Bounded Treewidth, Multiple Context-Free Grammars, and Downward Closures

C. AISWARYA, Chennai Mathematical Institute and CNRS, ReLaX, IRL 2000, India
PASCAL BAUMANN, Max Planck Institute for Software Systems (MPI-SWS), Germany
PRAKASH SAIVASAN, The Institute of Mathematical Sciences, HBNI, and CNRS, ReLaX, IRL 2000, India
LIA SCHÜTZE, Max Planck Institute for Software Systems (MPI-SWS), Germany
GEORG ZETZSCHE, Max Planck Institute for Software Systems (MPI-SWS), Germany

The reachability problem in multi-pushdown automata (MPDA), or equivalently, interleaved Dyck reachability, has many applications in static analysis of recursive programs. An example is safety verification of multi-threaded recursive programs with shared memory. Since these problems are undecidable, the literature contains many decidable (and efficient) underapproximations of MPDA.

A uniform framework that captures many of these underapproximations is that of *bounded treewidth*: To each execution of the MPDA, we associate a graph; then we consider the subset of all graphs that have a treewidth at most k, for some constant k. In fact, bounding treewidth is a generic approach to obtain classes of systems with decidable reachability, even beyond MPDA underapproximations. The resulting systems are also called *MSO-definable bounded-treewidth systems*.

While bounded treewidth is a powerful tool for reachability and similar types of analysis, the word languages (i.e. action sequences corresponding to executions) of these systems remain far from understood. For the slight restriction of bounded *special treewidth*, or "bounded-stw" (which is equivalent to bounded treewidth on MPDA, and even includes all bounded-treewidth systems studied in the literature), this work reveals a connection with multiple context-free languages (MCFL), a concept from computational linguistics. We show that the word languages of MSO-definable bounded-stw systems are exactly the MCFL.

We exploit this connection to provide an optimal algorithm for computing *downward closures* for MSO-definable bounded-stw systems. Computing downward closures is a notoriously difficult task that has many applications in the verification of complex systems: As an example application, we show that in programs with dynamic spawning of MSO-definable bounded-stw processes, safety verification has the same complexity as in the case of processes with sequential recursive processes.

CCS Concepts: • **Theory of computation → Models of computation**.

Additional Key Words and Phrases: Treewidth, Verification, Monadic second-order logic, Multiple context-free grammar, Languages, Downward closures, Abstractions

1 Introduction

Reachability in multi-pushdown automata. We study the complexity of static analysis tasks for multi-threaded shared-memory (recursive) programs and related systems. Such programs consist of concurrent threads, each of which performs a recursive sequential computation. The threads have access to shared variables. Moreover, we assume that all program variables range over finite domains. Safety verification of such programs can be modeled as reachability in multi-pushdown automata (MPDA). These feature a finite set of control states (which model the shared memory) and have access to multiple pushdown stores (which model the call stacks of the individual threads).

For this reason, the reachability problem in MPDA has received a vast amount of attention in recent decades (see below). Another reason for this strong interest is that MPDA reachability is also

Authors' Contact Information: C. Aiswarya, Chennai Mathematical Institute and CNRS, ReLaX, IRL 2000, Chennai, India, aiswarya@cmi.ac.in; Pascal Baumann, Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany, pbaumann@mpi-sws.org; Prakash Saivasan, The Institute of Mathematical Sciences, HBNI, and CNRS, ReLaX, IRL 2000, Chennai, India, prakashs@imsc.res.in; Lia Schütze, Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany, lschuetze@mpi-sws.org; Georg Zetzsche, Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany, georg@mpi-sws.org.

equivalent to the problem of *interleaved Dyck reachability*, which can model many important verification tasks beyond safety, but in recursive sequential programs [61], such as structure-transmitted data-dependence [73], typestate analysis [75], alias analysis [24, 76], and taint analysis [8, 45].

Parameterized underapproximations. Since in full generality, reachability in MPDA is undecidable [71, 73], a popular approach to the above tasks is to consider *parameterized underapproximations*. Here, for some parameter, one considers only executions of the MPDA (resp. multi-threaded program) where that parameter is bounded by some $k \geq 1$. Imposing such bounds then often makes reachability decidable. An example is *context-bounding*, where one considers only those executions that switch between stacks (i.e. threads) at most k times [69]. This turned out to be a remarkably useful approach for finding bugs in practice, since empirically, buggy behavior often manifests in executions where such parameters are small [46, 66, 69]. This general approach inspired numerous underapproximations of MPDA. Each comes with its own tailor-made algorithms and results on expressiveness (i.e. which behaviors can be captured by the underapproximation). Examples include context-bounded [69], ordered [18], phase-bounded [55], and scope-bounded [9] MPDA.

Bounded treewidth. Given this plethora of underapproximations, an important contribution was the discovery of a powerful unifying approach: A seminal paper by Madhusudan and Parlato [63], together with a line of follow-up work [2, 31, 32, 57] revealed that most of the decidable underapproximations of MPDA (including the ones mentioned above) are *bounded-treewidth* [18, 55, 58, 69]. This means, the executions can be represented as graphs that have small tree decompositions. This implies that decidability of these underapproximations can be uniformly explained by Courcelle's Theorem [28], which says that among structures of bounded treewidth, satisfiability of any property definable in monadic second-order logic (MSO) is decidable in polynomial time.

In fact, the bounded-treewidth framework applies not only to MPDA, but can be viewed as a general framework that yields analysis algorithms in infinite-state systems: It also applies to systems with FIFO queues [2, 63] and even timed multi-pushdown systems [3–5]. For this reason, for infinite-state systems (whether they are MPDA or not) whose valid runs can be represented as bounded-treewidth graphs, we informally use the term *bounded-treewidth systems*.

Analyzing traces: Downward closures. However, while Courcelle's theorem yields algorithms for yes-or-no queries (such as safety), it is of limited use for closer analysis of its set of program traces (i.e. sequences of actions), also called its *(word) language*. An example task for which the above decision procedures fall short is the computation of downward closures. The *downward closure* of a language $L \subseteq \Sigma^*$ is the set of all (not necessarily contiguous) subsequences of members of L. By a fundamental result of Higman [44], the downward closure of *every* language is regular. For example, the non-regular language $L = \{a^n b^n \mid n \geq 0\}$ has the regular downward closure $\{a^n b^m \mid m, n \geq 0\}$. Unfortunately, computing an NFA for the downward closure of a language L (when given some representation of L) is a notoriously challenging task: Algorithms to compute downward closures (especially optimal ones) are usually non-trivial [12, 25, 41, 80, 81] and require deep insights into the structure of the set of runs of a system.

Downward closure computations are useful for analyzing systems that consist of several components, which are infinite-state systems themselves. In such situations, one can often replace each component by its (finite-state) downward closure, without affecting important properties such as safety. For example, *asynchronous programs* [39] or, more generally, *concurrent programs* [13], consist of threads that are modeled by infinite-state systems. Moreover, in these models, threads can be spawned dynamically during the run. It was observed recently that any algorithm for computing downward closures for threads, can be used to decide safety, boundedness, and termination of the entire asynchronous program [64, Section 6] or concurrent programs [13]. Another example

is paramerized verification of shared memory systems with non-atomic reads and writes [42]: As shown by LaTorre, Muscholl, and Walukiewicz [56], any algorithm for computing downward closures of the leader process yields a procedure for safety verification.

Downward closures and bounded treewidth. Therefore, an algorithm for computing downward closures of bounded treewidth systems would directly yield procedures for verifying various complex systems with bounded-treewidth components.

Computing downward closures of bounded-treewidth systems is far from understood. There is an abstract result implying the existence of an algorithm[1] [80], but that algorithm comes with no complexity upper bounds (not even beyond primitive-recursive): It enumerates potential finite automata and then checks which one is the downward closure.

Special treewidth (stw). A slight restriction of "bounded treewidth" is to bound the *special treewidth* [30] (in short *stw*). Consider graphs of bounded degree, as in particular, systems with stacks or queues lead to graphs of degree ≤ 3. For such graphs, bounded treewidth is equivalent to bounded stw (for degree $\leq d$, treewidth $\leq t$ implies stw $\leq 20dt$ [30]). In fact, to our knowledge, all bounded-treewidth infinite-state systems studied in the literature are bounded-stw: Even the underapproximations of the (unbounded-degree) timed multi-pushdown systems [3–5] have in fact bounded stw. See Appendix A for a detailed comparison.

Contributions. Our *first main result* reveals a connection between bounded-stw systems and a concept from computational linguistics: We show that the languages of bounded-stw systems are precisely those generated by *multiple context-free grammars* (MCFG) [74]. MCFG are an established concept in computational linguistics. Their main motivation is to extend the expressive power of context-free grammars while retaining their algorithmic properties. In this way, they are an example model of *mildly context-sensitive languages*. As we show, the stw is reflected in what we call the *width* of the MCFG. In an MCFG, a nonterminal generates a k-tuple of words rather than a single word (as for CFG). Here, k is the *arity* of that nonterminal, and the maximal arity occurring in an MCFG is its *dimension*, denoted d. The *rank*, denoted r, of an MCFG is the maximal number of nonterminal occurrences in a production. Then the width of the MCFG is $k = d \cdot r$. Starting from graphs with stw k, the resulting MCFG has width $2k$. Conversely, from an MCFG of width k, we obtain graphs of stw $6k$.

In our *second main result*, we exploit this new connection for the computation of downard closures. We show that for a given MCFG (of fixed dimension), one can construct a downward closure NFA of at most doubly exponential size. We also show that this is optimal. In particular, this yields optimal downward constructions for bounded-stw underapproximations for MPDA.

Application I: technology transfer. Coincidentally, a recent paper by Conrado, Kjelstrøm, van de Pol, and Pavlogiannis [26] construct a direct, bespoke underapproximation of MPDA (there phrased as interleaved Dyck reachability) in the form of an MCFG. This appears to be a promising approach: The paper provides efficient (and almost conditionally optimal) bounds for language-theoretic problems for MCFG. Moreover, the authors show that empirically, their underapproximation has remarkably high coverage (of executions) in practice. Our results show that the two types of underapproximation allow transferring algorithmic techniques: (i) the algorithms by Conrado et. al. (in fact, any algorithms for MCFG) apply to any bounded-stw underapproximation (and thus all bounded-treewidth systems in the literature), but also (ii) all algorithms for bounded-stw systems based on Courcelle's Theorem apply to the languages of Conrado et. al. In particular, the underapproximation of Conrado et. al. can be viewed as part of the bounded-treewidth framework.

[1] One can deduce this from [80] because the languages of bounded-treewidth systems have semilinear Parikh images [63].

Application II: Concurrent Programs. As mentioned above, downward closures can be used to verify systems with dynamic spawning. A *dynamic network of concurrent pushdown systems* (DCPS) [10] consists of recursive processes, modeled as pushdown systems. These processes can access the shared memory of finite domain, but also spawn new processes. At each moment, there is one active process, which can be interrupted up to K times. Thus, DCPS are multi-threaded shared-memory programs with dynamic spawning and K-bounded context-switching. Deciding safety of DCPS is complete for 2EXPSPACE [10, 15] (i.e. two-fold exponential space).

Our final application concerns DCPS where the individual processes can not only be recursive programs, but *any bounded-stw system*, such as many underapproximations of MPDA, but also queue systems, etc. The resulting systems are then *Concurrent Programs over MCFG* (CPM), since by our first contribution, the individual processes can be represented by MCFG. Here, our result is that for CPM, safety verification is still 2EXPSPACE-complete. Hence, although the model of CPM permits much more involved individual processes than DCPS, our result shows that the complexity of safety verification remains the same.

To this end, we rely on the idea that for safety in DCPS, one computes downward closures of context-free languages, which results in exponentially large finite automata. These automata are then used to construct an exponential-sized vector addition system with states (VASS), which is checked for coverability. Since coverability in VASS is EXPSPACE-complete, this results in a 2EXPSPACE procedure overall. Applying our aforementioned downward closure result directly would thus only lead to a 3EXPSPACE upper bound. Therefore, we require another trick: We introduce the notion of the *partially permuting downward closure*. For this, we show that (i) using an additional *compression step*, one can compute a *singly-exponential* automaton, and that (ii) these new closures can be used in place of the classical downward closure in safety of concurrent programs.

Structure of the paper. In Section 2, we will introduce and explain basic concepts. In Section 3, we outline the main results of this paper. Then, Sections 4 and 5 present the translations between MCFG and bounded-stw systems. In Section 6, we present the downward closure construction for MCFG. Finally, Section 7 is about the application to concurrent programs over MCFG. Due to space constraints, many proof details are only available in the appendix.

Related work. While this work is about *underapproximations* of MPDA reachability, there has also been a significant amount of work on *overapproximations*. Note underapproximating the set of behaviors of a system can be used to find bugs (any undesirable behavior in an underapproximation also exists in the original system), whereas overapproximations can certify the absence of bugs (a bug-free overapproximation means the original system is correct). Prominent examples include bidirected interleaved Dyck reachability [38, 53, 54, 60, 61], LCL-reachability [82], overapproximations based on integer linear programming [62], synchronized pushdown reachability [75], and techniques for refining overapproximations [27, 33]. See [68] for a survey on algorithmic aspects.

Another application of treewidth in static analysis of recursive programs is to exploit the empirical observation that many programs in practice have control-flow graphs of small treewidth [20, 77]. This motivates the study of parameterized complexity of static analysis problems under bounded treewidth [19, 21–23, 40]. However, this approach is fundamentally different from bounded treewidth underapproximations: The latter achieve decidability by restricting the *search space*, i.e. the set of *program executions*, to those with bounded treewidth. In the parameterized complexity setting, however, one studies how much more efficient the (already decidable) analysis becomes if the *input program*, specifically its control-flow graph, has small treewidth.

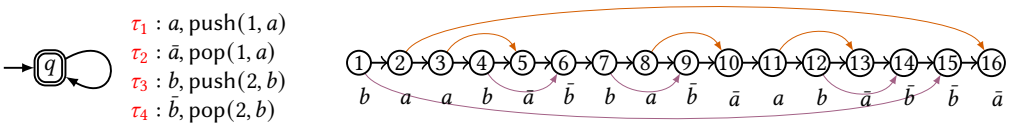


Fig. 1. A 2-stack MPDA $\mathcal{M}_{\text{int-Dyck}}$ on the left and an accepting rungraph of $baab\bar{a}\bar{b}ba\bar{b}\bar{a}ab\bar{a}\bar{b}\bar{b}\bar{a}$ on the right.

2 Preliminaries

Multi-pushdown automata and languages. A *multi-pushdown automaton* (MPDA) with s stacks, over a finite alphabet Σ is a tuple of the form $\mathcal{M} = (Q, \Gamma, q_0, \Delta, f)$, where Q is a finite set of control states, Γ is a finite stack alphabet, $q_0 \in Q$ is the initial state, $f \in Q$ is the final state and $\Delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times \text{ops}_s \times Q$, where $\text{ops}_s = \{\text{push}(i, a), \text{pop}(i, a) \mid i \in [1..s], a \in \Gamma\} \cup \{\text{nop}\}$. It accepts a word $w \in \Sigma^*$ if there is a run on w that ends in the final state with all stacks empty.

Example 2.1. Recall that the Dyck-1 language $D_1 \subseteq \{a, \bar{a}\}^*$ consists of all "well-bracketed" words, i.e. words w with $|w|_a = |w|_{\bar{a}}$ and every prefix u of w admits $|u|_a \geq |u|_{\bar{a}}$. Consider the MPDA with 2 stacks depicted in Figure 1 (left). It accepts all interleavings of two Dyck-1 languages, one over the alphabet $\{a, \bar{a}\}$ and the other over $\{b, \bar{b}\}$. For instance, it accepts the word $baab\bar{a}\bar{b}ba\bar{b}\bar{a}ab\bar{a}\bar{b}\bar{b}\bar{a}$. An accepting run on this word is graphically represented in Fig 1 (right), where the matching push-pop relations are depicted by curved arrows. The matching relation of stack 1 (resp. stack 2) is shown by an upward (resp. downward) curve.

A *rungraph* is a structure $\rho = (V, \texttt{Succ}, \texttt{Match}, \lambda)$ where $V = \{v_1, \cdots, v_n\}$ is a finite set of vertices, Succ is the successor relation of a total order on V, $\texttt{Match} \subseteq V \times V$ is a binary matching relation and the labelling $\lambda : V \mapsto \Sigma$ maps vertices to letters. A rungraph is accepted by an MPDA if there is a labelling $\text{trans} : V \mapsto \Delta$ that is consistent with λ, Succ and Match. Consistency of trans with Match also requires that the last-in-first-out access policy of stacks is respected (well-nesting). See Appendix B for details. Further, the first (resp. last) event must be labelled by an initial (resp. final) transition.

The *(run)graph language* of an MPDA $\mathcal{M}$, denoted $\mathcal{L}_G(\mathcal{M})$, is the set of all rungraphs that are accepted by $\mathcal{M}$. For a rungraph $\rho = (V, \texttt{Succ}, \texttt{Match}, \lambda)$ let $\text{word}(\rho)$ be the word in Σ^* given by the Succ relation and λ. The *word language* of an MPDA, denoted $\mathcal{L}_W(\mathcal{M})$ or simply $\mathcal{L}(\mathcal{M})$, is given by $\mathcal{L}(\mathcal{M}) = \{w \mid w = \text{word}(\rho) \text{ for some } \rho \in \mathcal{L}_G(\mathcal{M})\}$.

Multiple context-free grammars and languages. Multiple context-free grammars (MCFG) generalize context-free grammars (CFG) by allowing each nonterminal to derive a tuple of words, rather than a single word. A production $A \to BC$ in a CFG tells us that if we take any word u derivable by B, and any word v derivable by C, then uv is considered derivable by A. Thus, we combine words derivable by B, C to obtain words derivable by A. In an MCFG, we do the same for tuples. A rule is of the form

$$A(s_1, \ldots, s_n) \leftarrow \{B_1(x_{1,1}, \ldots, x_{1,n_1}), \ldots, B_m(x_{m,1}, \ldots, x_{m,n_m})\}. \tag{Prod}$$

Here, the $x_{i,j}$ are variables, and $s_1, \ldots, s_n$ are words over both variables and terminal letters. Such a rule tells us that if the tuple $(u_{i,1}, \ldots, u_{i,n_i})$ is derivable by B_i for each i, then the tuple $(s'_1, \ldots, s'_n)$ is derivable by A. Here, s'_j is obtained from s_j by replacing each occurrence of $x_{i,\ell}$ by $u_{i,\ell}$. Importantly, every variable can occur at most once on each side; and every variable that occurs on the left also occurs on the right. This means, the words in the tuples $(u_{i,1}, \ldots, u_{i,n_i})$ can be rearranged (or forgotten), but not repeated. The number n is the *arity* of A, and m is the *rank* of the rule (Prod).

Formally, an MCFG is a tuple $\mathcal{G} = (\Sigma, \hat{N}, X, P, A_{\text{start}})$ where Σ is a finite alphabet, $\hat{N} = (N, \text{arity})$ is a finite set of nonterminals disjoint from Σ along with their arities given by $\text{arity} : N \to \mathbb{N}$, X is a finite set of variables disjoint from N and Σ, P is a finite set of *productions* of the form (Prod) and $A_{\text{start}} \in N$ is the start nonterminal with $\text{arity}(A_{\text{start}}) = 1$.

The *dimension*, denoted d, of an MCFG $\mathcal{G} = (\Sigma, \hat{N}, X, P, A_{\text{start}})$ is the maximum arity of the nonterminals. The *rank*, denoted r, of $\mathcal{G}$ is the maximum rank of its production rules; but we define it as 1 if all rules have rank 0. The *width* of the MCFG is defined as $k = d \cdot r$. A d-MCFG is an MCFG of dimension $\leq d$; and a d-MCFG(r) is one that also has rank $\leq r$. Note that the 1-MCFG are precisely the context-free grammars (CFG).

Next we define the language of an MCFG. First, a nonterminal A of arity n generates a set of n-tuples of words, denoted by $R(A)$. We also write $A(u_1, \ldots, u_n)$ to mean $(u_1, \ldots, u_n) \in R(A)$. We define $R(A) \subseteq (\Sigma^*)^n$ inductively by the following deduction rules.

$$\frac{A(s_1, \ldots, s_n) \leftarrow \emptyset}{A(s_1, \ldots, s_n)} \quad (1) \qquad \frac{\text{(Prod)} \qquad B_i(u_{i,1}, \ldots, u_{i,n_i})\ \forall i \in \{1, 2, \ldots, m\}}{A(\eta(s_1), \ldots, \eta(s_n))} \quad (2)$$

Note that in the deduction rule (1), each $s_i \in \Sigma^*$. In the deduction rule (2) η is a homorphism $\eta : (X \cup \Sigma) \to \Sigma^*$ given by $\eta(x_{i,j}) = u_{i,j}$ and $\eta(a) = a$ for $a \in \Sigma$.

Notice also that if $(u_1, \ldots u_n) \in R(A)$ then the deduction rules naturally give a *derivation tree*. These notions are more rigorously explained in Appendix C.

The *language* of an MCFG $\mathcal{G} = (\Sigma, \hat{N}, X, P, A_{\text{start}})$, denoted $\mathcal{L}(\mathcal{G})$, is just $\mathcal{L}(\mathcal{G}) = R(A_{\text{start}})$. A language $L \subseteq \Sigma^*$ is a *multiple context-free language (MCFL)* if there is an MCFG $\mathcal{G}$ with $L = \mathcal{L}(\mathcal{G})$.

Example 2.2. Consider the MCFG $\mathcal{G} = (\{a, b\}, \hat{N}, X, P, A_{\text{start}})$ where $N = \{A, B, A_{\text{start}}\}$ with $\text{arity}(A) = 2, \text{arity}(B) = 3, \text{arity}(A_{\text{start}}) = 1$, $X = \{x_1, \ldots, x_5\}$ and P listed below. Figure 2 shows a parse tree with yield *baaabb*.

$$\begin{aligned} A_{\text{start}}(x_2x_1) &\leftarrow \{A(x_1, x_2)\} \\ A(x_2x_5x_4, x_3x_1) &\leftarrow \{A(x_1, x_2), B(x_3, x_4, x_5)\} \\ A(a, aa) &\leftarrow \emptyset \\ B(b, b, b) &\leftarrow \emptyset \end{aligned}$$

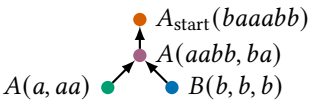


Fig. 2. A parse tree

About MCFL. MCFL are considered a promising candidate for a class of *mildly context-sensitive languages* as postulated by Joshi [48]. One aspect that makes them appealing is that there are several characterizations of MCFL by different formalisms, such as linear context-free rewriting systems [74, Lemma 2.2(f3)] (see also the comment on [74, p. 193]), hyperedge replacement grammars and deterministic tree-walking transducers [35, 79], minimalist grammars [43, 65], and multi-component tree-adjoining grammars [78]. In terms of complexity, the membership problem for MCFG is EXPTIME-complete in general [50, 51], NP-complete for fixed dimension $d \geq 2$ [49, 50], and P-complete if both dimension and rank are fixed [49, 50]. The membership problem was (among other problems) also recently studied by Conrado, Kjelstrøm, van de Pol, and Pavlogiannis [26], who provide close-to-optimal fine-grained complexity bounds.

Regarding their relationship with other language classes studied in verification, the MCFL are incomparable with the *indexed languages* [1], which are exactly the languages accepted by second-order pushdown automata [67, Theorem 2.2]. First, the indexed languages are not included in the MCFL, since all MCFL have semilinear Parikh images [78] (see also [74, Theorem 3.7]), whereas there are indexed languages, such as $\{a^{2^n} \mid n \in \mathbb{N}\}$, which have a non-semilinear Parikh image. Conversely, the MCFL are not included in the indexed languages: As observed by Kanazawa and

Salvati [52], the "triple copying theorem" by Engelfriet and Skyum [37, Theorem 2] implies that for every $K \subseteq \Sigma^*$, if the language $\{w\#w\#w \mid w \in K\}$ is indexed, then K must be an EDT0L language (see [36] for a definition). Now, for example the Dyck-1 language $D_1 \subseteq \{a, \bar{a}\}^*$ (see Example 2.1) is not EDT0L [34, Theorem 5], but context-free. Because of the latter, we can easily construct an MCFG for the language $\{w\#w\#w \mid w \in D_1\}$, which then cannot be indexed.

Downward closures. Given two words $u, v \in \Sigma^*$ for some finite alphabet Σ, we say u is a *subword* of v, denoted as $u \preceq v$ if u can be obtained by dropping some letters from v, that is, if $u = u_1 \cdots u_n$ and $v = v_0 u_1 v_1 \cdots u_n v_n$ for some words $v_0, u_1, \ldots, u_n, v_n \in \Sigma^*$. For a language $L \subseteq \Sigma^*$, its *downward closure* $L{\downarrow}$ is the set of all subwords of words in L, in symbols: $L{\downarrow} := \{u \in \Sigma^* \mid \exists v \in L \colon u \preceq v\}$.

Graphs and MSO definable classes of graphs. Let Σ and Γ be alphabets. An *lo-graph over* (Σ, Γ) is a directed graph with an underlying linear order, where (i) the vertices carry labels in $\Sigma \cup \{\varepsilon\}$ and (ii) edge relations have names in Γ. Formally, it is a tuple $G = \big(V, \textsc{Succ}, (E_\gamma)_{\gamma \in \Gamma}, \lambda\big)$ where (i) V is a finite set of vertices, (ii) $E_\gamma \subseteq V \times V$ is an edge relation (directed) for each $\gamma \in \Gamma$, (iii) $\textsc{Succ} \subseteq V \times V$ is the successor relation of a linear order on the vertices, and (iv) $\lambda : V \to \Sigma \cup \{\varepsilon\}$ is a labeling function. The set of all lo-graphs over (Σ, Γ) is denoted $\textsc{lo-GraphSet}(\Sigma, \Gamma)$.

The labels of the sequence of vertices visited according to the linear order of an lo-graph $G = \big(V, \textsc{Succ}, (E_\gamma)_{\gamma \in \Gamma}, \lambda\big)$ naturally give a word in Σ^*. We call this word $\mathsf{word}(G)$. That is, if $v_1, v_2 \ldots v_n$ is the sequence of all vertices in V according to the linear order (i.e., $(v_i, v_{i+1}) \in \textsc{Succ}$ for all $1 \leq i < n$), then $\mathsf{word}(G) = \lambda(v_1)\lambda(v_2)\ldots\lambda(v_n)$.

Let S be a set of lo-graphs over (Σ, Γ). We define $\mathsf{word}(S) = \{\mathsf{word}(G) \mid G \in S\}$.

Monadic second-order logic (MSO) on directed node-labelled graph with an underlying linear order over a finite alphabet Σ, denoted $\mathsf{MSO}(\Sigma, \Gamma)$, is defined in the standard way. An $\mathsf{MSO}(\Sigma, \Gamma)$ formula is given by the grammar

$$\phi = a(x) \mid x \in X \mid \textsc{Succ}(x, y) \mid E_\gamma(x, y) \mid \neg\phi \mid \phi \vee \phi \mid \exists x \phi \mid \exists X \phi$$

where $a \in \Sigma$, $\gamma \in \Gamma$, x, y are first-order variables ranging over vertices and X is a second-order variable ranging over subsets of vertices. The semantics is as expected.

