306]. It is weaker than TSO [307] since a thread now has one FIFO buffer per variable. Into this buffer, the thread can only push write events to the particular variable. The memory can nondeterministically choose to flush any of the buffers. Consequently, writes to distinct variables may arrive at the memory not following the program order. But this means that the program order among write events is only

preserved if they access the same variable. We need to relax it accordingly. Like for TSO, early reads can be performed from all the buffers of a thread. Hence, the reads-from relation of PSO is similar to TSO. We formalize.

Definition 9.27. The memory model *Partial Store Order* (PSO) is given by

$$\text{PSO} = (\textit{po-pso}, \textit{rf-pso}).$$

In the tuple, the preserved program order *po-pso* does neither contain write-read pairs nor write-write pairs. It is defined by

$$\textit{po-pso} = \textit{po} \setminus (WR \times RD \cup WR \times WR).$$

The relaxation of the reads-from relation is $\textit{rf-pso} = \textit{rf}_e$.

When applying our framework to PSO, we obtain a consistency algorithm for the memory model which runs in time $O(2^k \cdot k^2 \cdot n^2)$. Like for SC and TSO, we prove in Section 9.4 that the algorithm is optimal in the fine-grained sense.

Corollary 9.28. *The problem PSO-consistency can be solved in time* $O(2^k \cdot k^2 \cdot n^2)$.

Relaxed Memory Order We adjust our framework to also capture SPARC's *Relaxed Memory Order* (RMO) [305, 306]. The model needs an explicit test for *out of thin air values* and allows for so-called *load-load hazards*. We show how both modifications can be built into the framework without affecting the complexity of the resulting consistency algorithm.

While the program order and the reads-from relation are sufficient to describe SC, TSO, and PSO, the memory model RMO requires an additional *dependency relation* [11]. It models address and data dependencies among events in a history. For instance, if a read event has influence on the value written by a subsequent write event due to an arithmetic operation. We extend our notion of histories to also contain the dependency relation.

Definition 9.29. An *(extended) history* $h = \langle O, po, rf, dp \rangle$ consists of a program order po, a reads-from relation rf, and a *dependency relation*

$$dp \subseteq po \cap (RD \times O).$$

Note that dp always starts in a read event.

The dependency relation is required to detect values that come *out of thin air* [266]. These occur when a read event has to read a value that stems from a write event which is scheduled later. We given an example [11].

Example 9.30. Consider the history $h = \langle O, po, rf, dp \rangle$ given in Figure 9.10. The event $rd(x, 1)$ can only load 1 from x if $wr(x, 1)$ can provide it. However, the write event is scheduled later in the history since we have

$$rd(x,1) \xrightarrow{dp} wr(y,1) \xrightarrow{rf} rd(y,1) \xrightarrow{dp} wr(x,1).$$

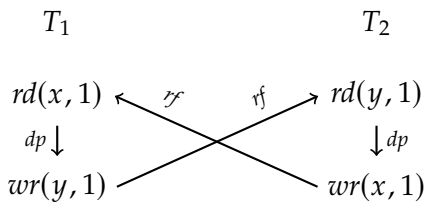


Figure 9.10: An extended history $h = \langle O, po, rf, dp \rangle$ with a dependency cycle. Executing the history h requires out of thin air values.

This yields a dependency cycle involving the relations $dp \cup rf$. Hence, an execution of the history would create the value 1 for x or y *out of thin air*.

The non-existence of out of thin air values like in Example 9.30 is an additional requirement that we need to add to our notion of consistency. To this end, we formulate it in terms of an acyclicity requirement.

Definition 9.31. Let $h = \langle O, po, rf, dp \rangle$ be an extended history. Then h does *not contain out of thin air values* if the following graph is acyclic:

$$\mathcal{G}_{thin} = (O, dp \cup rf).$$

The second step to describe consistency under RMO is to model so-called *load-load hazards* [11]. These occur when two reads of the same variable are scheduled contrary to program order. This behavior is indeed allowed by RMO. We define a relaxed relation that captures load-load hazards.

Definition 9.32. Let $h = \langle O, po, rf, dp \rangle$ be an extended history. The relation

$$po\text{-}loc_{llh} = po\text{-}loc \setminus RD \times RD$$

takes away the read-read pairs from *po-loc*. Hence, it does not order read events on the same variable and allows for reordering the same.

To obtain an algorithm from our framework for memory models that require an out of thin air check and allow for load-load hazards, we need to adjust the notion of consistency to this case. For the former, we need to add the acyclicity check for out of thin air values. For the latter, we have to weaken the uniprocessor check to allow for load-load hazards.

Definition 9.33. Let $h = \langle O, po, rf, dp \rangle$ be an arbitrary extended history and MM = (*po-mm*, *rf-mm*) some memory model. We call h MM-*consistent* if there exists a total order $tw \subseteq WR \times WR$ such that the following are satisfied.

(1) History h does not contain out of thin air values, $\mathcal{G}_{thin}$ is acyclic.

(2) The graph $\mathcal{G}_{\textit{loc-llh}} = (O, \textit{po-loc}_{\textit{llh}} \cup \textit{rf} \cup \textit{tw} \cup \textit{cf})$ is acyclic.

(3) The graph $\mathcal{G}_{\textit{mm}} = (O, \textit{po-mm} \cup \textit{rf-mm} \cup \textit{tw} \cup \textit{cf})$ is acyclic.

Our framework can be generalized to check whether an extended history is MM-consistent. In fact, we only need to make two minor adjustments. First, we add an initial test for the acyclicity of $\mathcal{G}_{\textit{thin}}$. Note that the graph does not depend on the total order *tw*. Hence, the acyclicity test can be performed before the actual algorithm from the framework starts. We employ Kahn's algorithm [226] to solve the task in time $O(n^2)$. Second, we replace the graph $\mathcal{G}_{\textit{loc}}$ from the uniprocessor check by the graph $\mathcal{G}_{\textit{loc-llh}}$. This ensures that Requirement (2) of the above definition is captured.

The changes to the framework do not affect its correctness. In fact, the crucial results, namely Lemma 9.22 and Lemma 9.16, still hold. Moreover, the running time of the resulting algorithm does not change. Applying the framework with out of thin air test and weaker uniprocessor check still yields a consistency algorithm running in time $O(2^k \cdot k^2 \cdot n^2)$.

Corollary 9.34. *Given an extended history $h = \langle O, \textit{po}, \textit{rf}, \textit{dp} \rangle$, the problem of checking MM-consistency of h can be solved in time $O(2^k \cdot k^2 \cdot n^2)$.*

We want to apply the adjusted framework in order to obtain a consistency algorithm for RMO. The formal definition of the model is given below.

Definition 9.35. The memory model *Relaxed Memory Oder* (RMO) is given by

$$\text{RMO} = (\textit{po-rmo}, \textit{rf-rmo}).$$

where $\textit{po-rmo} = \textit{dp}$ and $\textit{rf-rmo} = \textit{rf}_e$.

Now we are ready to apply the framework to RMO. Like for the other models, it yields a consistency algorithm running in time $O(2^k \cdot k^2 \cdot n^2)$.

Corollary 9.36. *The problem RMO-consistency can be solved in time $O(2^k \cdot k^2 \cdot n^2)$.*

9.4 Lower Bounds

We show that our framework provides optimal consistency algorithms for the memory models SC, TSO, and PSO. To this end, we prove that none of the consistency problems of the mentioned models can be solved in time $2^{o(k)}$ unless the ETH fails. Since the algorithms obtained in Section 9.3 match this lower bound, they are indeed optimal in the fine-grained sense. In Section 9.4.1, we present the lower bound for SC-Cons. Technically, it is a reduction from 3-SAT to the problem. Then, in Section 9.4.2, we adjust the reduction to also work for the problems TSO-Cons and PSO-Cons.

$T_0(x)$	$T_1(x)$	$T_0(\ell)$	$T_1(\ell)$	$T^1(C)$	$T^2(C)$	$T^3(C)$
$wr(x,0)$	$wr(x,1)$	$rd(x,0)$	$rd(x,1)$	$rd(\ell_3,0)$	$rd(\ell_1,0)$	$rd(\ell_2,0)$
		$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
		$wr(\ell,c)$	$wr(\ell,d)$	$rd(\ell_1,1)$	$rd(\ell_2,1)$	$rd(\ell_3,1)$
		$\downarrow$	$\downarrow$			
		$rd(x,0)$	$rd(x,1)$			

Figure 9.11: Part of the history h_φ for a particular variable $x \in Var$, a literal $\ell \in L$, and a clause $C = \ell_1 \vee \ell_2 \vee \ell_3$. The values of c and d depend on the literal ℓ. If $\ell = x$, then $c = 0, d = 1$. Otherwise, if $\ell = \neg x$, we have $c = 1, d = 0$.

9.4.1 Sequential Consistency

To prove that the problem SC-Cons cannot be solved in time $2^{o(k)}$, we construct a linear reduction from 3-SAT to the problem. Technically, let φ be an 3-SAT-instance with n many variables and m many clauses. We construct a history h_φ which contains $k = O(n+m)$ write events and which is consistent under SC if and only if φ is satisfiable. By invoking Lemma 5.6, we then obtain that an $2^{o(k)}$-time algorithm for SC-Cons is unlikely. We summarize.

Theorem 9.37. *Unless ETH fails, SC-Cons cannot be solved in time* $2^{o(k)}$.

It is left to construct the reduction. The main difficulty is the fact that the history h_φ needs to be data independent. Many reductions for proving hardness in memory consistency rely on the availability of multiple write events storing the same value in a variable [77, 180, 183]. Then the computational power comes from choosing a particular write event. In the data-independent case, we do not have multiple writes. Instead, it is only allowed to write a certain value to a variable once. This limits the source of computational power to interleaving the events of the history. However, we show that this is still sufficient to evaluate given 3-SAT-instances.

We fix some notations. Let the variables of φ be given by $Var = \{x_1, \ldots, x_n\}$ and let the clauses of φ be $C_1, \ldots, C_m$. By $L = \{\ell \mid \ell \in C_i\}$ we denote the set of literals. We assume that a literal ℓ occurs in exactly one clause. This means that we may have several copies of the same literal but with different names. The main idea of our reduction is to mimic an evaluation φ by an interleaving of the events in the history h_φ. To this end, we divide evaluating φ into three steps: (1) choose an evaluation of the variables x_i, (2) evaluate the literals $\ell \in L$ accordingly, and (3) check whether the clauses C_j are all satisfied. For each of these steps we have separate threads taking care of the task. Scheduling them in different orders will yield different evaluations.

Figure 9.11 provides an overview of these threads. Technically, the figure presents a part of the history h_φ that we are about to construct in the following. Note that it only shows a collection of individual threads without an explicit reads-from relation. However, the relation is given implicitly since each value is written at most once to a particular variable. We will refer to the figure several times when we elaborate on the details of the reduction.

Choosing an Evaluation To realize the first step, namely choosing an evaluation of the variables, we construct two threads $T_0(x)$ and $T_1(x)$ for each variable $x \in \mathit{Var}$ occurring in φ. These determine the evaluation of the variable x and consist of only a single write event. As illustrated in Figure 9.11, the thread $T_0(x)$ writes 0 to x, the thread $T_1(x)$ writes 1 to x.

If $T_0(x)$ gets scheduled before $T_1(x)$, variable x is evaluated to 1. Otherwise, the variable is evaluated to 0. This means that the thread that is scheduled later will determine the actual evaluation of x. Hence, choosing an evaluation amounts to ordering the write events accordingly. Note that write events that get overwritten are still required for generating certain dependency cycles. This will become more clear during the next two steps.

Evaluating the Literals In the second step, we propagate the evaluation of the variables to the literals. Let $\ell \in \{x, \neg x\}$ be a literal of φ on some variable $x \in \mathit{Var}$. We construct two threads that mimic the evaluation of ℓ.

The first thread is denoted by $T_0(\ell)$. It is responsible for evaluating ℓ when x got evaluated to 0 in the first step. To this end, $T_0(\ell)$ performs a read event $rd(x, 0)$, followed by a write $wr(\ell, c)$ and another read $rd(x, 0)$. It is illustrated in Figure 9.11. The value of c depends on the literal ℓ, we have

$$c = \begin{cases} 0, \text{ if } \ell = x \\ 1, \text{ if } \ell = \neg x. \end{cases}$$

Note that the read events $rd(x, 0)$ guard the write event $wr(\ell, c)$. This ensures that $T_0(\ell)$ can only run if x is already evaluated to 0 and once $T_0(\ell)$ is running, the evaluation of x cannot change until the thread finishes. In fact, if x is set to 1 during $T_0(\ell)$ is running, the thread is unable to perform its last read event.

The second thread for the literal is denoted by $T_1(\ell)$. It behaves similar to $T_0(\ell)$ and evaluates the literal ℓ when variable x got evaluated to 1 in the first step. To this end, $T_1(\ell)$ consists of the following events:

$$rd(x, 1) \xrightarrow{po} wr(\ell, d) \xrightarrow{po} rd(x, 1),$$

where $d = 1$ if $\ell = x$ and $d = 0$ if $\ell = \neg x$. Again, the two outer read events guard the write in between. Note that both threads, $T_0(\ell)$ and $T_1(\ell)$, cannot interfere. Like for the variable threads from Step (1), the thread that is scheduled later determines the actual evaluation of the literal ℓ. Hence,

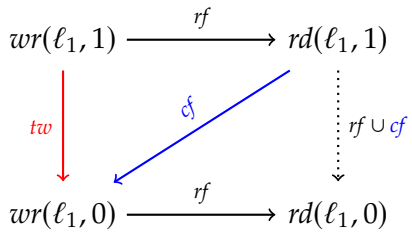


Figure 9.12: If the write event $wr(\ell_1, 1)$ precedes $wr(\ell_1, 0)$, then there is a corresponding tw-edge. It induces an cf-edge from $rd(\ell_1, 1)$ to $wr(\ell_1, 0)$. Hence, there is a path from $rd(\ell_1, 1)$ to $rd(\ell_1, 0)$ with edges from $rf \cup cf$.

choosing an evaluation of the variables and the literals amounts to choosing a total order tw on the write events of the history h_φ.

Evaluating the Clauses We are left with the third step, evaluating the clauses. We consider a particular clause $C = \ell_1 \vee \ell_2 \vee \ell_3$. For C, we construct three threads $T^1(C)$, $T^2(C)$, and $T^3(C)$ as they are shown in Figure 9.11. It is the task of these to ensure that at least one literal in C evaluates to 1.

To understand how the threads work, assume we have chosen an evaluation of the variables (and the literals) such that C is not satisfied — the literals ℓ_1, ℓ_2, and ℓ_3 all evaluate to 0. Let tw be a total order that corresponds to the chosen evaluation. Then, by the end of tw, the variables ℓ_1, ℓ_2, and ℓ_3 all store the value 0. Due to the construction of h_φ, ℓ_1 storing 0 implies that $wr(\ell_1, 1)$ preceded the write event $wr(\ell_1, 0)$ in tw. Hence, there is an edge

$$wr(\ell_1, 1) \xrightarrow{tw} wr(\ell_1, 0).$$

Hence intuitively, the read event $rd(\ell_1, 1)$ in $T^1(C)$ has to be scheduled before the read event $rd(\ell_1, 0)$ in $T^2(C)$. Otherwise, the value 1 gets overwritten by 0 and cannot be read anymore. In the history, this is reflected as follows. Since the read event $rd(\ell_1, 1)$ reads from $wr(\ell_1, 1)$ and the read event $rd(\ell_1, 0)$ reads from $wr(\ell_1, 0)$, we get two corresponding rf-edges. They are illustrated in Figure 9.12. As a consequence, we also obtain an cf-edge from $rd(\ell_1, 1)$ to $wr(\ell_1, 0)$ and therefore a path which reflects the order of the reads:

$$rd(\ell_1, 1) \xrightarrow{rf \cup cf} rd(\ell_1, 0).$$

Since the variables ℓ_2 and ℓ_3 also store 0 by the end of the total order tw, we obtain similar paths among the corresponding read events:

$$rd(\ell_2, 1) \xrightarrow{rf \cup cf} rd(\ell_2, 0) \text{ and } rd(\ell_3, 1) \xrightarrow{rf \cup cf} rd(\ell_3, 0).$$

Due to the construction of the threads $T^1(C)$, $T^2(C)$, and $T^3(C)$, there are program order edges between these reads. Putting them together with the above paths yields a cycle in $\mathcal{G}_{sc} = (O, po \cup rf \cup tw \cup cf)$:

$$rd(\ell_1, 1) \xrightarrow{rf \cup cf} rd(\ell_1, 0) \xrightarrow{po} rd(\ell_2, 1) \xrightarrow{rf \cup cf} rd(\ell_2, 0)$$
$$\xrightarrow{po} rd(\ell_3, 1) \xrightarrow{rf \cup cf} rd(\ell_3, 0) \xrightarrow{po} rd(\ell_1, 1).$$

Correctness The above shows that a non-satisfying assignment for formula φ yields a total order tw with a cyclic graph $\mathcal{G}_{sc}$. Assignments that satisfy φ amount to total orders that keep the graph acyclic. However, the construction of the precise orders is subtle. We provide details in Appendix B.4.4. The following lemma states the correctness of the reduction. Recall that for SC-consistency, it is sufficient that the graph $\mathcal{G}_{sc}$ is acyclic.

Lemma 9.38. *Formula φ is satisfiable if and only if the history h_φ is SC-consistent.*

It is left to argue that we have constructed a linear reduction. To this end, we determine the number of write events in h_φ. For each variable $x \in Var$ and each literal $\ell \in L$, we introduce two write events. We obtain that

$$k = 2 \cdot n + 2 \cdot |L|.$$

Since there are at most $3 \cdot m$ many literals in φ, we get that k is bounded by $2 \cdot n + 6 \cdot m$. This is linear in $n + m$ and completes the proof of Theorem 9.37. Moreover, note that the data domain employed in the above reduction is of size only 2. This implies that SC-Cons parameterized by the size of the domain is not in the class XP unless P = NP.

9.4.2 Total and Partial Store Order

We show that consistency checking under the memory models TSO and PSO admits a lower bound similar to SC-Cons. The idea is to adjust the reduction from the preceding section to the two relaxed models. In fact, we will add read events to the reduction to enforce a sequential behavior. Intuitively, we force the FIFO buffers of TSO and PSO to push each issued write immediately to the memory. Then the models behave like sequential consistency. The advantage is that the correctness argument of the reduction for SC then also applies here. Moreover, we do not change the number of write events and therefore obtain a linear reduction from 3-SAT to TSO-Cons and PSO-Cons. This yields a lower bound for both problems.

Theorem 9.39. *Unless ETH fails, neither the problem TSO-Cons nor the problem PSO-Cons can be solved by an algorithm running in time $2^{o(k)}$.*

Fix an 3-SAT-instance φ with n many variables and m many clauses. We consider the history h_φ constructed in the reduction from Section 9.4.1 under TSO and PSO. Recall that both models relax the program order. We have:

$$\begin{aligned} \textit{po-tso} &= \textit{po} \setminus \textit{WR} \times \textit{RD}, \\ \textit{po-pso} &= \textit{po} \setminus (\textit{WR} \times \textit{RD} \cup \textit{WR} \times \textit{WR}). \end{aligned}$$

While there are no *po*-edges of the form $\textit{WR} \times \textit{WR}$ in the history h_φ, there are *po*-edges leading from a write event to a read event. Indeed, the threads $T_0(\ell)$ and $T_1(\ell)$ for a particular literal ℓ contain a write event that is guarded by two reads. The threads were defined by

$$\begin{aligned} T_0(\ell) &: rd(x,0) \xrightarrow{po} wr(\ell,c) \xrightarrow{po} rd(x,0), \\ T_1(\ell) &: rd(x,1) \xrightarrow{po} wr(\ell,d) \xrightarrow{po} rd(x,1), \end{aligned}$$

where c and d were chosen accordingly to ℓ. The *po*-edges connecting the write events with the latter read events will vanish under TSO and PSO. This means that for a total order tw on the write events of h_φ, the graphs $\mathcal{G}_{tso}$ and $\mathcal{G}_{pso}$ are strict subgraphs of $\mathcal{G}_{sc}$ that miss out some edges. Phrased differently, a cycle in $\mathcal{G}_{sc}$ does not necessarily imply a cycle in $\mathcal{G}_{tso}$ or $\mathcal{G}_{pso}$. Hence, the history might be TSO or PSO-consistent while not being SC-consistent.

Construction We overcome this problem by splitting the threads $T_0(\ell)$ and $T_1(\ell)$ into two separate parts. The construction is shown in Figure 9.13. We still have the first part of $T_0(\ell)$ that reads $rd(x,0)$ and writes $wr(\ell,c)$. But the latter guarding read is replaced by a new thread $T_0'(\ell)$, defined by

$$rd(\ell,c) \xrightarrow{po} rd(x,0).$$

The construction for $T_1(\ell)$ is similar. The remaining part of h_φ is copied without further changes. We denote the obtained history by h_φ'. The advantage of the construction is that for any total order tw on the write events of h_φ', the three graphs $\mathcal{G}_{sc}$, $\mathcal{G}_{tso}$, and $\mathcal{G}_{pso}$ all coincide. In fact, there are no *po*-edges of the form $\textit{WR} \times \textit{WR}$ or $\textit{WR} \times \textit{RD}$ that could be deleted by considering TSO or PSO. This means that we actually enforce sequential behavior.

Correctness The number of write events in h_φ' is clearly linear in $n+m$ since we did only add read events to h_φ. Hence, we obtain a linear reduction and it is only left to show that the construction is indeed correct. We need to prove that φ is satisfiable if and only if h_φ' is SC-consistent. Note that the latter is equivalent to h_φ' being TSO-consistent and PSO-consistent. Instead of proving the equivalence directly, we use that the result already holds for h_φ by Lemma 9.38. Then we show that h_φ is SC-consistent if and only if the new history h_φ' is SC-consistent. The desired equivalence follows. Hence, to obtain correctness we need to prove the following lemma.

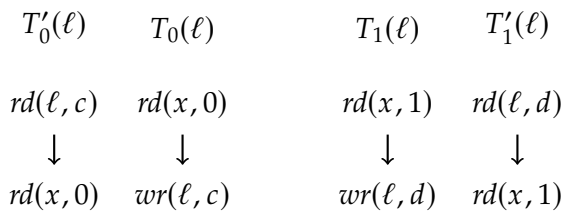


Figure 9.13: Parts of the newly constructed history h'_φ for some variable x and some literal $\ell = x/\neg x$. The values of c and d depend on the literal. If $\ell = x$, then we have $c = 0, d = 1$. Otherwise, we have $c = 1, d = 0$.

Lemma 9.40. *History h_φ is SC-consistent if and only if h'_φ is SC-consistent.*

Proof. The idea is simple. The history h_φ contains program order edges

$$wr(\ell, c) \xrightarrow{po} rd(x, 0) \text{ and } wr(\ell, d) \xrightarrow{po} rd(x, 1)$$

that do not occur in h'_φ. Here, x is a variable and $\ell = x/\neg x$. However, these edges can be mimicked by the history h'_φ via the two paths

$$wr(\ell, c) \xrightarrow{rf} rd(\ell, c) \xrightarrow{po} rd(x, 0),$$
$$wr(\ell, d) \xrightarrow{rf} rd(\ell, d) \xrightarrow{po} rd(x, 1).$$

This means if we have a total order tw on the write events of h_φ such that the graph $\mathcal{G}_{sc}(h_\varphi)$ over the history contains a cycle, then the cycle also appears in the graph $\mathcal{G}_{sc}(h'_\varphi)$ over h'_φ. Indeed, if the cycle contains one of the above program order edges, we replace it by the corresponding path in $\mathcal{G}_{sc}(h'_\varphi)$. Other edges of the cycle are anyhow present in $\mathcal{G}_{sc}(h'_\varphi)$.

The other direction also holds. A cycle in $\mathcal{G}_{sc}(h'_\varphi)$ induces a cycle in the graph $\mathcal{G}_{sc}(h_\varphi)$. The idea is similar. The occurrence of one of the above paths can be replaced by the corresponding po-edge in $\mathcal{G}_{sc}(h_\varphi)$. Other edges are anyhow available in the graph due to the construction of h'_φ.

The argument shows that acyclicity, and hence SC-consistency, is preserved across the histories. We provide a formal proof in Appendix B.4.5. □

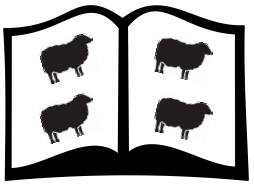
With Lemma 9.38 and Lemma 9.40, we obtain that φ is satisfiable if and only if h'_φ is SC or TSO or PSO-consistent. This proves Theorem 9.39 and shows that TSO-Cons and PSO-Cons parameterized by the size of the data domain behave like SC-Cons: both problems are not in XP unless P = NP.

Part III

Discussion

10. Related Work

There have been previous efforts in studying the fixed-parameter tractability or even the fine-grained complexity of verification tasks and decision problems from closely related fields. We discuss the corresponding references. The related work for bounded context switching, broadcast networks, leader contributor systems, and memory consistency has already been considered in the introductions to the chapters 6, 7, 8, and 9.



10.1 Program Verification

We discuss the related work on the fine-grained complexity of verification tasks. Some of the references are not necessarily *classical* parameterized complexity results or fine-grained analyses. Nevertheless, they may state hardness results or running times that we consider a contribution to the fine-grained complexity of program verification. We grouped the references into three fields: *Model Checking* in Section 10.1.1, *Concurrency Bug Prediction and Consistency* in Section 10.1.2, and *Static Analysis and Parsing* in Section 10.1.3.

10.1.1 Model Checking

In their paper [264], Madhusudan and Parlato employ arguments based on treewidth to prove the decidability of emptiness problems for various automata models common in model checking. Besides showing decidability, the authors provide fine-grained complexity upper bounds for each of the considered problems. In [127], Demri, Laroussinie, and Schnoebelen consider the parameterized complexity of various model checking problems for systems consisting of synchronized components — among them LTL and CTL model checking. The chosen parameter is the number of components. The authors show that all considered problems are intractable. Similar model checking questions are examined in the work of Göller [191]. This time the parameter is the size of the property that needs to be checked. The author shows that, even if the problem admits an FPT-algorithm, the running time will typically depend on the parameter in a non-elementary way. More generally, Flum and Grohe [165, 166] consider model checking problems for fragments of first order logic and characterize intractability in terms of these problems. The parameterized complexity of symbolic model checking questions was considered by de Haan and Szeider in [117].

10.1.2 Concurrency Bug Prediction and Consistency

The parameterized complexity of *Data Race Prediction* was considered by Mathur, Pavlogiannis, and Viswanathan [270]. A data race occurs when a shared resource is accessed simultaneously by different threads and at least one access is a write. The authors give an upper bound for the prediction of data races and they prove that, assuming the ETH, their algorithm is optimal. Moreover, they show that in general, the problem is W[1]-hard. The intractability was further examined by the same authors in [271]. They show that predicting races reversing synchronization operations like acquiring and releasing locks one time is already W[1]-hard. If the order of synchronization operations is respected, data race prediction can be solved in polynomial time. In [155], Farzan and Madhusudan consider the parameterized complexity of *Predicting Atomicity Violations*. These are bugs where the intended atomicity is interfered by another thread. When the prediction ignores the underlying synchronization mechanism, the authors provide a singly exponential FPT-algorithm for predicting these violations. If the mechanism is respected and based on locks, the problem is proven to be W[1]-hard.

Related to the prediction of concurrency bugs is the problem of checking the consistency of computations. The complexity of checking consistency under the memory model SC was first considered by Gibbons and Korach [183]. Although they mostly prove NP-hardness for several variants of the problem, they also provide fine-grained polynomial-time upper bounds which hold under certain assumptions. This was further investigated by Furbach, Meyer, Schneider, and Senftleben [180]. The authors considered different memory models and besides proving NP-hardness, they provided polynomial-time algorithms if certain parameters are assumed to be constant. The complexity of *Transactional Consistency* was considered by Biswas and Enea [43]. The authors provide an FPT-algorithm for checking this kind of consistency, a problem proven to be NP-hard by Papadimitriou in 1979 [288]. *Serializability* under TSO was considered by Enea and Farzan [149]. It is shown that checking a trace for TSO-serializability is NP-hard but also FPT.

10.1.3 Static Analysis and Parsing

In static analysis and parsing, fine-grained complexity results mostly focus on optimal polynomial-time algorithms. Since compilers are required to perform highly efficient, each bit or factor of an incorporated polynomial-time algorithm that can be improved is significant. But there are also algorithmic barriers that are unlikely to be breached. A prominent provider for lower bounds in this setting is the so-called *Boolean Matrix Multiplication Conjecture* BMM [1, 329]. This is the assumption that two $n \times n$ Boolean matrices cannot be multiplied by a *combinatorial* algorithm running in subcubic time: $O(n^{3-\varepsilon})$ for an $\varepsilon > 0$. The term *combinatorial* refers to a multiplication algorithm that

does not incorporate an algebraic approach like Strassen did [309]. Therefore, *combinatorial* is commonly interpreted as *non-Strassen-like algorithm* [30, 205].

Mathiasen and Pavlogiannis give a cubic upper bound for *Andersen's Pointer Analysis* [269]. They prove the optimality of their algorithm with an application of the BMM. A fine-grained analysis of *Dyck Reachability* was performed by Chatterjee, Choudhary, and Pavlogiannis in [80]. The problem is fundamental to static analysis and for instance builds the basis of alias analysis and data-dependence analysis. The authors show that Dyck reachability can be solved in almost linear time and they provide a corresponding lower bound for the problem showing that their algorithm is indeed optimal. The fine-grained complexity of *On-Demand Data Flow Analysis* was considered in [82]. The goal of such an analysis is to preprocess a given program in an offline phase and then to answer on-demand queries efficiently during an online phase, based on the preprocessed information. Roughly, the authors present a linear-time and space algorithm for on-demand analyses and argue that it is optimal. Fine-grained polynomial-time algorithms for various problems on recursive state machines were given in [83].

Parsing general context free grammars can be performed by the well-known CYK algorithm in cubic time [335]. Lee showed in [252] that a subcubic parser is unlikely to exist assuming the BMM. Note that there are subcubic parsers for the task like the one by Valiant [317]. However, Lee's result proves the existence of a combinatorial algorithm running faster than $O(n^3)$ unlikely and Valiant's parser relies on faster, Strassen-like, matrix multiplication. Knuth found that parsing deterministic context-free languages can be done in linear time by a so-called *LR-parser* [235]. LR-parsers are important since they are often employed to parse programming languages. Since Knuth's work, many more practical variations of LR-parsers were invented, for instance *Simple LR* [129], *LALR* [128, 130], and *Minimal LR* [287].

10.2 Automata Theory

Parameterized and fine-grained complexity analyses have been performed for various problems in automata theory. In [159], Fernau, Heggernes, and Villanger considered the complexity of the problems *Synchronizing word* and *DFA Consistency*. The former seeks for a word that is uniformly accepted from all initial states, the latter separates two sets of words by a DFA. The authors performed two fine-grained complexity analyses of the problems including FPT-upper bounds, ETH and SETH-based lower bounds, as well as kernel lower bounds. Fernau and Krebs, in [160], considered the complexity of *Universality*, *Equivalence*, and *Intersection Emptiness* for finite automata over various alphabets. They provide lower bounds for these problems relying on ETH and SETH. The so-called *Bounded DFA Intersection Emptiness Problem* was considered by Wareham [323]. The work provides a separation of

tractable and intractable parameters of the problem. Variants of the intersection emptiness problem for finite automata and related models have also been considered from a fine-grained perspective in [119, 312, 325].

Fine-grained polynomial upper and lower bounds for generalized Büchi games were provided by Chatterjee, Dvorák, Henzinger, and Loitzenbauer in [81]. Their lower bounds do not solely rely on the BMM but also on other hardness assumptions like the *Orthogonal Vectors Conjecture* (OVC) and the *Strong Triangle Conjecture* (STC). The fine-grained complexity of reachability on pushdown systems was considered by Chatterjee and Osang [84] and Hansen, Kjelstrøm, and Pavlogiannis [200]. The former work shows that the problem is unlikely to admit a subcubic combinatorial algorithm, even on simple *path-like* pushdown systems. The latter paper provides a quadratic-time algorithm for sparse pushdown systems with bounded stack height as well as a corresponding lower bound. The parameterized complexity of safety in threshold automata was examined by Balasubramanian [25]. The author proves the general problem to be W[1]-hard but restrictions of the problem that reflect practical instances are shown to be fixed-parameter tractable.

11. Future Work

We elaborate on problems and ideas that we consider future work. We differentiate between two directions. In Section 11.1 we discuss tasks within the realm of program verification that are suitable for a fine-grained complexity analysis. Moreover, we give ideas on extending the results obtained in this thesis. In Section 11.2, we explain our ideas on related problems. This includes a question on the space consumption of our verification algorithms and a meta problem from fine-grained complexity theory.

11.1 Verification Tasks

We consider two reachability problems of models that are well-known from program verification. Then we discuss possible extension of our results to checking consistency under certain memory models.

Reversal Bounded Counter Machines A *counter machine* is a finite automaton equipped with a set of integer *counters*. These are manipulated by the transitions of the machine: they can be increased, decreased, and even tested for zero. Reachability on the model is undecidable since already two counters suffice to simulate a Turing machine [278]. To regain decidability, Ibarra [214] introduced the class of *reversal bounded counter machines*. A *reversal* occurs on a counter if it swaps from an increasing to a decreasing phase or vice versa. The corresponding reachability problem then asks whether a state or a configuration can be reached within a given number of reversals.

The reachability problem for reversal bounded counter machines is known to be NP-complete [197]. Although Gurari and Ibarra investigate the intractability and tractability of parameters up to some extent in their work, a fine-grained complexity analysis of the problem is still missing. We aim for such an analysis. Significant parameters that could be investigated are the number of allowed reversals, the number of counters, and the size of the underlying finite automaton (its number of states).

Communication Free Petri Nets and Basic Parallel Processes *Communication free Petri Nets* [150, 208] are a restricted class of Petri Nets where each transition requires only one token for activation. Unlike for general Petri

Nets, where the reachability problem was recently proven to be Ackermann-complete [115, 253], the reachability problem for communication free Petri Nets is known to be NP-complete [150]. The importance of this model comes from the fact that communication free Petri Nets correspond to so-called *Basic Parallel Processes* [99, 178, 208, 209, 220, 221, 222], a simple class of recursive expressions capable of capturing concurrent behavior.

The NP-completeness makes a fine-grained complexity analysis of the reachability problem appealing. There are various parameters that could be of interest: the number of places, the number of transitions, or the highest number of tokens that occur while firing a transition.

Memory Consistency In Chapter 9, we developed a framework that yields consistency algorithms and we applied it to the memory models SC, TSO, PSO, and RMO. We would like to explore the limitations of the framework and examine whether it can be instantiated to other memory models like POWER as well. If such an application is not immediate, we want to generalize the framework accordingly. This requires a deeper understanding of the CAT [12] formulation of the desired memory models and how much expressiveness of CAT we can handle with the framework.

A further idea with the framework is to develop a tool for checking consistency. Since our algorithm runs in time $O^*(2^k)$, where k is the number of write events of the given history, a preprocessing that reduces this parameter would significantly help to improve the algorithm on practical examples. Such a preprocessing was developed and implemented by Bouajjani, Enea, Erradi, and Zennou [336]. In the corresponding tool, the authors currently combine their preprocessing with an enumeration that takes time $O^*(k^k)$. We believe that replacing the enumeration by our $O^*(2^k)$-time dynamic programming algorithm would result in a tool that is fast on larger instances.

Concerning the parameterized complexity of checking consistency, two further questions remained open. First, it is not yet clear how the number of variables in the given history affects the parameterized complexity of the problem. Gibbons and Korach [183] have shown that checking consistency with two variables is already NP-hard but their reduction heavily relies on the fact that the given history is not data-independent. We believe that proving the potential intractability of the parameter in the data-independent case requires a quite clever construction like for the number of threads [270].

The second question is how the fine-grained complexity of checking consistency, measured in the number of write events k, behaves in the data-dependent case. We know that there is an algorithm for this case which runs in time $O^*(k^k)$. It iterates over all total orders of write events and then greedily reads. However, it is not yet clear whether our $O^*(2^k)$-time algorithm can be generalized to this case or whether there is a lower bound of the form $2^{o(k \cdot \log(k))}$. The former would require a more powerful FPT-technique

which tames the complexity stemming from the missing reads-from relation. The latter would require a clever reduction, for instance from $k \times k$-Clique.

11.2 Further Problems

We discuss ideas on two related problems. The first idea concerns the space consumption of our verification algorithms and whether it can be reduced, the second is on the expressiveness of (fast) subset convolution.

Space Usage Most algorithms that rely on dynamic programming, also the ones presented in this thesis, require exponential space. This is typically a bottleneck. In their work [261], Lokshtanov and Nederlof present techniques to reduce the space consumption of dynamic programming algorithms significantly — to polynomial space — without increasing the running time by too much. The main idea is to perform an *algebraization*. This requires the application of fast transforms, like the ones presented in Section 3.3.1, to translate the underlying problem or structure into the algebraic world.

We would like to figure out to what extend the techniques presented by Lokshtanov and Nederlof can be applied in our setting, as space consumption is often critical for verification algorithms. Especially for the problem Shuffle Mem and for checking consistency under a given memory model, we would like to investigate a potential application. Currently, we solve the former problem in time $O^*(2^k)$ via fast subset convolution but we require exponential space. With the techniques of [261], there might be a more elaborate algebraization that allows for an $O^*(2^k)$-time and polynomial-space algorithm. For checking consistency, we currently employ a dynamic programming algorithm that requires exponential space. With an algebraization, we hope to reduce the required memory to polynomial space. It would not only be rather surprising but also mathematically very appealing if an algebraization could help to obtain a more efficient algorithm for checking memory consistency.

Expressiveness of Subset Convolution In [89], we considered a new logging procedure for hardware monitoring. Roughly, a signal is traced on a certain interval of *clock cycles* and each such cycle is associated with a bitvector from a set V that uniquely identifies it. Any time the signal turns from 0 to 1 or vice versa, the logging procedure adds up the corresponding bit vector. As a result, we obtain a *target vector* t and the precise number of signal flips k. The corresponding testing problem for the procedure then asks whether a given target vector t can be reached by adding up exactly k vectors from V.

This testing problem can be understood as an *exact coverability problem* over the field $\mathbb{F}_2$ and is clearly NP-hard. In [92], we considered its fine-grained complexity. We solved the problem with an extension of subset convolution faster than the naive dynamic programming algorithm. This extension is

capable of expressing the requirement of having *exactly k distinct vectors*, which is not immediate. This raised the question of how much requirements we can express or compute with the convolution without increasing the running time of either fast subset convolution [48] — in the case of the union operator on sets, or of the Walsh-Hadamard transform [8] — in the case of addition/XOR. At best we would like to have an algebraic description of the properties or the domain where the convolution operator can be computed without harming the running time. As we have already seen for Shuffle Mem and the above problem, this could have applications in testing.

12. Conclusion

We have presented a comprehensive study of the fine-grained complexity of program verification. To this end, we examined how well-known techniques from parameterized and fine-grained complexity theory carry over to typical safety, liveness, and testing problems from program verification. Our results do not only provide a detailed picture of the complexity of each considered problem but also provably optimal verification algorithms for the same.

First, we considered the complexity of BCS and established an $O^*(m^{cs} \cdot 2^{cs})$-time algorithm and a lower bound of $m^{o(cs/\log(cs))}$. The algorithm shows that BCS benefits from FPT-techniques like fast subset convolution. Further, we have seen that lower bound assumptions like the SCON and the ETH, as well as intractability theory known from parameterized complexity theory help to understand the complexity of BCS in much finer detail.

We also examined the fine-grained complexity of safety and liveness in parameterized systems, namely in broadcast networks and leader contributor systems. For the former we considered the liveness verification problems BNL and FBNL and provided a polynomial-time algorithm for both, thereby closing a long-standing gap in their complexities. For the latter, we considered the safety verification problem LCR. We gave two algorithms, one running in time $(d + l)^{O(d+l)}$ and one running in time $O^*(2^c)$. Both algorithms were matched by corresponding lower bounds, one based on the ETH, one based on the SCON. Other parameterizations of LCR were shown to be intractable. Subsequently, we considered LCL, the liveness verification problem of leader contributor systems. We showed that LCL can be decomposed into LCR and a cycle finding problem CYC. By solving CYC in polynomial time, we concluded that the complexity of LCR and LCL match up to a polynomial factor.

Finally, we considered the problem of checking consistency of computations over a shared memory. We provided a framework that takes a memory model and yields a consistency algorithm that runs in time $O^*(2^k)$. We applied the framework to the models SC, TSO, PSO, and RMO. For the former three models, we matched the running time of the obtained consistency algorithm with a corresponding lower bound relying on the ETH.

Altogether, our results show that techniques from parameterized and fine-grained complexity help to understand the complexity of program verification in more detail. Not only are these techniques capable of providing new and optimal approaches to verification, they also help to distinguish between simple and intricate instances. This matches our goals formulated initially.

Bibliography

[1] A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443. IEEE, 2014.

[2] P. A. Abdulla, M. F. Atig, and O. Rezine. Verification of directed acyclic ad hoc networks. In *FORTE*, volume 7892 of *LNCS*, pages 193–208. Springer, 2013.

[3] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, volume 7795 of *LNCS*, pages 324–338. Springer, 2013.

[4] P. A. Abdulla, A. P. Sistla, and M. Talupur. Model checking parameterized systems. In *Handbook of Model Checking*, pages 685–725. Springer, 2018.

[5] K. R. Abrahamson, J. A. Ellis, M. R. Fellows, and M. E. Mata. On the complexity of fixed parameter problems. In *FOCS*, pages 210–215. IEEE, 1989.

[6] F. N. Abu-Khzam. A kernelization algorithm for d-hitting set. *J. Comput. Syst. Sci.*, 76(7):524–531, 2010.

[7] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *SPAA*, pages 251–260. ACM, 1993.

[8] N. Ahmed and K. R. Rao. *Orthogonal Transforms for Digital Signal Processing*. Springer, 1975.

[9] H. Akhiani, D. Doligez, P. Harter, L. Lamport, J. Scheid, M. R. Tuttle, and Y. Yu. Cache coherence verification with TLA+. In *FM*, volume 1709 of *LNCS*, pages 1871–1872. Springer, 1999.

[10] M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider. *Distributed Systems: Methods and Tools for Specification. An Advanced Course*. Springer, 1985.

[11] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods Syst. Des.*, 41(2):178–210, 2012.

[12] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[13] N. Alon, R. Yuster, and U. Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[14] B. Alpern, A. J. Demers, and F. B. Schneider. Safety without stuttering. *Inf. Process. Lett.*, 23(4):177–180, 1986.

[15] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[16] R. Alur, K. L. McMillan, and D. A. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.

[17] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[18] S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[19] M. F. Atig. Global model checking of ordered multi-pushdown systems. In *FSTTCS*, volume 8 of *LIPIcs*, pages 216–227. Schloss Dagstuhl, 2010.

[20] M. F. Atig, A. Bouajjani, K. N. Kumar, and P. Saivasan. On bounded reachability analysis of shared memory systems. In *FSTTCS*, volume 29 of *LIPIcs*, pages 611–623. Schloss Dagstuhl, 2014.

[21] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS*, volume 5505 of *LNCS*, pages 107–123. Springer, 2009.

[22] M. F. Atig, A. Bouajjani, and T. Touili. Analyzing asynchronous programs with preemption. In *FSTTCS*, volume 2 of *LIPIcs*, pages 37–48. Schloss Dagstuhl, 2008.

[23] M. F. Atig, A. Bouajjani, and T. Touili. On the reachability analysis of acyclic networks of pushdown systems. In *CONCUR*, volume 5201 of *LNCS*, pages 356–371. Springer, 2008.

[24] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[25] A. R. Balasubramanian. Parameterized complexity of safety of threshold automata. In *FSTTCS*, volume 182 of *LIPIcs*, pages 37:1–37:15. Schloss Dagstuhl, 2020.

[26] A. R. Balasubramanian, N. Bertrand, and N. Markey. Parameterized verification of synchronization in constrained reconfigurable broadcast networks. In *TACAS*, volume 10806 of *LNCS*, pages 38–54. Springer, 2018.

[27] R. Balasubramanian, M. R. Fellows, and V. Raman. An improved fixed-parameter algorithm for vertex cover. *Inf. Process. Lett.*, 65(3):163–168, 1998.

[28] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS*, volume 2031 of *LNCS*, pages 158–173. Springer, 2001.

[29] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.

[30] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *J. ACM*, 59(6):32:1–32:23, 2012.

[31] N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In *ISAAC*, volume 1741 of *LNCS*, pages 247–258. Springer, 1999.

[32] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold, 1990.

[33] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[34] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[35] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, 1962.

[36] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[37] N. Bertrand, P. Fournier, and A. Sangnier. Playing with probabilities in reconfigurable broadcast networks. In *FOSSACS*, volume 8412 of *LNCS*, pages 134–148. Springer, 2014.

[38] N. Bertrand, P. Fournier, and A. Sangnier. Distributed local strategies in broadcast networks. In *CONCUR*, volume 42 of *LIPIcs*, pages 44–57. Schloss Dagstuhl, 2015.

[39] D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *(ICSE*, pages 326–335. IEEE, 2004.

[40] T. C. Biedl and M. Vatshelle. The point-set embeddability problem for plane graphs. *IJCGA*, 23(4-5):357–396, 2013.

[41] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC*, pages 317–320. ACM, 1999.

[42] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[43] R. Biswas and C. Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA):165:1–165:28, 2019.

[44] A. Björklund. Determinant sums for undirected hamiltonicity. In *FOCS*, pages 173–182. IEEE, 2010.

[45] A. Björklund. Counting perfect matchings as fast as Ryser. In *SODA*, pages 914–921. SIAM, 2012.

[46] A. Björklund. Exploiting sparsity for bipartite hamiltonicity. In *ISAAC*, volume 123 of *LIPIcs*, pages 3:1–3:11. Schloss Dagstuhl, 2018.

[47] A. Björklund, B. Gupt, and N. Quesada. A faster Hafnian formula for complex matrices and its benchmarking on a supercomputer. *ACM J. Exp. Algorithmics*, 24(1):1.11:1–1.11:17, 2019.

[48] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets Möbius: fast subset convolution. In *STOC*, pages 67–74. ACM, 2007.

[49] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Computing the Tutte polynomial in vertex-exponential time. In *FOCS*, pages 677–686. IEEE, 2008.

[50] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Narrow sieves for parameterized paths and packings. *J. Comput. Syst. Sci.*, 87:119–139, 2017.

[51] A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009.

[52] A. Björklund, P. Kaski, and I. Koutis. Directed hamiltonicity and out-branchings via generalized Laplacians. In *ICALP*, volume 80 of *LIPIcs*, pages 91:1–91:14. Schloss Dagstuhl, 2017.

[53] A. Björklund, P. Kaski, and L. Kowalik. Constrained multilinear detection and generalized graph motifs. *Algorithmica*, 74(2):947–967, 2016.

[54] A. Björklund, P. Kaski, and R. Williams. Solving systems of polynomial equations over GF(2) by a parity-counting self-reduction. In *ICALP*, volume 132 of *LIPIcs*, pages 26:1–26:13. Schloss Dagstuhl, 2019.

[55] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[56] H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin. On problems without polynomial kernels (extended abstract). In *ICALP*, volume 5125 of *LNCS*, pages 563–574. Springer, 2008.

[57] H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin. On problems without polynomial kernels. *J. Comput. Syst. Sci.*, 75(8):423–434, 2009.

[58] H. L. Bodlaender, B. M. P. Jansen, and S. Kratsch. Cross-composition: A new technique for kernelization lower bounds. In *STACS*, volume 9 of *LIPIcs*, pages 165–176. Schloss Dagstuhl, 2011.

[59] H. L. Bodlaender, B. M. P. Jansen, and S. Kratsch. Kernelization lower bounds by cross-composition. *SIAM J. Discret. Math.*, 28(1):277–305, 2014.

[60] J.A. Bondy and V. Chvatal. A method in graph theory. *Discrete Mathematics*, 15(2):111 – 135, 1976.

[61] A. Bouajjani, M. Emmi, and G. Parlato. On sequentializing concurrent programs. In *SAS*, volume 6887 of *LNCS*, pages 129–145. Springer, 2011.

[62] A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza. On verifying causal consistency. In *POPL*, pages 626–638. ACM, 2017.

[63] A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *POPL*, pages 285–296. ACM, 2014.

[64] A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In *ICALP*, volume 6756 of *LNCS*, pages 428–440. Springer, 2011.

[65] P. Bouyer, N. Markey, M. Randour, A. Sangnier, and D. Stan. Reachability in networks of register protocols under stochastic schedulers. In *ICALP*, volume 55 of *LIPIcs*, pages 106:1–106:14. Schloss Dagstuhl, 2016.

[66] R. N. Bracewell. *The Fourier Transform and Its Applications*. McGraw-Hill, third edition, 2000.

[67] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439. IEEE, 1990.

[68] J. F. Buss and J. Goldsmith. Nondeterminism within P. *SIAM J. Comput.*, 22(3):560–572, 1993.

[69] J. M. Byskov. Enumerating maximal independent sets with applications to graph colouring. *Operations Research Letters*, 32(6):547 – 556, 2004.

[70] J. R. Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[71] J. R. Büchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[72] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, 2008.

[73] L. Cai, J. Chen, R. G. Downey, and M. R. Fellows. Advice classes of parameterized tractability. *Ann. Pure Appl. Log.*, 84(1):119–138, 1997.

[74] L. Cai, J. Chen, R. G. Downey, and M. R. Fellows. On the parameterized complexity of short computation and factorization. *Arch. Math. Log.*, 36(4-5):321–337, 1997.

[75] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NFM*, volume 9058 of *LNCS*, pages 3–11. Springer, 2015.

[76] G. Calin, E. Derevenetc, R. Majumdar, and R. Meyer. A theory of partitioned global address spaces. In *FSTTCS*, volume 24 of *LIPIcs*, pages 127–139. Schloss Dagstuhl, 2013.

[77] J.F. Cantin, M.H. Lipasti, and J.E. Smith. The complexity of verifying memory coherence and consistency. *TPDS*, 16(7):663–671, 2005.

[78] M. Cesati. Perfect code is $W[1]$-complete. *Inf. Process. Lett.*, 81(3):163–168, 2002.

[79] M. Cesati. The turing way to parameterized complexity. *J. Comput. Syst. Sci.*, 67(4):654–685, 2003.

[80] K. Chatterjee, B. Choudhary, and A. Pavlogiannis. Optimal Dyck reachability for data-dependence and alias analysis. *Proc. ACM Program. Lang.*, 2(POPL):30:1–30:30, 2018.

[81] K. Chatterjee, W. Dvorák, M. Henzinger, and V. Loitzenbauer. Conditionally optimal algorithms for generalized Büchi games. In *MFCS*, volume 58 of *LIPIcs*, pages 25:1–25:15. Schloss Dagstuhl, 2016.

[82] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, and A. Pavlogiannis. Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In *ESOP*, volume 12075 of *LNCS*, pages 112–140. Springer, 2020.

[83] K. Chatterjee, R. Ibsen-Jensen, A. Pavlogiannis, and P. Goyal. Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In *POPL*, pages 97–109. ACM, 2015.

[84] K. Chatterjee and G. Osang. Pushdown reachability with constant treewidth. *Inf. Process. Lett.*, 122:25–29, 2017.

[85] J. Chen, B. Chor, M. R. Fellows, X. Huang, D. W. Juedes, I. A. Kanj, and G. Xia. Tight lower bounds for certain parameterized NP-hard problems. *Inf. Comput.*, 201(2):216–231, 2005.

[86] J. Chen, X. Huang, I. A. Kanj, and G. Xia. Linear FPT reductions and computational lower bounds. In *STOC*, pages 212–221. ACM, 2004.

