Context-Bounded Verification of Context-Free Specifications

PASCAL BAUMANN, Max Planck Institute for Software Systems (MPI-SWS), Germany
MOSES GANARDI, Max Planck Institute for Software Systems (MPI-SWS), Germany
RUPAK MAJUMDAR, Max Planck Institute for Software Systems (MPI-SWS), Germany
RAMANATHAN S. THINNIYAM, Max Planck Institute for Software Systems (MPI-SWS), Germany
GEORG ZETZSCHE, Max Planck Institute for Software Systems (MPI-SWS), Germany

A fundamental problem in refinement verification is to check that the language of behaviors of an implementation is included in the language of the specification. We consider the refinement verification problem where the implementation is a multithreaded shared memory system modeled as a multistack pushdown automaton and the specification is an input-deterministic multistack pushdown language. Our main result shows that the *context-bounded* refinement problem, where we ask that all behaviors generated in runs of bounded number of context switches belong to a specification given by a Dyck language, is decidable and coNP-complete. The more general case of input-deterministic languages follows, with the same complexity.

Context-bounding is essential since emptiness for multipushdown automata is already undecidable, and so is the refinement verification problem for the subclass of regular specifications. Input-deterministic languages capture many non-regular specifications of practical interest and our result opens the way for algorithmic analysis of these properties. The context-bounded refinement problem is coNP-hard already with deterministic regular specifications; our result demonstrates that the problem is not harder despite the stronger class of specifications. Our proof introduces several general techniques for formal languages and counter programs and shows that the search for counterexamples can be reduced in non-deterministic polynomial time to the satisfiability problem for existential Presburger arithmetic.

These techniques are essential to ensure the coNP upper bound: existing techniques for regular specifications are not powerful enough for decidability, while simple reductions lead to problems that are either undecidable or have high complexities. As a special case, our decidability result gives an algorithmic verification technique to reason about reference counting and re-entrant locking in multithreaded programs.

CCS Concepts: • **Theory of computation → Concurrency**; • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: refinement verification, multithreaded programs, context bounded, Dyck language, inclusion problem, computational complexity

ACM Reference Format:
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, Article 73 (January 2023), 30 pages. https://doi.org/10.1145/3571266

Authors' addresses: Pascal Baumann, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, pbaumann@mpi-sws.org; Moses Ganardi, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, ganardi@mpi-sws.org; Rupak Majumdar, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, rupak@mpi-sws.org; Ramanathan S. Thinniyam, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, thinniyam@mpi-sws.org; Georg Zetzsche, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, georg@mpi-sws.org.

2475-1421/2023/1-ART73
https://doi.org/10.1145/3571266

1 INTRODUCTION

1.1 Context-Bounded Refinement Verification

A fundamental problem in refinement verification is to check that the language of behaviors of an implementation is included in the language of a given specification. We consider the problem of refinement verification where the implementation is a multithreaded shared memory program modeled as a multistack pushdown automaton (MPDA) and the specification is given by an *input-deterministic* multistack pushdown automaton. An MPDA generalizes usual pushdown automata by maintaining multiple stacks. It is input-deterministic if every step is completely determined by the input. While the general refinement verification problem is undecidable already for regular specifications, our main result shows that the *context-bounded* refinement verification problem is decidable and coNP-complete. Context-bounded refinement for regular specifications is already coNP-hard; thus, our result shows that the complexity of the problem does not increase even when the specification comes from a substantially larger class!

Context-bounding is a popular and general technique to construct a parameterized sequence of under-approximations of all behaviors of a program [Qadeer and Rehof 2005]: for each K, a K-context-bounded analysis considers only those behaviors in which threads have been context switched at most K times. As K increases, more and more behaviors are considered and, in the limit, all behaviors are covered. Context-bounded analyses are decidable for many verification problems that are undecidable without the restriction. In practice, it has been very successful as a bug finding tool, since many bugs in practical instances can be discovered even with small values of K [Inverso et al. 2022; La Torre et al. 2009; Musuvathi and Qadeer 2007; Qadeer and Rehof 2005].

Our result is surprising because most problems relating to context-free specifications are undecidable. Moreover, existing techniques crucially depend on reducing context-bounded analysis to a sequential program analysis and do not work if the specification carries its own stack. Indeed, our proof requires several new constructions of independent interest and, as we mention below, requires new analysis techniques to avoid close-by problems with high complexity or undecidability status.

The ability to capture input-deterministic MPDA specifications allows us to apply algorithmic analysis techniques to a number of common design patterns, such as re-entrant locks or concurrent reference counting, whose specifications are non-regular. In particular, the specification for a reference counted object is a counter (the count) that increases every time a new reference is taken and decremented when it is released. The invariant tracked by the system is that the object is not deallocated while at least one reference is active. While one can model the counter explicitly, the resulting multithreaded program with a counter does not fall into a known decidable class and existing techniques either apply manual proof steps or apply heuristics with no general termination guarantees [Emmi et al. 2009; Farzan et al. 2014].

The core technical step in our result involves showing a coNP upper bound when the specification is a *Dyck language* (that is, a non-regular language of "matched parentheses" with several different kinds of parentheses). The more general setting follows through standard techniques. Existing techniques to prove decidability of context-bounded reachability sequentialize the set of context-bounded executions to a single-threaded program (see [Lal and Reps 2009] for a clear exposition). Unfortunately, this technique does not work when the specification is non-regular: the stacks of the program and the stack of the specification may interact in complex ways and an unbounded amount of information may need to be passed across context switch points. Instead, we use characterizations of Dyck languages and develop a number of new techniques.

We note that even in the *sequential* setting, most interprocedural dataflow analysis algorithms restrict attention to regular properties—either finite-state *typestates* or weighted reachability with

weights satisfying strong algebraic properties [Reps et al. 1995, 2005; Sagiv et al. 1996]. While there are a few instances of context-free specifications [Ferles et al. 2021; Madhavan et al. 2015], these are based on heuristics and come without proofs of decidability. Instead, we use techniques from the theories of formal languages and counter machines. Our starting point is the result that checking whether a context-free language is contained in a Dyck language can be performed in polynomial-time [Tozawa and Minamide 2007a] (the problem has a rich history described in more detail below). In the next section, we describe the additional obstacles posed by our setting, which require significant new constructions.

It should be mentioned that just because the words produced by a multi-threaded program are contained in a (context-free) Dyck language, it does not mean that the program is behaviorally equivalent to a sequential program. First, there are languages of (context bounded) multi-threaded recursive programs that are included in the Dyck language, but that are not context-free[1]. Second, even if the set of parenthesis expressions produced by a multi-threaded program happens to be context-free, automatically transforming the program into a sequential one would be even more difficult than merely checking inclusion. Thus, in any case, we need to deal with multi-threaded programs as input.

Before we go into the technical aspects, let us mention potential practical implications. While we explore theoretical foundations, we do show that the non-inclusion problem can be reduced (non-deterministically in polynomial time) to the truth problem in existential Presburger arithmetic, for which there are established and efficient tools available [de Moura and Bjørner 2008]. Whether our reductions are fast in practice without modification, however, remains to be seen in future work.

1.2 Challenges and Key Ingredients

As mentioned above, existing methods for checking containment in Dyck languages of sequential recursive programs do not suffice in the presence of context switching. These methods hinge on the saturation technique, and in particular the fact that if the inclusion holds, then each entire procedure can be summarized by one of finitely many reduced words, even if this procedure is able to generate infinitely many words. This is not true in the concurrent setting. For example, consider the language $L = \{u^n v^m \bar{u}^n \bar{v}^m \mid m, n \geq 0\}$, where u and v are words of opening parentheses and $\bar{u}$ and $\bar{v}$ are the corresponding matching strings of closing parentheses, i.e. $u\bar{u}$ and $v\bar{v}$ are well-bracketed. Then the language L can be generated by a program with 2 threads and 3 context switches: One thread produces u^n and $\bar{u}^n$, the other v^m and $\bar{v}^m$ (using the call stack to store u^n, and v^m, respectively).

Now one can observe that L only contains well-bracketed words if and only if the equation $u^x = v^y$ has a solution for numbers $x, y \geq 0$. This means, there can be non-trivial (and infinite) interaction between concurrently executed procedures. In particular, to prove inclusion, we cannot summarize the two procedures (one producing $u^n\bar{u}^n$, the other producing $v^m\bar{v}^m$) by finitely many possible effects, but we need to reason about what they have produced *when they are interrupted.*

To overcome this, in addition to existing techniques (compression by straight-line programs, Parikh images), we employ *vector addition systems with states (VASS)*, which are an abstract model of computation with counters. To our knowledge, VASS have not been used before in the context of inclusion checking of recursive programs. However, the reachability problem in VASS has the extremely high complexity of Ackermann-completeness [Czerwiński and Orlikowski 2021; Leroux 2021; Leroux and Schmitz 2019] and even two-dimensional VASS (i.e. the case of two counters,

[1] $L = \{(a\bar{a})^n(b\bar{b})^m(a\bar{a})^n(b\bar{b})^m \mid m, n \geq 0\}$ is included in the Dyck language, but is not context-free. Here, a and b are opening brackets, and $\bar{a}$ and $\bar{b}$ are matching closing brackets. It only requires 3 context switches.

which we are able to reduce to) is still PSPACE-complete [Blondin et al. 2021]. Therefore, we need *three novel ingredients* to achieve the coNP upper bound. We expect these to be applicable to a wider range of inclusion problems and give an overview of these techniques below.