Every $\mathsf{MSO}(\Sigma, \Gamma)$ sentence ϕ defines a set of lo-graphs over (Σ, Γ), denoted by $\mathcal{L}_{\mathrm{G}}(\phi)$. It also defines a set of words over Σ, denoted $\mathcal{L}_{\mathrm{W}}(\phi)$, given by

$$\mathcal{L}_{\mathrm{W}}(\phi) = \mathsf{word}(\mathcal{L}_{\mathrm{G}}(\phi)) = \{\mathsf{word}(G) \mid G \in \mathcal{L}_{\mathrm{G}}(\phi)\}.$$

One can show the following by declaring a second-order variable for each transition of an MPDA:

Claim 2.3. *For every MPDA $\mathcal{M}$ over the alphabet Σ, there is an MSO$(\Sigma, \{\mathtt{Match}\})$-formula $\phi_{\mathcal{M}}$ such that $\mathcal{L}_{\mathrm{G}}(\phi_{\mathcal{M}}) = \mathcal{L}_{\mathrm{G}}(\mathcal{M})$.*

Example 2.4. Consider the class of all directed grids over the node labels $\{a, b\}$. Linear order is given by rows in the successive order. There is an MSO sentence $\phi_{\mathrm{grid}} \in \mathsf{MSO}(\{a, b\}, \{\rightarrow, \downarrow\})$ for this. Consider the set of grids in which the node-label is the same in every column. This set is now definable by the following sentence. $\phi_{\mathrm{copy}} = \phi_{\mathrm{grid}} \wedge \forall x \forall y (E_{\downarrow}(x, y) \implies ((a(x) \wedge a(y)) \vee (b(x) \wedge b(y)))$. Note that $\mathcal{L}_{\mathrm{W}}(\phi_{\mathrm{copy}}) = \{u^k \mid k \geq 1, u \in \{a, b\}^*\}$.

Bounded (special) treewidth languages. A *tree decomposition* of a (directed) graph is a rooted tree where each tree node is labelled by a bag of graph vertices such that 1) each graph vertex appears in at least one bag, 2) the bags containing a graph vertex form a connected subtree, and 3) every adjacent pair of vertices of the graph must belong together in at least one bag.

A tree decomposition is *special* if the condition 2 above is strengthened as follows: 2′) the bags containing a graph vertex form a path along a branch of the rooted tree. In particular, this forbids

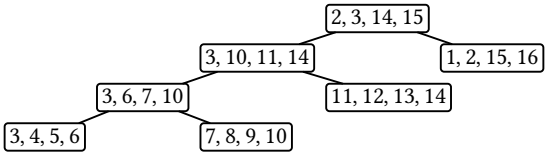


Fig. 3. A special tree decomposition of the rungraph in Figure 1.

two sibling bags to share a graph vertex. The width of a special tree decomposition is the maximum size of its bags. The *special treewidth* (stw) of a graph is the minimum width over all special tree decompositions.

Example 2.5. A tree decomposition of the rungraph in Figure 1 is given in Figure 3. It is a special tree decomposition, its width is 4 and hence the special treewidth of this rungraph is at most 4.

Given an $\mathsf{MSO}(\Sigma, \Gamma)$ sentence ϕ and a bound k on special treewidth, we define the *k-bounded language of ϕ* as follows: $\mathcal{L}^k_{\mathrm{G}}(\phi)$ the set of all lo-graphs that satisfy ϕ and have stw $\leq k$. Accordingly, we have $\mathcal{L}^k_{\mathrm{W}}(\phi) = \mathsf{word}(\mathcal{L}^k_{\mathrm{G}}(\phi))$. A language $L \subseteq \Sigma^*$ is an *MSO-definable bounded-stw language* if $L = \mathcal{L}^k_{\mathrm{W}}(\phi)$ for some $\mathsf{MSO}(\Sigma, \Gamma)$-formula ϕ for some Γ and $k \in \mathbb{N}$.

Similarly, we define the k-bounded rungraph language and word language of an MPDA as well: $\mathcal{L}^k_{\mathrm{G}}(\mathcal{M})$ be the set of all rungraphs of the MPDA $\mathcal{M}$ with stw at most k, $\mathcal{L}^k_{\mathrm{W}}(\mathcal{M}) = \mathsf{word}(\mathcal{L}^k_{\mathrm{G}}(\mathcal{M}))$.

Tree automata. A width-k tree decomposition of a graph can be viewed as a tree with finitely many node labels: for each possible bag, we use a separate node label. It is well known that the sets of trees that correspond to an MSO-definable bounded-stw set of graphs are precisely those recognized by *finite (bottom-up) tree automata*. Essentially, such a tree automaton works like a deterministic finite automaton on words: Initially all leaves are assigned an initial state, and then the state of an inner node v is determined by the states of v's children states and by v's label. Finally, a tree is accepted if the root node is assigned a final state. For now informally, we will call a deterministic bottom-up tree automaton that accepts width-k special tree decompositions a *width-k tree automaton*. This informal definition should be enough to understand our results; a formal definition will follow in Section 4.

Fact 2.6. *[28, 70] Given an MSO(Σ, Γ) sentence ϕ and a bound $k \in \mathbb{N}$, there is a width-k tree automaton $\mathcal{A}$ such that $\mathcal{L}_{\mathrm{G}}(\mathcal{A}) = \mathcal{L}^k_{\mathrm{G}}(\phi)$.*

3 Main results

Our first main result is the characterization of MSO-definable bounded-stw languages by MCFL:

Theorem 3.1. *The MSO-definable bounded-stw languages are precisely the MCFLs.*

Theorem 3.1 will follow from Theorems 3.2 and 3.3, which provide complexity bounds. First, for the translation from an MSO-definable bounded-stw language to an MCFG, we need to translate an MSO formula to an MCFG. However, in algorithmic settings, MSO-definable bounded-stw languages are usually represented by tree automata (cf. Fact 2.6), which have much better algorithmic properties (e.g. emptiness is decidable in polynomial time, compared to non-elementary complexity for MSO). Hence, the following implies that MSO-definable bounded-stw languages are MCFL:

Theorem 3.2. *Fix $k \geq 0$. Given a width-k tree automaton $\mathcal{A}$, we can construct a k-MCFG(2) for $\mathcal{L}_{\mathrm{W}}(\mathcal{A})$ in time $|\mathcal{A}| \cdot 2^{O(k \log k)}$.*

Hence, the resulting MCFG is of size $|\mathcal{A}| \cdot 2^{O(k \log k)}$ and width $2k$. For the converse translation:

Theorem 3.3. *Given a d-MCFG(r) $\mathcal{G}$, we can construct an MSO formula ϕ with $\mathcal{L}_{\mathrm{W}}^{6dr}(\phi) = \mathcal{L}(\mathcal{G})$.*

In particular, starting from an MCFG of width k, we obtain a special treewidth bound of $6k$. Since Theorem 3.3 is not used in our applications (and rather shows that Theorem 3.1 is an exact characterization), complexity is not crucial, so we present that translation using MSO.

Downward closures of MCFL. Theorem 3.2 implies that a procedure for downward closure computation of MCFL yields such a procedure for MSO-definable bounded-stw languages. This motivates:

Theorem 3.4. *Fix $d \geq 1$. Given a d-MCFG $\mathcal{G}$, we can compute an NFA for $\mathcal{L}(\mathcal{G})\downarrow$ in time $2^{2^{\mathrm{poly}(|\mathcal{G}|)}}$.*

In particular, the resulting NFA will be of size $2^{2^{\mathrm{poly}(|\mathcal{G}|)}}$. Note that here, the dimension is fixed, but the rank of $\mathcal{G}$ is part of the input. We also show that the doubly exponential upper bound in Theorem 3.4 is optimal, by constructing a 2-MCFG(2) $\mathcal{G}_n$ of size linear in n for the language $L_n = \{ww \mid w \in \{a,b\}^{2^n}\}$ (see Appendix H for details):

Theorem 3.5. *There is a family of 2-MCFG(2) $(\mathcal{G}_n)_{n\in\mathbb{N}}$ with $|\mathcal{G}_n| = O(n)$ such that any NFA for $\mathcal{L}(\mathcal{G}_n)\downarrow$ has at least 2^{2^n} states.*

Multi-pushdown automata. An important application domain of our results concerns bounded-treewidth underapproximations of MPDA: By imposing a treewidth bound on the rungraphs of MPDA, one obtains an MSO-definable bounded-stw language. One can thus use Theorem 3.2 and Theorem 3.4 to compute the downward closure of a bounded-treewidth underapproximation of a given MPDA (this is the setting where our later applications will be used). This raises the question of whether the translation MPDA $\rightsquigarrow$ width-k tree automaton $\rightsquigarrow$ MCFG $\rightsquigarrow$ downward closure NFA introduces an unnecessary blow-up. We will now see that this is not the case, provided that we fix the stw (equivalently, the treewidth [30]) and the number s of stacks. First, as shown in [2, 4, 31, 32, 63], given an MPDA, we can efficiently construct a width-k tree automaton:

Fact 3.6. *Given an MPDA $\mathcal{M}$ with s stacks and a bound k, one can construct a width-k tree automaton $\mathcal{A}$ for $\mathcal{L}_{\mathrm{G}}^k(\mathcal{M})$ in time $|\mathcal{M}|^k \times 2^{poly(s,k)}$.*

From Fact 3.6, Theorem 3.2 and Theorem 3.4, we can conclude a doubly exponential upper bound for downward closures of MSO-definable bounded-treewidth MPDA languages:

Corollary 3.7. *Fix s and k. Given an MPDA $\mathcal{M}$ with s stacks, we can compute in doubly exponential time an NFA for $\mathcal{L}_{\mathrm{G}}^k(\mathcal{M})\downarrow$.*

The upper bound in Corollary 3.7 is optimal (see Appendix I):

Theorem 3.8. *There is a bound k and a family of 2-stack MPDA $(\mathcal{M}_n)_{n\in\mathbb{N}}$ with $|\mathcal{M}_n| = O(n)$ and stw at most k such that any NFA for the downward closure $\mathcal{L}(\mathcal{M}_n)\downarrow$ has at least 2^{2^n} states.*

Concurrent Programs. As mentioned in Section 1, an important underapproximation of MPDAs is that of k-bounded context switching: A run is allowed to switch between stacks only k times. In the same section we also mention that this can be lifted to dynamic pushdown systems: Stacks can dynamically spawn during execution, and each stack may switch k times. Finally we lift this even further so that each dynamically spawned process is not a mere pushdown, but itself an MPDA underapproximation, or some other MSO-definable bounded-stw system (however, we still require that each process as a whole, is only interrupted at most k times). With our previous results, we assume all processes have been transformed into *MCFG*, that have fixed dimension d and rank r. This gives rise to *concurrent programs* (CP) over d-MCFG(r). With some modifications to our

downward closure computation, we obtain an optimal complexity upper bound for k-context-bounded safety:

Theorem 3.9. *Fix $d, r \geq 0$. The following problem is* 2EXPSPACE*-complete: Given a* CP *over the d-MCFG(r), one of its global states q_f, and a unary encoded $k \geq 1$, is q_f reachable under k-bounded context switching?*

The lower bound already follows from the case of CP over the CFG for a fixed $k \geq 1$ [13].

4 Bounded-special-treewidth languages are MCFL

Let us give an overview of the proof of Theorem 3.2. Details can be found in Appendix E. We begin with a formal definition of width-k tree automata, which were only sketched in Section 2. This definition relies on graph algebras [30], which allow us to express the graphs of bounded special treewidth by terms (or trees) in the algebra.

A graph algebra for bounded-stw graphs [30]. We need some terminology. We first define *colored piecewise linearly ordered* graphs (cp-lo-graphs).

A structure $(V, \textsc{Succ})$ is called *piecewise linearly ordered* if V and $\textsc{Succ}$ can be partitioned into a number of parts $V = \uplus_i V_i$ and $\textsc{Succ} = \uplus_i S_i$ such that each part S_i is the successor relation of a total order on V_i. Each part (V_i, S_i) is called a piece. That is, each piece is totally ordered by $\textsc{Succ}$, but there is no order between the pieces.

A cp-lo-graph does not require the vertices to be totally ordered by $\textsc{Succ}$ like in an lo-graph. Instead they only need to be piecewise linearly ordered. Further, some of the vertices of a cp-lo-graph are uniquely colored. In particular we require the extreme vertices of every piece to be colored.

Formally, a *cp-lo-graph* is a tuple $\hat{G} = \big(V, \textsc{Succ}, (E_\gamma)_{\gamma\in\Gamma}, \lambda, \chi\big)$, where $\textsc{Succ} \subseteq \textsc{Succ}'$ for some $\textsc{Succ}'$ such that $\big(V, \textsc{Succ}', (E_\gamma)_{\gamma\in\Gamma}, \lambda\big)$ is a lo-graph. A maximal $\textsc{Succ}$-connected component is called a *piece*. The coloring $\chi : V \to \mathsf{C}$ is a partial injective function from vertices to colors, and must keep colored the vertices that either have no incoming $\textsc{Succ}$ edge or no outgoing $\textsc{Succ}$ edge.

Given a cp-lo-graph $\hat{G} = \big(V, \textsc{Succ}, (E_\gamma)_{\gamma\in\Gamma}, \lambda, \chi\big)$, we denote by $\diamond(\hat{G})$ the (colorless) piecewise-lo-graph $\big(V, \textsc{Succ}, (E_\gamma)_{\gamma\in\Gamma}, \lambda\big)$ where the last entry χ of $\hat{G}$ is omitted. Note that an lo-graph is also a piecewise-lo-graph with only one piece.

We will define some operations to iteratively construct cp-lo-graphs. These operations can produce structures $\big(V, \textsc{Succ}, (E_\gamma)_{\gamma\in\Gamma}, \lambda, \chi\big)$ which need not be cp-lo-graphs in a strict sense. For instance, $\textsc{Succ}$ can be an arbitray edge relation instead of the successor relation of a piecewise linear order, or the extreme vertices of a piece may be uncolored. However, even the relaxed cp-lo-graphs insist that coloring $\chi : V \to \mathsf{C}$ is a partial injective function from vertices to colors. We call such structures *relaxed cp-lo-graph*.

First we have a set of unary operations UOP and a single binary operation $\textsc{Join}(\cdot)$. The set of unary operators is $\mathsf{UOP} = \{\textsc{Node}(c, a) \mid c \in \mathsf{C}, a \in \Sigma \cup \{\varepsilon\}\} \cup \{\textsc{Succ}(c_1, c_2) \mid c_1, c_2 \in \mathsf{C}\} \cup \{\textsc{Edge}(\gamma, c_1, c_2) \mid c_1, c_2 \in \mathsf{C}, \gamma \in \Gamma\} \cup \{\textsc{Free}(c) \mid c \in \mathsf{C}\}$. For each $\mathrm{op} \in \mathsf{UOP}$, we define $\mathrm{op}(\big(V, \textsc{Succ}, (E_\gamma)_{\gamma\in\Gamma}, \lambda, \chi\big)) = \Big(V', \textsc{Succ}', (E'_\gamma)_{\gamma\in\Gamma}, \lambda', \chi'\Big)$ as follows.

- $\mathrm{op} \equiv \textsc{Node}(c, a)(\cdot)$: Requires that $c \notin \mathrm{Img}(\chi)$. The resulting graph contains a new vertex v colored c and labelled a. Notice that a may also be ε. Note that v will be a piece by itself. That is, $V' = V \uplus \{v\}$, $\textsc{Succ}' = \textsc{Succ}$, $E'_\gamma = E_\gamma$ for each $\gamma \in \Gamma$, $\lambda' = \lambda \uplus \{(v, a)\}$, $\chi' = \chi \uplus \{(v, c)\}$.
- $\mathrm{op} \equiv \textsc{Succ}(c_1, c_2)(\cdot)$: Adds a $\textsc{Succ}$ edge from the unique c_1-colored vertex u to the unique c_2-colored vertex v. That is, $V' = V$, $E'_\gamma = E_\gamma$ for each $\gamma \in \Gamma$, $\lambda' = \lambda$, $\chi' = \chi$, $\textsc{Succ}' = \textsc{Succ} \uplus \{(u, v) \mid \chi(u) = c_1, \chi(v) = c_2\}$.

- op ≡ $\textsc{Edge}(\gamma, c_1, c_2)(\cdot)$: Adds a γ-labelled edge from the unique c_1-colored vertex u to the unique c_2-colored vertex v. That is, $V' = V$, $\textsc{Succ}' = \textsc{Succ}$, $\lambda' = \lambda$, $\chi' = \chi$, $E'_\gamma = E_\gamma \uplus \{(u,v) \mid \chi(u) = c_1, \chi(v) = c_2\}$, $E'_{\gamma'} = E_{\gamma'}$ for each $\gamma' \in \Gamma \setminus \{\gamma\}$.
- op ≡ $\textsc{Free}(c)(\cdot)$: This frees the color c. That is, $V' = V$, $\textsc{Succ}' = \textsc{Succ}$, $E'_\gamma = E_\gamma$ for each $\gamma \in \Gamma$, $\lambda' = \lambda$, $\chi' = \chi \setminus \{(u,c) \mid \chi(u) = c\}$.

The binary operation Join takes two relaxed cp-lo-graphs with disjoint colors, and constructs their disjoint union. For $i \in \{1,2\}$, let $\hat{G}_i = \big(V_i, \textsc{Succ}_i, (E^i_\gamma)_{\gamma\in\Gamma}, \lambda_i, \chi_i\big)$. Their join is $\hat{G} = \big(V, \textsc{Succ}, (E_\gamma)_{\gamma\in\Gamma}, \lambda, \chi\big)$ defined with $V = V_1 \uplus V_2$, $\textsc{Succ} = \textsc{Succ}_1 \uplus \textsc{Succ}_2$, $E_\gamma = E^1_\gamma \uplus E^2_\gamma$ for each $\gamma \in \Gamma$, $\lambda = \lambda_1 \uplus \lambda_2$, and $\chi = \chi_1 \uplus \chi_2$.

An empty cp-lo-graph is one with no vertices. It is denoted $\bot$. We are interested in structures obtained by (repeated) application of the above operations starting from $\bot$. This can be described by expressions given by the following grammar.

$$\psi := \bot \mid \textsc{Node}(c,a)(\psi) \mid \textsc{Succ}(c_1,c_2)(\psi) \mid \textsc{Edge}(\gamma,c_1,c_2)(\psi) \mid \textsc{Free}(c)(\psi) \mid \textsc{Join}(\psi,\psi)$$

where $c, c_1, c_2 \in \mathsf{C}$, $a \in \Sigma \cup \{\epsilon\}$ and $\gamma \in \Gamma$. The set of all expressions generated by the above grammar is denoted $\mathtt{expr}(\mathsf{C}, \Sigma, \Gamma)$. The structure defined by an expression ψ is denoted $\hat{G}_\psi$. This is defined inductively as expected. The special treewidth of $\hat{G}_\psi$ is bounded by $|\mathsf{C}|$:

Remark 4.1. *Note that the term ψ is a special tree decomposition of $\hat{G}_\psi$. Indeed, if a node x is the root of subterm ψ' of ψ, then the bag at x will be the colored vertices of the graph $\hat{G}_{\psi'}$.*

The *special treewidth* (stw) of a cp-lo-graph $\hat{G}$ is defined to be m where $m = \inf\big\{\, |\mathsf{C}| \;\big|\; \hat{G} = \hat{G}_\psi \text{ for some } \psi \in \mathtt{expr}(\mathsf{C},\Sigma,\Gamma) \,\big\}$. Note that due to the $\textsc{Free}(\cdot)$-operation, m may be larger than the final set of colors used in $\hat{G}$.

Width-k tree automata. A *width-k tree automaton* is a deterministic bottom-up tree automaton over expressions from $\mathtt{expr}(\{1,2,\ldots,k\},\Sigma,\Gamma)$, i.e., with the color set $\mathsf{C} = \{1,2,\ldots,k\}$. It is given by a tuple $\mathcal{A} = \big(Q, \delta, q_\bot, Q_{\text{accept}}\big)$ where Q is a finite set of states, $Q_{\text{accept}} \subseteq Q$ is the set of accepting states and $\delta = \bigcup_{\text{op}\in\text{UOP}} \delta_{\text{op}} \cup \delta_{\textsc{Join}}$ is the set of transition functions: $\delta_{\text{op}} : Q \to Q$ for all op $\in$ UOP, and $\delta_{\textsc{Join}} : Q \times Q \to Q$.

A deterministic bottom-up tree automaton $\mathcal{A} = \big(Q, \delta, q_\bot, Q_{\text{accept}}\big)$ defines a (partial) function $[\![\mathcal{A}]\!] : \mathtt{expr}(C,\Sigma,\Gamma) \to Q$, inductively as follows

- $[\![\mathcal{A}]\!](\bot) = q_\bot$
- if $\psi_1 \equiv \text{op}(\psi_2)$ for some op $\in$ UOP, then $[\![\mathcal{A}]\!](\psi_1) = \delta_{\text{op}}([\![\mathcal{A}]\!](\psi_2))$
- if $\psi_1 \equiv \textsc{Join}(\psi_2,\psi_3)$, then $[\![\mathcal{A}]\!](\psi_1) = \delta_{\textsc{Join}}([\![\mathcal{A}]\!](\psi_2), [\![\mathcal{A}]\!](\psi_3))$

The language of a tree automaton $\mathcal{A} = \big(Q, \delta, q_\bot, Q_{\text{accept}}\big)$, denoted $\mathcal{L}(\mathcal{A})$, is defined as $\mathcal{L}(\mathcal{A}) = \{\psi \mid [\![\mathcal{A}]\!](\psi) \in Q_{\text{accept}}\}$. We also define $\mathcal{L}_{\mathsf{G}}(\mathcal{A})$ as the set of structures defined by $\mathcal{L}(\mathcal{A})$ that are lo-graphs, $\mathcal{L}_{\mathsf{G}}(\mathcal{A}) = \{\Diamond(\hat{G}_\psi) \mid \psi \in \mathcal{L}(\mathcal{A})\} \cap \textsc{lo-GraphSet}(\Sigma,\Gamma)$, and $\mathcal{L}_{\mathsf{W}}(\mathcal{A})$ as $\mathcal{L}_{\mathsf{W}}(\mathcal{A}) = \{\mathsf{word}(G) \mid G \in \mathcal{L}_{\mathsf{G}}(\mathcal{A})\}$. Recall that $\Diamond(\cdot)$ removes the coloring χ.

Challenges. Consider a tree automaton $\mathcal{A} = \big(Q, \delta^{\mathcal{A}}, q_\bot, Q_{\text{accept}}\big)$ over $\mathtt{expr}(C,\Sigma,\Gamma)$. We defined $\mathcal{L}(\mathcal{A}) = \{\psi \mid [\![\mathcal{A}]\!](\psi) \in Q_{\text{accept}}\}$. Indeed for any state $q \in Q$, we can define $\mathcal{L}(\mathcal{A},q) = \{\psi \mid [\![\mathcal{A}]\!](\psi) = q\}$. Let $\mathcal{L}_{\hat{\mathsf{G}}}(\mathcal{A}) = \{\hat{G}_\psi \mid \psi \in \mathcal{L}(\mathcal{A})\}$. Note that $\mathcal{L}_{\hat{\mathsf{G}}}(\mathcal{A})$ is different from $\mathcal{L}_{\mathsf{G}}(\mathcal{A})$ as we are not intersecting it with $\textsc{lo-GraphSet}(\Sigma,\Gamma)$. In particular, $\hat{G}_\psi$ may be a relaxed cp-lo-graph, not a strict cp-lo-graph, since the Succ-edges can branch or form cycles. On the other hand, if $\hat{G}$ is a strict cp-lo-graph, Succ still need not induce a total order but only a piecewise linear order. Here we extend the definition of $\mathsf{word}(\cdot)$ to $\mathsf{word}(\hat{G})$, but now it defines a multiset of words, with one word corresponding to each piece.

In our MCFG, we will have a nonterminal A_q for each state q of the tree automaton. For $\psi \in \mathcal{L}(\mathcal{A}, q)$, if $\hat{G}_\psi$ is a cp-lo-graph, then the nonterminal A_q should be able to generate word$(\hat{G}_\psi)$. There are several challenges: 1) For any $\psi \in \mathcal{L}(\mathcal{A}, q)$, we need to check whether $\hat{G}_\psi$ is a cp-lo-graph. 2) For a $\hat{G}_\psi$ that is a cp-lo-graph, word$(\hat{G}_\psi)$ is a multiset of words, but $\mathcal{L}(A_q)$ generates a tuple of words. Hence, we need to order the words in the multiset, so as to correspond to the tuple. 3) For $\psi_1, \psi_2 \in \mathcal{L}(\mathcal{A}, q)$, the number of pieces in $\hat{G}_{\psi_1}$ may be different from that of $\hat{G}_{\psi_2}$. Hence fixing an arity for the nonterminal A_q is problematic. One possible solution is to have copies of A_q one of each possible arity.

Solution. Since there are only a fixed number of colors in any expression, we can address these issues by maintaining a finite *summary* for each expression. The summary contains the start and end colors of each piece (note that they need to be colored, if a Succ-edge is to be added in the future). This already has the information about the number of pieces (cf. challenge 3 above). We assume an ordering between the colors, and order the pieces based on their start colors (cf. challenge 2). Storing the start and end colors will also allow us to forbid cycles or adding multiple successors/predecessors with Succ-edges (cf. challenge 1).

These summaries can be computed by a tree automaton. We then consider the Cartesian product $\mathcal{B}$ of the tree automaton $\mathcal{A}$ and the summary-computing automaton. The states in the product can be thought of as annotating the states of $\mathcal{A}$ with the summary information.

Syntactic translation. Now $\mathcal{B}$ translates synctactically into an MCFG. Indeed, the nonterminals are the states, with the number of pieces in the summary as the arity. The transitions translate to productions. For a Join-transition $(B, C) \mapsto A$ of $\mathcal{B}$, we will have a production of the form $A(s_1, \dots s_\ell) \leftarrow \{B(x_1, \dots, x_m), C(y_1, \dots y_n)\}$ with $\{s_1, \dots, s_\ell\} = \{x_1, \dots, x_m\} \cup \{y_1, \dots, y_m\}$. Corresponding to a Node(c, a)-transition $B \mapsto A$, there will be a production of the form $A(s_1, \dots s_\ell) \leftarrow \{B(x_1, \dots, x_m)\}$ where $\ell = m + 1$. Every s_i is some x_j, except for one which is the letter a. The nonterminal A keeps track of the fact that this a is a piece by itself and is colored c. Corresponding to a Succ(c_1, c_2)-transition, there will be a production of the form $A(s_1, \dots s_\ell) \leftarrow \{B(x_1, \dots, x_m)\}$ where $\ell = m - 1$. Every s_i is some x_j, except for one which is now a concatenation of the form $x_k x_{k'}$. The indices k, k' are indeed determined by the summary kept at B: The end vertex of x_k is colored c_1 and the start vertex of $x_{k'}$ is colored c_2. The nonterminal A keeps track of the fact that this new piece gets the pair (c_3, c_4) of colors if c_3 was coloring the start vertex of x_k and c_4 was coloring the end vertex of $x_{k'}$. Corresponding to an Edge(γ, c_1, c_2)-transition there will be a production of the form $A(x_1, \dots, x_m) \leftarrow \{B(x_1, \dots, x_m)\}$ as the pieces are not changed, only the states are updated.

The ordering of the arguments $s_1, \dots, s_\ell$ follows the ordering we fixed on the colors: Each piece s_i is summarized by a pair of colors of their start and end vertices. The pieces are ordered according to their start-vertex color. For details, see Appendix E.

5 MCFL are MSO-definable bounded-stw languages

Towards Theorem 3.3, we show that every MCFL can be described by an MSO sentence over words with an additional binary relation $\curvearrowright$ that we call matching. The matching relation $\curvearrowright$ is, in a sense, a generalisation of the nesting/matching relation for context-free languages[6, 59]. We call these lo-graph over $(\Sigma, \{\curvearrowright\})$ *nested words*.

Nested words of an MCFG. Consider an MCFG $\mathcal{G} = (\Sigma, \hat{N}, X, P, A_{\text{start}})$ and a word $w \in \mathcal{L}(\mathcal{G})$. For the construction, we assume that $\mathcal{G}$ has the "information-lossless" property, meaning for each production Prod (see p. 5), every variable that occurs on the right also occurs on the left. This can be achieved using a translation due to Seki, Matsumura, Fuji, and Kasami [74, Lemma 2.2(f3)].