[87] J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theor. Comput. Sci.*, 411(40-42):3736–3756, 2010.

[88] J. Chen, S. Lu, S. Sze, and F. Zhang. Improved algorithms for path, matching, and packing problems. In *SODA*, pages 298–307. SIAM, 2007.

[89] P. Chini, R. Drechsler, H. M. Le, R. Massoud, R. Meyer, and P. Saivasan. Temporal tracing of on-chip signals using timeprints. In *DAC*. ACM, 2019.

[90] P. Chini and F. Furbach. Petri net invariant synthesis. In *To appear in NETYS*, 2021.

[91] P. Chini, J. Kolberg, A. Krebs, R. Meyer, and P. Saivasan. On the complexity of bounded context switching. In *ESA*, volume 87 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl, 2017.

[92] P. Chini, R. Massoud, R. Meyer, and P. Saivasan. Fast witness counting. *CoRR*, abs/1807.05777, 2018.

[93] P. Chini, R. Meyer, and P. Saivasan. Fine-grained complexity of safety verification. In *TACAS*, volume 10806 of *LNCS*, pages 20–37. Springer, 2018.

[94] P. Chini, R. Meyer, and P. Saivasan. Complexity of liveness in parameterized systems. In *FSTTCS*, volume 150 of *LIPIcs*, pages 37:1–37:15. Schloss Dagstuhl, 2019.

[95] P. Chini, R. Meyer, and P. Saivasan. Liveness in broadcast networks. In *NETYS*, volume 11704 of *LNCS*, pages 52–66. Springer, 2019.

[96] P. Chini, R. Meyer, and P. Saivasan. Fine-grained complexity of safety verification. *J. Autom. Reason.*, 64(7):1419–1444, 2020.

[97] P. Chini, R. Meyer, and P. Saivasan. Liveness in broadcast networks. *Computing*, 2021.

[98] P. Chini and P. Saivasan. A framework for consistency algorithms. In *FSTTCS*, volume 182 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl, 2020.

[99] S. Christensen, Y. Hirshfeld, and F. Moller. Bisimulation equivalence is decidable for basic parallel processes. In *CONCUR*, volume 715 of *LNCS*, pages 143–157. Springer, 1993.

[100] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[101] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[102] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[103] E. M. Clarke, M. Talupur, and H. Veith. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS*, volume 4963 of *LNCS*, pages 33–47. Springer, 2008.

[104] Compaq. *Alpha Architecture Reference Manual*. Compaq Computer Corp., 2002.

[105] S A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[106] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[107] North American Electric Reliability Council. Technical analysis of the august 14, 2003, blackout: What happened, why, and what did we learn? Technical report, Final NERC Report, 2004.

[108] R. Crowston, G. Z. Gutin, M. Jones, V. Raman, and S. Saurabh. Parameterized complexity of MaxSat above average. In *LATIN*, volume 7256 of *LNCS*, pages 184–194. Springer, 2012.

[109] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *(ICSE*, pages 422–431. ACM, 2005.

[110] M. Cygan, H. Dell, D. Lokshtanov, D. Marx, J. Nederlof, Y. Okamoto, R. Paturi, S. Saurabh, and M. Wahlström. On problems as hard as CNF-SAT. In *CCC*, pages 74–84. IEEE, 2012.

[111] M. Cygan, H. Dell, D. Lokshtanov, D. Marx, J. Nederlof, Y. Okamoto, R. Paturi, S. Saurabh, and M. Wahlström. On problems as hard as CNF-SAT. *ACM Trans. Algorithms*, 12(3):41:1–41:24, 2016.

[112] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[113] M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J. M. M. van Rooij, and J. O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *FOCS*, pages 150–159. IEEE, 2011.

[114] M. Cygan, M. Pilipczuk, M. Pilipczuk, and J. O. Wojtaszczyk. Solving the 2-disjoint connected subgraphs problem faster than 2^n. *Algorithmica*, 70(2):195–207, 2014.

[115] W. Czerwinski and L. Orlikowski. Reachability in vector addition systems is Ackermann-complete. *CoRR*, abs/2104.13866, 2021.

[116] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[117] R. de Haan and S. Szeider. Parameterized complexity results for symbolic model checking of temporal logics. In *KR*, pages 453–462. AAAI Press, 2016.

[118] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[119] M. de Oliveira Oliveira and M. Wehar. On the fine grained complexity of finite automata non-emptiness of intersection. In *DLT*, volume 12086 of *LNCS*, pages 69–82. Springer, 2020.

[120] F. Dederichs and R. Weber. Safety and liveness from a methodological point of view. *Inf. Process. Lett.*, 36(1):25–30, 1990.

[121] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *CAV*, volume 1855 of *LNCS*, pages 53–68. Springer, 2000.

[122] G. Delzanno, J.-F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded Java programs. In *TACAS*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002.

[123] G. Delzanno, A. Sangnier, R. Traverso, and G. Zavattaro. On the complexity of parameterized reachability in reconfigurable broadcast networks. In *FSTTCS*, volume 18 of *LIPIcs*, pages 289–300. Schloss Dagstuhl, 2012.

[124] G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR*, volume 6269 of *LNCS*, pages 313–327. Springer, 2010.

[125] G. Delzanno, A. Sangnier, and G. Zavattaro. On the power of cliques in the parameterized verification of ad hoc networks. In *FOSSACS*, volume 6604 of *LNCS*, pages 441–455. Springer, 2011.

[126] G. Delzanno, A. Sangnier, and G. Zavattaro. Verification of ad hoc networks with node and communication failures. In *FORTE*, volume 7273 of *LNCS*, pages 235–250. Springer, 2012.

[127] S. Demri, F. Laroussinie, and P. Schnoebelen. A parametric analysis of the state explosion problem in model checking. In *STACS*, volume 2285 of *LNCS*, pages 620–631. Springer, 2002.

[128] F. DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, Massachusetts Institute of Technology, 1969.

[129] F. DeRemer. Simple lr(k) grammars. *Commun. ACM*, 14(7):453–460, 1971.

[130] F. DeRemer and T. J. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649, 1982.

[131] E. Derevenetc and R. Meyer. Robustness against power is PSPACE-complete. In *ICALP*, volume 8573 of *LNCS*, pages 158–170. Springer, 2014.

[132] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[133] G. A. Dirac. Some theorems on abstract graphs. *Proceedings of the London Mathematical Society*, s3-2(1):69–81, 1952.

[134] M. Dom, D. Lokshtanov, and S. Saurabh. Incompressibility through colors and ids. In *ICALP*, volume 5555 of *LNCS*, pages 378–389. Springer, 2009.

[135] R. G. Downey and M. R. Fellows. Fixed-parameter intractability. In *Structure in Complexity Theory Conference*, pages 36–49. IEEE, 1992.

[136] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness. In *Manitoba Conference on Numerical Mathematics and Computing*, number Bd. 1 in Congressus numerantium, pages 161–178. Utilitas Mathematica Publishing, 1992.

[137] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness I: basic results. *SIAM J. Comput.*, 24(4):873–921, 1995.

[138] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theor. Comput. Sci.*, 141(1&2):109–131, 1995.

[139] R. G. Downey and M. R. Fellows. Parameterized computational feasibility. In *Cornell Workshop on Feasible Mathematics*, Feasible mathematics II, pages 219–244. Birkhäuser, 1995.

[140] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[141] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

[142] S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[143] A. Durand-Gasselin, J. Esparza, P. Ganty, and R. Majumdar. Model checking parameterized asynchronous shared-memory systems. In *CAV*, volume 9206 of *LNCS*, pages 67–84. Springer, 2015.

[144] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[145] S. Eilenberg. *Automata, languages, and machines. A*. Pure and applied mathematics. Academic Press, 1974.

[146] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems. In *CAV*, volume 1102 of *LNCS*, pages 87–98. Springer, 1996.

[147] E. A. Emerson and K. S. Namjoshi. On model checking for nondeterministic infinite-state systems. In *LICS*, pages 70–80. IEEE, 1998.

[148] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[149] C. Enea and A. Farzan. On atomicity in presence of non-atomic writes. In *TACAS*, volume 9636 of *LNCS*, pages 497–514. Springer, 2016.

[150] J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundam. Informaticae*, 31(1):13–25, 1997.

[151] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE, 1999.

[152] J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. In *CAV*, volume 8044 of *LNCS*, pages 124–140. Springer, 2013.

[153] J. Esparza, P. Ganty, and T. Poch. Pattern-based verification for multi-threaded programs. *ACM TOPLAS*, 36(3):9:1–9:29, 2014.

[154] J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[155] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *TACAS*, volume 5505 of *LNCS*, pages 155–169. Springer, 2009.

[156] M. R. Fellows, F. V. Fomin, D. Lokshtanov, E. Losievskaja, F. A. Rosamond, and S. Saurabh. Distortion is fixed parameter tractable. In *ICALP*, volume 5555 of *LNCS*, pages 463–474. Springer, 2009.

[157] M. R. Fellows and M. A. Langston. An analogue of the Myhill-Nerode theorem and its use in computing finite-basis characterizations. In *FOCS*, pages 520–525. IEEE, 1989.

[158] M. R. Fellows and M. A. Langston. On search, decision and the efficiency of polynomial-time algorithms. In *STOC*, pages 501–512. ACM, 1989.

[159] H. Fernau, P. Heggernes, and Y. Villanger. A multi-parameter analysis of hard problems on deterministic finite automata. *J. Comput. Syst. Sci.*, 81(4):747–765, 2015.

[160] H. Fernau and A. Krebs. Problems on finite automata and the exponential time hypothesis. In *CIAA*, volume 9705 of *LNCS*, pages 89–100. Springer, 2016.

[161] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, volume 2305 of *LNCS*, pages 262–277. Springer, 2002.

[162] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003.

[163] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[164] R. W. Floyd. Assigning meanings to programs. In *American Mathematical Society Symposia on Applied Mathematics*, pages 19–31, 1967.

[165] J. Flum and M. Grohe. Fixed-parameter tractability, definability, and model-checking. *SIAM J. Comput.*, 31(1):113–145, 2001.

[166] J. Flum and M. Grohe. Model-checking problems as a basis for parameterized intractability. In *LICS*, pages 388–397. IEEE, 2004.

[167] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006.

[168] F. V. Fomin, F. Grandoni, A. V. Pyatkin, and A. A. Stepanov. Combinatorial bounds via measure and conquer: Bounding minimal dominating sets and applications. *ACM Trans. Algorithms*, 5(1):9:1–9:17, 2008.

[169] F. V. Fomin, P. Kaski, D. Lokshtanov, F. Panolan, and S. Saurabh. Parameterized single-exponential time polynomial space algorithm for steiner tree. In *ICALP*, volume 9134 of *LNCS*, pages 494–505. Springer, 2015.

[170] F. V. Fomin and D. Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. Springer, 2010.

[171] F. V. Fomin, D. Kratsch, and G. J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *Graph-Theoretic Concepts in Computer Science*, volume 3353 of *LNCS*, pages 245–256. Springer, 2004.

[172] F. V. Fomin, D. Lokshtanov, S. Saurabh, and M. Zehavi. *Kernelization - Theory of Parameterized Preprocessing*. Cambridge University Press, 2019.

[173] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[174] M. Fortin, A. Muscholl, and I. Walukiewicz. Model-checking linear-time properties of parametrized asynchronous shared-memory pushdown systems. In *CAV*, volume 8044 of *LNCS*, pages 155–175. Springer, 2017.

[175] L. Fortnow and R. Santhanam. Infeasibility of instance compression and succinct pcps for NP. In *STOC*, pages 133–142. ACM, 2008.

[176] P. Fournier. *Parameterized verification of networks of many identical processes*. PhD thesis, University of Rennes 1, 2015.

[177] M. Frances and A. Litman. On covering problems of codes. *Theory Comput. Syst.*, 30(2):113–119, 1997.

[178] S. B. Fröschle, P. Jancar, S. Lasota, and Z. Sawa. Non-interleaving bisimulation equivalences on basic parallel processes. *Inf. Comput.*, 208(1):42–62, 2010.

[179] B. Fuchs, W. Kern, D. Mölle, S. Richter, P. Rossmanith, and X. Wang. Dynamic programming for minimum Steiner trees. *Theory Comput. Syst.*, 41(3):493–500, 2007.

[180] F. Furbach, R. Meyer, K. Schneider, and M. Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM TECS*, 14(4):63:1–63:25, 2015.

[181] M. Fürer. Faster integer multiplication. In *STOC*, pages 57–66. ACM, 2007.

[182] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[183] P. B. Gibbons and E. Korach. Testing shared memories. *SICOMP*, 26(4):1208–1244, 1997.

[184] P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV*, volume 531 of *LNCS*, pages 176–185. Springer, 1990.

[185] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54. ACM, 2007.

[186] P. Godefroid. Higher-order test generation. In *PLDI*, pages 258–269. ACM, 2011.

[187] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[188] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *EMSOFT*, pages 207–216. ACM, 2008.

[189] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.

[190] O. Goldreich. *Computational complexity - a conceptual perspective*. Cambridge University Press, 2008.

[191] S. Göller. The fixed-parameter tractability of model checking concurrent systems. In *CSL*, volume 23 of *LIPIcs*, pages 332–347. Schloss Dagstuhl, 2013.

[192] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin-Madison, 1991.

[193] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277. ACM, 2007.

[194] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Data reduction and exact algorithms for clique cover. *ACM J. Exp. Algorithmics*, 13, 2008.

[195] J. Gramm, R. Niedermeier, and P. Rossmanith. Fixed-parameter algorithms for CLOSEST STRING and related problems. *Algorithmica*, 37(1):25–42, 2003.

[196] J. Guo and R. Niedermeier. Invitation to data reduction and problem kernelization. *SIGACT News*, 38(1):31–45, 2007.

[197] E. M. Gurari and O. H. Ibarra. The complexity of decision problems for finite-turn multicounter machines. In *ICALP*, volume 115 of *LNCS*, pages 495–505. Springer, 1981.

[198] M. Hague. Parameterised pushdown systems with non-atomic writes. In *FSTTCS*, volume 13 of *LIPIcs*, pages 457–468. Schloss Dagstuhl, 2011.

[199] M. Hague, R. Meyer, S. Muskalla, and M. Zimmermann. Parity to safety in polynomial time for pushdown and collapsible pushdown systems. In *MFCS*, volume 117 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl, 2018.

[200] J. C. Hansen, A. H. Kjelstrøm, and A. Pavlogiannis. Tight bounds for reachability problems on one-counter and pushdown systems. *Inf. Process. Lett.*, 171:106135, 2021.

[201] T. D. Hansen, H. Kaplan, O. Zamir, and U. Zwick. Faster k-SAT algorithms using biased-PPSZ. In *STOC*, pages 578–589. ACM, 2019.

[202] J. Hartmanis. Context-free languages and turing machine computations. In *Symposia in Applied Mathematics*, volume 19, pages 42–51, 1967.

[203] A. Heddaya and H. Sinha. Coherence, non-coherence and local consistency in distributed shared memory for parallel computing. Technical Report BU-CS-92-004, Boston University, 1992.

[204] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[205] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30. ACM, 2015.

[206] T. Hertli. 3-SAT faster and simpler - unique-SAT bounds for PPSZ hold in general. *SIAM J. Comput.*, 43(2):718–729, 2014.

[207] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[208] Y. Hirshfeld. Petri nets and the equivalence problem. In *CSL*, volume 832 of *LNCS*, pages 165–174. Springer, 1993.

[209] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial-time algorithm for deciding bisimulation equivalence of normed basic parallel processes. *Math. Struct. Comput. Sci.*, 6(3):251–259, 1996.

[210] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[211] L. Holík, R. Meyer, T. Vojnar, and S. Wolff. Effect summaries for thread-modular analysis - sound analysis despite an unsound heuristic. In *SAS*, volume 10422 of *LNCS*, pages 169–191. Springer, 2017.

[212] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[213] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[214] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25(1):116–133, 1978.

[215] IBM. CPLEX. `https://www.ibm.com/analytics/cplex-optimizer`. Accessed: 2020-10-30.

[216] IBM. *Power ISA - Version 2.07*. IBM Corp., 2013.

[217] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001.

[218] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

[219] F. Jaeger, D. L. Vertigan, and D. J. A. Welsh. On the computational complexity of the Jones and Tutte polynomials. *Mathematical Proceedings of the Cambridge Philosophical Society*, 108(1):35–53, 1990.

[220] P. Jancar. Strong bisimilarity on basic parallel processes is PSPACE-complete. In *(LICS*, page 218. IEEE, 2003.

[221] P. Jancar, M. Kot, and Z. Sawa. Complexity of deciding bisimilarity between normed BPA and normed BPP. *Inf. Comput.*, 208(10):1193–1205, 2010.

[222] P. Jancar, A Kucera, and F. Moller. Deciding bisimilarity between BPA and BPP processes. In *CONCUR*, volume 2761 of *LNCS*, pages 157–171. Springer, 2003.

[223] J. Jeong, S. H. Sæther, and J. A. Telle. Maximum matching width: New characterizations and a fast algorithm for dominating set. In *IPEC*, volume 43 of *LIPIcs*, pages 212–223. Schloss Dagstuhl, 2015.

[224] S. Joshi and B. König. Applying the graph minor theorem to the verification of graph transformation systems. In *CAV*, volume 5123 of *LNCS*, pages 214–226. Springer, 2008.

[225] H. W. Lenstra Jr. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[226] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962.

[227] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, volume 6174 of *LNCS*, pages 645–659. Springer, 2010.

[228] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley, 2005.

[229] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, 1972.

[230] R. M. Karp and R. J. Lipton. Some connections between nonuniform and uniform complexity classes. In *STOC*, pages 302–309. ACM, 1980.

[231] N. Kidd, S. Jagannathan, and J. Vitek. One stack to run them all - reducing concurrent analysis to sequential analysis under priority scheduling. In *SPIN*, volume 6349 of *LNCS*, pages 245–261. Springer, 2010.

[232] E. Kindler. Safety and liveness properties: A survey. *Bull. EATCS*, 53:268–272, 1994.

[233] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[234] G. Kirchhoff. Ueber die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Vertheilung galvanischer Ströme geführt wird. *Annalen der Physik*, 148(12):497–508, 1847.

[235] D. E. Knuth. On the translation of languages from left to right. *Inf. Control.*, 8(6):607–639, 1965.

[236] B. König and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *TACAS*, volume 3920 of *LNCS*, pages 197–211. Springer, 2006.

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

[238] B. Korel. A dynamic approach of test data generation. In *ICSM*, pages 311–317. IEEE, 1990.

[239] S. R. Kosaraju and G. F. Sullivan. Detecting cycles in dynamic graphs in polynomial time. In *STOC*, pages 398–406. ACM, 1988.

[240] I. Koutis. Faster algebraic algorithms for path and packing problems. In *ICALP*, volume 5125 of *LNCS*, pages 575–586. Springer, 2008.

[241] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266. IEEE, 1977.

[242] S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE, 2007.

[243] S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS*, volume 4963 of *LNCS*, pages 299–314. Springer, 2008.

[244] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, volume 5643 of *LNCS*, pages 477–492. Springer, 2009.

[245] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.

[246] S. La Torre, A. Muscholl, and I. Walukiewicz. Safety of parametrized asynchronous shared-memory systems is almost always decidable. In *CONCUR*, volume 42 of *LIPIcs*, pages 72–84. Schloss Dagstuhl, 2015.

[247] S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, volume 6901 of *LNCS*, pages 203–218. Springer, 2011.

[248] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, volume 5123 of *LNCS*, pages 37–51. Springer, 2008.

[249] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[250] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[251] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[252] L. Lee. Fast context-free grammar parsing requires fast Boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.

[253] J. Leroux. The reachability problem for petri nets is not primitive recursive. *CoRR*, abs/2104.12695, 2021.

[254] N.G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[255] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Logics of Programs*, volume 193 of *LNCS*, pages 196–218. Springer, 1985.

[256] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[257] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, 1992.

[258] D. Lokshtanov. *New methods in parameterized algorithms and complexity*. PhD thesis, University of Bergen, 2009.

[259] D. Lokshtanov, D. Marx, and S. Saurabh. Slightly superexponential parameterized problems. In *SODA*, pages 760–776. SIAM, 2011.

[260] D. Lokshtanov, N. Misra, and S. Saurabh. Kernelization - preprocessing with a guarantee. In *The Multivariate Algorithmic Revolution and Beyond*, volume 7370 of *LNCS*, pages 129–161. Springer, 2012.

[261] D. Lokshtanov and J. Nederlof. Saving space by algebraization. In *STOC*, pages 321–330. ACM, 2010.

[262] L. Lovasz. Coverings and coloring of hypergraphs. In *Southeastern Conference on Combinatorics, Graph Theory, and Computing*, Utilitas Math, pages 3–12, 1973.

[263] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339. ACM, 2008.

[264] P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, pages 283–294. ACM, 2011.

[265] M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *J. Algorithms*, 31(2):335–354, 1999.

[266] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, pages 378–391. ACM, 2005.

[267] D. Marx. Can you beat treewidth? In *FOCS*, pages 169–179. IEEE, 2007.

[268] D. Marx and M. Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism (but were afraid to ask). In *STACS*, volume 25 of *LIPIcs*, pages 542–553. Schloss Dagstuhl, 2014.

[269] A. A. Mathiasen and A. Pavlogiannis. The fine-grained and parallel complexity of Andersen's pointer analysis. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[270] U. Mathur, A. Pavlogiannis, and M. Viswanathan. The complexity of dynamic data race prediction. In *LICS*, pages 713–727. ACM, 2020.

[271] U. Mathur, A. Pavlogiannis, and M. Viswanathan. Optimal prediction of synchronization-preserving races. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[272] J. Matoušek and R. Thomas. On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Math.*, 108(1–3):343–364, 1992.

[273] D. W. Matula. Subtree isomorphism in $O(n^{5/2})$. In *Algorithmic Aspects of Combinatorics*, volume 2 of *Annals of Discrete Mathematics*, pages 91–106. Elsevier, 1978.

[274] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[275] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, volume 2 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984.

[276] R. Meyer. On boundedness in depth in the π-calculus. In *IFIP TCS*, volume 273 of *IFIP*, pages 477–489. Springer, 2008.

[277] R. Meyer and T. Strazny. Petruchio: From dynamic networks to nets. In *CAV*, volume 6174 of *LNCS*, pages 175–179. Springer, 2010.

[278] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

[279] B. Monien. How to find long paths efficiently. In *Analysis and Design of Algorithms for Combinatorial Problems*, volume 109 of *North-Holland Mathematics Studies*, pages 239–254. North-Holland, 1985.

[280] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455. ACM, 2007.

[281] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[282] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley, 3rd edition, 2011.

[283] J. Nederlof. Fast polynomial-space algorithms using Möbius inversion: Improving on Steiner tree and related problems. In *ICALPs*, volume 5555 of *LNCS*, pages 713–725. Springer, 2009.

[284] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[285] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[286] R. Niedermeier and P. Rossmanith. New upper bounds for MaxSat. In *ICALP*, volume 1644 of *LNCS*, pages 575–584. Springer, 1999.

[287] D. Pager. A practical general method for constructing lr(k) parsers. *Acta Informatica*, 7:249–268, 1977.

[288] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[289] D. A. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV*, volume 818 of *LNCS*, pages 377–390. Springer, 1994.

[290] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[291] A. Pnueli and Y. Sa'ar. All you need is compassion. In *VMCAI*, volume 4905 of *LNCS*, pages 233–247. Springer, 2008.

[292] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[293] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24. ACM, 2004.

[294] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[295] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[296] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[297] W. Rudin. *Functional Analysis*. McGraw-Hill, second edition, 1991.

[298] M. Saksena, O. Wibling, and B. Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. In *TACAS*, volume 4963 of *LNCS*, pages 18–32. Springer, 2008.

[299] P. Scheffler. A practical linear time algorithm for disjoint paths in graphs with bounded tree-width. Preprint 396, Technische Universität Berlin, Institut für Mathematik, 1994.

[300] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971.

[301] P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[302] A. Singh, C. R. Ramakrishnan, and S. A. Smolka. Query-based model checking of ad hoc network protocols. In *CONCUR*, volume 5710 of *LNCS*, pages 603–619. Springer, 2009.

[303] M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[304] Y. Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 1st edition, 2015.

[305] SPARC. *The SPARC Architecture Manual - Version 8*. SPARC International Inc., 1992.

[306] SPARC. *The SPARC Architecture Manual - Version 9*. SPARC International Inc., 1994.

[307] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

[308] L. Stockmeyer. Planar 3-colorability is polynomial complete. *SIGACT News*, 5(3):19–25, 1973.

[309] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[310] D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded Java programs. In *SPIN*, volume 5156 of *LNCS*, pages 270–287. Springer, 2008.

[311] I. Suzuki. Proving properties of a ring of finite-state machines. *Inf. Process. Lett.*, 28(4):213–214, 1988.

[312] J. Swernofsky and M. Wehar. On the complexity of intersecting regular, context-free, and tree languages. In *ICALP*, volume 9135 of *LNCS*, pages 414–426. Springer, 2015.

[313] R. E. Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1(2):146–160, 1972.

[314] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183. ACM, 1995.

[315] D. M. Thilikos, M. J. Serna, and H. L. Bodlaender. Cutwidth I: A linear time fixed parameter algorithm. *Journal of Algorithms*, 56(1):1–24, 2005.

[316] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[317] L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[318] A. Valmari. A stubborn attack on state explosion. In *CAV*, volume 531 of *LNCS*, pages 156–165. Springer, 1990.

[319] J. M. M. van Rooij, H. L. Bodlaender, and P. Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *ESA*, volume 5757 of *LNCS*, pages 566–577. Springer, 2009.

[320] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 322–331. IEEE, 1986.

[321] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA*, pages 97–107. ACM, 2004.

[322] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inf. Theory*, 13(2):260–269, 1967.

[323] T. Wareham. The parameterized complexity of intersection and composition operations on sets of finite-state automata. In *CIAA*, volume 2088 of *LNCS*, pages 302–310. Springer, 2000.

[324] I. Wegener. *Complexity theory - exploring the limits of efficient algorithms*. Springer, 2005.

[325] M. Wehar. Hardness results for intersection non-emptiness. In *ICALP*, volume 8573 of *LNCS*, pages 354–362. Springer, 2014.

[326] H. Wei, Y. Huang, J. Cao, X. Ma, and J. Lu. Verifying PRAM consistency over read/write traces of data replicas. *CoRR*, abs/1302.5161, 2013.

[327] T. Wies, D. Zuffrey, and T. A. Henzinger. Forward analysis of depth-bounded processes. In *FoSSaCS*, volume 6014 of *LNCS*, pages 94–108. Springer, 2010.

[328] R. Williams. Finding paths of length k in $O^*(2^k)$ time. *Inf. Process. Lett.*, 109(6):315–318, 2009.

[329] V. Vassilevska Williams and R. R. Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5):27:1–27:38, 2018.

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

[331] S. Wolff. *Verifying Non-blocking Data Structures with Manual Memory Management*. PhD thesis, Technische Universität Braunschweig, Aug 2021.

[332] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL*, pages 184–193. ACM, 1986.

[333] C. K. Yap. Some consequences of non-uniform conditions on uniform classes. *Theor. Comput. Sci.*, 26:287–300, 1983.

[334] F. Yates. The design and analysis of factorial experiments. In *Technical Communication No. 35.* Imperial Bureau of Soil Science, 1937.

[335] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189 – 208, 1967.

[336] R. Zennou, A. Bouajjani, C. Enea, and M. Erradi. Gradual consistency checking. In *CAV*, volume 11562 of *LNCS*, pages 267–285. Springer, 2019.

[337] D. Zufferey. *Analysis of Dynamic Message Passing Programs (a framework for the analysis of depth-bounded systems)*. PhD thesis, Institute of Science and Technology, 2013.

Appendix

A. Additional Material for Parameterized and Fine-Grained Complexity

We provide additional material and proofs for the chapters 3, 4, and 5.

A.1 Additional Material and Proofs for Chapter 3

A.1.1 Proof of Lemma 3.9

Proof. We prove the equation by showing two inequalities. For the first one, let a cover $C \subseteq \{F_1, \ldots, F_j\}$ of X be given. There are two cases: (1) $F_j \notin C$ or (2) $F_j \in C$. In Case (1), we have that C is contained in $\{F_1, \ldots, F_{j-1}\}$ and hence $|C| \geq T[X, j-1]$. For Case (2), consider that $X \setminus F_j$ is covered by $C \setminus \{F_j\} \subseteq \{F_1, \ldots, F_{j-1}\}$. We obtain that $|C| \geq 1 + T[X \setminus F_j, j-1]$. Putting both cases together we can derive the inequality

$$T[X, j] \geq \min(T[X, j-1], 1 + T[X \setminus F_j, j-1]).$$

For the other direction, let a cover $C \subseteq \{F_1, \ldots, F_{j-1}\}$ of X be given. Then, C also lies in $\{F_1, \ldots, F_j\}$ and appears in the definition of $T[X, j]$. Hence, $|C| \geq T[X, j]$. If a cover $C \subseteq \{F_1, \ldots, F_{j-1}\}$ of $X \setminus F_j$ is given, we can complete it to a cover $C \cup \{F_j\}$ of X within $\{F_1, \ldots, F_j\}$. Consequently, $|C| + 1 \geq T[X, j]$. Combining both observations, we obtain a second inequality:

$$T[X, j] \leq \min(T[X, j-1], 1 + T[X \setminus F_j, j-1]).$$

The inequalities show that the recurrence, as it is stated above, is indeed correct. This completes the proof of the lemma. □

A.1.2 Proof of Theorem 3.20

Proof. We begin with the ranked zeta transform. We interpret the function f over vectors. Then, the ranked zeta transform $\hat{\zeta} f$ is given by

$$(\hat{\zeta} f)(k, x_1, \ldots, x_n) = \sum_{\substack{y_1, \ldots, y_n \in \{0,1\} \\ y_1 + \cdots + y_n = k}} [y_1 \leq x_1 \wedge \ldots y_n \leq x_n] \cdot f(y_1, \ldots, y_n).$$

Note that the sum of the y_i needs to be equal to k. This ensures that the current set in the summation is actually of correct size k.

To compute $\hat{\zeta} f$, we define a table $\hat{\zeta}_k[(x_1, \ldots, x_n), j]$ with an entry for each vector $(x_1, \ldots, x_n) \in \{0,1\}^n$ and integers $k, j \in [0..n]$. For $j \geq 1$, an entry $\hat{\zeta}_k[(x_1, \ldots, x_n), j]$ stores the following sum:

$$\sum_{\substack{y_1,\ldots,y_j \in \{0,1\} \\ y_1+\cdots+y_j=k}} [y_1 \leq x_1 \wedge \ldots y_j \leq x_j] \cdot f(y_1, \ldots, y_j, x_{j+1}, \ldots, x_n).$$

For $j = 0$, we define $\hat{\zeta}_k[(x_1, \ldots, x_n), 0] = f(x_1, \ldots, x_n)$. Note that the definition immediately implies that $\hat{\zeta}_k[(x_1, \ldots, x_n), n] = \hat{\zeta}(k, x_1, \ldots, x_n)$.

To fill the table, we employ a dynamic programming along the following recurrence, quite similar to the fast zeta transform.

$$\hat{\zeta}_k[(x_1, \ldots, x_n), j] = \begin{cases} \hat{\zeta}_k[(x_1, \ldots, x_n), j-1] & \text{if } x_j = 0, \\ \hat{\zeta}_k[(x_1, \ldots, x_{j-1}, 0, x_{j+1}, \ldots, x_n), j-1] \\ \quad + \hat{\zeta}_{k-1}[(x_1, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_n), j-1] & \text{if } x_j = 1. \end{cases}$$

As for the fast zeta transform, the first case $x_j = 0$ is immediate and the second case $x_j = 1$ requires a distinction of $y_j = 0$ and $y_j = 1$. Note that for $y_j = 1$, we must take the sum $y_1 + \cdots + y_j = k$ into account. In fact, when removing y_j from the summation, we obtain $y_1 + \cdots + y_{j-1} = k - 1$. This explains the occurrence of $\hat{\zeta}_{k-1}[(x_1, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_n), j-1]$ in the recurrence.

We apply the recurrence for increasing $k = 0, 1, \ldots, n$. Hence, we can compute the ranked zeta transform in $O(2^n \cdot n^2)$ arithmetic operations. For computing the ranked Möbius transform, one can reuse the recurrence of the fast Möbius transform and derive an algorithm similar to the fast ranked zeta transform. It needs $O(2^n \cdot n^2)$ arithmetic operations. □

A.1.3 Further Applications of Subset Convolution in Counting

We show further applications of fast subset convolution. In the following, we employ the technique to determine the *chromatic number*, the *domatic number*, and to count the number of *spanning forests* of a graph.

Chromatic Number The *chromatic number* of a graph is the minimum number of colors needed to color the graph. For the example given in Figure 3.4, this number is 3. We refer to the problem that, given a graph $G = (V, E)$, compute its chromatic number, as Chromatic [112]. With being able to compute the number of k-colorings, according to Theorem 3.24, solving Chromatic is simple. Let $\mathcal{F}$ again denote the family of independent sets within V and $\chi_{\mathcal{F}}$ its characteristic function. We compute $*^k \chi_{\mathcal{F}}$ for increasing k, starting from 1. Since $(*^k \chi_{\mathcal{F}})(V) > 0$ if and only if there is a k-coloring of G, we stop as soon

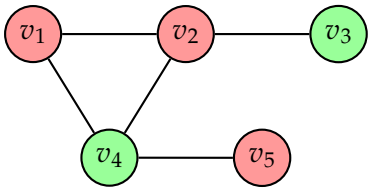


Figure A.1: The red marked vertices $A_1 = \{v_1, v_2, v_5\}$ and the green marked vertices $A_2 = \{v_3, v_4\}$ show two dominating sets of the underlying graph. The pair (A_1, A_2) is a domatic partition. Note that there is no domatic partition with three or more sets. The domatic number of the graph is 2.

as we arrive at the minimal k with $(*^k \chi_{\mathcal{F}})(V) > 0$. Then, k is the chromatic number. The algorithm terminates as the chromatic number of a graph is at most its number of vertices. Altogether, it takes $O^*(2^n)$ time.

Domatic Number Theorem 3.23 can be used to determine the *domatic number* of a graph. To clarify the notion, we start by defining *dominating sets*. Let $G = (V, E)$ be a graph. A subset $X \subseteq V$ is called *dominating set* if each vertex in $V \setminus X$ is connected to a vertex in X. Examples are given in Figure A.1.

Deciding the existence of a dominating set of a certain size in a given graph is a well-studied NP-complete decision problem, called Dominating Set [112]. The problem will be of interest later. We are interested in partitions into dominating sets. Let $k \in \mathbb{N}$ be an integer. A *k-domatic partition* of G is a k-partition $(A_1, \ldots, A_k)$ of V, where each set A_i is a dominating set of G. The *domatic number* of G is the maximal size of such a partition. Formally, the maximal $k \in \mathbb{N}$ such that there exists a k-domatic partition of G.

To compute the domatic number, we can employ Theorem 3.23 with $\mathcal{F}$ being the family of dominating sets. Computing $\mathcal{F}$ and its characteristic function $\chi_{\mathcal{F}}$ can be established in time $O^*(2^n)$. The value $(*^k \chi_{\mathcal{F}})(V)$ is non-negative if and only if there is a k-domatic partition of G. Hence, we compute $*^k \chi_{\mathcal{F}}$ iteratively for increasing k and we stop as soon as we reach a k such that $(*^{k+1} \chi_{\mathcal{F}})(V) = 0$. Then, k is the domatic number. Note that the algorithm terminates since the domatic number is bounded by n and it takes time $O^*(2^n)$.

Counting Spanning Forests We can count other types of partitions by varying the family $\mathcal{F}$ and its characteristic function $\chi_{\mathcal{F}}$ in Theorem 3.23. As long as we guarantee that $\chi_{\mathcal{F}}$ can be computed in time $O^*(2^n)$, like we did for independent sets and dominating sets, the theorem yields an $O^*(2^n)$-time algorithm for counting partitions into the corresponding sets.

But fast subset convolution can also handle more general functions. For instance, it can be used to to count the number of *spanning forests* [170], a problem that is known for its #P-completeness [219]. A *spanning forest* of

an undirected graph $G = (V, E)$ is an acyclic subgraph of G that spans all vertices in V. If the subgraph is connected, it is a *spanning tree*. Hence, a spanning forest consists of various spanning trees that are not connected to each other. But this means that a spanning forest actually induces a partition of the vertices into sets that can be spanned by trees.

The latter observation lets us determine the number of spanning forests via subset convolution. To this end, define $s : \mathcal{P}(V) \to \mathbb{Z}$ to be the function mapping a subset $X \subseteq V$ to the number of spanning trees in the induced subgraph $G[X]$. Moreover, let $k \in \mathbb{N}$ be an integer. Then, the number of spanning forests that consist of k spanning trees is given by

$$(*^k s)(V) = \sum_{\substack{A_1 \cup \ldots \cup A_k = V \\ A_i \cap A_j = \emptyset, i \neq j}} s(A_1) \cdots s(A_k).$$

Assuming we are given the map s, we can compute $*^k s$ for each $k \in [1..n]$ in time $O^*(2^n)$. Hence, we can determine the total number $\sum_{k=1}^{n} (*^k s)(V)$ of spanning forests of G within the same time complexity. To get the function s, we apply Kirchhoff's matrix tree theorem [234]. It shows that the number of spanning trees in a graph can be computed in polynomial time. By computing the number for each subset X of V, we can obtain s in time $O^*(2^n)$.

A.1.4 Proof of Lemma 3.28

Proof. In order to show the correctness of the representation, we prove that $\zeta(f *_c g) = (\zeta f) \cdot (\zeta g)$. Then, the result follows from the fact that the Möbius transform μ is the inverse operator of the zeta transform ζ, see Lemma 3.15. We begin by expanding the left-hand side. To this end, let $X \subseteq U$.

$$\begin{aligned} \zeta(f *_c g)(X) &= \sum_{Y \subseteq X} (f *_c g)(Y) \\ &= \sum_{Y \subseteq X} \sum_{A \cup B = Y} f(A) \cdot g(B) \\ &= \sum_{A \cup B \subseteq X} f(A) \cdot g(B). \end{aligned}$$

The latter reformulation is correct since it does not make a difference whether to sum over all covers of subsets of X or to sum over the subsets in the cover directly. But if we do so, the sum is actually a product of two sums.

$$\begin{aligned} \sum_{A \cup B \subseteq X} f(A) \cdot g(B) &= \Big(\sum_{A \subseteq X} f(A) \Big) \cdot \Big(\sum_{B \subseteq X} g(B) \Big) \\ &= \big((\zeta f)(X) \big) \cdot \big((\zeta g)(X) \big) \\ &= \big((\zeta f) \cdot (\zeta g) \big)(X). \end{aligned}$$

We can derive that $\zeta(f *_c g) = (\zeta f) \cdot (\zeta g)$ which completes the proof. □

A.2 Additional Material and Proofs for Chapter 4

A.2.1 Proof of Theorem 4.6

Proof. Let $(\mathcal{F}, r)$ be an instance of Set Cover with universe U and family of sets $\mathcal{F} = \{S_1, \ldots, S_m\}$. We construct an instance (G, k) of Dominating Set as follows. First, we set the parameter $k = r$. Then we construct the graph $G = (V, E)$. It contains vertices for each element in U and for each set in $\mathcal{F}$. Formally, $V = U \cup S$ where $S = \{s_1, \ldots, s_m\}$ contains m fresh vertices. There is an edge from a vertex $u \in U$ to a vertex s_i in S if and only if $u \in S_i$. Moreover, the vertices in S form a clique. Formally,

$$E = \{\{u, s_i\} \mid u \in S_i\} \cup \{\{s_i, s_j\} \mid i \neq j\}.$$

The edges of the form $\{u, s_i\}$ represent the cover problem. The clique is needed to remove additional domination conditions from the vertices in S. The only condition should be to dominate vertices of U and not other vertices in S. An illustration of the graph can be found in Figure A.2.

Since G can be constructed in polynomial time, it is left to prove the correctness of the reduction. To this end, let $(\mathcal{F}, r)$ be a yes-instance of Set Cover. Then, there is a cover of U consisting of at most r sets in $\mathcal{F}$. Phrased differently, there is a set of indices I with $|I| \leq r$ and $U = \bigcup_{i \in I} S_i$. Consider the set of vertices $X = \{s_i \mid i \in I\} \subseteq S$. We have $|X| \leq r$ and by construction of G, the set X is dominating. In fact, each vertex $s \in S$ either belongs to X or is adjacent to a vertex of X since $G[S]$ is a clique. A vertex $u \in U$ belongs to a set S_i with $i \in I$ since we assumed to have a cover. Hence, by construction there is an edge $\{u, s_i\} \in E$ with vertex $s_i \in X$.

Now let (G, k) with $k = r$ be a yes-instance of Dominating Set. Then, there is a dominating set X of size at most r in G. The set X may contain vertices from U and S. We show that there is a dominating set of equal or smaller size containing only vertices of S. To this end, we assume that each element $u \in U$ belongs to some set $S_i \in \mathcal{F}$. If this is not the case, we clearly started with a no-instance of Set Cover(r) and the reduction is redundant. Let X contain a vertex $u \in U$. By our assumption, there is a vertex $s_i \in S$ such that $\{u, s_i\} \in E$. Then, the set $X \setminus \{u\} \cup \{s_i\}$ is still a dominating set of G and of same or smaller size than X. Iterating this process yields a dominating set Y of size at most r with $Y \cap U = \emptyset$. From Y, we can construct a cover of U.

Consider the set of indices $I = \{i \mid s_i \in Y\}$. Then, the sets $S_i \in \mathcal{F}$ with $i \in I$ cover the universe U. In fact, let $u \in U$ be an element. Since Y is a dominating set of G, there is a vertex $s_i \in Y$ such that $\{u, s_i\} \in E$. By construction of G this means that $u \in S_i$ and $i \in I$. We obtain $U = \bigcup_{i \in I} S_i$. □

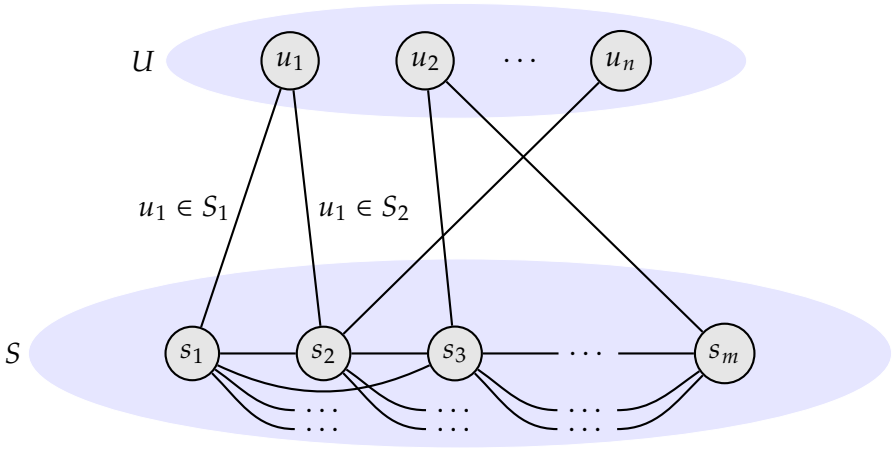


Figure A.2: The constructed graph $G = (V, E)$ with vertices $V = U \cup S$, where $U = \{u_1, \ldots, u_n\}$ and $S = \{s_1, \ldots, s_m\}$. The lower part of the graph, $G[S]$, forms a clique. There are no edges between the vertices of U. An edge $\{u_i, s_j\}$ is present in the graph if and only if u_i belongs to the set S_j.

A.2.2 Proof of Lemma 4.11

Proof. To prove the lemma, we first define a family of circuits that replaces the 2 in the definition of weft by another constant. Let $c \in \mathbb{N}$ be a constant with $c \geq 2$. Moreover, let $d \geq 1$. The family $\mathcal{C}(t, c, d)$ consists of all circuits that are of depth at most d and that have at most t gates with fan-in larger than c on each input-output path. Note that $\mathcal{C}(t, d) = \mathcal{C}(t, 2, d)$.

Our next step is to give a parameterized reduction from WCS$[\mathcal{C}(t, c, d)]$, the problem WCS restricted to circuits of the family $\mathcal{C}(t, c, d)$, to the problem WCS$[\mathcal{C}(t, d \cdot c)]$. Let (C, k) be an instance of the former, with $C \in \mathcal{C}(t, c, d)$. We construct a circuit C' as follows: each gate in C with fan-in at most c is transformed into a chain of at most $c - 1$ gates, each with fan-in 2. For each such transformation, we need to add at most c to the depth. Since the depth of C was bounded by d, the depth of the resulting circuit C' is at most $d \cdot c$. The construction is correct: C has a satisfying assignment of weight k if and only if C' has. Hence, we have a parameterized reduction.

Now we interchange the class $\mathcal{C}(t, d)$ of circuits in the definition of $\mathsf{W}[t]$ by the new family $\mathcal{C}(t, c, d)$. Let $\mathsf{W}'[t]$ be the class of problems that are reducible to WCS$[\mathcal{C}(t, c, d)]$ for some $d \in \mathbb{N}$. By the above reduction, we obtain the inclusion $\mathsf{W}'[t] \subseteq \mathsf{W}[t]$. Hence, it is left to show the other inclusion as well. To this end, note that each circuit of weft bounded by t has at most t gates with fan-in larger than 2 on each input-output path. Hence, it also has at most t gates of fan-in larger than c on each input-output path. Phrased differently, we have $\mathcal{C}(t, d) \subseteq \mathcal{C}(t, c, d)$. This immediately yields a parameterized reduction from WCS$[\mathcal{C}(t, d)]$ to WCS$[\mathcal{C}(t, c, d)]$, namely the identity. Hence, we obtain

the other inclusion as well: $\mathsf{W}[t] \subseteq \mathsf{W}'[t]$. □

A.2.3 Proof of Theorem 4.19

Proof. Let $(\mathcal{G}, r)$ be an instance of Hitting Set and let $\mathcal{G}$ be over the universe U. We construct an instance $(\mathcal{F}, r)$ of Set Cover. The new universe that we consider is $V = \mathcal{G}$, the family of sets of the Hitting Set-instance. The family $\mathcal{F} \subseteq \mathcal{P}(V)$ consists of the subsets $F_u \subseteq V$, one for each $u \in U$. A set F_u contains all sets of V where element u is present:

$$F_u = \{G \in V \mid u \in G\}.$$

It is left to prove that there is a hitting set of size at most r intersecting each set of $\mathcal{G}$ if and only if there is a set cover of V consisting of at most r sets from $\mathcal{F}$. First, we may assume that there is a hitting set $H \subseteq U$ with $|H| \leq r$. Consider the subfamily $\mathcal{C} = \{F_u \in \mathcal{F} \mid u \in H\}$ of $\mathcal{F}$. It has size at most r. Moreover, it forms a set cover of V. To see the latter, let $G \in V$. Since H is a hitting set, there exists an element $u \in U$ that lies in the intersection $H \cap G$. Hence, $G \in F_u$ and F_u is a set from $\mathcal{C}$. We obtain $V = \bigcup_{F_u \in \mathcal{C}} F_u$.

For the other direction, let a set cover of V, a subfamily $\mathcal{C} \subseteq \mathcal{F}$ of size at most r with $V = \bigcup_{F_u \in \mathcal{C}} F_u$ be given. We construct a hitting set H by setting $H = \{u \in U \mid F_u \in \mathcal{C}\}$. The set is of size at most r and intersects with each set from $\mathcal{G}$: let $G \in \mathcal{G} = V$, then there is a set $F_u \in \mathcal{C}$ so that $G \in F_u$. But this means that $u \in H$ and $u \in G$. Hence, the intersection $H \cap G$ is non-empty. □

A.3 Additional Material and Proofs for Chapter 5

A.3.1 Proof of Lemma 5.2

Proof. Assume the contrary, namely that there is an algorithm $\mathcal{A}$ solving the problem 3-SAT in time $2^{o(n)}$. Our goal is to derive a contradiction. Algorithm $\mathcal{A}$ runs in time $O^*(2^{n/g(n)})$ where $g(n)$ is an unbounded and monotonously increasing function. Since $\delta_3 > 0$ by ETH, we can conclude that there is an integer $n_0 \in \mathbb{N}$ such that for each $n \geq n_0$, we have

$$g(n) > \frac{2}{\delta_3}.$$

The inequality implies that the running time of algorithm $\mathcal{A}$ can be asymptotically bounded. Therefore, $\mathcal{A}$ runs in time

$$O^*(2^{n/g(n)}) = O^*(2^{\frac{1}{2}\cdot\delta_3\cdot n}).$$

By the definition of δ_3, we get that $0 \leq \delta_3 \leq \frac{1}{2} \cdot \delta_3$ and hence $\delta_3 = 0$. This contradicts the exponential time hypothesis and completes the proof. □

A.3.2 Proof of Theorem 5.10

Proof. We follow a linear reduction from 3-SAT to Directed Ham. Cycle and obtain the lower bound for the latter from Lemma 5.6. To this end, let φ be a formula in 3-CNF with n variables and m clauses. We construct a directed graph G with at most $7 \cdot m$ many vertices such that G has a Hamiltonian cycle if and only if the formula φ is satisfiable.

For each variable x in φ, we construct a variable gadget, a subgraph simulating the evaluation of x. It contains $2 \cdot \ell(x)$ many vertices, where $\ell(x)$ is the number of times that x appears as a literal in φ. Let these vertices be called $v_1, v_2, \ldots, v_{2 \cdot \ell(x)}$. We chain them by adding the directed edges (v_i, v_{i+1}) and (v_{i+1}, v_i). An illustration is given in Figure A.3. Evaluating x is then simulated by traveling through the gadget: going from left to right corresponds to setting x to 0. Going from right to left amounts to setting x to 1.

Clauses are represented as follows. For a clause C, we add a single new vertex to the graph that we also refer to as C. Assume C contains a variable x and let C be the i-th clause that holds x or $\neg x$ as a literal, according to some enumeration. Then we tie the variable gadget of x with the vertex C, depending on the precise literal. If x appears as literal in C, we add the edges $(v_{2 \cdot i+1}, C)$ and $(C, v_{2 \cdot i})$. Otherwise, $\neg x$ is a literal in C and we add the two edges $(v_{2 \cdot i}, C)$ and $(C, v_{2 \cdot i+1})$. Assume we are in the latter case and we take a tour from left to right in the variable gadget of x, we evaluate x to 0. Then, we can visit the vertex C and return to the variable gadget. If we go from right to left, we evaluate x to 1, we cannot visit C and return. Hence, visiting C and thus satisfying the clause is left to another variable.

Finally, we add some edges that let us choose an assignment of the variables. Let $x_1, \ldots, x_n$ be an enumeration of the variables and $v_1, \ldots, v_{2 \cdot \ell(x_i)}$ the vertices of the variable gadget of x_i and similarly $u_1, \ldots, u_{2 \cdot \ell(x_{i+1})}$ the vertices of x_{i+1}'s gadget. We add the following four edges:

$$(v_1, u_1),\ (v_1, u_{2 \cdot \ell(x_{i+1})}),\ (v_{2 \cdot \ell(x_i)}, u_1),\ (v_{2 \cdot \ell(x_i)}, u_{2 \cdot \ell(x_{i+1})}).$$

We further add these four edges between the variable gadgets of x_n and x_1.

Intuitively, the edge $(v_{2 \cdot \ell(x_i)}, u_1)$ means that from the right end of the variable gadget of x_i, so after evaluating x_i to 0, we go to the left end of the variable gadget of x_{i+1}, so we evaluate x_{i+1} to 0 as well.