Formally, we consider the following problem. We are given a *multi-pushdown automaton (MPDA)* as input, together with a context bound K. An MPDA is an automaton with n stacks, each of which represents the call stack of a thread in a multi-threaded shared-memory recursive program. The input letters of MPDA are drawn from A and $\bar{A} = \{\bar{a} \mid a \in A\}$. Here, the letters in A (respectively, $\bar{A}$) are opening (respectively, closing) parentheses. We want to check whether all words produced by the MPDA, with at most K context switches, belong to D_A, the set of well-bracketed words over $A \cup \bar{A}$.

We work with a characterization of D_A that says non-membership of w in D_A is due to three possible violations [Ritchie and Springsteel 1972]:

(DV) a *dip violation*, where w has a prefix with more $\bar{A}$ letters than A letters,
(OV) an *offset violation*, where the number of A letters in w differs from the number of $\bar{A}$ letters,
(MV) a *mismatch violation*, meaning an opening bracket is closed by the wrong type of closing bracket; in other words, there is an infix $au\bar{b}$ for some letters $a \neq b$, such that u has no dip violation and no offset violation.

We provide an NP algorithm that detects these violations, yielding the coNP upper bound for inclusion. Checking an MPDA for dip and offset violations works using slight adaptations of existing techniques; the difficult part is detecting mismatch violations. To do the latter, we need to find infixes $au\bar{b}$ as above. The set of words u that appear between mismatched letters a,$\bar{b}$ can also be described by an MPDA. To make sure that such a word u has no dip violation and no offset violation, one could naively equip such an MPDA with an additional counter and try to decide reachability. Unfortunately, already for pushdown automata with an additional counter, decidability of the reachability problem is a long-standing open problem [Englert et al. 2021; Leroux et al. 2015].

Ingredient I: Run decomposition and 2-VASS. Therefore, another approach is needed. Our *first key ingredient* is to decompose the MPDA run reading u into a polynomial number of run pairs (π_1, π_2). Here, the parts π_1 and π_2 of each pair (π_1, π_2) together form the run of a pushdown automaton corresponding to a single stack in the run on u (but in this MPDA run, there might be an interruption between π_1 and π_2). A similar decomposition has been used in [Shetty et al. 2021], but the crucial trick here is to simulate each run pair (π_1, π_2) by a two-dimensional VASS (2-VASS). Specifically, we simulate (π_1, π_2) *top-down* instead of *left-to-right*: Each step of the 2-VASS corresponds to a letter on the stack that π_1 leaves behind (and π_2 consumes). In this way, π_1 is simulated forward, but π_2 is simulated backward. Since the reachability relation of 2-VASS are definable in Presburger arithmetic [Leroux and Sutre 2004]—the first-order theory of the structure $(\mathbb{N}, +, 0)$—this would allow us to build one formula in this logic that connects all the 2-VASS runs, and expresses the existence of u.

However, this would result in much higher complexity than NP. The first issue is that in order to directly construct the 2-VASS that simulates the pushdown runs, we would need to know the *minimal dip* of any word generated by a given pushdown automaton. Here, the *dip* of a word v is the maximal difference $|p|_{\bar{A}} - |p|_A$ among all prefixes p of v. In terms of complexity, computing this number is equivalent to the coverability problem of machines with one stack and one counter. This problem is known to be decidable [Leroux et al. 2015], but PSPACE-hard [Englert et al. 2021], and the best known upper bound is EXPSPACE [Englert et al. 2021].

Ingredient II: Annotations. Therefore, the *second key ingredient* is to circumvent this computation. Instead of computing the minimal dips for the pushdown machines arising from the input, we first

```
1  bool stoppingFlag = 0;
2  bool stoppingEvent = 0;
3  status = STOPPED;
4
5  main() {
6    take();
7    status = STARTED;
8  }
9  worker(id i) {
10   assume(status==STARTED);
11   work();
12 }
13 stopper() {
14   assume(status==STARTED);
15   stoppingFlag = 1;
16   drop();
17   wait(stoppingEvent);
18   status = STOPPED;
19   // reclaim device
20 }
21 work() {
22   ret = take();
23   if ret == 0 { // work on device
24     assert(status != STOPPED);
25     if (*) work(); // do more work
26     drop();
27   }
28 }
29
30 take() {
31   if stoppingFlag return -1; // no more allowed
32   IncRef();
33   return 0;
34 }
35
36 drop() {
37   DecRef();
38   if ZeroRef()
39     notify(stoppingEvent);
40 }
41 program  main || worker(1) || ... || worker(n) || stopper; // for constant n
```

Fig. 1. Simplified device management in Windows drivers.

enlarge the input language L: We show that one can expand the MPDAs so that (i) minimal dips are easy to compute for the new machines and (ii) the enlarged language L' is included in D_A if and only if L is. The latter argument employs new insights into the Dyck language that we have not seen in the literature. We call this process *annotation* (of minimal dips) and believe that this technique will be useful for testing inclusion in D_A for other types of programs.

Ingredient III: Offset-uniform 2-VASS. A further complexity issue is that in order to achieve our NP upper bound (for non-inclusion), we need the resulting (existential) Presburger formulas to be computable in NP. Unfortunately, it is not possible (unless NP = PSPACE) to compute in NP an existential Presburger formula for the reachability relation of a given 2-VASS (if, as in our case, the counter updates are given in binary): This is because already the reachability problem for 2-VASS is PSPACE-complete [Blondin et al. 2021]. Therefore, our *third key ingredient* is to observe that the resulting 2-VASS inhabit a newly-identified subclass (offset-uniform 2-VASS), for which we show that one can in NP compute an existential Presburger formula for their reachability relation.

Putting all these ingredients together, we can construct in NP an existential Presburger formula that expresses the existence of a word u as above. Since truth of existential Presburger arithmetic is decidable in NP [Borosh and Treybig 1976], we obtain the desired NP algorithm for non-inclusion.

1.3 Concrete Motivating Examples

Let us consider a concrete example. Figure 1 shows a simplified example of a concurrent reference-counted implementation in device management within the Windows kernel (this example is a simplification of the code from Qadeer and Wu [2004], which can be considered an early precursor to context-bounded analyses). Each device maintains a reference count. When the device needs to be stopped, a protocol ensures that the device is not unloaded until the reference counts are given up. That is, a `stopper` process sets a `stoppingFlag` preventing further requests and waits for the existing references to be given up. When the reference count reaches zero, the stopper is notified and the device is stopped. We assume each statement is executed atomically but context switches can occur in between. In language-theoretic terms, the concurrent implementation generates a language over `IncRef`, `DecRef`, `ZeroRef`, and `Error` (error is raised if the assertion on line 24

fails). A natural specification for the program is that (1) there is no `Error`, (2) on termination, each behavior has exactly the same number of increment and decrement operators, (3) every prefix has at least as many increments as decrements, and (4) zero checks are correct. Seen as a sequence over the reference counts, the counter should always be non-negative, be equal to zero iff a zero test succeeds, and be zero at the end. Since the counter can grow unboundedly, the specification is not regular and the example is beyond the scope of known decidable classes. In contrast, the specification is an input-deterministic MPDA language (in fact, it can be encoded by a two-letter Dyck language) and our result shows that it is algorithmically decidable. We note that the program violates the specification (how?) and a violation can be found with a context bound of 1.[2]

Another situation where we need to check inclusion in the Dyck language is the verification of programs that generate code or documents with nesting structures [Madhavan et al. 2015] (see also [Ferles et al. 2021]). First, most modern programming languages feature several kinds of parentheses (for control flow, function calls, arithmetical expressions) that need to be well-nested. Second, document formats with tree-like structure (such as XML) require opening and closing tags to match. Thus, for verifying syntactic correctness of the generated code or documents, checking inclusion in the Dyck language is a necessary step.

1.4 Related Work

Context-bounded reachability was introduced by Qadeer and Rehof [2005] as a decidable procedure for underapproximating the reachable state space of a multi-threaded shared memory program. The restriction was inspired by the empirical observation that a small context bound is often sufficient to identify many bugs in concurrent code [Musuvathi and Qadeer 2007]. Since then, the decidability and complexity frontier of context-bounding in several associated models has been explored intensively [Atig et al. 2011; Baumann et al. 2020; Inverso et al. 2022; La Torre et al. 2009, 2010; Lal and Reps 2009; Madhusudan and Parlato 2011; Meyer et al. 2018; Shetty et al. 2021; Torre et al. 2020]. Atig et al. [2011] extended decidability of context-bounded reachability to a model with dynamic spawning of threads. Baumann et al. [2021] showed liveness verification is also decidable in this setting. Madhusudan and Parlato [2011] prove a very general decidability result based on bounded treewidth. Lal et al. [2008] showed that context-bounded analysis can be generalized to a setting in which the data comes from an infinite domain that satisfies certain "nice" properties, namely, it forms a bounded idempotent semiring. Their result implies, in particular, decidability of some quantitative properties such as the length of the shortest path between configurations.

Visibly pushdown automata (or equivalently, automata over nested words) have been studied as subclasses of context-free languages with good decidability properties [Alur and Madhusudan 2004, 2009]. They have been the basis for defining specification logics and decidability results for model checking [Alur et al. 2011, 2004]. However, most work on software model checking with nested word specifications have been applied to the sequential setting. Further, since the language inclusion problem for arbitrary CFLs into visibly pushdown automata is undecidable [Filiot et al. 2018, Prop. 11], we cannot derive analogous results with visibly pushdown specifications. There are practical specifications (such as reference counting on two objects) that are input-deterministic MPDA languages but not visibly pushdown languages.