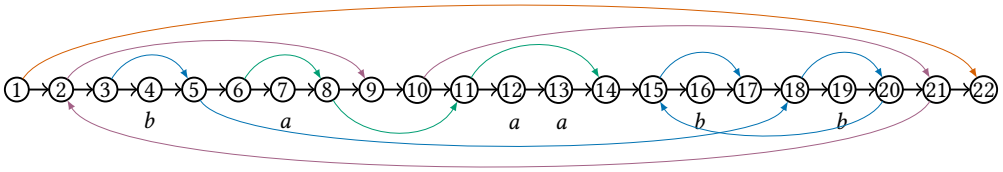


Fig. 4. The nested-word corresponding to the parse tree in Fig 2 generating *baaabb*.

Suppose the parse tree for w has a subtree for $A(u_1, u_2, \ldots, u_k)$ where $u_i \in \Sigma^*$ for $i \in \{1, 2, \ldots, k\}$. Then we can write w as $w = w_0 u_{i_1} w_1 u_{i_2} \ldots u_{i_k} w_k$ such that $\{i_1, i_2, \ldots i_k\} = \{1, \ldots, k\}$. That is, the entries $u_1, \ldots, u_k$ in the tuple generated by A eventually appear as non-overlapping infixes in the final word. We would like to mark the endpoints of these infixes and connect them, indicating they were related together as one tuple in the derivation tree. Towards this, we will introduce extra vertices in w, labelled ϵ as they should not change the word w, bordering each entry u_i. We then connect these extra vertices using the additional $\curvearrowright$ edge. This would give us an lo-graph G_w with $\mathrm{word}(G_w) = w$ and G_w would correspond to a parsetree-structure of w. See Appendix F.1 for details. We call such lo-graphs *nested words* of $\mathcal{G}$.

Example 5.1. Consider the parse tree in Figure 2, generating the word *baaabb* (cf. Example 2.2). The graph with $\curvearrowright$ relation corresponding to this parsetree is depicted in Figure 4. The label of a node, if not specified, is ϵ. The ϵ-labelled nodes participate in the $\curvearrowright$ relation, and the Σ labelled nodes do not. There are 4 different maximal $\curvearrowright$-connected paths in this graph, each colored differently. Each of these paths correspond to one node of the parse-tree. We can notice the well-nesting: The open interval $(1, 22)$ contains $((10, 21), (2, 9))$ which in turn contains $((6, 8), (11, 14))$ and $((3, 5), (18, 20), (15, 17))$. Note that every maximal $\curvearrowright$-path corresponds to a node in the parsetree. The number of nodes in a $\curvearrowright$-connected path is exactly twice the arity of the respective nonterminal.

Note that, every lo-graph over $(\Sigma, \{\curvearrowright\})$ is not necessarily a nested word of some MCFG $\mathcal{G}$. The purpose of this section is to show that, for every d-MCFG(r) $\mathcal{G}$ we can construct an $\mathsf{MSO}(\Sigma, \{\curvearrowright\})$ sentence ϕ such that $\mathcal{L}(\phi) = \mathcal{L}(\mathcal{G})$. The sentence ϕ has two parts. The first part (called the well-nesting property) is independent of $\mathcal{G}$, but depends on the dimension d. The second part assumes well-nesting and states that it corresponds to a valid parsetree of $\mathcal{G}$.

We will use $\leq$ to mean the reflexive transitive closure of Succ. Note that this is expressible in $\mathsf{MSO}(\mathrm{Succ})$. Once we assume the definition of $\leq$, the rest of the formula can be written in *existential* MSO (EMSO). We will first declare some more shorthands. We will also write pEq instead of $(p, q) \in E$, and $p \rightarrow q$ instead of $\mathrm{Succ}(p, q)$. Further we write $p_1 E_1 p_2 E_2 p_3$ to mean $p_1 E_1 p_2 \wedge p_2 E_2 p_3$.

Well-nesting matching relation $\curvearrowright$. Recall that d is the dimension, i.e. the maximum arity among nonterminals of the MCFG. The matching relation $\curvearrowright$ must satisfy the following properties.

(1) The length (i.e. the number of nodes) of any $\curvearrowright$-connected path is at most $2d$.
(2) The length of any maximal $\curvearrowright$-connected path is even.
(3) A node can have at most one $\curvearrowright$-successor and at most one $\curvearrowright$-predecessor.
(4) A vertex u participates in the matching relation $\curvearrowright$ iff it is labelled ϵ.
(5) Let $v_1 \curvearrowright v_2 \curvearrowright \ldots \curvearrowright v_{2k}$ be a maximal $\curvearrowright$-connected path. Then
    (a) the matching edges at odd positions (which enclose an entry) must agree with the linear order. That is, for all $i : 1 \leq i \leq k$, $v_{2i-1} \leq v_{2i}$. The matching edges at even positions have no restrictions, and are used to represent the right-neighbor entry in the tuple.

(b) entries do not intersect. That is, for $i, j : 1 \leq i, j \leq k, i \neq j$, $v_{2j} < v_{2i-1}$ or $v_{2i} < v_{2j-1}$.

(6) If a nesting path (representing a tuple t) intersects an entry (corresponding to a tuple t') then the tuple t must be completely contained inside tuple t'. Formally, if $v_1 \curvearrowright v_2 \curvearrowright \ldots \curvearrowright v_{2k}$ and $u_1 \curvearrowright u_2 \curvearrowright \ldots \curvearrowright u_{2k'}$ both are maximal $\curvearrowright$-connected paths and if ($u_{2i-1} < v_{2j} < u_{2i}$ or $u_{2i-1} < v_{2j-1} < u_{2i}$) for some $i : 1 \leq i \leq k'$ and $j : 1 \leq j \leq k$, then for all $j : 1 \leq j \leq k$ there exists $i : 1 \leq i \leq k'$ such that $u_{2i-1} < v_{2j-1} < v_{2j} < u_{2i}$.

A matching relation $\curvearrowright$ is called *well-nesting* if it satisfies all the above properties. There is sentence $\phi_{\text{well-nesting}}$ in $\mathsf{MSO}(\Sigma, \{\curvearrowright\})$ (in fact in $\mathsf{FO}(\Sigma, <, \curvearrowright)$) which asserts that the relation $\curvearrowright$ is well-nesting. See Appendix F.2 for the concrete formula.

Ensuring a valid parsetree. We assume well-nesting of $\curvearrowright$. Given an MCFG $\mathcal{G} = (\Sigma, N, X, P, A_{\text{start}})$, our $\mathsf{MSO}(\Sigma, \{\curvearrowright\})$ formula asserts the following.

(1) There is a partition of the ϵ-labelled vertices into $|N|$ many parts $(Y_A)_{A \in N}$. All vertices in a $\curvearrowright$-connected path will belong to the same part.

Hence it makes sense to talk about A-paths, for $A \in N$.

(2) The length of a maximal $\curvearrowright$-connected path labeled X_A (called an A-path) will be $2 \times \text{arity}(A)$.

(3) For each nonterminal A with $\text{arity}(A) = n$, we require the following condition. For all A-paths $y_1^b \curvearrowright y_1^e \curvearrowright y_2^b \curvearrowright y_2^e \ldots y_n^b \curvearrowright y_n^e$ (i.e., we use the first-order variable y_i^b to indicate the left boundary vertex of the ith entry, and y_i^e to indicate the right boundary vertex) the following conditions must hold for at least one production rule π. For a production rule π of the form (Prod) we require the existence of the boundaries for the arguments on the RHS, $\exists x_{1,1}^b, x_{1,1}^e, \ldots x_{1,n_1}^b, x_{1,n_1}^e, \ldots, x_{m,1}^b, x_{m,1}^e, \ldots x_{m,n_m}^b, x_{m,n_m}^e$, such that

(a) for all $i : 1 \leq i \leq n$, the the path from y_i^b to y_i^e that takes the nesting edge $x_{j,j'}^b \curvearrowright x_{j,j'}^e$ as a shortcut whenever possible (that is, this path corresponds to the topmost level within the nesting edge $y_i^b \curvearrowright y_i^e$ in our pictorial depictions), should be identical to s_i after the following slight modification: for all $j : 1 \leq j \leq m$, for all $j' : 1 \leq j' \leq n_j$, if $x_{j,j'}$ occurs in s_i it is replaced with $x_{j,j'}^b \curvearrowright x_{j,j'}^e$. For example, if $s_4 = bx_{2,1}abx_{1,3}$ in the production rule (Prod) then we will have the following as one of the conjuncts: $\exists z_1 \exists z_2 \exists z_3 (b(z_1) \wedge a(z_2) \wedge b(z_3) \wedge y_4^b \rightarrow z_1 \rightarrow x_{2,1}^b \curvearrowright x_{2,1}^e \rightarrow z_2 \rightarrow z_3 \rightarrow x_{1,3}^b \curvearrowright x_{1,3}^e \rightarrow y_4^e)$.

(b) for all $j : 1 \leq j \leq m$, the path $x_{j,1}^b \curvearrowright x_{j,1}^e \curvearrowright x_{j,2}^b \ldots \curvearrowright x_{j,n_j}^e$ is a maximal B_j-path.

(4) The entire nested-word is wrapped within an A_{start}-path.

All of the above conditions are expressible in the existential fragment of $MSO(\Sigma, \{\curvearrowright\})$, see Appendix F.3 for details. Call that sentence $\phi_{\mathcal{G}}$. Our final formula would then be

$$\phi \equiv \phi_{\text{well-nesting}} \wedge \phi_{\mathcal{G}}.$$

The correctness of the above formula, as well as the bound on special treewidth stated in Theorem 3.3 are proved in Appendix F.4.

6 Downward closures of MCFGs

In this section, we prove Theorem 3.4. For an intuition, we first give a rough description of Courcelle's downward closure construction for context-free grammars [29] (to be precise, its formulation in [7]). Imagine we are provided with a grammar recognizing the language $\{(ab)^n \#(ba)^n \mid n \in \mathbb{N}\}$:

$$S \rightarrow A \mid \# \qquad A \rightarrow aBa \qquad B \rightarrow bSb$$

We can see that by repeating the derivation sequence $S \Rightarrow A \Rightarrow aBa \Rightarrow abSba$, we can produce an arbitrary number of ab to the left of an S, and an arbitrary number of ba to the right. Since the downward closure can drop arbitrary letters, we can furthermore ignore the ordering of letters and

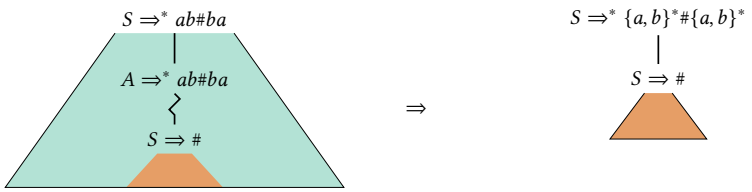


Fig. 5. Courcelle's construction for CFG: A repeated nonterminal (S) gets contracted by replacing the blue subtree with the Kleene closure of a suitable alphabet ($\{a,b\}^*$). The orange subtree is left unchanged.

the equal number of repetitions on either side. In other words, a sufficient number of repetitions allow us to produce arbitrary words from $\{a,b\}^*$ to the left and right of S.

Courcelle's construction can be viewed as introducing *accelerating productions* that capture this behavior. In our example, the productions $S \to A$, $A \to aBa$ and $B \to bSb$ would justify the introduction of the accelerating production $S \to \{a,b\}^*S\{a,b\}^*$ (see also Fig. 5). These are technically not context-free, but their semantics is clear. More generally, for every non-terminal A, we introduce a production $A \to \Sigma^*_{A,\text{left}} A \Sigma^*_{A,\text{right}}$, where $\Sigma_{A,\text{left}}$ (resp. $\Sigma_{A,\text{right}}$) is the set of letters that can appear in u (resp. v) in derivations $A \Rightarrow^* uAv$.

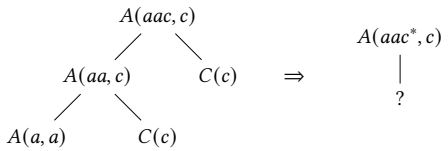
Such acceleration productions are *complete*, in the sense that one can show that every derivation tree can be cut down (by replacing pieces of the derivation tree by accelerating productions) to one of bounded height. Conversely, they are also *sound*: a production of the form $A \to \Sigma^*_{A,\text{left}} A \Sigma^*_{A,\text{right}}$ will only produce words in the downward closure. Soundness and completeness allows us to construct an NFA that just simulates bounded-height trees. This NFA is of exponential size and its language has the same downward closure as the input grammar. One can then easily modify the NFA so as to accept exactly the downward closure.

Ideally, we would like to transfer this approach to the setting of MCFGs. However, there are several reasons that make this a significant challenge.

(1) First of all, as nonterminals have multiple entries, a repetition of nonterminals may support acceleration only in some entries. As an example, consider the grammar

$$A(xy,z) \leftarrow A(x,y), C(z) \qquad A(a,a) \leftarrow \emptyset \qquad C(c) \leftarrow \emptyset$$

and the following derivation tree. Note that repetition of As allow an acceleration of cs in the first entry, but the second entry is fixed to a single c.



(2) Second, even when acceleration is restricted to a subset of the entries, the act of acceleration may restrict possible values in the remaining entries. Consider, for example, if we augment our grammar from the previous point by the productions $\pi : A(x,y) \leftarrow D(x), D(y)$ and $D(d) \leftarrow \emptyset$. In general, this allows the second entry of an A to contain the letter d. However, after applying π, we will not be able to repeat any As. Therefore in any sequence of productions that permits acceleration, the second entry must in the end contain a c. We thus must take care to not permit such constructions.

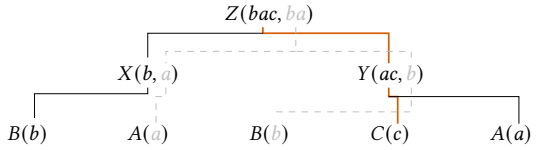
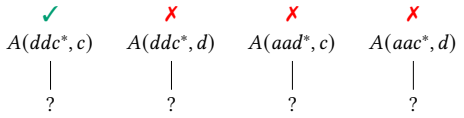
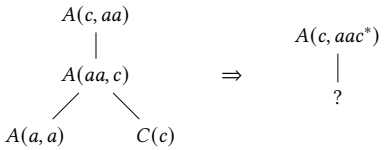


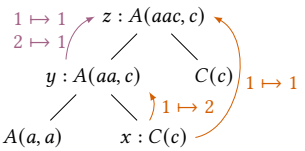
Fig. 6. The embedding map of C into Z (orange, bold), and its watershed (black, solid). The derivation of the second entries of Z (light gray, dashed) is what we consider to occur "alongside" or "in parallel" to the watershed.



(3) Lastly, we observe that production rules are allowed to shuffle entries, e.g. $\pi : A(x, y) \leftarrow A(y, x)$. Our acceleration procedure needs to account for such productions.



To approach these challenges, we search not only for repeating nonterminals, but also look at their *embedding maps*. These embedding maps track the flow of information through the entries in a derivation tree. For example, in the following derivation tree, the embedding map of x into y is $1 \mapsto 2$ and the embedding map from x into z is $1 \mapsto 1$.



These maps help us identify the acceleratable entries of Item 1: acceleration will happen naturally along the information flow indicated by the map as additional letters join this flow from the sides. Informally, we will call this the *watershed* of the embedding map, and our tree cutting procedure will cut out (part of) the watershed, but keep entries derived *in parallel* to the watershed intact, see Fig. 6. We formalize this notion when we give the explicit construction of accelerated grammars in Section 6.3.

As a simplified example of how we use embedding maps, consider Fig. 7. We identify an embedding map of $\{1, 2, 3\} \mapsto 1$ (not shown) between the A nonterminals. Acceleration justifies the removal of parts of the purple and green subtrees where entries are part of the watershed (i.e. they themselves embed into the first entry of the root nonterminal). The root nonterminal is also where we introduce our accelerated production, specifically in the first entry. The orange subtree may have non-acceleratable content. To keep it intact, we project the green subtree to the first entry (containing, by the embedding map, all entries of the orange subtree) and lift it below the root A.

A particularly interesting instance is the case where the embedding map is *idempotent*, that is $f \circ f = f$. Such maps have the property that there is at least one fixpoint x with $f(x) = x$, and all other points are mapped to one of these fixpoints. This eliminates the shuffling we noted in Item 3.

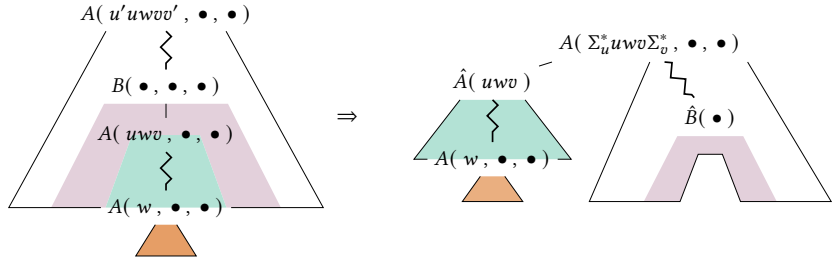


Fig. 7. Downward closure construction for MCFGs. Assume the embedding f between the nonterminals A has a single fixpoint in the first entry (in general, there may be more). The teal subtree rooted at the middle A is moved to the root, but projected to the parts that are relevant for the contents of the fixpoint. In particular, this guarantees that all the (potentially non-acceleratable) entries of the orange subtree are fully contained. The subtree rooted at B is projected to contain only those nodes that do not contribute to the fixpoint. (Our actual construction is slightly more involved, but only to simplify the proof that the tree gets smaller.)

We will later see that restricting ourselves to these idempotent embeddings only adds a constant factor to the size of our downward closure automaton.

Our most complicated adjustments come from the dependencies mentioned in Item 2. We are retaining the parts of the trees that are not part of the watershed. This requires some modification of the underlying grammar, and we need to make sure that this modification only allows us to generate infixes that are generated *alongside* the watershed. For this, we introduce the notion of "decorated" nonterminals. A decorated nonterminal behaves almost identical to its plain version from the original grammar, but is augmented slightly. In our case, we are primarily interested in ensuring that a decorated nonterminal produces a derivation tree if and only if the plain version would have produced a derivation tree with a specific embedding.

An additional challenge is that decorated nonterminals will themselves also need decorated variants (so that we can repeatedly cut trees until all paths are short), and so forth. However, since decoration will always strictly decrease arity (decorated variant of A will only simulate non-fixpoint entries of A, and an idempotent has at least one fixpoint), and we assume that the dimension of the input MCFG is fixed, we will only need to perform decoration a constant number of times, ensuring a merely polynomial blow-up.

6.1 Nonterminal Embeddings

Let us formalize embedding maps now. We will initially define them within individual productions for convenience, though the notion lifts quite naturally to nodes in derivation trees, which is the setting for the maps that we will be primarily concerned with for the remainder of the section.

Let $\pi = A(s_1, \dots, s_n) \leftarrow S$ be a production and $B(x_1, \dots, x_m) \in S$. We say that $B(\vec{x})$ embeds into $A(\vec{s})$ with map $f : [1, m] \to [1, n]$, written $B(\vec{x}) \hookrightarrow_{\pi,f} A(\vec{s})$ if $\mathrm{tgt}(x_i) = s_{f(i)}$. Where π is clear, we may omit it. As these embeddings are closed under composition, we can extend this notion to derivation trees as expected.

Given two maps $f : [1, m] \to [1, n]$ and $g : [1, \ell] \to [1, n]$ we write f/g for the set of functions $h : [1, m] \to [1, \ell]$ satisfying $f = g \circ h$.

6.2 Decorating Nonterminals

As we have mentioned above, we will extend the grammar in order to support the cutting of trees; in order to make sure this extension is not overly broad we "decorate" nonterminals in order to

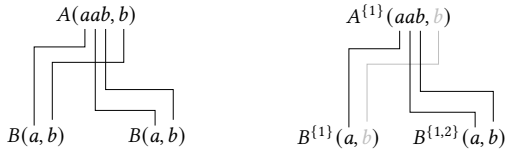


Fig. 8. A derivation tree and its projection. Lines between nodes represent the embedding maps. We project the root nonterminal to its first entry. This means that the righthand B is kept intact, as all its entries map to the first entry of A. The lefthand B is projected to its first entry, as the second is no longer used. In a larger tree, this would cause the children of the left B to be projected in turn.

derive more restrictive behavior. Decorated nonterminals and productions are syntactically derived from existing ones and largely share the semantics of their original counterparts. For example, one of our decorations will be the projection of nonterminals to a subset of their entries. The other is obtained by annotating an existing nonterminal with a target nonterminal and target embedding, ensuring that the decorated nonterminal is productive if and only if there is a derivation sequence from the original nonterminal leading to the target nonterminal with the specified embedding.

Projective Decorations. We will call the first kind of decoration a *projective decoration*. Given a nonterminal A of arity n and a subset of indices $I = \{i_1, \ldots, i_m\} \subseteq [1, n]$ we introduce the projected nonterminal A^I of arity $|I|$. The semantics will be such that $A^I \to^* (w_{i_1}, \ldots, w_{i_m})$ if and only if $A \to^* (w_1, \ldots, w_n)$.

For every production $\pi = A(s_1, \ldots, s_n) \leftarrow S$ we introduce a production $\pi^I = A^I(s_{i_1}, \ldots, s_{i_m}) \leftarrow S'$ where $B^J(z_{j_1}, \ldots, z_{j_{m'}}) \in S'$ if and only if $B(z_1, \ldots, z_{n'}) \hookrightarrow_{\pi,f} A(\vec{s})$ and $J = f^{-1}(I) = \{j_1, \ldots, j_{m'}\}$. We also extend this notion to derivation trees in the expected way: If t is a tree with root (A, π) then t^I is the tree with root (A^I, π^I) whose children are obtained by the appropriate projection of the children of t. As a visual guide, refer to Fig. 8

Searching Decorations. The second kind of decoration ensures the presence of a particular nonterminal and embedding map inside a derivation tree. As such, we call these decorations *searching*. As we will talk about the relationship between subtrees a lot in this section, we will find it useful to introduce a notion of addressing subtrees by the path required to reach them. We address subtrees using paths $p \in \mathbb{N}^*$. If $p = \varepsilon$, then the subtree of a tree t at p, written t/p, is t. If $p = i \cdot p'$, then t/p is t_i/p', where t_i is the i-th child of t.

Let A and B be nonterminals with arity n_0 and m respectively, and let $f : [1, m] \to [1, n_0]$. Let $I = \mathrm{Img}(f) = \{i_1, \ldots, i_m\}$ and $\bar{I} = [1, n_0] \setminus I = \{\bar{i}_1, \ldots, \bar{i}_{n_1}\}$. We introduce a new nonterminal $A^{B,f}$ of arity $|\bar{I}| = n_1$. Intuitively, $A^{B,f}$ will faithfully recreate those entries of A that are generated alongside the watershed of a B-rooted subtree with embedding map f, while ignoring those in the image of the map. In other words, the semantics are such that an $A^{B,f}$-rooted derivation tree t with $\mathrm{yield}(t) = (w_{\bar{i}_1}, \ldots, w_{\bar{i}_{n_1}})$ exists if and only if an A-rooted derivation tree r with $\mathrm{yield}(r) = (w_1, \ldots, w_{n_0})$ and a B-rooted subtree r/p with $r/p \hookrightarrow_f r$ exists for some path p.

To go with our decorated nonterminals, we need decorated productions. Each nonterminal embeds trivially into itself with the identity embedding, so we have $A^{A,\mathrm{id}}() \leftarrow \emptyset$ for all nonterminals A. For more complicated embeddings f, we introduce for every production $\pi = A(\vec{s}) \leftarrow S$ a number of productions that each choose one nonterminal C_i from the body S and a suitable embedding and pursues a search in that branch of the derivation tree. Among all such productions, the choice of C_i is then nondeterministic. More specifically, for every $C_i(\vec{x}) \in S$ with $C_i(\vec{x}) \hookrightarrow_{\pi,g} A(\vec{s})$, and every $h \in f/g$, we first introduce the production $\pi^{B,f,i,h} = A^{B,f}(\vec{s}) \leftarrow S'$. For every $C_j(\vec{y}) \in S$ with $j \neq i$

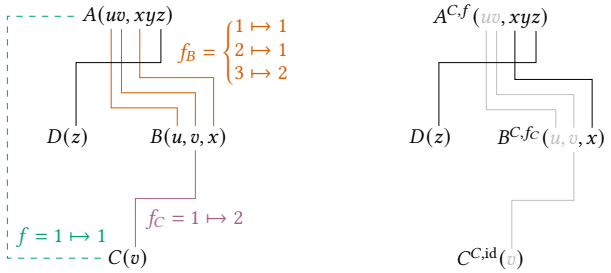


Fig. 9. A visualization of a derivation tree and a search-decoration of it. Lines represent embedding maps. The subtree with root C embeds into the root with the map $f = 1 \mapsto 1$. On the right, we construct the tree $t^{p,f}$, with $p = 21$ being the path to C. To this end, the root is decorated with the nonterminal, C, and the embedding, f. As the image of f is 1, we project away the first entry. We repeat the same procedure with the children. Note that $f_C \in f/f_B$ and therefore the constructed tree is valid according to the decorated production rules we derived.

we add $C_j(\vec{y})$ to S', and we also add $C_i^{B,h}(\vec{x})$. However, these productions currently use the wrong arity for $A^{B,f}$, and we are only interested in entries that do not contain information from B anyway. Therefore, we finally replace $A^{B,f}$ and each of its productions by the version projected to $\bar{I}$.

Let us now explain how to construct a corresponding tree of searching nonterminals, when given a concrete derivation tree t_0 where we know that a subtree at some path p embeds with f into the root of t_0. As we know the exact position p in this case, we will use it to parameterize the following operation: Let t_0 be an A-rooted tree with n children and $t_0/j \hookrightarrow_{f_j} t_0$ for $j \leq n$. Let $t_1 = t_0/p$ be a B-rooted subtree with $t_1 \hookrightarrow_f t_0$. We construct the tree $t_0^{p,f}$ as follows: If $p = \varepsilon$, then $f = \mathrm{id}$ and $t_0^{\varepsilon,\mathrm{id}}$ is just $A^{\emptyset}() \leftarrow \emptyset$. Otherwise, $p = i \cdot p'$. Let h be the embedding such that $t_1 \hookrightarrow_h t_0/i$. The tree $t_0^{p,f}$ is given by the root node labeled $A^{B,f}$ using the production $\pi^{B,f,i,h}$. The children are, for $j \neq i$, $(t_0/j)^J$ with $J = f_j^{-1}(\bar{I})$. For $j = i$ we use $((t_0/i)^{p',h})^J$ as the ith child node, where again $J = f_i^{-1}(\bar{I})$. As a helpful visualization, refer to Fig. 9.

Termination. Given a grammar of size n whose arity is bounded by k, our decorations introduce $O(n^2 \cdot k^k)$ new nonterminals and for each such nonterminal, up to $O(n^2 \cdot k^k)$ new productions and correspondingly many new variables. For fixed k, the size of the new grammar is therefore $O(n^4)$. However, as we introduced new nonterminals, we need to iterate the decoration procedure. We may assume that every decorated nonterminal has strictly smaller arity than its corresponding nondecorated version. Therefore we need to repeat the construction at most k times, leading to a final grammar size of $O(n^{4^k})$, polynomial in n for fixed k.

We conclude with the following technical lemma that shows the correspondence between languages generate by the projected and original nonterminals.

Lemma 6.1. *Let A be a nonterminal of arity n and $I = \{i_1, \ldots, i_m\} \subseteq [1, n]$ a subset of indices. We have $(w_1, \ldots, w_n) \in L(A)$ if and only if $(w_{i_1}, \ldots, w_{i_m}) \in L(A^I)$.*

Proof. The only-if direction follows directly from the construction of A^I.

For the other direction, if A^I can generate $(w_{i_1}, \ldots, w_{i_m})$ directly via some terminal production π^I, then there is a corresponding terminal production π for A and the statement immediately holds.

Let therefore t be an A^I-rooted derivation tree whose first level corresponds to some production $\pi^I = A^I(s_{i_1}, \ldots, s_{i_m}) \leftarrow S'$. Each nonterminal in S' is of the form $C_i^{f_i^{-1}(I)}(\vec{x}_i)$. Let $\vec{u}_i = (u_{i,i_1}, \ldots, u_{i,i_{m'}})$ be the output of the i-th subtree of t. By induction, we have $\vec{u}_i$ in $L(C_i)$.