The constructed graph G has one vertex for each clause and $2 \cdot \ell(x)$ vertices for a variable x. Since φ contains at most $3 \cdot m$ many literals, we can bound the number of vertices of G by the following inequality:

$$m + \sum_{i=1}^{n} 2 \cdot \ell(x_i) \leq m + 2 \cdot 3 \cdot m = 7 \cdot m.$$

Hence, it is left to argue on the correctness. Like mentioned above, an assignment can be simulated by G by traveling through the variable gadgets

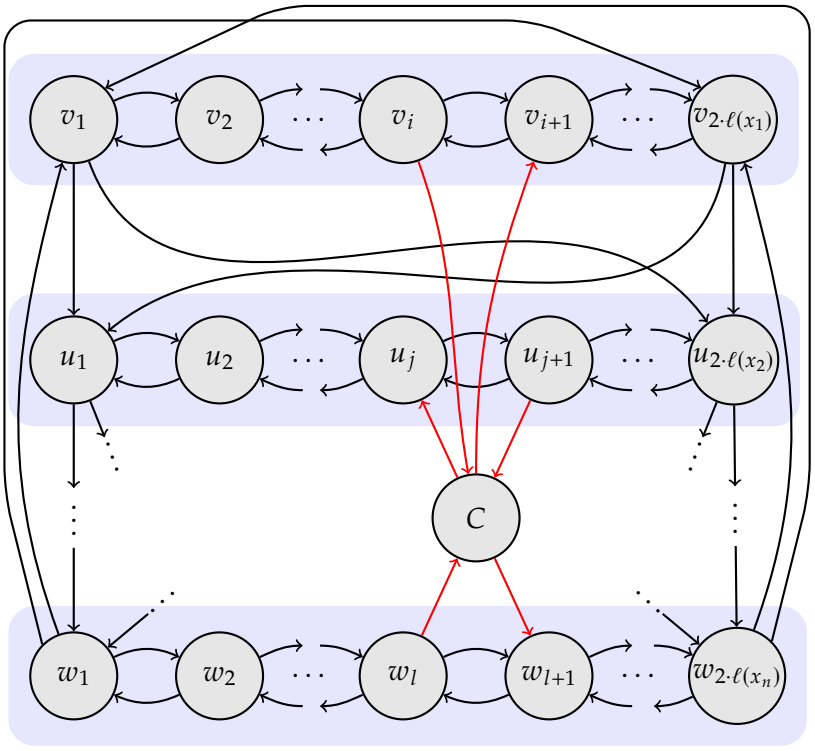


Figure A.3: Graph G constructed from φ on variables $x_1, \ldots, x_n$. Variable gadgets are marked blue. We show the gadget of x_1 containing the vertices v_i, the gadget of x_2 containg the vertices u_i, and the gadget of x_n containing the w_i. Going from left to right through a gadget means evaluating the corresponding variable to 0. Similarly, going from right to left means evaluating it to 1. Note the edges between the variable gadgets. They are used to choose the evaluation. The clause C in the figure is given by $\neg x_1 \vee x_2 \vee \neg x_n$. It is connected appropriately to the variable gadgets by the red marked edges.

in the appropriate direction. During this travel, the clause vertices can be visited and thus satisfied if the literal matches the chosen assignment. Hence, a Hamiltonian cycle in G corresponds to a satisfying assignment of φ. □

A.3.3 Proof of Theorem 5.12

Proof. We follow the classical reduction from 3-SAT. To this end, let φ be a formula in 3-CNF with n variables and m clauses. We construct a graph G such that G is 3-colorable if and only if φ is satisfiable.

First, we introduce three vertices: F, T, and N, that are arranged in a triangle. In a proper 3-coloring all three vertices need to have different colors. The color that is assigned to F models the evaluation *false*, the color of T

models *true*, and the color of N is the *base* color. To ease the notation, we refer to these colors by T, F, and N when we consider colorings of the graph.

To simulate variable assignments, we introduce a gadget for each variable x. It consists of two vertices, x and $\neg x$, that are connected to each other. Moreover, x and $\neg x$ are both adjacent to N. Hence, in a 3-coloring of the graph, x and $\neg x$ cannot be assigned the same color and one of them needs to be colored by T, the other one by F. This means that exactly one of the literals is evaluated to true and the other one to false.

Clauses are simulated by clause gadgets. For a clause $C = \ell_1 \vee \ell_2 \vee \ell_3$ with the three literals ℓ_i, we construct it as follows. There are two triangles that simulate the *or* gates in the clause. The first triangle has as *inputs* an edge from ℓ_1 and ℓ_2. The third node is the *output*, called out_1, of this *or* gate. The second triangle has as inputs an edge from ℓ_3 and from out_1 and an output vertex called out_2. Hence, the triangles are stacked and can be seen as appending one *or* gate to another. The vertex out_2 is moreover adjacent to F and N implying that in a 3-coloring out_2 has to be colored by T which represents the evaluation of C. An illustration is given in Figure A.4.

Concerning the correctness of the construction, note that a clause gadget can be colored by 3 colors in G if and only if one of its input literals is colored by T. The latter is true if and only if one of the literals is assigned true.

The graph has a total of $3 + 2 \cdot n + 6 \cdot m$ vertices. Hence, the reduction is indeed linear and the result follows by Lemma 5.6. □

A.3.4 Proof of Lemma 5.17

Proof. Assume there is an algorithm solving Q in time $f(t) \cdot n^{o(t)}$ where n is the size of the input and $f(t)$ is a computable function. We show that this contradicts the ETH. There is a linear parameterized reduction from Clique to Q. Let the reduction take an instance (G, k) of Clique and output an instance (I, t) of Q in time $g(k) \cdot |G|^d$ where $g(k)$ is a computable function and $d \in \mathbb{N}$ is a constant. Moreover, let $t = c \cdot k$ for a constant $c \in \mathbb{N}$. Note that the size of I is bounded by the running time of the reduction, we get the estimation $|I| \leq g(k) \cdot |G|^d$. By appending the algorithm for Q to the reduction, we obtain an algorithm for Clique that runs in time

$$\begin{aligned} f(t) \cdot |I|^{o(t)} + g(k) \cdot |G|^d &\leq f(c \cdot k) \cdot (g(k) \cdot |G|^d)^{o(k)} + g(k) \cdot |G|^d \\ &\leq f(c \cdot k) \cdot g(k)^{o(k)} \cdot |G|^{o(k)}. \end{aligned}$$

The function $f(c \cdot k) \cdot g(k)^{o(k)}$ is computable [12] since $f(t)$, the multiplication function $c \cdot k$, and $g(k)$ are. Hence, the described algorithm for Clique contradicts Theorem 5.15 and therefore the ETH. This completes the proof. □

[12] Recall that $o(k) = k/s(k)$ for a monotonously increasing and unbounded function $s(k)$.

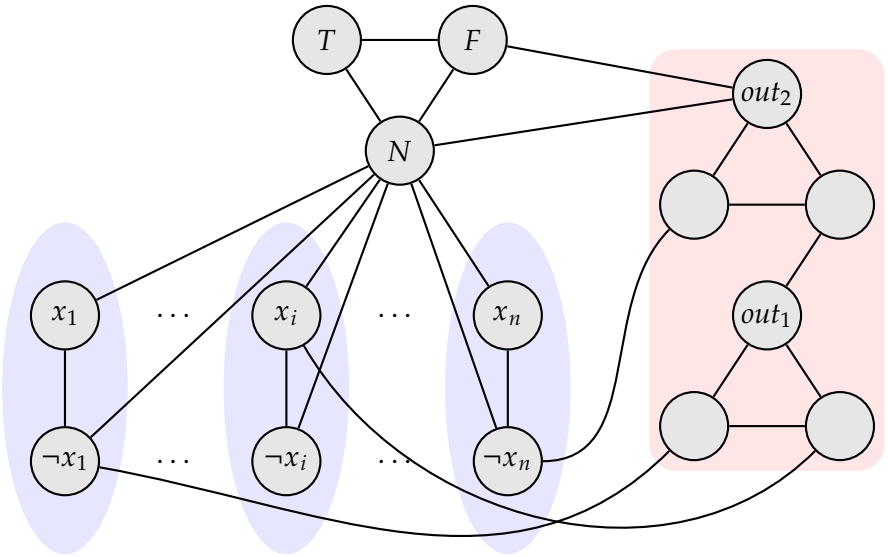


Figure A.4: The graph constructed from a formula φ with variables $x_1, \ldots, x_n$. Variable gadgets are marked blue. The red marked part of the graph is the gadget of the clause $C = \neg x_1 \vee x_i \vee \neg x_n$. It is connected to its literals. Its output out_2 represents the evaluation of C. The vertex is adjacent to F and N. In a 3-coloring, it has to be colored by T — C must evaluate to true.

A.3.5 Proof of Theorem 5.22

Proof. We show that, if the SETH holds then the ETH holds as well. To this end, assume the latter does not hold. Formally this means that $\delta_3 = 0$. Our goal is to show that this already implies $\delta_q = 0$ for each $q \in \mathbb{N}$ which contradicts the SETH. To start with, let $q \in \mathbb{N}$ be a constant with $q \geq 4$ and $\delta_q > 0$. Note that such a constant exists since the SETH holds true.

We construct an algorithm for q-SAT that breaks the SETH. Let a formula φ in q-CNF with n variables be given. First, we decompose φ by sparsification. To this end, set $\varepsilon = \frac{1}{4} \cdot \delta_q > 0$. By Theorem 5.3, there exists a constant $K \in \mathbb{N}$ and an algorithm $\mathcal{D}$ running in time $O^*(2^{\varepsilon \cdot n})$ that computes a decomposition

$$\varphi = \bigvee_{i=1}^{t} \varphi_i,$$

where $t \leq 2^{\varepsilon \cdot n}$ and each φ_i is a formula in q-CNF with at most $K \cdot n$ clauses and n variables. Note that φ is satisfiable if and only if one of the φ_i is.

Now we transform each φ_i to an instance ψ_i of 3-SAT by the following reduction. Each clause $C = \ell_1 \vee \ldots \vee \ell_q$ of φ_i is replaced by introducing $q - 3$

new variables $x_1, \ldots, x_{q-3}$ in ψ_i and the following $q-2$ clauses:

$$\begin{aligned} C_1 &= \ell_1 \vee \ell_2 \vee x_1, \\ C_2 &= \neg x_1 \vee \ell_3 \vee x_2, \\ &\ldots \\ C_{q-2} &= \neg x_{q-3} \vee \ell_{q-1} \vee \ell_q. \end{aligned}$$

Clearly, the formula ψ_i can be computed in polynomial time and φ_i is satisfiable if and only if ψ_i is. Since φ_i has at most $K \cdot n$ clauses, the total number of variables of ψ_i can be estimated as follows:

$$n + (q-3) \cdot K \cdot n \leq (1 + q \cdot K) \cdot n.$$

Now we check satisfiability of each formula ψ_i by a suitable 3-SAT-algorithm. We assumed that $\delta_3 = 0$. Since this is the infimum of the set

$$A = \{c \in \mathbb{R} \mid \text{There is an algorithm solving 3-SAT in time } O^*(2^{c \cdot n})\},$$

there is a sequence $(c_n)_{n \in \mathbb{N}}$ in A that converges to 0. Hence, there exists an index $n_0 \in \mathbb{N}$ such that the element $d = c_{n_0}$ from the sequence satisfies:

$$d \leq \frac{\delta_q}{4 \cdot (1 + q \cdot K)}.$$

Since d is in element of A, there is a corresponding algorithm $\mathcal{A}$ solving 3-SAT in time $O^*(2^{d \cdot n})$. We apply $\mathcal{A}$ to each formula ψ_i. Since ψ_i has at most $(1 + q \cdot K) \cdot n$ many variables, this step takes time

$$O^*(2^{d \cdot (1+q \cdot K) \cdot n}) = O^*(2^{\frac{1}{4} \cdot \delta_q \cdot n}).$$

As soon as one of the ψ_i is found to be satisfiable, the original formula φ is satisfiable as well and we can report a yes-instance. Otherwise, if none of the ψ_i has a satisfying assignment, φ is a no-instance. Hence, the described algorithm solves the problem q-SAT. Summing up, it runs in time

$$O^*(\underbrace{2^{\frac{1}{4} \cdot \delta_4 \cdot n}}_{\mathcal{A} \text{ on } \psi_i} \cdot \underbrace{2^{\varepsilon \cdot n}}_{\text{nr. of } \psi_i} + \underbrace{2^{\varepsilon \cdot n}}_{\mathcal{D} \text{ on } \varphi}) = O^*(2^{\frac{1}{4} \cdot \delta_q \cdot n + \varepsilon \cdot n}) = O^*(2^{\frac{1}{2} \cdot \delta_q \cdot n}).$$

Altogether, we found an algorithm solving q-SAT in time $O^*(2^{\frac{1}{2} \cdot \delta_q \cdot n})$. By the definition of δ_q we obtain $0 \leq \delta_q \leq \frac{1}{2} \cdot \delta_q$ and therefore $\delta_q = 0$ which contradicts the above $\delta_q > 0$. This completes the proof. □

A.3.6 Proof of Lemma 5.23

Proof. Assume that the SETH holds. By definition this means that the sequence $(\delta_q)_{q\in\mathbb{N}}$ converges to 1. We are interested in the constant δ which reasons over the running time of algorithms for the general problem SAT. It is defined to be the infimum of the following set:

$$A = \{c \in \mathbb{R} \mid \text{There is an algorithm solving SAT in time } O^*(2^{c\cdot n})\}.$$

We show that $\delta_q \leq \delta$ for each $q \in \mathbb{N}$. To this end, let $q \in \mathbb{N}$ and let $c \in A$ be arbitrary. Then, there exists an algorithm $\mathcal{A}$ solving SAT in time $O^*(2^{c\cdot n})$. Clearly, $\mathcal{A}$ also solves the problem q-SAT. But this means $\delta_q \leq c$ and since c is an arbitrary element of A and δ is the infimum, we obtain that $\delta_q \leq \delta$.

Next, note that $\delta \leq 1$ since we can always solve SAT via a simple brute force approach in time $O^*(2^n)$. This implies that for each $q \in \mathbb{N}$ we have

$$\delta_q \leq \delta \leq 1.$$

We already know that $(\delta_q)_{q\in\mathbb{N}}$ converges to 1. By the above inequality, this means that $\delta = 1$. Now assume that there is an algorithm solving SAT in time $O^*((2-\varepsilon)^n)$ for some $\varepsilon > 0$. Set $c = \log(2-\varepsilon)$. Then, $c < 1$ and we have

$$2^{c\cdot n} = (2-\varepsilon)^n.$$

Hence, the aforementioned algorithm runs in time $O^*(2^{c\cdot n})$ and therefore we get that $1 = \delta \leq c < 1$ which is a contradiction. We can conclude that such an algorithm cannot exist which completes the proof. □

A.3.7 Proof of Lemma 5.28

Proof. To prove the lemma, consider the constant λ, defined similarly to the λ_q but demanding an algorithm for the general problem Set Cover. Formally,

$$\lambda = \inf\{c \in \mathbb{R} \mid \text{There is an algorithm solving Set Cover in time } O^*(2^{c\cdot n})\}.$$

Note that $\lambda \leq 1$ since we can solve Set Cover in time $O^*(2^n)$. Moreover, for each $q \in \mathbb{N}$, we have $\lambda_q \leq \lambda$. Indeed, any algorithm for Set Cover also solves the problem q-Set Cover within the same running time. Altogether:

$$\lambda_q \leq \lambda \leq 1.$$

Since $(\lambda_q)_{q\in\mathbb{N}}$ converges to 1 by the SCON, we obtain that $\lambda = 1$. Now assume there is an algorithm solving Set Cover in time $O^*((2-\varepsilon)^n)$ for some $\varepsilon > 0$. By setting $c = \log(2-\varepsilon) < 1$, the running time can be estimated by $O^*(2^{c\cdot n})$. Hence, by definition of λ, we can conclude that $1 = \lambda \leq c < 1$ which is a contradiction. Consequently, the algorithm cannot exist. □

A.3.8 Proof of Lemma 5.29

Proof. We describe the reduction. Let $(U, \mathcal{F}, r)$ be an instance of q-Set Cover. First, we may assume that $r \leq n$. Otherwise we could solve the instance in polynomial time since for each element $u \in U$ we could just search for a set in $\mathcal{F}$ containing u and add it to the cover. Moreover, we assume that r is divisible by q. If this is not the case, we need to add at most q many elements to U and corresponding singleton sets to $\mathcal{F}$ to achieve it [13]. Now we construct the family $\hat{\mathcal{F}}$ in such a way that it contains all the unions of q sets from $\mathcal{F}$:

$$\hat{\mathcal{F}} = \{\bigcup_{i=1}^{q} S_i \mid S_1, \ldots, S_q \in \mathcal{F} \text{ are distinct sets}\}.$$

Note that each set in the family $\hat{\mathcal{F}}$ has at most q^2 many elements. By iterating over all possible q-tuples of sets from $\mathcal{F}$, we can compute $\hat{\mathcal{F}}$ in time $O(m^q)$, where $m = |\mathcal{F}|$. This is indeed a polynomial expression since q is a constant. We define $\hat{r} = \frac{r}{q} \in \mathbb{N}$. Since $r \leq n$, we obtain the bound $\hat{r} \leq \frac{1}{q} \cdot n$.

It is left to show the correctness of the construction. First assume that $(U, \mathcal{F}, r)$ is a yes-instance of q-Set Cover. Then there are sets $S_1, \ldots, S_r \in \mathcal{F}$ that cover the universe U. Since r is divisible by q and $\hat{r} = \frac{r}{q}$, we can divide these sets into $\hat{r}$ many groups of q sets each:

$$\{S_1, \ldots, S_q\}, \{S_{q+1}, \ldots, S_{2 \cdot q}\}, \ldots, \{S_{(\hat{r}-1) \cdot q+1}, \ldots, S_r\}.$$

Since $\hat{\mathcal{F}}$ contains all unions of q sets from $\mathcal{F}$, it also contains the unions of the above groups. Let C_i be the union of the i-th group. Since the C_i cover U and all C_i are in $\hat{\mathcal{F}}$, the tuple $(U, \hat{\mathcal{F}}, \hat{r})$ is a yes-instance of q^2-Set Cover.

For the other direction, let the sets $C_1, \ldots, C_{\hat{r}} \in \hat{\mathcal{F}}$ cover U. By construction, each C_i is a union of q sets from $\mathcal{F}$. Let these be given by

$$\begin{aligned} C_1 &= S_1 \cup \ldots \cup S_q, \\ &\ldots \\ C_{\hat{r}} &= S_{(\hat{r}-1) \cdot q+1} \cup \ldots \cup S_r \end{aligned}$$

with each $S_j \in \mathcal{F}$. Then $S_1, \ldots, S_r$ is a cover of U with sets from $\mathcal{F}$. □

A.3.9 Proof of Theorem 5.30

Proof. To prove the result, we need to determine the limit of an appropriate sequence $(\sigma_q)_{q \in \mathbb{N}}$. Each element σ_q of it is defined to be the infimum among

[13] This obviously changes the universe U to some universe $\hat{U}$ other than stated in the lemma. However, we only add a constant number of elements, at most q many. For our purpose — and this is mainly Theorem 5.30 — we can ignore this change.

the exponents $d \in \mathbb{R}$ appearing in the running time $O^*(2^{d\cdot(n+r)})$ of an algorithm solving the problem q-Set Cover. Formally, we set σ_q to be the infimum of

$$B_q = \{d \in \mathbb{R} \mid \text{There is an algorithm solving } q\text{-Set Cover in } O^*(2^{d\cdot(n+r)})\}.$$

Note the difference to the definition of λ_q where we demand an algorithm depending solely on the parameter n in the exponent. Despite this difference, we show that both sequences, $(\lambda_q)_{q\in\mathbb{N}}$ and $(\sigma_q)_{q\in\mathbb{N}}$, actually have the same limit. To this end, first note that $(\sigma_q)_{q\in\mathbb{N}}$ is a convergent sequence as it is increasing, $\sigma_q \leq \sigma_{q+1}$. Moreover, the sequence is bounded by 1 since there are algorithms for Set Cover running in time $O^*(2^n) = O^*(2^{n+r})$.

For each $q \in \mathbb{N}$ we have $\lambda_q \geq \sigma_q$. Clearly, an algorithm solving q-Set Cover in time $O^*(2^{c\cdot n})$ also solves the problem in time $O^*(2^{c\cdot(n+r)})$. Hence, each such $c \in \mathbb{R}$ satisfies $c \geq \sigma_q$ and thus the infimum among these c, namely λ_q, also satisfies the inequality. We obtain that $\lambda_q \geq \sigma_q$.

Now we estimate certain elements of $(\sigma_q)_{q\in\mathbb{N}}$ from below. Fix $q \in \mathbb{N}$. We employ the powering technique of Lemma 5.29. Recall that it reduces an instance $(U, \mathcal{F}, r)$ of q-Set Cover to an instance $(U, \hat{\mathcal{F}}, \hat{r})$ of q^2-Set Cover such that $\hat{r} \leq \frac{1}{q} \cdot n$ and the universe U is not affected by the reduction. Let $d \in B_{q^2}$. Then, there is an $O^*(2^{d\cdot(n+r)})$-time algorithm $\mathcal{A}$ solving q^2-Set Cover. We construct an algorithm solving q-Set Cover. To an instance $(U, \mathcal{F}, r)$, we apply the powering technique and obtain an instance $(U, \hat{\mathcal{F}}, \hat{r})$. To the latter we apply $\mathcal{A}$. The algorithm solves q-Set Cover in time

$$O^*(2^{d\cdot(n+\hat{r})}) = O^*(2^{d\cdot(1+\frac{1}{q})\cdot n}).$$

By definition of λ_q, we obtain that $d \cdot (1 + \frac{1}{q}) \geq \lambda_q$ and since d was an arbitrary element of B_{q^2}, we obtain that $\sigma_{q^2} \cdot (1 + \frac{1}{q}) \geq \lambda_q$. With the above, we can conclude that for each $q \in \mathbb{N}$, the following inequality holds:

$$\lambda_{q^2} \cdot (1 + \frac{1}{q}) \geq \sigma_{q^2} \cdot (1 + \frac{1}{q}) \geq \lambda_q.$$

Since both sequences, $(\lambda_q)_{q\in\mathbb{N}}$ and $(\sigma_q)_{q\in\mathbb{N}}$, are convergent, also their subsequences converge to the corresponding limit. Therefore, we can deduce:

$$\lim_{q\to\infty} \lambda_q = \lim_{q\to\infty} \sigma_q.$$

Now we can show that an $O^*((2-\varepsilon)^{n+r})$-time algorithm for Set Cover contradicts the SCON. Assume the SCON holds but such such an algorithm, called C, still exists for some $\varepsilon > 0$. Consider σ, defined to be the infimum of

$$\{d \in \mathbb{R} \mid \text{There is an algorithm solving Set Cover in time } O^*(2^{d\cdot(n+r)})\}.$$

Clearly, an algorithm for Set Cover also solves instances of q-Set Cover. Hence, we obtain for each $q \in \mathbb{N}$ that $\sigma_q \leq \sigma$. Moreover, we have that $\sigma \leq 1$.

Since the sequence $(\sigma_q)_{q\in\mathbb{N}}$ converges to 1 by the SCON, we can conclude that $\sigma = 1$. Now set $d = \log(2-\varepsilon) < 1$. Then, algorithm C solves the problem Set Cover in time $O^*(2^{d\cdot(n+r)})$. This mans that $1 = \sigma \leq d < 1$, a contradiction. Hence, algorithm C cannot exist which completes the proof. □

A.3.10 Proof of Theorem 5.44

Proof. Let C be the cross-composition of S into Q and assume there is a polynomial compression $\mathcal{B}$ of Q into some problem R. Consider the problem

$$R^{\#} = \{d_1.\#.d_2.\#\ldots\#.d_\ell \mid \ell \in \mathbb{N} \text{ and there is an } i \in [1..\ell] \text{ with } d_i \in R\}.$$

The problem $R^{\#}$ accepts a string of the form $d_1.\#\ldots\#.d_\ell$ if one of the instances d_i already lies in R. Note that the symbol # is a fresh symbol, not used by R. Our goal is to show that appending $\mathcal{B}$ to C actually yields an OR-distillation of S into problem $R^{\#}$. However, according to Corollary 5.39, the latter does not exist unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$ holds.

In the following, we describe how the OR-distillation is set up. To this end, let $\mathcal{R}$ denote the polynomial equivalence relation which exists along with the cross-composition C. Moreover, let $s_1, \ldots, s_t$ be t given instances and $n = \max_{i\in[1..t]} |s_i|$. The OR-distillation proceeds in five steps:

In Step 1, duplicates among the given instances $s_1, \ldots, s_t \in \Sigma^*$ are removed. Note that this does not change whether there exists an instance s_i in S or not. Hence, we can now assume that the strings $s_1, \ldots, s_t$ are actually all distinct and since $|s_i| \leq n$ for each i, we get that $t \leq \sum_{j=0}^{n} |\Sigma|^j \leq |\Sigma|^{n+1}$. Phrased differently, we have that $\log(t) = O(n)$. Removing duplicates can clearly be done in time polynomially in n and t which is compatible with the polynomial-time constraint put on an OR-distillation.

Step 2 of the OR-distillation partitions the instances $s_1, \ldots, s_t$ into classes $C_1, \ldots, C_b$ according to $\mathcal{R}$. Since $\mathcal{R}$ has at most polynomially many equivalence classes when restricted to $\Sigma^{\leq n}$, we obtain that $b \leq p_0(n)$ for some polynomial $p_0(x)$. Setting up the classes C_j can be done in time polynomially in n and t since deciding equivalence under $\mathcal{R}$ takes polynomial time.

Now, in Step 3, we apply the cross composition C iteratively to each C_j. For increasing j, we feed C with the instances of C_j. Note that these are equivalent under $\mathcal{R}$. Consequently, we obtain an instance (y_j, k_j) such that

$$(y_j, k_j) \in Q \text{ if and only if there is an } s \in C_j \text{ with } s \in S,$$

and there is a polynomial $p_1(x)$ with $k_j \leq p_1(n + \log(|C_j|))$. Since $|C_j| \leq t$ and $\log(t) = O(n)$, we can further estimate the size of the parameter:

$$k_j \leq p_1(n + \log(t)) \leq p_2(n),$$

for some polynomial $p_2(x)$. Note that, after applying Step 3, we have the following equivalence: there exists an $i \in [1..t]$ with $s_i \in S$ if and only if

there is a $j \in [1..b]$ such that $(y_j, k_j) \in Q$. Computing Step 3 again takes time polynomially in n and t since C does and there are $b \leq t$ many classes C_j.

Step 4 applies the polynomial compression $\mathcal{B}$ to each instance (y_j, k_j). The output will be instances w_j, one for each $j \in [1..b]$ such that $w_j \in R$ if and only if $(y_j, k_j) \in Q$ and $|w_j| \leq p_3(k_j)$ for a polynomial $p_3(x)$. Since $k_j \leq p_2(n)$ is already bounded, we obtain that $|w_j| \leq p_4(n)$ for some polynomial $p_4(x)$. Note that the instances w_j still satisfy the equivalence: there exists an $i \in [1..t]$ with $s_i \in S$ if and only if there is a $j \in [1..b]$ such that $w_j \in R$. Moreover, applying Step 4 takes time polynomially in t and n since $\mathcal{B}$ does.

In Step 5 we construct the instance $w = w_1.\#\ldots\#.w_b$. Due to the above equivalence, we have that $w \in R^{\#}$ if and only if there is an $i \in [1..t]$ with $s_i \in S$. This meets Condition (1) of an OR-distillation. Moreover, we have

$$\begin{aligned} |w| \leq& (b-1) + b \cdot \max_{j \in [1..b]} |w_j| \\ \leq& (b-1) + b \cdot p_4(n) \\ \leq& p_0(n) + p_0(n) \cdot p_4(n) = p_5(n), \end{aligned}$$

for an appropriate polynomial $p_5(x)$. Hence, Condition (2) of an OR-distillation is also satisfied. Finally, the time constraint formulated in Condition (3) is also met since each of the steps takes time polynomially in n and t. □

B. Detailed Proofs and Further Concepts

We provide detailed proofs and elaborate on further concepts that were part of our fine-grained complexity analyses in Chapters 6, 7, 8, and 9.

B.1 Proofs for Chapter 6

B.1.1 Proof of Theorem 6.21

Proof. We show the correctness of the construction. To this end, assume that the constructed instance of BCS has a solution, namely a word $u \in L(S)$ with at most $2e$ context switches. Since $u \in L(M)$, we know it is of the form

$$u = (v_1, w_{i_1})^{d_1}.(v_2, w_{i_2})^{d_2} \dots (v_k, w_{i_k})^{d_k}$$

with $i_1 < \dots < i_k$ and $\sum_{i \in [1..k]} d_i = 2e$. Hence, u is of length exactly $2e$. The word u also lies in the shuffle $\text{⧢}_{i \in [1..e]} L(A_i)$. Since each A_i only accepts a word of length exactly 2, each A_i needs to be involved in the shuffle.

We define the map $\varphi : V(H) \to V(G)$ according to u by setting $\varphi(v_j) = w_{i_j}$. Since each vertex $v \in V(H)$ is adjacent to an edge $e_i \in E(H)$ and the corresponding thread A_i contributes to the shuffle, u contains a letter of the form (v, w_{i_j}) and therefore, φ has an entry for each vertex of H. Moreover, note that φ is injective due to the order $i_1 < \dots < i_k$.

We need to show that φ preserves the edges of H. Let $e_i = \{v_s, v_t\} \in E(H)$ be an edge and let $\varphi(v_s) = w_{i_s}$ and $\varphi(v_t) = w_{i_t}$ satisfy $i_s < i_t$. Towards a contradiction, assume that $\{\varphi(v_s), \varphi(v_t)\} \notin E(G)$. Then, A_i would not accept a subword of u of the form $(v_s, w_{i_s}).(v_t, w_{i_t})$. This contradicts the fact that A_i contributes in the shuffle and hence, all edges of H are mapped to edges of G.

Now assume that an embedding φ of H into G is given. We order the vertices $v_1, \dots, v_k \in V(H)$ with $\varphi(v_j) = w_{i_j}$ in such a way that $i_1 < \dots < i_k$. We define the word u, supposed to be in the language of the SMCP:

$$u = (v_1, w_{i_1})^{d_1} \dots (v_k, w_{i_k})^{d_k},$$

where d_j is the degree of v_j, the number of adjacent edges.

By definition, we have that $u \in L(M)$. Moreover, the word lies in the shuffle $\text{⧢}_{i \in [1..e]} L(A_i)$ as each edge $\{v_s, v_t\} \in E(H)$ is mapped to an edge $\{\varphi(v_s), \varphi(v_t)\} \in E(G)$. Note that, moreover, $|u| = 2e$. Hence, the word has at most $2e = cs$ many context switches.

Finally, we need to show that the reduction can be compute in polynomial time. To this end, we show that the size of the shared memory M and the threads A_i is polynomially bounded. The number of states of M is at most $O(|V(H)| \cdot |V(G)| \cdot |E(H)|)$. The automaton needs to remember the last letter and the number of letters already produced. Each of the A_i needs $|V(G)| + 2$ states as it only needs to remember the vertex of G read in the first letter. □

B.1.2 Proof of Theorem 6.23

Proof. We formally describe the cross-composition and the construction of the corresponding SMCP. The underlying alphabet is defined by

$$\Sigma = (\{x_1, \ldots, x_k\} \times \{?0, !0, ?1, !1\}) \cup \{\#\}.$$

Intuitively, $(x_i, ?0)$ corresponds to querying if variable x_i evaluates to 0 and $(x_i, !0)$ corresponds to verifying that x_i indeed evaluates to 0. The symbol # was added to prevent that the empty word lies in the language of the resulting SMCP. The latter is defined by $S = (\Sigma, M, (A_i)_{i \in [1..k]}, B)$.

For each variable x_i, the thread A_i keeps track of the value assigned to x_i. To this end, A_i has exactly three states: an initial state and two final states, one for each possible value. We have

$$L(A_i) = (x_i, !0)^+ \cup (x_i, !1)^+.$$

Thread B is responsible for checking whether one out of the t given formulas is satisfiable. To this end, B has the states

$$\{p, p_0, p_f\} \cup \{p_i^j \mid j \in [1..t], i \in [1..\ell - 1]\},$$

where p is the initial state, and p_f is the final state. For a fixed $j \in [1..t]$, the states $p_1^j, \ldots p_{\ell-1}^j$ are used to iterate through the ℓ clauses of φ_j. The transitions between p_i^j and p_{i+1}^j are labeled by $(x_s, ?0)$ or $(x_s, ?1)$, depending on whether variable x_s occurs with or without negation in the $(i+1)$-st clause of φ_j for $i \in [1..\ell - 2]$. Note that there are at most three transitions between p_i^j and p_{i+1}^j. For the first clause, the transitions start in p_0 and end in p_1^j, while for the ℓ-th clause, the transitions start in $p_{\ell-1}^j$ and end in p_f. Further, there is a transition from p to p_0 that is labeled by #.

To answer the requests of B, we use the shared memory M. It ensures that each request of the form $(x_s, ?1)$ is also followed by a confirmation of the form $(x_s, !1)$, and similar for value 0. The memory M has an initial state q_{init}, a final state q_f, and for each variable x_s, two states q_s^0 and q_s^1. We get a transition between q_f and each q_s^0, labeled by the request $(x_s, ?0)$. To get the confirmation, we introduce a transition from q_s^0 back to q_f, labeled by $(x_s, !0)$. We proceed similarly for q_f and q_s^1. To get from q_{init} to q_f initially, we

introduce a transition from q_{init} to q_f, labeled by the symbol #, preventing M from accepting the empty word.

We show the correctness of the construction: There exists a $j \in [1..t]$ such that φ_j is satisfiable if and only if $L(S) \cap \text{Context}(\Sigma, k+1, 2\ell) \neq \emptyset$.

First, let a $j \in [1..t]$ be given such that φ_j is satisfiable. Then there is a satisfying evaluation with assigns a value v_s to each variable $x_s, s \in [1..k]$. A word $w \in L(S) \cap \text{Context}(\Sigma, k+1, 2\ell)$ can be constructed as follows. Let $x_{s_1}, \ldots, x_{s_\ell}$ denote variables (repetition allowed) that contribute to satisfying the clauses of φ_j. This means that x_{s_i} can be used to satisfy the i-th clause of φ_j. We define the word

$$w = \#.(x_{s_1}, ?v_{s_1}).(x_{s_1}, !v_{s_1}) \ldots (x_{s_\ell}, ?v_{s_\ell}).(x_{s_\ell}, !v_{s_\ell}).$$

Then, we obtain that w indeed lies in $L(S) \cap \text{Context}(\Sigma, k+1, 2\ell)$.

For the other direction, let $w \in L(S) \cap \text{Context}(\Sigma, k+1, 2\ell)$ be given. Then w is of the following form:

$$w = \#.(x_{s_1}, ?v_{s_1}).(x_{s_1}, !v_{s_1}) \ldots (x_{s_\ell}, ?v_{s_\ell}).(x_{s_\ell}, !v_{i_\ell}).$$

Due to the construction of the threads A_i, note that $x_{s_i} = x_{s_{i'}}$ implies $v_{s_i} = v_{s_{i'}}$ in the word. Hence, we can construct a satisfying assignment v for one of the given φ_j. We assign each x_{s_i} occurring in w the value v_{s_i}. For variables that do not occur in w, we can assign 0 or 1.

By construction, B iterates through the clauses of one of the given φ_j. Since B also accepts during the computation of w, there is a $j \in [1..t]$ such that all the clauses of φ_j can be satisfied by v. Hence, φ_j is satisfiable.

Finally, the parameters of the constructed BCS-instance are the size of the memory, $m = 2k + 2$ and the number of context switches $cs = 2\ell$. Both are bounded by $\max_{j \in [1..t]} |\varphi_j|$ and therefore independent of t. Hence, all requirements on a cross-composition are met. □

B.1.3 Proof of Lemma 6.26

Proof. Formally, we construct the SMCP $S' = (\Gamma, M', (B_i)_{i \in [1..cs+1]})$. The first step is to define the underlying alphabet:

$$\Gamma = \Sigma \cup \{\#_1, \ldots, \#_t\} \cup \{\#\}.$$

To define the B_i, assume that the threads of the SMCP S are given by $A_j = (P_j, \Sigma \times \{j\}, \delta_j, p_j^0, p_j^f)$. Automaton $B_i = (P_i', \Gamma \times \{i\}, \delta_i', p_i^{init}, F_i)$ is the union of all A_j with a new preceding initial state and an additional state which simulates that B_i is inactive. Formally, the states of B_i are given by

$$P_i' = \bigcup_{j \in [1..t]} P_j \cup \{p_i^{init}, p_i^{inact}\}.$$

The initial state is p_i^{init}. The set of final states F_i is given by the final states of each of the A_j and the inactive state. We have $F_i = \bigcup_{i\in[1..t]} p_j^f \cup \{p_i^{inact}\}$. Note that we did not formally define the threads of an SMCP to have a set of final states. However, the case of a single final state can always be recovered by simply adding one additional new final state. The transition relation δ_i' contains each δ_j. Moreover, we add the transitions

$$p_i^{init} \xrightarrow{\#_j} p_j^0 \text{ for } j \in [1..t] \text{ and } p_i^{init} \xrightarrow{\#} p_i^{inact}.$$

The language of B_i is then indeed given by $L(B_i) = \# \cup \bigcup_{j\in[1..t]} \#_j.L(A_j)$.

To define the shared memory of S', let $M = (Q, \Sigma, \delta, q_0, q_f)$ be the memory of S. We define $M' = (Q', \Gamma, \delta', q^{init}, q_f)$, where

$$Q' = Q \cup \{q_{(\#_j,i)} \mid j \in [1..t] \text{ and } i \in [1..cs]\} \cup \{q^{init}\}.$$

A state $q_{(\#_j,i)}$ stores the last seen symbol $\#_j$ and the number i of automata that has already been activated. Such a state has only outgoing transitions with $\#_{j'}$ where $j < j'$. Note that the initial state is q^{init} and the final state q_f is the final state of M. The transition relation δ' is the union of the transition relation of M and the transitions that are explained below.

To force each B_i to make a choice of which thread A_j to simulate, we add

$$q_{(\#_j,i)} \xrightarrow{\#_l} q_{(\#_l,i+1)}$$

for $j \in [1..t]$, $j < l \leq t$ and $i \in [1..cs-1]$. For the initial choice, we add

$$q^{init} \xrightarrow{\#_l} q_{(\#_l,1)}$$

for all $l \in [1..t]$. Finally, for the last choice, we add the transitions

$$q_{(\#_j,cs)} \xrightarrow{\#_l} q_0$$

for $j \in [1..t]$ and $j < l \leq t$.

To give the B_i the opportunity not to simulate any of the A_j and to be deactivated, we add the following transitions

$$q_{(\#_j,i)} \xrightarrow{\#} q_{(\#_j,i+1)} \text{ for } j \in [1..t] \text{ and } q_{(\#_j,cs)} \xrightarrow{\#} q_0.$$

In the first phase of a computation, namely where the B_i choose a thread A_j to simulate, we do exactly cs context switches. Then we need at most cs further context switches for the simulation of the chosen A_j and M. The construction immediately implies the equivalence:

$$L(S') \cap \text{Context}(\Gamma, cs+1, 2cs) \neq \emptyset \text{ if and only if } L(S) \cap \text{Context}(\Sigma, t, cs) \neq \emptyset.$$

This shows the correctness and completes the proof. □

B.1.4 Proof of Lemma 6.27

Proof. In the given instance (S, cs) of BCS with SMCP $S = (\Sigma, M, (A_i)_{i\in[1..t]})$, let the shared memory be given by $M = (Q, \Sigma, \delta, q_0, q_f)$ and let the threads be given by the tuples $A_i = (P_i, \Sigma \times \{i\}, \delta_i, p_i^0, p_i^f)$.

The Turing Machine N works in three different modes, stored in its states: a *switch* mode which performs the context switches and chooses a thread to simulate, a *simulation* mode which simulates contexts, and an *accept* mode which checks if M and all simulated A_i arrive at their final state. Formally,

$$N = (Q_N, \Sigma_N, \Gamma_N, \delta_N, q_{init}, q_{acc}, q_{rej}),$$

where the states are given by

$$Q_N = \{switch, sim\} \times Q \times [0..cs+2] \cup \{acc\} \times [0..t] \cup \{q_{acc}, q_{rej}\}.$$

Note that a state either stores the current mode *switch* or *sim* along with the current state of M and the context counter, or the mode *acc* along with the number of threads that are accepting. The initial state is given by $q_{init} = (switch, q_0, 0)$. Hence, N starts in switch mode since it has to choose a thread for the first context. The accepting state is q_{acc} and the rejecting state is q_{rej}. The tape alphabet of N is given by

$$\Gamma_N = \bigcup_{i\in[1..t]} P_i \cup \{S_1, \ldots, S_t, \$\}.$$

It consists of all states of the A_i. We assume them to be disjoint. Moreover, there are fresh symbols $S_1, \ldots, S_t$. Their meaning will become clear in a moment. The symbol \$ represents the *left-end marker* of the machine's tape.

The input word of N is $w = S_1 \ldots S_t$. Instead of putting the initial state q_i^0 of A_i into the i-th cell, we use the additional symbol S_i to describe that A_i did not participate in the computation of S that is simulated by N. If S_i still remains after the simulation has been completed, we know that A_i did not contribute. Note that the same reasoning does not apply when we replace S_i by the initial state q_i^0 since the state can be the final state as well.

We elaborate on the transitions δ_N. Whenever N is in switch mode, a thread A_i is chosen to simulate. We allow N for moving left and right on the tape, remembering the current state of M and the number of contexts taken but without changing the tape content. For $q \in Q$ and $j \leq cs$, we have

$$((switch, q, j), p) \to ((switch, q, j), p, D),$$

where $D \in \{L, R\}$ is a direction and $p \in \Gamma_N$. Although not explicitly stated, we forbid that, on the left-end marker \$, we can further turn to the left. The machine N can also change the mode to *sim*. In this case, we start, or continue, the computation of the chosen thread A_i. For $j \leq cs$, we have

$$((switch, q, j), p) \to ((sim, q, j), p, D).$$

Once the simulation mode is activated, there are two cases to distinguish. Either the chosen thread A_i did not move before. Then, the currently visited cell holds S_i. Or it has moved before. Then, the cell holds the current state of A_i. In the first case, we need a transition that activates A_i and performs a first context. This has to be synchronized with M. We have

$$((sim, q, j), S_i) \to ((switch, q', j+1), p', D),$$

for states $q, q' \in Q$ and $p' \in P_i$ such that $L(M(q, q')) \cap L(A_i(p_i^0, p')) \neq \emptyset$.

In the latter case, we continue the computation of A_i. The corresponding context is synchronized with the shared memory M. We have the transitions

$$((sim, q, j), p) \to ((switch, q', j+1), p', D),$$

for all states $q, q' \in Q$ and $p, p' \in P_i$ with $L(M(q, q')) \cap L(A_i(p, p')) \neq \emptyset$.

Note that after changing the mode from *sim* to *switch*, a context has been successfully simulated. To track their number, N increases the context counter by 1. The counter is not allowed to go beyond $cs + 1$. Once this happens, the machine N will immediately reject. For all $q \in Q$ and $p \in \Gamma_N$, we have

$$((switch, q, cs + 2), p) \to q_{rej}.$$

If N arrives in a state of the form $(switch, q, j)$, where $q = q_f$ is the final state of M and $j \in [1..cs + 1]$, it can enter accept mode.

$$((switch, q, j), p) \to ((acc, 0), p, D).$$

Once N is in mode *acc*, it moves the head to the left end of the tape without changing the tape content and its current state. Then the idea is then to scan the tape, from left to right, and to check whether each participating A_i arrived at a final state. In this case, we increase a particular counter. For threads A_i that did not contribute, there is still the symbol S_i in the i-th cell. We also count these as accepting. We need the following transitions:

$$((acc, j), p) \to ((acc, j+1), p, R),$$

for p a final state of one of the A_i or $p \in \{S_1, \ldots, S_t\}$. As soon as N detects that all threads were either accepting or not participating at all, the machine will accept the input. For $p \in \Gamma_N$, we have

$$((acc, t), p) \to q_{acc}.$$

To simulate at most $cs + 1$ different contexts of the A_i with M, the machine N needs to take at most $t \cdot (cs + 1)$ many steps. Once TM enters the mode *acc*, it takes an additional $2t$ steps to verify that each A_i is in its final state or did not participate at all. Hence, we are looking for a computation of N of length at most $t \cdot (cs + 1) + 2t = t \cdot (cs + 3)$. The correctness is immediate and clearly, N and w can be constructed in polynomial time. □

B.1.5 Relating Scheduling Dimension and Carving Width

In order to relate scheduling dimension and carving width, we first formally introduce the latter. Like many other graph measures, also the carving width relies on a notion of decompositions.

Definition B.1. Let $G = (V, E)$ be an undirected multigraph. A *carving decomposition* of G is a tuple (T, φ), where T is a binary tree and φ is a bijection mapping the leaves of T to the vertices V.

For an edge e of T, removing e from T partitions the tree into two connected components. Let $S_1, S_2 \subseteq V$ be the images under φ, of the leaves falling into these components. We define the *width* of e to be the integer

$$width(e) = E(S_1, S_2) = \sum_{u \in S_1, v \in S_2} E(u, v).$$

The *width* of the decomposition (T, φ) is the maximum width of all edges in T. The *carving width of* G is the minimum width among all decompositions:

$$cw(G) = \min\{width(T, \varphi) \mid (T, \varphi) \text{ a carving decomposition of } G\}.$$

We are now ready to prove Lemma 6.34.

Proof of Lemma 6.34. First we show that from a given carving decomposition (T, φ) of G' of width k, we can construct a contraction process π of G of degree at most k. This implies $sdim(G) \leq cw(G')$. The idea is to inductively assign to each node of T a partial contraction process of G of degree at most k. We start at the leaves of T and go bottom-up. At the end, the needed contraction process will be the one assigned to the root.

Before we start, we fix some notation. Let w be a vertex occurring in a partial contraction processes starting in G. By $V(w) \subseteq V$, we denote the set of vertices of G that get contracted to w. Note that, if two vertices u, v get contracted to w during the process, we have $V(w) = V(u) \cup V(v)$. For a node n of th tree T, we use $leaf(n) \subseteq V$ to denote the image, under φ, of the leaves of the subtree of T rooted in n.

Now we show the following. We can assign to any node n in T a pair (π_n, w), where $\pi_n = G_1, \ldots, G_\ell$ is a partial contraction process such that $G_1 = G$, $\deg(G_i) \leq k$, for each $i \in [1..\ell]$ and w is a vertex in $V(G_\ell)$ with $V(w) = leaf(n)$, and $V(G_\ell) = V \setminus leaf(n) \cup \{w\}$. The latter invariants ensure that the process contracts the vertices in $leaf(n)$ to w and furthermore, no other vertices in G are contracted. The process that we assign to the root r is thus a proper contraction process of G, contracting all vertices of G to a single vertex. Moreover, the degree of the process is bounded by k.

To start the induction, we assign any leaf n of T the pair $(G, \varphi(n))$. Note that $leaf(n) = \{\varphi(n)\} = V(\varphi(n))$. Hence, we only need to elaborate on the

degree of G since the invariants are satisfied. For any $v \in V$, we have

$$\begin{aligned}\deg(v) &= \max\{\text{indeg}(v), \text{outdeg}(v)\} \\ &= \max\{\sum_{u \in V} E(u,v), \sum_{u \in V} E(v,u)\} \\ &\leq \sum_{u \in V} \max\{E(u,v), E(v,u)\} \\ &= \sum_{u \in V} E'(v,u) = E'(v, V \setminus \{v\}).\end{aligned}$$

Let n' denote the leaf with $\varphi(n') = v$. Furthermore, let e be the edge of T connecting n' with its parent node. Then $\mathit{width}(e) = E'(v, V \setminus \{v\})$. Since the width of e is bounded by k, we also get that $\deg(v) \leq k$ and thus, $\deg(G) \leq k$.

Now suppose we have a node n of T with two children n_1 and n_2 that are already assigned pairs (π_1, w_1) and (π_2, w_2) with partial contraction processes

$$\pi_1 = G_1, \ldots, G_\ell \text{ and } \pi_2 = H_1, \ldots, H_t,$$

where $G_1 = H_1 = G$ and $\deg(G_i), \deg(H_j) \leq k$ for $i \in [1..\ell], j \in [1..t]$. Furthermore, $w_1 \in V(G_\ell)$ and $w_2 \in V(H_t)$ satisfy the invariants:

$$\begin{aligned}V(w_1) = \mathit{leaf}(n_1) \text{ and } V(G_\ell) = V \setminus \mathit{leaf}(n_1) \cup \{w_1\}, \\ V(w_2) = \mathit{leaf}(n_2) \text{ and } V(H_t) = V \setminus \mathit{leaf}(n_2) \cup \{w_2\}.\end{aligned}$$

We also fix the notation for the contractions applied in π_2. Let σ_i be the contraction applied to H_i to obtain H_{i+1}. Hence, $H_{i+1} = H_i[\sigma_i]$ for $i \in [1..t-1]$.

We construct the partial contraction process that performs the contractions of π_1, the contractions of π_2, and contracts w_1 and w_2 to a vertex w. Set

$$\pi_n = G_1, \ldots, G_\ell, G_{\ell+1}, \ldots, G_{\ell+t},$$

where $G_{\ell+i} = G_{\ell+i-1}[\sigma_i]$, for $i \in [1..t-1]$ performs the contractions of π_2 and $G_{\ell+t} = G_{\ell+t-1}[w_1, w_2 \mapsto w]$. Then π_n is well-defined. Since the intersection $\mathit{leaf}(n_1) \cap \mathit{leaf}(n_2)$ is empty, we have that $V(G_\ell) = V \setminus \mathit{leaf}(n_1) \cup \{w_1\}$ contains $\mathit{leaf}(n_2)$. Thus, it is possible to apply the contractions $\sigma_1, \ldots, \sigma_{t-1}$ to G_ℓ since they only contract vertices from $\mathit{leaf}(n_2)$. Assume that a vertex $v \in V \setminus \mathit{leaf}(n_2)$ would be contracted during π_2. Then $v \notin V(H_t) = V \setminus \mathit{leaf}(n_2) \cup \{w_2\}$. Since $v \neq w_2$, we would get that v is in $\mathit{leaf}(n_2)$ which is a contradiction.

We assign to n the pair (π_n, w). What is left to prove is that the above invariants are satisfied. For the vertex $w \in V(G_{\ell+t})$, we have:

$$V(w) = V(w_1) \cup V(w_2) = \mathit{leaf}(n_1) \cup \mathit{leaf}(n_2) = \mathit{leaf}(n).$$

Since we apply the contractions of π_1 and π_2 to obtain $G_{\ell+t-1}$, we get:

$$\begin{aligned}V(G_{\ell+t-1}) &= (V(G_\ell) \cap V(H_t)) \cup \{w_1, w_2\} \\ &= (V \setminus \mathit{leaf}(n_1) \cap V \setminus \mathit{leaf}(n_2)) \cup \{w_1, w_2\} \\ &= V \setminus (\mathit{leaf}(n_1) \cup \mathit{leaf}(n_2)) \cup \{w_1, w_2\} \\ &= V \setminus \mathit{leaf}(n) \cup \{w_1, w_2\}.\end{aligned}$$

The graph $G_{\ell+t}$ is obtained by contracting w_1 and w_2 in $G_{\ell+t-1}$. Hence, we have that $V(G_{\ell+t}) = V(G_{\ell+t-1}) \setminus \{w_1, w_2\} \cup \{w\} = V \setminus \mathit{leaf}(n) \cup \{w\}$.