Our proof uses several language-theoretic techniques that have been applied to the inclusion problem for context-free languages. Balanced context free languages have been studied since the

[2]Consider the execution in which `main` finishes, followed by a worker thread that executes line 31, finds `stoppingFlag` is false, and then gets swapped out just before executing line 32. At this point, the `stopper` is scheduled, sets `stoppingFlag` to true, and calls `drop`. Since the reference count is one, `drop` decrements, checks that the reference count is zero, and sets `stoppingEvent`. The `stopper` continues to run and stops and reclaims the device. At this point, the interrupted worker runs again, performs the increment and fails the assertion since the device is stopped.

1960s. The Dyck inclusion problem for CFLs has been studied by many researchers. A polynomial time algorithm for inclusion in the unary Dyck language (balanced parenthesis languages with one kind of parentheses) was given by Knuth [1967]. Berstel and Boasson [2002] built on this result, giving an algorithm for Dyck inclusion over an arbitrary alphabet. Finally, Tozawa and Minamide [2007a] showed a polynomial-time procedure for the problem[3].

Inclusion in Dyck languages has also been studied for languages beyond the context-free languages. Maneth and Seidl [2018] have shown that if there is only *one pair of parentheses*, then inclusion is decidable for ranges of MSO tree-to-string transducers. They also provide various complexity bounds, depending on how the language is specified. Moreover, Löbel et al. [2021] show that for general Dyck languages, inclusion is decidable in polynomial time for ranges of two-copy tree-to-string transducers. Their result is incomparable to ours: First, there are MPDA languages[4] (with context-switching bound) that cannot be produced by two-copy tree-to-string transducers. Second, as far as we can see, their methods are fundamentally insufficient to deal with our setting[5]. However, their procedure also deals with non-trivial phenomena that do not occur in our setting. On the negative side, Kobayashi [2019] shows that for order-2 pushdown automata (a level in the hierarchy of higher-order pushdown automata), inclusion in the Dyck language is undecidable, already for one pair of parentheses.

The inclusion problem is known to be decidable between arbitrary context-free languages and those accepted by superdeterministic pushdown automata [Greibach and Friedman 1980]. We leave to future work if our techniques extend to the class of superdeterministic specifications.

In the context of program verification, Madhavan et al. [2015] studied heuristic algorithms for context-free inclusion, inspired by algorithms for decidable instances of context-free language *equivalence* between LL(k) grammars [Olshansky and Pnueli 1977; Rosenkrantz and Stearns 1970]. They showed that their rewrite rules could prove inclusion for several problems in practice, but did not show any general decidability result. Unfortunately, decidability of equivalence does not entail decidability of inclusion: checking if a linear LL(1) language is included in another is already undecidable [Friedman 1976]. Thus, the heuristic procedures of Madhavan et al. [2015] are unlikely to lead to decidability. Furthermore, the fact that *equivalence* is decidable for deterministic pushdown automata [Sénizergues 1997] has been applied to equivalence checking of Idealized Algol [Murawski et al. 2005; Ong 2002]. However, since inclusion is undecidable for deterministic pushdown automata, this does not apply to our setting.

Our work was inspired by Ferles et al. [2021], who studied safety verification of *sequential* programs against context-free specifications. We observed that the examples of context-free specifications in [Ferles et al. 2021] fall into the subclass of input-deterministic MPDA languages. Our result is a generalization of their work to the multithreaded setting. Since their implementation is based on the heuristic rewrites for checking context-free language inclusion from [Madhavan et al. 2015], they do not provide decidability or tractability results.

[3]There is also a polynomial-time algorithm due to Bertoni et al. [2011, Theorem 2] for inclusion of context-free languages in certain languages K defined by rewriting systems. It is claimed in [Bertoni et al. 2011, Lemma 5] that the algorithm also applies to $K = D_A$, but unfortunately, this is not the case. They assume that the monoid induced by the Dyck reduction $\twoheadrightarrow$ (see Section 2) is cancellative, but it is not: The words $a\bar{a}a = a \cdot \bar{a}a$ and $a = a \cdot \varepsilon$ are congruent, but $\bar{a}a$ and ε are not.

[4]For example, one can show that the language $\{a^m b^n a^m b^n \# c^k d^\ell c^k d^\ell \mid m, n, k, \ell \geq 0\}$ is not produced by a two-copy tree-to-string transducer.

[5]Intuitively, the two-copy restriction means that the words are built from two "context-free parts". Then, inclusion can only hold if the left part (resp. right part) has a Dyck normal form of only opening (resp. closing) parentheses. This allows Löbel et al. [2021] to analyze the two parts separately and then apply techniques for free groups. Roughly speaking, our setting allows arbitrary numbers of alternations among context-free parts, yielding more varied cross-part cancellation patterns.

2 CONTEXT-BOUNDED INCLUSION FOR MULTISTACK PUSHDOWN AUTOMATA

Programs. We take a language-theoretic view and define our programming model as multistack pushdown automata [Madhusudan and Parlato 2011].

We define some notation. For $n \in \mathbb{N}$ we define $[n] = \{1, 2, \ldots, n\}$, and furthermore for $m, n \in \mathbb{Z}$ with $m \leq n$ we define $[m, n] = \{m, m+1, \ldots, n\}$. We use bold letters such as $\mathbf{b}$ to denote vectors over $\mathbb{N}$ and $\mathbf{x}$ for a vector of variables over $\mathbb{N}$. We will also write b_i to denote the i^{th} component of $\mathbf{b}$. Let A be a finite alphabet. We write A_ε to denote $A \cup \{\varepsilon\}$. We write $\bar{A} = \{\bar{a} \mid a \in A\}$. Furthermore, if $x = \bar{a}$ for some $a \in A$, then we define $\bar{x} = a$; that is, the function $\bar{\cdot} : A \cup \bar{A} \to A \cup \bar{A}$ is an involution. Moreover, given a string $w \in (A \cup \bar{A})^*$, $w = a_1 \cdots a_n$, $a_1, \ldots, a_n \in A \cup \bar{A}$, we define $\bar{w} = \bar{a}_n \cdots \bar{a}_1$.

A *Multistack Pushdown Automaton* with m stacks (m-MPDA, or just MPDA) is a tuple $\mathcal{M} = (Q, q_0, \Sigma, \Gamma, \delta, Q_F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite input alphabet, Γ is a finite stack alphabet, $\delta = (\delta_{\mathsf{push}}, \delta_{\mathsf{pop}})$ defines a transition relation, and $Q_F \subseteq Q$ is a set of final states. We define $\delta_{\mathsf{push}} \subseteq Q \times \Sigma_\varepsilon \times Q \times \Gamma \times [m]$ as the set of *push moves* and $\delta_{\mathsf{pop}} \subseteq Q \times \Sigma_\varepsilon \times \Gamma \times Q \times [m]$ as the set of *pop moves*. The size of $\mathcal{M}$ is defined as $|\mathcal{M}| = |Q| + |\Sigma| + m \cdot |\Gamma| + |\delta_{\mathsf{push}}| + |\delta_{\mathsf{pop}}|$.

A *configuration* of $\mathcal{M}$ is a tuple $c = (q, w_1, w_2, \ldots, w_m)$, where $q \in Q$ and for each $i \in [m]$, $w_i \in \Gamma^*$ is the stack content of stack i. There is a transition $c \xrightarrow[\mathsf{push}_j(\gamma)]{a} c'$ from configuration $c = (q, w_1, w_2, \ldots, w_m)$ to configuration $c' = (q', w_1', w_2', \ldots, w_m')$ iff there is $(q, a, q', \gamma, j) \in \delta_{\mathsf{push}}$, $w_j' = \gamma w_j$ and for all $i \neq j, i \in [m]$ we have $w_j' = w_j$. Similarly, $c \xrightarrow[\mathsf{pop}_j(\gamma)]{a} c'$ iff there is $(q, a, \gamma, q', j) \in \delta_{\mathsf{pop}}$, $w_j = \gamma w_j'$ and for all $i \neq j, i \in [m]$, we have $w_j' = w_j$. In both cases we write $c \xrightarrow{a}_j c'$ to denote any transition involving stack j, or just $c \xrightarrow{a} c'$, if both stack number and operation do not matter.

We furthermore extend this notation to words $w \in \Sigma^*$. For $w = a_1 a_2 \cdots a_n$ and a sequence of transitions $c_0 \xrightarrow{a_1}_j c_1 \xrightarrow{a_2}_j \ldots \xrightarrow{a_n}_j c_n$ all involving the same stack j, we also write $c_0 \xrightarrow{w}_j c_n$. Similarly, for $w = a_1 a_2 \cdots a_n$ and a sequence of transitions $c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} c_n$ involving any number of stacks, we write $c_0 \xrightarrow{w} c_n$.

A *run* of $\mathcal{M}$ over a word $w \in \Sigma^*$ is a sequence of transitions $\pi = (q_0, \varepsilon, \varepsilon, \ldots, \varepsilon) \xrightarrow{w} (q_f, \varepsilon, \varepsilon, \ldots, \varepsilon)$, where $q_f \in Q_F$ is a final state. In this case we say that w is a word *accepted* by $\mathcal{M}$ and the language $L(\mathcal{M})$ is the set of all such words.