We construct a tree $\tilde{t}$ for A, using at the first level the production π. For the i-th child, we use the tree that yields the tuple $\vec{u}_i$. Let $(w'_1, \ldots, w'_n)$ be the output of $\tilde{t}$. If $i \in I$, then by construction the set of variables from the j-th child used in s_i is contained in $f_j^{-1}(I)$, and therefore $w'_i = w_i$. □

Observing in this proof that $\tilde{t}$ projected to I is exactly t, we obtain the following as a corollary.

Corollary 6.2. *Let t be a derivation tree with yield $(u_1, \ldots, u_n)$. Let $I = \{i_1, \ldots, i_m\}$ be a set of indices. Then* $\mathrm{yield}(t^I) = (u_{i_1}, \ldots, u_{i_m})$.

6.3 Accelerated Grammars

As the acceleration of entries introduces languages of the shape Γ^* for $\Gamma \subseteq \Sigma$, we introduce here a generalization of MCFGs, called accelerated MCFGs: In an accelerated MCFG $\mathcal{G}$, construction rules for entries may not only use symbols $a \in \Sigma$ and variables $x \in X$, but also languages of the shape Γ^* for $\Gamma \subseteq \Sigma$. The yield of a tree of $\mathcal{G}$ is therefore a tuple of *languages*, and we say that a tuple of words $(w_1, \ldots, w_n)$ is in the language of a nonterminal A if $A \to^* L_1, \ldots, L_n$ and $w_i \in L_i$.

We can construct an accelerated MCFG $\mathcal{G}_\Sigma$ from a base MCFG $\mathcal{G}$. Intuitively, for any nonterminal A we would like to be able to guess an idempotent map f and a path p such that there exists an A-rooted tree t such that the subtree t/p is also A-rooted and embeds with f. As we have discussed above, we then introduce productions that

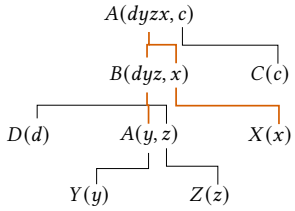
- project away the watershed,
- introduce a searching decoration along the first step of p, and
- introduce a copy of A to provide the possibly non-acceleratable parts of the fixpoints.

Formally, for every arity-n nonterminal A, every idempotent $f : [1, n] \to [1, n]$, every production $\pi = A(s_1, \ldots, s_n) \leftarrow S$, every nonterminal $C_i(\vec{x})$ with $C_i(\vec{x}) \hookrightarrow_{\pi, f_i} A(\vec{s})$ and every $g \in f/f_i$, we introduce a production $\pi^{f,i,g} = A(s'_1, \ldots, s'_n) \leftarrow S'$ (for fixed dimension, this adds only polynomially many productions). Recall that $I = \mathrm{Img}(f)$ and $\bar{I} = [1, n] \setminus I$. The set S' will contain the following:

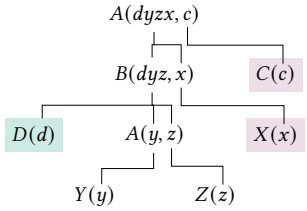
- *project away the watershed:* for every nonterminal $C_j(\vec{y}) \in S$ with $j \neq i$, we introduce a projected version $C_j^{I_j}(\vec{y})$, where $I_j = f_j^{-1}(\bar{I})$;
- *introduce a searching decoration:* for the case of $j = i$ specifically, we introduce the projected searching nonterminal $(C_i^{A,g})^{I_i}(\vec{x})$, which searches for an A-rooted subtree that embeds into C_i with g, and is projected to $I_i = f_i^{-1}(\bar{I})$;
- *lift A:* to derive the non-acceleratable parts of the fixpoints, we introduce $A^I(z_{i_1}, \ldots, z_{i_m}) \in S$.

We also need to modify the construction rules for the individual entries. Since the entries $i \in \bar{I}$ are those that are not accelerated, we set $s'_i = s_i$. For the accelerated entries, that is for $i \in I$, we set $s'_i = \Sigma^*_{i,\mathrm{left}} z_i \Sigma^*_{i,\mathrm{right}}$. Intuitively, these alphabets should contain exactly those letters that can be produced on the watershed of an idempotent embedding and hence we call them *watershed alphabets*.

Watershed Alphabets. To formalize watershed alphabets, we introduce the notion of a *join point* of two entries in a derivation tree, that is, the point in the tree where they first occur in the same entry of the same node. If a derivation tree with an idempotent embedding exists and some letter joins the the fixpoint somewhere inbetween the two nonterminals, we add that letter to the watershed alphabet of that fixpoint. However, since we cannot guarantee that anything rooted below the bottom nonterminal can be accelerated, we need to ignore those letters (see Fig. 10).



(a) The first entry of $t/12$ and the first entry of $t/13$ meet in the first entry of t.



(b) The first entry of A is the fixpoint of the embedding. d will be in $\Sigma_{1,\text{left}}$ (it joins from the left in $t/1$). x will be in $\Sigma_{1,\text{right}}$ (it joins from the right in t). c, despite not joining in the displayed tree, will be in $\Sigma_{1,\text{right}}$ as we can easily construct a larger tree where it does. Note that z is not in $\Sigma_{1,\text{right}}$ despite joining from the right, because it can only ever occur rooted below the lowest A.

Fig. 10. The construction of watershed alphabets.

All these conditions can be checked with a simple graph search in the connectivity graph of the grammar.

Formally, let t be a tree, t/p_1 an A_1-rooted and t/p_2 an A_2-rooted subtree of arity n_1 and n_2 respectively and $\ell_1 \in [1, n_1]$, $\ell_2 \in [1, n_2]$. We say that the ℓ_1-th entry of t/p_1 and the ℓ_2-th entry of t/p_2 *join at* t/q *in entry* ℓ if $t/p_1 \hookrightarrow_{f_1} t/q \hookleftarrow_{f_2} t/p_2$ with $f_1(\ell_1) = \ell = f_2(\ell_2)$ and q is the longest path for which this condition holds.

To decide whether a letter should be a part of the left or the right watershed alphabet, $\Sigma_{i,\text{left}}$ or $\Sigma_{i,\text{right}}$, we also need a notion of order. Let $B(\vec{s}) \leftarrow S$ be the root production of to t/q. Let $C_i(\vec{x})$ and $C_j(\vec{y}) \in S$ be the nonterminals of the (not necessarily distinct) i-th and j-th child of t/q such that $t/p_1 \hookrightarrow_{g_1} t/(q \cdot i)$ and $t/p_2 \hookrightarrow_{g_2} t/(q \cdot j)$. We say that the ℓ_1-th entry of t/p_1 joins the ℓ_2-th entry of t/p_2 *from the left* if $x_{g_1(\ell_1)} y_{g_2(\ell_2)} \preceq s_\ell$, or joins *from the right* if $y_{g_2(\ell_2)} x_{g_1(\ell_1)} \preceq s_\ell$.

We may assume that terminal letters are only introduced individually and by leaf nodes. We denote by $L_a(a)$ a nonterminal introducing the letter a as such. We say that a letter a is in $\Sigma_{i,\text{left}}$ if there exists an A-rooted tree t with an A-rooted subtree t/p and an L_a-rooted subtree t/q such that there is an idempotent embedding $t/p \hookrightarrow_f t$ and (the only entry of) t/q joins the i-th entry of t/p from the left. Additionally, we require that t/q is not a child of t/p.

6.4 Concise Overapproximations

Having introduced the notion of accelerated grammars and described how to construct an accelerated grammar $\mathcal{G}_\Sigma$ from a standard MCFG $\mathcal{G}$, we show that $\mathcal{G}_\Sigma$ does not only overapproximate $\mathcal{G}$, but that the two are in fact equivalent with respect to their downward closure. Let $\preceq^n$ denote the subword ordering lifted to tuples of words of dimension n.

Lemma 6.3. *$L(\mathcal{G}) \subseteq L(\mathcal{G}_\Sigma)$, and for every tree t' of $\mathcal{G}_\Sigma$ and every word $(w_1, \ldots, w_n) \in L(t')$ there is a tree t of $\mathcal{G}$ such that $(w_1, \ldots, w_n) \preceq^n \text{yield}(t)$.*

Proof. The first statement follows from the fact that every tree of $\mathcal{G}$ is also a valid tree of $\mathcal{G}_\Sigma$.

For the second statement, observe that if t' contains no accelerated productions, then it will also not contain any decorated nonterminals. In that case t' is also a tree of $\mathcal{G}$ and the statement holds.

We therefore show how to eliminate accelerated productions. WLOG assume that the root production is an accelerated rule $\pi^{f,i,g} = A(s_1, \ldots, s_k) \leftarrow S'$, derived from a production $\pi = A(s_1, \ldots, s_k) \leftarrow S \in \mathcal{G}$. Let $(w_1, \ldots, w_n) \in \text{yield}(t')$.

From Lemma 6.1 we know that if the i-th child of t' is a B_i^I-rooted tree with yield $(u_i)_{i\in I}$ then there exists a B_i-rooted tree with yield $(u_1, \ldots, u_m)$. Similarly, if $(v_i)_{i\in J}$ is the yield of the $C^{A,g}$-rooted child, then there exists a C-rooted tree whose yield is $(v_1, \ldots, v_n)$. We can therefore construct $\hat{t} = A[t_1, \ldots, t_{n-1}, t_n]$ with t_i being the B_i-rooted tree and t_n being the C-rooted tree.

Observe that for all $i \notin \mathrm{Img}(f)$, this construction already satisfies the requirement by having $\mathrm{yield}(\hat{t})[i] = w_i$. Furthermore, there is a path p in t_n such that the subtree $r = t_n/p$ is A-rooted and embeds with g into t_n and hence with f into $\hat{t}$. We will therefore aim to substitute r with an A-rooted subtree with yield $(z_1, \ldots, z_n)$ such that for $i \in \mathrm{Img}(f)$, $w_i \preceq z_i$.

To this end, let $\hat{r}$ be the A^I-rooted child of t' that corresponds to the nonterminal $A^I((y_i)_{i\in I})$ that was newly added to the body of $\pi^{f,i,g}$ during its construction. We have $\mathrm{yield}(\hat{r}) = (o_{i_1}, \ldots, o_{i_k})$. Additionally, we know that $w_i = uo_iv$ for $u \in \Sigma^*_{i,\mathrm{left}}, v \in \Sigma^*_{i,\mathrm{right}}$.

We first construct an A-rooted tree $\hat{r}_0$ using Lemma 6.1. This tree satisfies $o_i \preceq \mathrm{yield}(\hat{r}_0)[i]$. After that, we will iteratively construct trees $\hat{r}_1, \hat{r}_2$ and so on that will incorporate successive letters from u or v into the i-th entry.

Let $u = u_1 \cdots u_n$. By the construction of Σ_{left}, this means that there exists an A-rooted tree τ and an A-rooted subtree τ' of τ with $\tau' \hookrightarrow_f \tau$ such that the i-th entry is of the form $u'u_nu''x_iv'$, where x_i is traced back to the i-th entry of τ'. Therefore substituting $\hat{r}_0$ for τ' in τ yields a new tree, $\hat{r}_1$, whose output in the i-th entry is z with $u_no_i \preceq z$. We iterate this process until we have eliminated all letters from u, and repeat a similar process with the letters of v (note that for v we must proceed from the first letter in order to obtain the correct ordering). The resulting subtree will be r. Finally, we substitute r in $\hat{t}$ at position $n \cdot p$ to obtain t. □

All that remains to show is that accelerated grammars even allow for *concise* representations of downward closures. We show this by showing that every "large" tree of $\mathcal{G}_\Sigma$ is already subsumed by a smaller tree. Some of the proof details can be found in Appendix G.1.

Lemma 6.4. *Fix a bound d on the dimension of $\mathcal{G}_\Sigma$. There exists a constant c such that for every tree t of $\mathcal{G}_\Sigma$ there exists a tree t' of $\mathcal{G}_\Sigma$ such that $L(t)\downarrow \subseteq L(t')\downarrow$ and* $\mathrm{height}(t') \leq c|\mathcal{G}_\Sigma|$.

Proof. Assume first that c exists and we know its value. We proceed by induction on $|t|$. For the empty tree, the statement trivially holds. Assume therefore that the statement holds for all trees with less than m nodes and let t have $m+1$ nodes. If $\mathrm{height}(t) \leq c|\mathcal{G}_\Sigma|$, the statement holds. Otherwise, our choice of c will be such that we are able to find a path p in t such that along p we find four distinct A-rooted subtrees t_0, t_1, t_2, and t_3 such that $t_i \hookrightarrow_f t_{i-1}$ for some idempotent f. In that case, we can use the accelerated rules to cut out part of the tree. The resulting tree may not be shorter, as p may not be the unique longest path in the tree, but it will contain less nodes. For the details of the tree cutting construction, please refer to Appendix G.1.

Our goal therefore is to chose c in a way that guarantees the existence of such subtrees with idempotent embeddings. Let t be a tree, p be a path on t and $\mathcal{S}_A \subseteq \mathrm{prefixes}(p)$ the set of all occurrences of A-labeled nodes along p. We consider the complete graph obtained by using $\mathcal{S}_A$ as a vertex set and color each edge $\{p', p'q\}$ with the embedding f from $t/p'q$ into t/p'. If we find a monochromatic 4-clique $\{p_0, p_0p_1, p_0p_1p_2, p_0p_1p_2p_3\}$, this implies $f \circ f = f$ and therefore f is idempotent. Ramsey's theorem [72] tells us that there exists some value $c(d)$ such that every d-colored complete graph of at least $c(d)$ vertices contains such a monocolored 4-clique. In particular, since we assume the dimension k and therefore the number of embeddings between nonterminals is fixed, $c(d)$ becomes a constant. We conclude by observing that on any path of length $c(d)|\mathcal{G}_\Sigma|$, at least one nonterminal occurs $c(d)$ many times. □

Having established the conciseness of accelerated grammars, all that remains is to show how to construct a small NFA for the downward closure.

Proof of Theorem 3.4. Assume that $|\mathcal{G}| = n$ and the dimension is some fixed k. We construct the corresponding accelerated MCFG $\mathcal{G}'$. The size of $\mathcal{G}'$ is $|\mathcal{G}'| = n^{2^{O(k)}} = \text{poly}(n)$.

It follows that to compute the downward closure of $\mathcal{G}$, it suffices to consider trees of $\mathcal{G}'$ of height at most $\text{poly}(n)$. Such trees have a size of $\text{poly}(n)^{\text{poly}(n)} = 2^{\text{poly}(n)}$. Therefore the yield of such a tree is also of length $2^{\text{poly}(n)}$. As the language of an accelerated grammar is a simple concatenation of individual letters and languages of the shape Γ^* for $\Gamma \subseteq \Sigma$, for a given tree of size m we can compute an $O(m)$-sized NFA accepting its downward closure. Therefore, our final NFA for the downward closure is the union of the NFAs for the trees of $\mathcal{G}'$ of height at most $\text{poly}(n)$. There are $\text{poly}(n)^{\text{poly}(n)^{\text{poly}(n)}} = 2^{2^{\text{poly}(n)}}$ many of those, each of size $2^{\text{poly}(n)}$, yielding a final size of $2^{2^{\text{poly}(n)}}$. □

Note that our polynomial bound on derivation tree heights only yields a doubly exponential NFA upper bound, compared to a singly exponential bound for CFG (because for CFG, all polynomial-height trees can be simulated by a polynomial-stack-height pushdown automaton). As our lower bound shows (Theorem 3.5), the additional blowup for MCFG is unavoidable.

7 Application: Context-Bounded Safety in Concurrent Programs

Let us provide an application of our results beyond the immediate settings of bounded-stw systems and MCFGs. We consider the model of concurrent programs, where each individual concurrent process already computes an MCFL, and more processes can be created dynamically during execution. Here we can adapt our results on downward closures to reason about safety verification. To be precise, we consider safety for concurrent programs over MCFL under the restriction of bounded context switching. This restriction is important, because without it safety becomes undecidable (even if each process only computes a CFL [69, 71]).

To properly talk about this setting, we first need to recall some auxiliary notions.

Multisets and the Parikh image. Since we assume no relations between our concurrent processes, it makes sense to model the collection of their individual configurations as some unordered structure. However, a mere set does not suffice, since that would not properly track duplicate configurations. As such we use the common generalization to multisets, where each potential element is assigned a natural number that specifies how often it occurs. A set can then be seen as a multiset, where every assignment is either to 0 or to 1.

Formally, a *multiset* $\mathbf{m} \colon X \to \mathbb{N}$ over a set X maps each element of X to a natural number. The size $|\mathbf{m}|$ of a multiset $\mathbf{m}$ is defined as $|\mathbf{m}| = \sum_{x \in X} \mathbf{m}(x)$. We write $\mathbb{M}[X]$ to denote set of all multisets over X. For a subset $Y \subseteq X$, we identify it with the multiset in $\mathbb{M}[X]$ mapping each $x \in Y$ to 1 and everything else to 0.

In our model, when several processes are spawned in a row without interruptions in-between, the precise order of these spawns does not matter. As such, let us recall how to obtain an unordered multiset from a sequence of elements: The Parikh image $\Psi(w) \in \mathbb{M}[\Sigma]$ of a word $w \in \Sigma^*$ is the multiset such that for each letter $a \in \Sigma$ we have $\Psi(w)(a) = |w|_a$, where $|w|_a$ denotes the number of occurrences of a in w. When denoting Parikh images, we also identify multisets in $\mathbb{M}[\Sigma]$ with vectors in $\mathbb{N}^{|\Sigma|}$ (for some arbitrary, but fixed, total order on Σ).

Concurrent Programs. We consider the model of concurrent programs, which are defined over an arbitrary class of languages C (with some mild restrictions on C) [13], though here we look at the case where C is the class of MCFLs. The complete definition of the model and its related notions can be found in Appendix J.1; we only give the relevant intuition here.

A *concurrent program* (CP) $\mathfrak{P}$ over MCFLs consists of a set of *global states* D, an alphabet of *process names* Σ, and for each $a \in \Sigma$ a *process language* L_a that is an MCFL (given e.g. as an MCFG). For each process name $a \in \Sigma$, its corresponding language is of the form $L_a \subseteq aD\big(\Sigma \cup (D \times D)\big)^*(D \cup D \times D)$,

where the letters in D denote the *initial* and a possible *final state* of a particular process' execution, the letters in Σ denote its *spawns*, and the letters in $D \times D$ denote its *context switches*.

The program $\mathfrak{P}$ starts in some initial global state d_0 with some initial multiset of processes. To run a process from current state d, $\mathfrak{P}$ nondeterministically schedules a process whose execution either has not yet started and can start in d, or whose execution when it was last scheduled ended in a context switch (e, d) for some global state $e \in D$. The execution is then extended until either a context switch (d', d'') or a final state d' occurs, upon which the current global state transitions to d'. All spawns encountered along the way are added to the multiset of processes, and whenever a process reaches the final state in its execution, it is removed from said multiset. Note that any spawned process with name $a \in \Sigma$ is restricted to executions from the process language L_a.

Bounded context switching. Most decision problems are undecidable for concurrent programs, if we impose no restrictions on their behavior [69, 71]. As such we consider the popular restriction of bounded context switching. Like before, we do not give the full formal definitions from [13], which can also be found in Appendix J.1. Rather, we only give the relevant intuition.

For a number $k \in \mathbb{N}$, also called a *context bound*, a run of a CP is k-context-bounded, if each individual process is scheduled at most $k + 1$ times. Alternatively, one says that each process *experiences* k context switches. Note that one considers only the part of a process' execution that is actually scheduled as experienced, and if the symbol reached after $k + 1$ schedulings is in $D \times D$, it is not counted as an experienced $(k + 1)$th context switch.

Now we can define the decision problem of interest.

Problem 7.1 (Context-bounded safety for concurrent programs).

Given *A unary-encoded context-bound $k \in \mathbb{N}$, a concurrent program $\mathfrak{P}$ over the MCFLs, and one of its global states $d_f \in D$, where each process language L_a of $\mathfrak{P}$ is given as an MCFG of constant dimension and rank.*

Question *Is there a k-context-bounded run of $\mathfrak{P}$ that reaches the global state d_f?*

Theorem 7.2. *Problem 7.1 is* 2EXPSPACE*-complete.*

Note that 2EXPSPACE-hardness already holds already when $k = 1$ and $\mathfrak{P}$ is over the CFLs [15].

The rest of this section is dedicated to proving Theorem 7.2. To give a brief idea, we want to replace each MCFG process with a finite-state process, where the latter overapproximates the former in a way that preserves context-bounded safety, by computing a partially permuting variant of its downward closure (see below). Then, in the finite-state setting, we can use a known algorithm to decide context-bounded safety in EXPSPACE. Since the aforementioned replacement of processes constitutes an exponential blow-up, we get a 2EXPSPACE-procedure in total.

We require constant dimension and rank for the MCFGs in the input, so that we can perform constructions for certain closure properties in polynomial time (see Lemma 7.3 below). However, these requirements are quite natural: For example, when we invoke Theorem 3.2 to turn a constant-width tree automaton into an MCFG, the latter will also have constant dimension, and even a constant rank of 2.

Special downward closures. To prove the above theorem, we need to define (and later compute) some special downward closure variants. The first variant essentially allows us to retain information about all k context switches, but potentially lose any spawns occurring between these switches. For a language $L \subseteq \Gamma^*$, a subalphabet $\Theta \subseteq \Gamma$, and a number k let $L{\downarrow_{k,\Theta}}$ denote its *alphabet-preserving downward closure*, where subwords are not allowed to drop letters from Θ, and have to contain exactly k of them. This is slightly different from the definition in [16], where words with fewer than k letters are also included, but our version fits our purpose slightly better. Formally,

$L{\downarrow_{k,\Theta}} := \{u \in \Gamma* \mid \exists v \in L\colon u \preceq v \wedge |u|_\Theta = |v|_\Theta = k\}$. Here, $|w|_\Theta$ denotes the number of occurrences of letters from the alphabet Θ in the word w.

The next downward closure variant allows us to also lose information about the exact order of spawns in each segment of a process. Formally, we define the *partially permuting downward closure*: Let $L{\downarrow_{k,\Theta}^{\Psi}}$ denote the variant of $L{\downarrow_{k,\Theta}}$, where every maximal infix in $(\Gamma \setminus \Theta)^*$ only needs to be preserved up to its Parikh-image $\Psi(-)$. Formally $u = u_0\theta_1 u_1 \cdots \theta_k u_k \in L{\downarrow_{k,\Theta}^{\Psi}}$ if and only if $u_0, \ldots, u_k \in (\Gamma \setminus \Theta)^*$, $\theta_1, \ldots, \theta_k \in \Theta$, and there is $v = v_0\theta_1 v_1 \cdots \theta_k v_k \in L{\downarrow_{k,\Theta}}$ such that $\Psi(u_i) = \Psi(v_i)$ for every $0 \leq i \leq k$.

Deciding context-bounded safety. We need two more auxiliary lemmas before we can prove Theorem 7.2. The first allows us to compute simple closure properties for MCFGs in polynomial time, provided they have constant dimension and rank. The second computes an NFA for the partially permuting downward closure. In particular, with the relaxation of preservation up to the Parikh-image, we can compute downward closures of MCFGs in EXPSPACE instead of 2EXPSPACE.

Lemma 7.3. *Given an MCFG $\mathcal{G}$ of constant dimension and rank, and an NFA $\mathcal{A}$, we can compute the following in polynomial time:*

(a) an MCFG for the intersection $\mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{A})$, and
(b) an MCFG for the language $\mathrm{pref}(\mathcal{L}(\mathcal{G}))$ *containing all prefixes of words in $\mathcal{L}(\mathcal{G})$.*

We describe the full proofs in Appendix J.2 (see Corollaries J.2 and J.4). To give an idea, the proof of Lemma 7.3(a) is a relatively simple adaptation of the triple construction for intersecting a CFG and an NFA.

For Lemma 7.3(b) the idea is not difficult either, but requires some additional tools from language theory. In fact, we show something more general, namely that given an MCFG $\mathcal{G}$ and a finite-state transducer $\mathcal{T}$, we can compute the image of $\mathcal{L}(\mathcal{G})$ under $\mathcal{T}$ in polynomial time. Here a *(finite-state) transducer* is an NFA that in addition to reading a symbol may also write a symbol on each transition, thus basically generating a pair of input word and output word for each accepting run. It is not hard to see how to construct a transducer that maps any word to each of its prefixes.

To compute the image of $\mathcal{L}(\mathcal{G})$ under $\mathcal{T}$ we construct an intermediary grammar $\mathcal{G}'$ that arbitrarily shuffles transitions of $\mathcal{T}$ between words generated by $\mathcal{G}$. Then we use Lemma 7.3(a) to intersect with an NFA that checks whether the transitions form a valid run over the input word given by the letters of $\mathcal{G}$. Finally, we project away the input letters and replace transitions with their output letters to obtain the desired MCFG. None of this is hard to do in polynomial time.

As a remark, it was previously known that the MCFL are closed under taking the image under a transducer (also called a *rational transduction*) [74], but we could not find a source regarding effective polynomial complexity for our case. Also note that to break up the image under a transducer $\mathcal{T}$ into several steps, we use an adaptation of Nivat's theorem (see e.g. [17, Chapter III, Theorem 3.2]).

Lemma 7.4. *Let $\mathcal{G}$ be an accelerated MCFG with $\Theta \subseteq \Gamma$, where for every tree t of $\mathcal{G}$ there is a tree t' of $\mathcal{G}$, such that $L(t){\downarrow} \subseteq L(t'){\downarrow}$ and* $\mathrm{height}(t') \leq c|\mathcal{G}|$ *for a constant c as in Lemma 6.4. We can compute an NFA for $\mathcal{L}(\mathcal{G}){\downarrow_{k,\Theta}^{\Psi}}$ in time $2^{\mathrm{poly}(k \cdot |\mathcal{G}|)}$.*

Before we argue how to prove Lemma 7.4, let us see how it implies Theorem 7.2. The proof idea is rather simple. We use our downward-closure construction and MCFG closure properties to turn each process language L_a into one that might drop process names, but preserves up to the k first context switches. This replaces each MCFG with an NFA that is at most exponentially larger, meaning it yields a concurrent program over the regular languages. For these, we can decide k-context-bounded safety in EXPSPACE [10], and so we obtain a 2EXPSPACE algorithm for k-context-bounded safety over MCFLs. It is known that context-bounded safety is preserved under

dropping process names, but preserving context-switches [13]. The additional Parikh-relaxation does not change this. The latter is easy to see from just the semantics of concurrent programs.

Proof of Theorem 7.2. The lower bound follows from the special case of CPs over context-free languages, where k-context-bounded reachability was shown to be 2EXPSPACE-hard in [15] for unary-encoded k and even fixed $k \geq 1$.

For the upper bound, consider a CP $\mathfrak{P}$ over the MCFLs with a context bound $k \in \mathbb{N}$, where for each $a \in \Sigma$, L_a is given as an *MCFG* $\mathcal{G}_a$ of constant dimension and rank. Let $\Gamma = \Sigma \cup D \cup (D \times D)$, where D are the global states of $\mathfrak{P}$. Our goal is to compute for each $L_a \subseteq \Gamma^*$ an NFA for its downward closure, that preserves the first up to k context-switches. We accomplish this in the following five steps. First, we construct an MCFG $\mathcal{G}_{a,\mathrm{pref}}$ for $\mathrm{pref}(L_a)$ using Lemma 7.3(b). Second, for each $i = 0$ to k we intersect $\mathcal{G}_{a,\mathrm{pref}}$ with the regular language $aD\big(\Sigma^*(D \times D)\big)^i\Sigma^*(D \cup D \times D)$, which ensures that there are exactly i context switches. Using Lemma 7.3(a), we obtain $k+1$ grammars $\mathcal{G}_{a,\mathrm{pref},0}$ to $\mathcal{G}_{a,\mathrm{pref},k}$. Third, we turn each MCFG $\mathcal{G}_{a,\mathrm{pref},i}$ into a downward-closure-faithful accelerated MCFG $\mathcal{G}'_{a,\mathrm{pref},i}$, as guaranteed by Lemma 6.4. Fourth, for each $\mathcal{G}'_{a,\mathrm{pref},i}$ we compute an NFA $\mathcal{A}_i$ for the downward closure $\mathcal{L}(\mathcal{G}_{a,\mathrm{pref},i})\downarrow^{\Psi}_{i+2,D\cup D\times D}$ using Lemma 7.4. This preserves the first and last letter in $D \cup D \times D$, as well as the i context-switches in between. Finally, we construct an NFA $\mathcal{A}_a$ for the union $\bigcup_{i=0}^{k} \mathcal{L}(\mathcal{A}_{a,i})$ intersected with the regular language $aD\big(\Sigma \cup (D \times D)\big)^*(D \cup D \times D)$. The last intersection is to ensure that words are of the correct form for a process language of a CP.