No we prove that all occurring graphs in π_n have degree bounded by k. It is clear by assumption that this holds for $G_1, \ldots, G_\ell$. We show the same for $G_{\ell+i}$ with $i \in [1..t-1]$. Let $u \in V(G_{\ell+i})$. We distinguish three cases.

(1) If $V(u) \subseteq \mathit{leaf}(n_1)$, then we have that none of the σ_j acts on u since this would imply that a vertex from $\mathit{leaf}(n_1)$ gets contracted by σ_j which is not possible. Hence, $u \in V(G_\ell)$ and $\deg_{G_{\ell+i}}(u) = \deg_{G_\ell}(u) \leq k$. Note that by $\deg_A(v)$, we indicate the degree of a vertex v in a graph A.

(2) If $V(u) \subseteq \mathit{leaf}(n_2)$, then no contraction of π_1 acts on u and u is a vertex that occurs during the application of $\sigma_1, \ldots, \sigma_i$. Hence, $u \in V(H_{i+1})$ and the degree is bounded: $\deg_{G_{\ell+i}}(u) = \deg_{H_{i+1}}(u) \leq k$.

(3) If $V(u) \subseteq V \setminus (\mathit{leaf}(n_1) \cup \mathit{leaf}(n_2))$, then u is neither involved in the contractions of π_1 nor in the contractions of π_2. Hence, $u \in V$ and, again the degree of the vertex is bounded: $\deg_{G_{\ell+i}}(u) = \deg_G(u) \leq k$.

Finally, we prove that the graph $G_{\ell+t}$ has degree bounded by k. For a vertex $u \neq w$ in $V(G_{\ell+t})$, we have $\deg_{G_{\ell+t}}(u) = \deg_{G_{\ell+t-1}}(u)$ since u is not involved in the contraction $[w_1, w_2 \mapsto w]$ that is applied to $G_{\ell+t-1}$ in order to obtain $G_{\ell+t}$. Now we consider w. First note, that $\mathrm{indeg}_{G_{\ell+t}}(w) = E(V \setminus V(w), V(w))$ and $\mathrm{outdeg}_{G_{\ell+t}}(w) = E(V(w), V \setminus V(w))$. Then we can derive:

$$
\begin{aligned}
\deg_{G_{\ell+t}}(w) &= \max\{\mathrm{indeg}_{G_{\ell+t}}(w), \mathrm{outdeg}_{G_{\ell+t}}(w)\} \\
&= \max\{E(V \setminus V(w), V(w)), E(V(w), V \setminus V(w))\} \\
&= \max\{\sum_{u \in V(w), v \in V \setminus V(w)} E(v,u), \sum_{u \in V(w), v \in V \setminus V(w)} E(u,v)\} \\
&\leq \sum_{u \in V(w), v \in V \setminus V(w)} \max\{E(v,u), E(u,v)\} \\
&= \sum_{u \in V(w), v \in V \setminus V(w)} E'(v,u) \\
&= E'(V(w), V \setminus V(w)).
\end{aligned}
$$

Let e denote the edge between n and its parent node. Then the width is given by $\mathit{width}(e) = E'(V(w), V \setminus V(w))$. Since the width is bounded by k, we get that $\deg_{G_{\ell+t}}(w)$ is bounded by k and hence, $\deg(G_{\ell+t}) \leq k$. Note that in the case where n is the root, the degree of w is 0. This completes the proof of the first stated inequality: $\mathit{sdim}(G) \leq \mathit{cw}(G')$.

To prove that $\mathit{cw}(G') \leq 2\mathit{sdim}(G)$, we show how to turn a given contraction process π of G of degree k into a carving decomposition (T, φ) of G' of width at most $2k$. Let $\pi = G_1, \ldots, G_{|V|}$ be the given process. We inductively construct a tree T with a labeling $\lambda : V(T) \to \bigcup_{i \in [1..|V|]} V(G_i)$ that assigns to each node of T a vertex from one of the G_i. We start with a root node r and set $\lambda(r) = w$,

where w is the latest vertex that was introduced by the contraction process:

$$G_{|V|} = G_{|V|-1}[w_1, w_2 \mapsto w].$$

Now suppose, we are given a node n of T with $\lambda(n) = v$. Assume v occurs in π on the right hand side of a contraction. This means there is a j such that $G_{j+1} = G_j[v_1, v_2 \mapsto v]$. We add two children n_1 and n_2 to the tree T and set $\lambda(n_i) = v_i$ for $i = 1, 2$. If v does not occur on the right hand side of a contraction in π then v is clearly a vertex of G. In this case, we stop the process on this branch and n remains as a leaf of T.

By the process, we obtain a tree T where the leaves are labeled by vertices of G. If we set φ to be λ restricted to the leaves, then φ is a bijection between the leaves of T and V. Consequently, (T, φ) is a carving decomposition of G'.

Now we show by induction on the structure of T that for each node n of T we have: $V(\lambda(n)) = \mathit{leaf}(n)$. Recall that $\mathit{leaf}(n)$ is the image, under φ, of the leaves of the subtree of T rooted in n. We start at the leaves of T. Let l be a leaf, then we have: $\mathit{leaf}(l) = \{\lambda(l)\}$ and moreover $V(\lambda(l)) = \{\lambda(l)\}$. For a node n of T with children n_1, n_2 such that the equations $\mathit{leaf}(n_i) = V(\lambda(n_i))$ already hold for $i = 1, 2$, we get the following:

$$\mathit{leaf}(n) = \mathit{leaf}(n_1) \cup \mathit{leaf}(n_2) = V(\lambda(n_1)) \cup V(\lambda(n_2)).$$

Since n_1, n_2 are the children of n, we get by the construction of T that there is a contraction in π of the form $G_{j+1} = G_j[\lambda(n_1), \lambda(n_2) \mapsto \lambda(n)]$ and hence:

$$V(\lambda(n_1)) \cup V(\lambda(n_2)) = V(\lambda(n)).$$

Finally, we show that the width of the carving decomposition (T, φ) is at most $2k$. To this end, let e be an edge in T, connecting the node n with its parent node. Further, let w denote $\lambda(n)$ and $w \in V(G_j)$. Then we have:

$$\begin{aligned}
\mathit{width}(e) &= E'(\mathit{leaf}(n), V \setminus \mathit{leaf}(n)) \\
&= E'(V(w), V \setminus V(w)) \\
&= \sum_{u \in V(w), v \in V \setminus V(w)} E'(u, v) \\
&= \sum_{u \in V(w), v \in V \setminus V(w)} \max\{E(u, v), E(v, u)\} \\
&\leq \sum_{u \in V(w), v \in V \setminus V(w)} (E(u, v) + E(v, u)) \\
&= E(V(w), V \setminus V(w)) + E(V \setminus V(w), V(w)) \\
&= \mathrm{outdeg}_{G_j}(w) + \mathrm{indeg}_{G_j}(w).
\end{aligned}$$

Since $\deg(\pi)$ is bounded by k, also the degree of G_j is bounded by k and hence, $\mathit{width}(e)$ is at most $2k$. Altogether, the width of (T, φ) is at most $2k$. □

B.1.6 BCS under Bounded Scheduling Dimension

We consider BCS restricted to schedules of bounded dimension. To formally define the problem, let $Sched(\Sigma, t, s)$ be those words over the alphabet $\Sigma \times [1..t]$ the scheduling graphs of which have dimension bounded by s:

$$Sched(\Sigma, t, s) = \{w \in (\Sigma \times [1..t])^* \mid sdim(G(w)) \leq s\}.$$

The *bounded scheduling* problem (Bounded Sched) then asks whether a given SMCP contains a word with dimension bounded scheduling graph.

Bounded Sched

Input: An SMCP $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ and a bound $s \in \mathbb{N}$.
Question: Is $L(S) \cap Sched(\Sigma, t, s) \neq \emptyset$?

We provide an algorithm for Bounded Sched showing that the parameterization in m, s, and t is fixed-parameter tractable. It is yet an open problem whether two of these parameters already suffice for an FPT-algorithm.

Theorem B.2. *Bounded Sched can be solved in time* $O^*((2m)^{4s} \cdot 4^t)$.

We present a fixed-point iteration that mimics the definition of contraction processes by iteratively joining the interface sequences of neighboring threads. To this end, we need a suitable notion of a composition operation.

Definition B.3. Let $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ be an SMCP with shared memory $M = (Q, \Sigma, \delta_M, q_0, q_f)$ and let $\rho \in (Q \times Q)^*$ be an interface sequence. A *contraction of* ρ is an interface sequence ρ' obtained from summarizing subsequences of ρ. Summarizing $(q_1, q_1') \dots (q_k, q_k')$ where $q_i' = q_{i+1}$ for $i \in [1..k-1]$ means contracting the sequence to (q_1, q_k'). By $con(\rho) \subseteq (Q \times Q)^*$ we denote the set of all interface sequences that are contractions of ρ.

For two interface sequences $\sigma, \tau \in (Q \times Q)^*$, we define their *contraction product* to be the following set of interface sequences:

$$\sigma \otimes \tau = \bigcup_{\rho \in \sigma \amalg \tau} con(\rho).$$

The contraction product contains all interface sequences that are obtained from shuffling σ and τ and then summarizing subsequences. If the resulting interface sequences should be of length at most $k \in \mathbb{N}$, we use the notation $\sigma \otimes^k \tau$ for the corresponding product. Formally, $\sigma \otimes^k \tau = (\sigma \otimes \tau) \cap (Q \times Q)^{\leq k}$.

Given an SMCP $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ and a bound $s \in \mathbb{N}$, the idea of our algorithm is to compute a fixed point over the powerset lattice

$$GIF = \mathcal{P}((Q \times Q)^{\leq s} \times \mathcal{P}([1..t])).$$

The elements in *GIF* are sets of *generalized interface sequences*, pairs consisting of an interface sequence together with the set of threads that has been used to construct it. We generalize the contraction product $\otimes^k$ to the domain.

Definition B.4. Let $S = (\Sigma, M, (A_i)_{i\in[1..t]})$ be an SMCP with shared memory $M = (Q, \Sigma, \delta_M, q_0, q_f)$ and $k \in \mathbb{N}$ an integer. Moreover, let the pairs $(\sigma_1, T_1), (\sigma_2, T_2) \in GIF$ be two generalized interface sequences. Their *contraction product* is the set of generalized interface sequences

$$(\sigma_1, T_1) \otimes^k (\sigma_2, T_2) = \begin{cases} (\sigma_1 \otimes^k \sigma_2) \times (T_1 \cup T_2), \text{ if } T_1 \cap T_2 = \emptyset, \\ \emptyset, \text{ otherwise.} \end{cases}$$

The definition states that $(\sigma_1, T_1), (\sigma_2, T_2)$ can only be contracted if the sets of threads T_1 and T_2 are disjoint. Otherwise, σ_1 and σ_2 would contain an interface sequence stemming from the same thread *id*. Contracting the two sequences would result in a computation where we have two copies of *id*. Therefore, it would not be conform with the program S.

Now we can formulate the fixed-point iteration that constitutes the basis of our algorithm. It begins by computing the interface languages of the threads and then combines them with the contraction product:

$$L_1 = \bigcup_{j\in[1..t]} IF(A_j) \times \{\{j\}\},$$
$$L_{i+1} = L_i \cup (L_i \otimes^s L_i).$$

Note that the fixed-point iteration terminates since the underlying domain *GIF* is finite. The following lemma states that the iteration is correct.

Lemma B.5. *We have that $L(S) \cap Sched(\Sigma, t, s) \neq \emptyset$ if and only if the least fixed point contains the pair $((q_0, q_f), T)$ for some $T \subseteq [1..t]$.*

Before we provide a proof of the lemma, we analyze the complexity of the iteration. First, we determine the precise number of steps that we require for the computation of the fixed point. To this end, note that $\otimes^s$ requires that the sets of threads of its operands are disjoint. This means that, after t many steps, we have covered all subsets of $[1..t]$ and the computation will stop.

Note that $L_i \subseteq (Q \times Q)^{\leq s} \times \mathcal{P}([1..t])$. Hence, $|L_i| \leq m^{2(s+1)} \cdot 2^t$. This means that, in each step, we need to compute the product $\otimes^s$ for at most

$$(m^{2(s+1)} \cdot 2^t)^2 = m^{4s+4} \cdot 4^t$$

many combinations of generalized interface sequences.

Computing a contraction product $(\sigma_1, T_1) \otimes^s (\sigma_2, T_2)$ requires to consider all $\rho \in \sigma_1 \text{Ш} \sigma_2$. Forming a shuffle of σ_1 and σ_2 can be understood as setting

$|\sigma_2|$ bits in a bitstring of length $|\sigma_1| + |\sigma_2|$. Hence, the number of shuffles is

$$\binom{|\sigma_1| + |\sigma_2|}{|\sigma_2|} \leq 2^{|\sigma_1|+|\sigma_2|} \leq 2^{2s} = 4^s.$$

Given ρ, we determine $con(\rho)$ by iteratively forming summaries. In the worst case, ρ has length $2s$. We mark an even number of positions in ρ and summarize the intervals between every pair of markers $2i$ and $2i + 1$. Since we can choose at most s pairs of these positions, we obtain that

$$con(\rho) \leq \sum_{i=0}^{s} \binom{2s}{2i} \leq 4^s.$$

Altogether, the algorithm takes time

$$O(t \cdot m^{4s+4} \cdot 4^t \cdot 4^s \cdot 4^s) = O^*((2m)^{4s} \cdot 4^t).$$

To complete the proof of Theorem B.2, it is left to show the correctness of the fixed-point iteration, as it was stated in Lemma B.5.

Proof of Lemma B.5. First, suppose that $L(S) \cap Sched(\Sigma, t, s) \neq \emptyset$. Then there exists a word w in the intersection with scheduling graph $G(u) = (V, E)$ such that $sdim(G(w)) \leq s$. We may assume that $V = [1..t]$. This means that all given threads participate in the computation. If this is not the case, we can delete the non-participating threads in the instance. By assumption, there is a contraction process $\pi = G_1, \ldots, G_{|V|}$ of $G(w)$ such that $\deg(\pi) \leq s$.

We now associate to each vertex in G_i an generalized interface sequence from $(Q \times Q)^{\leq s} \times \mathcal{P}([1..t])$. To this end, let $w = w_1 \ldots w_k$ be the context decomposition of w according to the computation of S on w. Furthermore, let q_j be the memory state of M, reached after reading $w_1 \ldots w_j$ with $j \in [1..k]$. Note that $q_k = q_f$. We get the interface sequence $\alpha = (q_0, q_1).(q_1, q_2) \ldots (q_{k-1}, q_k)$ by taking the pairs of states corresponding to each w_j.

From α, we get an interface sequence σ_i for each thread $i \in [1..t]$ by deleting those pairs of the contexts in which thread i was not active. Note that the length (the number of pairs) of σ_i is the number of times process i is active in w. Further note that this is the degree of vertex i in $G(w)$.

We define the map $\lambda_1 : [1..t] \to (Q \times Q)^{\leq s} \times \mathcal{P}([1..t])$ associating to each vertex of G_1 the corresponding generalized interface sequence:

$$\lambda_1(i) = (\sigma_i, \{i\})$$

for $i \in [1..t]$. Note that $\sigma_i \in IF(A_i)$ and therefore, $\lambda_1(i) \in L_1$ for any i.

The idea is to inductively construct a similar map

$$\lambda_j : V(G_j) \to (Q \times Q)^{\leq s} \times \mathcal{P}([1..t])$$

for each $j \in [1..t]$. Since we already considered the induction basis, assume that we are given λ_j with $j \leq t$. In order to construct the map λ_{j+1}, let $G_{j+1} = G_j[v_1, v_2 \mapsto v]$. Then, $V(G_{j+1}) = (V(G_j) \setminus \{v_1, v_2\}) \cup \{v\}$.

For $u \in V(G_j) \setminus \{v_1, v_2\}$, we set $\lambda_{j+1}(u) = \lambda_j(u)$. For the image of v, let $\lambda_j(v_1) = (\tau_1, T_1)$ and $\lambda_j(v_2) = (\tau_2, T_2)$. Set $T = T_1 \cup T_2$. Further, let σ_v be an interface sequence, obtained from α as follows. First mark all the pairs in α that were induced by a thread i in T. We contract adjacent pairs that are marked: if $(q_{i-1}, q_i)(q_i, q_{i+1})$ are both marked, then we contract it to (q_{i-1}, q_{i+1}) and mark the resulting pair. We repeat the process until we can no longer find an adjacently marked pair. Then we delete all the memory pairs remaining unmarked. The resulting interface sequence is denoted by σ_v. We set the image of v to be $\lambda_{j+1}(v) = (\sigma_v, T)$.

Note that contracting adjacently marked pairs corresponds to deleting edges between T_1 and T_2 in the scheduling graph $G(w)$. Hence, it is the same as contracting the corresponding nodes v_1 and v_2 in the graph G_j. We get that the length of σ_v is the degree of v in G_j, which is bounded by s. Moreover, since $\lambda_j(v_1), \lambda_j(v_2) \in L_j$ by induction, we get that

$$\lambda_{j+1}(v) \in (Q \times Q)^{\leq s} \cap (\lambda_j(v_1) \otimes^k \lambda_j(v_2)) \subseteq L_{j+1}.$$

The map λ_t is a map from a single element $V(G_t) = \{z\}$ to the generalized interface sequences $(Q \times Q)^{\leq s} \times \mathcal{P}([1..t])$. By the above induction, we obtain that $\lambda_t(z) = ((q_0, q_f), [1..t]) \in L_t = L_{t+1}$, which is the least fixed point.

For the other direction, assume that $((q_0, q_f), T) \in L_k$ for a $k \leq t$ with $T \subseteq [1..t]$. We may assume that $T = [1..t]$. Otherwise, we delete the non-participating threads from the instance. We show that $L(S) \cap \mathit{Sched}(\Sigma, t, s) \neq \emptyset$. To this end, we construct an execution tree $\mathcal{T}$ together with a labeling function $\lambda : V(\mathcal{T}) \to (Q \times Q)^{\leq s} \times \mathcal{P}([1..t])$ based on the interface sequences that were used to obtain the pair (q_0, q_f).

We start with a single root node r and set $\lambda(r) = ((q_0, q_f), [1..t])$. Now given a partially constructed execution tree, we show how to extend it. We stop the process as soon as for all leaves l of the tree we have $\lambda(l) = (\tau, T)$ with $|T| = 1$. Otherwise, pick a leaf l with $|T| > 1$. By the fixed-point iteration, there are generalized interface sequences (τ_1, T_1) and (τ_2, T_2) such that $(\tau, T) \in (\tau_1, T_1) \otimes^s (\tau_2, T_2)$. Note that (τ_1, T_1) and (τ_2, T_2) are not unique. But we can arbitrarily pick any pair of them. To extend the tree, we add two children l_1 and l_2 of l and set $\lambda(l_i) = (\tau_i, T_i)$ for $i = 1, 2$.

The procedure clearly terminates and yields an execution tree $\mathcal{T}$ where the leaves l satisfy $\lambda(l) \in L_1$. Hence, the leaves show the interface sequences that were used to obtain $((q_0, q_f), [1..t])$ by the fixed-point algorithm.

Now we make use of the tree to construct a word in $L(S)$ the scheduling graph of which has bounded dimension. To this end, we inductively define the map $\Pi : V(\mathcal{T}) \to (Q \times Q)^*$. We start at the leaves. For a leaf l, we set $\Pi(l) = \tau$, where τ is the first component of $\lambda(l)$ and $\lambda(l) = (\tau, \{i\})$. Note that

$\tau \in IF(A_i)$. This means for $\tau = (q_{j_1}, q'_{j_2})(q_{j_2}, q'_{j_3}) \dots (q_{j_\ell}, q'_{j_{\ell+1}})$, there are words $u^j_1, \dots, u^j_\ell$ such that $u^j_1 \dots u^j_\ell \in L(A_i)$ and $u^j_d \in L(M(q_{j_d}, q'_{j_{d+1}}))$, for $d \in [1..\ell]$.

Let l be a node in $\mathcal{T}$ with children l_1 and l_2. Further, let $\lambda(l) = (\tau, T)$, $\Pi(l_1) = \tau'_1$ and $\Pi(l_2) = \tau'_2$. By construction, we know that $\tau \in \tau'_1 \otimes^s \tau'_2$ Hence, there is a $\tau' \in \tau'_1 \text{Ш} \tau'_2$ such that $\tau \in con(\tau')$. We set $\Pi(l) = \tau'$. We stop the procedure if we assigned the root a value under Π.

Now we have that for any node l in $\mathcal{T}$ with

$$\Pi(l) = (q_{i_1}, q'_{i_2})(q_{i_2}, q'_{i_3}) \dots (q_{i_\ell}, q'_{i_{\ell+1}})$$

and $\lambda(l) = (\tau, T)$, there are words $u_1, \dots, u_\ell$ such that $u_1 \dots u_\ell \in \text{Ш}_{i \in T} L(A_i)$. For the root r this means that there is a word w which lies in the intersection $L(M) \cap \text{Ш}_{i \in [1..t]} L(A_i)$. Hence, $w \in L(S)$.

It is left to show that w has a scheduling graph of bounded dimension. To this end, consider the interface sequence associated to r:

$$\Pi(r) = (q_0, q_1)(q_1, q_2) \dots (q_{\ell-1}, q_\ell),$$

with $q_\ell = q_f$. For each tuple (q_j, q_{j+1}), $j \in [1..\ell - 1]$, there is a unique leaf l_j in $\mathcal{T}$ such that (q_j, q_{j+1}) belongs to the interface sequence $\Pi(l_j) = \tau_j$. Let $\lambda(l_j) = (\tau_j, \{c_j\})$. Then we fix the order in which the thread take turns to: $c_0, \dots, c_{\ell-1}$. Note that c_0 is the thread corresponding to (q_0, q_1), c_1 is the thread corresponding to (q_1, q_2) and so on. Clearly, the computation of S reading the word w follows the described order. Hence, we can construct the scheduling graph $G = G(w)$.

To show that $G = (V, E)$ has scheduling dimension bounded by s, we first consider the undirected multigraph $G' = (V, E')$. Recall that we obtain G' by taking all the vertices of G and setting

$$E'(u, v) = \max\{E(u, v), E(v, u)\}$$

for $u, v \in V$. For any leaf l_j of $\mathcal{T}$, we set $\varphi(l_j) = c_j$, where $\lambda(l_j) = (\tau_j, \{c_j\})$. Then $(\mathcal{T}, \varphi)$ is a carving decomposition of G'. We show that the decomposition has width at most s.

Let (n, k) be an edge of $\mathcal{T}$, where n is a child node of k. Moreover, let $\lambda(n) = (\tau, T)$. Removing the edge (n, k) from $\mathcal{T}$ partitions the vertices of G' into T and $V \setminus T$. Note that the number of pairs in τ shows how often T, seen as one thread, participates in the computation of S on w. Hence, we get $E'(T, V \setminus T) \leq |\tau|$. As $|\tau|$ is bounded by s, we get that the width of (n, k) is also bounded by s. Hence, $width(T, \varphi) \leq s$. Finally, by Lemma 6.34, we get that $sdim(G) \leq cw(G') \leq s$. □

B.1.7 Proof of Lemma 6.39

Proof. First, assume that the intersection $L(S) \cap Sched(\Sigma, t, G)$ is non-empty. Then there is a word $w \in L(S)$ such that $G(w) = G$ with its decomposition

into contexts $w = w_1 \dots w_k$. Let $q_j \in Q$ be the state of M reached after reading the prefix $w_1 \dots w_j$. Note that $q_k = q_f$ is the final state. We get an interface sequence α that is induced by the word w. It is of the following form:

$$\alpha = (q_0, q_1).(q_1, q_2) \dots (q_{k-1}, q_k).$$

Let $v \in \bigcup_{\ell \in [1..t]} V(G_\ell)$ be a vertex appearing in the contraction process π. The idea is to assign to each such v an interface sequence σ_v with $\sigma_v \in S_v$.

The sequence σ_v is obtained from α the following way. Let $V(v) \subseteq V$ be those vertices that are contracted to v by π. Note that $V(u) = [1..t] = V$. Then, σ_v is obtained by projecting away those pairs of α that do not stem from a thread in $V(v)$ and then by contracting the remaining pairs. Contracting means that whenever there are two neighboring pairs $(q_i, q_{i+1}).(q_{i+1}, q_{i+2})$ from different threads, we contract it to (q_i, q_{i+2}). This is applied until no more contraction is possible. Note that the definition implies $\sigma_u = (q_0, q_f)$.

By an induction along the contraction process we show that for each vertex v, the sequence σ_v is indeed in S_v. Consequently, we get $\sigma_u \in S_u$.

In the base case, we consider all vertices $v \in V(G_1) = V$. For those vertices, the sequence σ_v obtained from α is exactly the interface sequence contributed by the thread v during the computation of w. Therefore, we obtain that $\sigma_v \in IF(A_v)$. Moreover, note that $|\sigma_v| = \deg(v)$ because $\deg(v)$ is the number of times that thread v contributes during the computation. Since $\deg(v) \leq s$, we obtain that $\sigma_v \in S_v$. Note that this holds as well for initial vertex v_0 and final vertex v_f since α clearly starts in q_0 and ends in q_f.

For the induction step, assume that we consider the ℓ-th contraction of π, namely $G_{\ell+1} = G_\ell[v_1, v_2 \mapsto v']$, and that $\sigma_{v_1} \in S_{v_1}$, $\sigma_{v_2} \in S_{v_2}$. We show that this implies $\sigma_{v'} \in S_{v'}$. To this end, set

$$i = E_\ell(v_1, v_2)$$
$$j = E_\ell(v_2, v_1),$$

where E_ℓ denotes the edge weights of G_ℓ. We can also write the two weights as $i = E(V(v_1), V(v_2))$ and $j = E(V(v_2), V(v_1))$ since $V(v_i)$ is the set of vertices contracted to v_i. This means that in the sequence α, there are i switches from a thread in $V(v_1)$ to a thread in $V(v_2)$ and j switches into the other direction.

We now obtain an interface sequence from α preserving these switches. First, we project away the pairs that do not belong to $V(v') = V(v_1) \cup V(v_2)$. Then, we contract those of the remaining pairs that stem from the threads of $V(v_1)$. Similarly, we contract those that stem from the threads of $V(v_2)$. Note that $V(v_1)$ and $V(v_2)$ are disjoint. Hence, the contractions happen within the corresponding pairs. We obtain a new sequence $\rho \in \sigma_1 \text{Ш} \sigma_2$.

The sequence ρ has exactly i in-contractions and j out-contractions. Indeed, the process for obtaining ρ preserves the corresponding switches between the pairs of α. But this implies that the interface sequence $\sigma_{v'}$ is a member of the set $con_{(i,j)}(\rho)$. Recall that $\sigma_{v'}$ is formed by projecting away all

pairs of α not in $V(v')$ and by contracting the remaining pairs. The same result can be achieved by contracting the i in-contractions and j out-contractions of ρ. Hence, we obtain that

$$\sigma_{v'} \in \sigma_1 \odot_{(i,j)} \sigma_2 \subseteq S_{v_1} \odot_{(i,j)} S_{v_2} = S_{v'}.$$

For the other direction, we show that $(q_0, q_f) \in S_u$ implies the existence of a word $w \in L(S)$ such that the scheduling graph of w is the given graph G. To this end, we inductively construct interface sequences $\sigma_v \in S_v$ for each v appearing in π. We begin with the last vertex u of the process and define $\sigma_u = (q_0, q_f)$. By assumption we have that $\sigma_u \in S_u$.

For the induction step, let $G_{\ell+1} = G_\ell[v_1, v_2 \mapsto v']$. By induction we assume that we have already constructed $\sigma_{v'}$ such that

$$\sigma_{v'} \in S_{v'} = S_{v_1} \odot_{(i,j)} S_{v_2},$$

where $i = E_\ell(v_1, v_2)$ and $j = E_\ell(v_2, v_1)$. Hence, there are sequences $\sigma_{v_1} \in S_{v_1}$ and $\sigma_{v_2} \in S_{v_2}$ such that $\sigma_{v'} \in \sigma_{v_1} \odot_{(i,j)} \sigma_{v_2}$.

Note that the induction constructs interface sequences $\sigma_v \in S_v = IF(A_v)$ for each thread $v \in V$. The idea is now to consider a suitable shuffle of these sequences in order to obtain the desired word. More precise, we want to construct an *extended* interface sequence $\alpha \in \text{Ш}_{v \in V} \sigma_v$ such that contracting the pairs in α yields the sequence σ_u. *Extended* means that α_u has clearly marked pairs that can be employed in a contraction operator that we define later. The idea requires a second induction along π. Formally, we show that for each vertex v' in π, we can construct an extended interface sequence $\alpha_{v'}$ such that the following three conditions hold:

(1) $\alpha_{v'} \in \text{Ш}_{v \in V(v')} \sigma_v$,

(2) $\mathit{contract}(\alpha_{v'}) = \sigma_{v'}$, and

(3) for each $v, z \in V(v')$ with $v \neq z$ we have: a marked pair of z appears immediately after a marked pair of v exactly $E(v, z)$ many times in $\alpha_{v'}$.

The operation $\mathit{contract}(\alpha_{v'})$ contracts adjacent pairs $(q_k, q_{k+1}).(q_{k+1}, q_{k+2})$ in $\alpha_{v'}$ that are marked. It yields a new marked pair (q_k, q_{k+2}). The contraction is repeated until no more contraction can be done.

The base case of the induction is simple. To each vertex $v \in V$ we assign the above constructed interface sequence σ_v. Set $\alpha_v = \sigma_v$ without marking any positions. It clearly satisfies the three conditions.

For the induction step, consider the ℓ-th contraction of process π, given by $G_{\ell+1} = G_\ell[v_1, v_2 \mapsto v']$. By induction, we have the two interface sequences α_{v_1} and α_{v_2}. Note that $\sigma_{v'} \in \sigma_{v_1} \odot_{(i,j)} \sigma_{v_2}$, where $i = E_\ell(v_1, v_2)$ and $j = E_\ell(v_2, v_1)$. By definition, there is an interface sequence

$$\rho \in \sigma_{v_1} \text{Ш} \sigma_{v_2} = \mathit{contract}(\alpha_{v_1}) \text{ Ш } \mathit{contract}(\alpha_{v_2})$$

such that $\sigma_{v'} \in con_{(i,j)}(\rho)$. Phrased differently $\sigma_{v'}$ is an i-j-contraction of ρ which is itself a shuffle of contractions.

We construct $\alpha_{v'}$ as a shuffle of α_{v_1} and α_{v_2} in such a way that the following holds: (a) marked pairs of α_{v_1} and α_{v_2} are taken over to $\alpha_{v'}$. Between two marked pairs of α_{v_1}, there cannot appear a pair of α_{v_2} and vice versa. (b) when contracting the pairs of $\alpha_{v'}$ belonging to $V(v_1)$ and then the pairs belonging to $V(v_2)$ without crossing, we obtain ρ. Without crossing means that we do not contract adjacent pairs where one belongs to $V(v_1)$ and the other one to $V(v_2)$. (c) $\alpha_{v'}$ contains i out-contractions and j-in contractions that are marked at exactly those positions, where ρ gets contracted to $\sigma_{v'}$. This is done by respecting the schedule G. A marked pair of $v \in V(v_2)$ appears immediately after a marked pair of $z \in V(v_2)$ exactly $E(v_1, v_2)$ many times. Note that $\alpha_{v'}$ exists by the existence of $\sigma_{v'}$ and ρ.

With the constructed sequence $\alpha_{v'}$, the above conditions are satisfied. In fact, we satisfy the first condition since we have:

$$\begin{aligned}\alpha_{v'} \in \alpha_{v_1} \text{Ш} \alpha_{v_2} &\subseteq \left(\text{Ш}_{v \in V(v_1)} \sigma_v\right) \text{ Ш } \left(\text{Ш}_{v \in V(v_2)} \sigma_v\right) \\ &= \text{Ш}_{v \in V(v_1) \cup V(v_2)} \sigma_v \\ &= \text{Ш}_{v \in V(v')} \sigma_v.\end{aligned}$$

The second condition is satisfied since $\alpha_{v'}$ contains i out-contractions and j in-contractions at exactly those positions where ρ gets contracted to $\sigma_{v'}$. Since ρ is a contraction of α_{v_1} and α_{v_2} by (b), we obtain that $\sigma_{v'} = \mathit{contract}(\alpha_{v'})$.

The third condition is also satisfied. Let $v, z \in V(v') = V(v_1) \cup V(v_2)$ with $v \neq z$. If v, z are both in $V(v_1)$, then each marked pair of v immediately followed by a marked pair of z in $\alpha_{v'}$ also appears that way in α_{v_1} since marked pairs cannot be reshuffled due to (a). By induction, such pairs appear exactly $E(v, z)$ many times. The case $v, z \in V(v_2)$ is similar. Moreover, if v and z are not both in $V(v_i)$, it follows from the construction of $\alpha_{v'}$.

Finally, we obtain the sequence $\alpha = \alpha_u$ that contracts to $\sigma_u = (q_0, q_f)$. Since α is a shuffle of the $\sigma_v \in IF(A_v)$ and valid, it is an induced interface sequence and therefore witnesses the existence of a word $w \in L(S)$ by Lemma 6.15. Each context of w refers to a pair in the sequence α. Hence, when we construct the scheduling graph $G(w)$, we get G since (3) holds for α. □

B.1.8 Complexity of Round Robin

For the complexity estimation of the algorithm for Round Rob, it is left to show that applying Algorithm 6.2 to the constructed instance $(S, G(S, cs), \pi)$ takes time at most $O(m^{4cs} \cdot cs \cdot t)$. We provide a corresponding lemma.

Lemma B.6. *Let $S = (\Sigma, M, (A_i)_{i \in [1..t]})$ be an SMCP and $cs \in \mathbb{N}$. Applying Algorithm 6.2 to $(S, G(S, cs), \pi)$ takes time at most $O(m^{4cs} \cdot cs \cdot t)$.*

Proof. The most expensive part of Algorithm 6.2 is the computation of the directed contraction products. Let $\ell \in [1..t-2]$ and $G_{\ell+1} = G_\ell[v_1, v_2 \mapsto v']$ the corresponding contraction in π. We need to compute the set $S_{v'} = S_{v_1} \odot_{(i,j)} S_{v_2}$, where $i = E_\ell(v_1, v_2)$ and $j = E_\ell(v_2, v_1)$.

Due to the definition of π, we have that $v_2 = \ell + 1$ and v_1 is the contraction of all vertices $1, \ldots, \ell$ since we contract one vertex after another. This implies that $E_\ell(v_1, v_2) = cs$ and $E_\ell(v_2, v_1) = 0$. For the latter, it is important that $v_2 = \ell + 1 < t$. Hence, we need to compute the product

$$S_{v'} = S_{v_1} \odot_{(cs,0)} S_{v_2}.$$

Since $\deg(v_1) = \deg(v_2) = cs$, we have $S_{v_1}, S_{v_2} \subseteq (Q \times Q)^{cs}$. For computing $S_{v'}$ we need to evaluate the directed contraction product $\odot_{(cs,0)}$ of

$$m^{2cs} \cdot m^{2cs} = m^{4cs}$$

many pairs of sequences. The product of two sequences $\sigma, \tau \in (Q \times Q)^{cs}$ can be computed in linear time $O(cs)$. Note that for any $\rho' \in \sigma \odot_{(cs,0)} \tau$, there is a sequence $\rho \in \sigma \amalg \tau$ such that $\rho' \in con_{(cs,0)}(\rho)$. Phrased differently, this means that ρ' is obtained from ρ by contracting exactly cs many out-contractions. Since σ and τ both have length cs, ρ has to be the unique sequence where the pairs of σ and τ alternatively take turns. Hence, $\sigma \odot_{(cs,0)} \tau$ either contains one element, namely the contraction of ρ which can be computed in time $O(cs)$, or the product is empty, if ρ does not contain cs many out-contractions. Summing up, we can compute $S_{v'}$ in time $O^*(m^{4cs} \cdot cs)$.

For the last contraction in π, $G_t = G_{t-1}[v_1, v_2 \mapsto v']$, we have

$$S_{v'} = (S_{v_1} \odot_{(cs,cs-1)} S_{v_2}).$$

The reasoning is as before. Note that $v_2 = t$ and v_1 is the contraction of all vertices $1, \ldots, t-1$. This implies $E_{t-1}(v_1, v_2) = cs$ and $E_{t-1}(v_2, v_1) = cs - 1$. The product $\sigma \odot_{(cs,cs-1)} \tau$ for two sequences $\sigma, \tau \in (Q \times Q)^{cs}$ can be computed in time $O(cs)$ similar to the above.

In total, Algorithm 6.2 requires $O(t)$ computations of the product. Each of these takes time $O(m^{4cs} \cdot cs)$. Note that we also know the initial vertex 1 and the final vertex t. This completes the complexity estimation. □

B.1.9 Proof of Theorem 6.44

Proof. Let the edges of G be denoted by E. We formally define the underlying alphabet Σ to be the following union:

$$\Sigma = V \cup \{(i, \#) \mid i \in [1..k]\} \cup \{e((i', j'), (i, j)) \mid \{(i', j'), (i, j)\} \in E\}.$$

We construct the threads A_i, $i \in [1..k]$ for each row i of G. The automaton has $k + 1$ states, $p_0, \ldots, p_k$, and the following transitions. To pick and store a

vertex from the i-th row, we have

$$p_0 \xrightarrow{(i,j)} p_j$$

for each $j \in [1..k]$. For performing a trivial context we have the transitions

$$p_j \xrightarrow{(i,\#)} p_j.$$

And for enumerating edges containing the stored vertex, we have

$$p_j \xrightarrow{e((i',j')(i,j))} p_j,$$

if there is an edge $\{(i', j'), (i, j)\}$ with $i' < i$.

We construct the memory automaton M along the two mentioned phases. In the first phase, M synchronizes with each thread A_i on a letter (i, j) with $j \in [1..k]$. This amounts to picking the j-th vertex from row i. To this end, M has exactly $k + 1$ states $\{q_0, \dots q_k\}$ and the transitions

$$q_{i-1} \xrightarrow{(i,j)} q_i,$$

for all $i, j \in [1..k]$. Thus, in the first phase M reads a word of the form $(1, j_1) \dots (k, j_k)$ which constitutes a clique candidate. Note that the contribution of thread A_i to the word is (i, j_i). The thread stores the vertex.

In the second phase, M performs exactly $k - 1$ verification rounds. Again, we describe the i-th round. The memory has states

$$\{(p_1^i, \bot), \dots, (p_i^i, \bot)\}$$

for performing trivial contexts and states

$$\{(p_{i'}^i, j) \mid i' \in [i + 1..k], j \in [1..k]\}$$

for enumerating edges. The last state in round i is given by $(p_{k+1}^i, \bot)$. Further, we set $q_{k+1} = (p_1^1, \bot)$ and $(p_{k+1}^{i-1}, \bot) = (p_1^i, \bot)$ for $i \in [2..k - 1]$ to connect the different rounds. The final state of M is the last state in round $k-1$: $(p_{k+1}^{k-1}, \bot)$.

In round i, we get the following transitions. To perform the trivial contexts along with the threads A_ℓ where $\ell < i$, we obtain

$$(p_\ell^i, \bot) \xrightarrow{(\ell,\#)} (p_{\ell+1}^i, \bot).$$

To recall the vertex chosen in row i, we have the transitions

$$(p_i^i, \bot) \xrightarrow{(i,j)} (p_{i+1}^i, j)$$

for all $j \in [1..k]$. And to enumerate edges involving the recalled vertex, we have for each $\ell \in [i+1..k-1]$ and $j, j' \in [1..k]$ the transition

$$(p^i_\ell, j) \xrightarrow{e((i,j)(\ell,j'))} (p^i_{\ell+1}, j)$$

The last transitions in round i are given by the transitions that enumerate the last edge involving (i, j). We have:

$$(p^i_k, j) \xrightarrow{e((i,j)(k,j'))} (p^i_{k+1}, \bot)$$

for each $j, j' \in [1..k]$. This completes the construction of M and the shared-memory concurrent program $S = (\Sigma, M, (A_i)_{i\in[1..k]})$. Note that a computation of S always follows the cyclic schedule $(1.2 \dots k)^k$. Hence, each word in $L(S)$ will automatically lie in the set $Cyclic(\Sigma, k, k)$. Therefore, for the correctness we need to show that G contains a clique of size k with a vertex from each row if and only if $L(S) \neq \emptyset$.

Assume that $(1, j_1), \dots, (k, j_k)$ is a clique in G. We construct a word in the language $L(S)$. In the first phase, M and the threads accept the word

$$w_c = (1, j_1) \dots (k, j_k).$$

The memory chooses the correct vertices in each row and the threads store it accordingly. Now the second phase begins and M computes in rounds. In the i-th round, M and the threads synchronize on the word

$$w_i = (1, \#). \dots (i-1, \#). e((i, j_i)(i+1, j_{i+1})) \dots e((i, j)(k, j_k)).$$

The threads A_ℓ store the vertices (ℓ, j_ℓ). Hence, they can read the symbols $e((i, j_i)(\ell, j_\ell))$ since there is an edge $\{(i, j_i), (\ell, j_\ell)\} \in E$.

After computing for $k-1$ rounds, we have enumerated all the edges and get that the word $w = w_c.w_1 \dots w_{k-1}$ is in $L(S)$.

If a word $w \in L(S)$ is given, we can split the word into the according phases and rounds: $w = w_c.w_1 \dots w_{k-1}$. The word w_c is of the form

$$w_c = (1, j_1) \dots (k, j_k)$$

and chooses a clique candidate $(1, j_1), \dots, (k, j_k)$. The words $w_1, \dots, w_{k-1}$ ensure that there are enough edges among the chosen vertices. As above, they are of the particular form

$$w_i = (1, \#). \dots (i-1, \#). e((i, j_i)(i+1, j_{i+1})) \dots e((i, j)(k, j_k)).$$

Hence, they can only be read if the corresponding edges exist. We get that the chosen candidate is indeed a clique.

Note that the size of the memory is $O(k^3)$ and that each thread performs exactly $cs = k$ many context. This completes the proof. □

B.1.10 Proof of Theorem 6.54

Proof. We formally define the automaton $P(S) = (C, OP(D), \Delta, (c_0, 0, 0), F)$. The set of states is given by

$$C = Conf(S) \times [0..t] \times [0..k].$$

Hence, $P(S)$ stores the current configuration, the currently writing thread $i \in [1..t]$, and the number of stages $\ell \in [0..k]$. Final stats are given by

$$F = \{(c, i, \ell) \mid c \in Fin(S), i \in [0..t], \ell \in [0..k]\}.$$

The initial state is $(c_0, 0, 0)$, where c_0 is the initial configuration. The transition relation Δ is defined as follows. If $\rightarrow_S$ has a transition of the form $c \xrightarrow{?a/\varepsilon}_S c'$ where thread A_i is reading or performing an interior ε-transition, Δ contains

$$(c, i, \ell) \xrightarrow{?a} (c', i, \ell).$$

If $\rightarrow_S$ has a transition $c \xrightarrow{!a}_S c'$ where thread $A_{i'}$ is writing, then Δ contains

$$(c, i, \ell) \xrightarrow{?a} (c', i', \ell'),$$

where the integer ℓ' is defined by

$$\ell' = \begin{cases} \ell, \text{ if } i' = i, \\ \ell + 1, \text{ otherwise.} \end{cases}$$

We clearly have $L(P(S)) = L(S) \cap Stage(D, t, k)$. Hence, correctness is guaranteed. Note that $P(S)$ has at most

$$\left(\Pi_{i \in [1..t]} |P_i|\right) \cdot d \cdot (t+1) \cdot (k+1) \leq p^t \cdot d \cdot (t+1) \cdot (k+1)$$

many states and $|\Delta| \leq p^{2t} \cdot d^2 \cdot (t+1)^2 \cdot (k+1)^2$ many transitions. Hence, constructing the automaton $P(S)$ takes time at most $O(|\Delta|)$ and checking it for non-emptiness can be done in the same time as well. Consequently, the problem BSR can be solved in time

$$O(p^{2t} \cdot d^2 \cdot (t+1)^2 \cdot (k+1)^2) = O(p^{2t} \cdot d^2 \cdot t^2 \cdot k^2).$$

The estimation completes the proof. □

B.1.11 Proof of Theorem 6.55

Proof. For the formal construction of S, we define for each $i \in [1..k]$, the thread $P_i = (Q_i, OP(D) \times \{i\}, \delta_i, q_i^0, Q_i^f)$, where

$$Q_i = \{q_{ij} \mid j \in [1..k]\} \cup \{q_{ij}^\ell \mid j, \ell \in [1..k]\} \cup \{q_i^0\}.$$

The states q_{ij} are used to store one of the k vertices of the i-th row, the states q_{ij}^{ℓ} are needed to check the edge relations to vertices from other rows. The final states are $Q_i^f = \{q_{ij}^k \mid j \in [1..k]\}$. Note that we can easily reduce to the case of a single final state via introducing ε-transitions.

The transition relation δ_i is given by the following rules. Note that we omit the thread identifiers. For choosing and storing a vertex of the i-th row we have the transitions

$$q_i^0 \xrightarrow{?a_0} q_{ij}$$

for any $j \in [1..k]$. For checking the adjacency to vertices from a different row, we have for each $\ell, j \in [1..k]$ the transition

$$q_{ij}^{\ell-1} \xrightarrow{?(\ell,j')} q_{ij}^{\ell}$$

if $\ell \neq i$ and if there is an edge between (i, j) and (ℓ, j') in G. Note that we identify q_{ij} as q_{ij}^0. Finally, to test the equivalence in the case of the same row:

$$q_{ij}^{i-1} \xrightarrow{?(i,j)} q_{ij}^{i}$$

for each column $j \in [1..k]$.

The writing thread P is given by $P = (Q, OP(D) \times \{P\}, \delta_P, p_0, p_f)$, where $Q = \{p_0, \ldots, p_k\}$, $p_f = p_k$, and δ_P is given by the transitions

$$q_{i-1} \xrightarrow{!(i,j)} q_i$$

for each $i, j \in [1..k]$. Altogether, we define the RW-SMCP S to be the tuple $S = (D, a_0, ((P_i)_{i\in[1..k]}, P))$. We clearly have $t = k + 1$ and $p = O(k^2)$.

It is left to prove the correctness. First note that any word in $L(S)$ has one stage since the only writing thread is P. We show that $L(S) \neq \emptyset$ if and only if there is a clique of size k in G with one vertex from each row.

Let such a clique in G be given. It consists of the vertices $(1, j_1), \ldots, (k, j_k)$. Then it is simple to construct a corresponding word in $L(S)$. First, each P_i guesses the vertex (i, j_i) by reading a_0. Since each two vertices in the clique are adjacent, the thread P can write $(i', j_{i'})$ to the memory and the P_i will read it without deadlocking. Hence, the following is a word in $L(S)$:

$$(?a_0)^k \,.\, !(1, j_1) \,.\, (?(1, j_1))^k \ \ldots\, !(k, j_k) \,.\, (?(k, j_k))^k.$$

For the other direction, let a word $w \in L(S)$ be given. Since each P_i arrives at a final state $q_{ij_i}^k$ upon reading w, we have that w is of the form:

$$w = (?a_0)^k \,.\, !(1, j_1) \,.\, (?(1, j_1))^k \ \ldots\, !(k, j_k) \,.\, (?(k, j_k))^k.$$

The above form yields the clique candidate $(1, j_1), \ldots, (k, j_k) \in V$. Moreover, its implies that two vertices (i, j_i) and $(i', j_{i'})$ are indeed adjacent. To see this note that, once $(i', j_{i'})$ is written to the memory by P, the thread P_i reads the vertex. This can only happen if there is an edge connecting both vertices in the graph. Hence, G contains the desired clique. □

B.1.12 Proof of Theorem 6.56

Proof. We prove the correctness of the construction. Let

$$S = (D, a_0, (P_{x_i})_{i \in [1..n]}, P_w, (P_b)_{b \in [1..\log(I)]})$$

be the constructed RW-SMCP. Since $L(S)$ contains only words with one stage, we show that $L(S) \neq \emptyset$ if and only if there is an $\ell \in [1..I]$ such that φ_ℓ is satisfiable. To this end, first assume we are given a word $w \in L(S)$. Since each thread of S reaches its final state, the word sis of the following form:

$$w = (?a_0)^n . w_1 \ldots w_m,$$

where $w_j = !(\ell_j, j, i_j, v_j).(?(\ell_j, j, i_j, v_j))^{n+\log(I)}$.

In fact, the initial part $(?a_0)^n$ forces the P_{x_i} to store a valuation of the variables by choosing a branch. Each tuple (ℓ_j, j, i_j, v_j) written by P_w to the memory encodes a clause $C_j^{\ell_j}$. By construction, the tuple can only be read by $P_{x_{i_j}}$ if the clause $C_j^{\ell_j}$ is satisfied by x_{i_j} evaluating to v_j and if v_j is the valuation stored by $P_{x_{i_j}}$. Threads P_{x_i} with $i \neq i_j$ can read it the tuple without further requirements. But this means that the encoded clauses $C_j^{\ell_j}$ with $j \in [1..m]$ are all satisfied by the valuation stored in the P_{x_i}.

Since also the bit checkers P_b read the tuples and each of them is accepting in the end, we obtain that the b-th bits of ℓ_1 up to ℓ_m coincide. Hence, we obtain that $\ell_1 = \cdots = \ell_m$. Thus, the clauses were chosen from a single instance φ_{ℓ_1} which is prove to be satisfiable.

For the other direction, assume that φ_ℓ is satisfiable. It is simple to construct a word in $L(S)$. First, the P_{x_i} choose the valuation that satisfies the formula, by reading $(?a_0)^n$. Then, P_w writes the tuples (ℓ, j, i_j, v_j) to the memory, where x_{i_j} is the variable that satisfies clause C_j^ℓ with valuation v_j. By construction, each P_{x_i} and the P_b can read the tuple. Hence, we get the word $w = (?a_0)^n . w_1 \ldots w_m$, where $w_j = !(\ell, j, i_j, v_j).(?(\ell, j, i_j, v_j))^{n+\log(I)}$. □

B.1.13 Proof of Lemma 6.58

Proof. We prove that φ is satisfiable if and only if $L(S) \neq \emptyset$. Note that this shows the correctness of the construction as $L(S)$ only consists of 1-stage words. Assume there is a satisfying assignment v for φ. We construct a word

in $L(S)$ mimicking the assignment. First, we let the threads P_{x_i} choose the correct valuation according to v. To this end, they all read the initial memory value a_0. We set $w_0 = (?a_0)^n$ to be the first part of our word.

Then, P_v starts its computation. It picks a literal ℓ of the first clause C_1 of φ which satisfies it under v. Then it writes $\text{enc}(\ell)$ to the memory, taking turns with the threads P_{x_i} reading the symbols. Note that each P_{x_i} can read the whole string $\text{enc}(\ell)$ since it either involves a different variable or matches the stored valuation v. We get the word w_1, obtained from interleaving sending the symbols of $\text{enc}(\ell)$, immediately followed by receiving the symbols.

The process is repeated for each clause of φ. We obtain the words $w_2, \ldots, w_m$. Note that each P_{x_i} has read exactly m encodings of literals. Hence, they accept. This means that $w = w_0.w_1 \ldots w_m \in L(S)$.

For the other direction, assume that $w \in L(S)$. By construction, w is of the form $w = w_0.w_1 \ldots w_m$, where $w_0 = (?a_0)^n$ and w_i is the word obtained from sending and receiving the symbols of an encoding $\text{enc}(\ell)$, where ℓ is a literal of the j-th clause of φ. Since all P_{x_i} accept, the literal matches the valuation stored by these threads. Phrased differently, the stored valuation satisfies the clause. Altogether we get that the valuation satisfies φ. □

B.2 Proofs for Chapter 7

B.2.1 Proof of Lemma 7.11

Proof. We fix some notation that we use throughout the proof. Let $c \in Q^k$ be a configuration and $s \in Set(c)$. By $Pos_c(s) = \{i \in [1..k] \mid c(i) = s\}$ we denote the components of c storing the state s. Given a second configuration $d \in Q^k$ with k components, we use the set

$$Target_c(s, d) = \{d(i) \mid i \in Pos_c(s)\}$$

to capture those states that occur in d at the components of $Pos_c(s)$. Intuitively, if there is a sequence of transitions from c to d, these are the target states of the components of c that store s.