Intuitively, an m-MPDA represents a shared memory multithreaded program with m (recursive) threads, with shared state space Q. A configuration $(q, w_1, \ldots, w_m)$ describes a shared state q and the stacks of the m threads containing w_1 to w_m, respectively. Each transition corresponds to a step of some thread; in a step, a thread can push symbols on to its stack (a function call) or pop symbols from its stack (a return), based on the shared state Q. A run corresponds to an execution of the program. The alphabet Σ consists of visible events of a run, such as API calls like `lock` or `unlock` or program events like acquiring or releasing a reference.

Example 2.1. We present some simple examples to give an intuition on how "usual" multithreaded shared memory programs can be encoded as MPDAs. Fig. 2(a) contains a simple multithreaded program that runs n copies of a thread `T`. Each thread `T` nondeterministically calls a recursive procedure `a` with a Boolean argument ("choose"). The procedure `a` writes out a string of bits, by taking a re-entrant lock and nondeterministically deciding to call itself recursively or to return by giving up the lock. We model the program as an MPDA with n stacks, one for each thread. We use the stack alphabet 0 and 1 to store the execution stack for `a`, and additional letters to track the program location of each thread. We use `lock` and `unlock` as the alphabet of visible events. The control states in Q capture the control flow structure of the program. Since there is no other shared state, a single control state coordinates the scheduling. For each thread, the scheduler pops the top

```
1  T() {
2    choose  a(0); || a(1);
3  }
4  a(bool b) {
5    lock();
6    write(b);
7    if *
8      choose a(0); || a(1);
9    unlock();
10 }
11 program
12   T() || T() || ... || T()
```

```
1  global resource[1..t];
2  // initially, all resources are valid
3  W() {
4    rsrc = resource[1]; assume(rsrc.valid);
5    IncRef(rsrc);
6    while (*) {
7      assert(rsrc.valid);
8      choose i in [1..t];
9      new_rsrc = resource[i]; assume(new_rsrc.valid);
10     DecRef(rsrc);
11     rsrc = new_rsrc;
12     IncRef(rsrc);
13   }
14   DecRef(rsrc);
15 }
16 Cleanup() {
17   choose j in [1..t];
18   if ZeroRef(resource[j])
19     resource[j].valid = 0;
20 }
21 program W() || W() || ... || W() || Cleanup()
```

Fig. 2. (a) Re-entrant locks (b) reference counting

(to find the current control state), applies the operation (lock, unlock, write, or a nondeterministic choice among further calls), and pushes the new control state.

Fig. 2(b) contains a simple example with reference counts. There is a set of reference counted resources. Resources are valid (can be used) or invalid (freed). When the reference count for a resource reaches zero, it can be garbage collected by a `Cleanup` process. Each thread goes over the resources, acquiring a new resource and releasing the old one. Again, each thread has its own stack. The visible events, in red, correspond to taking and releasing references, and using a resource. In an MPDA encoding, the global state will contain, in addition to control coordination states, the state for each resource. Reasoning whether `assert` is only called on valid resources is still nontrivial. In fact, the program has a bug.[6] □

Specifications: Dyck Languages. We are interested in specifications given by *Dyck languages*. Let A be an alphabet and let $\Sigma = A \cup \bar{A}$ for the involution $\bar{\cdot}$. For a word $w \in \Sigma^*$, the *Dyck reduction* $\twoheadrightarrow \subseteq \Sigma^* \times \Sigma^*$ is defined by $w \twoheadrightarrow w'$ iff there exist words $u, v \in \Sigma^*$ and $a \in A$ such that $w = ua\bar{a}v$ and $w' = uv$. We represent the reflexive, transitive closure of $\twoheadrightarrow$ by $\twoheadrightarrow^*$ as usual. A word w is said to be *reduced* if $a\bar{a}$ is not an infix of w for any $a \in A$. It is well known that there exists a unique reduced word $\mathsf{DyckNF}(w)$ called the *normal form* of w, for any given word w such that $w \twoheadrightarrow^* \mathsf{DyckNF}(w)$. The *Dyck language (over A)* D_A is defined as $D_A = \{w \in \Sigma^* \mid \mathsf{DyckNF}(w) = \varepsilon\}$.

Intuitively, think of letters a in A as "opening brackets" and $\bar{a}$ as the corresponding "closing brackets." Viewed this way, a Dyck language is the language of all words with matched brackets.

Example 2.2. We give some examples of Dyck specifications. We use context-free grammar (CFG) notation for the languages. A re-entrant lock satisfies the specification

$$S \longrightarrow \varepsilon \mid \mathtt{lock}\, S\, \mathtt{unlock}\, S$$

Now, in multithreaded programs, a re-entrant lock is "blocked" for other threads as soon as one thread takes one lock. So, one could reduce the problem to checking correct usage for each lock.

[6] If `rsrc` and `new_rsrc` refer to the same resource, then an interruption of `W` on line 11 by `Cleanup` can free the resource. When `W` is resumed, it increments the reference to an invalid resource on line 12 and fails the assertion on line 7.

However, this is not true for other re-entrant lock types, such as Android's WakeLock, that prevents the device from going to sleep when some threads are performing important work. A WakeLock satisfies the same specification as a re-entrant lock, but allows multiple threads to hold the lock (multiple times) simultaneously (when it is used in reference counting mode).

A single reference counted object, such as those in the examples above, satisfies the specification

$$S \to \varepsilon \mid \texttt{IncRef}\, S\, \texttt{DecRef}\, S$$

This specification does not allow checking if a reference count is zero, but we can allow such checks by a slight encoding trick. We introduce a new letter Z (to mark an empty counter explicitly) and use the two-letter Dyck language

$$S \to \varepsilon \mid \texttt{Z}\, S\, \bar{\texttt{Z}} S \mid \texttt{IncRef}\, S\, \texttt{DecRef}\, S$$

In the program, we start by emitting a Z on initializing a refcount, and we emit a $\bar{\texttt{Z}}$ on termination. Each successful zero test emits $\bar{\texttt{Z}}\,\texttt{Z}$ and each unsuccessful zero test emits DecRef IncRef. This encoding ensures that a valid run of the program performs the zero tests correctly. □

Context-bounded Inclusion. Ideally, we would like to check if the language $L(\mathcal{P})$ of a program over an alphabet Σ is contained in a specification D_A. Unfortunately, this problem is undecidable, since the emptiness problem for 2-MPDAs is already undecidable. Thus, we focus on the *context-bounded* language of a program.

A *context-switch* in a run $\pi = c_0 \xrightarrow[s_1]{a_1} \cdots \xrightarrow[s_n]{a_n} c_n$ of an m-MPDA is a pair of consecutive transitions $c_i \xrightarrow[s_{i+1}]{a_{i+1}} c_{i+1} \xrightarrow[s_{i+2}]{a_{i+2}} c_{i+2}$ such that s_{i+1} and s_{i+2} are stack operations on stacks j and k, respectively, for some $j \neq k$. This context-switch is *associated* with stack j. The total number of context-switches $cs_\pi(i)$ associated with stack i in a run π is called the *context-switch number* of stack i. A contiguous sequence of transitions between two context-switches where all stack operations are performed on stack i (alternately between c_0 and the first context switch, or between the last context switch and c_n) is called a *segment* of stack i. The run π is said to be *K-context-bounded* if its total number of context switches does not exceed K, i.e. $\sum_{i\in[m]} cs_\pi(i) \leq K$. The *$K$-context-bounded language of $\mathcal{M}$*, denoted $L_K(\mathcal{M})$, is the set of words accepted by the K-context-bounded runs of $\mathcal{M}$.

The context-bounded **Dyck inclusion problem** is given as:

Input: An MPDA $\mathcal{M}$ with alphabet $\Sigma = A \cup \bar{A}$ and a number K in unary.
Question: Is $L_K(\mathcal{M}) \subseteq D_A$?

Our main result is the following theorem.

Theorem 2.3. *The context-bounded Dyck inclusion problem is* coNP*-complete.*

coNP-hardness for Dyck inclusion, even with $|A| = 1$, follows easily from the NP-hardness of context-bounded state reachability [Qadeer and Rehof 2005]. Given an instance of state reachability, we transform the program to output ε on each transition and to output a single $\bar{a}$ on reaching the target state. Thus, the language of the program is $\{\varepsilon\}$ if the target state is never reached and $\{\bar{a}\} \not\subseteq D_{\{a\}}$ if it is reached.

When $m = 1$, we note that an m-MPDA reduces to the model of *pushdown automata*. Pushdown automata accept exactly the class of *context-free languages* (CFLs). For this special case, Tozawa and Minamide [2007a] show that inclusion can be checked in polynomial time.

Theorem 2.4 ([Tozawa and Minamide 2007a]). *Given a CFL $L \subseteq (A \cup \bar{A})^*$, we can decide in polynomial time whether $L \subseteq D_A$.*

Our proof will extend this result to the K-context-bounded language $L_K(\mathcal{M})$ for $m > 1$. As we shall see, the extension requires several new techniques.

More General Specifications. Although Theorem 2.3 is ostensibly about the Dyck languages, it applies to significantly more complex specifications. An MPDA, over some input alphabet Σ (not necessarily $A \cup \bar{A}$) is called *input-deterministic* if (i) there are no transitions reading ε from the input and (ii) for every state q and every letter $x \in \Sigma$, there is at most one outgoing edge from q that reads x. Slightly extending the syntax of MPDA, we also allow an input-deterministic MPDA to (a) carry a sequence of stack operations on each transition (instead of just a single stack operation) and (b) specify an initial and final stack content for each stack (with the obvious semantics). Thus, the MPDA satisfies a strong determinism property that requires every step to solely depend on the input. (In contrast, in deterministic pushdown automata, a step may also depend on the stack; input-determinism disallows this.)