Almost every step here takes polynomial time; regarding the construction of accelerated grammars $\mathcal{G}'_{a,\mathrm{pref},i}$, Lemma G.1 in Appendix G.1 provides a more detailed time analysis. The exception is invoking Lemma 7.4, which introduces an exponential blow-up. However, no two applications of said lemma are in sequence, so a single exponential blow-up suffices.

Replacing $\mathcal{G}_a$ with $\mathcal{A}_a$ in $\mathfrak{P}$ for each $a \in \Sigma$, we obtain a concurrent program $\mathfrak{P}'$ over the regular languages. Using the algorithm from [10], we can decide k-context-bounded safety in EXPSPACE in this setting (simple reduction to coverability in vector addition systems). Since $\mathfrak{P}'$ may be exponentially larger than $\mathfrak{P}$ and the unary-encoded k due to the aforementioned blow-up, we get an overall complexity of 2EXPSPACE.

Let us now argue correctness of the construction. It is known that k-context-bounded reachability for CP is preserved under taking the downward closure that considers only prefixes with up to k context switches, and is not allowed to lose those context switches [10, 13]. Thus we only need to argue that k-context-bounded reachability is still preserved when we arbitrarily swap the infixes over process names Σ for Parikh-equivalent infixes. This, however, directly follows from the semantics of CP: any time a Σ^*-infix from a language L_a is considered for the step-relation, we first take its Parikh image (see Appendix J.1), meaning the actual order of the letters within such an infix never matters. □

Compression through Parikh images. It remains to prove Lemma 7.4. The idea is to construct an NFA that constructs a derivation tree of the accelerated MCFG $\mathcal{G}$ on the fly. To this end, the automaton picks productions nondeterministically, and explores the derivation tree via depth-first search, storing only the current branch. During this, variables of nonterminals are updated as they become assigned with terminal words. The key trick to obtaining the exponential (rather than doubly-exponential) size bound is that, instead of storing these terminal words directly, it suffices to *store their Parikh images*. Note that for a word w over an alphabet Γ, the Parikh image of w can be stored in $O(|\Gamma| \cdot \log |w|)$ bits, by using binary encodings, whereas storing w completely would require at least $|w|$ bits. Through this compression, the automaton only needs to maintain

polynomially many (instead of exponentially many) bits in its state. This is used when storing infixes consisting of letters in Σ or subalphabets of Σ.

Upon completely exploring the derivation tree, the NFA outputs any word that matches the letters and Parikh images stored for the starting nonterminal and then accepts. Due to our requirement of effectively $(c|\mathcal{G}|)$-bounded tree height, we only need to consider branches of this polynomial length, which also keeps the Parikh images small enough that this NFA only needs exponentially many states. The complete proof can be found in Appendix J.3.

8 Conclusion

Our results establish MCFG as a conceptually simple model that captures a wide range of decidable underapproximations of MPDA and other models. This calls for a broader investigation into algorithmic properties of MCFG. The MCFL reachability problem and the membership problem have been studied in [26, 49–51] (see Page 6). Another interesting direction is *refinement verification*: Can we decide whether the language of a given MCFG is included in a Dyck language? Inclusion in Dyck languages can be used to verify implementations of reference counting or programs that generate code [14]. In the case of MPDA with bounded context-switching (i.e. an example of a bounded-stw underapproximation), this problem was recently shown to be coNP-complete (if the context bound is part of the input) [14]. By our results, an algorithm for MCFG would apply to *all* bounded-stw underapproximations.

Our downward closure construction yields various algorithms for verifying complex systems whose components are specified by MSO-definable bounded-stw languages: one example is our application in Section 7. As mentioned in the introduction, another example is parameterized verification of shared memory systems with non-atomic reads and writes [42, 56]. Yet another example concerns certain networks of infinite-state systems communicating via lossy channels [11]. Specifically, our results yield elementary upper bounds for safety verification where so far, only non-primitive recursive upper bounds were known. This calls for further investigation into the exact complexity of these problems.

We leave open whether there is a doubly-exponential downward closure construction for MCFG when the dimension is part of the input. In that setting, our construction is five-fold exponential when applying Jecker's recent Ramsey bounds for n-point transformation monoids [47].

Acknowledgments

We are grateful to the anonymous reviewers for their helpful suggestions on the presentation.

Funded by the European Union (ERC, FINABIS, 101077902 (https://doi.org/10.3030/101077902)). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Alfred V Aho. 1968. Indexed grammars—an extension of context-free grammars. *Journal of the ACM (JACM)* 15, 4 (1968), 647–671. https://doi.org/10.1145/321479.321488

[2] C. Aiswarya, Paul Gastin, and K. Narayan Kumar. 2014. Verifying Communicating Multi-pushdown Systems via Split-Width. In *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8837)*, Franck Cassez and Jean-François Raskin (Eds.). Springer, 1–17. https://doi.org/10.1007/978-3-319-11936-6_1

[3] S. Akshay, Paul Gastin, and Shankara Narayanan Krishna. 2016. Analyzing Timed Systems Using Tree Automata. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada (LIPIcs, Vol. 59)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:14. https://doi.org/10.4230/LIPICS.CONCUR.2016.27

[4] S. Akshay, Paul Gastin, and Shankara Narayanan Krishna. 2018. Analyzing Timed Systems Using Tree Automata. *Log. Methods Comput. Sci.* 14, 2 (2018). https://doi.org/10.23638/LMCS-14(2:8)2018

[5] S. Akshay, Paul Gastin, Shankara Narayanan Krishna, and Ilias Sarkar. 2017. Towards an Efficient Tree Automata Based Technique for Timed Systems. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany (LIPIcs, Vol. 85)*, Roland Meyer and Uwe Nestmann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 39:1–39:15. https://doi.org/10.4230/LIPICS.CONCUR.2017.39

[6] Rajeev Alur and P. Madhusudan. 2006. Adding Nesting Structure to Words. In *Developments in Language Theory, 10th International Conference, DLT 2006, Santa Barbara, CA, USA, June 26-29, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4036)*, Oscar H. Ibarra and Zhe Dang (Eds.). Springer, 1–13. https://doi.org/10.1007/11779148_1

[7] Ashwani Anand and Georg Zetzsche. 2023. Priority Downward Closures. In *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium (LIPIcs, Vol. 279)*, Guillermo A. Pérez and Jean-François Raskin (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 39:1–39:18. https://doi.org/10.4230/LIPICS.CONCUR.2023.39

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[9] Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2012. Linear-Time Model-Checking for Multithreaded Programs under Scope-Bounding. In *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7561)*, Supratik Chakraborty and Madhavan Mukund (Eds.). Springer, 152–166. https://doi.org/10.1007/978-3-642-33386-6_13

[10] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. 2011. Context-Bounded Analysis For Concurrent Programs With Dynamic Creation of Threads. *Log. Methods Comput. Sci.* 7, 4 (2011). https://doi.org/10.2168/LMCS-7(4:4)2011

[11] Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili. 2008. On the Reachability Analysis of Acyclic Networks of Pushdown Systems. In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5201)*, Franck van Breugel and Marsha Chechik (Eds.). Springer, 356–371. https://doi.org/10.1007/978-3-540-85361-9_29

[12] David Barozzini, Lorenzo Clemente, Thomas Colcombet, and Pawel Parys. 2020. Cost Automata, Safe Schemes, and Downward Closures. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference) (LIPIcs, Vol. 168)*, Artur Czumaj, Anuj Dawar, and Emanuela Merelli (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 109:1–109:18. https://doi.org/10.4230/LIPICS.ICALP.2020.109

[13] Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2023. Context-Bounded Analysis of Concurrent Programs (Invited Talk). In *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany (LIPIcs, Vol. 261)*, Kousha Etessami, Uriel Feige, and Gabriele Puppis (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:16. https://doi.org/10.4230/LIPICS.ICALP.2023.3

[14] Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2023. Context-Bounded Verification of Context-Free Specifications. *Proc. ACM Program. Lang.* 7, POPL (2023), 2141–2170. https://doi.org/10.1145/3571266

[15] Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2020. The Complexity of Bounded Context Switching with Dynamic Thread Creation. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference) (LIPIcs, Vol. 168)*, Artur Czumaj, Anuj Dawar, and Emanuela Merelli (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 111:1–111:16. https://doi.org/10.4230/LIPICS.ICALP.2020.111

[16] Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2022. Context-bounded verification of thread pools. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498678

[17] Jean Berstel. 1979. *Transductions and context-free languages.* Springer-Verlag. https://doi.org/10.1007/978-3-663-09367-1

[18] Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. 1996. Multi-Push-Down Languages and Grammars. *Int. J. Found. Comput. Sci.* 7, 3 (1996), 253–292. https://doi.org/10.1142/S0129054196000191

[19] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2020. Optimal and Perfectly Parallel Algorithms for On-demand Data-Flow Analysis. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 112–140. https://doi.org/10.1007/978-3-030-44914-8_5

[20] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2017. JTDec: A Tool for Tree Decompositions in Soot. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10482)*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer, 59–66. https://doi.org/10.1007/978-3-319-68167-2_4

[21] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2018. Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components. *ACM Trans. Program. Lang. Syst.* 40, 3 (2018), 9:1–9:43. https://doi.org/10.1145/3210257

[22] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Optimal Reachability and a Space-Time Tradeoff for Distance Queries in Constant-Treewidth Graphs. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark (LIPIcs, Vol. 57)*, Piotr Sankowski and Christos D. Zaroliagis (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:17. https://doi.org/10.4230/LIPICS.ESA.2016.28

[23] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal. 2015. Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 97–109. https://doi.org/10.1145/2676726.2676979

[24] Ben-Chung Cheng and Wen-mei W. Hwu. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 57–69. https://doi.org/10.1145/349299.349311

[25] Lorenzo Clemente, Pawel Parys, Sylvain Salvati, and Igor Walukiewicz. 2016. The Diagonal Problem for Higher-Order Recursion Schemes is Decidable. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 96–105. https://doi.org/10.1145/2933575.2934527

[26] Giovanna Kobus Conrado, Adam Husted Kjelstrøm, Jaco van de Pol, and Andreas Pavlogiannis. 2025. Program Analysis via Multiple Context Free Language Reachability. *Proc. ACM Program. Lang.* 9, POPL (2025), 509–538. https://doi.org/10.1145/3704854

[27] Giovanna Kobus Conrado and Andreas Pavlogiannis. 2025. CFL-based methods for approximating interleaved Dyck reachability. *Int. J. Softw. Tools Technol. Transf.* 27, 2 (2025), 255–266. https://doi.org/10.1007/S10009-025-00787-0

[28] Bruno Courcelle. 1989. The Monadic Second-Order Logic of Graphs, II: Infinite Graphs of Bounded Width. *Math. Syst. Theory* 21, 4 (1989), 187–221. https://doi.org/10.1007/BF02088013

[29] Bruno Courcelle. 1991. On Constructing Obstruction Sets of Words. *Bull. EATCS* 44 (1991), 178–186.

[30] Bruno Courcelle. 2010. Special tree-width and the verification of monadic second-order graph properties. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 8)*, Kamal Lodaya and Meena Mahajan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13–29. https://doi.org/10.4230/LIPIcs.FSTTCS.2010.13

[31] Aiswarya Cyriac. 2014. *Verification of communicating recursive programs via split-width. (Vérification de programmes récursifs et communicants via split-width).* Ph.D. Dissertation. École normale supérieure de Cachan, France. https://tel.archives-ouvertes.fr/tel-01015561

[32] Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. 2012. MSO Decidability of Multi-Pushdown Systems via Split-Width. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7454)*, Maciej Koutny and Irek Ulidowski (Eds.). Springer, 547–561. https://doi.org/10.1007/978-3-642-32940-1_38

[33] Shuo Ding and Qirun Zhang. 2023. Mutual Refinements of Context-Free Language Reachability. In *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14284)*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer, 231–258. https://doi.org/10.1007/978-3-031-44245-2_12

[34] Andrzej Ehrenfeucht and Grzegorz Rozenberg. 1977. On some context free languages that are not deterministic ETOL languages. *RAIRO. Informatique théorique* 11, 4 (1977), 273–291. https://doi.org/10.1051/ITA/1977110402731

[35] Joost Engelfriet and Linda Heyker. 1991. The String Generating Power of Context-Free Hypergraph Grammars. *J. Comput. Syst. Sci.* 43, 2 (1991), 328–360. https://doi.org/10.1016/0022-0000(91)90018-Z

[36] Joost Engelfriet, Grzegorz Rozenberg, and Giora Slutzki. 1980. Tree Transducers, L Systems, and Two-Way Machines. *J. Comput. Syst. Sci.* 20, 2 (1980), 150–202. https://doi.org/10.1016/0022-0000(80)90058-6

[37] Joost Engelfriet and Sven Skyum. 1976. Copying Theorems. *Inf. Process. Lett.* 4, 6 (1976), 157–161. https://doi.org/10.1016/0020-0190(76)90086-7

[38] Moses Ganardi, Rupak Majumdar, Andreas Pavlogiannis, Lia Schütze, and Georg Zetzsche. 2022. Reachability in Bidirected Pushdown VASS. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France (LIPIcs, Vol. 229)*, Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff (Eds.).

Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 124:1–124:20. https://doi.org/10.4230/LIPICS.ICALP.2022.124

[39] Pierre Ganty and Rupak Majumdar. 2012. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 6:1–6:48. https://doi.org/10.1145/2160910.2160915

[40] Amir Kafshdar Goharshady and Ahmed Khaled Zaher. 2023. Efficient Interprocedural Data-Flow Analysis Using Treedepth and Treewidth. In *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13881)*, Cezara Dragoi, Michael Emmi, and Jingbo Wang (Eds.). Springer, 177–202. https://doi.org/10.1007/978-3-031-24950-1_9

[41] Peter Habermehl, Roland Meyer, and Harro Wimmel. 2010. The Downward-Closure of Petri Net Languages. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 6199)*, Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis (Eds.). Springer, 466–477. https://doi.org/10.1007/978-3-642-14162-1_39

[42] Matthew Hague. 2011. Parameterised Pushdown Systems with Non-Atomic Writes. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India (LIPIcs, Vol. 13)*, Supratik Chakraborty and Amit Kumar (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 457–468. https://doi.org/10.4230/LIPICS.FSTTCS.2011.457

[43] Henk Harkema. 2001. A Characterization of Minimalist Languages. In *Logical Aspects of Computational Linguistics, 4th International Conference, LACL 2001, Le Croisic, France, June 27-29, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2099)*, Philippe de Groote, Glyn Morrill, and Christian Retoré (Eds.). Springer, 193–211. https://doi.org/10.1007/3-540-48199-0_12

[44] Graham Higman. 1952. Ordering by Divisibility in Abstract Algebras. *Proceedings of The London Mathematical Society* (1952), 326–336. https://doi.org/10.1112/plms/s3-2.1.326

[45] Wei Huang, Yao Dong, Ana L. Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 106–117. https://doi.org/10.1145/2771783.2771803

[46] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2022. Bounded Verification of Multi-threaded Programs via Lazy Sequentialization. *ACM Trans. Program. Lang. Syst.* 44, 1 (2022), 1:1–1:50. https://doi.org/10.1145/3478536

[47] Ismaël Jecker. 2021. A Ramsey Theorem for Finite Monoids. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference) (LIPIcs, Vol. 187)*, Markus Bläser and Benjamin Monmege (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 44:1–44:13. https://doi.org/10.4230/LIPICS.STACS.2021.44

[48] Aravind Krishna Joshi. 1985. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural discriptions. In *Natural Language Parsind: Psychological, Computational, and Theoretical Perspectives*, D. Dowty, L. Karttunen, and A. Zwicky (Eds.). Cambridge University Press, 206–250. https://doi.org/10.1017/CBO9780511597855.007

[49] Yuichi Kaji, Ryuichi Nakanishi, Hiroyuki Seki, and Tadao Kasami. 1992. The universal recognition problems for parallel multiple context-free grammars and for their subclasses. *IEICE TRANSACTIONS on Information and Systems* 75, 4 (1992), 499–508.

[50] Yuichi Kaji, Ryuichi Nakanishi, Hiroyuki Seki, and Tadao Kasami. 1994. The Computational Complexity of the Universal Recognition Problem for Parallel Multiple Context-Free Grammars. *Comput. Intell.* 10 (1994), 440–452. https://doi.org/10.1111/J.1467-8640.1994.TB00008.X

[51] Yuichi Kaji, Ryuichi Nakanisi, Hiroyuki Seki, and Tadao Kasami. 1992. The universal recognition problems for multiple context-free grammars and for linear context-free rewriting systems. *IEICE Transactions on Information and Systems* 75, 1 (1992), 78–88.

[52] Makoto Kanazawa and Sylvain Salvati. 2010. The Copying Power of Well-Nested Multiple Context-Free Grammars. In *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6031)*, Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide (Eds.). Springer, 344–355. https://doi.org/10.1007/978-3-642-13089-2_29

[53] Adam Husted Kjelstrøm and Andreas Pavlogiannis. 2022. The decidability and complexity of interleaved bidirected Dyck reachability. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–26. https://doi.org/10.1145/3498673

[54] Shankaranarayanan Krishna, Aniket Lal, Andreas Pavlogiannis, and Omkar Tuppe. 2024. On-the-Fly Static Analysis via Dynamic Bidirected Dyck Reachability. *Proc. ACM Program. Lang.* 8, POPL (2024), 1239–1268. https://doi.org/10.1145/3632884

[55] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. 2007. A Robust Class of Context-Sensitive Languages. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 161–170. https://doi.org/10.1109/LICS.2007.9

[56] Salvatore La Torre, Anca Muscholl, and Igor Walukiewicz. 2015. Safety of Parametrized Asynchronous Shared-Memory Systems is Almost Always Decidable. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015 (LIPIcs, Vol. 42)*, Luca Aceto and David de Frutos-Escrig (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 72–84. https://doi.org/10.4230/LIPICS.CONCUR.2015.72

[57] Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. 2014. A Unifying Approach for Multistack Pushdown Automata. In *Proceedings of MFCS 2014*, Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 377–389. https://doi.org/10.1007/978-3-662-44522-8_32

[58] Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. 2020. Reachability of scope-bounded multistack pushdown systems. *Inf. Comput.* 275 (2020), 104588. https://doi.org/10.1016/J.IC.2020.104588

[59] Clemens Lautemann, Thomas Schwentick, and Denis Thérien. 1994. Logics For Context-Free Languages. In *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers (Lecture Notes in Computer Science, Vol. 933)*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer, 205–216. https://doi.org/10.1007/BFB0022257

[60] Yuanbo Li, Kris Satya, and Qirun Zhang. 2022. Efficient algorithms for dynamic bidirected Dyck-reachability. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498724

[61] Yuanbo Li, Qirun Zhang, and Thomas W. Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434340

[62] Yuanbo Li, Qirun Zhang, and Thomas W. Reps. 2023. Single-Source-Single-Target Interleaved-Dyck Reachability via Integer Linear Programming. *Proc. ACM Program. Lang.* 7, POPL (2023), 1003–1026. https://doi.org/10.1145/3571228

[63] P. Madhusudan and Gennaro Parlato. 2011. The tree width of auxiliary storage. *SIGPLAN Not.* 46, 1 (2011), 283–294. https://doi.org/10.1145/1925844.1926419

[64] Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2022. General Decidability Results for Asynchronous Shared-Memory Programs: Higher-Order and Beyond. *Log. Methods Comput. Sci.* 18, 4 (2022). https://doi.org/10.46298/LMCS-18(4:2)2022

[65] Jens Michaelis. 2001. Transforming Linear Context-Free Rewriting Systems into Minimalist Grammars. In *Logical Aspects of Computational Linguistics, 4th International Conference, LACL 2001, Le Croisic, France, June 27-29, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2099)*, Philippe de Groote, Glyn Morrill, and Christian Retoré (Eds.). Springer, 228–244. https://doi.org/10.1007/3-540-48199-0_14

[66] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI 2007, San Diego, CA, USA, June 10-13, 2007*. ACM, 446–455. https://doi.org/10.1145/1250734.1250785

[67] Rainer Parchmann, Jürgen Duske, and Johann Specht. 1980. On deterministic indexed languages. *Inf. Control.* 45, 1 (1980), 48–67. https://doi.org/10.1016/S0019-9958(80)90867-0

[68] Andreas Pavlogiannis. 2022. CFL/Dyck Reachability: An Algorithmic Perspective. *ACM SIGLOG News* 9, 4 (2022), 5–25. https://doi.org/10.1145/3583660.3583664

[69] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-540-31980-1_7

[70] Michael O. Rabin. 1969. Decidability of Second-Order Theories and Automata on Infinite Trees. *Trans. Amer. Math. Soc.* 141 (1969), 1–35. https://doi.org/10.1090/S0002-9947-1969-0246760-1

[71] Ganesan Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming languages and Systems (TOPLAS)* 22, 2 (2000), 416–430. https://doi.org/10.1145/349214.349241

[72] F. P. Ramsey. 1930. On a Problem of Formal Logic. *Proceedings of the London Mathematical Society* s2-30, 1 (1930), 264–286. https://doi.org/10.1112/plms/s2-30.1.264

[73] Thomas W. Reps. 2000. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186. https://doi.org/10.1145/345099.345137

[74] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On Multiple Context-Free Grammars. *Theor. Comput. Sci.* 88, 2 (1991), 191–229. https://doi.org/10.1016/0304-3975(91)90374-B

[75] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL (2019), 48:1–48:29. https://doi.org/10.1145/3290361

[76] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 387–400. https://doi.org/10.1145/1133981.1134027

[77] Mikkel Thorup. 1998. All Structured Programs have Small Tree-Width and Good Register Allocation. *Inf. Comput.* 142, 2 (1998), 159–181. https://doi.org/10.1006/INCO.1997.2697

[78] K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. 1987. Characterizing Structural Descriptions produced by Various Grammatical Formalisms. In *25th Annual Meeting of the Association for Computational Linguistics, Stanford University, Stanford, California, USA, July 6-9, 1987*, Candy L. Sidner (Ed.). ACL, 104–111. https://doi.org/10.3115/981175.981190

[79] David J. Weir. 1992. Linear Context-Free Rewriting Systems and Deterministic Tree-Walking Transducers. In *30th Annual Meeting of the Association for Computational Linguistics, 28 June - 2 July 1992, University of Delaware, Newark, Deleware, USA, Proceedings*, Henry S. Thompson (Ed.). ACL, 136–143. https://doi.org/10.3115/981967.981985

[80] Georg Zetzsche. 2015. An Approach to Computing Downward Closures. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9135)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer, 440–451. https://doi.org/10.1007/978-3-662-47666-6_35

[81] Georg Zetzsche. 2016. The Complexity of Downward Closure Comparisons. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy (LIPIcs, Vol. 55)*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 123:1–123:14. https://doi.org/10.4230/LIPICS.ICALP.2016.123

[82] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 344–358. https://doi.org/10.1145/3009837.3009848

A Special treewidth versus Treewidth

For readers familiar with the bag decomposition based definition of treewidth: Tree decompositions for treewidth require the bags containing a vertex to be a connected subtree. In the case of special treewidth, the bags containing a vertex need to form a connected path. For those familiar with treewidth algebra: The join operation can have vertices with the same color in both operands, and the operation "fuses" the respective vertices in the case of treewidth. In the case of special treewidth, join is allowed only if the active colors of the operands are disjoint, and hence it avoids the need for "fusion".

For bounded degree graphs with degree bound d, the special treewidth is at most $20dt$ where t is the bound on treewidth [30]. The behaviors of multi-pushdown systems, queue systems, message sequence charts etc. have degree at most 3 [31]. The behaviors of timed multi-pushdown systems were already analysed for bounded special treewidth [4].

B MPDA rungraphs and examples

We define the behavior of MPDA by means of *rungraphs*. Rungraphs consists of a sequence of vertices (also called *events*) labelled by transitions, linearly ordered to capture the execution order. In addition there is a binary matching relation that matches push events to the corresponding pop events. Indeed, the labelling by transitions must be locally compatible, and the matching relation must represent the LIFO policy of stacks.

Formally, it is given by the structure $\rho = (V, \mathsf{Succ}, \mathtt{Match}, \lambda)$ where $V = \{v_1, \cdots, v_n\}$ is a finite set of vertices, Succ is the successor relation of a total order on V, $\mathtt{Match} \subseteq V \times V$ is a binary relation and the labelling $\lambda : V \mapsto \Sigma$ labels vertices by letters. The rungraph ρ is accepted by the MPDA $\mathcal{M}$ if there is a map $\mathsf{trans} : V \mapsto \Delta$ that maps vertices to transitions satisfying the following conditions. We first set some notations and shorthands that use in these conditions: The transitive closure of Succ is denoted by $<$. For a transition $\tau = (q_1, a, \mathrm{op}, q_2)$, we will use $\mathtt{tgt}(\tau) = q_2$ and $\mathtt{src}(\tau) = q_1$ to refer to the target and the source state of the transition. Similarly, we will use $\mathsf{letter}(\tau) = a$ and $\mathtt{oper}(\tau) = \mathrm{op}$.

- The map trans must be consistent with the labelling λ: $\mathsf{letter}(\mathsf{trans}(v)) = \lambda(v)$ for all $v \in V$.
- The stack access policy must be respected by $\mathtt{Match}$: If $(u, v) \in \mathtt{Match}$ then
  M1) $u < v$,
  M2) there is an $i \in [1..s]$ and some $a \in \Gamma$ such that
  M2a) $\mathtt{oper}(\mathsf{trans}(u)) = \mathsf{push}(i, a)$ and $\mathtt{oper}(\mathsf{trans}(v)) = \mathsf{pop}(i, a)$
  M2b) there is no $(u', v') \in \mathtt{Match}$ such that for some $b \in \Gamma$, $\mathtt{oper}(\mathsf{trans}(u')) = \mathsf{push}(i, b)$, $\mathtt{oper}(\mathsf{trans}(v')) = \mathsf{pop}(i, b)$, and $u < u' < v < v'$ or $u' < u < v' < v$,
  M3) any event labelled by a push (resp. pop) transition must be the source (resp. target) of a $\mathtt{Match}$ edge.
- The map trans must be consistent between consecutive events: if $(u, v) \in \mathsf{Succ}$ then $\mathtt{tgt}(\mathsf{trans}(u)) = \mathtt{src}(\mathsf{trans}(v))$.
- For the first event u (i.e. there is no v such that $(u, v) \in \mathsf{Succ}$), $\mathtt{src}(\mathsf{trans}(u)) = q_0$, and
- For the last event v, $\mathtt{tgt}(\mathsf{trans}(v)) = f$.

Example B.1. Consider the non context-free language $L_{\mathrm{wstar}} = \{(\#w)^n \mid w \in \{a, b\}^*, n \geq 1\}$. We give a 2-stack MPDA $\mathcal{M}_{\mathrm{wstar}}$ in Figure 11 for L_{wstar}. It uses $\{\#, a, b\}$ as the stack alphabet for both the stacks, and # is used as a stack-bottom indicator. In state q_1 it reads the word w for the first time and stores it in stack 1 in reverse. In state q_2 it copies stack 1 to stack 2 on ϵ-moves to reverse it again. In state q_7 it reads w again from stack 2, and copies it to stack 1 in reverse. From q_1 or q_6 it can choose to stop producing more copies of w and just empty stack 1 in state q_{10} and accept in q_{11}.