Consider a computation $\sigma = c \to_N^+ c$ with $Set(c) = \{s_1, \ldots, s_k\}$. We show that there is a cycle $(\{s_1\}, \ldots, \{s_k\}) \to_{\mathcal{G}}^+ (\{s_1\}, \ldots, \{s_k\})$ in the abstraction graph. To this end, assume σ is of the form

$$\sigma = c \to c_1 \to \cdots \to c_\ell \to c.$$

Since $c \to c_1$ is a transition in the broadcast network N, there is an edge

$$(\{s_1\}, \ldots, \{s_k\}) \to_{\mathcal{G}} (Target_c(s_1, c_1), \ldots, Target_c(s_k, c_1))$$

in $\mathcal{G}$ where each state s_i gets replaced by the set of target states in c_1. Applying this argument inductively yields a path in the abstraction graph:

$$\begin{aligned}(\{s_1\},\dots,\{s_m\}) &\to_{\mathcal{G}} (\mathit{Target}_c(s_1,c_1),\dots,\mathit{Target}_c(s_k,c_1))\\ &\to_{\mathcal{G}} (\mathit{Target}_c(s_1,c_2),\dots,\mathit{Target}_c(s_k,c_2))\\ &\to_{\mathcal{G}} \dots\\ &\to_{\mathcal{G}} (\mathit{Target}_c(s_1,c),\dots,\mathit{Target}_c(s_k,c)).\end{aligned}$$

Since $\mathit{Target}_c(s_i,c) = \{s_i\}$, we found the desired cycle.

For the other direction, let a cycle $\rho = (\{s_1\},\dots,\{s_k\}) \to_{\mathcal{G}}^{+} (\{s_1\},\dots,\{s_k\})$ be given. From ρ, we construct a computation $\sigma = c \to^{+} c$ in the broadcast network such that $\mathit{Set}(c) = \{s_1,\dots,s_k\}$. The difficulty in constructing σ is to ensure that at any point in time there are enough clients in the appropriate states. For instance, if a transition $s \xrightarrow{?a} s'$ occurs, we need to decide how many clients have to move to s'. Having too few clients in s' may stall the computation at a later point: there might be a number of sends required that can only be obtained by transitions from s' and if there are too few clients in s', we cannot guarantee the sends. The solution is to start with a rather large number of clients in any state. With invariants we guarantee that at any point in time, the number of clients in the needed states suffices.

Let cycle ρ be $V_0 \to_{\mathcal{G}} V_1 \to_{\mathcal{G}} \cdots \to_{\mathcal{G}} V_\ell$ with $V_0 = V_\ell = (\{s_1\},\dots,\{s_k\})$. Furthermore, let $V_j = (S_j^1,\dots,S_j^k)$. We will construct the computation σ over configurations in Q^n where $n = k \cdot p^\ell$. The idea is to have p^ℓ many clients for each of the k components of the vertices V_i occurring in ρ. To access the clients belonging to a particular component, we split up configurations in Q^n into *blocks*, intervals of integers that are defined for each $i \in [1..k]$ by

$$I(i) = [(i-1)\cdot|Q|^\ell + 1 \,..\, i\cdot|Q|^\ell].$$

Let $d \in Q^n$ be an arbitrary configuration. For $i \in [1..k]$, let the set $B_d(i) = \{d(t) \mid t \in I(i)\}$ capture all states occurring in the i-th block of d. Moreover, we collect all clients in a particular block that currently admit some state $s \in Q$. Let the set $\mathit{Pos}_d(i,s) = \{t \in I(i) \mid d(t) = s\}$ be those components of d in the i-th block that store state s.

We fix the configuration $c \in Q^n$. For each component $i \in [1..k]$, it contains p^ℓ copies of the state s_i in the i-th block. Formally, $B_c(i) = \{s_i\}$. Our goal is to construct the computation $\sigma = c_0 \to^{+} c_1 \to^{+} \cdots \to^{+} c_\ell$ with $c_0 = c_\ell = c$ such that the following two invariants are satisfied.

(1) For each $j \in [0..\ell]$ and $i \in [1..k]$, we have $B_{c_j}(i) \subseteq S_j^i$.

(2) For any state s in a set S_j^i we have $|\mathit{Pos}_{c_j}(i,s)| \geq |Q|^{\ell-j}$.

s Intuitively, (1) means that during the computation σ, we visit at most those states that occur in the cycle ρ. Invariant (2) guarantees that at each configuration c_j, there are *enough* clients available in these states.

We construct σ inductively. The base case is given by configuration $c_0 = c$ which satisfies Invariants (1) and (2) by definition. For the induction step, assume c_j is already constructed and Invariants (1) and (2) hold for the configuration. Our first goal is to construct a configuration d such that $c_j \to^+ d$ and d satisfies (2). In a second step we construct a computation $d \to^* c_{j+1}$.

In the cycle ρ there is an edge $V_j \to_{\mathcal{G}} V_{j+1}$. From the definition of $\to_{\mathcal{G}}$, we get a component $t \in [1..k]$, states $s \in S^t_j$ and $s' \in S^t_{j+1}$, and a message $a \in D$ such that there is a send transition $s \xrightarrow{!a} s'$. Moreover, there are sets $Gen_t \subseteq succ_{?a}(S^t_j)$ and $Kill_t \subseteq enabled_{?a}(S^t_j)$ such that

$$S^t_{j+1} = (U_t \setminus Kill_t) \cup Gen_t \cup \{s'\}.$$

Here, U_t is either S^t_j or $S^t_j \setminus \{s\}$. We focus on t and take care of other components later. We apply a case distinction for the states in S^t_{j+1}.

Let q be a state in $S^t_{j+1} \setminus \{s'\}$. If $q \in Gen_t$, there exists a $p \in S^t_j$ such that $p \xrightarrow{?a} q$. We apply this transition to $|Q|^{\ell-(j+1)}$ many clients in the t-th block of configuration c_j. If $q \in U_t \setminus Kill_t$ and q not in Gen_t, then certainly $q \in U_t \subseteq S^t_j$. In this case, we let $|Q|^{\ell-(j+1)}$ many clients of block t stay idle in q. For state s', we apply a sequence of sends. More precisely, we apply the transition $s \xrightarrow{!a} s'$ to $|Q|^{\ell-(j+1)}$ many clients in block t of c_j. The first of these sends synchronizes with the previously described receive transitions. The other sends do not have any receivers. For components different from t, we apply the same procedure. Since there are only receive transitions, we also let them synchronize with the first send of a. This leads to a computation τ

$$c_j \xrightarrow{a} d^1 \xrightarrow{a} d^2 \xrightarrow{a} \ldots \xrightarrow{a} d^{|Q|^{\ell-(j+1)}} = d.$$

We argue that τ is *valid*: there are enough clients in c_j such that τ can be performed. Focus on component t, the reasoning for other components is similar. Let $q \in B_{c_j}(t) = S^t_j$. Note that the equality is due to Invariants (1) and (2). We count the number of clients of c_j in state q (in block t) that are needed to perform the computation τ. In fact, we need at most

$$|Q|^{\ell-(j+1)} \cdot |succ_{?a}(q) \cup \{q, s'\}| \leq |Q|^{\ell-(j+1)} \cdot |Q| = |Q|^{\ell-j}$$

many of these clients. The set $succ_{?a}(q) \cup \{q, s'\}$ appears as a consequence of the case distinction above: there may be transitions mapping q to a state in $succ_{?a}(q)$, it may happen that clients stay idle in q, and in the case $q = s$, we need to add s' for the send transitions. Since $|Pos_{c_j}(t, q)| \geq |Q|^{\ell-j}$ by Invariant (2), we get that τ is a valid computation. Moreover, note that configuration d satisfies Invariant (2) for $j + 1$: for each state $q \in S^t_{j+1}$, the computation τ was constructed such that $|Pos_d(t, q)| \geq |Q|^{\ell-(j+1)}$.

To satisfy Invariant (1), we need to erase states that are present in d but not in S^t_{j+1}. To this end, we reconsider the set $Kill_t \subseteq enabled_{?a}(S^t_j)$. For each state

$p \in \mathit{Kill}_t$, we know by the definition of $\to_{\mathcal{G}}$ that $\mathit{succ}_{?a}(p) \cap \mathit{Gen}_t \neq \emptyset$. Hence, there is a $q \in S^t_{j+1}$ such that $p \xrightarrow{?a} q$. We apply this transition to all clients in d currently in state p that were not active in the computation τ. In the case $U_t = S^t_j \setminus \{s\}$, we apply the send $s \xrightarrow{!a} s'$ to all clients that are still in s and were not active in τ. Altogether, this leads to a computation $\eta = d \to^* c_{j+1}$.

There is a subtlety in the definition of η. There may be no send transition for the receivers to synchronize with since s may not need to be erased. In this case, we synchronize the receive transitions of η with the last send of τ.

Computation η substitutes the states in Kill_t and state s, depending on U_t, by states in S^t_{j+1}. But this means that in the t-th block of c_{j+1}, there are only states of S^t_{j+1} left. Hence, $B_{c_{j+1}}(t) \subseteq S^t_{j+1}$, and Invariant (1) holds.

After the construction of $\sigma = c \to^+ c_\ell$, it is left to argue that $c_\ell = c$. But this is due to the fact that (1) holds for c_ℓ and $S^t_\ell = (\{s_1\}, \ldots, \{s_m\})$. □

B.2.2 Proof of Lemma 7.25

Before we turn to the proof of the lemma, we need some notation that captures the contribution of an individual client to a transition or computation.

Definition B.7. Let N be a broadcast network and $c \xrightarrow{a} c'$ a transition between two configurations. Let $i \in \mathit{idx}(c \xrightarrow{a} c')$. The *contribution of client* $i \in \mathbb{N}$, denoted by $\mathit{ctr}_i(c \xrightarrow{a} c')$, is the transition that client i internally takes. Formally,

$$\mathit{ctr}_i(c \xrightarrow{a} c') = c(i) \xrightarrow{!a/?a} c'(i),$$

depending on whether i is the sender or a receiver. For $i \notin \mathit{idx}(c \xrightarrow{a} c')$, there is no contribution. Consequently, we set $\mathit{ctr}_i(c \xrightarrow{a} c') = \varepsilon$.

Note that the definition can be extended to computations by appending the individual contributions to the transitions of the computation.

Proof. First note that by construction, a prefix $c_0 \to^* c$ exists in N if and only if $\tilde{c}_0 \to^* \tilde{c}$ exists in N_{Q_F}. Since one can always move from configuration $\tilde{c}$ to c in N_{Q_F} by appending $!n$-transitions for each client that no other client receives, we obtain that $c_0 \to^* c$ in N implies a prefix $\tilde{c}_0 \to^* c$ of N_{Q_F}. Vice versa, a prefix $\tilde{c}_0 \to^* c$ in N_{Q_F} can be shortened to a prefix $\tilde{c}_0 \to^* \tilde{c}$ by removing the $!n$-transitions. This corresponds to $c_0 \to^* c$ in N.

It is left to show that there is a good cycle $c \Rightarrow_{Q_F} c$ if and only if there is a cycle $c \to^+ c$ in N_{Q_F}. We first show that good cycles entail cycles in the instrumentation N_{Q_F}. To this end, we prove a slightly more general result. If there is a computation $\sigma = c \xrightarrow{w}_N d$, we can derive a computation $c \xrightarrow{w'}_{N_{Q_F}} d(\sigma)$ where w' is a word over D' such that its projection to D is w: we have $\pi_D(w') = w$. The configuration $d(\sigma)$ is defined componentwise. Assume $d \in Q^k$ and let $i \in [1..k]$. Moreover, let $d(i) = q$, then the i-th component of configuration $d(\sigma)$ is defined via a case distinction:

(1) $d(\sigma)(i) = q$ if $ctr_i(\sigma) = \varepsilon$. So if client i does not contribute to σ, it will not change its state. Note that $c(i) = q$ in this case.

(2) $d(\sigma)(i) = \tilde{q}$ if $ctr_i(\sigma) = c(i) \to^+ e(i) \to^* d(i) = q$ with $e(i) \in Q_F$. Here, e is some intermediary configuration visited along σ. Hence, client i visits a final state during the computation, leading it into $\tilde{Q}$ in N_{Q_F}.

(3) $d(\sigma)(i) = \hat{q}$ otherwise. In this case, client i contributes to computation σ but does not visit a final state along it.

We construct the computation $c \xrightarrow{w'}_{N_{Q_F}} d(\sigma)$ by induction. In the case $w = \varepsilon$, we set $w' = \varepsilon$ and obtain $d(\sigma) = d$. Hence, we get the desired (empty) computation in the instrumentation N_{Q_F}.

Now let $w = u.a$ for a word $u \in D^*$ and $a \in D$. Then the computation $\sigma = c \xrightarrow{w}_N d$ splits into two parts. There is an intermediary configuration f with $\sigma = \sigma_1.\sigma_2$ where $\sigma_1 = c \xrightarrow{u}_N f$ and $\sigma_2 = f \xrightarrow{a}_N d$. We apply the induction to σ_1 and obtain a computation $\sigma_1' = c \xrightarrow{u'}_{N_{Q_F}} f(\sigma_1)$. Let $J = idx(\sigma_2)$ be the clients contributing to transition σ_2. We extend the computation σ_1' for each $j \in J$ as follows. Let $f(j) = q \in Q$ and $d(j) = p \in Q$ and let $ctr_j(\sigma_2) = q \xrightarrow{op} p$ with $op \in OP(D)$. We distinguish three cases.

(1) If $ctr_j(\sigma_1) = \varepsilon$, then by definition $f(\sigma_1)(j) = q$. Hence, by construction of N_{Q_F}, there is a transition $q \xrightarrow{op} \hat{p}$ which we perform in the j-th component.

(2) If $ctr_j(\sigma_1)$ visits an intermediary final state of Q_F, then $f(\sigma_1)(j) = \tilde{q}$. In this case, we extend σ_1' by the transition $\tilde{q} \xrightarrow{op} \tilde{p}$ in the j-th component.

(3) If $ctr_j(\sigma_1) \neq \varepsilon$ and does not visit a final state, we get that $f(\sigma_1)(j) = \hat{q}$. Then we extend σ_1 by the transition $\hat{q} \xrightarrow{op} \hat{p}$ in the j-th component.

Clients $j \notin idx(\sigma_2)$ do not change their current state during the extension of σ_1'. Altogether, we obtain a computation of the form $c \xrightarrow{u'}_{N_{Q_F}} f(\sigma_1) \xrightarrow{a}_{N_{Q_F}} g$, where g is the resulting configuration of the above extension. We extend the computation such that it reaches $d(\sigma)$. To this end, let $j \in [1..k]$. If $g(j) = \hat{p}$ and $p \in Q_F$, we add the transition $\hat{p} \xrightarrow{!n} \tilde{p}$ without any receiving clients in N_{Q_F}. By this, we get that all clients visiting a final state along σ, finally move to phase $\tilde{Q}$. Hence, we obtain the following desired computation:

$$c \xrightarrow{u'}_{N_{Q_F}} f(\sigma_1) \xrightarrow{a}_{N_{Q_F}} g \xrightarrow{n}_{N_{Q_F}} \cdots \xrightarrow{n}_{N_{Q_F}} d(\sigma).$$

Now assume a good cycle $\rho = c \Rightarrow_{Q_F} c$ in N is given. We apply the above result and obtain a computation $c \to^+ c(\rho)$ in N_{Q_F}. Since ρ is good for Q_F, each participating client visits a final state along ρ. Hence, the configuration $c(\rho)$ can be described as follows. Let $j \in [1..k]$ and $c(j) = q$. We have

$$c(\rho)(j) = \begin{cases} q, \text{ if } ctr_j(\sigma) = \varepsilon, \\ \tilde{q}, \text{ otherwise.} \end{cases}$$

Note that there is no state in phase $\hat{Q}$.

We extend the computation $c \to^+ c(\rho)$ in N_{Q_F} for each $j \in [1..k]$ with $ctr_j(\rho) \neq \varepsilon$. Since $c(\rho)(j) = \tilde{q}$ in this case, we add the transition $\tilde{q} \xrightarrow{!n} q = c(j)$. Hence, we obtain a proper cycle in the instrumentation:

$$c \to^+_{N_{Q_F}} c(\rho) \xrightarrow{!n}_{N_{Q_F}} \cdots \xrightarrow{!n}_{N_{Q_F}} c.$$

For the other direction, assume that a cycle $c \to^+ c$ in N_{Q_F} is given. By construction, such a cycle can be decomposed as follows:

$$c_0 = c \xrightarrow{w_1} d_1 \xrightarrow{n}{}^* c_1 \xrightarrow{w_2} d_2 \xrightarrow{n}{}^* c_2 \to \cdots \to c_{n-1} \xrightarrow{w_n} d_n \xrightarrow{n}{}^* c.$$

Here, each $w_i \in D^*$ and the computation between d_i and c_i is a repeated sending of n without receivers, needed to transition between different phases.

Out of this computation, we construct a good cycle $c \Rightarrow_{Q_F} c$. To this end, we define for each configuration d of N_{Q_F} with k components, a configuration $N(d)$ that retrieves the original states out of d. For each $j \in [1..k]$, we set

$$N(d)(j) = q \text{ if } d(j) = q/\tilde{q}/\hat{q}.$$

By construction of the instrumentation N_{Q_F}, we can mimic the transitions occurring in $c_{i-1} \xrightarrow{w_i}_{N_{Q_F}} d_i$ by transitions on Q and hence infer a corresponding computation on the original broadcast network of the form $N(c_{i-1}) \xrightarrow{w_i}_N N(d_i)$. A computation $d_i \xrightarrow{n^*}_{N_{Q_F}} c_i$ can be ignored on the original network since $N(d_i) = N(c_i)$. The reason is that sending n only shifts phases within N_{Q_F}. Altogether, we obtain a cycle in N:

$$c = N(c_0) \xrightarrow{w_1} N(d_1) \xrightarrow{w_2} N(d_2) \xrightarrow{w_3} \cdots \xrightarrow{w_n} N(d_n) = c.$$

The obtained cycle in N is good. Let $j \in [1..k]$ be a client participating in the cycle. Then, $j \in idx(c \to^+_{N_{Q_F}} c)$. This means that j starts in a state $q \in Q$ and reaches back to q. By construction of N_{Q_F}, this is only possible if j visits a state $\hat{p}$ with $p \in Q_F$ during the cycle. Hence, p is also visited during the obtained cycle on the original network N. □

B.3 Proofs for Chapter 8

B.3.1 Proof of Lemma 8.20

Proof. First assume that we are given an initialized computation $\rho = c^0 \to^*_S c$ with $\pi_L(c) = q$. We prove the existence of a witness $x = (w, q, \sigma)$ and a first-write sequence β such that $\text{Valid}_\beta(x)$. To this end, let $\rho = \rho_1 \dots \rho_m$ be the transitions. These can be leader and contributor transitions.

Let $\rho_L = \rho_{l_1} \dots \rho_{l_n}$ be the projection of ρ to transitions of the leader. Out of it, we construct the word w. The projection ρ_L takes the following form:

$$q_L^0 = q_0 \underbrace{\xrightarrow{x_0}_L}_{\rho_{l_1}} q_1 \underbrace{\xrightarrow{x_1}_L}_{\rho_{l_2}} \cdots \underbrace{\xrightarrow{x_{n-1}}_L}_{\rho_{l_n}} q_n = q,$$

where $x_i = !a_i/\varepsilon/?a_i$ for some symbol $a_i \in D$. We set the target state of x to be $q = q_n$ and the word $w = (q_0, y_0).(q_1, y_1) \dots (q_{n-1}, y_{n-1})$ with

$$y_i = \begin{cases} \perp, \text{ if } x_i = \varepsilon/?a_i, \\ a_i, \text{ otherwise.} \end{cases}$$

To determine σ and the first-write sequence β, we project ρ to its first-write transitions. For each $a \in D$, this is the unique transition (if it exists) of a contributor, where a is written for the first time. Let the resulting projection be $\rho_C = \rho_{f_1} \dots \rho_{f_k}$ where ρ_{f_i} is the first write of a symbol β_i.

Clearly, the desired first write sequence is then defined by $\beta = \beta_1 \dots \beta_k$. It is left to define the map $\sigma : [1..k] \to [1..n]$. Let $i \in [1..k]$. The first write of β_i occurs in ρ in the transition ρ_{f_i}. Let ℓ_i be the number of leader transitions occurring before ρ_{f_i} in ρ. Formally, we set

$$\ell_i = |\{\rho_{l_i} \mid l_i < f_i\}|.$$

Then we can define $\sigma(i) = \ell_i$.

Leader validity of x along β follows immediately from the construction of w along the computation ρ. All required leader transitions exist. The same is true for contributor validity. Since ρ is a proper computation, the first writes $\rho_{f_1}, \dots, \rho_{f_k}$ can all be provided by the contributors. Moreover, they can only read the available writes. We obtain that $\mathrm{LValid}_\beta(x)$ evaluates to true.

For the other direction, assume that there is a witness $x = (w, q, \sigma) \in \mathit{Wit}$ that is valid along a first-write sequence $\beta \in \mathit{FW}$. We show that there is an initialized computation $c^0 \to_S^* c$ with $\pi_L(c) = q$. Let $w = (q_1, a_1) \dots (q_n, a_n)$ and let $\beta = \beta_1 \dots \beta_k$. By assumption, we have that both predicates, $\mathrm{LValid}_\beta(x)$ and $\mathrm{CValid}_\beta(x)$, evaluate to true.

Since $\mathrm{LValid}_\beta(x)$ is true, there exists a sequence of transitions of the leader

$$\rho_L = \rho_1 \dots \rho_n$$

such that $\rho_i = q_i \xrightarrow{a_i}_L q_{i+1}$ if $a_i \neq \perp$. Otherwise, for $a_i = \perp$, we have $q_i = q_{i+1}$, $\rho_i = q_i \xrightarrow{\varepsilon}_L q_{i+1}$, or $\rho_i = q_i \xrightarrow{?b}_L q_{i+1}$ for some $b \in S_\beta(i)$. Note that we also represent the case $q_i = q_{i+1}$ by a transition $\rho_i = q_i \xrightarrow{\varepsilon}_L q_{i+1}$.

The predicate $\mathrm{CValid}_\beta(x)$ also evaluates to true. Fix $i \in [1..k]$. We use the notation $\beta^{<i} = \beta_1 \dots \beta_{i-1}$. For each $i \in [1..k]$ we have

$$\mathit{LAW}(x, \beta^{<i}) \cap \pi_{RD}(\mathit{Trace}_C(Q_i)) \neq \emptyset.$$

Recall that $LAW(x, \beta^{<i}) = \Gamma_1^*\{a_1, \varepsilon\}\Gamma_2^*\{a_2, \varepsilon\} \dots \Gamma_{\sigma(i)}^*$ is the language of available writes and $\Gamma_j = Loop(q_j, S_{\beta^{<i}}(j)) \cup S_{\beta^{<i}}(j)$. Fix a string r_i in the intersection $LAW(x, \beta^{<i}) \cap \pi_{RD}(Trace_C(Q_i))$. These are the reads that a contributor performs to reach Q_i. Moreover, let $w_i \in Trace_C(Q_i)$ such that $\pi_{RD}(w_i) = r_i$ and τ_i the sequence of transitions of the contributor belonging to w_i:

$$\tau_i = p_C^0 \xrightarrow{w_i} p \in Q_i.$$

Note that the reads in τ_i are along $LAW(x, \beta^{<i})$.

Now we define the map γ_i, which maps each letter of r_i (each read of w_i/τ_i) to the exact position where it occurs along the language $LAW(x, \beta^{<i})$. We set $\gamma_i : [1..|r_i|] \to [1..\sigma(i)]$, with $\gamma_i(j) = \ell$ if $r_i(j) \in \Gamma_\ell \cup \{a_\ell\}$. Moreover, we assign one of three *types* to each symbol of r_i. In fact, a symbol is either a read of a leader write a_i (*ld*), a read of a first write (*fw*), or a read which is due to a loop of the leader (*lp*). We define $\lambda_i : [1..|r_i|] \to \{ld, lp, fw\}$, where

$$\lambda_i(j) = \begin{cases} ld, \text{ if } \gamma_i(j) = \ell \wedge r_i(j) = a_\ell, \\ lp, \text{ if } \gamma_i(j) = \ell \wedge r_i(j) \in Loop(q_\ell, S_{\beta<i}(\ell)), \\ fw, \text{ otherwise.} \end{cases}$$

For each j with $\gamma_i(j) = \ell$ and $\lambda_i(j) = lp$, let $lp(i, j)$ be the sequence of transitions of the leader P_L that forms a loop over q_ℓ which writes the symbol $r_i(j)$. Hence, $lp(i, j)$ is a sequence of the form

$$q_\ell \xrightarrow{u.!r_i(j).v} q_\ell,$$

with $\pi_{RD}(u.!r_i(j).v) \in S_{\beta^{<i}}(\ell)$.

We will now add all the required loop transitions to the transition sequence ρ_L of the leader inductively. The resulting run of the leader will be denoted by ρ'_L. Fix $i \in [1..k]$. For each $j \in [1..|r_i|]$, we add $lp(i, j)$ immediately before the transition ρ_ℓ, where $\ell = \gamma_i(j)$. Then we go on with the next value for i. Note that we fill the space between two transitions of the leader from the right — with increasing i. Intuitively, we add the required loops in the correct place so that they can provide the symbols that the contributors want to read.

Now we estimate the number of required contributors to make the computation possible. To this end, we need the number of read transitions in ρ'_L. For any $d \in D$, let $\#_d(\rho'_L)$ be the number of transitions of ρ'_L that read the value d from the memory. Moreover, let $\#_d(r_i)$ be the number of times that d occurs as a read of a first write in r_i:

$$\#_d(r_i) = |\{j \mid r_i(j) = d \wedge \lambda_i(j) = fw\}|.$$

We set $\#_d = \#_d(\rho'_L) + \sum_{i=1}^{k} \#_d(r_i)$. This is the number of contributors that we need to provide the symbol d during the desired computation. Note that

$\#_d$ is only non-zero if d occurs as a first write in the sequence β. If $d$does not occur as a first write, it can neither be read by the leader nor is it counted within the definition of the number $\#_d(r_i)$.

Now we formally construct the required computation ρ in the leader contributor system. We start with a configuration c^0, consisting of the leader in its initial state q_L^0, the initial memory value a^0 and for each $d \in \{\beta_1, \ldots, \beta_k\}$, we have $\#_d$ many contributors in their initial state p_C^0. We will refer to the set contributors that want to provide d collectively as $[d]$. The intention is that one of them moves and the others just copy the taken transition. Hence, they behave similarly. To construct ρ, we need a pointer into the run ρ'_L of the leader and into the words $r_1, \ldots, r_k$. To this end, we use the following tuple:

$$idx = (idx(\rho'_L), idx(r_1), \ldots, idx(r_k)) \in [1..|\rho'_L|] \times [1..|r_1|] \times \cdots \times [1..|r_k|],$$

which stores a position into each. We use idx to formulate and maintain invariants along the constructed computation. Initially, we have $idx = (1, \ldots, 1)$.

We construct ρ while maintaining three invariants.

(1) We always have a suffice amount of contributors waiting to provide $d \in D$. For each occurring tuple $idx = (idx(\rho'_L), idx(r_1), \ldots, idx(r_k))$, we have at least $\#_d(idx)$ many contributors on their way to provide d. Here,

$$\#_d(idx) = \#_d(\rho'_L[idx(\rho'_L)..]) + \#_d(r_1[idx(r_1)..]) + \ldots + \#_d(r_k[idx(r_k)..]),$$

where $\alpha[idx...]$ indicates the subsequence or substring of α starting from the index idx.

(2) Let $idx = (idx(\rho'_L), idx(r_1), \ldots, idx(r_k))$ be an occurring tuple and the number of taken leader transitions ρ_j be ℓ. The latter means that

$$|\{\rho_j \mid \rho_j \in \rho'_L[1..idx(\rho'_L)]\}| = \ell.$$

Then, we have that for each i with $\sigma(i) \leq \ell$ that $idx(r_i) = |r_i|$. This means that once a first-write position is arrived, the corresponding contributors providing the symbol are indeed available.

(3) Finally, for each occurring $idx = (idx(\rho'_L), idx(r_1), \ldots, idx(r_k))$, the leader in ρ is in the target state of the transition $\rho'_L(idx(\rho'_L))$.

We begin with the inductive construction. We have already seen the base case. Hence, assume that the computation ρ has been constructed partially and $idx = (idx(\rho'_L), idx(r_1), \ldots, idx(r_k))$ is the current tuple up to which we have processed ρ'_L and the strings $r_1, \ldots r_k$. For each $i \in [1..k]$, we first perform any possible transition of τ_i, starting from the last transition that was executed in this sequence. We distinguish two cases.

If $\lambda_i(idx(r_i)) = fw$ and the read symbol is $r_i(idx(r_i)) = \beta_j$ for some $j \leq i$ with $\sigma(j) \leq \ell$. Here, $\ell = |\{\rho_j \mid \rho_j \in \rho'_L[1..idx(\rho'_L)]\}|$. By Invariant (1) and

Invariant (2), we obtain that there are $\#_{\beta_j}(idx)$ many contributors in a state waiting to provide β_j. We use one of these to provide the symbol and let it write β_j to the memory. All contributors in $[\beta_i]$ read the symbol and execute the corresponding read transition of τ_i. We increment the index $\#_{\beta_j}(r_i)$. Hence, $\#_{\beta_j}(idx)$ decreases by one and all invariants are maintained.

Now let $\lambda_i(idx(r_i)) = lp$ with $\gamma_i(idx(r_i)) = \ell - 1$ and the read symbol be $r_i(idx(r_i)) = b \in D$. The integer ℓ is defined as above. Then there is a loop $lp(i, idx(r_i))$ of the leader on $q_{\ell-1}$ which provides the symbol b. The leader is in state $q_{\ell-1}$ by Invariant (3). Hence, it can execute the required loop and write the symbol b to the memory. All contributors in $[b_i]$ execute the corresponding read transition. Then, the leader finishes the loop. Note that executing $lp(i, idx(r_i))$ may require contributors that write needed symbols. By our invariants, we can ensure that there are enough contributors to feed the leader during the loop. We update idx by moving $idx(\rho'_L)$ to the end of the loop and we increment $idx(r_i)$ by 1.

Finally we process the leader. If the current transition $\rho'_L(idx(\rho'_L)$ is a read of the contributor, then we move one contributor to write the corresponding value to memory and make the leader move. We also increment $idx(\rho'_L)$ by 1. If it is an interior ε transition, we make the leader move and again increment $idx(\rho'_L)$ by 1. If the move of the leader is a write, we let the leader write the symbol and again increase $idx(\rho'_L)$. But in this case, we need to move contributors that read the written symbol. For each $i \in [1..k]$ such that $\lambda_i(idx(r_i)) = ld$ and $\gamma_i(idx(r_i)) = \ell$, ($\ell$ defined as above), we execute the corresponding read transition from τ_i which reads the value written by the leader. We update $idx(r_i)$ accordingly.

The constructed computation satisfies the three invariants and is therefore a valid computation of the system. This completes the proof. □

B.3.2 Proof of Lemma 8.26

Before we turn to the proof of Lemma 8.26, we show some auxiliary statements that are needed to simplify the proof. The first lemma states that leader validity of a witness is preserved under repeatedly applying *Shrink*.

Lemma B.8. *Let $\beta \in FW$ be a first-write sequence and $x \in Wit$ a (long) witness with* $\mathrm{LValid}_\beta(x) = true$. *Then, we have that* $\mathrm{LValid}_\beta(Shrink^*(x)) = true$.

Proof. We show that $\mathrm{LValid}_\beta(Shrink(x)) = true$. Then, the above statement follows by induction. To this end, assume x is given by (w, q, σ) with $w = (q_1, a_1) \dots (q_n, a_n)$. If $Shrink(x) = x$, there is nothing to show. Otherwise, there are indices $r < t$ such that $Shrink(x) = (w', q, \sigma')$ where

$$w' = (q_1, a_1) \dots (q_{r-1}, a_{r-1}).(q_t, a_t) \dots (q_n, a_n)$$

The map σ' of the witness is defined by $\sigma'(\ell) = \sigma(\ell)$ if $\sigma(\ell) < r$, $\sigma'(\ell) = r$ if $r \leq \sigma(\ell) \leq t$, and $\sigma'(\ell) = \sigma(\ell) - t + r$ for $\sigma(\ell) > t$.

For proving leader validity, let a_i be a symbol in w'. Since a_i also occurs in w and $\mathrm{LValid}_\beta(x) = \mathit{true}$, we get one of the following.

(1) There is a write transition $q_i \xrightarrow{!a_i}_L q_{i+1}$,

(2) $q_i = q_{i+1}$,

(3) there is an ε-transition $q_i \xrightarrow{\varepsilon}_L q_{i+1}$, or

(4) there is a read transition $q_i \xrightarrow{?b}_L q_{i+1}$ with $b \in S_\beta(i) = \{\beta_\ell \mid \sigma(\ell) \leq i\}$.

For Cases (1) and (3), note that write and ε-transitions carry over from x to $\mathit{Shrink}(x)$. The only subtlety occurs when $i = r-1$. Validity of x guarantees a transition $q_{r-1} \xrightarrow{!a_{r-1}/\varepsilon}_L q_r$. But $q_r = q_t$. Hence, we also obtain the needed transition for the witness $\mathit{Shrink}(x)$.

In Case (2), we get that $q_i = q_{i+1}$. Since the operator *Shrink* cuts out the first occurrence of a repeating state, Case (2) can only happen when $i \geq t$. Then, the equality of states is also true in $\mathit{Shrink}(x)$.

In the last case, we have to show that the read transition carries over to $\mathit{Shrink}(x)$. Essentially, we need to prove that the index shift that occurs when passing from w to w' is consistent with the sets $S_\beta(i)$ and

$$S'_\beta(i) = \{\beta_\ell \mid \sigma'(\ell) \leq i\}$$

This means that the read symbol b has to lie in the corresponding set $S'_\beta(i')$. To this end, we make precise the relations among the sets $S_\beta(j)$ and $S'_\beta(j)$ for each index $j \in [1..n]$.

If $j \in [1..r-1]$, we immediately obtain

$$S'_\beta(j) = \{\beta_\ell \mid \sigma'(\ell) \leq j\} = \{\beta_\ell \mid \sigma(\ell) \leq j\} = S_\beta(j)$$

from the definition of σ'. Hence, the sets are equal for $j < r$.

For $j \in [r..t]$, first note that $S_\beta(j) \subseteq S_\beta(t)$ since these sets grow monotonically. The latter set can be rewritten as

$$\begin{aligned} S_\beta(t) = \{\beta_\ell \mid \sigma(\ell) \leq t\} &= \{\beta_\ell \mid \sigma(\ell) < r \text{ or } r \leq \sigma(\ell) \leq t\} \\ &= \{\beta_\ell \mid \sigma'(\ell) < r \text{ or } \sigma'(\ell) = r\} \\ &= \{\beta_\ell \mid \sigma'(\ell) \leq r\} \\ &= S'_\beta(r). \end{aligned}$$

Hence, $S_\beta(j) \subseteq S'_\beta(r)$ for each $j \in [r..t]$.

In the last case, j is an index in $[t+1..n]$. Consider the following following:

$$\begin{aligned} S_\beta(j) &= \{\beta_\ell \mid \sigma(\ell) \leq j\} \\ &= \{\beta_\ell \mid \sigma(\ell) \leq t \text{ or } t < \sigma(\ell) \leq j\} \\ &= S_\beta(t) \cup \{\beta_\ell \mid t < \sigma(\ell) \leq j\} \\ &= S'_\beta(r) \cup \{\beta_\ell \mid t < \sigma(\ell) \leq j\}. \end{aligned}$$

Note that in the last step we used that $S_\beta(t) = S'_\beta(r)$. Now we find an equivalent description for the latter set in the union. For an index ℓ with $\sigma(\ell) > t$, we get by definition that $\sigma'(\ell) = \sigma(\ell) - t + r$. Hence, we have that $t < \sigma(\ell) \leq j$ if and only if $r < \sigma'(\ell) \leq j - t + r$. We can derive:

$$\begin{aligned} S'_\beta(r) \cup \{\beta_\ell \mid t < \sigma(\ell) \leq j\} =& S'_\beta(r) \cup \{\beta_\ell \mid r < \sigma'(\ell) \leq j - t + r\} \\ =& S'_\beta(j - t + r). \end{aligned}$$

Hence, $S_\beta(j) = S'_\beta(j - t + r)$.

Assume, from Case (4) we get a transition $q_i \xrightarrow{?b}_L q_{i+1}$ with $b \in S_\beta(i)$. If $i \in [1..r-1]$, we obtain by the above discussion that $b \in S_\beta(i) = S'_\beta(i)$. If $i = t$, we obtain that $b \in S_\beta(t) = S'_\beta(r)$. In the last case, $i \in [t+1..n]$, we get that $b \in S_\beta(i) = S'_\beta(i - t + r)$. This proves leader validity of $\mathit{Shrink}(x)$. □

The following lemma extends the results from Lemma B.8. It shows that shrinking operator , leader validity, and witness concatenation behave well with respect to each other. Moreover, it provides a way to replace a witness in a concatenation as long as leader validity is guaranteed.

Lemma B.9. *Let $x = (w, q, \sigma)$ be a witness of order k and y a witness of order p with $\mathit{init}(y) = q$. Moreover, let $\beta = \beta_1 \dots \beta_{k+p}$ be a first-write sequence and the predicate $\mathrm{LValid}_\beta(x \times y) = \mathit{true}$.*

a) We have $\mathrm{LValid}_\beta(x \times \mathit{Shrink}^*(y)) = \mathit{true}$.

b) Let $x' = (w', q, \sigma')$ be a witness of order k and let $\beta' = \beta_1 \dots \beta_k$ be the prefix of β of length k. If $\mathrm{LValid}_{\beta'}(x') = \mathit{true}$, then $\mathrm{LValid}_\beta(x' \times y) = \mathit{true}$.

Proof. We first prove Part a). To this end, we fix some notation that is used throughout the proof. Let y be the tuple (v, p, τ). Assume the words w and v of the witnesses x and y are given by:

$$\begin{aligned} w &= (q_1, a_1) \dots (q_m, a_m) \\ v &= (q_{m+1}, a_{m+1}) \dots (q_n, a_n) \end{aligned}$$

with $q_{k+1} = q$. Then, for the concatenation we get $x \times y = (w.v, p, \sigma.\tau)$. Recall that $\sigma.\tau$ maps the first writes as depicted in the definition of the concatenation: $\sigma.\tau(\ell) = \sigma(\ell)$ for $\ell \in [1..k]$ and $\sigma.\tau(\ell) = \tau(\ell - k) + m$ for $\ell \in [k+1..k+p]$.

When applying the shrink operator to y, we get that $\mathit{Shrink}(y) = (v', p, \tau')$. Assume that $\mathit{Shrink}(y) \neq y$, otherwise there is nothing to prove. Then, there are indices $r < t$ such that $q_r = q_t$ and

$$v' = (q_{m+1}, a_{m+1}) \dots (q_{r-1}, a_{r-1}).(q_t, a_t) \dots (q_n, a_n).$$

A concatenation with x therefore yields $x \times \mathit{Shrink}(y) = (w.v', p, \sigma.\tau')$ with

$$w.v' = (q_1, a_1) \dots (q_m, a_m) \dots (q_{r-1}, a_{r-1}).(q_t, a_t) \dots (q_n, a_n).$$

and map $\sigma.\tau'$, defined similarly to $\sigma.\tau$.

Now the reasoning is similar to Lemma B.8. We prove leader validity of $x \times \mathit{Shrink}(y)$. The above claim then follows. First, we obtain relations among the available first writes

$$S_\beta(j) = \{\beta_\ell \mid \sigma.\tau(\ell) \leq j\} \text{ and } S'_\beta(j) = \{\beta_\ell \mid \sigma.\tau'(\ell) \leq j\}.$$

For $j \in [1..r-1]$, we have that $S_\beta(j) = S'_\beta(j)$. For $j \in [r..t]$, we get $S_\beta(j) \subseteq S'_\beta(r)$, and $S_\beta(t) = S'_\beta(r)$. Finally, if $j \in [t+1..n]$, we obtain $S_\beta(j) = S'_\beta(j-t+r)$.

To prove leader validity of $x \times \mathit{Shrink}(y)$, fix a symbol a_i in the concatenation $w.v'$. Since $\mathrm{LValid}_\beta(x \times y) = \mathit{true}$, there are four cases:

(1) There is a write transition $q_i \xrightarrow{!a_i}_L q_{i+1}$. This transition immediately carries over to the witness $x \times \mathit{Shrink}(y)$.

(2) The states q_i and q_{i+1} are equal. The equality is also true in $x \times \mathit{Shrink}(y)$.

(3) There is an ε-transition $q_i \xrightarrow{\varepsilon}_L q_{i+1}$ which also carries over.

(4) There is a read transition $q_i \xrightarrow{?b}_L q_{i+1}$ with $b \in S_\beta(i)$. By the above considerations, b lies in the suitable set of first writes of $x \times \mathit{Shrink}(y)$.

For the proof of Part b), we adjust the above notation. Again, let the witness $x = (w, q, \sigma)$ contain the word

$$w = (q_1, a_1) \dots (q_m, a_m).$$

Let $y = (v, p, \tau)$, where $v = (s_1, b_1) \dots (s_n, b_n)$ and $x' = (w', q, \sigma')$ with

$$w' = (p_1, c_1) \dots (p_t, c_t).$$

We consider $x \times y = (w.v, p, \sigma.\tau)$ and $x' \times y = (w'.v, p, \sigma'.\tau)$ with words

$$w.v = (q_1, a_1) \dots (q_m, a_m).(s_1, b_1) \dots (s_n, b_n),$$
$$w'.v = (p_1, c_1) \dots (p_t, c_t).(s_1, b_1) \dots (s_n, b_n),$$

and maps

$$\sigma.\tau(\ell) = \begin{cases} \sigma(\ell), \text{ if } \ell \in [1..k], \\ \tau(\ell), \text{ if } \ell \in [k+1..k+p], \end{cases} \quad \sigma'.\tau(\ell) = \begin{cases} \sigma'(\ell), \text{ if } \ell \in [1..k], \\ \tau(\ell), \text{ if } \ell \in [k+1..k+p]. \end{cases}$$

To prove leader validity of $x' \times y$, pick a symbol in the word $w'.v$. Assume it is c_i for an $i \in [1..t]$. By the assumption $\mathrm{LValid}_{\beta'}(x') = \mathit{true}$, we get that either

there is a transition $p_i \xrightarrow{!c_i/\varepsilon}_L p_{i+1}$ or $p_i = p_{i+1}$ or there is a read transition $p_i \xrightarrow{?b}_L p_{i+1}$ for a $b \in S^{x'}_{\beta'}(i) = \{\beta_\ell \in \beta' \mid \sigma'(\ell) \leq i\}$. The first two cases immediately carry over to $x' \times y$. In the latter case, we need to show that b lies in the correct set $S^{x'\times y}_{\beta}(i) = \{\beta_\ell \mid \sigma'.\tau(\ell) \leq i\}$. Recall that $i \leq t$ and that $\sigma'.\tau(\ell) \leq t$ if and only if $\sigma'.\tau(\ell) = \sigma'(\ell)$ by definition. But this means that $S^{x'}_{\beta'}(i) = S^{x'\times y}_{\beta}(i)$. Note that the discussion also covers $p_{t+1} = q = s_1$.

Assume the picked symbol is a b_i with index $i \in [1..n]$. Since the predicate $\mathrm{LValid}_\beta(x \times y)$ evaluates to *true*, we either get $s_i = s_{i+1}$ or a transition $s_i \xrightarrow{!b_i/?b/\varepsilon}_L s_{i+1}$ where $b \in S^{x\times y}_{\beta}(i+m) = \{\beta_\ell \mid \sigma.\tau(\ell) \leq i+m\}$. Note the index $i+m$ in the set of first writes. The simple cases carry over to $x' \times y$. In the case of a read transition, consider

$$\begin{aligned} S^{x\times y}_{\beta}(i+m) =& \{\beta_\ell \mid \sigma.\tau(\ell) \leq i+m\} \\ =& \{\beta_1, \ldots, \beta_k\} \cup \{\beta_\ell \mid m < \sigma.\tau(\ell) \leq i+m\}. \end{aligned}$$

The last equation holds by the definition of $\sigma.\tau$. Moreover, we have that $\sigma.\tau(\ell) = \tau(\ell-k)+m$ if and only if $\sigma.\tau(\ell) > m$. Similarly $\sigma'.\tau(\ell) = \tau(\ell-k)+t$ if and only if $\sigma'.\tau(\ell) > t$. Hence, we obtain:

$$\begin{aligned} S^{x\times y}_{\beta}(i+m) &= \{\beta_1, \ldots, \beta_k\} \cup \{\beta_\ell \mid m < \tau(\ell-k)+m \leq i+m\} \\ &= \{\beta_1, \ldots, \beta_k\} \cup \{\beta_\ell \mid t < \tau(\ell-k)+t \leq i+t\} \\ &= \{\beta_1, \ldots, \beta_k\} \cup \{\beta_\ell \mid t < \sigma'.\tau(\ell) \leq i+t\} \\ &= \{\beta_\ell \mid \sigma'.\tau(\ell) \leq i+t\} \\ &= S^{x'\times y}_{\beta}(i+t). \end{aligned}$$

This shows that b lies in the correct set $S^{x'\times y}_{\beta}(i+t)$ and completes the proof. □

The previous results can be used to show that short validity always implies leader validity. This constitutes a key step to our main result.

Lemma B.10. *Let z be a short witness of order k and $\beta = \beta_1 \ldots \beta_k$ a fist-write sequence. If $\mathrm{Valid}^{sh}_{\beta}(z) = true$, then we have $\mathrm{LValid}_\beta(z) = true$.*

Proof. We prove the lemma by a case distinction. If $ord(z) = 0$, we get by the definition of short validity that $\beta = \varepsilon$ and $\mathrm{LValid}_\varepsilon(z) = \mathrm{Valid}^{sh}_{\varepsilon}(z) = true$.

If $ord(z) = k+1 > 0$ for a $k < d$ then, by the recursive definition of short validity, there are witnesses $x \in Ord(k)$ and $y \in Ord(1)$ such that $z = x \otimes y$ and $\mathrm{LValid}_\beta(x \times y) = true$. Since $z = Shrink^*(x \times y)$, we get $\mathrm{LValid}_\beta(z) = true$ by an application of Lemma B.8. □

We use languages of available writes $LAW(x, \alpha)$ to make visible the writes that contributors can rely on when providing a particular first write. If all

first writes of a sequence were provided, the language slightly changes due to the availability of all first writes. In this case, we speak of *full expressions*. The definition is as follows.

Definition B.11. Let $x = (w, q, \sigma)$ be a witness with $w = (q_1, a_1) \dots (q_n, a_n)$ and β a first-write sequence with $|\beta| = ord(x)$. The *full expression* of x with respect to β is the regular language

$$FullExpr(x, \beta) = \Gamma_1^*\{a_1, \varepsilon\}\Gamma_2^*\{a_2, \varepsilon\} \dots \Gamma_n^*\{a_n, \varepsilon\},$$

where $\Gamma_i = Loop(q_i, S_\beta(i)) \cup S_\beta(i)$.

The next lemma shows that full expressions are preserved under applying the shrinking operator as long as leader validity is provided.

Lemma B.12. *For an $\beta \in FW$ and a $x \in Wit$ with $\mathrm{LValid}_\beta(x) = true$, we have*

$$FullExpr(x, \beta) = FullExpr(Shrink^*(x), \beta).$$

Proof. We show that the full expressions are invariant under the shrink operator. Formally, $FullExpr(x, \beta) = FullExpr(Shrink(x), \beta)$. Since leader validity is invariant under *Shrink* as well by Lemma B.8, the lemma follows by induction.

Let $x = (w, q, \sigma)$ be the given witness with $w = (q_1, a_1) \dots (q_n, q_n)$. If $Shrink(x) = x$, there is nothing to show. Otherwise, there exist indices $r < t$ with $q_r = q_t$ such that $Shrink(x) = (w', q, \sigma')$ where

$$w' = (q_1, a_1) \dots (q_{r-1}, a_{r-1}).(q_t, a_t) \dots (q_n, a_n).$$

The map σ' is given by $\sigma'(\ell) = \sigma(\ell)$ if $\sigma(\ell) < r$, $\sigma'(\ell) = r$ if $r \le \sigma(\ell) \le t$, and $\sigma'(\ell) = \sigma(\ell) - t + r$ otherwise.

Considering the full expression defined by the witness x, we obtain

$$FullExpr(x, \beta) = \Gamma_1^*.\{a_1, \varepsilon\} \dots \Gamma_r^*.\{a_r, \varepsilon\} \dots \Gamma_t^*.\{a_t, \varepsilon\} \dots \Gamma_n^*.\{a_n, \varepsilon\}$$

where $\Gamma_i = Loop(q_i, S_\beta(i)) \cup S_\beta(i)$. The full expression defined by the shrunk witness $Shrink(x)$ is given by

$$FullExpr(Shrink(x), \beta) = \Sigma_1^*.\{a_1, \varepsilon\} \dots \Sigma_{r-1}^*.\{a_{r-1}, \varepsilon\}.\Sigma_t^*.\{a_t, \varepsilon\} \dots \Sigma_n^*.\{a_n, \varepsilon\}.$$

To describe Σ_i, we use the notation $S'_\beta(i) = \{\beta_\ell \mid \sigma'(\ell) \le i\}$. Then, the sets are given by $\Sigma_i = Loop(q_i, S'_\beta(i)) \cup S'_\beta(i)$ for $i \in [1..r-1]$, $\Sigma_t = Loop(q_t, S'_\beta(r)) \cup S'_\beta(r)$, and for $i \in [t+1..n]$ we have $\Sigma_i = Loop(q_i, S'_\beta(i-t+r)) \cup S'_\beta(i-t+r)$. Note that we need the case distinction for the sets Σ_i due to the index shift that occurs when going from witness x to $Shrink(x)$.

Now we show equality of both full expressions. To this end, we split them into three parts and show equality of each parts separately.

Step 1: We prove the following equation to be correct:

$$\Gamma_1^*.\{a_1, \varepsilon\} \dots \Gamma_{r-1}^*.\{a_{r-1}, \varepsilon\} = \Sigma_1^*.\{a_1, \varepsilon\} \dots \Sigma_{r-1}^*.\{a_{r-1}, \varepsilon\}.$$

It is enough to show that $\Gamma_i = \Sigma_i$ for $i \in [1..r-1]$. In the proof of Lemma B.8, we have seen that $S'_\beta(i) = S_\beta(i)$ for indices $i \in [1..r-1]$. Hence, by definition of Γ_i and Σ_i, we get their equality and hence, the above equation holds.

Step 2: We show the middle parts of the expressions to be equal:

$$\Gamma_r^*.\{a_r, \varepsilon\} \dots \Gamma_t^*.\{a_t, \varepsilon\} = \Sigma_t^*.\{a_t, \varepsilon\}.$$

From the proof of Lemma B.8, we know that $S_\beta(t) = S'_\beta(r)$. Hence, we obtain the equation $\Sigma_t = \mathit{Loop}(q_t, S'_\beta(r)) \cup S'_\beta(r) = \mathit{Loop}(q_t, S_\beta(t)) \cup S_\beta(t) = \Gamma_t$. Taking the equivalence into account and dropping a_t, it is left to show that

$$\Gamma_r^*.\{a_r, \varepsilon\} \dots \Gamma_t^* = \Gamma_t^*.$$

One inclusion is immediate. For the other one, we show that $a_r, \dots, a_{t-1}$ are contained in Γ_t and that $\Gamma_r, \dots, \Gamma_{t-1}$ are actually subsets of Γ_t.