The (context-bounded) **input-deterministic MPDA inclusion problem** is the following:

Given An MPDA $\mathcal{M}$, a number K in unary, and an input-deterministic MPDA $\mathcal{N}$.
Question Does $L_K(\mathcal{M}) \subseteq L(\mathcal{N})$?

Note that the size of an input-deterministic MPDA compared to a general MPDA is (a) increased multiplicatively by the length of the longest sequence of stack operations occurring on one of its transitions, and then (b) increased additively by the lengths of each initial and final stack content.

Corollary 2.5. *The input-deterministic MPDA inclusion problem is* coNP*-complete.*

Proof. Suppose $\mathcal{N}$ has ℓ stacks with the stack alphabet A. For each $i \in \{1, \ldots, \ell\}$, we construct an MPDA $\mathcal{M}_i$ such that $L_K(\mathcal{M}_i) \subseteq (A \cup \bar{A})^*$ and we have $L_K(\mathcal{M}) \subseteq L(\mathcal{N})$ if and only if $L_K(\mathcal{M}_i) \subseteq D_A$ for every $i \in \{1, \ldots, \ell\}$. We achieve this using a simple product construction of $\mathcal{M}$ and $\mathcal{N}$: A run of $\mathcal{M}$, reading $w \in \Sigma^*$, results in a run of $\mathcal{M}_i$ that reads as input the word $us\bar{v}$, where (i) u is the initial content of stack i, (ii) $s \in (A \cup \bar{A})^*$ is the unique sequence of stack operations that $\mathcal{N}$ would perform on stack i when reading w, and (iii) v is the final content of stack i specified in $\mathcal{N}$. Here, $a \in A$ stands for pushing of a and $\bar{a}$ stands for popping a. If w does not lead $\mathcal{N}$ into a final control state, then $\mathcal{M}_i$ notices this, because the copy of $\mathcal{M}$ has reached a final state, but $\mathcal{N}$ has not. In that case, $\mathcal{M}_i$ also reads $a\bar{b}$ for some $a, b \in A$, $a \neq b$. Then, we clearly have $L_K(\mathcal{M}_i) \subseteq D_A$ for every $i \in \{1, \ldots, \ell\}$ if and only if $L_K(\mathcal{M}) \subseteq L(\mathcal{N})$. Thus, Theorem 2.3 yields the coNP upper bound.

Since every Dyck language is also the language of an input-deterministic MPDA, the lower bound is inherited from Theorem 2.3 as well. □

Example 2.6. The natural specification for Fig. 2(b) is an input-deterministic t-MPDA, where the previous specification for reference counts with zero tests is maintained for each of the t resources. In fact, with input-deterministic MPDAs, the specification for reference counts with zero tests becomes simpler because the machine can e.g. perform multiple zero tests simultaneously (by employing transitions carrying sequences of stack operations). We remark that both Fig. 1 and Fig. 2(b) do not satisfy the specification because both can raise `Error`. We invite the reader to find the problems, and note that $K = 1$ is sufficient to find the problems in each case (for the solutions, see Footnotes 2 and 6, respectively). These examples are taken from real bugs in Windows drivers (see [Qadeer and Wu 2004]) and in the Python runtime system (see https://www.python.org/doc/essays/refcnt/). □

The rest of the paper is devoted to show the upper bound in Theorem 2.3.

3 CHECKING INCLUSION

Let $\mathcal{M}$ be an m-MPDA over an alphabet $\Sigma = A \cup \bar{A}$ and let K be given in unary. We show the *non*-inclusion problem $L_K(\mathcal{M}) \not\subseteq D_A$ is in NP, by showing a sequence of nondeterministic polynomial time reductions to a problem in NP.

We shall assume familiarity with some basic concepts of language theory in our proof, such as pushdown automata (PDA), context-free grammars (CFG), and straight-line programs (SLP) representing compressed words; see, e.g., [Lohrey 2012; Sipser 2012] for more details.

3.1 Ways to Violate Dyck Inclusion and Proof Outline

For any alphabet Θ and words $u, v \in \Theta^*$, we say that u is a *prefix* of v if one can write $v = uw$ for some $w \in \Theta^*$. In this case, we write $u \leq_{\text{pre}} v$. For a language $L \subseteq \Theta^*$, we define $\text{pref}(L) = \{u \in \Theta^* \mid \exists v \in L \colon u \leq_{\text{pre}} v\}$ to be the set of all prefixes of words in L. Let A be an alphabet, let $\Sigma = A \cup \bar{A}$, and consider the Dyck language D_A. For a word $w \in \Sigma^*$, define the *offset* $\Delta(w)$ of w as $\Delta(w) = |w|_A - |w|_{\bar{A}}$ where $|w|_\Gamma$ denotes the number of letters in w belonging to $\Gamma \subseteq \Sigma$. A language L is *offset-uniform* if $\Delta(u) = \Delta(v)$ for all $u, v \in L$. The *dip* (or *drop*) $d(w)$ of w is defined as $d(w) = \max\{-\Delta(u) \mid u \leq_{\text{pre}} w\}$. We define $e(w) = (d(w), \Delta(w))$. Observe that if $|A| = 1$ then $w \in D_A$ if and only if $e(w) = (0, 0)$.

Take an arbitrary language $L \subseteq \Sigma^*$ such that $L \not\subseteq D_A$. We will rely on the fact that there is a word $w \in L$ witnessing the non-inclusion that satisfies one of the following violation conditions:

(OV) an *offset violation* $\Delta(w) \neq 0$,
(DV) a *dip violation*, where there is a prefix u of w with $d(u) > 0$, or
(MV) a *mismatch violation*, where there exists a pair $a, \bar{b}$ (for some $a \neq b$) of *mismatched* letters in w, i.e. w contains an infix $av\bar{b}$ where $e(v) = (0, 0)$.

For example, $w_1 = a\bar{a}\bar{a}a$ has a dip violation due to the prefix $u = a\bar{a}\bar{a}$; $w_2 = aa\bar{a}$ has an offset violation and $w_3 = aa\bar{a}\bar{b}$ has a mismatch violation.

This characterization of the language D_A was first observed by Ritchie and Springsteel [1972], who used it to show that membership in D_A can be decided in deterministic logspace. It is known as the "level trick" ([Lipton and Zalcstein 1977]) and has also been used by Tozawa and Minamide [2007b] in a polynomial-time algorithm to check inclusion of a given context-free language in D_A.

Algorithm 1 shows the outline of our procedure. Starting with the K-context-bounded language of an MPDA, we first decompose the language into a finite union of *shuffles* of context-free languages (Section 3.2); intuitively, each shuffle corresponds to one of (exponentially many) context switch sequences. Our algorithm guesses the specific context switch sequence (and so the shuffle) that leads to a violation.

Working directly with the shuffle is not enough, since we cannot efficiently compute dips. Therefore, our first step (Section 3.3) is to apply a procedure that expands the language of the shuffle into an *annotated shuffle* such that Dyck inclusion is maintained but dips are easy to compute.

The rest of the algorithm works with the annotated shuffle. We first check for offset violations in polynomial time (Section 3.4). If the check succeeds, we nondeterministically guess if there is a dip (Section 3.5) or a mismatch violation (Section 3.6), and we provide NP algorithms for each check.

Overall, the algorithm runs in nondeterministic polynomial time.

3.2 From MPDA Runs to Shuffles

Next, we define a class of languages, called *shuffles*, such that the context-bounded inclusion problem reduces to checking violations on shuffles.

Let $\Sigma = A \cup \bar{A}$. Let $\mathcal{M} = (Q, q_0, \Sigma, \Gamma, \delta, Q_F)$ be an m-MPDA. A *context-switch sequence* of length k is a sequence $\sigma = (q_0, t_1, q_1, \ldots, t_k, q_k)$ where $t_1, \ldots, t_k \in [1, m]$ are stack numbers, and $q_0, \ldots, q_k \in Q$ are states with $q_k \in Q_F$. The language $L_\sigma(\mathcal{P}) \subseteq \Sigma^*$ contains all words $u_1 \ldots u_k$ so that there exists an accepting run $c_0 \xrightarrow{u_1}_{t_1} c_1 \xrightarrow{u_2}_{t_2} \ldots \xrightarrow{u_k}_{t_k} c_k$ of $\mathcal{P}$ where configuration c_i is in state q_i for all $i \in [0, k]$. We can write $L_K(\mathcal{M}) = \bigcup_\sigma L_\sigma(\mathcal{M})$ where σ ranges over all exponentially many context-switch

Algorithm 1: Checking non-inclusion of $L_K(\mathcal{M})$ for an MPDA M and context bound K in the Dyck language in NP

MPDA $\mathcal{M}$ over the alphabet $\Sigma = A \cup \bar{A}$, K in unary
View $L_K(\mathcal{M})$ as a finite union of shuffles and guess a shuffle L (Section 3.2)
Turn L into an annotated shuffle L_{ann} with $L \subseteq D_A$ iff $L_{\text{ann}} \subseteq D_A$ (Proposition 3.1)
Check in polynomial time if L_{ann} has an offset violation (Lemma 3.2)
if L_{ann} *has offset violation* **then return** *"found violation"* ;
`/* Now we know that each context-free language in` L_{ann} `is offset-uniform */`
guess *violation type* **do**
  violation *dip* **do**
    Detect a dip violation in L_{ann} in NP (Lemma 3.5)
    if *dip violation found* **then return** *"found violation"*;
  violation *mismatch* **do**
    Detect a mismatch violation in L_{ann} in NP (Lemma 3.6)
    if *mismatch violation found* **then return** *"found violation"*;

sequences of length at most K. Each context-switch sequence is encoded by a string polynomial in $|\mathcal{M}|$ and in K.