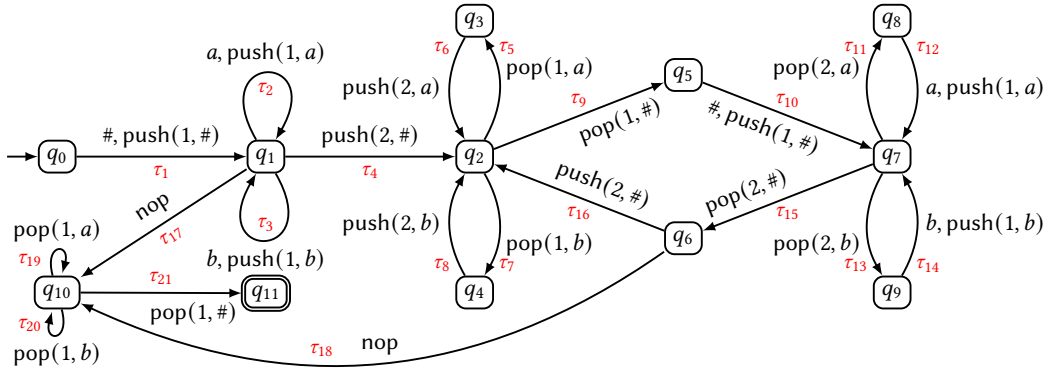


Fig. 11. A 2-stack MPDA $\mathcal{M}_{\text{wstar}}$ accepting the language $L_{\text{wstar}} = \{(\#w)^n \mid w \in \{a, b\}^*, n \geq 1\}$. We omit ϵ from a transition label for readability.

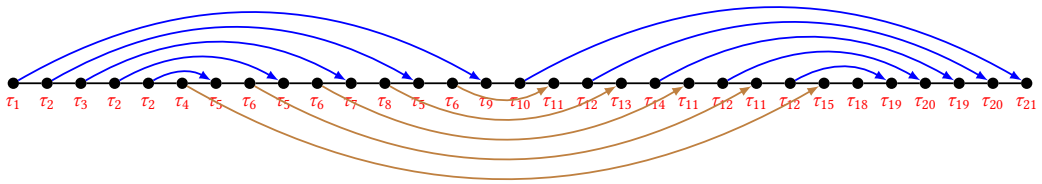


Fig. 12. A rungraph

A rungraph is depicted in Figure 12. The map trans is explicitly given by the node labeling. The Match relation is given by the curved edges, where the upward (blue) curves correspond to stack 1 and the downward (brown) curves correspond to stack 2. This rungraph generates the word #*abaa*#*abaa*. The pattern repeats for every additional #*abaa*.

C Multiple context-free grammars and languages

To make the definition precise, we need some notation.

Notation. A *ranked set* $\hat{A}$ is a set A together with a function arity : $A \to \mathbb{N}$ denoting the rank/arity of each element in A. We may write a ranked set as a pair $\hat{A} = (A, \text{arity})$. If $A = \{a_1, \ldots, a_k\}$ is a finite set, we may also write the ranked set as $\hat{A} = \{a_1^{\text{arity}(a_1)}, \ldots, a_k^{\text{arity}(a_k)}\}$, where each element is tagged with its arity. Given a set B disjoint from A, the set of *atomic terms* of $\hat{A}$ over B, denoted $\hat{A}(B)$, is the set $\{a(b_1, \ldots, b_k) \mid a \in A, \text{arity}(a) = k, b_i \in B, \text{ for all } i \in \{1, 2, \ldots, k\}\}$. A subset S of $\hat{A}(B)$ is called *repetition-free* if

(1) for all terms $a(b_1, \ldots, b_k) \in S$, $b_i \neq b_j$ if $i \neq j$, and
(2) for every pair of distinct terms $a(b_1, \ldots, b_k)$ and $a'(b'_1, \ldots, b'_{k'})$ in S, $b_i \neq b'_j$ for all $i \in \{1, 2, \ldots, k\}$, $j \in \{1, 2, \ldots, k'\}$.

That is, every argument is unique in a repetition-free set. We denote that S is a repetition-free subset of $\hat{A}(B)$ by $S \subseteq_{\text{rf}} \hat{A}(B)$.

Multiple Context-Free Grammars. A *multiple context free grammar* (MCFG) is a tuple of the form $\mathcal{G} = (\Sigma, \hat{N}, X, P, A_{\text{start}})$ where:

- Σ is a finite alphabet.
- $\hat{N} = (N, \text{arity})$ is a finite ranked set of nonterminals with $\text{arity}(A) \geq 1$ for all $A \in N$. N is disjoint from Σ.
- $A^{(1)}_{\text{start}} \in \hat{N}$ is the start nonterminal.
- X is a finite set of variables disjoint from N and Σ.
- P is a finite set of *productions* π of the form $A(s_1, \dots s_k) \leftarrow S$ where
  - $A \in N$ is a nonterminal with $\text{arity}(A) = k$. A is called the *head* of π, denoted $\text{head}(\pi)$.
  - S is a repetition-free subset of $\hat{N}(X)$. That is, $S \subseteq_{\text{rf}} \hat{N}(X)$. S is called the *body* of π, denoted $\text{body}(\pi)$.
  - $s_j \in (\Sigma \cup X)^*$ for $1 \leq j \leq k$.
  - Let $s = s_1 s_2 \dots s_k$, be the concatenation of the arguments of A. Then, for every term $t = B(y_1, y_2, \dots, y_n) \in S$, each argument y_i of t should appear at most once in s. That is, the productions are copyless, but may permute the arguments (and may concatenate additional symbols from Σ) and may also forgot some arguments.

If $\pi = A(s_1, \dots, s_n) \leftarrow S$ is a production rule and $S = \{B_1(x_{1,1}, \dots, x_{1,n_1}), \dots\}$ we define $\text{tgt}_\pi(x_{\ell,i}) = j$ if $x_{\ell,i}$ occurs in s_j. We define $\text{src}_{\pi,\ell}(s_j) = \{i \mid \text{tgt}_\pi(x_{\ell,i}) = j\}$. Where the production is clear from context, we omit it.

A nonterminal A of an MCFG $\mathcal{G}$ with $\text{arity}(A) = k$ generates a k-tuple of words $(u_1, u_2, \dots u_k) \in (\Sigma^*)^k$ if there is a parse tree/derivation tree for it (to be defined below). We write $A \to^h (u_1, u_2, \dots u_k)$ to say that there is *an A-rooted* parse tree t of A of height h with yield $(u_1, u_2, \dots u_k)$ (denoted $\text{yield}(t) = (u_1, u_2, \dots u_k)$). We define this inductively:

- $A \to^0 (u_1, u_2, \dots u_k)$ if $A(u_1, u_2, \dots u_k) \leftarrow \emptyset$ is a production rule. $A(u_1, u_2, \dots u_k)$ is an A-rooted derivation tree with yield $(u_1, u_2, \dots u_k)$.
- $A \to^h (u_1, u_2, \dots u_k)$ if there is production rule $\pi = A(s_1, \dots s_k) \leftarrow S$ and a homomorphism $\eta : X \to \Sigma^*$ such that
  (1) For all terms $B(y_1, \dots, y_n) \in S$, $B \to^\ell (w_1, \dots w_n)$ for some $\ell < h$ where $w_j = \eta(y_j)$.
  (2) $u_i = \bar{\eta}(s_i)$ where $\bar{\eta}$ extends η to $\Sigma \cup X$ in the obvious way ($\bar{\eta}(a) = a$ for all $a \in \Sigma$).
  Let $S = \{B_i(x_{i,1}, \dots, x_{i,m}) \mid i \in \{1, 2, \dots, |S|\}\}$, and suppose t_i is a B_i-rooted tree. Then the tree constructed with root node (A, π) and children t_i (written $(A, \pi)[t_1, \dots, t_{|S|}]$) is an ($A$-rooted) derivation tree.

We address subtrees using paths $p \in \mathbb{N}^*$. If $p = \varepsilon$, then the subtree of a tree t at p, written t/p, is t. If $p = i \cdot p'$, then t/p is t_i/p', where t_i is the i-th child of t.

We write $A \to^* (u_1, u_2, \dots u_k)$ if $A \to^h (u_1, u_2, \dots u_k)$ for some $h \in \mathbb{N}$. A nonterminal A with $\text{arity}(A) = k$ generates a k-ary relation, denoted $\mathcal{R}_\mathcal{G}(A) \subseteq (\Sigma^*)^k$. That is, $\mathcal{R}_\mathcal{G}(A) = \{(u_1, u_2, \dots u_k) \mid A \to^* (u_1, u_2, \dots u_k)\}$. When the MCFG $\mathcal{G}$ is clear from the context, we simply write $\mathcal{R}(A)$ instead of $\mathcal{R}_\mathcal{G}(A)$. Note that the start nonterminal A_{start} generates a unary relation, i.e. a subset of Σ^*, which we also call the language of the MCFG $\mathcal{G}$. That is, $\mathcal{L}(\mathcal{G}) = \mathcal{R}(A_{\text{start}})$. A language L is called a *multiple context-free language* (MCFL) if there is an MCFG $\mathcal{G}$, such that $L = \mathcal{L}(\mathcal{G})$. We say that an MCFG $\mathcal{G} = (\Sigma, \hat{N}, X, P, A_{\text{start}})$ is a k-MCFG if for all $A \in N$, $\text{arity}(A) \leq k$.

D MCFL examples

Example D.1. Consider the 2-MCFG over the nonterminals $\{B^{(1)}, A^{(2)}\}$ with the following productions.

$$B(xby) \leftarrow \{A(x,y)\} \tag{π_1}$$
$$A(a,xy) \leftarrow \{A(x,y)\} \tag{π_2}$$
$$A(xy,a) \leftarrow \{A(x,y)\} \tag{π_3}$$
$$A(a,a) \leftarrow \emptyset \tag{π_4}$$

Here, $\mathcal{R}(A) = (a^+, a) \cup (a, a^+)$, and $\mathcal{R}(B) = a^+ba + aba^+$. We have $\mathcal{R}(B)\downarrow = a^*(b+\epsilon)(a+\epsilon) + (a+\epsilon)(b+\epsilon)a^*$. We have $\mathcal{R}(A)\downarrow = (a^*, a+\epsilon) + (a+\epsilon, a^*)$. □

Example D.2. Consider the 2-MCFG $\mathcal{G}_{ww} = (\{a,b\}, \hat{N}, X, P, A_{\text{start}})$ where $\hat{N} = \{A^{(2)}, A^{(1)}_{\text{start}}\}$, $X = \{x_1, \ldots, x_4\}$ and P listed below

$$A_{\text{start}}(x_1x_2) \leftarrow \{A(x_1,x_2)\} \tag{π_1}$$
$$A(x_1x_3, x_2x_4) \leftarrow \{A(x_1,x_2), A(x_3,x_4)\} \tag{π_2}$$
$$A(a,a) \leftarrow \emptyset \tag{π_3}$$
$$A(b,b) \leftarrow \emptyset \tag{π_4}$$

Here, $\mathcal{R}(A) = \{(w,w) \mid w \in \Sigma^+\}$, and $\mathcal{L}(\mathcal{G}_{ww}) = \mathcal{R}(A_{\text{start}}) = \{ww \mid w \in \Sigma^+\}$. We have $\mathcal{R}(A_{\text{start}})\downarrow = (a+b)^*$, and $\mathcal{R}(A)\downarrow = ((a+b)^*, (a+b)^*)$. □

E Tree automata to MCFG

E.1 Summaries and their computation

Given a cp-lo-graph $\hat{G} = \big(V, \textsc{Succ}, (E_\gamma)_{\gamma\in\Gamma}, \lambda, \chi\big)$, we are interested in the summary wrt Succ and hence it suffices to consider the graph $\hat{G}_{\textsc{Succ}} = (V, \textsc{Succ}, \chi)$.

Piece Summary The graph $\hat{G}_{\textsc{Succ}}$ is a collection of pieces. We abstract a piece just by the pair of colors of the extreme vertices. That is, if $u_1u_2\ldots u_n$ is a piece (i.e., $(u_i, u_{i+1}) \in \textsc{Succ}$), then its abstraction is $(\chi(u_1), \chi(u_n))$. While strict cp-lo-graphs technically require all extreme vertices to be colored, removing the coloring, i.e. via $\Diamond(\cdot)$, could still lead to an lo-graph, even if the first and last vertex in the Succ-order are not colored. In particular, it is possible that $\chi(u_1)$ is undefined, which means that no predecessor can be added to u_1. In this case, we let $\chi(u_1)$ be $\vdash$. Clearly, we only need to consider at most one such vertex. Similarly, if $\chi(u_n)$ is undefined we let it be $\dashv$. By $\mathsf{piece\text{-}summ}(\hat{G})$ we denote the set of pairs of start and end colors of the pieces of $\hat{G}_{\textsc{Succ}} = (V, \textsc{Succ}, \chi)$. Note that, $\mathsf{piece\text{-}summ}(\hat{G}) \subseteq (\mathsf{C} \cup \{\vdash\}) \times (\mathsf{C} \cup \{\dashv\})$. Note that (c,c) is a possible abstraction in $\mathsf{piece\text{-}summ}(\hat{G})$ which means the corresponding piece has only one vertex, and that is colored c. Also, if (c_1,c_2) and (c_3,c_4) are both in $\mathsf{piece\text{-}summ}(\hat{G})$, then $\{c_1,c_2\} \cap \{c_3,c_4\} = \emptyset$.

For any fixed set of colors C, the set of all possible piece summaries is $\mathsf{pieceSmrySet}(\mathsf{C}) = 2^{(\mathsf{C}\cup\{\vdash\})\times(\mathsf{C}\cup\{\dashv\})}$. Next we show that the we can design a tree automaton $\mathcal{A}_{\mathsf{piece\text{-}smry}}$ over expressions from $\mathsf{expr}(C, \Sigma, \Gamma)$, such that for all $\psi \in \mathsf{expr}(C,\Sigma,\Gamma)$, $[\![\mathcal{A}_{\mathsf{piece\text{-}smry}}]\!](\psi) = \mathsf{piece\text{-}summ}(\hat{G}_\psi)$.

E.2 Tree automata computing summaries

The tree automaton $\mathcal{A}_{\mathsf{piece\text{-}smry}} = \big(\mathsf{pieceSmrySet}(\mathsf{C}), \delta^{\mathsf{piece}}, \emptyset, \{(\vdash,\dashv)\}\big)$ computes the piece summaries. We define $\delta^{\mathsf{piece}} = \bigcup_{\mathsf{op}\in\mathsf{UOP}} \delta^{\mathsf{piece}}_{\mathsf{op}} \cup \delta^{\mathsf{piece}}_{\textsc{Join}}$ as follows. Here, $\sigma, \sigma_1, \sigma_2 \in \mathsf{pieceSmrySet}(\mathsf{C})$.

- $\delta^{\mathsf{piece}}_{\textsc{Node}(c,a)}(\sigma) = \sigma \cup \{(c,c)\}$

- $\delta^{\text{piece}}_{\text{Succ}(c_1,c_2)}(\sigma) = \begin{cases} \text{undefined} & \text{if } (c_2, c_1) \in \sigma \\ \sigma' \cup \{(x,y)\} & \text{if } \sigma = \sigma' \cup \{(x,c_1),(c_2,y)\} \text{ for } x,y \in \mathsf{C} \cup \{\vdash, \dashv\} \\ \text{undefined} & \text{otherwise} \end{cases}$
- $\delta^{\text{piece}}_{\text{Edge}(\gamma,c_1,c_2)}(\sigma) = \sigma$
- $\delta^{\text{piece}}_{\text{Free}(c)}(X,\sigma) = \begin{cases} \sigma & \text{if } \neg\exists y((c,y) \in \sigma \vee (y,c) \in \sigma) \\ \sigma' \cup \{(\vdash, y)\} & \text{if } \sigma = \sigma' \cup \{(c,y)\}, y \neq c, \neg\exists x\ (\vdash, x) \in \sigma \\ \sigma' \cup \{(x, \dashv)\} & \text{if } \sigma = \sigma' \cup \{(x,c)\}, x \neq c, \neg\exists y\ (y, \dashv) \in \sigma \\ \sigma' \cup \{(\vdash, \dashv)\} & \text{if } \sigma = \sigma' \cup \{(c,c)\}, \neg\exists y\ ((\vdash, y) \in \sigma \vee (y, \dashv) \in \sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$
- $\delta^{\text{piece}}_{\text{Join}}(\sigma_1, \sigma_2) = \sigma_1 \cup \sigma_2$

Note that the transition $\delta^{\text{piece}}_{\text{Succ}(c_1,c_2)}$ forbids cycles and multiple Succ-successors/predecessors for a vertex. The transition $\delta^{\text{piece}}_{\text{Free}(c)}$ ensures that there is at most one $\vdash$ (or $\dashv$) labeled vertex in the graph. The accepting state guarantees that there is a single piece in the end.

Size bound We argue that number of states of $\mathcal{A}_{\text{piece-smry}}$ is at most the number of permutations of the set $\mathsf{C} \cup \{\vdash, \dashv\}$. We will show an injective map h from $\text{pieceSmrySet}(\mathsf{C})$ to permutations S_n of the set $\mathsf{C} \cup \{\vdash, \dashv, \bot\}$, where $n = |\mathsf{C}| + 3$. Let $h(\sigma) = g \in S_n$, where g is defined as follows. Fix an ordering $\prec$ on $\mathsf{C} \cup \{\vdash, \dashv, \bot\}$. If $(c_1, c_2) \in \sigma$, then $g(c_1) = c_2$ and $g(c_2) = c_1$. For every color that is not present in σ, let g form a big cycle (cyclically closing the $\prec$) with all those colors. Note that $\bot$ does not appear in σ and hence it always identifies the missing colors from σ. In conclusion, the number of states is at most $|S_n| = n! \leq n^n = 2^{O(n \log n)} = 2^{O(|C| \log |C|)}$.

E.3 Tree Automata to MCFG

Given a tree automaton $\mathcal{A} = \left(Q, \delta^{\mathcal{A}}, q_\bot, Q_{\text{accept}}\right)$ over $\mathtt{expr}(C, \Sigma, \Gamma)$, we construct the tree automaton $\mathcal{B}$, the Cartesian product of $\mathcal{A}$ and $\mathcal{A}_{\text{piece-smry}}$. That is, the states of $\mathcal{B}$ are of the form (q, σ), where $q \in Q$ and $\sigma \in \text{pieceSmrySet}(\mathsf{C})$. Since also $\mathcal{L}(\mathcal{A}_{\text{piece-smry}}) \subseteq \mathtt{expr}(C, \Sigma, \Gamma)$, we have

Claim E.1.

$$\begin{aligned} \mathcal{L}(\mathcal{B}) &= \mathcal{L}(\mathcal{A}) \cap \{\psi \mid \hat{G}_\psi \textit{ is a (colored) lo-graph}\}, \textit{ which implies} \\ \mathcal{L}_{\hat{\mathrm{G}}}(\mathcal{B}) &= \mathcal{L}_{\mathrm{G}}(\mathcal{B}) = \mathcal{L}_{\mathrm{G}}(\mathcal{A}), \textit{ which implies} \\ \mathcal{L}_{\mathrm{W}}(\mathcal{B}) &= \mathcal{L}_{\mathrm{W}}(\mathcal{A}). \end{aligned}$$

Given the automaton $\mathcal{B}$ over $\mathtt{expr}(C, \Sigma, \Gamma)$ with transition relation $\delta^{\mathcal{B}}$, we will construct an MCFG $\mathcal{G}_{\mathcal{B}} = (\Sigma, \hat{N}_{\mathcal{B}}, X_{\mathcal{B}}, P_{\mathcal{B}}, A_{\text{start}})$ as follows. Let $\prec$ be a total order on $\mathsf{C} \cup \{\vdash, \dashv\}$. For a summary $\sigma \subseteq (\mathsf{C} \cup \{\vdash, \dashv\})^2$, we define $\text{tup}(\sigma) = (x_{c_1,c_1'}, x_{c_2,c_2'} \ldots x_{c_m,c_m'})$ be the unique tuple such that $\sigma = \{(c_1, c_1'), (c_2, c_2'), \ldots, (c_m, c_m')\}$ and $c_j \prec c_{j+1}$ for all $j : 1 \leq j \leq m-1$. That is, we order the pairs in the set σ according to the first member of each pair wrt. $\prec$, to obtain a tuple of variables indexed by the pairs. The length of the tuple $\text{tup}(\sigma)$ is indeed $|\sigma|$. For a tuple of variables τ, $\tau[x \leftarrow\!\shortmid\, y]$ is the unique tuple obtained by replacing every occurrence of x with y.

- $\hat{N}_{\mathcal{B}} = (Q \times \text{pieceSmrySet}(\mathsf{C}), \text{arity})$ where $\text{arity}\big((q,\sigma)\big) = |\sigma|$. That is, the arity of a nonterminal is indeed the number of pieces.
- $X_{\mathcal{B}} = \{x_{c_1,c_2} \mid c_1, c_2 \in \mathsf{C} \cup \{\vdash, \dashv\}\}$
- $P_{\mathcal{B}} = \{\pi_t \mid t \in \delta^{\mathcal{B}}\}$. For $t \in \delta^{\mathcal{B}}$, π_t is defined as follows:
  - if $t \equiv \delta^{\mathcal{B}}_{\text{Node}(c,a)}((q,\sigma)) \mapsto (q', \sigma')$ then

$$\pi_t \equiv (q', \sigma')\ (\text{tup}(\sigma')[x_{c,c} \leftarrow\!\shortmid\, a]) \leftarrow \{(q,\sigma)\ (\text{tup}(\sigma))\}$$

– if $t \equiv \delta^{\mathcal{B}}_{\textsc{Succ}(c_1,c_2)}((q,\sigma)) \mapsto (q',\sigma')$ then
$$\pi_t \equiv (q',\sigma')\,(\mathsf{tup}(\sigma')[x_{c_3,c_4} \leftarrow\!\shortmid\, x_{c_3,c_1}x_{c_2,c_4}]) \leftarrow \{(q,\sigma)\,(\mathsf{tup}(\sigma))\}$$
where $(c_3,c_1),(c_2,c_4) \in \sigma$.
– if $t \equiv \delta^{\mathcal{B}}_{\textsc{Edge}(\gamma,c_1,c_2)}((q,\sigma)) \mapsto (q',\sigma)$ then
$$\pi_t \equiv (q',\sigma)\,(\mathsf{tup}(\sigma)) \leftarrow \{(q,\sigma)\,(\mathsf{tup}(\sigma))\}$$
– if $t \equiv \delta^{\mathcal{B}}_{\textsc{Free}(c)}((q,\sigma)) \mapsto (q',\sigma)$ then
$$\pi_t \equiv (q',\sigma)\,(\mathsf{tup}(\sigma)) \leftarrow \{(q,\sigma)\,(\mathsf{tup}(\sigma))\}$$
– if $t \equiv \delta^{\mathcal{B}}_{\textsc{Free}(c)}((q,\sigma)) \mapsto (q',\sigma')$ with $\sigma \neq \sigma'$ then
$$\pi_t \equiv (q',\sigma')\,(\mathsf{tup}(\sigma')[x_{c_1,c_2} \leftarrow\!\shortmid\, x_{c_3,c_4}]) \leftarrow \{(q,\sigma)\,(\mathsf{tup}(\sigma))\}$$
where $(c_1,c_2) \in \sigma' \setminus \sigma$, and $(c_3,c_4) \in \sigma \setminus \sigma'$.
– if $t \equiv \delta^{\mathcal{B}}_{\textsc{Join}}((q_1,\sigma_1),(q_2,\sigma_2)) \mapsto (q,\sigma)$ then
$$\pi_t \equiv (q,\sigma)\,(\mathsf{tup}(\sigma)) \leftarrow \{(q_1,\sigma_1)\,(\mathsf{tup}(\sigma_1)),(q_2,\sigma_2)\,(\mathsf{tup}(\sigma_2))\}$$

Finally, we add the production rule π_q for each $q \in Q_{\text{accept}}$.
$$\pi_q \equiv A_{\text{start}}(x_{\vdash,\dashv}) \leftarrow (q,\{(\vdash,\dashv)\})\,(x_{\vdash,\dashv})$$

Claim E.2. $\mathcal{L}_{\mathrm{W}}(\mathcal{B}) = \mathcal{L}(\mathcal{G}_{\mathcal{B}})$

Hence by Claim E.1 we get $\mathcal{L}_{\mathrm{W}}(\mathcal{A}) = \mathcal{L}(\mathcal{G}_{\mathcal{B}})$. Since $\mathcal{B}$ is the Cartesian product of $\mathcal{A}$ and $\mathcal{A}_{\text{piece-smry}}$ the size upper bounds claimed in Theorem 3.2 follow. This completes the proof of Theorem 3.2.

F MCFL to MSO

F.1 Nested word of a parsetree

The lo-graph of a parsetree is defined inductively. Corresponding to a subtree t we will have a piecewise-lo-graph G_t. Recall that a piecewise-lo-graph is defined like a cp-lo-graph, but without the vertex coloring χ. The end-points of the pieces in G_t will be connected by a matching edge $\curvearrowright$. More precisely if the root of t is a nonterminal $A^{(n)}$, then there will nodes $u^b_{t,1}, u^e_{t,1}, u^b_{t,2}, u^e_{t,2}, \ldots u^b_{t,n}, u^e_{t,n}$ such that for each i

(1) $u^b_{t,i} < u^e_{t,i}$ (recall, $<$ is the transitive closure of Succ)
(2) there is no u such that $\textsc{Succ}(u, u^b_{t,i})$ and
(3) there is no u such that $\textsc{Succ}(u^e_{t,i}, u)$.

Further, if the the production applied at the root of t is of the form Prod (see p. 5), then the root of t has m children, each rooting subtrees $t_1, \ldots t_m$ respectively. Then the piecewise-lo-graph G_t will be $(V_t, \textsc{Succ}_t, \curvearrowright_t, \lambda_t)$ where

- $V_t = \bigcup_{i\in\{1,2,\ldots,m\}} V_{t_i} \cup \{v_{t,i,j} \mid i \in \{1,2,\ldots,n\}, j\text{th letter of } s_i \text{ is from } \Sigma\}$
$\cup\{u^b_{t,1}, u^e_{t,1}, u^b_{t,2}, u^e_{t,2}, \ldots u^b_{t,n}, u^e_{t,n}\}$.
- $\textsc{Succ}_t = \bigcup_{i\in\{1,2,\ldots,m\}} \textsc{Succ}_{t_i}$
$\cup\{(u^b_{t,i}, v_{t,i,1}) \mid \text{ if 1st letter of } s_i \text{ is from } \Sigma\} \cup \{(u^b_{t,i}, u^b_{t_j,k}) \mid \text{ if 1st letter of } s_i \text{ is } x_{j,k}\}$
$\cup\{(v_{t,i,|s_i|}, u^e_{t,i}) \mid \text{ if the last letter of } s_i \text{ is from } \Sigma\}\cup\{(u^e_{t_j,k}, u^e_{t,i}) \mid \text{ if the last letter of } s_i \text{ is } x_{j,k}\}$
$\cup\{(v_{t,i,j}, v_{t,i,j+1}) \mid \text{ if } j\text{th letter and } (j+1)\text{th letters of } s_i \text{ are from } \Sigma\}$
$\cup\{(v_{t,i,j}, u^b_{t_j,k}) \mid \text{ if } j\text{th letter of } s_i \text{ is from } \Sigma \text{ and } (j+1)\text{th letter of } s_i \text{ is } x_{j,k}\}$

$\cup\{((u^e_{t_{j,k}}, v_{t,i,j+1}) \mid \text{ if } j\text{th letter of } s_i \text{ is } x_{j,k} \text{ and } (j+1)\text{th letter of } s_i \text{ is from } \Sigma\}$

$\cup\{((u^e_{t_{j,k}}, u^b_{t_{j',k'}}) \mid \text{ if } j\text{th letter of } s_i \text{ is } x_{j,k} \text{ and } (j+1)\text{th letter of } s_i \text{ is } x_{j',k'}\}$

- $\curvearrowright_t = \bigcup_{i\in\{1,2,\ldots,m\}} \curvearrowright_{t_i} \cup\{(u^b_{t,i}, u^e_{t,i}) \mid i \in \{1,2,\ldots,n\}\} \cup \{(u^e_{t,i}, u^b_{t,i+1}) \mid i \in \{1,2,\ldots,n-1\}\}$
- $\lambda_t = \bigcup_{i\in\{1,2,\ldots,m\}} \lambda_{t_i} \cup \{v_{t,i,j} \mapsto a \mid j\text{th letter of } s_i \text{ is a}\}$

Here, vertices unlabelled by λ_t are treated as having label ϵ.