Due to validity of x with respect to the leader, $\mathrm{LValid}_\beta(x) = \mathit{true}$, we get a run ρ on the leader P_L of the form

$$q_t = q_r \xrightarrow{!a_r/\perp}_L q_{r+1} \xrightarrow{!a_{r+1}/\perp}_L \dots \xrightarrow{!a_{t-1}/\perp}_L q_t,$$

where $q_i \xrightarrow{\perp}_L q_{i+1}$ denotes either a read of a symbol $b \in S_\beta(i)$ or an ε-transition. Since $S_\beta(i) \subseteq S_\beta(t)$ for each $i \in [r..t-1]$, all reads along ρ are only from the set $S_\beta(t)$. This means that each a_i with $i \in [r..t-1]$ is either $\perp$ or occurs as a write in a loop of q_t where reads are restricted to the set $S_\beta(t)$. Phrased differently, $a_r, \dots, a_{t-1} \in \mathit{Loop}(q_t, S_\beta(t)) \subseteq \Gamma_t$.

Fix $i \in [r..t-1]$. We show that $\Gamma_i \subseteq \Gamma_t$. To this end, we reconsider the run ρ from above and split it into two parts with middle q_i. We denote by ρ_1 the first part $q_t = q_r \to_L \dots \to_L q_i$. By ρ_2, we denote the latter part $q_i \to_L \dots \to_L q_t$. Let now $b \in \Gamma_i = \mathit{Loop}(q_i, S_\beta(i)) \cup S_\beta(i)$. Then, either $b \in S_\beta(i) \subseteq S_\beta(t) \subseteq \Gamma_t$ or b appears as a write on a loop in q_i where reading is restricted to $S_\beta(i) \subseteq S_\beta(t)$. If b appears as such a write, we can append ρ_1 as prefix and ρ_2 as postfix to the corresponding run. Then, b appears as a write in a loop in q_t while reading is restricted to $S_\beta(t)$. Hence, $b \in \mathit{Loop}(q_t, S_\beta(t)) \subseteq \Gamma_t$.

Step 3: We prove the equivalence of the latter parts of the expressions:

$$\Gamma_{t+1}^*.\{a_{t+1}, \varepsilon\} \dots \Gamma_n^*.\{a_n, \varepsilon\} = \Sigma_{t+1}^*.\{a_{t+1}, \varepsilon\} \dots \Sigma_n^*.\{a_n, \varepsilon\}.$$

It suffices to show that $\Gamma_i = \Sigma_i$ for $i \in [t+1..n]$. To this end, let $i \in [t+1..n]$. Like before, we refer to the proof of Lemma B.8 and obtain $S_\beta(i) = S'_\beta(i-t+r)$. This yields the equation:

$$\Sigma_i = \mathit{Loop}(q_i, S'_\beta(i-t+r)) \cup S'_\beta(i-t+r) = \mathit{Loop}(q_i, S_\beta(i)) \cup S_\beta(i) = \Gamma_i.$$

Altogether, the full expression is preserved under shrinking. □

A further tool that we use in the proof of Lemma 8.26 is the so-called *blow up* of a witness. It allows us to increase the order of a first-order witness.

Definition B.13. Let $x = (w, q, \sigma)$ be a *first-order witness* ($\mathit{ord}(x) = 1$). Moreover, let $k \in \mathbb{N}$ be a natural number such that $k < d$. Then, we extend x to a witness of order $k+1$ by mapping k first writes to the first position and the remaining first write to the position indicated by σ. The $(k+1)$-*blow up* of x is the witness $x^{(k+1)} = (w, q, \sigma^{(k+1)})$ where $\sigma^{(k+1)} : [1..k+1] \to [1..n]$ is given by

$$\sigma^{(k+1)}(i) = \begin{cases} 1, & \text{if } i \in [1..k], \\ \sigma(1), & \text{if } i = k+1. \end{cases}$$

The following lemma shows that languages of available writes and full expressions of witness concatenations can be decomposed. Namely into the full expression of the left factor and the available first writes/the full expression of the blow up of the right factor.

Lemma B.14. *Let x be a witness of order $k < d$ and y a first-order witness. Moreover, let $\beta = \beta_1 \dots \beta_{k+1}$ be a first-write sequence and β' the prefix $\beta_1 \dots \beta_k$. We have:*

a) $\mathit{FullExpr}(x \times y, \beta) = \mathit{FullExpr}(x, \beta').\mathit{FullExpr}(y^{(k+1)}, \beta)$,

b) $\mathit{LAW}(x \times y, \beta') = \mathit{FullExpr}(x, \beta').\mathit{LAW}(y^{(k+1)}, \beta')$.

Proof. We first prove Part a). To this end, let $x = (w, q, \sigma)$ and $y = (v, p, \tau)$ be the given witnesses and

$$w = (q_1, a_1) \dots (q_n, a_n),$$
$$v = (p_1, b_1) \dots (p_m, b_m)$$

with $p_1 = q$. Consider the witness concatenation $x \times y = (w.v, p, \sigma.\tau)$. The full expression of it is given by

$$\mathit{FullExpr}(x \times y, \beta) = \Gamma_1^*.\{a_1, \varepsilon\} \dots \Gamma_n^*.\{a_n, \varepsilon\}.\Sigma_1^*.\{b_1, \varepsilon\} \dots \Sigma_m^*.\{b_m, \varepsilon\}.$$

In this language, we have $\Gamma_i = \mathit{Loop}(q_i, S_\beta^{x\times y}(i)) \cup S_\beta^{x\times y}(i)$ for each $i \in [1..n]$ and similarly $\Sigma_i = \mathit{Loop}(p_i, S_\beta^{x\times y}(i+n)) \cup S_\beta^{x\times y}(i+n)$ for $i \in [1..m]$.

Let $i \in [1..n]$. Then, by definition of $\sigma.\tau$, we obtain the following:

$$S_\beta^{x\times y}(i) = \{\beta_\ell \mid \sigma.\tau(\ell) \leq i\} = \{\beta_\ell \in \beta' \mid \sigma(\ell) \leq i\} = S_{\beta'}^{x}(i).$$

This implies that $\Gamma_i = \mathit{Loop}(q_i, S_{\beta'}^x(i)) \cup S_{\beta'}^x$ and hence we get:

$$\mathit{FullExpr}(x, \beta') = \Gamma_1^*.\{a_1, \varepsilon\} \ldots \Gamma_n^*.\{a_n, \varepsilon\}.$$

It is left to show that $\mathit{FullExpr}(y^{(k+1)}, \beta) = \Sigma_1^*.\{b_1, \varepsilon\} \ldots \Sigma_m^*.\{b_m, \varepsilon\}$. Let the blow up of y be denoted by $y^{(k+1)} = (v, p, \tau^{(k+1)})$. Then, its full expression is given by the following:

$$\mathit{FullExpr}(y^{(k+1)}, \beta) = L_1^*.\{b_1, \varepsilon\} \ldots L_m^*.\{b_m, \varepsilon\},$$

where $L_i^* = \mathit{Loop}(p_i, S_\beta^{(k+1)}(i)) \cup S_\beta^{(k+1)}(i)$ with $S_\beta^{(k+1)}(i) = \{\beta_\ell \mid \tau^{(k+1)}(\ell) \leq i\}$. We show that $L_i = \Sigma_i$ for each $i \in [1..m]$. To this end, it is enough to prove the equality $S_\beta^{(k+1)}(i) = S_\beta^{x\times y}(i+n)$.

By definition, we get the following for $i \in [1..m]$:

$$S_\beta^{(k+1)}(i) = \{\beta_\ell \mid \tau^{(k+1)}(\ell) \leq i\} = \{\beta_1, \ldots, \beta_k\} \cup \begin{cases} \{\beta_{k+1}\}, \text{ if } \tau(1) \leq i, \\ \emptyset, \text{ otherwise.} \end{cases}$$

By definition of the map $\sigma.\tau$, the sets $\{\beta_1, \ldots, \beta_k\}$ and $\{\beta_\ell \mid \sigma.\tau(\ell) \leq n\}$ are equal. Hence, we can rewrite the above expression. Note that $\tau(1) > 0$. We obtain the following:

$$S_\beta^{(k+1)}(i) = \{\beta_\ell \mid \sigma.\tau(\ell) \leq n\} \cup \begin{cases} \{\beta_{k+1}\}, \text{ if } n < \tau((k+1)-k) + n \leq i + n, \\ \emptyset, \text{ otherwise.} \end{cases}$$

Then, by definition it follows

$$\begin{aligned} S_\beta^{(k+1)}(i) &= \{\beta_\ell \mid \sigma.\tau(\ell) \leq n\} \cup \begin{cases} \{\beta_{k+1}\}, \text{ if } n < \sigma.\tau(k+1) \leq i + n, \\ \emptyset, \text{ otherwise} \end{cases} \\ &= \{\beta_\ell \mid \sigma.\tau(\ell) \leq i + n\} \\ &= S_\beta^{x\times y}(i+n). \end{aligned}$$

For Part b), consider the available first writes of $x \times y$

$$\mathit{LAW}(x \times y, \beta') = \Gamma_1^*.\{a_1, \varepsilon\} \ldots \Gamma_n^*.\{a_n, \varepsilon\}.\Sigma_1^*.\{b_1, \varepsilon\} \ldots \Sigma_j^*,$$

where $j + n = \sigma.\tau(k+1)$. Note that this implies $j = \tau(1)$. The sets Γ_i and Σ_i are given by $\Gamma_i = \mathit{Loop}(q_i, S_{\beta'}^{x\times y}(i)) \cup S_{\beta'}^{x\times y}(i)$ for $i \in [1..n]$ and similarly $\Sigma_i = \mathit{Loop}(p_i, S_{\beta'}^{x\times y}(i+n)) \cup S_{\beta'}^{x\times y}(i+n)$ for $i \in [1..j]$. Note that the first writes refer to β', we have $S_{\beta'}^{x\times y}(i) = \{\beta_\ell \in \beta' \mid \sigma.\tau(\ell) \leq i\}$.

Let $i \in [1..n]$. Then we obtain from the definition of $\sigma.\tau$:

$$S^{x\times y}_{\beta'}(i) = \{\beta_\ell \in \beta' \mid \sigma(\ell) \leq i\} = S^{x}_{\beta'}(i).$$

Similarly to Part a, we obtain $\mathit{FullExpr}(x, \beta') = \Gamma^*_1.\{a_1, \varepsilon\} \ldots \Gamma^*_n.\{a_n, \varepsilon\}$.

It is left to show that the equality $\mathit{LAW}(y^{(k+1)}, \beta') = \Sigma^*_1.\{a_1, \varepsilon\} \ldots \Sigma^*_j$ holds. By definition, we obtain

$$\mathit{LAW}(y^{(k+1)}, \beta') = L^*_1.\{b_1, \varepsilon\} \ldots L^*_{j'},$$

where $j' = \tau^{(k+1)}(k+1) = \tau(1) = j$ and $L_i = \mathit{Loop}(p_i, S^{(k+1)}_{\beta'}(i)) \cup S^{(k+1)}_{\beta'}(i)$. Now let $i \in [1..j]$. Since $\tau^{(k+1)}$ maps the first writes $\beta_1, \ldots, \beta_k$ to position 1, we get:

$$S^{(k+1)}_{\beta'}(i) = \{\beta_\ell \in \beta' \mid \tau^{(k+1)}(\ell) \leq i\} = \{\beta_1, \ldots, \beta_k\}.$$

The map $\sigma.\tau$ maps the first writes $\beta_1, \ldots, \beta_k$ to positions smaller than n. Hence, we get

$$S^{x\times y}_{\beta'}(i+n) = \{\beta_\ell \in \beta' \mid \sigma.\tau(\ell) \leq i+n\} = \{\beta_1, \ldots, \beta_k\} = S^{(k+1)}_{\beta'}(i).$$

This implies $L_i = \Sigma_i$ and completes the proof. □

Under certain assumptions, shrinking operator and blow up commute. The next lemma formalizes this observation. The technical assumption that we have to make is that σ maps the (only) first write to the first position in the word of the witness.

Lemma B.15. *Let $x = (w, q, \sigma)$ be a first-order witness with $\sigma(1) = 1$. Moreover, let $y = \mathit{Shrink}^*(x)$. For each $k < d$, we have the equality $\mathit{Shrink}^*(x^{(k+1)}) = y^{(k+1)}$.*

Proof. The witness y is obtained by shrinking x. Hence, we get that y is of the form $y = (w', q, \sigma)$. Note that σ will not change under shrinking since $\sigma(1) = 1$ is its only value. Now consider the blow up of x, $x^{(k+1)} = (w, q, \sigma^{(k+1)})$. Due to the definition of the blow up, $\sigma^{(k+1)}$ is the constant 1-map.

Shrinking $x^{(k+1)}$ will result in a short witness $\mathit{Shrink}^*(x^{(k+1)}) = (w', q, \sigma^{(k+1)})$. Note that the word w' coincides with the word of y. Moreover, $\sigma^{(k+1)}$ is preserved under shrinking since it is the constant 1-map. If we blow up y, we get $y^{(k+1)} = (w', q, \sigma^{(k+1)})$. Hence, we obtain the desired equality. □

Finally, we need a lemma which transforms a witness into a similar witness that separates the last first write. Technically, we need that the first-write map σ is strictly increasing for the last element that it maps. The lemma is key to the induction step in the proof of Lemma 8.26.

Lemma B.16. *Let $x = (w, q, \sigma) \in Wit$ be a witness of order $k+1$ with $k < d$ and β a first-write sequence with $\mathrm{LValid}_\beta(x) \wedge \mathrm{CValid}_\beta(x) = true$. Then, we can construct a witness $\hat{x} = (\hat{w}, q, \hat{\sigma})$ with $init(\hat{x}) = init(x)$ and $\mathrm{LValid}_\beta(\hat{x}) \wedge \mathrm{CValid}_\beta(\hat{x}) = true$ that additionally satisfies*

$$\hat{\sigma}(i) < \hat{\sigma}(k+1) \text{ for each } i \in [1..k].$$

Proof. If x already satisfies $\sigma(i) < \sigma(k+1)$ for any $i \in [1..k]$, we set $\hat{x} = x$. Otherwise, let $\sigma(k+1) = p$. We can write the word w as follows:

$$w = (q_1, a_1) \dots (q_{p-1}, a_{p-1}).(q_p, a_p) \dots (q_n, a_n).$$

The idea in the construction of $\hat{w}$ is to prolong the word w by a copy of q_p so that two different positions in $\hat{w}$ refer to the state. To this end, set

$$\hat{w} = (q_1, a_1) \dots (q_{p-1}, a_{p-1}).(q_p, \bot).(q_p, a_p) \dots (q_n, a_n).$$

The map $\hat{\sigma}$ is defined by $\hat{\sigma}(i) = \sigma(i)$ for $i \in [1..k]$ and $\hat{\sigma}(k+1) = p+1$. Since σ is monotonically increasing, we obtain the desired property $\hat{\sigma}(i) < \hat{\sigma}(k+1)$ from the definition. Moreover, $\hat{x}$ satisfies $init(\hat{x}) = init(x)$. It is left to show that $\hat{x}$ is valid for the leader and the contributors along. β.

For leader validity, we fist compare the sets $S_\beta(j)$, associated to x, with $\hat{S}_\beta(j)$, associated to $\hat{x}$. Since we shift the index in the construction of $\hat{w}$, we will also get an index shift when moving from $S_\beta(j)$ to $\hat{S}_\beta(j)$. We reflect this in a case distinction. For the first case, let $j \in [1..p-1]$. Then we have that

$$\hat{S}_\beta(j) = \{\beta_\ell \mid \hat{\sigma}(\ell) \leq j\} = \{\beta_\ell \mid \sigma(\ell) \leq j\} = S_\beta(j).$$

The equation comes from the fact that $\hat{\sigma}(\ell) = \sigma(\ell)$ if $\sigma(\ell) \leq j$ and $j \leq p-1$.

For the case $j = p$, consider the following equivalence. It follows from $\hat{\sigma}(\ell) \leq p$ for each $\ell \in [1..k]$ and $\sigma(\ell) \leq p$ for any $\ell \in [1..k+1]$.

$$\hat{S}_\beta(p) = \{\beta_1, \dots, \beta_k\} = S_\beta(p) \setminus \{\beta_{k+1}\}.$$

In the last case, let $j \in [p+1..n+1]$. Then, $\hat{\sigma}$ maps all the elements of β to a position that is at most j. We have that $\hat{S}_\beta(j) = \{\beta_1, \dots, \beta_{k+1}\}$. The map σ maps to positions that are strictly smaller than j, $S_\beta(j-1) = \{\beta_1, \dots, \beta_{k+1}\}$. Hence, we obtain $\hat{S}_\beta(j) = S_\beta(j-1)$.

Now we prove leader validity for all positions $j \in [1..n+1]$ along the same case distinction. Let $j \in [1..p-1]$. We have to show that there is a transition $q_i \xrightarrow{!a_i/\varepsilon/?b}_L q_{i+1}$ with $b \in \hat{S}_\beta(j)$ or that $q_i = q_{i+1}$. By the leader validity of x we get that either the states are equal or that there is a transition $q_i \xrightarrow{!a_i/\varepsilon/?b}_L q_{i+1}$ with $b \in S_\beta(j)$. Since $\hat{S}_\beta(j) = S_\beta(j)$, leader validity holds for position j.

Consider the case $j = p$. By the definition of $\hat{w}$ we have that q_p is the state of position p and $p+1$. Hence, the states of the positions coincide and leader validity for position p holds.

For the last case, let $j \in [p+1..n+1]$. By $\mathrm{LValid}_\beta(x) = \mathit{true}$ we either get that $q_{j-1} = q_j$ or we obtain a transition $q_{j-1} \xrightarrow{!a_{j-1}/\varepsilon/?b}_L q_j$ with $b \in S_\beta(j-1) = \hat{S}_\beta(j)$. Hence, leader validity also holds in this case and we get that $\mathrm{LValid}_\beta(\hat{x}) = \mathit{true}$.

Now we prove that $\mathrm{CValid}_\beta(\hat{x}) = \mathit{true}$. To this end, we show that the positions of the first writes within β', a prefix of $\beta_1 \dots \beta_k$, under $\hat{\sigma}$ and σ are the same. Let $j \in [1..p]$. Then

$$\hat{S}_{\beta'}(j) = \{\beta_\ell \in \beta' \mid \hat{\sigma}(\ell) \leq j\} = \{\beta_\ell \in \beta' \mid \sigma(\ell) \leq j\} = S_{\beta'}(j).$$

Note that for the equality, it is important to consider prefixes β' which exclude the first write β_{k+1}. For $j \in [p+1..n+1]$ we have that

$$\hat{S}_{\beta'}(j) = \{\beta_1, \dots, \beta_d\} = S_{\beta'}(j-1)$$

where $\beta_1 \dots \beta_d = \beta'$ denotes the considered prefix.

Now we prove the equivalence of the available writes induced by $x, \hat{x}$ and β. Let $i \in [1..k]$ and $\beta' = \beta_1 \dots \beta_{i-1}$ some prefix of β. If we use the notation $\Sigma_j = \mathit{Loop}(q_j, \hat{S}_{\beta'}(j)) \cup \hat{S}_{\beta'}(j)$ and $\Gamma_j = \mathit{Loop}(q_j, S_{\beta'}(j)) \cup S_{\beta'}(j)$s, we get the following two languages:

$$\mathit{LAW}(\hat{x}, \beta') = \Sigma_1^*.\{a_1, \varepsilon\} \dots \Sigma_{\hat{\sigma}(i)}^*,$$
$$\mathit{LAW}(x, \beta') = \Gamma_1^*.\{a_1, \varepsilon\} \dots \Gamma_{\sigma(i)}^*.$$

Since $i \leq k$, we get that $\hat{\sigma}(i) = \sigma(i)$ and $\hat{\sigma}(i) \leq p$. Thus, $\hat{S}_{\beta'}(j) = S_{\beta'}(j)$ for each $j \in [1..\hat{\sigma}(i)]$. This implies that $\Sigma_j = \Gamma_j$ and that the languages are equal.

For $i = k+1$, the first-write sequence of interest is $\beta' = \beta_1 \dots \beta_k$. In this case, the languages are of the form

$$\mathit{LAW}(\hat{x}, \beta') = \Sigma_1^*.\{a_1, \varepsilon\} \dots \Sigma_p^*.\{\varepsilon\}.\Sigma_{p+1}^*,$$
$$\mathit{LAW}(x, \beta') = \Gamma_1^*.\{a_1, \varepsilon\} \dots \Gamma_p^*.$$

For $j \leq p$, we get that $\hat{S}_{\beta'}(j) = S_{\beta'}(j)$ by our earlier consideration. If $j = p+1$, we obtain $\hat{S}_{\beta'}(p+1) = S_{\beta'}(p)$. Hence, we get that $\Sigma_j = \Gamma_j$ for all $j \in [1..p]$ and $\Sigma_{p+1} = \Gamma_p$. Then the expressions again coincide.

Since $\mathrm{CValid}_\beta(x) = \bigwedge_{i \in [1..k+1]} \mathrm{CValid}_\beta^i(x) = \mathit{true}$, we get that for each index $i \in [1..k+1]$, the intersection $\mathit{LAW}(x, \beta') \cap h(\mathit{Trace}_C(Q_i))$ is non-empty, where $\beta' = \beta_1 \dots \beta_{i-1}$. Now we can replace $\mathit{LAW}(x, \beta')$ by $\mathit{LAW}(\hat{x}, \beta')$ in each intersection and obtain that $\mathrm{CValid}_\beta^i(\hat{x}) = \mathit{true}$ for each $i \in [1..k+1]$ which implies that the predicate $\mathrm{CValid}_\beta(\hat{x})$ evaluates to true. □

Finally, we turn to the proof of Lemma 8.26.

Proof. Fix a first-write sequence β. We show the two directions of the lemma.

First Direction: Let a witness $x = (w, q, \sigma) \in \mathit{Wit}$ be given with both predicates, $\mathrm{LValid}_\beta(x)$ and $\mathrm{CValid}_\beta(x)$, evaluating to true. By induction on the order of x, we prove a statement slightly stronger than depicted in the lemma. We show that there is a short witness $z = (w', q, \sigma')$ with $\mathit{init}(z) = \mathit{init}(x)$, $\mathit{ord}(z) = \mathit{ord}(x)$, $\mathit{FullExpr}(z, \beta) = \mathit{FullExpr}(x, \beta)$, and $\mathrm{Valid}^{sh}_\beta(z) = \mathit{true}$.

For the induction basis, consider the case $\mathit{ord}(x) = 0$. Then, $\beta = \varepsilon$. We set $z = \mathit{Shrink}^*(x)$. Note that shrinking preserves initial state, target state, and order. Hence, the short witness z is of the form (w', q, σ') and satisfies: $\mathit{init}(z) = \mathit{init}(x)$ and $\mathit{ord}(z) = 0$. Recall that in this case, validity of z is defined by $\mathrm{Valid}^{sh}_\varepsilon(z) = \mathrm{LValid}_\varepsilon(z)$. Hence, we need to show leader validity of z along β. Since $\mathrm{LValid}_\varepsilon(x) = \mathit{true}$ by assumption, we obtain from Lemma B.8 that $\mathrm{LValid}_\varepsilon(z) = \mathit{true}$. It is left to show that the full expressions of z and x coincide. But this follows immediately from Lemma B.12.

Now assume that $\mathit{ord}(x) = k{+}1$ for a $k \in \mathbb{N}$ with $k < d$. Then, $\beta = \beta_1 \dots \beta_{k+1}$. We denote the prefix $\beta_1 \dots \beta_k$ of the first-write sequence by β'. Let $\sigma(k{+}1) = p$. Then, we can write the word w as

$$w = (q_1, a_1) \dots (q_{p-1}, a_{p-1}).(q_p, a_p) \dots (q_n, a_n).$$

By Lemma B.16, we can assume that $\sigma(i) < p$ for each $i \in [1..k]$. We define the word $w_{pre} = (q_1, a_1) \dots (q_{p-1}, a_{p-1})$ to be the prefix of w up to the $(p-1)$-st letter. The remaining postfix is the denoted by $w_{po} = (q_p, a_p) \dots (q_n, a_n)$. Moreover, we define the map σ_{pre} to be the restriction of σ to $[1..k]$. Formally, $\sigma_{pre} : [1..k] \to [1..p-1]$ with $\sigma_{pre}(i) = \sigma(i)$. We further define σ_{po} to map a single first write to the first position 1, $\sigma_{po}(1) = 1$. Intuitively, σ_{po} is the map responsible for the last first write β_{k+1}. With these definitions we can split the witness x into the following two witnesses

$$x_{pre} = (w_{pre}, q_p, \sigma_{pre}) \text{ and } x_{po} = (w_{po}, q, \sigma_{po}).$$

By definition, we get that $x = x_{pre} \times x_{po}$. Moreover, the orders are given by $\mathit{ord}(x_{pre}) = k$ and $\mathit{ord}(x_{po}) = 1$. We want to apply the induction hypothesis to x_{pre}. To this end, we need to show that $\mathrm{LValid}_{\beta'}(x_{pre}) \wedge \mathrm{CValid}_{\beta'}(x_{pre}) = \mathit{true}$.

For the leader validity, we use the fact that $\mathrm{LValid}_\beta(x) = \mathit{true}$. Fix an index $j \in [1..p-1]$. By the leader validity of x, either $q_j = q_{j+1}$ or there exists a transition of the form

$$q_j \xrightarrow{!a_j/\varepsilon/?b}_L q_{j+1}$$

with $b \in S_\beta(j)$. For the set $S_\beta(j)$, we have the following equivalence:

$$\begin{aligned} S_\beta(j) &= \{\beta_\ell \in \beta \mid \sigma(\ell) \leq j\} \\ &= \{\beta_\ell \in \beta' \mid \sigma(\ell) \leq j\} \\ &= \{\beta_\ell \in \beta' \mid \sigma_{pre}(\ell) \leq j\} \\ &= S^{pre}_{\beta'}(j). \end{aligned}$$

The first equality is by definition, the second by the fact that $j \leq p-1 < \sigma(k+1)$. Hence, we obtain $\mathrm{LValid}_{\beta'}(x_{pre}) = \mathit{true}$.

In order to see that x_{pre} is contributor valid along β', consider the language of available writes induced by x and x_{pre}. Let $i \in [1..k]$. Since $S^{pre}_{\beta'}(j) = S_\beta(j)$ for $j \in [1..p-1]$, we get

$$\mathit{LAW}(x_{pre}, \beta_1 \dots \beta_{i-1}) = \mathit{LAW}(x, \beta_1 \dots \beta_{i-1}).$$

Hence, contributor validity carries over to the witness x_{pre} and we have: $\mathrm{CValid}^i_{\beta'}(x_{pre}) = \mathrm{CValid}^i_\beta(x) = \mathit{true}$. This means that also the conjunction of these values is true, $\mathrm{CValid}_{\beta'}(x_{pre}) = \mathit{true}$.

We apply the induction to x_{pre} and obtain a short witness $c = (w_c, q_p, \sigma_c)$ of order k with $\mathit{init}(c) = q_1$, $\mathrm{Valid}^{sh}_{\beta'}(c) = \mathit{true}$, and

$$\mathit{FullExpr}(c, \beta') = \mathit{FullExpr}(x_{pre}, \beta').$$

The witness c is the first of two short witnesses that we will use in the recursion for short validity. The second witness is denoted by $d = (w_d, q, \sigma_d)$ and is defined by $d = \mathit{Shrink}^*(x_{po})$. Then by definition, $d \in \mathit{Ord}(1)$, $\mathit{init}(d) = q_p$, and $\sigma_d(1) = 1$. Note that the target state of c and the initial state of d match. Hence, the witness concatenation $c \times d$ is well-defined.

The short witness of interest is then defined by $z = c \otimes d \in \mathit{Ord}(k+1)$. Hence, $\mathit{ord}(z) = \mathit{ord}(x)$. Furthermore, we immediately get that z is of the form $z = (w_z, q, \sigma_z)$ and that $\mathit{init}(z) = \mathit{init}(c) = q_1 = \mathit{init}(x)$. It is therefore left to show that z is valid, $\mathrm{Valid}^{sh}_\beta(z) = \mathit{true}$, and that the full expressions coincide, $\mathit{FullExpr}(z, \beta) = \mathit{FullExpr}(x, \beta)$.

We first focus on the validity of z. To this end, we make use of the recursive definition of $\mathrm{Valid}^{sh}_\beta(z)$. It is enough to show that $\mathrm{LValid}_\beta(c \times d) = \mathit{true}$ and that $\mathrm{CValid}^{k+1}_\beta(c \times d) = \mathit{true}$. Note that $[z = c \otimes d]$ is true by definition and $\mathrm{Valid}^{sh}_{\beta'}(c) = \mathit{true}$ holds by induction.

Leader validity of $c \times d$ along β is obtained from the implications

$$\mathrm{LValid}_\beta(x) \implies \mathrm{LValid}_\beta(x_{pre} \times x_{po}) \implies \mathrm{LValid}_\beta(c \times x_{po}) \implies \mathrm{LValid}_\beta(c \times d).$$

First note that $\mathrm{LValid}_\beta(x) = \mathit{true}$ by assumption. The first implication is due to the fact that $x = x_{pre} \times x_{po}$. For the second, we use that $\mathrm{Valid}^{sh}_{\beta'}(c) = \mathit{true}$. We apply Lemma B.10 and obtain that $\mathrm{LValid}_{\beta'}(c) = \mathit{true}$. Then, by Lemma B.9, we get that $\mathrm{LValid}_\beta(c \times x_{po}) = \mathit{true}$. The last implication is again an application of Lemma B.9 since $d = \mathit{Shrink}^*(x_{po})$.

Next, we show that $\mathrm{CValid}^{k+1}_\beta(c \times d) = \mathit{true}$. To this end, we prove

$$\mathit{LAW}(x, \beta') = \mathit{LAW}(c \times d, \beta').$$

Since $\mathrm{CValid}_\beta^{k+1}(x) = \mathit{true}$ by assumption, the equality implies that also $\mathrm{CValid}_\beta^{k+1}(c \times d)$ evaluates to *true*. Consider the expression of the concatenation $c \times d = (w_c.w_d, q, \sigma_{c\times d})$ along β'. We have that

$$\mathit{LAW}(c \times d, \beta') = \mathit{FullExpr}(c, \beta').\Gamma_p^*,$$

where $\Gamma_p = \mathit{Loop}(q_p, S_{\beta'}^{c\times d}(|w_c| + 1)) \cup S_{\beta'}^{c\times d}(|w_c| + 1)$. The set of first writes $S_{\beta'}^{c\times d}(|w_c| + 1)$ is given by $\{\beta_\ell \in \beta' \mid \sigma_{c\times d}(\ell) \leq |w_c| + 1\}$. The equality holds since $\sigma_{c\times d}(k + 1) = |w_c| + 1$, a fact that follows from $\sigma_d(1) = 1$. Since the full expressions $\mathit{FullExpr}(c, \beta') = \mathit{FullExpr}(x_{pre}, \beta')$ are equal, we get that

$$\mathit{LAW}(c \times d, \beta') = \mathit{FullExpr}(x_{pre}, \beta').\Gamma_p^*.$$

Now note that $S_{\beta'}^{c\times d}(|w_c| + 1) = \{\beta_1, \ldots, \beta_k\}$. This is due to the fact that $\sigma_{c\times d}(\ell) = \sigma_c(\ell) \leq |w_c|$ for all $\ell \in [1..k]$. Moreover, we have the following equality of available first writes:

$$S_{\beta'}^{x}(p) = \{\beta_\ell \in \beta' \mid \sigma(\ell) \leq p\} = \{\beta_1, \ldots, \beta_k\} = S_{\beta'}^{c\times d}(|w_c| + 1).$$

Hence, we obtain that $\Gamma_p = \mathit{Loop}(q_p, S_{\beta'}^{x}(p)) \cup S_{\beta'}^{x}(p)$. Considering the available writes of x along β', we then get

$$\mathit{LAW}(x, \beta') = \mathit{LAW}(x_{pre} \times x_{po}, \beta') = \mathit{FullExpr}(x_{pre}, \beta').\Gamma_p^*$$

since $\sigma(k + 1) = p$. Thus, we have the desired equality.

Finally, we need to prove that the full expressions of z and x coincide. To this end, we start with $\mathit{FullExpr}(x, \beta)$ and transform it step by step into $\mathit{FullExpr}(z, \beta)$. We begin with the following equality which is a consequence of $x = x_{pre} \times x_{po}$ and Lemma B.14:

$$\mathit{FullExpr}(x, \beta) = \mathit{FullExpr}(x_{pre} \times x_{po}, \beta) = \mathit{FullExpr}(x_{pre}, \beta').\mathit{FullExpr}(x_{po}^{(k+1)}, \beta).$$

Since $d = \mathit{Shrink}^*(x_{po})$ and $\sigma_{po}(1) = 1$, by Lemma B.15, we obtain that $d^{(k+1)} = \mathit{Shrink}^*(x_{po}^{(k+1)})$. Hence, we can apply Lemma B.12 and get that

$$\mathit{FullExpr}(x_{po}^{(k+1)}, \beta') = \mathit{FullExpr}(d^{(k+1)}, \beta')$$

Note that $x^{(k+1)}$ is leader valid wrt β since x is. Now we use that $\mathit{FullExpr}(x_{pre}, \beta') = \mathit{FullExpr}(c, \beta')$ and get the equality:

$$\mathit{FullExpr}(x_{pre}, \beta').\mathit{FullExpr}(x_{po}^{(k+1)}, \beta) = \mathit{FullExpr}(c, \beta').\mathit{FullExpr}(d^{(k+1)}, \beta).$$

Now we apply Lemma B.14 and Lemma B.12 again. Note that we have $z = \mathit{Shrink}^*(c \times d)$ by definition.

$$\mathit{FullExpr}(c, \beta').\mathit{FullExpr}(d^{(k+1)}, \beta) = \mathit{FullExpr}(c \times d, \beta) = \mathit{FullExpr}(z, \beta).$$

This completes the first direction of the proof.

Second Direction: Now let a short witness $z = (w', q, \sigma') \in Wit^{sh}$ be given such that $\text{Valid}^{sh}_{\beta}(z)$ evaluates to true. Like above, we employ an induction to prove a slightly stronger statement. We show that there is a witness $x = (w, q, \sigma) \in Wit$ with $init(x) = init(z)$, order $ord(x) = ord(z)$, $FullExpr(x, \beta) = FullExpr(z, \beta)$, and $\text{LValid}_{\beta}(x) \wedge \text{CValid}_{\beta}(x) = true$.

For the induction basis, let $ord(z) = 0$. In this case, $\beta = \varepsilon$. Set $x = z$. Then we only need to argue that $\text{LValid}_{\varepsilon}(x) = true$ and $\text{CValid}_{\varepsilon}(x) = true$. The latter holds since validity for contributors with empty first-write sequence is always true. Leader validity of x holds since

$$\text{LValid}_{\varepsilon}(x) = \text{LValid}_{\varepsilon}(z) = \text{Valid}^{sh}_{\varepsilon}(z) = true.$$

Let $ord(z) = k + 1$ for $k < d$. Then, the first-write sequence is given by $\beta = \beta'.\beta_{k+1}$ with $\beta' = \beta_1 \dots \beta_k$. Since $\text{Valid}^{sh}_{\beta}(z) = true$, we get by the recursive definition of short validity, two witnesses $c \in Ord(k)$ and $d \in Ord(1)$ such that $z = c \otimes d$, $\text{LValid}_{\beta}(c \times d) = true$, $\text{CValid}^{k+1}_{\beta}(c \times d) = true$, and $\text{Valid}^{sh}_{\beta'}(c) = true$. We denote c by (w_c, q_c, σ_c) and d similarly by (w_d, q_d, σ_d). Note that $init(c) = init(z)$ and $q_d = q$.

Since c is a valid short witness of order k, we can apply induction. We obtain a witness $x' = (w_{x'}, q_c, \sigma_{x'}) \in Wit$ with $init(x') = init(c) = init(z)$, order $ord(x') = k$, full expression $FullExpr(x', \beta') = FullExpr(c, \beta')$, and validity predicate $\text{LValid}_{\beta'}(x') \wedge \text{CValid}_{\beta'}(x') = true$. The desired witness is $x = x' \times d$. Note that the concatenation is well-defined and that it immediately satisfies $x = (w, q, \sigma)$, $init(x) = init(z)$, and $ord(x) = k + 1$. Hence, it is left to show that $\text{LValid}_{\beta}(x) = true$, $\text{CValid}_{\beta}(x) = true$, and that the full expressions of x and z coincide, $FullExpr(x, \beta) = FullExpr(z, \beta)$.

We begin with leader validity. Since $\text{LValid}_{\beta}(c \times d) = true$ and by induction also $\text{LValid}_{\beta'}(x') = true$, we can apply Lemma B.9. It guarantees that $\text{LValid}_{\beta}(x' \times d) = true$, which is what we wanted.

Now consider contributor validity. We know $\text{CValid}_{\beta'}(x') = true$ by induction. This means that each predicate $\text{CValid}^{i}_{\beta'}(x')$ in the conjunction evaluates to *true*. We look at the corresponding language of available writes. For x' and $x = x' \times d$, they are equivalent:

$$LAW(x', \beta_1 \dots \beta_{i-1}) = LAW(x, \beta_1 \dots \beta_{i-1})$$

for each $i \in [1..k]$. The equation is due to $\sigma(i) = \sigma_{x'}(i)$ for $i \leq k$. Since $\text{CValid}^{i}_{\beta'}(x') = true$, also the predicate $\text{CValid}^{i}_{\beta}(x)$ evaluates to true for each $i \in [1..k]$. It is left to argue that $\text{CValid}^{k+1}_{\beta}(x) = true$. We make use of the fact that $\text{CValid}^{k+1}_{\beta}(c \times d) = true$ and we show that the corresponding available writes of x and $c \times d$ coincide. To this end, consider

$$LAW(x, \beta') = LAW(x' \times d, \beta') = FullExpr(x', \beta').LAW(d^{(k+1)}, \beta').$$

The second equation follows by Lemma B.14. Since the full expressions of x' and c coincide by induction, we get the following equation by invoking Lemma B.14 once more:

$$\begin{aligned} \mathit{FullExpr}(x', \beta').\mathit{LAW}(d^{(k+1)}, \beta') &= \mathit{FullExpr}(c, \beta').\mathit{LAW}(d^{(k+1)}, \beta') \\ &= \mathit{LAW}(c \times d, \beta'). \end{aligned}$$

This proves that the languages are the same and that contributor validity carries over to x. We get $\mathrm{CValid}_\beta^{k+1}(x) = \mathit{true}$ and hence $\mathrm{CValid}_\beta(x) = \mathit{true}$.

We show that the full expressions of x and z coincide. To this end, consider

$$\begin{aligned} \mathit{FullExpr}(x, \beta) &= \mathit{FullExpr}(x', \beta').\mathit{FullExpr}(d^{(k+1)}, \beta) \\ &= \mathit{FullExpr}(c, \beta').\mathit{FullExpr}(d^{(k+1)}, \beta) \\ &= \mathit{FullExpr}(c \times d, \beta) \\ &= \mathit{FullExpr}(z, \beta). \end{aligned}$$

The first and the third equation are due to Lemma B.14. The second equation holds since the full expressions of x' and c are equivalent. Finally, the last equation is due to Lemma B.12 which we can apply since $z = \mathit{Shrink}^*(c \times d)$. □

B.3.3 Proof of Lemma 8.27

Proof. Algorithm 8.1 computes for each $\beta \in \mathit{FW}$ and $z \in \mathit{Wit}^{sh}$ the entry $T[\beta, z]$. Hence, we first determine the number of entries that need to be computed. Note that there are at most $O(d^d)$ first-write sequences. The number of short witnesses is bounded by

$$\begin{aligned} O((d \cdot l)^l \cdot l^d \cdot l) &= O(d^l \cdot l^l \cdot l^d \cdot l) \\ &= O((d+l)^l \cdot (d+l)^l \cdot (d+l)^d \cdot l) \\ &= (d+l)^{O(d+l)}. \end{aligned}$$

This means that T has $(d+l)^{O(d+l)} \cdot d^d = (d+l)^{O(d+l)}$ many entries.

To compute a single entry $T[\beta, z]$, we split the short witness z into x and y by iterating over the short witnesses of order $k-1$ and those of order 1. The iteration takes time proportional to the number of short witnesses $(d+l)^{O(d+l)}$. Checking whether $z = x \otimes y$, computing $w = x \times y$, and evaluating the two predicates $\mathrm{LValid}_\beta(w)$ and $\mathrm{CValid}_\beta^{k+1}(w)$ can be done in polynomial time. Moreover, the value $\mathrm{Valid}_{\beta'}^{sh}(x) = T[\beta', x]$ can be looked up in the table and does not need to be recomputed. Hence, computing an entry of T takes time $(d+l)^{O(d+l)}$. The complete table can thus be computed in time

$$(d+l)^{O(d+l)} \cdot (d+l)^{O(d+l)} = (d+l)^{O(d+l)}.$$

The estimation completes the proof. □

B.3.4 Proof of Lemma 8.31

Before we give a proof of Lemma 8.31, we need some new notation. Let $c \in \mathit{Conf}(S)$ be a configuration of the leader contributor system S. Moreover, let $p \in Q_C$ be a state of the contributor. We use $\#_C(c, p)$ to denote the number of contributors of c that are currently in state p. For $c = (q, a, pc)$, define

$$\#_C(c, p) = |\{i \mid pc(i) = p\}|.$$

With the new notation we can state the *copycat lemma* [152] in our setting. The lemma shows that, once $p \in Q_C$ is reached by a computation, there is a similar computation where p is reached by an arbitrary number of contributors. Phrased differently, we can assume $\#_C(c, p)$ to be as large as desired.

Lemma B.17. *Let $c^0 \to_S^* c$ be an initialized computation. Moreover, let $p \in Q_C$ be a contributor state with $\#_C(c, p) > 0$. Then, for all integer $k \in \mathbb{N}$ there is an initialized computation $c^0 \to_S^* d$ such that d satisfies the following:*

$$\pi_L(d) = \pi_L(c),\ \pi_D(d) = \pi_D(c),\ \textit{and}\ \#_C(d, p') = \begin{cases} \#_C(c, p') + k, \textit{ if } p' = p, \\ \#_C(c, p'), \textit{ otherwise.} \end{cases}$$

Now we can prove Lemma 8.31. In fact, it follows from the proof of a slightly stronger statement. For any reachable configuration c in the leader contributor system, there is a vertex v in $\mathcal{G}$, reachable from v^0, such that leader state and memory value of c and v coincide, and v contains a larger set of contributor states than c. We formalize:

Lemma B.18. *There is an initialized computation $c^0 \to_S^* c$ if and only if there is a path $v^0 \to_E^* (q, a, S)$ in $\mathcal{G}$, where $\pi_L(c) = q$, $\pi_D(c) = a$, and $\pi_C(c) \subseteq S$.*

Proof. First assume that an initialized computation $\rho = c^0 \to_S^* c$ is given. We prove the existence of the path in $\mathcal{G}$ by an induction on the length of ρ. In the base case, we have $|\rho| = 0$. This means that $c = c^0$ is the initial configuration. But then $\pi_L(c) = q_L^0$, $\pi_D(c) = a^0$, and $\pi_C(c) = \{q_C^0\}$. This characterizes the initial vertex v^0 of $\mathcal{G}$ and there is clearly a path $v^0 \to_E^* v^0$ of length 0.

Suppose $|\rho| = \ell+1$. Then, the computation can be split into $c^0 \to_S^* c' \to_S c$, where $\rho' = c^0 \to_S^* c'$ is a computation of length ℓ. By induction, there is a path $v^0 \to_E^* (q', a', S')$ in $\mathcal{G}$ such that $q' = \pi_L(c')$, $a' = \pi_D(c')$, and $\pi_C(c') \subseteq S'$. Now we need to distinguish two cases.

(1) If $c' \to_S c$ is induced by a transition of the leader, the leaders' state and the memory value get updated, but the contributor states do not. We have that $\pi_C(c) = \pi_C(c') \subseteq S'$. Now we set $q = \pi_L(c)$, $a = \pi_D(c)$ and $S' = S$. Then, on $\mathcal{G}$ we have an edge $(q', a', S') \to_E (q, a, S)$.

(2) If $c' \to_S c$ is induced by a transition of a contributor, we immediately get that $\pi_L(c) = \pi_L(c') = q'$. Let the transition of the contributor be of the

form $p' \xrightarrow{?a'/\varepsilon}_C p$. Then we have that $\pi_C(c) \subseteq \pi_C(c') \cup \{p\} \subseteq S' \cup \{p\}$ and $\pi_D(c') = \pi_D(c) = a'$. Note that it can happen that p' is not an element of $\pi_C(c)$ since there might be just one contributor in state p' which switches to state p. We set $q = q'$, $a = a'$, and $S = S' \cup \{p\}$. Then we have an edge $(q', a', S') \to_E (q, a, S)$ in the saturation graph, induced by the transition. Writes of the contributors are treated similarly.

For the other direction, we prove a slightly stronger statement: For each path $v^0 \to_E^* (q, a, S)$, there is an initialized computation $c^0 \to_S^* c$ such that $\pi_L(c) = q$, $\pi_D(c) = a$, and $\pi_C(c) = S$. Note that we ensure equality of S and $\pi_C(c)$. We prove the statement by an induction on the length of the given path. In the base case, the length is 0. Then, we have $(q, a, S) = v^0$. This means $q = q_L^0$, $a = a^0$, and $S = \{q_C^0\}$. Considering the initial configuration c^0 for an arbitrary number of contributors, we get the computation $c^0 \to_S^* c^0$ of length 0, with $\pi_L(c^0) = q$, $\pi_D(c^0) = a$, and $\pi_C(c^0) = S$.

For the induction step, let $v^0 \to_E^* (q, a, S)$ be a path of length $\ell + 1$. We split the path into a prefix $v^0 \to_E^* (q', a', S')$ of length ℓ and a single edge $(q', a', S') \to_E (q, a, S)$. Invoking the induction hypothesis, we get a computation $c^0 \to_S^* c'$ such that $\pi_L(c') = q'$, $\pi_D(c') = a'$, and $\pi_C(c') = S'$. Again, we need to distinguish two cases.

(1) Assume the edge $(q', a', S') \to_E (q, a, S)$ was induced by a transition of the leader. Since $\pi_L(c') = q'$ and $\pi_D(c') = a'$, the same transition also induces a step $c' \to_S c$ with $\pi_L(c) = q$, $\pi_D(c) = a$, and $\pi_C(c) = S = S'$.

(2) Assume the edge $(q', a', S') \to_E (q, a, S)$ was induced by a transition of a contributor and let $\tau = p' \xrightarrow{?a'/\varepsilon}_C p$ be this transition. The case of a write is similar. Then we get $S = S' \cup \{p\}$ and $p' \in S'$. Since $\pi_C(c') = S'$, we get that $\#_C(c', p') > 0$. By an application of Lemma B.17 with $k = 1$, we obtain a computation of the form $c^0 \to_S^* d$ such that $\#_C(d, p') > 1$ and for all $r \neq p'$ we have $\#_C(d, r) = \#_C(c', r)$. Furthermore, we get that $\pi_L(d) = \pi_L(c')$ and $\pi_D(d) = \pi_D(c')$. Hence, transition τ induces a move $d \to_S c$, where c is a configuration with $\pi_L(c) = \pi_L(d) = q' = q$, $\pi_D(c) = \pi_D(d) = a' = a$ and $\pi_C(c) = \pi_C(d) \cup \{p\} = S' \cup \{p\} = S$. □

B.3.5 Proof of Lemma 8.39

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

First, we write down the vertices of $\mathcal{G}_W$. This takes time $O(d \cdot l)$. Edges in the graph are either induced by transitions of the leader or by the contributor. All edges induced by the former can be added in time $O(d \cdot |\delta_L|) = O(d^2 \cdot l^2)$ since a single transition of the leader P_L may lead to d many edges.

To add the edges induced by the contributor, we browse δ_C for transitions of the form $s \xrightarrow{!a}_C s'$ with $s, s' \in W$. Each such transition may induce $d \cdot l$ many edges. Adding them takes time $O(|\delta_C| \cdot c \cdot d \cdot l) = O(c^3 \cdot d^2 \cdot l)$ since we have to test membership of s, s' in W. Note that we can omit transitions $s \xrightarrow{?a}_C s'$ with $s, s' \in W$ as their induced edges are self-loops in $\mathcal{G}_W$.

To complete the construction, we add the edges leading from $\mathcal{G}_W$ to $\mathcal{G}_S$. These are induced by transitions $r \xrightarrow{?a\,!a}_C p \in \delta_C$ with $r \in W$. Since each of these may again lead to $d \cdot l$ many different edges, adding all of them takes time $O(c^3 \cdot d^2 \cdot l)$. In total, the time for the construction is $O(c^3 \cdot d^2 \cdot l^2)$.

For the construction of $R(W, S)$, we apply a fixed-point iteration on the constructed slice. It finds all vertices (q, a, S) that are reachable from a vertex (q', a', W) with $(q', a') \in T[W]$. Since the last component of the vertices in the slice is limited to be either W or S, it contains $O(d \cdot l)$ many vertices. Hence, the iteration takes time at most $O(c^2 \cdot l^2)$. □

B.3.6 Proof of Theorem 8.40

Proof. We first elaborate on the formal construction. We have already seen that the leader contributor system $S = (D, a^0, P_L, P_C)$ relies on the data domain

$$D = \{\mathrm{row}(i), \mathrm{col}(i), \#_i \mid i \in [1..k]\} \cup \{a^0\}.$$

The contributor threads are defined by $P_C = (Q_C, OP(D), \delta_C, q_C^0)$. The set of states is given by

$$Q_C = \{q_{(i,j)}^{(r,\ell)}, q_{(i,j)}^{(c,\ell')} \mid i, j, \ell \in [1..k], \ell' \in [0..k]\} \cup \{q_C^0, q_C^f\}.$$

Intuitively, we use the states $q_{(i,j)}^{(r,\ell)}$, $q_{(i,j)}^{(c,\ell')}$ to indicate that the contributor has chosen to store the vertex (i, j) and to count the number of vertices that the contributor has read so far. More precise, the state $q_{(i,j)}^{(r,\ell)}$ reflects that the last symbol read was $\mathrm{row}(\ell)$, the row of the ℓ-th vertex. The state $q_{(i,j)}^{(c,\ell)}$ indicates that the last read symbol was the column symbol belonging to the ℓ-th vertex. Note that this can be any column and thus different from $\mathrm{col}(\ell)$. The transition relation δ_C of the contributor is defined below.

- We have a transition to choose a vertex: $q_C^0 \xrightarrow{?a^0}_C q_{(i,j)}^{(c,0)}$ for any $i, j \in [1..k]$.
- To read the ℓ-th row symbol, we have $q_{(i,j)}^{(c,\ell-1)} \xrightarrow{?\,\mathrm{row}(\ell)}_C q_{(i,j)}^{(r,\ell)}$ for any $\ell \in [1..k]$ and $i, j \in [1..k]$.
- For reading the ℓ-th column symbol, we get $q_{(i,j)}^{(r,\ell)} \xrightarrow{\mathrm{col}(j')}_C q_{(i,j)}^{(c,\ell)}$, but only if one of the following is satisfied: (1) We have that $i \neq \ell$ and there is an

edge between (ℓ, j') and (i, j) in G. Intuitively, the contributor stores a vertex (i, j) from a row i different than ℓ. But then it can only continue its computation if (i, j) and (ℓ, j') share an edge. (2) We have that $i = \ell$ and $j' = j$. This means that the contributor stores the vertex that it has read. Note that with this, we rule out all contributors storing other vertices from the i-th row.