Each language L_σ is a shuffle of m context-free languages $L_1, \ldots, L_m$, describing the behaviors of the m stacks. Such a context-free language L_t has the form $L_t \subseteq \Sigma^*\#\Sigma^*\# \cdots \#\Sigma^*$, see Fig. 3 for an illustration.

Let us introduce some notation for building words of the MPDA from the languages L_t. We will define this in a slightly more general setting, because our algorithms will work with languages over somewhat larger alphabets than Σ. Specifically, we assume that there is some alphabet $\Pi \supseteq \Sigma$ with the distinguished letter $\# \in \Pi \setminus \Sigma$. We call the letters in $\Pi \setminus \Sigma$ *auxiliary letters*. Moreover, we assume that the words in each language L_t contain the auxiliary letters *uniformly*, meaning there are $\$_1, \ldots, \$_k \in \Pi \setminus \Sigma$ with $L_t \subseteq \Sigma^*\$_1\Sigma^* \cdots \$_k\Sigma^*$. Here $\$_i$ and $\$_j$ for $i \neq j$ need not necessarily be different symbols. In particular, each word in L_t contains the same number of occurrences of #. We write $\Pi_{\setminus\#} := \Pi \setminus \{\#\}$.

For a word $w \in \Pi^*$, we write $w^{(1)}, \ldots, w^{(n)}$ for the factors obtained by cutting up w along the separator #. Thus, we have $w = w^{(1)}\#w^{(2)}\# \ldots \#w^{(n)}$ with $w^{(1)}, \ldots, w^{(n)} \in (\Pi_{\setminus\#})^*$. With the context-switch sequence $\sigma = (q_0, t_1, q_1, \ldots, t_k, q_k)$ we associate the *shuffle function*.

$$\text{shuf}_\sigma \colon (\Pi^*_{\setminus\#}\#)^{k_1-1}\Pi^*_{\setminus\#} \times \ldots \times (\Pi^*_{\setminus\#}\#)^{k_m-1}\Pi^*_{\setminus\#} \to \Pi^*_{\setminus\#}$$
$$\text{shuf}_\sigma(w_1, \ldots, w_m) = w_{t_1}^{(j_1)} \cdots w_{t_k}^{(j_k)}$$

where $k_t = |\{i \mid t_i = t\}|$ is the number of segments of stack t, and $j_i = |\{\ell \leq i \mid t_\ell = t_i\}|$ is the number of segments of stack t_i occurring before the (current) ith segment.

We extend this definition to languages. If $L_1, \ldots, _m \subseteq \Pi^*$ are languages with $L_t \subseteq (\Pi^*_{\setminus\#}\#)^{k_t-1}\Pi^*_{\setminus\#}$ for $1 \leq t \leq m$, then we define

$$\text{shuf}_\sigma(L_1, \ldots, L_m) = \{\text{shuf}_\sigma(w_1, \ldots, w_m) \mid w_t \in L_t \text{ for } 1 \leq t \leq m\}.$$

A *shuffle* is a language of the form $\text{shuf}_\sigma(L_1, \ldots, L_m)$ for a context-switch sequence σ and context-free languages $L_t \subseteq (\Pi^*_{\setminus\#}\#)^{k_t-1}\Pi^*_{\setminus\#}$. It is easy to see that every language $L_\sigma(\mathcal{M})$ is a shuffle.

Therefore, in order to show that $L_K(\mathcal{M}) \subseteq D_A$ can be checked in coNP, it suffices to show that given a shuffle $L \subseteq \Pi^*$, we can decide $L \subseteq D_A$ in coNP.

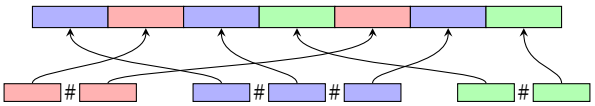


Fig. 3. An MPDA run consisting of seven segments which belong to three different stacks (blue, red, green). The behavior of each stack is described by a segmented context-free language.

3.3 Annotation

We begin with a preprocessing step. Let $\mathcal{P}$ be a pushdown automaton over the input alphabet Σ. We define the function $d\colon Q \times Q \to \mathbb{N} \cup \{\infty\}$ to yield the minimal drop of any word read between a state pair (from empty stack to empty stack), i.e.

$$d(p,q) = \inf\{d(w) \mid w \in \Sigma^*\colon (p,\varepsilon) \xrightarrow{w} (q,\varepsilon)\}.$$

If there is no $w \in \Sigma^*$ with $(p,\varepsilon) \xrightarrow{w} (q,\varepsilon)$, then we have $d(p,q) = \infty$.

An *annotated pushdown automaton* is a pushdown automaton $\mathcal{P}$ together with a table for the function d for $\mathcal{P}$. Specifically, if an algorithm takes an annotated pushdown automaton as an input, we simplify terminology and say it has an *annotated context-free language* as input.[7] Likewise, an *annotated shuffle* is a shuffle, where all the PDAs that describe the participating context-free languages are annotated.

Computing the function d for an arbitrary given PDA is polynomial-time inter-reducible with the *coverability problem for one-dimensional pushdown vector addition systems with states* [Leroux et al. 2015]. The best known complexity upper bound for this problem is EXPSPACE (according to [Englert et al. 2021]), and the problem is PSPACE-hard [Englert et al. 2021]. Therefore, in order to get an NP upper bound in our setting, we need a new idea. The first key insight of our algorithm is that we can slightly expand the input language so that (1) inclusion in D_A is preserved and (2) the function d can be computed for the expanded language in polynomial time. The following will be proven in Section 4.

Proposition 3.1 (Annotation). *Given a shuffle $L \subseteq \Sigma^*$, we can compute in polynomial time an annotated shuffle $L_{\mathsf{ann}} \subseteq \Sigma^*$ such that $L_{\mathsf{ann}} \subseteq D_A$ if and only if $L \subseteq D_A$.*

3.4 Checking Offset Violations in P

To check if L has an offset violation, we check if it is offset-uniform and, if so, check that all offsets are zero. We need a few preliminaries about straight line programs (SLPs) (see Section 4 for more details). Recall that an SLP is a CFG whose language consists of precisely one string. For an SLP $\mathbb{A}$, we write $\mathrm{eval}(\mathbb{A})$ for this string; note that $\mathrm{eval}(\mathbb{A})$ can be exponentially longer than the size of $\mathbb{A}$. Moreover, given a CFG $\mathcal{G}$, one can compute in polynomial time for each productive nonterminal X of $\mathcal{G}$ an SLP $\mathbb{A}_X$, such that one can derive the word $\mathrm{eval}(\mathbb{A}_X)$ from X in $\mathcal{G}$. It should be stressed that the latter procedure really computes an SLP $\mathbb{A}_X$ that encodes a *single arbitrary word* among those generated by X: Of course, X may generate infinitely many words, but for the purposes of our algorithm, it suffices to construct an SLP for one such word, and it does not matter which one.

[7] We can convert annotated PDAs to annotated CFGs, where an annotation maps nonterminals to $\mathbb{N} \cup \{\infty\}$. Let $\mathcal{P}$ be an annotated PDA for $L \subseteq \Sigma^*\#\Sigma^*$. Recall the standard conversion of $\mathcal{P}$ to a CFG [Sipser 2012]. The grammar has nonterminals of the form $X_{p,q}$ for each state pair p, q, and the following productions: (i) For states p, q, r add the production $X_{p,q} \to X_{p,r}X_{r,q}$. (ii) For all transitions $p \xrightarrow[\mathrm{push}(\gamma)]{a} p'$, $q' \xrightarrow[\mathrm{pop}(\gamma)]{b} q$ add the production $X_{p,q} \to aX_{p',q'}b$. Observe that $d(X_{p,q}) := d(p,q)$ is a valid *annotation* for the grammar, i.e. $d(X_{p,q}) = \inf\{d(w) \mid w \in \Sigma^*\colon (p,\varepsilon) \xrightarrow{w} (q,\varepsilon)\}$. Thus, we are justified in saying annotated context-free language.

Lemma 3.2 (Offset Violation in P). *Given a shuffle $L \subseteq \Sigma^*$, there is a polynomial-time procedure to check if $\Delta(w) = 0$ for all $w \in L$.*

Proof. Given a shuffle $L = \mathrm{shuf}_\sigma(L_1, \ldots, L_m)$, we test whether each L_i is offset-uniform, and, if so, compute the unique offset. If one of the languages L_i is not offset-uniform then L is not offset-uniform, and we can reject. Otherwise, we test whether the sum of all offsets is zero.