F.2 Well-nesting in MSO

We need some shorthands.

$$\mathsf{mpath}(x_1, \ldots x_n) \equiv x_1 \curvearrowright x_2 \curvearrowright \ldots x_n$$
$$\mathsf{maxmpath}(x_1, \ldots, x_n) \equiv \mathsf{mpath}(x_1, \ldots, x_n) \wedge \forall x(\neg\mathsf{mpath}(x, x_1, \ldots, x_n) \wedge \neg\mathsf{mpath}(x_1, \ldots, x_n, x))$$
$$\mathsf{eps}(x) \equiv \bigwedge_{a\in\Sigma} \neg a(x)$$

The shorthand $\mathsf{mpath}(x_1, \ldots x_n)$ (resp. $\mathsf{maxmpath}(x_1, \ldots x_n)$) states that its arguments form a $\curvearrowright$-connected path (resp. maximal path). Note that $\mathsf{mpath}(x, y)$ just means $x \curvearrowright y$. The shorthand $\mathsf{eps}(x)$ states absence of any label from Σ, which functionally means ϵ. Let us state the formulas:

$$\phi_1 \equiv \neg\exists x_1, x_2 \ldots x_{2d+1}\ \mathsf{mpath}(x_1, \ldots x_{2d+1})$$
$$\phi_2 \equiv \bigwedge_{i\in\{1,2,\ldots,d\}} \neg\exists x_1, x_2 \ldots x_{2i+1}\ \mathsf{maxmpath}(x_1, \ldots x_{2i+1})$$
$$\phi_3 \equiv \forall x_1, x_2, x_3\ (x_1 \curvearrowright x_2 \wedge x_1 \curvearrowright x_3 \implies x_2 = x_3) \wedge (x_1 \curvearrowright x_3 \wedge x_2 \curvearrowright x_3 \implies x_1 = x_2)$$
$$\phi_4 \equiv \forall x\, \mathsf{eps}(x) \iff \exists y\, (x \curvearrowright y \vee y \curvearrowright x)$$
$$\phi_5 \equiv \bigwedge_{k\in\{1,2,\ldots,d\}} \forall x_1, x_2, \ldots x_{2k}\ \mathsf{maxmpath}(x_1, x_2, \ldots x_{2k}) \implies \left(\bigwedge_{i\in\{1,2,\ldots,k\}} x_{2i-1} \le x_{2i} \right.$$
$$\left. \bigwedge_{i,j\in\{1,2,\ldots,k\}, i\neq j} x_{2i} < x_{2j-1} \vee x_{2j} < x_{2i-1} \right)$$
$$\phi_6 \equiv \bigwedge_{k,k'\in\{1,2,\ldots,d\}} \forall x_1, x_2, \ldots x_{2k}\, \forall x'_1, x'_2, \ldots x'_{2k'}$$
$$\left(\mathsf{maxmpath}(x_1, x_2, \ldots x_{2k}) \wedge \mathsf{maxmpath}(x'_1, x'_2, \ldots x'_{2k'}) \right.$$
$$\left. \wedge \bigvee_{i\in\{1,2,\ldots,k\}} \bigvee_{i'\in\{1,2,\ldots,k'\}} (x_{2i-1} < x'_{2i'} < x_{2i}) \vee (x_{2i-1} < x'_{2i'-1} < x_{2i}) \right)$$
$$\implies \bigwedge_{i'\in\{1,2,\ldots,k'\}} \bigvee_{i\in\{1,2,\ldots,k\}} (x_{2i-1} < x'_{2i'-1} < x'_{2i'} < x_{2i})$$

Finally

$$\phi_{\text{well-nesting}} \equiv \bigwedge_{i\in\{1,2,\ldots,6\}} \phi_i$$

F.3 The formula for valid parsetree

Let us give the concrete formula for this part now. The formulas will use second-order free variables Y_A for each $A \in N$. We first define the shorthands.

$$\mathsf{Ampath}(A, x_1, \dots x_n) \equiv \mathsf{mpath}(x_1, \dots x_n) \wedge \bigwedge_{i \in \{1,2,\dots,n\}} Y_A(x_i)$$

$$\mathsf{maxAmpath}(A, x_1, \dots, x_n) \equiv \mathsf{maxmpath}(x_1, \dots x_n) \wedge \mathsf{Ampath}(A, x_1, \dots x_n)$$

For a production rule π of the form Prod (see p. 5) we define the following shorthand. Note that n is the arity of the nonterminal in the LHS of π.

$$\begin{aligned}\mathsf{prod}(\pi, y_1^b, y_1^e, y_2^b, y_2^e, \dots, y_n^b, y_n^e) \equiv \exists x_{1,1}^b, x_{1,1}^e, \dots x_{1,n_1}^b, x_{1,n_1}^e, \dots, x_{m,1}^b, x_{m,1}^e, \dots x_{m,n_m}^b, x_{m,n_m}^e \\ \bigwedge_{i \in \{1,2,\dots,m\}} \mathsf{maxAmpath}(B_i, x_{i,1}^b, x_{i,1}^e, x_{i,2}^b, x_{i,2}^e, \dots, x_{i,n_i}^b, x_{i,n_i}^e) \\ \bigwedge_{i \in \{1,2,\dots,n\}} \mathsf{nextscan}(y_i^b, y_i^e, s_i)\end{aligned}$$

The macro $\mathsf{scan}(x, y, w)$ is supposed to scan the sequence w along the top level path from x to y. The macro $\mathsf{nextscan}(x, y, w)$ will do the same, but starting from the next position of x. We define these macros for each possible s_i in the grammar and their suffixes.

$$\begin{aligned}\mathsf{nextscan}(x, y, w) &\equiv \exists z\, \mathsf{Succ}(x, z) \wedge \mathsf{scan}(z, y, w) \\ \mathsf{scan}(x, y, \epsilon) &\equiv x = y \\ \mathsf{scan}(x, y, aw) &\equiv a(x) \wedge \exists z\, \mathsf{Succ}(x, z) \wedge \mathsf{scan}(z, y, w) && \text{for } a \in \Sigma \\ \mathsf{scan}(x, y, vw) &\equiv x = v^b \wedge \exists x'(x \curvearrowright x' \wedge \exists z(\mathsf{Succ}(x', z) \wedge \mathsf{scan}(z, y, w))) && \text{for } v \in X\end{aligned}$$

Let us define the formulas now.

$$\psi_1 \equiv \forall x\, \mathsf{eps}(x) \implies \bigvee_{A \in N} \left(Y_A(x) \wedge \bigwedge_{B \in N \setminus \{A\}} \neg Y_B(x) \right)$$

$$\psi_2 \equiv \bigwedge_{A^{(k)} \in \hat{N}} \left(\neg \exists x_1, \dots, x_{2k+1} \mathsf{Ampath}(x_1, \dots, x_{2k+1}) \right.$$
$$\left. \wedge \bigwedge_{m \in \{1,2,\dots,2k-1\}} \neg \exists x_1, \dots, x_m \mathsf{maxAmpath}(A, x_1, \dots, x_m) \right)$$

$$\psi_3 \equiv \bigwedge_{A^{(k)} \in \hat{N}} \exists y_1^b, y_1^e, y_2^b, y_2^e, \dots, y_k^b, y_k^e, \mathsf{maxAmpath}(A, y_1^b, y_1^e, y_2^b, y_2^e, \dots, y_k^b, y_k^e) \implies$$
$$\bigvee_{\pi \in P, \text{LHS of } \pi \text{ is } A} \mathsf{prod}(\pi, y_1^b, y_1^e, y_2^b, y_2^e, \dots, y_k^b, y_k^e)$$

$$\psi_4 \equiv \exists x_{\min} \exists x_{\max} \forall x\ x_{\min} \le x \le x_{\max} \wedge \mathsf{maxAmpath}(A_{\text{start}}, x_{\min}, x_{\max})$$

Finally we have

$$\phi_{\mathcal{G}} = \exists (Y_A)_{A \in N} \bigwedge_{i \in \{1,2,\dots,4\}} \psi_i\,.$$

F.4 Correctness Arguments

F.4.1 Correctness of the constructed formula.

Claim F.1. $\mathcal{L}_{\mathrm{W}}(\phi) = \mathcal{L}(\mathcal{G})$

Proof. $\subseteq$. Suppose $w \in \mathcal{L}_{\mathrm{W}}(\phi)$. By definition, there is an lo-graph $G \in \mathcal{L}_{\mathrm{G}}(\phi)$ such that $w = \mathrm{word}(G)$. We want to argue that there is a parse tree t such that

(1) G is the nested word of t, and
(2) $\mathrm{yield}(t) = w$.

Since G satisfies ϕ, we look at the satisfying assignment of Y_A to the $\curvearrowright$ paths, and extract a parsetree as follows. The root will be labelled by the nonterminal of the vertices $v_{\min}$ and $v_{\max}$, which must be A_{start} thanks to ψ_4. Following ψ_3 there is a production rule that can be applied to every node built in the top down fashion this way. Thanks to ψ_3 this procedure gives a valid parse tree t, which satisfies the above two conditions.

$\supseteq$. Suppose $w \in \mathcal{L}(\mathcal{G})$. By definition there is a parsetree t with $w = \mathrm{yield}(t)$. Following the construction in Appendix F.1 we get a nested word G_t corresponding to t. Notice that it satisfies all the conditions of $\phi_{\text{well-nesting}}$ and $\phi_{\mathcal{G}}$, and hence $G_t \in \mathcal{L}_{\mathrm{G}}(\phi)$. Further $\mathrm{word}(G_t) = w$. □

F.4.2 Special treewidth of the nested words.

Claim F.2. *Nested words of a d-MCFG(r) $\mathcal{G}$ have special treewidth at most $6dr$.*

The idea is that the well-nesting structure of the nested word would give the hint for a special tree decomposition, and more or less has the same shape as the parsetree (though a node in the parsetree is replaced by a path of bags). We define the tree-decomposition inductively.

Let $p = (v_1, \ldots, v_{2n})$ be a maximal $\curvearrowright$ path. The partially-complete nested word covered by such a path is the induced subgraph on $V_p = \{v_1, \ldots, v_{2n}\} \cup \{u \mid v_{2i-1} < u < v_{2i}, i \in \{1, 2, \ldots, n\}\}$, call it G_p. Claim F.2 would follow from the following claim:

Claim F.3. *There is a special tree decomposition for G_p of width at most $6dr$, and having only $\{v_1, \ldots, v_{2n}\}$ in the root of this special tree decomposition.*

We prove Claim F.3 by induction.

- The base case is for a maximal $\curvearrowright$ path $p = (v_1, \ldots, v_{2n})$, $n \leq d$, which does not cover another maximal $\curvearrowright$ path. In this case, we have a special tree decomposition of width $2n + 1$ for G_p. Indeed, G_p can be generated in a left to right manner, keeping all boundary vertices in the bag going forward, and discarding the inner vertices whose neighbours were all added. This special tree decomposition is indeed a special path decomposition, since the tree is just a path. For instance, consider Figure 4 and consider a maximal $\curvearrowright$ path $p_1 = (6, 8, 11, 14)$. Then G_{p_1} is the induced subgraph of Figure 4 on the vertices $\{6, 7, 8, 11, 12, 13, 14\}$. This is can be generated by the special path decomposition $\{6, 7, 8\}$, $\{6, 8, 11\}$, $\{6, 8, 11, 12\}$, $\{6, 8, 11, 12, 13\}$, $\{6, 8, 11, 13, 14\}$, $\{6, 8, 11, 14\}$.
- Consider a maximal $\curvearrowright$ path $p = (v_1, \ldots, v_{2n})$, $n \leq d$, which contains other $\curvearrowright$ paths within. A maximal $\curvearrowright$ path p_1 is said to be a child of a maximal $\curvearrowright$ path p_2 if 1) p_1 is contained in G_{p_2} and 2) there is no other maximal $\curvearrowright$ path p_3 such that p_1 is contained in p_3 which is in turn contained in p_2. For the inductive case, suppose a maximal $\curvearrowright$ path $p = (v_1, \ldots, v_{2n})$ has children $p_1, \ldots, p_m$. We assume that we have the special tree decompositions t_i for each of these G_{p_i}. We add a common parent node for all the t_is and have in this all the vertices in the path p and those in the paths p_i. This node has width at most $2d \times (r + 1)$ where r is the number of children and d is the maximum arity. Now we need to add the other nodes

that are not covered by any G_{p_i}. We follow again a left-to-right pass similar to the base case, and this would increase the bag-size by 2. Finally G_p is fully covered, we add one more parent which contains only the vertices of p as required.

Note that the maximum bag size is at most $2d(r+1)+2$ where $2d$ is the an upper bound on the length of a maximal $\curvearrowright$ path, and r is an upper bound on the number of children a path can have. Notice that nested words of d-MCFG(r) satisfy these bounds. Hence their special tree width is at most $2d(r+1)+2$ which is clearly below $6dr$ if $d, r, \geq 1$.

G Downward closures of MCFL

G.1 Concise Overapproximations

In the main body of the text, we have described how to construct an accelerated grammar $\mathcal{G}_\Sigma$ from a base grammar $\mathcal{G}$. Let us briefly show that the size of this construction is small and its construction does not take too much time:

Lemma G.1. *Given an MCFG $\mathcal{G}$ we can construct the accelerated MCFG $\mathcal{G}_\Sigma$ in time $|\mathcal{G}|^{\mathrm{poly}(k)}$, where k is dimension of $\mathcal{G}$.*

Proof. Let n be the size of $\mathcal{G}$. We already argued above that adding all our decorations leads to a grammar of size $O(n^{4^k})$, and this construction does not take longer to perform than its size suggests. To obtain $\mathcal{G}_\Sigma$ we now add $O(nk^{2k})$ more productions per Production of $\mathcal{G}$, for each of which we need to search for watershed alphabets. As mentioned before, the latter process is a simple search in the connectivity graph of $\mathcal{G}_\Sigma$, which takes polynomial time in n.

In total, we stay polynomial in n and doubly exponential in k to construct $\mathcal{G}_\Sigma$. □

Let us now turn towards shortening individual trees:

Assume we have a tree t of an accelerated MCFG $\mathcal{G}_\Sigma$ and a path p in t such that along p, we have four A-rooted subtrees t_0, t_1, t_2, and t_3 that all embed into each other with the same idempotent embedding f (that is, $t_i \hookrightarrow_f t_{i-1}$). In particular, we can assume that $t_0 = t$. We construct a tree $\tilde{t}$ such that $\tilde{t}$ accepts more words than t but is smaller in size.

As the root production for $\tilde{t}$, we choose the accelerated rule $\pi_{f,i,g} = A(s'_1, \dots s'_n) \leftarrow S$ obtained from the root production π of t, where i is the first letter of p and g is the embedding of $t_1 \hookrightarrow_g t/i$.

Recall that $\pi_{f,i,g}$ has $m+1$ children to the m children π. For $j \neq i$ and $j \neq m+1$, we obtain $\tilde{t}/j$ from t/j as by projection to J, the set of its entries that do not embed into one of the fixpoints.

The i-th child is obtained from t/i by using the search decoration to t_1 using the embedding g, that is $\tilde{t}/i = (t/i)^{q,g}$ where q is chosen such that $t/(i \cdot q) = t_1$. Note that in particular this means that the subtree in $\tilde{t}$ corresponding to t_1 in t is empty, which we will leverage to show that $\tilde{t}$ is smaller than t.

Finally, we need to construct the $(m+1)$-st child, which will produce the fixpoint entries for the new root production. Here we chose $(t_2)^I$, where I is the set of fixpoints of f.

Lemma G.2. $L(t) \subseteq L(\tilde{t})$.

Proof. By the construction of $\pi_{f,i,g}$, for all $i \notin \mathrm{Img}(f)$ we have $\mathrm{yield}(\tilde{t})[i] = \mathrm{yield}(t)[i]$.

Let therefore $i \in \mathrm{Img}(f)$. We can write $w \in L(\mathrm{yield}(t)[i])$ as $w = uw_iv$, where $w_i \in L(\mathrm{yield}(t_2)[i])$. All entries of t_3 are fully embedded in one of the fixpoint entries in t_2. Therefore the set of leaf nodes that produce u and v can not be rooted below t_3. This means that $\Sigma(u) \subseteq \Sigma_{i,\mathrm{left}}$ and $\Sigma(v) \subseteq \Sigma_{i,\mathrm{right}}$. We therefore have $L(\mathrm{yield}(t))[i]) \subseteq L(\mathrm{yield}(\tilde{t})[i])$. □

Lemma G.3. $|\tilde{t}| < |t|$.

Proof. First, observe that $|\tilde{t}/j| \leq |t_0/j|$ for $j \neq i$. Additionally $|\tilde{t}/(m+1)| \leq |t_2|$. Finally, because $t_1 \hookrightarrow_g t/i$, the annotation of $\tilde{t}/i$ will arrive at t_1 with a target embedding of id. Having zero entries, $t_1^{\varepsilon,\mathrm{id}}$ is empty. Therefore $|\tilde{t}/i| \leq |t_0/i| - |t_1|$. Observing that $|t_2| < |t_1|$ we get $|\tilde{t}| \leq (|t| - |t_1|) + |t_2| < |t_0|$. □

G.2 Upper Bound for Downward Closures without Fixed Dimension

If we now assume that the dimension k of the grammar is part of the input, then a naïve analysis yields the following:

(1) There are k^k possible embedding maps.
(2) A single step of the decoration procedure intoduces $O(n^2 \times k^k)$ new nonterminals and $O(n^2 \times k^k)$ new productions per nonterminal, increasing the grammar size to $O(n^4 \times k^{2k})$.
(3) This step may need to be iterated k times until it stabilizes, yielding a grammar size of $(nk)^{2^{O(k)}} = n^{2^{O(k)}}$.
(4) Adding accelerating productions adds a further $\mathrm{poly}(|\mathcal{G}'|) \times k^{O(k)}$ productions, so the grammar size remains $n^{2^{O(k)}}$.
(5) By [47], the path length to guarantee a monochromatic 4-clique is bounded above by $(k^{4k})^{O(k)} = 2^{poly(k)} \times n^{2^{O(k)}}$, which is still $n^{2^{O(k)}}$.
(6) Trees of this height have size N^N where $N = n^{2^{O(k)}}$ and there are N^{N^N} many of them, yielding a downward closure size of $2^{2^{n^{2^{O(k)}}}}$

Therefore a trivial upper bound for the space complexity of downward closures of MCFGs without fixed dimension is 5EXPSPACE.

H Lower bound for MCFL

In this section we prove Theorem 3.5. Consider the language

$$L_{ww,n} = \left\{ ww \;\middle|\; |w| = 2^n, w \in \{a,b\}^* \right\}.$$

We will give a 2-MCFG that is linear in n for it, and show the downward closure of the above language needs a double exponential sized NFA.

Example H.1. Consider the 2-MCFG $\mathcal{G}_{ww,n} = (\{a,b\}, \hat{N}, X, P, A_{\text{start}})$ where $\hat{N} = \{A_i^{(2)} \mid 0 \leq i \leq n\} \cup \{A_{\text{start}}^{(1)}\}$, $X = \{u_i, v_i, x_i, y_i \mid 1 \leq i < n\} \cup \{u_n, v_n\}$ and P contains the following rules, where $i \in \{0, 1, \ldots n-1\}$

$$A_{\text{start}}(u_n v_n) \leftarrow \{A_n(u_n, v_n)\} \tag{π_n}$$
$$A_{i+1}(u_i x_i, v_i y_i) \leftarrow \{A_i(u_i, v_i), A_i(x_i, y_i)\} \tag{π_i}$$
$$A_0(a,a) \leftarrow \emptyset \tag{π_a}$$
$$A_0(b,b) \leftarrow \emptyset \tag{π_b}$$

Here, $\mathcal{R}(A_i) = \left\{(w,w) \;\middle|\; |w| = 2^i, w \in \{a,b\}^*\right\}$, and $\mathcal{L}(\mathcal{G}_{ww,n}) = \mathcal{R}(A_{\text{start}}) = \left\{ww \;\middle|\; |w| = 2^n, w \in \{a,b\}^*\right\}$. □

A fooling-set argument proves the following claim which in turn proves Theorem 3.5.

Lemma H.2. *Any NFA for $L_{ww,n}{\downarrow}$ has at least 2^{2^n} states.*

Proof. Let $W = \left\{w \;\middle|\; |w| = 2^n, w \in \{a,b\}^*\right\}$ and let $u \in W$. Since $uu \in L_{ww,n}{\downarrow}$ but $vu \notin L_{ww,n}{\downarrow}$ for any $v \in W \setminus \{u\}$, there is a state q_u reached upon reading u which (i) leads to accepting state on reading u and (ii) is not reached by any other $v \in W$. Hence the map $W \to Q$, $u \mapsto q_u$, is injective, implying $|Q| \geq |W| = 2^{2^n}$, proving the claim. □

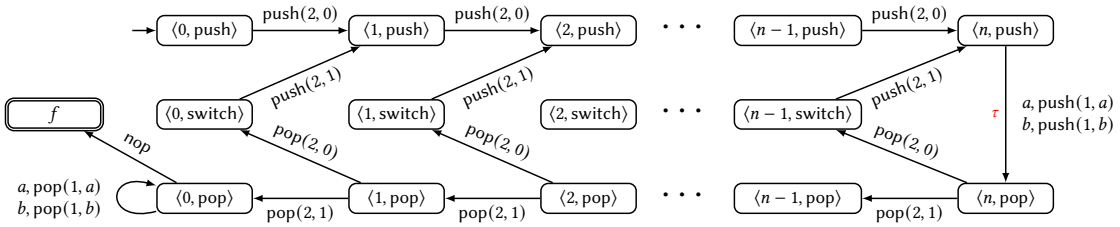


Fig. 13. A 2-stack MPDA with $3n + 3$ states that accepts $\{ww^{\text{rev}} \mid w \in (a + b)^{2^n}\}$. We omit ϵ from a transition label for readability. Here, all the transitions with operations on stack 2 are on ϵ.

I Lower bound for bounded-stw MPDA

In this section, we prove Theorem 3.8. For a word $w \in \Sigma^*$, let w^{rev} denote the reverse of w. Consider the language

$$L_{ww^{\text{rev}},n} = \left\{ww^{\text{rev}} \;\middle|\; |w| = 2^n, w \in \{a, b\}^*\right\}.$$

We will give a 2-stack MPDA of size linear in n for this language. By the same argument as in Appendix H, any NFA for the downward closure of $L_{ww^{\text{rev}},n}$ needs at least 2^{2^n} states.

Consider the MPDA given in Figure 13. The first time it reaches the state $\langle n, \text{push}\rangle$ the stack 2 contains 0^n. In each subsequent visit to the state $\langle n, \text{push}\rangle$, the MPDA will increment the n-bit number in stack 2. For each of these n-bit numbers it will take the transition τ reading a letter and pushing it to stack 1. Once the n-bit number becomes 1^n, it takes the transition τ for the last time and then goes through n transitions, each labelled $\text{pop}(2, 1)$, and reaches state $\langle 0, \text{pop}\rangle$. Now it would have read a word w of length 2^n which is now stored in Stack 1 in reverse. Furthermore, stack 2 is now empty. Finally it reads w^{rev} and empties stack 1, then moves to accepting state f.

It remains to argue that all runs of this MPDA have bounded special treewidth. Since in the rungraph of an MPDA, every vertex has degree at most 3, and graphs of degree d and treewidth t have special treewidth at most $20dt$ [30], it suffices to show that the treewidth of these MPDA is at most 4. Observe that every run of the MPDA can be decomposed into two *phases*: A phase consists of a sequence of transitions where only one stack can be popped, but all stacks can be pushed to. Indeed, during the counting from 0^n to 1^n, the MPDA never pops stack 1. Then, after emptying stack 2, it switches into a second phase, where it pops the content of stack 1 (so as to compare the second part of the input word to w^{rev}). It was shown by Madhusudan and Parlato [63] that MPDA with at most k phases have in fact rungraphs with treewidth at most 2^k. Thus, since our MPDA has at most 2 phases in each run, it has treewidth at most 4.

For Theorem 3.8, it remains to show the following. This follows by the same argument as in Lemma H.2.

Lemma I.1. *Any NFA for $L_{ww^{\text{rev}},n}{\downarrow}$ has at least 2^{2^n} states.*

J Concurrent Programs and Bounded Context Switching

J.1 Formal Definitions

More on Multisets. Let us first recall some additional definitions related to multisets. We will need this for the formal definition of concurrent programs.

First we need a way to denote a multiset with just a few elements. As an example, we write $\mathbf{m} = [\![a, a, c]\!]$ for the multiset $\mathbf{m} \in \mathbb{M}[\{a, b, c, d\}]$ such that $\mathbf{m}(a) = 2$, $\mathbf{m}(b) = \mathbf{m}(d) = 0$, and $\mathbf{m}(c) = 1$. Next we would like some multiset operations akin to addition and subtraction. Given two multisets $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[X]$ let $\mathbf{m} \oplus \mathbf{m}' \in \mathbb{M}[X]$ denote the multiset with $(\mathbf{m} \oplus \mathbf{m}')(a) = \mathbf{m}(a) + \mathbf{m}'(a)$

for all $a \in X$. If $\mathbf{m}(a) \geq \mathbf{m}'(a)$ for all $a \in X$, we also define $\mathbf{m}' \ominus \mathbf{m} \in \mathbb{M}[X]$: for all $a \in X$, we have $(\mathbf{m} \ominus \mathbf{m}')(a) = \mathbf{m}(a) - \mathbf{m}'(a)$. For $X \subseteq Y$ we identify $\mathbf{m} \in \mathbb{M}[X]$ with a multiset in $\mathbb{M}[Y]$ where undefined values are mapped to 0.

Concurrent Programs. We recall the definition of concurrent programs over an arbitrary class of languages C, as given in [13]. To be more precise, C is required to be a language class exhibiting the properties of a so-called "full trio", but it is known that the MCFLs fall under this category [74]. As such, in the sequel we will only consider the MCFLs for the class C.

Let C be a full trio. A *concurrent program* (CP) over C is a tuple $\mathfrak{P} = (D, \Sigma, (L_a)_{a\in\Sigma}, d_0, \mathbf{m}_0)$, where D is a finite set of *global states*, Σ is an alphabet of *process names*, the *process languages* $(L_a)_{a\in\Sigma}$ are a family of languages from C, each over the alphabet $\Sigma_D = D \cup \Sigma \cup (D \times D)$, $d_0 \in D$ is an *initial state*, and $\mathbf{m}_0 \in \mathbb{M}[\Sigma]$ is a multiset of *initial process instances*. We assume that each L_a, $a \in \Sigma$, satisfies the condition $L_a \subseteq aD\big(\Sigma \cup (D \times D)\big)^*(D \cup D \times D)$. This is slightly different from the definition in [13], but as we see below, this is not a meaningful change, and it helps for our proofs in Section 7.

A *configuration* $c = (d, \mathbf{m}) \in D \times \mathbb{M}[\Sigma_D^*]$ consists of a global state $d \in D$ and a multiset $\mathbf{m}$ of words, each representing a process instance. Given a configuration $c = (d, \mathbf{m})$, we write $c.d$ and $c.\mathbf{m}$ to denote the elements d and $\mathbf{m}$, respectively. The size of a configuration c is $|c.\mathbf{m}|$, i.e. the number of current process instances. We distinguish between processes that have been spawned but not executed (*pending processes*) and processes that have already been partially executed (and may be resumed later). The pending process instances are represented by single letters $a \in \Sigma$ (which corresponds to the name of the process), while the partially executed processes corresponding to a name $a \in \Sigma$ are represented by words in the prefix language $\mathrm{pref}(L_a)$ which end in a letter from $D \times D$.