- To end the computation in a contributor, we have $q_{(i,j)}^{(c,k)} \xrightarrow{!\#_i}_C q_C^f$ for any $i, j \in [1..k]$. The rule writes $\#_i$ to the memory. This reflects that (i, j) is indeed connected to all other vertices chosen by the leader.

The leader P_L is given by the tuple $P_L = (Q_L, OP(D), \delta_L, q_L^0)$, with states

$$Q_L = \{q_L^{(r,i)}, q_L^{(c,i)}, q_L^{(\#,i)} \mid i \in [1..k]\} \cup \{q_L^0\}.$$

Unlike for the contributor, the state $q_L^{(r,i)}$ indicates that the last written symbol was row(i), the row of the i-th vertex guessed by P_L. State $q_L^{(c,i)}$ reflects that the last written symbol was the column symbol belonging to the i-th vertex. The remaining states $q_L^{(\#,i)}$ are used to receive the confirmation symbols and count up to k.

The transition relation δ_L is described by the below rules.

- For transmitting the row symbols, we have $q_L^{(c,i-1)} \xrightarrow{!\,\mathrm{row}(i)}_L q_L^{(r,i)}$ for $i \in [1..k]$. Note that we identify q_L^0 as $q_L^{(c,0)}$.

- For writing the column symbols, we have $q_L^{(r,i)} \xrightarrow{!\,\mathrm{col}(j)}_L q_L^{(c,i)}$ for each two indices $i, j \in [1..k]$.

- After sending k row and column symbols, the leader receives the symbols $\#_i$ via the following transitions. We have $q_L^{(\#,i-1)} \xrightarrow{?\#_i}_L q_L^{(\#,i)}$ for $i \in [1..k]$. Here, we denote by $q_L^{(\#,0)}$ the state $q_L^{(c,k)}$.

Further, we set the single final state of P_L to be $Q_F = \{q_L^{(\#,k)}\}$. This is the state P_L reaches after receiving all confirmation symbols. We obtain that d and l are both in $O(k)$. It is left to prove the correctness of the construction. We need to show that there is an initialized computation $c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$ if and only if G contains a clique of size k with one vertex from each row.

First assume that G contains the clique and let $(1, j_1), \ldots, (k, j_k)$ be its vertices. We construct a computation of S involving k contributors. It starts in the initial configuration c^0 and ends in a c with $\pi_L(c) \in Q_F$. Since we have k contributors, let them be denoted by $P_1, \ldots, P_k$.

The computation starts with P_i choosing the vertex (i, j_i) to store. The contributor performs the transition $q_C^0 \xrightarrow{?a^0}_i q_{(i,j_i)}^{(c,0)}$. Combining the transitions for each contributor, we get a computation of the form:

$$c^0 \to_S^* (q_L^0, a^0, q_{(1,j_1)}^{(c,0)}, \ldots, q_{(k,j_k)}^{(c,0)}) = c_0.$$

Then P_L writes the symbol row(1) and each contributor reads it. We get

$$c_0 \to_S^* (q_L^{(r,1)}, \mathrm{row}(1), q_{(1,j_1)}^{(r,1)}, \ldots, q_{(k,j_k)}^{(r,1)}) = c_{(r,1)}.$$

After transmitting the first row to all contributors, P_L then communicates the first column by writing $\mathrm{col}(j_1)$ to the memory. Again, each contributor reads it. Note that P_1 can read $\mathrm{col}(j_1)$ since it stores exactly the vertex $(1, j_1)$. A contributor P_i with $i \neq 1$ can also read $\mathrm{col}(j_1)$ since P_i stores (i, j_i), a vertex which shares an edge with $(1, j_1)$ due to the assumption. Hence, we get the following computation of S:

$$c_{(r,1)} \to_S^* (q_L^{(c,1)}, \mathrm{col}(j_1), q_{(1,j_1)}^{(c,1)}, \ldots, q_{(k,j_k)}^{(c,1)}) = c_1.$$

Similarly, we can construct a computation leading to a configuration c_2. By iterating this process, we get

$$c^0 \to_S^* c_k = (q_L^{(c,k)}, \mathrm{col}(j_k), q_{(1,j_1)}^{(c,k)}, \ldots, q_{(k,j_k)}^{(c,k)}).$$

Then each contributor P_i can write the symbol $\#_i$. This is done in ascending order. First, P_1 writes $\#_1$ and P_L reads it. Then, it is P_2's turn and it writes $\#_2$ to the memory. Again, the leader reads the symbol. After k rounds, we reach the configuration $(q_L^{(\#,k)}, q_{C_1}^f, \ldots, q_{C_k}^f, \#_k)$ which contains the final state of P_L.

For the other direction, assume that S has a computation $\rho = c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$. Let ρ_L be the subcomputation of ρ carried out by P_L. Technically, the projection of ρ to the transitions of P_L. Then the subcomputation ρ_L is of the form $\rho_L = \rho_L^1.\rho_L^2$ with

$$\rho_L^1 = q_L^0 \xrightarrow{!\,\mathrm{row}(1)}_L q_L^{(r,1)} \xrightarrow{!\,\mathrm{col}(j_1)}_L q_L^{(c,1)} \xrightarrow{!\,\mathrm{row}(2)}_L \ldots \xrightarrow{!\,\mathrm{col}(j_k)}_L q_L^{(c,k)},$$
$$\rho_L^2 = q_L^{(c,k)} \xrightarrow{?\#_1}_L q_L^{(\#,1)} \xrightarrow{?\#_2}_L \ldots \xrightarrow{?\#_k}_L q_L^{(\#,k)}.$$

We show that the vertices $(1, j_1), \ldots, (k, j_k)$ form a clique in G.

Since in ρ_L^2, the leader is able to read the symbols $\#_1$ up to $\#_k$, there must be at least k contributors writing them. Due to the structure of P_C, it is not possible to write different confirmation symbols $\#_i$. Hence, we get at least one contributor for each of the symbol $\#_i$.

Let P_i be a contributor writing $\#_i$. Then P_i stores the vertex (i, j_i), it performs the initial transition $q_C^0 \xrightarrow{?a^0}_i q_{(i,j_i)}^{(c,0)}$. Assume P_i stores the vertex (i', j').

Since the contributor writes $\#_i$ in the end, we get $i' = i$ due to the structure of P_C. During the computation ρ, P_i performs the step $q^{(r,i)}_{(i,j')} \xrightarrow{?\,\mathrm{col}(j_i)}_i q^{(c,i)}_{(i,j')}$ since P_L writes the symbol $\mathrm{col}(j_i)$ to the memory and the computation on P_i does not deadlock. Note that we make use the following here. The leader writes $\mathrm{row}(i)$ before $\mathrm{col}(j_i)$. This ensures that $\mathrm{col}(j_i)$ is indeed the column of the i-th transmitted vertex and the above transition is correct. However, the contributor P_i can only do the transition if $j' = j_i$. Thus, we get that $(i', j') = (i, j_i)$. Hence, the contributor P_i indeed stores (i, j_i).

Let $P_{(i,j_i)}$ denote a contributor that writes $\#_i$ during ρ. Since the contributor $P_{(i,j_i)}$ stores the vertex (i, j_i), the leader P_L has written $\mathrm{row}(i)$ and $\mathrm{col}(j_i)$ to the memory. Now let $P_{(i',j_{i'})}$ be another contributor with $i' \neq i$. Then also this thread needs to perform the transition $q^{(r,i)}_{(i',j_{i'})} \xrightarrow{?\,\mathrm{col}(j_i)}_{i'} q^{(c,i)}_{(i',j_{i'})}$ since the computation does not end on $P_{(i',j_{i'})}$ at this point. But by definition, the transition can only be carried out if there is an edge between (i, j_i) and $(i', j_{i'})$. Hence, each two vertices of $(1, j_1), \ldots, (k, j_k)$ share an edge. □

B.3.7 Proof of Theorem 8.41

Proof. We formally construct the leader contributor system $S = (D, a^0, P_L, P_C)$. The data domain D is given by $D = \{u, u^\# \mid u \in U\} \cup \{a^0\}$. The leader is defined by the tuple $P_L = (Q_L, OP(D), \delta_L, q^1)$ with set of states

$$\begin{aligned} Q_L = {} & \{q^i \mid i \in [1..r+1]\} \\ & \cup \{q_T^{(i,j)} \mid T \in \mathcal{F}, j \in [0..|T|-1] \text{ and } i \in [1..r]\} \\ & \cup \{q_\#^i \mid i \in [1..n]\}. \end{aligned}$$

Recall that $n = |U|$. The states q^i are needed to choose a set $T \in \mathcal{F}$. The $q_T^{(i,j)}$ are used to iterate over the elements in T. For the final phase in P_L, the states $q_\#^i$ are needed to read all elements $u^\#$ for $u \in U$.

The transition relation δ_L contains the following transitions:

- For choosing a set, we have transitions of the form $q^i \xrightarrow{\varepsilon}_L q_T^{(i,0)}$ for each $T \in \mathcal{F}$ and $i \in [1..r]$.
- Iterating through a set $T = \{v_1, \ldots, v_{|T|}\}$ is done via the transitions $q_T^{(i,j)} \xrightarrow{!v_{j+1}}_L q_T^{(i,j+1)}$ for $j \in [0, |T|-2]$. For the last element, we have a transition $q_T^{(i,|T|-1)} \xrightarrow{!v_{|T|}}_L q^{i+1}$ that enters the new phase.
- Fix an order on $U = \{u_1, \ldots, u_n\}$. The final check is realized by the transitions $q^{r+1} \xrightarrow{?u_1^\#}_L q_\#^1$ and $q_\#^i \xrightarrow{?u_{i+1}^\#}_L q_\#^{i+1}$ for $i \in [1..n-1]$.

The leader only reaches a final state after the last check: $Q_F = \{q_\#^n\}$.

A contributor is defined by the tuple $P_C = (Q_C, OP(D), \delta_C, p^0)$ where the set of states is given by $Q_C = \{p_u \mid u \in U\} \cup \{p^0\}$. The transition relation δ_C contains transitions to store elements of U in the state space: $p^0 \xrightarrow{?u}_C p_u$, for each $u \in U$. Once an element is stored, the contributor can write it to the memory via the transition $p_u \xrightarrow{!u^\#}_C p_u$.

It is left to prove correctness of the construction. More precise, we need to show that S has an initialized computation of the form $c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$ if and only if there are sets $T_1, \ldots, T_r \in \mathcal{F}$ that cover U.

First assume we have the desired set cover $T_1, \ldots, T_r$. We construct a computation of S with n many contributors. The leader first guesses the set T_1. It writes all elements $u \in T_1$ to the memory. There is one contributor for each element, storing it in its states by reading the corresponding symbol u. Then, the leader chooses T_2 and writes the elements in the set to the memory. Now, only the new elements got stored by a contributor. Elements that were seen already are ignored. We proceed for r phases. Then, the contributors store exactly those elements that got covered by $T_1, \ldots, T_r$. Since these sets cover U, the contributors can write all symbols $u^\#$ to the memory in any order. The leader P_L can thus read the required string and reach its final state.

Now assume there is a computation ρ of S from c^0 to a configuration c with $\pi_L(c) \in Q_F$. Consider ρ_L, the projection of ρ to the leader P_L. Then, the computation ρ_L is of the form $\rho_L = \rho_L^1 \ldots \rho_L^r . \rho_L^f$ with:

$$\rho_L^i = q^i \xrightarrow{\varepsilon}_L q_{T_i}^{(i,0)} \xrightarrow{!u_1^{T_i}}_L q_{T_i}^{(i,1)} \xrightarrow{!u_2^{T_i}}_L \ldots \xrightarrow{!u_{n_i-1}^{T_i}}_L q_{T_i}^{(i,n_i-1)} \xrightarrow{!u_{n_i}^{T_i}}_L q^{i+1},$$

where $T_i = \{u_1^{T_i}, \ldots, u_{n_i}^{T_i}\}$ is a set in $\mathcal{F}$, and

$$\rho_L^f = q^{r+1} \xrightarrow{?u_1^\#}_L q_\#^1 \xrightarrow{?u_2^\#}_L \ldots \xrightarrow{?u_n^\#}_L q_\#^n.$$

The candidate for the set cover of U is $T_1, \ldots, T_r \in \mathcal{F}$. These are the sets selected by P_L during its r initial phases.

A contributor can only read hose symbols that the leader writes to the memory during its ρ_L^i. This means that contributors can only store elements that got covered by the chosen sets T_i. Moreover, they can only write what they have stored. Since ρ_L^f can be carried out by the leader, the contributors can write all elements $u \in U$ to the memory. Phrased differently, all elements $u \in U$ were stored by contributors and hence covered by $T_1, \ldots, T_r$. □

B.3.8 Proof of Theorem 8.42

Proof. We define the leader contributor system $S = (D, a^0, P_L, P_C)$. First, we fix the data domain. It is given by

$$D = \begin{array}{l} \{(u, \ell) \mid u \in \{0,1\}, \ell \in [1..\log(t)]\} \\ \cup \{(x_i, v) \mid i \in [1..n], v \in \{0,1\}\} \\ \cup \{\#_j \mid j \in [1..m]\} \\ \cup \{a^0\}. \end{array}$$

Thus, we have that $d = O(\log(t) + n + m)$, which satisfies the requirements of a cross-composition.

The leader is defined by the tuple $P_L = (Q_L, OP(D), \delta_L, q_L^{(b,0)})$ with states

$$Q_L = \{q_L^{(b,\ell)}, q_L^{(x,i)}, q_L^{(\#,j)} \mid \ell \in [0..\log(t)], i \in [1..n], j \in [1..m]\}.$$

Hence, we have $l = O(\log(t) + n + m)$. The transition relation δ_L of the leader is defined as follows.

- For transmitting the binary representation in the first phase, we have for each index $\ell \in [1..\log(t)]$ the transition $q_L^{(b,\ell-1)} \xrightarrow{!(u,\ell)}_L q_L^{(b,\ell)}$ with the bit $u \in \{0,1\}$.
- For choosing the assignment of variables in the second phase, we have for $i \in [1..n]$ the transition $q_L^{(x,i-1)} \xrightarrow{!(x_i,v)}_L q_L^{(x,i)}$ with $v \in \{0,1\}$. We denoted the state $q_L^{(b,\log(t))}$ by $q_L^{(x,0)}$ to connect the phases.
- In the third phase, P_L wants to read the symbols $\#_j$. Thus, we get for any $j \in [1..m]$ the transition $q_L^{(\#,j-1)} \xrightarrow{?\#_j}_L q_L^{(\#,j)}$. Here we denote by $q_L^{(\#,0)}$ the state $q_L^{(x,n)}$.

The only final state is given by $Q_F = \{q^{(\#,m)}\}$.

The contributor P_C is defined by $P_C = (Q_C, OP(D), \delta_C, q_C^{\varepsilon})$. The set of states Q_C is the union of the following three sets:

$$\begin{array}{l} \{q^w \mid w \in \{0,1\}^{\leq \log(t)}\}, \\ \{q^{(ch,\ell)} \mid \ell \in [1..t]\}, \\ \{q^{\ell}_{(x_i,v)} \mid \ell \in [1..t], i \in [1..n], v \in \{0,1\}\}. \end{array}$$

Intuitively, the states q^w with$w \in \{0,1\}^{\leq \log(t)}$ form the nodes of the tree of the first phase. The remaining states are needed to store the chosen instance, a variable, and its guessed evaluation.

The relation δ_C contains transitions for the three phases. In the first phase, P_C reads the bits chosen by the leader. According to the value of the bit, it branches to the next state, we have

$$q^w \xrightarrow{?(u,|w|+1)}_C q^{w.u}$$

for $u \in \{0,1\}$ and $w \in \{0,1\}^{\leq \log(t)-1}$. Then we get an ε-transition from those leaves of the tree that encode a proper index, lying in $[1..t]$. We have $q^w \xrightarrow{\varepsilon}_C q^{(ch,\ell)}$ if $w = \text{bin}(\ell)$ and $\ell \in [1..t]$.

For the second phase, we need transitions to store a variable and its evaluation. For each $\ell \in [1..t]$, $i \in [1..n]$, and $v \in \{0,1\}$, we have

$$q^{(ch,\ell)} \xrightarrow{?(x_i,v)}_C q^{\ell}_{(x_i,v)}.$$

In the third phase, the contributor loops. We have $q^{\ell}_{(x_i,v)} \xrightarrow{!\#_j}_C q^{\ell}_{(x_i,v)}$ if x_i evaluated to v satisfies clause C^{ℓ}_j.

The instance of LCR can thus be computed in polynomial time and the significant parameters d and l are both bounded by $O(\log(t) + n + m)$. Hence, for a cross-composition it is only left to show correctness of the construction: there is an initialized computation $c^0 \to^*_S c$ with $\pi_L(c) \in Q_F$ if and only if there is an $\ell \in [1..t]$ such that φ_ℓ is satisfiable.

We first assume that there is an $\ell \in [1..t]$ such that φ_ℓ is satisfiable. Let $v_1, \ldots, v_n$ be the evaluation of the variables $x_1, \ldots, x_n$ that satisfies φ_ℓ. Further, let $\text{bin}(\ell) = u_1 \ldots u_{\log(t)}$ be the binary representation of ℓ. We construct a computation of S with n many contributors. Denote them by $P_1, \ldots, P_n$. Intuitively, P_i is responsible for variable x_i.

The computation proceeds as in the aforementioned phases. In the first phase, P_L starts to guess the first bit of $\text{bin}(\ell)$. It is read by all the contributors. We get a computation

$$\begin{aligned} c^0 &\xrightarrow{!(u_1,1)}_L (q_L^{(b,1)}, (u_1,1), q^{\varepsilon}, \ldots, q^{\varepsilon}) \\ &\xrightarrow{?(u_1,1)}_{P_1} (q_L^{(b,1)}, (u_1,1), q^{u_1}, q^{\varepsilon}, \ldots, q^{\varepsilon}) \\ \ldots &\xrightarrow{?(u_1,1)}_{P_n} (q_L^{(b,1)}, (u_1,1), q^{u_1}, \ldots, q^{u_1}) = c^{(b,1)}. \end{aligned}$$

The computation proceeds with P_L guessing the remaining bits and the contributors reading them. We get

$$c^0 \to^*_S c^{(b,\log(t))} = (q_L^{(b,\log(t))}, (u_{\log(t)}, \log(t)), q^{\text{bin}(\ell)}, \ldots, q^{\text{bin}(\ell)}).$$

Before the second phase starts, the contributors perform an ε-transition to the state $q^{(ch,\ell)}$. This is possible since $\text{bin}(\ell)$ encodes a proper index in $[1..t]$. We get the computation

$$c^{(b,\log(t))} \to^*_S (q_L^{(b,\log(t))}, (u_{\log(t)}, \log(t)), q^{(ch,\ell)}, \ldots, q^{(ch,\ell)}) = c^{(x,0)}.$$

In the second phase P_L chooses the correct evaluation for the variables, it writes (x_i, v_i) for each variable x_i. Contributor P_i reads (x_i, v_i) and stores it. Hence, we get the computation

$$\begin{aligned} c^{(x,0)} &\xrightarrow{!(x_1,v_1)}_L (q_L^{(x,1)}, (x_1, v_1), q^{(ch,\ell)}, \ldots, q^{(ch,\ell)}) \\ &\xrightarrow{?(x_1,v_1)}_{P_1} (q_L^{(x,1)}, (x_1, v_1), q^{\ell}_{(x_1,v_1)}, q^{(ch,\ell)}, \ldots, q^{(ch,\ell)}) \\ &\xrightarrow{!(x_2,v_2)}_L (q_L^{(x,2)}, (x_2, v_2), q^{\ell}_{(x_1,v_1)}, q^{(ch,\ell)}, \ldots, q^{(ch,\ell)}) \\ &\xrightarrow{?(x_2,v_2)}_{P_2} (q_L^{(x,2)}, (x_2, v_2), q^{\ell}_{(x_1,v_1)}, q^{\ell}_{(x_2,v_2)}, q^{(ch,\ell)}, \ldots, q^{(ch,\ell)}) \\ \ldots &\xrightarrow{?(x_n,v_n)}_{P_n} (q_L^{(x,n)}, (x_n, v_n), q^{\ell}_{(x_1,v_1)}, \ldots, q^{\ell}_{(x_n,v_n)}) = c^{(x,n)}. \end{aligned}$$

In the last phase, the contributors write the symbols $\#_j$. Since φ_ℓ is satisfied by the evaluation $v_1, \ldots, v_n$, there is a variable $i_1 \in [1..n]$ such that x_{i_1} evaluated to v_{i_1} satisfies clause C^ℓ_1. Hence, due to the transition relation δ_C, we can let P_{i_1} write the symbol $\#_1$. After that, the leader reads it and moves to the next state. This amounts to the computation

$$\begin{aligned} c^{(x,n)} &\xrightarrow{!\#_1}_{P_{i_1}} (q_L^{(x,n)}, \#_1, q^{\ell}_{(x_1,v_1)}, \ldots, q^{\ell}_{(x_n,v_n)}) \\ &\xrightarrow{?\#_1}_L (q_L^{(\#,1)}, \#_1, q^{\ell}_{(x_1,v_1)}, \ldots, q^{\ell}_{(x_n,v_n)}) = c^{(\#,1)}. \end{aligned}$$

Similarly, we can extend the computation to reach the configuration

$$c^{(\#,m)} = (q_L^{(\#,m)}, \#_m, q^{\ell}_{(x_1,v_1)}, \ldots, q^{\ell}_{(x_n,v_n)})$$

which contains the final state of P_L. This proves the first direction.

Now we assume the existence of a computation ρ from c^0 to a configuration c with $\pi_L(c) \in Q_F$. Let ρ_L be the subcomputation of ρ carried out by the leader. Then ρ_L can be split into $\rho_L = \rho^1_L.\rho^2_L.\rho^3_L$ such that

$$\begin{aligned} \rho^1_L &= q_L^{(b,0)} \xrightarrow{!(u_1,1)}_L q_L^{(b,1)} \xrightarrow{!(u_2,2)}_L \ldots \xrightarrow{!(u_{\log(t)},\log(t))}_L q_L^{(b,\log(t))}, \\ \rho^2_L &= q_L^{(b,\log(t))} \xrightarrow{!(x_1,v_1)}_L q_L^{(x,1)} \xrightarrow{!(x_2,v_2)}_L \ldots \xrightarrow{!(x_n,v_n)}_L q_L^{(x,n)}, \\ \rho^3_L &= q_L^{(x,n)} \xrightarrow{?\#_1}_L q_L^{(\#,1)} \xrightarrow{?\#_2}_L \ldots \xrightarrow{?\#_m}_L q_L^{(\#,m)}. \end{aligned}$$

Let ℓ be the natural number such that $\mathrm{bin}(\ell) = u_1 \ldots u_{\log(t)}$. We show that $\ell \in [1..t]$ and φ_ℓ is satisfied by evaluating the variables x_i to v_i.

In ρ^3_L, the leader can read the symbols $\#_1, \ldots, \#_m$. This means that there is at least one contributor writing them. Let P_C be a contributor writing such a symbol. Then, after P_L has finished ρ^1_L, the contributor P_C is still active and performs the step $q^{\mathrm{bin}(\ell)} \xrightarrow{\varepsilon}_C q^{(ch,\ell)}$. This is true since P_C did not miss a bit

transmitted by P_L and P_C has to reach a state where it can write the #-symbols. Thus, we get that $\ell \in [1..t]$ and P_C stores ℓ in its state space.

We denote the number of contributors writing a #-symbol in ρ by $t' \geq 1$. Each of these contributors gets labeled by $C(j) = \{\#_{j_1}, \ldots, \#_{j_{k_j}}\}$, the set of #-symbols it writes during the computation ρ. Hence, we have the contributors $P_{C(1)}, \ldots, P_{C(t')}$ and since each symbol in $\{\#_1, \ldots, \#_m\}$ is written at least once,

$$\{\#_1, \ldots, \#_m\} = \bigcup_{j=1}^{t'} C(j).$$

Now we show that each $P_{C(j)}$ with $C(j) = \{\#_{j_1}, \ldots, \#_{j_{k_j}}\}$ stores a tuple (x_i, v_i) such that x_i evaluated to v_i satisfies the clauses $C^\ell_{j_1}, \ldots, C^\ell_{j_{k_j}}$. We already know that $P_{C(j)}$ is in state $q^{(ch,\ell)}$ after P_L has executed ρ^1_L. During P_L executing ρ^2_L, the contributor $P_{C(j)}$ has to read a tuple (x_i, v_i) since it has to reach a state where it can write the #-symbols. More precise, $P_{C(j)}$ has to perform a transition

$$q^{(ch,\ell)} \xrightarrow{?(x_i,v_i)}_{P_{C(j)}} q^\ell_{(x_i,v_i)}$$

for some i. Then the contributor writes the symbols $\#_{j_1}, \ldots, \#_{j_{k_j}}$ while looping in the current state. But by the definition of the transition relation for the contributors, this means that x_i evaluated to v_i satisfies the clauses $C^\ell_{j_1}, \ldots, C^\ell_{j_{k_j}}$.

Since each #-symbol is written at least once, we can deduce that every clause in φ_ℓ is satisfied by the chosen evaluation. Hence, φ_ℓ is satisfiable. □

B.3.9 Proof of Theorem 8.44

Proof. We first give construction and proof for the W[1]-hardness of LCR(l). Set the data domain to be

$$D = \{(v, i), (v^\#, i), \#_i \mid v \in V, i \in [1..k]\} \cup \{a^0\}.$$

The leader P_L is given by the tuple $P_L = (Q_L, OP(D), \delta_L, q^0)$ with states

$$Q_L = \{q^i_V, q^i_{V^\#}, q^i_\# \mid i \in [1..k]\} \cup \{q^0\}.$$

The transition relation δ_L is defined by the following. For the first phase, we have for each $i \in [1..k]$ and $v \in V$: $q^{i-1}_V \xrightarrow{!(v,i)}_L q^i_V$. We identify the vertex q^0_V by q^0. For the second phase, we have for each $i \in [1..k]$ and $v \in V$, the transition $q^{i-1}_{V^\#} \xrightarrow{!(v^\#,i)}_L q^i_{V^\#}$. Here, we denote by $q^0_{V^\#}$ the vertex q^k_V. Finally, the third phase is realized by $q^{i-1}_\# \xrightarrow{?\#_i}_L q^i_\#$ for each $i \in [1..k]$. Here we assume $q^0_\# = q^k_{V^\#}$. Further, we set $Q_F = \{q^k_\#\}$ to be the final state of interest.

The contributor is defined by the tuple $P_C = (Q_C, OP(D), \delta_C, q_C^0)$. The set of states is given by

$$Q_C = \{q_{(v,i)}^j \mid i \in [1..k], j \in [0..k]\} \cup \{q_C^0, q_C^f\}.$$

We define the transition relation δ_C as follows. In the first phase, we have for each $i \in [1..k]$ and $v \in V$ the transition $q_C^0 \xrightarrow{?(v,i)}_C q_{(v,i)}^0$. In the second phase, we have for each $i \in [1..k]$, $j \in [0..k]$ and $v, w \in V$ the transition

$$q_{(v,i)}^{j-1} \xrightarrow{?(w,j)}_C q_{(v,i)}^j$$

if (1) $j = i$ and $v = w$, or if (2) $i \neq j$, $v \neq w$, and v and w are adjacent in G. In the third phase, the contributors confirm by the transition $q_{(v,i)}^k \xrightarrow{!\#_i}_C q_C^f$ for each $i \in [1..k]$ and $v \in V$.

Note that $l = O(k)$. It is left to show the correctness of the construction: there is an initialized computation $c^0 \to_S^* c$ with $\pi_L(c) \in Q_F$ if and only if G has a clique of size k.

We first assume that G contains a clique of size k. Let it consist of the vertices $v_1, \ldots, v_k$. We construct a computation of S with k contributors. Denote them by $P_1, \ldots, P_k$ We proceed in the three phases described above.

In the first phase, the leader writes the values $(v_1, 1), \ldots, (v_k, k)$ to the memory. Contributor P_i reads value (v_i, i) and stores it in its state space:

$$\begin{aligned}
c^0 &\xrightarrow{!(v_1,1)}_L (q_V^1, (v_1,1), q_C^0, \ldots, q_C^0) \\
&\xrightarrow{?(v_1,1)}_{P_1} (q_V^1, (v_1,1), q_{(v_1,1)}^0, q_C^0, \ldots, q_C^0) \\
&\xrightarrow{!(v_2,2)}_L (q_V^2, (v_2,2), q_{(v_1,1)}^0, q_C^0, \ldots, q_C^0) \\
&\xrightarrow{?(v_2,2)}_{P_2} (q_V^2, (v_2,2), q_{(v_1,1)}^0, q_{(v_2,2)}^0, q_C^0, \ldots, q_C^0) \\
\ldots &\xrightarrow{?(v_k,k)}_k (q_V^k, (v_k,k), q_{(v_1,1)}^0, \ldots, q_{(v_k,k)}^0) = c_0.
\end{aligned}$$

After reaching c_0, the leader starts the second phase. It writes $(v_1^\#, 1)$ and each contributor reads the symbol:

$$\begin{aligned}
c_0 &\xrightarrow{!(v_1^\#,1)}_L (q_{V^\#}^1, (v_1^\#,1), q_{(v_1,1)}^0, \ldots, q_{(v_k,k)}^0) \\
&\xrightarrow{?(v_1^\#,1)}_{P_1} (q_{V^\#}^1, (v_1^\#,1), q_{(v_1,1)}^1, q_{(v_2,2)}^0, \ldots, q_{(v_k,k)}^0) \\
\ldots &\xrightarrow{?(v_1^\#,1)}_{P_k} (q_{V^\#}^1, (v_1^\#,1), q_{(v_1,1)}^1, \ldots, q_{(v_k,k)}^1) = c_1.
\end{aligned}$$

Note that P_1 can read $(v_1^\#, 1)$ and move onward since it stores exactly $(v_1, 1)$. Any P_i with $i \neq 1$ can read $(v_1^\#, 1)$ and continue its computation since $v_i \neq v_1$

and the two vertices are adjacent. Similarly, one can continue the computation: $c_1 \to_S^* c_k = (q^k_{V^\#}, (v^\#_k, k), q^k_{(v_1,1)}, \ldots, q^k_{(v_k,k)})$.

In the third phase, contributor P_i writes the symbol $\#_i$ to the memory. The leader waits to read the complete string $\#_1 \ldots \#_k$. This yields the following computation which brings the leader to its final state:

$$\begin{aligned} c_k &\xrightarrow{!\#_1}_{P_1} (q^k_{V^\#}, \#_1, q^f_C, q^k_{(v_2,2)}, \ldots, q^k_{(v_k,k)}) \\ &\xrightarrow{?\#_1}_{L} (q^1_\#, \#_1, q^f_C, q^k_{(v_2,2)}, \ldots, q^k_{(v_k,k)}) \\ \ldots &\xrightarrow{?\#_k}_{L} (q^k_\#, \#_k, q^f_C, \ldots, q^f_C). \end{aligned}$$

For the other direction, $\rho = c^0 \to_S^* c$ be a computation with $\pi_L(c) \in Q_F$. We denote by ρ_L the part of the computation that is carried out by the leader P_L. Then we can factor ρ_L into $\rho_L = \rho^1.\rho^2.\rho^3$ with

$$\begin{aligned} \rho^1 &= q^0 \xrightarrow{!(v_1,1)}_L q^1_V \xrightarrow{!(v_2,2)}_L \ldots \xrightarrow{!(v_k,k)}_L q^k_V, \\ \rho^2 &= q^k_V \xrightarrow{!(w^\#_1,1)}_L q^1_{V^\#} \xrightarrow{!(w^\#_2,2)}_L \ldots \xrightarrow{!(w^\#_k,k)}_L q^k_{V^\#}, \\ \rho^3 &= q^k_{V^\#} \xrightarrow{?\#_1}_L q^1_\# \xrightarrow{?\#_2}_L \ldots \xrightarrow{?\#_k}_L q^k_\#. \end{aligned}$$

We show that $w_i = v_i$ for any $i \in [1..k]$ and that $v_i \neq v_j$ for $i \neq j$. Furthermore, we prove that each two vertices v_i, v_j are adjacent. Hence, $v_1, \ldots, v_k$ forms a clique of size k in the graph G.

Since P_L is able to read the symbols $\#_1, \ldots, \#_k$ in ρ^3, there are at least k contributors writing them. But a contributor can only write $\#_i$ in its computation if it reads (and stores) the symbol (v_i, i) from ρ^1. Hence, there is at least one contributor storing (v_i, i). We denote it by P_{v_i}.

The computation ρ^2 starts by writing $(w^\#_1, 1)$ to the memory. The contributor P_{v_i} has to read it to finally reach a state where it can write the symbol $\#_i$. Hence, P_{v_1} reads the symbol $(w^\#_1, 1)$ without deadlocking. By the definition of the transition relation of P_{v_1} this means that $w_1 = v_1$. Now consider P_{v_i} with $i \neq 1$. This contributor also reads $(w^\#_1, 1) = (v^\#_1, 1)$. By definition this implies that $v_i \neq v_1$ and the two vertices are adjacent. By induction, we get that $w^\#_i = v_i$ for all $i \in [1..k]$, that the v_i are all distinct and that each two of the vertices v_i share an edge in G. Hence, G contains a clique of size k.

To prove the $\mathsf{W}[1]$-hardness of LCR(d), we go back to our idea to transmit vertices in binary. More precise, we construct a leader contributor system S with $d = O(k)$ and moreover, there is a computation of S bringing the leader to a final state if and only if G contains a clique of size k. Recall that $t = \log(|V|)$ and that $\text{bin} : V \to \{0,1\}^t$ is some binary encoding of the vertices of G.

The system S acts like the leader contributor system above, in three phases. In the first phase, the leader chooses the vertices of a clique candidate. This is

done by repeatedly writing a string $\#.(b_1, i).\# \ldots \#.(b_t, i).\#$ to the memory, for each $i \in [1..k]$. Intuitively, this is the binary encoding of some vertex $v \in V$ with $\text{bin}(v) = b_1 \ldots b_t$. Hence, instead of guessing a vertex immediately, the leader guesses the binary representation of a vertex. Like above, the contributors then non-deterministically decide to store a written vertex. To this end, a contributor that wants to store the i-th suggested vertex has a binary tree branching on the symbols $(0, i)$ and $(1, i)$. Leaves of the tree correspond to binary encodings of vertices. Hence, a particular vertex can be stored in the contributor's states. Note that we did not assume $|V|$ to the a power of 2. This means there might be leaves of the tree that do not correspond to encodings of vertices. If a computation reaches such a leaf it will deadlock.

In the second phase, the leader again writes the binary encoding of k vertices to the memory. But this time, it uses a different set of symbols: Instead of 0 and 1, the leader uses $0^\#$ and $1^\#$ to separate Phase two from Phase one. The contributors need to compare the suggested vertices as in the above construction. To this end, a contributor storing the vertex (v, i) proceeds in k stages. In stage $j \neq i$ it can only read the encodings of those vertices which are adjacent to v. Hence, if the leader suggests a wrong vertex, the computation will deadlock. In stage i, the contributor can only read the encoding of the stored vertex v. This allows for a verification of the clique as above.

The last phase is identical to the last phase of the above construction. The contributors write the symbols $\#_i$, while the leader waits to read the string $\#_1 \ldots \#_k$. Once the leader has read the symbols it reaches its final state. This ensures that the chosen vertices form a clique. We omit a formal construction as it would be quite similar to the one given above. □

B.3.10 Proof of Lemma 8.53

Proof. If we are given a computation $c^0 \to^* c \to^+_{sat} c$ with $I(c)$, we split it into the prefix $c^0 \to^* c$ and the cycle $c \to^+_{sat} c$. The interface I is clearly matched.

For the other direction, let computations $d^0 \to^* d$ and $f \to^+_{sat} f$ with $I(d) \wedge I(f)$ be given. We construct a computation $c^0 \to^* c \to^+_{sat} c$ with $I(c)$ as desired. To this end, recall that for a configuration e and contributor state $p \in Q_C$, the value $\#_C(e, p)$ denotes the number of contributors of e that are currently in the state p.

We construct the required computation. Let c be a configuration that contains for each contributor state p the maximal number of contributors of d and f that are currently in p. Memory and leader state of c are identical to those of d and f. Formally we have, $\#_C(c, p) = \max(\#_C(d, p), \#_C(f, p))$ for each state $p \in Q_C$. Moreover, $\pi_L(c) = \pi_L(d)$ and $\pi_D(c) = \pi_D(d)$. This implies that c matches the interface I: we have $I(c)$.

In the following, we show that prefix and cycle can be composed. By the *copycat lemma* [152], we can enrich the computation $d^0 \to^* d$ by contributors such that we get $c^0 \to^* c$. If we have $\max(\#_C(d, p), \#_C(f, p)) = \#_C(d, p)$, we do

not have to add contributors for state p. If $\max(\#_C(d,p), \#_C(f,p)) > \#_C(d,p)$, we add contributors for the difference $t = \max(\#_C(d,p), \#_C(f,p)) - \#_C(d,p)$. Let P be any contributor in d currently in state p. Then, we add t copies $P_1^c, \ldots, P_t^c$ of P to d. Since the behavior of the leader and the memory do not change, we get the prefix $c^0 \rightarrow^* c$.

The cycle $f \rightarrow_{sat}^+ f$ can be simulated on the larger configuration c. Intuitively, the contributors that do not participate in the cycle, can be ignored. Hence, we obtain the desired cycle $c \rightarrow_{sat}^+ c$. Note that it is saturated. □

B.3.11 Proof of Lemma 8.60

Proof. Let $\Gamma \subseteq \Gamma'$ be two subsets of D. Since the writes $Writes(SCCdcmp_T(\Gamma))$ splits into $Writes_C(SCCdcmp_T(\Gamma))$ and $Writes_L(SCCdcmp_T(\Gamma))$, we show the following two inclusions which prove the lemma:

$$Writes_C(SCCdcmp_T(\Gamma)) \subseteq Writes_C(SCCdcmp_T(\Gamma')) \text{ and}$$
$$Writes_L(SCCdcmp_T(\Gamma)) \subseteq Writes_L(SCCdcmp_T(\Gamma')).$$

To this end, let $SCCdcmp_T(\Gamma) = (S_1, \ldots, S_\ell)$ be the Γ-SCC decomposition of T and $SCCdcmp_T(\Gamma') = (T_1, \ldots, T_k)$ the Γ'-SCC decomposition.

For the first inclusion, take an element $b \in Writes_C(S_1, \ldots, S_\ell)$. By definition, there are states p, p' in a component S_i and a transition $p \xrightarrow{!b}_C p'$. Since p, p' are in S_i, they are strongly connected in the graph $\mathcal{G}_T(\Gamma)$. Hence, the states are also strongly connected in $\mathcal{G}_T(\Gamma')$. In fact, $\Gamma \subseteq \Gamma'$ implies that all the edges of $\mathcal{G}_T(\Gamma)$ are also present in $\mathcal{G}_T(\Gamma')$. Given that $(T_1, \ldots T_k)$ is the Γ'-SCC decomposition of S, the states p and p' have to lie in one set T_j. Hence, b occurs as a write within a set of $(T_1, \ldots, T_k)$ which means $b \in Writes_C(T_1, \ldots, T_k)$.

It is left to show the second inclusion. Let $b \in Writes_L(S_1, \ldots, S_\ell)$. Then, there are words $u, v \in OP(D)^*$ such that $(q,a) \xrightarrow{u.!b.v}_{L'(\Gamma)} (q,a)$. Recall that $\rightarrow_{L'(\Gamma)}$ is the transition relation of the automaton $P_{L'(\Gamma)}$. It restricts the transitions of the leader to reads within the set $Writes_C(S_1, \ldots, S_\ell)$ and keeps track of the current memory content. The latter may change due to a contributor write in $Writes_C(S_1, \ldots, S_\ell)$. Since we already know the inclusion $Writes_C(S_1, \ldots, S_\ell) \subseteq Writes_C(T_1, \ldots, T_k)$, the automaton $P_{L'(\Gamma')}$ contains all the transitions of $P_{L'(\Gamma)}$. Hence, the sequence of transitions $(q,a) \xrightarrow{u.!b.v}_{L'(\Gamma)} (q,a)$ in $P_{L'(\Gamma)}$ can also be performed in $P_{L'(\Gamma')}$. By definition, we obtain that $b \in Writes_L(T_1, \ldots, T_k)$. □

B.3.12 Proof of Lemma 8.63

Proof. It remains to give a formal proof of the second direction of the lemma. Let a non-empty set Γ be given such that $SCCdcmp_T(\Gamma) = (T_1, \ldots, T_\ell)$ is stable. This means that $\Gamma = Writes(T_1, \ldots, T_\ell)$. We split the set $\Gamma = \Gamma_C \cup \Gamma_L$, where $\Gamma_C = Writes_C(T_1, \ldots, T_\ell)$ are the writes provided by the contributors and $\Gamma_L = Writes_L(T_1, \ldots, T_\ell)$ are the writes of the leader.

We fix a run of the leader. It is of the form $\pi = (q, a) \xrightarrow{w}_{L'} (q, a)$ and it writes every symbol in Γ_L at least once. Formally, for each $g \in \Gamma_L$ there are $u, v \in OP(D)^*$ such that $w = u!gv$. Note that such a run exists. Potentially, we have to compose several cycles from (q, a) back to (q, a). We denote the length of the run π by t.

For each element $b \in \Gamma_C$, let $p_0(b)$ and $p_1(b)$ be two states belonging to a set $T_{i(b)}$ of the Γ-SCC decomposition such that there is a transition of the form

$$p_0(b) \xrightarrow{!b} p_1(b).$$

Note that such a transition exists by definition. We call the set of states $Gen = \{p_0(b) \mid b \in \Gamma_C\}$ the *symbol generators*. Further, we fix a cycle for each symbol $b \in \Gamma_C$. Let

$$cycle(b) = p_0(b) \to_C p_1(b) \to_C p_2(b) \to \cdots \to_C p_k(b) = p_0(b)$$

be a cyclic run in within $T_{i(b)}$, reading only symbols from Γ. Such a run exists since $T_{i(b)}$ is strongly connected in the graph $\mathcal{G}_T(\Gamma)$. We use $States(cycle(b))$ to refer to the set $\{p_0(b), \ldots, p_{k-1}(b)\}$ of states that appear in $cycle(b)$. Moreover, given a configuration $f = (p, b, pc)$ and a state $s \in Q_C$, we use $f[s]$ to denote the indices of the contributors that are currently in state s:

$$f[s] = \{j \mid pc(j) = s\}.$$

We construct a computation ρ. The idea is to support the run π of the leader and to provide all the needed symbols along its way. Moreover, we need to balance the computation: the number of contributors in a particular state is preserved after executing ρ. This is achieved by moving the contributors along the fixed cycles.

For the construction, we start with $t + 1$ many contributors in each state of $cycle(b)$, for all symbols $b \in \Gamma_C$. Formally, we choose our initial configuration c in such a way that for each $s \in T$ we have

$$|c[s]| = \begin{cases} (t+1) \cdot |\{b \in \Gamma_C \mid s \in cycle(b)\}|, & \text{if } s \text{ lies in any cycle} \\ 1, & \text{otherwise.} \end{cases}$$

Note that we add a single contributor in s if the state does not appear in any cycle. This contributor does not move during the computation. The reason is that we can then ensure $\pi_C(c) = T$ throughout the computation which keeps ρ saturated. Moreover, we start with the appropriate leader state and memory value, $\pi_L(c) = q$, $\pi_D(c) = a$.

During ρ, each contributor in a cycle moves to its neighbor state by making exactly one move. To this end, we split ρ into two phases: $\rho = \rho_1.\rho_2$. In the first phase ρ_1, only the contributors move and the leader stays idle. The purpose of this phase is to ensure that all contributors can go to their neighbor

in the cycle when reading of a symbol from Γ_C is required or when writing. Reading of other symbols is handled in ρ_2.

Note that we have enough contributors in c to provide each symbol in Γ_C exactly $t+1$ many times. During ρ_1, we use up one of these contributors for each symbol and provide each symbol in Γ_C once. To realize ρ_1, let $b \in \Gamma_C$. Pick one of the contributors currently in the state $p_0(b)$. It makes a move to $p_1(b)$ and writes b to the memory. This is followed by a transition of every contributor in each of the cycles that can read b and move to their neighbor. After the move, these contributors stay idle for the remainder of ρ. Then we proceed with the next symbol in Γ_C.

Let $c \to^* c_1'$ be the resulting computation. At the end of the computation, each transition in each copy of a cycle that involves reading a symbol from Γ_C is already executed. Furthermore, one copy of the symbol generators is exhausted, the corresponding contributors made a move to the next state in the cycle. We still have t contributors in the symbol generators left. Indeed, $|c_1'[p_0(b)]| \geq t$ for each $b \in \Gamma_C$.

We complete the computation ρ_1. For any contributor currently in a state $s \in \mathit{cycle}(b)$ that is not a symbol generator, $s \notin \mathit{Gen}$, we do the following. If the contributor can write a symbol from Γ_C and move to its neighbor state in $\mathit{cycle}(b)$, we execute the transition. The written symbol is ignored by the other contributors and by the leader. After executing these write transitions, we are at a configuration $\hat{c}_1$. We get $\rho_1 = c \to^* \hat{c}_1$. Still, we have t contributors in the symbol generators left, $|\hat{c}_1[p_0(b)]| \geq t$ for each $b \in \Gamma_C$. Hence, the contributors on the cycles that did not do a move so far are either the ones in the symbol generators or ones that require a symbol of Γ_L, written by the leader.

We construct the second phase ρ_2 which shows how the leader runs. Recall that we already fixed the run π of the leader providing all symbols in Γ_L. We execute each transition of π interleaved with transitions of the contributors while maintaining two invariants. To formalize them, let $i \in [1..t]$. By $\Gamma_L^i \subseteq \Gamma_L$ we denote the set of symbols that the leader has written after i many steps of π. The invariants are as follows. (1) all contributors that are currently in a state $s \in \mathit{cycle}(b)$ for a $b \in \Gamma_C$ but not in Gen and that can reach their neighbor while reading a symbol from Γ_L^i, have already performed this transition before the $(i+1)$-st step of π is taken. (2) Before the $(i+1)$-st step of π gets executed, for each $b \in \Gamma_C$, there are exactly $t - i$ many contributors left that can provide b. These are in the state $p_0(b)$.

We construct the computation inductively. Assume, we already executed $i-1$ many steps of π. We denote the interleaved computation with the transitions of the contributors by ρ_2^{i-1}. We need a case distinction.

If the i-th step of π, denoted by $\pi(i)$, is a write transition, we do not need to provide a symbol for the leader. The idea is to execute $\pi(i)$ and to let the contributors read the written symbol. Let $b \in \Gamma_L$ be that symbol. Then $\Gamma_L^i = \Gamma_L^{i-1} \cup \{b\}$. We first execute $\pi(i)$ and write b to the shared memory. Now, each contributor on a cycle that needs to read a b to arrive at its neighbor takes

the corresponding read transition. This maintains Invariant (1). To ensure that (2) also holds, we add the following computation. For each symbol $b \in \Gamma_C$ we pick exactly one contributor in $p_0(b)$ and let it write b to the memory. The write is ignored by others. This way, we consume exactly one copy of these contributors, maintaining (2).

If $\pi(i)$ is a read of a symbol $b \in \Gamma_C$, we pick one contributor that is currently in $p_0(b)$. We let it execute its transition

$$p_0(b) \xrightarrow{!b}_C p_1(b)$$

to provide b. The transition is followed by the leader taking $\pi(i)$. Invariant (1) is already ensured at this point since $\Gamma_L^i = \Gamma_L^{i+1}$. To guarantee (2), we consume copies for symbols different from b. Let $b' \in \Gamma_C$, $b' \neq b$. We let one copy of a contributor, currently in $p_0(b')$, perform its write transition on b'. The write is ignored by others. After executing these transitions, (2) holds.

Depending on the case, we add the resulting computation to ρ_2^{i-1} and obtain a new computation ρ_2^i. Then we can define $\rho^2 = \rho_2^t$. Putting things together, we obtain

$$\rho = \rho_1.\rho_2 = c \to^* \hat{c}_1 \to^* c_1.$$

By the maintained invariants, we get that c_1 is a permutation of c. All contributors took one transition along a cycle. Hence, the number of contributors in a certain state in c and c_1 are equal. For each s we have: $|c[s]| = |c_1[s]|$. Moreover, since π is a cycle, we get $\pi_L(c_1) = a = \pi_L(c)$ and $\pi_D(c_1) = a = \pi_D(c)$. Hence, ρ is a *balanced* computation and can be applied again to c_1. Since there are only finitely many permutations of c, applying ρ repeatedly will eventually yield a computation $c \to^* e \to_{sat}^+ e$ and hence, a saturated cycle. □

B.3.13 Proof of Lemma 8.65

Proof. It is left to show that the expression $Writes_{SCC}(X)$ can be evaluated in time $O(d \cdot (c^2 + d^2 \cdot l^2))$. By definition, $Writes_{SCC}(X) = Writes(SCCdcmp_T(X))$. We first compute the SCC decomposition $SCCdcmp_T(X)$. To this end, we need to construct the corresponding graph $\mathcal{G}_T(X)$. To obtain it, we iterate over the transitions in δ_C. If the current transition is a read within X or a write, we keep it as an edge. Hence, we need $O(|\delta_C|) = O(d \cdot c^2)$ time for the construction. Note that a look-up in X can be performed in constant time if we assume that X is a bit-vector with $X(b) = 1$ if and only if $b \in X$.

Now we can apply Tarjan's algorithm to obtain the strongly connected components $(T_1, \ldots, T_\ell)$ of $\mathcal{G}_T(X)$. Since the algorithm runs in time linear in the number of edges and vertices, it takes time $O(c + |\delta_C|) = O(d \cdot c^2)$. We obtain the X-SCC decomposition $SCCdcmp_T(X) = (T_1, \ldots, T_\ell)$.

It is left to compute the set of writes $Writes(T_1, \ldots, T_\ell)$. First, we focus on $Writes_C(T_1, \ldots, T_\ell)$. To compute the set, we iterate over all transitions in δ_C.

If the current transition is a write between two states p, p' belonging to the same set T_i, we add the corresponding symbol to $Writes_C(T_1, \dots, T_\ell)$. We need $O(|\delta_C|) = O(d \cdot c^2)$ time for the iteration. Note that we can perform the check whether p and p' lie in the same set T_i again in constant time. Summing up, we needed $O(d \cdot c^2)$ time so far.

For computing $Writes_L(T_1, \dots, T_\ell)$, we first need to construct the automaton $P_{L'}$. The states $Q_L \times D$ can be added in time $O(d \cdot l)$. The transitions of $P_{L'}$ are obtained by an iteration over δ_L. If the current transition is a write, $s \xrightarrow{!b'}_L s'$, then we add d many transitions: $(s, b) \xrightarrow{!b'}_{L'} (s, b')$, one for each $b \in D$. If the transition is a read of a symbol b, we test whether $b \in Writes_C(T_1, \dots, T_\ell)$ and add the single transition $(s, b) \xrightarrow{?b}_{L'} (s', b)$. Adding these transitions takes time $O(d \cdot |\delta_L|) = O(d^2 \cdot l^2)$ where the additional factor d appears since we add d many transitions in the case of a write. The ε-transitions in $P_{L'}$ can be added in time $O(d^2 \cdot l)$: we iterate over each symbol $b' \in Writes_C(T_1, \dots, T_\ell)$ and add $d \cdot l$ many transitions $(s, b) \xrightarrow{\varepsilon}_{L'} (s, b')$, one for each pair (s, b). Hence, we constructed the automaton $P_{L'}$ in time $O(d^2 \cdot l^2)$. Note that this limits the size of $\delta_{L'}$ to $O(d^2 \cdot l^2)$.