Hence, we only need to show a polynomial time procedure to test whether a context-free L is offset-uniform, and, if so, compute the offset. Suppose L is given by a context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$. Checking whether L is offset-uniform can be done using [Bertoni et al. 2011, Theorem 2] but we include an easy proof for the sake of completeness. For each productive nonterminals X of $\mathcal{G}$ let $\mathbb{A}_X$ be an SLP such that the word $\mathrm{eval}(\mathbb{A}_X)$ can be derived from X in $\mathcal{G}$. Such SLPs can be computed in polynomial-time by Lemma 4.3(2). We stress that here, all we need from $\mathbb{A}_X$ is that $\mathrm{eval}(\mathbb{A}_X)$ can be generated by X—it does not matter for the algorithm which word we pick. This is because in the next step, we compute the offset Δ_X of $\mathrm{eval}(\mathbb{A}_X)$ for each productive nonterminal $X \in N$: If L is offset-uniform, then all words derivable by X have the same offset. Thus, by computing Δ_X as the offset of $\mathrm{eval}(\mathbb{A}_X)$, we compute the offset that every word derivable from X should have, if L is offset-uniform. The final step is to verify this:

Observe that now, L is offset-uniform if and only if (i) for all $(X \to a) \in P$ we have $\Delta_X = \Delta(a)$ and (ii) for all $(X \to YZ) \in P$ we have $\Delta_X = \Delta_Y + \Delta_Z$. This can be verified easily by checking the equality for each production of $\mathcal{G}$. □

3.5 Checking Dip Violations in NP

The check for dip violations depends on the Parikh image of a shuffle and the representation of the dip as an existential Presburger formula. Recall that the *Parikh image* of a word $u \in \Sigma^*$ is a function $\mathrm{Parikh}(u) : \Sigma \to \mathbb{N}$ such that, for every $x \in \Sigma$, we have $\mathrm{Parikh}(u)(x) = |u|_x$, where $|u|_x$ denotes the number of occurrences of x in u. We extend the definition to the Parikh image of a language $L \subseteq \Sigma^*$: $\mathrm{Parikh}(L) = \{\mathrm{Parikh}(u) \mid u \in L\}$. We employ the usual isomorphism between $\Sigma \to \mathbb{N}$ and $\mathbb{N}^{|\Sigma|}$, corresponding to any fixed total ordering on the alphabet Σ, and consider the functions as vectors of natural numbers.

We shall use a result of Verma et al. [2005] that the Parikh image of a context-free language is representable in existential Presburger arithmetic.

Recall that Presburger arithmetic is the first order theory of the structure $(\mathbb{N}, +, \leq, 0, 1)$. The *existential* fragment of Presburger arithmetic (denoted ∃PA below) consists of existentially quantified formulas in prenex normal form. Satisfiability of ∃PA is NP-complete [Borosh and Treybig 1976].

We say that we can *compute an ∃PA formula in* NP *for a relation* $R \subseteq \mathbb{N}^k$ if there is a nondeterministic polynomial-time algorithm where each branch computes an ∃PA formula φ_i with k free variables such that if $\varphi_1, \ldots, \varphi_n$ are the formulas of all the branches, then

$$(n_1, \ldots, n_k) \in R \iff \bigvee_{i \in [1,n]} \varphi_i(n_1, \ldots, n_k).$$

Proposition 3.3 ([Verma et al. 2005]). *There is a polynomial time procedure that takes as input a context-free language L and produces an ∃PA formula $\Psi_L(x_1, \ldots, x_{|\Sigma|})$ such that $\mathbf{x} \in \mathrm{Parikh}(L)$ iff $\Psi_L(\mathbf{x}_1, \ldots, \mathbf{x}_{|\Sigma|})$.*

Moreover, we will tacitly use the well-known fact that given an SLP $\mathbb{A}$, one can compute the vector $\mathrm{Parikh}(\mathrm{eval}(\mathbb{A}))$ in polynomial time (with entries represented in binary) [Lohrey 2012].

We extend Proposition 3.3 to shuffles.

Lemma 3.4. *Given a shuffle $L \subseteq \Sigma^*$, one can compute in polynomial time an ∃PA formula for* $\mathrm{Parikh}(\mathrm{pref}(L))$.

Proof. If $L = \mathrm{shuf}_\sigma(L_1, \ldots, L_m)$, then $\mathrm{Parikh}(L) = \sum_{i=1}^m \mathrm{Parikh}(\pi_\Sigma(L_i))$ where π_Σ removes all occurrences of #. We can compute ∃PA formulas for each $\mathrm{Parikh}(\pi_\Sigma(L_i))$ in polynomial-time by Proposition 3.3 and combine them to get $\mathrm{Parikh}(L)$.

Next, we can write $\mathrm{pref}(L)$ as a union of k shuffles where k is the total number of segments in σ. The i-th shuffle contains those prefixes of words in L which end in the i-th segment. More precisely, suppose that $L = \mathrm{shuf}_\sigma(L_1, \ldots, L_m)$ where $\mathrm{shuf}_\sigma(w_1, \ldots, w_m) = w_{t_1}^{(j_1)} \ldots w_{t_k}^{(j_k)}$ for some numbers j_i, t_i. We can write $\mathrm{pref}(L) = P_1 \cup \cdots \cup P_k$ where

$$P_i = \{w_{t_1}^{(j_1)} \ldots w_{t_{i-1}}^{(j_{i-1})} v \mid w_1 \in L_1, \ldots, w_m \in L_m,\ v \text{ is a prefix of } w_{t_i}^{(j_i)}\}.$$

Observe that P_i is a shuffle of the languages $K_{i,1}, \ldots, K_{i,m}$, which are defined by

$$K_{i,t_i} = \{w_{t_i}^{(1)} \# \ldots \# w_{t_i}^{(j_i-1)} \# v \mid w_{t_i} \in L_{t_i},\ v \text{ is a prefix of } w_{t_i}^{(j_i)}\}$$

and

$$K_{i,t} = \{w_t^{(1)} \# \ldots \# w_t^{(k_t)} \mid w_t \in L_t\} \quad \text{where } k_t = |\{\ell < i \mid t_\ell = t\}|$$

for all $t \neq t_i$. It is easy to construct CFGs for these languages from the ones for L_1 to L_m. Computability of ∃PA formulas then again follows from Proposition 3.3. □

Lemma 3.5 (Dip Violation in NP). *Given a shuffle $L \subseteq \Sigma^*$, there is a nondeterministic polynomial-time procedure to check if there is a $w \in L$ and a prefix u of w with $d(u) < 0$.*

Proof. To verify whether L contains a dip violation, we construct the ∃PA formula ψ from Lemma 3.4 and use the NP satisfiability procedure of ∃PA to check whether there exists a vector $\boldsymbol{x} \in \mathbb{N}^\Sigma$ and a letter $a \in A$ with $\boldsymbol{x}[a] < \boldsymbol{x}[\bar{a}]$ satisfying ψ. □

In fact, a slightly more involved procedure shows dip violations can be checked in polynomial time. We omit this construction for simplicity.

3.6 Mismatch Violations in NP

It remains to detect mismatch violations in NP.

Lemma 3.6 (Mismatch Violations in NP). *Given an annotated, offset-uniform shuffle $L \subseteq \Sigma^*$, we can decide in* NP *whether L has a mismatch violation.*

The Marked One-counter Problem. We shall reduce detecting mismatch violations to the following combinatorial problem. A word $w \in (\Sigma \cup \{\#\})^*$ of the form $w = \#v\#$ with $e(v) = (0, 0)$ is a *marked one-counter word*. A factor that is a marked one-counter word is called a *marked one-counter factor*.

The *marked one-counter problem* is the following:

Given An offset-uniform, annotated shuffle $L \subseteq \Sigma^*\#\Sigma^*\#\Sigma^*$.
Question Does L contain a marked one-counter factor?

Proposition 3.7. *Given an offset-uniform annotated shuffle $L \subseteq \Sigma^*$, one can compute in polynomial time offset-uniform, annotated shuffles $L_1', \ldots, L_n' \subseteq \Sigma^*\#\Sigma^*\#\Sigma^*$ such that L contains a mismatch violation if and only if some L_i' contains a marked one-counter factor.*

The intuition behind the reduction is as follows. Let $L = \mathrm{shuf}_\sigma(L_1, \ldots, L_m)$ for a context-switch sequence $\sigma(q_0, t_1, q_1, \ldots, t_k, q_k)$ and context-free languages $L_l \subseteq (\Sigma^*\#)^{k_l - 1}\Sigma^*$ with $\sum_{l=1}^m k_l = k$ being the total number of segments.

For the reduction, we guess the location of the letters $a, \bar{b}$ such that $aw'\bar{b}$ is the infix witnessing the mismatch violation. Since the a and $\bar{b}$ could potentially be located in segments belonging to different stacks, we also need to guess which segment contains each of these two letters a priori, as well as the identity of these letters. Thus we have a polynomial (quadratic in the number of segments

k and quadratic in $|A|$) number of languages $L'_{i,j,a,\bar{b}}$ where $i, j \in [0,k], i \leq j, a \in A, b \in A \setminus \{a\}$, such that a (resp. $\bar{b}$) is located in segment i (resp. j). To construct these shuffles from L, we modify the PDA for the language that provides the ith (resp. jth) segment of the shuffle, such that it reads a #-symbol instead of exactly one a (resp. $\bar{b}$) in the input at the correct position.

Given Proposition 3.7, it remains to solve the marked one-counter problem for shuffles of offset-uniform annotated context-free languages.

3.7 Reachability Relations

We will need some more notation. For a word $w \in \Sigma^*$, we define $\rho(w) \subseteq \mathbb{N} \times \mathbb{N}$ to be the set of all $(n,m) \in \mathbb{N} \times \mathbb{N}$ such that $n \geq d(w)$ and $m = n + \Delta(w)$. In other words, $\rho(w)$ is the *reachability relation induced by w, interpreted as counter instructions*: We have $(n,m) \in \rho(w)$ if and only if interpreting w as a sequence of counter instructions (a letter $a \in A$ signifying an increment and $\bar{a} \in \bar{A}$ signifying decrement) leads from counter value n to counter value m, while staying above zero. In particular, we have $e(w) = (0,0)$ if and only if $(0,0) \in \rho(w)$.