Formally, the semantics of a CP $\mathfrak{P}$ are defined as a labelled transition system over the set of configurations with the transition relation $\Rightarrow \subseteq (D \times \mathbb{M}[\Sigma_D^*]) \times (D \times \mathbb{M}[\Sigma_D^*])$. The initial configuration is $c_0 = (d_0, \mathbf{m}_0)$.

The transition relation is defined using four different types of rules, as shown below. All four types are of the general form $d \xrightarrow{[\![w]\!],\mathbf{n}'} d'$. A rule of this form allows the program to transition from a configuration $(d, \mathbf{m})$ to a configuration $(d', \mathbf{m}')$, written as $(d, \mathbf{m}) \Rightarrow (d', \mathbf{m}')$, iff $d \xrightarrow{[\![w]\!].\mathbf{n}'} d'$ matches a rule and $(\mathbf{m} \ominus [\![w]\!]) \oplus \mathbf{n}' = \mathbf{m}'$. Note that by the definition of $\ominus$, $\mathbf{m}$ has to contain w for such a rule to be applicable. We also write $\overset{w}{\Longrightarrow}$ to specify the particular w used in the transition. As usual, $\Rightarrow^*$ denotes the reflexive transitive closure of $\Rightarrow$. A configuration c is said to be *reachable* if $c_0 \Rightarrow^* c$.

(R1) $d \xrightarrow{[\![a]\!],\Psi(w)\oplus[\![adw(d',d'')]\!]} d'$ if $\exists w \in \Sigma^* \colon adw(d', d'') \in \mathrm{pref}(L_a)$.

Rule (R1) allows us to pick some pending process a from $\mathbf{m}$ and atomically execute it until its next context switch (d', d''). Note that the final letter (d', d'') indicates that this process transitions the global state to d' and can be resumed later when the global state is d'' (unless the prefix $adw(d', d'')$ cannot be extended further).

(R2) $d \xrightarrow{[\![a]\!],\Psi(w)} d'$ if $\exists w \in \Sigma^* \colon adwd' \in L_a$.

Rule (R2) allows us to pick a pending process a from $\mathbf{m}$ and atomically execute it to completion.

(R3) $d \xrightarrow{[\![aw'(d'',d)]\!],\Psi(w)\oplus[\![aw'(d'',d)w(d',d''')]\!]} d'$ if $\exists w \in \Sigma^* \colon aw'(d'', d)w(d', d''') \in \mathrm{pref}(L_a)$

Rule (R3) allows us to pick a partially executed process and execute it atomically until its next context switch.

(R4) $d \xrightarrow{[\![aw'(d'',d)]\!],\Psi(w)} d'$ if $\exists w \in \Sigma^* : aw'(d'',d)wd' \in L_a$

Rule (R4) allows us to pick some partially executed process and execute it to completion.

Note that our change from [13], allowing words from L_a to end in $D \times D$ in addition to D, does not meaningfully change the semantics here. The only thing that might happen is a process may stick around in the multiset, even though its execution cannot be continued. This is because the only ways to remove processes are rules (R2) and (R4), which require words ending in D. However, a process whose prefix cannot be extended does not contribute to further transitions, so in particular it does not change global state reachability in any way.

Bounded context switching. Most decision problems are undecidable for concurrent programs, if we impose no restrictions on their behavior [69, 71]. As such we consider the popular restriction of bounded context switching. To properly introduce this restriction, we need to define a few notions related to runs of concurrent programs.

A *prerun* of a concurrent program $\mathfrak{P} = (D, \Sigma, (L_a)_{a\in\Sigma}, d_0, \mathbf{m}_0)$ is a finite sequence $\rho = (e_0, \mathbf{n}_0)$, w_1, $(e_1, \mathbf{n}_1)$, w_2, …, $(e_\ell, \mathbf{n}_\ell)$ of alternating elements of configurations $(e_i, \mathbf{n}_i') \in D \times \mathbb{M}[\Sigma_D^*]$ and words $w_i \in \Sigma^*$.

We use $\mathsf{Preruns}(\mathfrak{P})$ to denote The set of preruns of $\mathfrak{P}$. Note that if two concurrent programs $\mathfrak{P}$ and $\mathfrak{P}'$ have the same global states D and process names Σ, then $\mathsf{Preruns}(\mathfrak{P}) = \mathsf{Preruns}(\mathfrak{P}')$. The length $|\rho|$ of a prerun ρ is the number of configurations in ρ.

A *run* of a CP $\mathfrak{P} = (D, \Sigma, (L_a)_{a\in\Sigma}, d_0, \mathbf{m}_0)$ is a prerun $\rho = (d_0, \mathbf{m}_0), w_1, (d_1, \mathbf{m}_1), w_2, \ldots, (d_\ell, \mathbf{m}_\ell)$ starting with the initial configuration $(d_0, \mathbf{m}_0)$, such that for each $i \geq 0$, we have $(d_i, \mathbf{m}_i) \xRightarrow{w_{i+1}} (d_{i+1}, \mathbf{m}_{i+1})$. We use $\mathsf{Runs}(\mathfrak{P})$ to denote The set of runs of $\mathfrak{P}$.

For a number k, the run ρ is said to be k-context-bounded (k-CB for short) if for each transition $c \xRightarrow{w} c'$ in ρ we have $|w|_{D\times D} \leq k$. This definition of is slightly changed from [13] as not to count symbols from $D \times D$ at the end of a process' execution, if the process is not continued. The set of k-context-bounded runs of $\mathfrak{P}$ is denoted by $\mathsf{Runs}_k(\mathfrak{P})$. We say a configuration c is *k-reachable* if there is a finite k-context-bounded run ρ ending in c.

J.2 Closure Properties of MCFGs

For an MCFG, let its *dimension* d be the maximum arity among its nonterminals, and its *rank* r be the maximum number of nonterminals appearing on the right-hand side of any of its productions.

Intersecting MCFGs and NFAs. We prove the following:

Theorem J.1. *Given an MCFG $\mathcal{G}$ of dimension d and rank r, as well as an NFA $\mathcal{A}$, one can compute a MCFG $\mathcal{G}_\cap$ for the language $\mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{A})$ in time $(|\mathcal{G}| \cdot |\mathcal{A}|)^{\mathrm{poly}(d\cdot r)}$.*

If d and r are fixed, we therefore obtain:

Corollary J.2. *Given an MCFG $\mathcal{G}$ of constant dimension and rank, as well as an NFA $\mathcal{A}$, one can compute a MCFG for the language $\mathcal{L}(\mathcal{G}) \cap \mathcal{L}(\mathcal{A})$ in polynomial time.*

This is part (a) of Lemma 7.3.

The proof of Theorem J.1 is an adaptation of the well-known *triple construction* for intersecting CFGs and NFAs. Therein for every nonterminal A of the grammar and every pair of states p, q of the automaton, one introduces a new nonterminal $A_{p,q}$ that derives only those words that are derivable by A, but also induce a transition sequence from p to q.

To adapt this to the setting of MCFGs, we need to consider a state pair for every variable of a nonterminal A, i.e. arity of A many state pairs. Let $\mathcal{G} = (\Sigma, \hat{N}, X, P, A_{\mathrm{start}})$ be an MCFG, and let $\mathcal{A}$ be an NFA over Σ^* with set of states Q, initial state q_0, and final states F.

We construct the MCFG $\mathcal{G}_\cap = (\Sigma, \hat{N}_\cap, X, P_\cap, A_{\text{start}\cap})$ as follows:

- $A^{(\ell)}_{\vec{\mathbf{q}}} \in \hat{N}_\cap$ iff $A^{(\ell)} \in \hat{N}$, and $\vec{\mathbf{q}} = \big((q_1, q'_1), (q_2, q'_2), \dots (q_\ell, q'_\ell)\big) \in (Q \times Q)^\ell$.
- $\pi_\cap = A_{\vec{\mathbf{q}}}(s_1, \dots, s_\ell) \leftarrow S_\cap \in P_\cap$ with $S_\cap = \{B_{1,\vec{\mathbf{p}}_1}(x_{1,1}, \dots, x_{1,\ell_1}), \dots\}$ iff the following all hold:
  - $\pi = A(s_1, \dots, s_\ell) \leftarrow S \in P$ with $S = \{B_1(x_{1,1}, \dots, x_{1,\ell_1}), \dots\}$.
  - $\vec{\mathbf{q}} = \big((q_1, q'_1), (q_2, q'_2), \dots (q_\ell, q'_\ell)\big)$, $\vec{\mathbf{p}}_i = \big((p_{i,1}, p'_{i,1}), (p_{i,2}, p'_{i,2}), \dots (p_{i,\ell_i}, p'_{i,\ell_i})\big)$ for all i, and we say $(p_{i,j}, p'_{i,j})$ is the *state pair* for $x_{i,j}$.
    Further, for all i', $s_{i'} = w_{0,i'} y_{1,i'} w_{1,i'} \cdots y_{n_{i'},i'} w_{n_{i'},i'}$, where each $w_{j',i'}$ is a (possibly empty) word from Σ^*, and each $y_{j',i'}$ is a variable from X.
    Then $q_{i'} \xrightarrow{w_{0,i'}} \tilde{q}_{1,i'}$, $\tilde{q}'_{j'-1,i'} \xrightarrow{w_{j'-1,i'}} \tilde{q}_{j',i'}$, and $\tilde{q}'_{n_{i'},i'} \xrightarrow{w_{n_{i'},i'}} q'_{i'}$ in $\mathcal{A}$ for each i', j', where $(\tilde{q}_{j',i'}, \tilde{q}'_{j',i'})$ is the state pair for $y_{j',i'}$.
- $A^{(1)}_{\text{start}\cap}(x) \leftarrow \{A_{\text{start},(q,q')}(x)\} \in P_\cap$ iff $q = q_0$ and $q' \in F$.

By the notation $p \xrightarrow{w} q$ for a (possibly empty) word $w \in \Sigma^*$ we mean that there is a transition sequence from p to q over w in $\mathcal{A}$. If w is empty, then we just consider ε-transitions.

Intuitively, an index $\vec{\mathbf{q}}$ for a nonterminal $A_{\vec{\mathbf{q}}}$ gives a state pair for each member of a tuple derived by $A_{\vec{\mathbf{q}}}$, that needs to be realized as a transition sequence in $\mathcal{A}$ by that member. The productions of $\mathcal{G}_\cap$ are constructed in such a way that state pairs on the right-hand side concatenate properly to ones on the left-hand side.

Let us now prove the theorem.

Proof of Theorem J.1. We construct $\mathcal{G}_\cap$ as above. Towards the theorem, we prove the following statement by induction: A tuple $(v_1, \dots, v_\ell) \in (\Sigma^*)^\ell$ can be derived from $A^{(\ell)}_{\vec{\mathbf{q}}}$ in $\mathcal{G}_\cap$ iff $(v_1, \dots, v_\ell)$ can be derived from $A^{(\ell)}$ in $\mathcal{G}$, and for $\vec{\mathbf{q}} = \big((q_1, q'_1), (q_2, q'_2), \dots (q_\ell, q'_\ell)\big)$ we have $q_i \xrightarrow{v_i} q'_i$ in $\mathcal{A}$ for all i. Then by definition of the productions of $A_{\text{start}\cap}$, a terminal word is derivable by $\mathcal{G}_\cap$ iff it is derivable by $\mathcal{G}$, and induces a run on $\mathcal{A}$ from the initial state to some final state, thus proving the construction correct.

- Base case, $A_{\vec{\mathbf{q}}}(s_1, \dots, s_\ell) \leftarrow \emptyset \in P_\cap$. Then each s_i is a word over atoms a and Γ^*, meaning we require $q_i \xrightarrow{s_i} q'_i$ by definition. Furthermore, $A(s_1, \dots, s_\ell) \leftarrow \emptyset \in P$ is also guaranteed by definition.
  For the other direction, we assume $A(s_1, \dots, s_\ell) \leftarrow \emptyset \in P$ and $q_i \xrightarrow{s_i} q'_i$ for each i. Then, again, $A_{\vec{\mathbf{q}}}(s_1, \dots, s_\ell) \leftarrow \emptyset \in P_\cap$ follows by definition.
- Inductive case, $\pi_\cap = A_{\vec{\mathbf{q}}}(s_1, \dots, s_\ell) \leftarrow S_\cap \in P_\cap$ with $S_\cap = \{B_{1,\vec{\mathbf{p}}_1}(x_{1,1}, \dots, x_{1,\ell_1}), \dots\}$, where for the words derivable from each $B_{i,\vec{\mathbf{p}}_i}$ we assume the statement already holds, as long as the derivation is shorter. By definition we have a production $\pi = A(s_1, \dots, s_\ell) \leftarrow S \in P$ with $S = \{B_1(x_{1,1}, \dots, x_{1,\ell_1}), \dots\}$, and it is easy to see how an analogous derivation is now possible in $\mathcal{G}$. Regarding the validity of $\vec{\mathbf{q}}$ and $\vec{\mathbf{p}}_i$, we also refer to the definition of $P_\cap$: For $s_{i'} = w_{0,i'} y_{1,i'} w_{1,i'} \cdots y_{n_{i'},i'} w_{n_{i'},i'}$, we have $q_{i'} \xrightarrow{w_{0,i'}} \tilde{q}_{1,i'}$, $\tilde{q}'_{j'-1,i'} \xrightarrow{w_{j'-1,i'}} \tilde{q}_{j',i'}$, and $\tilde{q}'_{n_{i'},i'} \xrightarrow{w_{n_{i'},i'}} q'_{i'}$ by definition. Furthermore, for each $y_{j',i'}$ by induction we have a word $v_{j',i'}$ witnessing $\tilde{q}_{j',i'} \xrightarrow{v_{j',i'}} \tilde{q}'_{j',i'}$. Stitching all these transition sequences together yields a run from $q_{i'}$ to $q'_{i'}$ for $s_{i'}$ and each i' as required.
  The other direction is similar.

It remains to argue about the time it takes to construct $\mathcal{G}_\cap$. For each nonterminal $A^{(\ell)}$ of $\mathcal{G}$, we need to add $|Q|^{2\ell}$ many nonterminals to $\mathcal{G}_\cap$, which takes time exponential in d, since d is the upper bound for ℓ, but otherwise requires time polynomial in $|\mathcal{G}|$ and $|\mathcal{A}|$.

Similarly for each Production $\pi = A(s_1, \dots, s_\ell) \leftarrow S$ of $\mathcal{G}$, we may need to add up to $|Q|^{2\ell} \cdot |Q|^{2d \cdot |S|}$ many productions to $\mathcal{G}_\cap$, depending on whether such a production could lead to a valid path in $\mathcal{A}$ or not. Basically, in the worst case, we need to go over all possibilities of indices $\vec{\mathbf{q}}$ and $\vec{\mathbf{p}}_i$ on both sides of the production, of which there are $|Q|^{d \cdot (r+1)}$ many in the worst case, since r is the bound for $|S|$. This is of course exponential in d and r, but otherwise polynomial in $|\mathcal{G}|$ and $|\mathcal{A}|$.

Finally, for each production we add, it may be necessary multiple times to check existence of a path $p \xrightarrow{w} q$ in $\mathcal{A}$ for some terminal word $w \in \Sigma^*$ appearing as an infix in a production. However, this check is clearly polynomial in $|w|$ and $|\mathcal{A}|$, and $|w|$ is clearly dependant on the length of said production. So this just adds a polynomial factor to the already described runtime. □

Prefix language of an MCFG using rational transductions. We prove the following statement:

Theorem J.3. *Given an MCFG $\mathcal{G}$ of constant dimension and rank, as well as a transducer $\mathcal{T}$, one can compute an MCFG for the language $\mathcal{T}(\mathcal{L}(\mathcal{G}))$ in polynomial time.*

Let us first explain what a transducer is and what the notation $\mathcal{T}(-)$ means. For an alphabet Σ we define $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. A *transducer* is basically an NFA, where rather than one label from Σ_ε, each transition carries two such labels, one for the input and one for the output. An run of a transducer reaching a final state thus accepts a pair of words $(u, v) \in \Sigma^* \times \Sigma^*$, where u is the concatenation of all input labels, and v is the concatenation of all the output labels. Thus, a transducer computes a relation $\subseteq \Sigma^* \times \Sigma^*$, called a *rational transduction*.

The notation $\mathcal{T}(L)$ for a transducer $\mathcal{T}$ and a language L refers to the image of L under the rational transduction computed by $\mathcal{T}$. More formally, $v \in \mathcal{T}(L)$ iff there is a $u \in L$ such that (u, v) is accepted by $\mathcal{T}$. Note that the notion of "full trio" alluded to in Appendix J.1 refers to exactly the language classes that are closed under rational transductions (see e.g. [13]). It is known that this closure property holds for MCFL [74], but we also need a specific time complexity.

With transducers explained, it is clear that Theorem J.3 implies the following:

Corollary J.4. *Given an MCFG $\mathcal{G}$ of constant dimension and rank, we can compute an MCFG for its prefix language* $\operatorname{pref}\mathcal{L}(\mathcal{G})$ *in polynomial time.*

This is part (b) of Lemma 7.3.

Proof. The following 2-state transducer maps a language $L \subseteq \Sigma^*$ to the set of all prefixes of its words. For each $a \in \Sigma$ there is loop labelled by (a, a) on the initial state q_0, and a loop labelled by (a, ε) on the only final state q_f. Then there is one more transition $q_0 \xrightarrow{(\varepsilon,\varepsilon)} q_f$. The transducer outputs each letter in the input word, until a nondeterministically chosen point, after which all letters are dropped. This is clearly the same as computing any prefix. The statement then follows from Theorem J.3. □

Let us now describe how to prove Theorem J.3. We write $\sqcup\!\sqcup$ to denote the shuffle operator. Formally, given two languages $L_1, L_2 \subseteq \Sigma^*$, we have $w = u_0 v_1 u_1 \cdots v_n u_n \in L_1 \sqcup\!\sqcup L_2$ iff $u = u_0 \cdots u_n \in L_1$ and $v = v_1 \cdots v_n \in L_2$, where $u_0, \dots, u_n, v_1, \dots, v_n$ are (possibly empty) infixes in Σ^*. Now, given a transducer $\mathcal{T}$ and a language L, we argue $\mathcal{T}(L) = h\big((L \sqcup\!\sqcup \Delta^*) \cap R\big)$, where Δ is the set of transitions of $\mathcal{T}$, h is a homomorphism mapping each $\delta \in \Delta$ to its output and removing all letters $\notin \Delta$, and R is a regular language that checks that each δ is preceded by its input and that the letters from Δ form an accepting run of $\mathcal{T}$. Then we compute the intersection with R using Corollary J.2, whereas applying h and shuffling with Δ^* are simple constructions.

Note that this way of writing $\mathcal{T}(L)$ is an adaptation of Nivat's theorem (see e.g. [17, Chapter III, Theorem 3.2]).

Proof of Theorem J.3. Let $\mathcal{G} = (\Sigma, \hat{N}, X, P, A_{\text{start}})$ be an MCFG of constant dimension and rank, and let $\mathcal{T}$ be a transducer over Σ with set of transitions Δ.

We define the homomorphism $h\colon \Sigma \cup \Delta \to \Sigma$ as $h\colon a \mapsto \varepsilon$ for each $a \in \Sigma$ and $h\colon \delta \mapsto \text{output}(\delta)$ for each $\delta \in \Delta$. Furthermore, we construct the NFA $\mathcal{A}$ from $\mathcal{T}$ as follows: For each transition δ with input $\sigma \in \Sigma_\varepsilon$ in $\mathcal{T}$ we replace its label by the concatenation $\sigma\delta$, splitting it into two transitions and adding a fresh state in-between, if necessary. The resulting NFA clearly accepts all words over $\Sigma \cup \Delta$, where each transition in Δ is preceded by its input, and the maximal subword in Δ^* forms an accepting run of $\mathcal{T}$.

Let $L = \mathcal{L}(\mathcal{G})$. Now we argue how to compute an MCFG for $h\big((L \sqcup\!\sqcup \Delta^*) \cap R\big)$ in polynomial time. For $L \sqcup\!\sqcup \Delta^*$, we transform $\mathcal{G}$ as follows. First, we introduce a new nonterminal $C^{(1)}$, and add each $\delta \in \Delta$ as a terminal. Then, for every production π, we insert a fresh variable before and after every variable and terminal symbol appearing in the left side of π, resulting in say ℓ new variables. Then we add ℓ instances of C to the right side of π, each pertaining to a different one of the new variables. Finally we add for each $\delta \in \Delta$ the production $C(x\delta) \leftarrow \{C(x)\}$ for some $x \in X$, and we also add the production $C(\varepsilon) \leftarrow \emptyset$. Clearly the resulting MCFG derives words from L with arbitrary many elements of Δ shuffled between their letters. We at most triple the length of each production, and add $|\Delta| + 1$ new productions of fixed length, so the whole process takes polynomial time.

We continue by intersecting the resulting MCFG with the NFA $\mathcal{A}$. This takes polynomial time via Corollary J.2, and results in another MCFG. Finally, applying $h(-)$ to the latter grammar is easy. We simply replace every terminal symbol in every production of the grammar according to h, which clearly takes polynomial time.

It remains to argue that the final resulting MCFG indeed derives the language $\mathcal{T}(L)$. To see this, consider what our constructions do on the side of formal languages. We start with L and shuffle an arbitrary transition sequence of $\mathcal{T}$ into each of its words. Then we check that each transition sequence is a valid accepting run, each transition is preceeded by its input, and each symbol in Σ is succeeded by a transition that it inputs into. Finally, we remove all input symbols and replace all transitions by their output symbols. In sequence this is exactly applying $\mathcal{T}(-)$. □

J.3 Partially Permuting Downward Closure

Let us recall the definition of our partially permuting downward closure, and its accompanying lemma. We will then prove the latter.

For a language $L \subseteq \Gamma^*$, a subalphabet $\Theta \subseteq \Gamma$, and a number k let $L{\downarrow_{k,\Theta}}$ denote its alphabet-preserving downward closure, where subwords are not allowed to drop letters from Θ, and have to contain exactly k of them. This is slightly different from the definition in [16], where words with fewer than k letters are also included, but our version fits our purpose slightly better. Formally, $L{\downarrow_{k,\Theta}} := \{u \in \Gamma* \mid \exists v \in L\colon u \preceq v \wedge |u|_\Theta = |v|_\Theta = k\}$. Here, $|w|_\Theta$ denotes the number of occurrences of letters from the alphabet Θ in the word w.

Furthermore, we define the partially permuting downward closure: Let $L{\downarrow^{\Psi}_{k,\Theta}}$ denote the variant of $L{\downarrow_{k,\Theta}}$, where every maximal infix in $(\Gamma \setminus \Theta)^*$ only needs to be preserved up to its Parikh-image $\Psi(-)$. Formally $u = u_0\theta_1u_1 \cdots \theta_k u_k \in L{\downarrow^{\Psi}_{k,\Theta}}$ if and only if $u_0, \ldots, u_k \in (\Gamma \setminus \Theta)^*$, $\theta_1, \ldots, \theta_k \in \Theta$, and there is $v = v_0\theta_1v_1 \cdots \theta_k v_k \in L{\downarrow_{k,\Theta}}$ such that $\Psi(u_i) = \Psi(v_i)$ for every $0 \leq i \leq k$.

Lemma 7.4. *Let $\mathcal{G}$ be an accelerated MCFG with $\Theta \subseteq \Gamma$, where for every tree t of $\mathcal{G}$ there is a tree t' of $\mathcal{G}$, such that $L(t){\downarrow} \subseteq L(t'){\downarrow}$ and $\text{height}(t') \leq c|\mathcal{G}|$ for a constant c as in Lemma 6.4. We can compute an NFA for $\mathcal{L}(\mathcal{G}){\downarrow^{\Psi}_{k,\Theta}}$ in time $2^{\text{poly}(k\cdot|\mathcal{G}|)}$.*

Proof. Let $\mathcal{G} = (\Gamma, \hat{N}, X, P, A_{\text{start}})$ be an accelerated MCFG, let $\Theta \subseteq \Gamma$ be a subalphabet, and let $\Sigma = \Gamma \setminus \Theta$. We construct an NFA $\mathcal{A}$ for $\mathcal{L}(\mathcal{G}){\downarrow^{\Psi}_{k,\Theta}}$ in the following manner. Each state of $\mathcal{A}$ encodes

a path in a derivation tree of $\mathcal{G}$, storing at each step the production used at that node and the yield of all left siblings. Starting with the empty branch, $\mathcal{A}$ guesses a production for A_{start}, then builds a branch according to left-first depth-first search, until a leaf is encountered. At a leaf, or whenever a nonterminal A is assigned a full tuple containing no more variables, $\mathcal{A}$ moves onto its parent node and replaces the variables of A in the production at said parent according to this assignment, subject to the following modifications: Instead of a letter $a \in \Sigma$, its Parikh image $\Psi(a)$ is stored, and whenever several Parikh images appear concatenated, we instead add them together. So if we identify $\mathbb{M}[\Sigma]$ and $\mathbb{N}^{|\Sigma|}$, then for each nonterminal of arity ℓ in a production, we store a tuple consisting of ℓ words over $X \cup \mathbb{N}^{|\Sigma|} \cup \Theta$. Variables from X remain in an instance of a production until all its children in the derivation tree have been explored.

Since $\mathcal{G}$ is not a normal MCFG, $\mathcal{A}$ also needs to handle terminals of the form $\tilde{\Gamma}^*$ for some $\tilde{\Gamma} \subseteq \Gamma$. To this end we extend the values in vectors for Parikh images to $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$, where ω denotes arbitrarily many occurrences. For adding vectors together we use the obvious semantics $\omega + n = n + \omega = \omega + \omega = \omega$ for any $n \in \mathbb{N}$. Since $\tilde{\Gamma}$ may contain letters from Θ, which we do not want to drop, whenever $\mathcal{A}$ encounters a terminal $\tilde{\Gamma}^*$, it first guesses an arbitrary word w of length $\leq k$ with $w \in \tilde{\Gamma}^* \cap \Theta^*$. Then before and after each letter of w, $\mathcal{A}$ inserts a vector containing ω for each $a \in \Gamma \cap \Sigma$. The resulting word is then used in place of $\tilde{\Gamma}^*$.

During the traversal of the tree, the NFA checks that the computed tuple of words contains at most k letters of Θ. If any assignment contains more than k such letters, $\mathcal{A}$ rejects. Finally, once every variable for A_{start} is assigned, $\mathcal{A}$ checks that the resulting word contains exactly k letters in Θ. Then $\mathcal{A}$ starts outputting letters according to the stored assignment. For vectors in $\mathbb{N}_\omega^{|\Sigma|}$, letters are output in a nondeterministically chosen permutation, and for each value of ω, a nondeterministic amount is output on a loop, before moving on. Once the whole word for A_{start} has been output, $\mathcal{A}$ moves to an accepting state. From the construction, it is clear that $\mathcal{A}$ computes the language $\mathcal{L}(\mathcal{G})\downarrow_{k,\Theta}^{\Psi}$.

It remains to analyze the size of $\mathcal{A}$. Let d be dimension of $\mathcal{G}$ and let r be its rank. Any assignment to a single nonterminal contains at most $2+k+d \cdot r$ many non-vector symbols. Since two vectors will always get added together if there is no other symbol between them, we have at most $4 + 2k + 2d \cdot r$ vectors in $\mathbb{N}_\omega^{|\Sigma|}$ stored per production. By our requirements, we only need to consider derivation trees of height $c|\mathcal{G}|$. Let m be the maximum number of terminal symbols in a production in P. Then the non-ω values in the vectors across the entire tree are bounded by $m \cdot \ell^{c|\mathcal{G}|}$, which for vectors of dimension $|\Sigma|$ require only polynomially many bits to store. Since each path is at most $c|\mathcal{G}|$ long, each state of $\mathcal{A}$ therefore has polynomial description size, meaning there are exponentially many possibilities for the states. □