To identify the elements in the set $Writes_L(T_1, \dots, T_\ell)$, we iterate over all $b \in D$ and test for each, whether it occurs as a write $!b$ on a cycle from (q, a) to (q, a) in $P_{L'}$. The test can be reduced to a non-emptiness problem. To this end, let $P_{L'}(q, a)$ be the automaton $P_{L'}$ with (q, a) as initial and final state. Then, $b \in Writes_L(T_1, \dots, T_\ell)$ if and only if

$$OP(D)^*.!b.OP(D)^* \cap L(P_{L'}(q, a)) \neq \emptyset.$$

Since the corresponding automaton for $OP(D)^*.!b.OP(D)^*$ has a constant number of states, building the product and deciding non-emptiness can be done in time $O(|\delta_{L'}|) = O(d^2 \cdot l^2)$. Since the above non-emptiness test has to be executed for each $b \in D$, we get a total time of $O(d^3 \cdot l^2)$ to construct the set of writes provided by the leader, $Writes_L(T_1, \dots, T_\ell)$.

Putting the sets $Writes_C(T_1, \dots, T_\ell)$ and $Writes_L(T_1, \dots, T_\ell)$ together, we obtain the complete set of writes, $Writes(T_1, \dots, T_\ell) = Writes_{SCC}(X)$. Adding up the complexities, we need $O(d \cdot (c^2 + d^2 \cdot l^2))$ time for the evaluation. □

B.3.14 Restriction to Increasing Prefixes

We prove that for solving LCL, we can restrict to increasing prefix computations. To this end, we first need to generalize Lemma 8.51.

Lemma B.19. *Let $q \in Q_L$ and $a \in D$. There is an initialized finite computation $c^0 \to^* c \to^+_{sat} c$ with $\pi_L(c) = q$ and $\pi_D(c) = a$ if and only if there is an initialized finite computation $d^0 \to^*_{inc} d \to^+_{sat} d$ with $\pi_L(d) = q$ and $\pi_D(d) = a$.*

Proof. One direction is trivial. For the other direction, let a computation $c^0 \to^* c \to^+_{sat} c$ with $\pi_L(c) = q$ and $\pi_D(c) = a$ be given. Let $\rho = c^0 \to^* c$ be the

prefix. If ρ is increasing, we are done. Otherwise, we construct a new prefix ρ_{inc} that behaves like ρ but does not delete contributor states.

Let $\rho = c^0 \to_S c^1 \to_S \cdots \to_S c^\ell$. Then, there are configurations c^i and c^{i+1} in ρ such that $\pi_C(c^{i+1})$ does not contain $\pi_C(c^i)$. This means, there is a state $p \in \pi_C(c^i) \setminus \pi_C(c^{i+1})$. This state gets lost by performing the transition $c^i \to c^{i+1}$. Phrased differently, there is only one contributor P with current state p which does a transition to another state.

We apply the copycat lemma to get an additional contributor P^c that mimics P. It copies every move of P. Once P^c reaches state p, it keeps staying in the state. With the new contributor, the state does not get deleted and is preserved throughout the computation.

We introduce such an additional contributor for each state p that is deleted along ρ. Hence, we obtain an increasing computation $\rho_{inc} = d^0 \to^*_{inc} d$ with $\pi_C(d) \supseteq \pi_C(c)$. Leader and memory act the same way as before. We obtain $\pi_L(d) = \pi_L(c)$ and $\pi_D(d) = \pi_D(c)$.

Note that ρ_{inc} is obtained from ρ by only adding contributors. This means that d contains at least as many contributors in a particular state p as c. We have $\#_C(d, p) \geq \#_C(c, p)$ for each state $p \in Q_C$. We can therefore simulate the saturated cycle $c \to^+_{sat} c$ on the larger configuration d. Leader and memory act as before. Whenever there is a contributor in a certain state p acting in $c \to^+_{sat} c$, we can mimic the move. Hence, we get a cycle $d \to^+_{sat} d$. Note that the cycle is indeed saturated since $\pi_C(d) \supseteq \pi_C(c)$. □

The above lemma allows for splitting LCL along reachable interfaces. Technically, we obtain a generalization of Lemma 8.53.

Lemma B.20. *Let $q \in Q_L$ and $a \in D$. There is a computation $c^0 \to^* c \to^+_{sat} c$ with $\pi_L(c) = q$ and $\pi_D(c) = a$ if and only if there is a reachable interface $I = (q, a, T)$ and a saturated cycle $f \to^+_{sat} f$ with $I(f)$.*

Proof. Let the computation $c^0 \to^* c \to^+_{sat} c$ with $\pi_L(c) = q$ and $\pi_D(c) = a$ be given. By Lemma B.19, we obtain a computation $f^0 \to^*_{inc} f \to^+_{sat} f$ with $\pi_L(f) = q$ and $\pi_D(f) = a$. Set $T = \pi_C(f)$. Then, the interface $I = (q, a, T)$ is reachable and we clearly have $I(f)$.

For the other direction, assume $I = (q, a, T)$ is a reachable interface and there is a saturated cycle $f \to^+_{sat} f$ with $I(f)$. Since I is reachable, we obtain an increasing prefix $d^0 \to^*_{inc} d$ with $I(d)$. Now we can apply Lemma 8.53. It yields a computation $c^0 \to^* c \to^+_{sat} c$ with $I(c)$. This completes the proof. □

B.3.15 Proof of Lemma 8.68

We give two algorithms for computing the set of reachable interfaces. They are based on the algorithms for LCR and admit the same running times.

Parameterization by Contributor We first modify Algorithm 8.2. The modification relies on the following characterization which generalizes the correctness criterion of the algorithm, stated in Lemma B.18.

Lemma B.21. *Let $I = (q, a, S) \in IF$. Then, I is reachable if and only if $(q, a) \in T[S]$.*

Proof. First assume that we have a reachable interface $I = (q, a, S)$. Then we obtain an increasing prefix $c^0 \to^*_{inc} c$ with $I(c)$. By Lemma B.18, we obtain a path $v^0 \to^*_E (q, a, S)$ in $\mathcal{G}$. Note that $S = \pi_C(c)$ is preserved since the prefix is increasing. This can be observed from the proof of the lemma. By definition of the table T, we get that $(q, a) \in T[S]$.

Now assume that $(q, a) \in T[S]$. By definition, we obtain a path of the form $v^0 \to^*_E (q, a, S)$ in $\mathcal{G}$. Again, by following the proof of Lemma B.18, we get a prefix computation $c^0 \to^*_{inc} c$ with $I(c)$ which means that I is reachable. □

The lemma shows that the set of reachable interfaces can be determined once the table T is computed. In fact, it is given by

$$\mathcal{I} = \{(q, a, S) \mid (q, a) \in T[S]\}.$$

Since Algorithm 8.2 is basically a table-filling algorithm, we can let it compute each entry of $T[S]$. Then, we add a step which outputs the set $\mathcal{I}$. The complexity is as before, we only add an output step: $O(2^c \cdot c^4 \cdot d^2 \cdot l^2)$.

Parameterization by Domain and Leader We elaborate on the idea of how to modify Algorithm 8.1 so that it computes the set of reachable interfaces. To this end, we first need to determine which interfaces a (short) witness yields.

Definition B.22. Let $x = (w, q, \sigma) \in Wit$ be a (short) witness with word $w = (q_1, a_1) \dots (q_n, a_n)$ and let β be a first-write sequence with $|\beta| = ord(x) = k$ and $\text{Valid}_\beta(x)$ (or $\text{Valid}^{sh}_\beta(x)$). Recall the notion of a full expression from Definition B.11. An interface $I = (q, a, S)$ is *associated to the witness x* if the following two conditions are satisfied.

(1) The symbol a is the last written symbol that occurs in a computation along x. It is either a_n or a first write from β, we have $a \in \{a_n, \beta_1, \dots, \beta_k\}$.

(2) The set S contains all states of the contributor that are reachable along a computation of the witness. We have

$$S = \{p \in Q_C \mid \pi_{RD}(Trace_C(p)) \cap FullExpr(x, \beta).\Gamma^*_{n+1} \neq \emptyset\},$$

where $\Gamma_{n+1} = Loop(q, \{b_i \mid i \in [1..k]\}) \cup \{b_i \mid i \in [1..k]\}$.

Given a witness and a first-write sequence, we can clearly compute the associated interfaces in polynomial time. But they do not immediately correspond to the reachable interfaces of the underlying system. In fact, they correspond to the maximal interfaces.

Definition B.23. A reachable interface $I = (q, a, S)$ is called *maximal* if the set S is inclusion maximal among all reachable interfaces (q, a, S').

We can clearly restate Lemma B.20 in terms of maximal reachable interfaces. Moreover, Algorithm 8.4 also works with this kind of interfaces. Hence, we need to argue that associated interfaces correspond to associated interfaces since the latter can be computed efficiently along the above definition.

Lemma B.24. *Let $I = (q, a, S) \in IF$. Then, I is a maximal reachable interface if and only if there is a (short) witness $x = (w, q, \sigma)$ and a first-write sequence β such that* $\mathrm{Valid}_\beta(x)$ *(or* $\mathrm{Valid}^{sh}_\beta(x)$*) and I is associated to x.*

We omit the proof of the criterion. It can be obtained by following the proofs of Lemma 8.20 and Lemma 8.26. It is left to change Algorithm 8.1 so that it outputs the associated interfaces. The algorithm is a table-filling for $T[\beta, x]$ where x is a short witness and β is a first-write sequence with $ord(x) = \beta$. We let the algorithm compute each entry of the table. Then, for each entry $T[\beta, x]$ with $\mathrm{Valid}^{sh}_\beta(x)$ we compute the associated interfaces. After computing these for each corresponding entry of the table, we output them. The complexity does not change since the computation of the associated interfaces takes only polynomial time.

B.3.16 Correctness of Algorithm 8.4

The correctness of Algorithm 8.4 follows from Lemma B.20 and from the correctness of the modified algorithms for LCR. Indeed, if the algorithm returns *true*, it has found a reachable interface $I = (q, a, T)$ with $q \in Q_F$ and a saturated cycle $f \rightarrow_{sat} f$ with $I(f)$. This witnesses a live computation in q.

If a live computation in the state $q \in Q_F$ exists, we can witness it by finding a reachable interface $I = (q, a, T)$ and a saturated cycle $f \rightarrow_{sat} f$ with $I(f)$. Since the algorithm iterates over all reachable interfaces and tests the existence of a cycle with each, we eventually find the corresponding interface and cycle. The algorithm then returns *true*.

B.4 Proofs for Chapter 9

B.4.1 Proof of Lemma 9.16

Proof. Let $V \subseteq WR$ be non-empty. We have to prove two directions. To this end, first assume $T[V] = 1$. We show that there is an element $v \in V$ such that $\mathcal{G}_{loc}[V \setminus \{v\}, v]$ and $\mathcal{G}_{mm}[V \setminus \{v\}, v]$ are both acyclic and $T[V \setminus \{v\}, v] = 1$.

Since $T[V] = 1$, there is a snapshot order $tw[V] = t[V] \cup r[V]$ with a total order $t[V]$ on V. Moreover, the snapshot order satisfies that the graphs

$$\mathcal{G}_{loc}(tw[V]) = (O, po\text{-}loc \cup rf \cup tw[V] \cup cf[V]),$$
$$\mathcal{G}_{mm}(tw[V]) = (O, po\text{-}mm \cup rf\text{-}mm \cup tw[V] \cup cf[V])$$

are both acyclic. We extract the suitable write event v. Since $t[V]$ is total on V, there is a unique minimal element $v \in V$ according to the order. We set $V' = V \setminus \{v\}$ and show the following three facts:

(1) $\mathcal{G}_{loc}[V', v]$ is a subgraph of $\mathcal{G}_{loc}(tw[V])$,

(2) $\mathcal{G}_{mm}[V', v]$ is a subgraph of $\mathcal{G}_{mm}(tw[V])$, and

(3) $T[V'] = 1$.

Since $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V])$ are acyclic by assumption, any subgraph of these are as well. Hence, once the three facts are proven, we can conclude that v is indeed the element we are looking for.

We begin by proving (1). To this end, we show that each edge of the coherence graph of V' and v

$$\mathcal{G}_{loc}[V', v] = (O, po\text{-}loc \cup rf \cup r[V', v] \cup cf[V', v])$$

is already present in the graph $\mathcal{G}_{loc}(tw[V])$. Since *po-loc* and *rf* are already present in $\mathcal{G}_{loc}(tw[V])$, we need to show that the edges of the two relations $r[V', v]$ and $cf[V', v]$ are also in the graph.

By definition, $r[V', v] = r[V] \cup \{(v, w) \mid w \in V'\}$. Since v was selected to be the minimal element of $t[V]$ on V and $t[V]$ is total, we get that each edge (v, w) with $w \in V'$ is also contained in $t[V]$. Hence, we can deduce

$$r[V', v] \subseteq r[V] \cup t[V] = tw[V].$$

For the edges of $cf[V', v]$ we then obtain

$$cf[V', v] = rf^{-1} \circ \bigcup_{x \in Var} r[V', v]_x \subseteq rf^{-1} \circ \bigcup_{x \in Var} tw[V]_x = cf[V],$$

showing that $\mathcal{G}_{loc}[V', v]$ is a subgraph of $\mathcal{G}_{loc}(tw[V])$.

The proof of (2) follows from (1). We have to show that each edge of

$$\mathcal{G}_{mm}[V', v] = (O, po\text{-}mm \cup rf\text{-}mm \cup r[V', v] \cup cf[V', v])$$

is contained in $\mathcal{G}_{mm}(tw[V])$. The edges of *po-mm* and *rf-mm* are already present in the graph. Since $r[V', v] \subseteq tw[V]$ and $cf[V', v] \subseteq cf[V]$ hold by (1), we get that $\mathcal{G}_{mm}[V', v]$ is a proper subgraph of $\mathcal{G}_{mm}(tw[V])$.

It is left to prove (3). To this end, we construct a snapshot order $tw[V']$ on V' such that $\mathcal{G}_{loc}(tw[V'])$ is a subgraph of $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V'])$ is a subgraph of $\mathcal{G}_{mm}(tw[V])$. This shows that $T[V'] = 1$ since the two latter graphs are acyclic by assumption. We construct $tw[V']$ as follows. Set

$$tw[V'] = t[V'] \cup r[V'],$$

where $r[V'] = \{(\overline{w}, w) \mid \overline{w} \in \overline{V'}, w \in V'\}$ and $t[V'] = t[V] \cap (V' \times V')$ is the restriction of $t[V]$ to the set V'. Note that $t[V']$ is total on V'. Hence, $tw[V']$ is indeed a proper snapshot order.

By definition, we get that $t[V'] \subseteq t[V]$. Now consider an edge $(\overline{w}, w)$ from $r[V']$ with $\overline{w} \in \overline{V'}$ and $w \in V'$. There are two cases: (1) For $\overline{w} = v$, the edge (v, w) is already contained in $t[V]$ since v was chosen to be $t[V]$-minimal and $t[V]$ is total on V. (2) For $\overline{w} \neq v$, we get that $\overline{w} \in \overline{V}$. Hence, the edge $(\overline{w}, w)$ is already contained in $r[V]$. Putting the cases together, we obtain:

$$tw[V'] = t[V'] \cup r[V'] \subseteq t[V] \cup r[V] = tw[V],$$
$$cf[V'] = rf^{-1} \circ \bigcup_{x \in Var} tw[V']_x \subseteq rf^{-1} \circ \bigcup_{x \in Var} tw[V]_x = cf[V].$$

From these inclusions, we immediately obtain that $\mathcal{G}_{loc}(tw[V'])$ is a subgraph of $\mathcal{G}_{loc}(tw[V])$ and that $\mathcal{G}_{mm}(tw[V'])$ is a subgraph of $\mathcal{G}_{mm}(tw[V])$.

For the other direction of the proof, assume the existence a write event $v \in V$ such that the coherence graphs $\mathcal{G}_{loc}[V', v]$ and $\mathcal{G}_{mm}[V', v]$ are acyclic and such that $T[V'] = 1$. Here, $V' = V \setminus \{v\}$. In order to show that $T[V] = 1$, we need to construct a snapshot order $tw[V]$ on V such that the two graphs $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V])$ are both acyclic.

By the assumption $T[V'] = 1$, there is a snapshot order $tw[V']$ on V'. It is given by $tw[V'] = t[V'] \cup r[V']$ and it satisfies that the graphs

$$\mathcal{G}_{loc}(tw[V']) = (O, po\text{-}loc \cup rf \cup tw[V'] \cup cf[V']),$$
$$\mathcal{G}_{mm}(tw[V']) = (O, po\text{-}loc \cup rf\text{-}mm \cup tw[V'] \cup cf[V'])$$

are acyclic. We extend the order $t[V']$ by adding v as new minimal element. Define $t[V] = t[V'] \cup \{(v, w) \mid w \in V'\}$. Then, $t[V]$ is a total order on V. Thus, the order defined by

$$tw[V] = t[V] \cup r[V]$$

with relation $r[V] = \{(\overline{w}, w) \mid \overline{w} \in \overline{V}, w \in V\}$ is a snapshot order on V.

We show the acyclicity of $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V])$ in two steps. First, we define the two intermediary graphs

$$\mathcal{J}_{loc}[V, v] \text{ and } \mathcal{J}_{mm}[V, v]$$

and show that $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V])$ are subgraphs of theses. In the second step we prove that $\mathcal{J}_{loc}[V, v]$ and $\mathcal{J}_{mm}[V, v]$ are acyclic. In fact, we show that a cycle in one of the two graphs would induce a cycle in one of the coherence graphs $\mathcal{G}_{loc}[V', v]$ and $\mathcal{G}_{mm}[V', v]$ or in one of the graphs $\mathcal{G}_{loc}(tw[V'])$ and $\mathcal{G}_{mm}(tw[V'])$, which are all acyclic by assumption. The acyclicity of $\mathcal{G}_{loc}(tw[V])$ and $\mathcal{G}_{mm}(tw[V])$ then follows and we obtain $T[V] = 1$.

We begin with the first step. Before we define the intermediary graphs, we need two new relations depending on the fact that v is the new minimal element of V. We define them as follows:

$$inc(v) = \{(\overline{w}, v) \mid \overline{w} \in \overline{V}\} \text{ and } cfinc(v) = rf^{-1} \circ \bigcup_{x \in Var} inc(v)_x.$$

Note that a pair (r, v) is in $cfinc(v)$ if r is a read event with a write event $\overline{w} \in \overline{V}$ such that $(\overline{w}, r) \in rf$, $(\overline{w}, v) \in inc(v)$, and $\overline{w}$ and v write to the same variable.

The graphs $\mathcal{J}_{loc}[V, v]$ and $\mathcal{J}_{mm}[V, v]$ are now defined as follows:

$$\mathcal{J}_{loc}[V, v] = (O, po\text{-}loc \cup rf \cup tw[V'] \cup inc(v) \cup cf[V'] \cup cfinc(v)),$$
$$\mathcal{J}_{mm}[V, v] = (O, po\text{-}mm \cup rf\text{-}mm \cup tw[V'] \cup inc(v) \cup cf[V'] \cup cfinc(v)).$$

We show that $\mathcal{G}_{loc}(tw[V])$ is a subgraph of $\mathcal{J}_{loc}[V, v]$. First note that the edges of *po-loc* and *rf* are already present in $\mathcal{J}_{loc}[V, v]$. It is left to argue that $tw[V]$ and $cf[V]$ are included in the edges of $\mathcal{J}_{loc}[V, v]$ as well. To this end, consider the following inclusion:

$$t[V] = t[V'] \cup \{(v, w) \mid w \in V'\} \subseteq t[V'] \cup r[V'] = tw[V'].$$

The first equality is the definition of $t[V]$. The inclusion holds since $v \in \overline{V'}$ and thus $\{(v, w) \mid w \in V'\} \subseteq r[V']$. Relation $r[V]$ is embedded as follows:

$$\begin{aligned} r[V] &= \{(\overline{w}, w) \mid \overline{w} \in \overline{V}, w \in V\} \\ &= \{(\overline{w}, w) \mid \overline{w} \in \overline{V}, w \in V'\} \cup \{(\overline{w}, v) \mid \overline{w} \in \overline{V}\} \\ &\subseteq r[V'] \cup inc(v). \end{aligned}$$

The latter inclusion holds due to the fact that $\overline{V} \subseteq \overline{V'}$. Combining the above inclusions then yields $tw[V] \subseteq tw[V'] \cup inc(v)$. For the conflict relation, we consequently obtain that

$$\begin{aligned} cf[V] &= rf^{-1} \circ \bigcup_{x \in Var} tw[V]_x \\ &\subseteq rf^{-1} \circ \bigcup_{x \in Var} (tw[V'] \cup inc(v))_x \\ &= rf^{-1} \circ \bigcup_{x \in Var} (tw[V']_x \cup inc(v)_x) \\ &= \left(rf^{-1} \circ \bigcup_{x \in Var} tw[V']_x\right) \cup \left(rf^{-1} \circ \bigcup_{x \in Var} inc(v)_x\right) \\ &= cf[V'] \cup cfinc(v). \end{aligned}$$

Hence, all edges of $\mathcal{G}_{loc}(tw[V])$ are present in $\mathcal{J}_{loc}[V, v]$ which proves the subgraph relation. The fact that $\mathcal{G}_{mm}(tw[V])$ is a subgraph of $\mathcal{J}_{mm}[V, v]$ follows

easily from the above observations. Since *po-mm* and *rf-mm* are present in $\mathcal{J}_{mm}[V, v]$ and $tw[V] \subseteq tw[V'] \cup inc(v)$ as well as $cf[V] \subseteq cf[V'] \cup cfinc(v)$ hold independently from the considered graph, we obtain the subgraph relation.

In the second step, we show the acyclicity of $\mathcal{J}_{loc}[V, v]$ and $\mathcal{J}_{mm}[V, v]$. We focus on $\mathcal{J}_{loc}[V, v]$ since the proof for $\mathcal{J}_{mm}[V, v]$ is similar. Assume there is a cycle C in $\mathcal{J}_{loc}[V, v]$. If C does neither contain an edge from $inc(v)$ nor from $cfinc(v)$, the cycle has only edges over $po\text{-}loc \cup rf \cup tw[V'] \cup cf[V']$. Hence, C is a cycle in $\mathcal{G}_{loc}(tw[V'])$ which is a contradiction since the graph is acyclic. Therefore, C goes through at least one edge from $inc(v)$ or $cfinc(v)$. In both cases, this means that C passes through the write event v. We may think of C as a cycle that starts and ends in v: C is of the form

$$C = e_0.e_1 \dots e_\ell$$

with e_i edges and $e_0 = (v, w_1)$, $e_\ell = (w_\ell, v)$ for events $w_1, w_\ell \in O$. Moreover, we assume that C is short. The write event v is only visited once. Otherwise, we would clearly get a shorter cycle.

From C, we show how to construct a cycle $\hat{C}$ in $\mathcal{G}_{loc}[V', v]$ which contradicts the assumption that the coherence graphs are acyclic. To this end, we induct over the edges of C and construct $\hat{C}$ while keeping the invariant that all constructed edges of $\hat{C}$ are in the coherence graph $\mathcal{G}_{loc}[V', v]$.

Initially, $\hat{C}$ does not have any edges. The induction step is as follows. Assume we have already constructed a part of $\hat{C}$ while iterating to the i-th edge e of C. We get the following case distinction, based upon the type of e:

- If e is an edge in $po\text{-}loc \cup rf$. Then e is also present in $\mathcal{G}_{loc}[V', v]$ and we can add it to $\hat{C}$ by setting: $\hat{C} = \hat{C}.e$.

- If e is an edge in $tw[V']$ we get two subcases: (1) If e is in $t[V']$. Then, $e = (w, w')$, where $w, w' \in V'$ are write events. There is an edge $(v, w') \in r[V', v]$ by definition. We delete the content of $\hat{C}$ and start a new cycle with this edge: $\hat{C} = (v, w')$. It lies in $\mathcal{G}_{loc}[V', v]$.

  (2) If e is an edge in $r[V']$, then $e = (\overline{w}, w)$ where $\overline{w} \in \overline{V'}$ and $w \in V'$. The edge then also lies in $r[V', v]$ and thus in $\mathcal{G}_{loc}[V', v]$. We add it to $\hat{C}$ by setting $\hat{C} = \hat{C}.e$.

- If e is an edge in $inc(v)$,then e is of the form $(\overline{w}, v)$ with $\overline{w} \in \overline{V}$. Thus, e lies in $r[V', v]$ and therefore in $\mathcal{G}_{loc}[V', v]$. We add the edge by $\hat{C} = \hat{C}.e$.

- If e is an edge in $cf[V']$. Then $e = (r, w)$, where r is a read event and $w \in V'$ is a write event. There is an edge $(v, w) \in r[V', v]$ and thus in $\mathcal{G}_{loc}[V', v]$. We delete $\hat{C}$ and start a new cycle via $\hat{C} = (v, w)$.

- If e is an edge in $\mathit{cfinc}(v)$. Then e lies in $\mathit{cf}[V', v]$ since $\mathit{inc}(v) \subseteq r[V', v]$ and we have

$$\mathit{cfinc}(v) = \mathit{rf}^{-1} \circ \bigcup_{x \in \mathit{Var}} \mathit{inc}(v)_x \subseteq \mathit{rf}^{-1} \circ \bigcup_{x \in \mathit{Var}} r[V', v]_x = \mathit{cf}[V', v].$$

  Hence, e can be added by $\hat{C} = \hat{C}.e$.

In the construction, the first edge of $\hat{C}$ always leaves the write event v, it is of the form (v, w) for some event $w \in O$. Moreover, the edges in $\mathit{inc}(v)$ and $\mathit{cfinc}(v)$ are the only edges in $\mathcal{J}_{\mathit{loc}}[V, v]$ that are incoming for v. Such an edge is always the last edge of C and does never get deleted during the construction of $\hat{C}$. Note that we assumed the existence of such an edge. Hence, by construction $\hat{C}$ is a non-empty cycle in $\mathcal{G}_{\mathit{loc}}[V', v]$ that starts and ends in v. This contradicts the acyclicity of the coherence graph. Altogether, we obtain that the graph $\mathcal{J}_{\mathit{loc}}[V, v]$ is acyclic. □

B.4.2 Proof of Lemma 9.17

Proof. We show how $\mathcal{G}_{\mathit{loc}}[V, v]$ is constructed and tested for acyclicity. The proof for $\mathcal{G}_{\mathit{mm}}[V, v]$ is similar. First, we focus on the construction of $\mathcal{G}_{\mathit{loc}}[V, v]$. Recall the definition of the coherence graph:

$$\mathcal{G}_{\mathit{loc}}[V, v] = (O, \mathit{po\text{-}loc} \cup \mathit{rf} \cup r[V, v] \cup \mathit{cf}[V, v]).$$

Constructing the vertices can clearly be done in time $O(n)$, as $n = |O|$. The edges of $\mathit{po\text{-}loc} \cup \mathit{rf}$ are part of the given history. Hence, we can iterate over these edges and add them to the graph. Since $\mathit{po\text{-}loc} \cup \mathit{rf}$ is a relation in $O \times O$, this takes time at most $O(n^2)$. Next, we construct the edges of the relation

$$r[V, v] = \{(\overline{w}, w) \mid \overline{w} \in \overline{V \cup \{v\}}, w \in V \cup \{v\}\} \cup \{(v, w) \mid w \in V\}.$$

The latter part is simple to construct: we add an edge (v, w) for each $w \in V$. These are at most $O(k)$ many edges since $|V| \leq k$ and consequently, it takes at most $O(k)$ time adding them. For constructing the former relation, we iterate over $\overline{w} \in \overline{V \cup \{v\}}$ and $w \in V \cup \{v\}$ and add the edge $(\overline{w}, w)$. This takes time at most $O(k^2) = O(k \cdot n)$ time. Hence, the relation $r[V, v]$ contains at most $O(k \cdot n)$ many edges and can be constructed in time $O(k \cdot n)$.

It is left to construct the conflict relation $\mathit{cf}[V, v] = \mathit{rf}^{-1} \circ \bigcup_{x \in \mathit{Var}} r[V, v]_x$. To this end, we first construct the relation rf^{-1} by turning around the edges stored in rf. Note that rf consists of at most $O(n)$ many of these since it contains exactly one edge for each read event. Hence, rf^{-1} can be constructed in time $O(n)$. The relations $r[V, v]_x$ can be constructed from $r[V, v]$. We iterate over the edges in $r[V, v]$ and put an edge (w, w') to the corresponding projection $r[V, v]_x$ if both write events w, w' write to variable x. This takes time at most $O(k \cdot n)$. The composition $\mathit{cf}[V, v] = \mathit{rf}^{-1} \circ \bigcup_{x \in \mathit{Var}} r[V, v]_x$ is then

obtained as follows. We iterate over all edges (r, w) in rf^{-1} and $(w', \hat{w})$ in one of the $r[V, v]_x$ and add $(r, \hat{w})$ to $cf[V, v]$ if $w = w'$. Since rf^{-1} contains at most $O(n)$ many edges and the union of the $r[V, v]_x$ contains at most $O(k \cdot n)$ many edges, constructing $cf[V, v]$ takes time $O(k \cdot n^2)$.

Hence, the graph $\mathcal{G}_{loc}[V, v]$ can be constructed in time $O(k \cdot n^2)$. It is left to show that cycles in $\mathcal{G}_{loc}[V, v]$ can be detected within the same amount of time. To this end, we apply Kahn's algorithm [226]. It finds a topological sorting for a given graph. Such a sorting only exists if the graph is acyclic. If this is not the case, the algorithm outputs an error. Kahn's algorithm runs in time linear in the vertices and edges. In our setting, it needs at most $O(n^2)$ time since we can have at most n^2 many edges. This lies within the bound $O(k \cdot n^2)$ and therefore finishes the proof. □

B.4.3 Proof of Lemma 9.21

Proof. First, we consider the structure of the acyclic graph $\mathcal{G}_{loc}^{ww}$ in more detail. Recall that $\mathcal{G}_{loc}^{ww} = (O, po\text{-}loc \cup rf \cup ww \cup fr)$. We show that $\mathcal{G}_{loc}^{ww}$ decomposes into a disjoint union of its projections to the variables:

$$\mathcal{G}_{loc}^{ww} = \bigcup_{x \in Var} \mathcal{G}_{loc}^{ww}(x),$$

where $\mathcal{G}_{loc}^{ww}(x) = (O(x), po\text{-}loc_x \cup rf_x \cup ww_x \cup fr_x)$ is the projection to variable x and the union is taken over vertices and edges.

It is clear that each projection $\mathcal{G}_{loc}^{ww}(x)$ is contained in $\mathcal{G}_{loc}^{ww}$ as $O(x) \subseteq O$ and each projected relation is a subset of the original relation. For the other inclusion, first note that the set of vertices O is contained in the union since we can write $O = \bigcup_{x \in Var} O(x)$. Phrased differently, each event in O refers to exactly one location. It is left to show that all edges of $\mathcal{G}_{loc}^{ww}$ are contained in the union. By definition, each of the relations *po-loc*, *rf*, *ww*, and *fr* only relates events to the same location. Hence, an edge (w, w') from one of the relations is an edge among events on a variable x and therefore contained in the graph $\mathcal{G}_{loc}^{ww}(x)$. The union is disjoint since there is no edge in $\mathcal{G}_{loc}^{ww}$ that involves events on different variables.

The order *tw* contains the store order *ww* and for each $x \in Var$, order ww_x is total on $WR(x)$. This implies that $tw_x = ww_x$. Hence, *tw* differs from *ww* by additional edges among write events on different variables. Formally, we can therefore write *tw* as a disjoint union:

$$tw = ww \cup ext,$$

where $ext = \{(w, w') \in tw \mid var(w) \neq var(w')\}$. This means that the graph $\mathcal{G}_{loc}^{tw} = (O, po\text{-}loc \cup rf \cup tw \cup fr)$ of interest has a structure similar to $\mathcal{G}_{loc}^{ww}$ but with edges connecting the projections:

$$\mathcal{G}_{loc}^{tw} = (O, ext) \cup \bigcup_{x \in Var} \mathcal{G}_{loc}^{ww}(x).$$

Now assume that there is a cycle C in the graph $\mathcal{G}_{loc}^{tw}$. We want to derive a contradiction. The cycle C takes the following form:

$$C = o_1 \xrightarrow{\pi_1} o_1' \xrightarrow{ext} o_2 \xrightarrow{\pi_2} o_2' \xrightarrow{ext} \ldots \xrightarrow{ext} o_\ell \xrightarrow{\pi_\ell} o_\ell',$$

where (1) o_i, o_i' are write events in a graph $\mathcal{G}_{loc}^{ww}(x_i)$ for a variable x_i, (2) $o_\ell' = o_1$, (3) each π_i is a path within $\mathcal{G}_{loc}^{ww}(x_i)$, and (4) each $o_i' \xrightarrow{ext} o_{i+1}$ is an edge in ext.

Since o_i and o_i' are write events on the same variable x_i and ww_{x_i} is a total order on these events, there is a relation between the two writes: either $(o_i, o_i') \in ww_{x_i}$ or $(o_i', o_i) \in ww_{x_i}$. In the latter case, we would immediately get a cycle in the graph $\mathcal{G}_{loc}^{ww}(x_i)$. Namely,

$$o_i' \xrightarrow{ww_{x_i}} o_i \xrightarrow{\pi_i} o_i'.$$

But as a subgraph of $\mathcal{G}_{loc}^{ww}$, the graph is acyclic and the cycle cannot appear. Hence, we get that $(o_i, o_i') \in ww_{x_i}$ for each i. Since ww_{x_i} is contained in tw, we have an edge $(o_i, o_i') \in tw$ for each i. Hence, we can shorten the cycle C to a cycle C^{tw} of the form:

$$C^{tw} = o_1 \xrightarrow{tw} o_1' \xrightarrow{tw} o_2 \xrightarrow{tw} o_2' \xrightarrow{tw} \ldots \xrightarrow{tw} o_\ell \xrightarrow{tw} o_\ell'.$$

Note that we used the fact $ext \subseteq tw$. The cycle C^{tw} contradicts the fact that tw is a strict order on WR. Hence, cycle C cannot exist and $\mathcal{G}_{loc}^{tw}$ is acyclic. □

B.4.4 Proof of Lemma 9.38

Proof. Let φ be satisfiable. We show that h_φ is SC-consistent. Since the formula is satisfiable, there is an evaluation

$$v : Var \to \{0, 1\}$$

that evaluates φ to 1. In order to prove that h_φ is SC-consistent, we need to construct a total order tw on the write events of h_φ such that $\mathcal{G}_{sc}$ is acyclic. Note that the acyclicity of $\mathcal{G}_{loc}$ is implied by this. Technically, we do not immediately show the acyclicity of the graph but we construct a topological sorting of the vertices which implies acyclicity of $\mathcal{G}_{sc}$.

First, we need to construct a total order tw on the write events. To this end, we extract an ordering on the literals of each clause. For any clause

$$C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i,$$

let $L(C_i, 1) = \{\ell_j^i \in C_i \mid v(\ell_j^i) = 1\}$ be the set of literals in C_i that evaluate to 1 under v. Since v is satisfying, we get that for each $i \in \{1, \ldots, m\}$, the set $L(C_i, 1)$ is non-empty. Similarly, we define $L(C_i, 0) = \{\ell_j^i \in C_i \mid v(\ell_j^i) = 0\}$.

We begin by constructing a partial order among the literals of the clauses. To this end, we define the following indices:

$$Nxt(j) = (j \mod 3) + 1 \text{ for } j = 1, 2, 3.$$

We first let all literals of a clause that evaluate to 0 be smaller than the literals that evaluate to 1 under v. Formally, we set $\ell < \ell'$ for each $\ell \in L(C_i, 0)$, $\ell' \in L(C_i, 1)$, and $i \in \{1, \ldots, m\}$. If we find that $|L(C_i, 0)| > 1$, there are two literals evaluating to 0. Note that it cannot be three since the formula is satisfied. In this case, let $L(C_i, 0) = \{\ell^i_j, \ell^i_{j'}\}$ where $j' = Nxt(j)$. Note that the literals in $L(C_i, 0)$ always have this form. Then we also set $\ell^i_j < \ell^i_{j'}$. The reason why we construct the order like this is that in a topological sorting of $\mathcal{G}_{sc}$, the read events of the literals will respect this order.

The total order: We construct the total order tw on the write events of h_φ. To this end, we consider the following two sets of threads:

$$First(Var) = \{T_0(x) \mid v(x) = 1\} \cup \{T_1(x) \mid v(x) = 0\},$$
$$Sec(Var) = \{T_0(x) \mid v(x) = 0\} \cup \{T_1(x) \mid v(x) = 1\}.$$

The set $First(Var)$ contains those threads that will write the complement evaluation of v into the variables. These threads have to run first in an interleaving/topological sorting. The variables then get overwritten by the threads of $Sec(Var)$. These write the correct evaluation v to the variables. We need further notation to construct tw. Let ℓ be a literal over $x \in Var$. We set

$$T_{neg}(\ell) = \begin{cases} T_0(\ell), \text{ if } \ell = x, \\ T_1(\ell), \text{ otherwise.} \end{cases}$$

This is the thread that stores 0 in ℓ. Similarly, we define a notation for the thread of h_φ that stores 1 in ℓ. It is given by:

$$T_{pos}(\ell) = \begin{cases} T_0(\ell), \text{ if } \ell = \neg x, \\ T_1(\ell), \text{ otherwise.} \end{cases}$$

We go on with the definition of further sets. Let C_i be a clause. $First(C_i)$ is the set of threads that write the complement of v into the literals. These are threads that write 0 to the literals which are evaluated to 1 under v and those that write 1 to literals that are evaluated to 0. Formally, we have

$$First(C_i) = \{T_{neg}(\ell) \mid \ell \in C_i, v(\ell) = 1\} \cup \{T_{pos}(\ell) \mid \ell \in C_i, v(\ell) = 0\}.$$

The idea is similar as above. In an interleaving of all events, the threads of $First(C_i)$ run first. The variables then get overwritten by the threads of $Sec(C_i)$. These forward the evaluation v of the variables to the literals:

$$Sec(C_i) = \{T_{pos}(\ell) \mid \ell \in C_i, v(\ell) = 1\} \cup \{T_{neg}(\ell) \mid \ell \in C_i, v(\ell) = 0\}.$$

The total order tw consists of several parts obtained from ordering the above sets. Let $Lin_{WR}(First(Var))$ be some total order on the write events of the threads occurring in $First(Var)$, based on some assumed order on the variables. Similarly, let $Lin_{WR}(Sec(Var))$ be a total order on the write events of the threads in $Sec(Var)$. Also the second order respects the assumed order on the variables. Further, let $Lin_{WR}(First(C_i))$ be a total order on the write events of threads in $First(C_i)$ that respects the above order on literals. Similarly, let $Lin_{WR}(Sec(C_i))$ be a total order on write events from the threads of $Sec(C_i)$.

To finally define tw, we use a suitable *product operator*. Let t and r be two total orders. Then $t.r$ is the total order obtained from ordering the elements of t to be smaller than the elements of r while preserving the orders t and r, meaning $t, r \subseteq t.r$. The total order tw on all write events of h_φ is then given by the product $tw = tw_{first}.tw_{sec}$, where

$$tw_{first} = Lin_{WR}(First(Var)).Lin_{WR}(First(C_1)) \dots Lin_{WR}(First(C_m)),$$
$$tw_{sec} = Lin_{WR}(Sec(Var)).Lin_{WR}(Sec(C_1)) \dots Lin_{WR}(Sec(C_m)).$$

Interleaving the events: We construct an interleaving / a topological sorting of all events following the total order tw. First, we store the complement evaluation $\bar{v}$ of v. This is achieved by scheduling $First(Var)$ in the beginning. We run these threads in the order given by $Lin_{WR}(First(Var))$. Then, we forward the evaluation $\bar{v}$ to the literals by running the threads in $First(C_i)$ for each i. Since these threads are guarded by read events, there is a read dependency. This means that the threads in $First(Var)$ need to provide the correct values for the read events in the threads in $First(C_i)$. But since the threads in $First(Var)$ actually store $\bar{v}$ in the variables and the threads in $First(C_i)$ demand it, they provide the correct values. Similarly, note that running the threads in $Sec(Var)$ will store the evaluation v in the variables and the threads in $Sec(C_i)$ will push it to the literals.

The total order tw provides the write events in such a way that the read events of the clause threads $T^1(C)$, $T^2(C)$, and $T^3(C)$ can be scheduled properly. Let $C = \ell_1 \vee \ell_2 \vee \ell_3$ be a clause. We distinguish three cases:

(1) If C is satisfied by all three literals, we have $v(\ell_i) = 1$ for $i = 1, 2, 3$. Then, under $\bar{v}$, we have $\bar{v}(\ell_i) = 0$ for $i = 1, 2, 3$. Since the write events in $First(C)$ evaluate the literals under $\bar{v}$, we can schedule the read events $rd(\ell_1, 0)$, $rd(\ell_2, 0)$, and $rd(\ell_3, 0)$ since the values are provided. Technically, we schedule these events after tw_{first} and before tw_{sec}. The remaining reads $rd(\ell_1, 1)$, $rd(\ell_2, 1)$, and $rd(\ell_3, 1)$ can then be scheduled after tw_{sec}.

(2) C is satisfied by two literals. Without loss of generality, we assume that $v(\ell_1) = 1$, $v(\ell_2) = 1$, and $v(\ell_3) = 0$. Then $\bar{v}(\ell_1) = 0$, $\bar{v}(\ell_2) = 0$ and $\bar{v}(\ell_3) = 1$. To schedule all the reads of the clause threads properly, we do the following. After tw_{first}, we schedule the reads in the following order

$rd(\ell_1, 0).rd(\ell_2, 0).rd(\ell_3, 1)$. Note that this conforms to program order and that all the values are provided by the write events in tw_{first}. After tw_{sec}, where the literals are evaluated according to v, we can then schedule the reads $rd(\ell_3, 0).rd(\ell_1, 1).rd(\ell_2, 1)$.

(3) C is satisfied by one literal. Let us assume $v(\ell_1) = 1$, $v(\ell_2) = 0$, and $v(\ell_3) = 0$. Then, we get $\bar{v}(\ell_1) = 0$, $\bar{v}(\ell_2) = 1$, and $\bar{v}(\ell_3) = 1$. In this case, we schedule the reads $rd(\ell_1, 0).rd(\ell_2, 1)$ immediately after tw_{first}. We cannot schedule $rd(\ell_3, 1)$ since it is blocked by the read $rd(\ell_2, 0)$ the value of which we have not provided yet.

For providing the value, consider $T_{neg}(\ell_2)$. The thread occurs in $Sec(C)$ and is scheduled within tw_{sec}. Immediately after it performed its write $wr(\ell_2, 0)$, we schedule the read $rd(\ell_2, 0)$ which was blocking. Since we did not change the content of variable ℓ_3 yet, we can schedule $rd(\ell_3, 1)$. Note that this fact relies on the order among literals defined above. We know that in $Lin_{WR}(Sec(C))$, the thread $T_{neg}(\ell_2)$ precedes $T_{neg}(\ell_3)$. After the described schedule, $T^2(C)$ and $T^3(C)$ are completely executed. After tw_{sec}, the ℓ_i store the evaluation under v. We can then schedule the remaining reads $rd(\ell_3, 0).rd(\ell_1, 1)$.

By constructing a schedule following these rules for each clause, we obtain a proper interleaving of all events in h_φ that only reads values that were written before accordingly. Such an interleaving is a topological sorting of $\mathcal{G}_{sc}$ which means that the graph is indeed acyclic and h_φ is SC-consistent.

For the other direction, assume that h_φ is SC-consistent. We show that φ is satisfiable. By definition, we obtain a total order tw on the write events of h_φ such that $\mathcal{G}_{sc}$ is acyclic. We construct an evaluation $v : X \to \{0, 1\}$ along this total order as follows:

$$v(x) = 1 \text{ if and only if } wr(x, 0) \xrightarrow{tw} wr(x, 1).$$

Hence, variable x admits the value that is written latest in tw. We show that the evaluation can be consistently extended to the literals. Indeed, for each literal ℓ of the formula, we have:

$$v(\ell) = 1 \text{ if and only if } wr(\ell, 0) \xrightarrow{tw} wr(\ell, 1).$$

To prove this, let ℓ be a literal evaluating to 1 under v. For ℓ evaluating to 0, the proof is similar. Towards a contradiction, suppose that

$$wr(\ell, 1) \xrightarrow{tw} wr(\ell, 0).$$

Without loss of generality, we assume that $\ell = x$. The proof for $\ell = \neg x$ is similar as well. Since $v(x) = 1$ as well, we get the tw-edge

$$wr(x, 0) \xrightarrow{tw} wr(x, 1).$$

This yields the following cycle in the graph $\mathcal{G}_{sc}$:

$$wr(\ell,1) \xrightarrow{tw} wr(\ell,0) \xrightarrow{po} rd(x,0) \xrightarrow{cf} wr(x,1) \xrightarrow{rf} rd(x,1) \xrightarrow{po} wr(\ell,1).$$

Hence, we get the edge $wr(\ell,0) \xrightarrow{tw} wr(\ell,1)$. This proves the above equivalence. Now we show that for each clause, there is at least one literal that evaluates to 1 under v. Assume the contrary, then there is a clause $C = \ell_1 \vee \ell_2 \vee \ell_3$ such that $v(\ell_i) = 0$ for $i = 1,2,3$. By the equivalence above, we obtain:

$$wr(\ell_i,1) \xrightarrow{tw} wr(\ell_i,0)$$

for each $i = 1,2,3$. From this, we can obtain the following cycle in $\mathcal{G}_{sc}$:

$$\begin{aligned} rd(\ell_1,0) &\xrightarrow{po} rd(\ell_2,1) \xrightarrow{cf} wr(\ell_2,0) \xrightarrow{rf} rd(\ell_2,0) \\ &\xrightarrow{po} rd(\ell_3,1) \xrightarrow{cf} wr(\ell_3,0) \xrightarrow{rf} rd(\ell_3,0) \\ &\xrightarrow{po} rd(\ell_1,1) \xrightarrow{cf} wr(\ell_1,0) \xrightarrow{rf} rd(\ell_1,0). \end{aligned}$$

Hence, the clauses are satisfied under v. This completes the proof. □

B.4.5 Proof of Lemma 9.40

Proof. First note that each total order tw on the write events of h_φ is also a total order on the write events of h'_φ and vice versa. The reason is that the write events of both histories are identical. In fact, h'_φ is obtained from h_φ by only adding read events. We fix a total order tw. Since we are treating two distinct histories in the following, we use $\mathcal{G}_{sc}(h_\varphi)$ and $\mathcal{G}_{sc}(h'_\varphi)$ to denote the graphs from the definition of consistency over the corresponding history.

Now we show the following. The order tw induces a cycle in $\mathcal{G}_{sc}(h_\varphi)$ if and only if it induces a cycle in $\mathcal{G}_{sc}(h'_\varphi)$. Note that this implies the desired equivalence: h_φ is SC-consistent if and only if h'_φ is SC-consistent.

Assume there is a cycle C in $\mathcal{G}_{sc}(h_\varphi)$. We show how to inductively construct a cycle C' in $\mathcal{G}_{sc}(h'_\varphi)$. Initially, C' does not have any edges. For the induction step, assume that the current edge of C that we mimic is called e. We make a case distinction depending on the type of e:

- If e is an rf-edge, then it also exists in $\mathcal{G}_{sc}(h'_\varphi)$. Assume e is of the form

  $$wr(x,v) \xrightarrow{rf} o = rd(x,v),$$

  for a variable x and a value $v \in \{0,1\}$ Then, o is either the first read event of a thread $T_v(\ell)$ for a literal ℓ or the second one. If it is the first one, the edge e exists in $\mathcal{G}_{sc}(h'_\varphi)$. Otherwise, e exists in the graph but o is in the thread $T'_v(\ell)$. In both cases, we append e to C'.

If e is not of the above form, it takes the following shape:

$$wr(\ell, v) \xrightarrow{rf} o = rd(\ell, v),$$

where ℓ is some literal, $v \in \{0, 1\}$ and o is a read event in a thread $T^i(C)$. Due to the construction, the same edge exists in the graph $\mathcal{G}_{sc}(h'_\varphi)$ and we can append it to C'.

- If e is an tw-edge, it also exists in $\mathcal{G}_{sc}(h'_\varphi)$ and we can immediately append it to C'.
- Assume e is an po-edge. In $\mathcal{G}_{sc}(h'_\varphi)$ all po-edges of $\mathcal{G}_{sc}(h_\varphi)$ exist except for one kind. Let e be of the form

  $$wr(\ell, v') \xrightarrow{po} o = rd(x, v),$$

  where x is a variable, ℓ a literal over x, $v, v' \in \{0, 1\}$ and o the second guarding read event in the thread $T_v(\ell)$. Then we mimic e by the path in $\mathcal{G}_{sc}(h'_\varphi)$

  $$wr(\ell, v') \xrightarrow{rf} rd(\ell, v') \xrightarrow{po} rd(x, v)$$

  which leads to the thread $T'_v(\ell)$. We append the path to C'.
- If e is an cf-edge, then we can also mimic it on $\mathcal{G}_{sc}(h'_\varphi)$. In the graph, all cf-edges of $\mathcal{G}_{sc}(h_\varphi)$ exist and we only need to argue for one type. If e is of the form

  $$o = rd(x, v) \xrightarrow{cf} wr(x, v'),$$

  where $v \neq v'$ and o is the second guarding read event in some thread $T_v(\ell)$. Since by induction, the path C' arrives at the read event $rd(x, v)$ in thread $T'_v(\ell)$ in this case, we can mimic e by

  $$rd(x, v) \xrightarrow{cf} wr(x, v')$$

  in $\mathcal{G}_{sc}(h'_\varphi)$. Note that the edge exists since the tw-edges and the inverted rf-edges carry over accordingly.

Altogether, we obtain a cycle C' in $\mathcal{G}_{sc}(h'_\varphi)$ which proves the first direction of the statement. We go on with the remaining direction.

For the other direction, assume that there is a cycle C in the graph $\mathcal{G}_{sc}(h'_\varphi)$. Similar to the above, we inductively construct a cycle C' in $\mathcal{G}_{sc}(h_\varphi)$ with a case distinction on the type of the current edge e of C. The individual cases

are almost the same and only one requires a different argument. Let e be an rf-edge of the following form

$$wr(\ell, v') \xrightarrow{rf} o = rd(\ell, v'),$$

where o is in the thread $T'_v(x)$. In this case, we also need to consider the next edge f of C. There are two cases

- If f is an po-edge of the form

  $$o = rd(\ell, v') \xrightarrow{po} rd(x, v),$$

  then we replace $e.f$ by the single po-edge

  $$wr(\ell, v') \xrightarrow{po} rd(x, v)$$

  which exists in $\mathcal{G}_{sc}(h_\varphi)$.

- If f is not an po-edge of the above form, then it is an cf-edge of the form

  $$rd(\ell, v') \xrightarrow{cf} wr(\ell, v)$$

  where $v' \neq v$. But this means that there is also an tw-edge in $\mathcal{G}_{sc}(h'_\varphi)$ of the following form

  $$wr(\ell, v') \xrightarrow{tw} wr(\ell, v).$$

  Since the tw-edge also exists in the graph $\mathcal{G}_{sc}(h_\varphi)$, we can add it to the path C' to mimic the path $e.f$.

Like above, this yields a cycle in $\mathcal{G}_{sc}(h_\varphi)$ and completes the proof. □