Consider again the marked one-counter problem. We are given a shuffle $L \subseteq \Sigma^*\#\Sigma^*\#\Sigma^*$ and we want to decide, in NP, whether there exists a word $u\#v\#w \in L$ with $(0,0) \in \rho(v)$. Let $F = \{v \in \Sigma^* \mid \exists u, w \in \Sigma^* \colon u\#v\#w \in L\}$ be the set of those factors in between #. So, the marked one-counter problem can be solved in NP if we can check in NP if $(0,0) \in \rho(w)$ for some $w \in F$.

Let $L = \mathrm{shuf}_\sigma(L_1, \ldots, L_m)$ for some offset-uniform, annotated context-free languages $L_1, \ldots, L_m$. We will need a slight modification of the notation $w^{(i)}$ for $w \in \Gamma^*$: While $w^{(i)}$ means we split up w along the separator # and then take the i-th factor, we want $w^{\langle i \rangle}$ to mean that we split up w at *both* auxiliary letters, # and #, and then take the i-th factor. Thus, if $w = abbb\#\bar{b}\bar{b}\bar{b}c\#\bar{c}abc\#\bar{c}\bar{b}\bar{a}\bar{a}$, then we obtain the two decompositions

$$w = \overbrace{\underbrace{abbb}_{w^{\langle 1\rangle}}\#\underbrace{\bar{b}\bar{b}\bar{b}c}_{w^{\langle 2\rangle}}}^{w^{(1)}}\#\overbrace{\underbrace{\bar{c}abc}_{w^{\langle 3\rangle}}\#\underbrace{\bar{c}\bar{b}\bar{a}\bar{a}}_{w^{\langle 4\rangle}}}^{w^{(2)}}\#\overbrace{\phantom{\underbrace{\bar{c}abc}_{w^{\langle 3\rangle}}\#\underbrace{\bar{c}\bar{b}\bar{a}\bar{a}}_{w^{\langle 4\rangle}}}}^{w^{(3)}}.$$

The set of possible factors v is obtained by concatenating certain segments of the participating languages $L_1, \ldots, L_m$. This means, by inspection of the context-switch sequence σ, we can find indices $t_1, \ldots, t_r$ and $j_1, \ldots, j_r$ such that

$$F = \{w_{t_1}^{\langle j_1 \rangle} \cdots w_{t_r}^{\langle j_r \rangle} \mid w_{t_i} \in L_{t_i} \text{ for } i \in [1,r]\}.$$

Thus, we need to check if there is some $w \in F$ such that $(0,0) \in \rho(w)$ in NP. The following proposition is the key technical step in the computation.

Proposition 3.8. *Given an offset-uniform, annotated context-free language $L \subseteq (\Sigma^*\#)^k\Sigma^*$, we can compute in* NP *an $\exists$PA formula for the relation*

$$\bigcup_{w \in L} \rho(w^{(1)}) \times \cdots \times \rho(w^{(k+1)}) \subseteq \mathbb{N}^{2(k+1)}.$$

Assuming offset-uniformity and annotations in Proposition 3.8 is crucial. Without these assumptions, we could use the proposition to test whether a context-free language over $\{a, \bar{a}\}$ intersects the Dyck-language $D_{\{a\}}$. This is equivalent to the *reachability problem* for one-dimensional pushdown vector addition systems with states, which is not known to be decidable [Leroux et al. 2015].

Proposition 3.8 is sufficient to compute $\exists$PA formulas for each portion of a word in F. In the proof of Lemma 3.6, we will apply Proposition 3.8 and then combine these formulas using the following observation, which directly follows from the definition of ρ.

Lemma 3.9. *Let $u_1, \dots, u_s \in \Sigma^*$. Then $(n, m) \in \rho(u_1 \cdots u_s)$ if and only if there are $n_1, m_1, \dots, n_s, m_s \in \mathbb{N}$ with $(n_i, m_i) \in \rho(u_i)$ for every $i \in [1, s]$ and $n = n_1$, $m_1 = n_2, \dots, m_{s-1} = n_s$, and $m_s = m$.*

Before we prove Proposition 3.8, let us see how it can be used to check $(0, 0) \in \rho(F)$ in NP, and thus to prove Lemma 3.6.

Proof of Lemma 3.6. By Proposition 3.8, we can in NP compute an ∃PA formula φ_t for $t \in [1, m]$ with free variables $n_{t,1}, m_{t,1}, \dots, n_{t,k}, m_{t,k}$ such that

$$\varphi_t(n_{t,1}, m_{t,1}, \dots, n_{t,k}, m_{t,k}) \iff \exists w \in L_t \colon (n_{t,j}, m_{t,j}) \in \rho(w^{\langle j \rangle}) \text{ for all } j \in [1, k].$$

Hence, by Lemma 3.9, we have $(0, 0) \in \rho(F)$ if and only if

$$\bigwedge_{t=1}^{m} \varphi_t \quad \wedge \quad 0 = n_{t_1,j_1} \wedge m_{t_1,j_1} = n_{t_2,j_2} \wedge \cdots \wedge m_{t_{r-1},j_{r-1}} = n_{t_r,j_r} \wedge m_{t_r,j_r} = 0. \tag{1}$$

Finally, note that our application of Proposition 3.8 means that each φ_t is a disjunction of (exponentially many) ∃PA formulas and we can only guess one of these disjuncts. Therefore, we guess, for each $t \in [1, m]$, a disjunct of φ_t, and then replace φ_t in Eq. (1) by this disjunct. Then clearly, one of the resulting formulas is satisfiable if and only if $(0, 0) \in \rho(F)$. Together with the fact that satisfiability of ∃PA is in NP, this allows us to decide $(0, 0) \in \rho(F)$ in NP and so to solve the marked one-counter problem. By Proposition 3.7, this is sufficient to also prove that checking mismatch violations is in NP (Lemma 3.6). □

It remains to prove Proposition 3.8. We accomplish this in two steps. The first step is the special case $k = 1$, i.e., $L \subseteq \Sigma^*\#\Sigma^*$. For this special case, we will employ two-dimensional vector addition systems. The second step will lift the construction to all k using *cancellation graphs*.

3.8 Proposition 3.8: Special Case $k = 1$

Vector addition systems with states. A (*d-dimensional*) *vector addition system with states* (short *d-VASS*) is a pair (Q, T), where Q is a finite set of *states*, $T \subseteq Q \times \mathbb{Z}^d \times Q$ is its set of *transitions*. When we represent a d-VASS in memory, we will store the numbers in transitions *in binary*. A *configuration* is an element of $Q \times \mathbb{N}^d$ and instead of $(q, \mathbf{x})$, we also write $q(\mathbf{x})$. For two configurations $p(\mathbf{x})$ and $q(\mathbf{y})$, we write $p(\mathbf{x}) \to q(\mathbf{y})$ if there exists a transition $(p, \mathbf{z}, q) \in T$ such that $\mathbf{y} = \mathbf{x} + \mathbf{z}$. By $\xrightarrow{*}$, we denote the reflexive, transitive closure of $\to$.

Hence, computation steps in VASS can only happen if all counter values remain non-negative. However, sometimes, it will be useful to talk about hypothetical steps between so-called "pseudo-configurations", which may contain negative numbers. A *pseudo-configuration* is a pair in $Q \times \mathbb{Z}^d$. For pseudo-configurations $p(\mathbf{x}), q(\mathbf{y}) \in Q \times \mathbb{Z}^d$, we write $p(\mathbf{x}) \to_{\mathbb{Z}} q(\mathbf{y})$ if there exists a transition $(p, \mathbf{z}, q) \in T$ with $\mathbf{y} = \mathbf{x} + \mathbf{z}$. Again, $\xrightarrow{*}_{\mathbb{Z}}$ is the reflexive, transitive closure of $\to_{\mathbb{Z}}$.

A 2-VASS is called *offset-uniform* if for any two states p, q and any runs $p(n_1, n_2) \xrightarrow{*}_{\mathbb{Z}} q(m_1, m_2)$ and $p(n_1', n_2') \xrightarrow{*}_{\mathbb{Z}} q(m_1', m_2')$, we have

$$(m_2 - m_1) - (n_2 - n_1) = (m_2' - m_1') - (n_2' - n_1').$$

Proposition 3.10. *Given an offset-uniform 2-VASS and states p, q, we can compute in* NP *an ∃PA formula for the reachability relation*

$$\{(n_1, n_2, m_1, m_2) \in \mathbb{N}^4 \mid p(n_1, n_2) \xrightarrow{*} q(m_1, m_2)\}. \tag{2}$$

We will prove Proposition 3.10 in Section 5. Now Proposition 3.10 allows us to prove Proposition 3.8 in the special case $k = 1$:

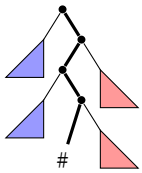


Fig. 4. Illustration of Proposition 3.11

Proposition 3.11. *Given an offset-uniform, annotated context-free language $L \subseteq \Sigma^*\#\Sigma^*$, we can compute in* NP *an* $\exists$*PA formula for the relation*

$$\bigcup_{w \in L} \rho(w^{(1)}) \times \rho(w^{(2)}) \subseteq \mathbb{N}^4.$$

Proof. Let $\mathcal{G} = (N, \Sigma, P, S)$ be an annotated grammar for $L \subseteq \Sigma^*\#\Sigma^*$, after establishing Chomsky normal form. We compute for every nonterminal X the offset $\Delta(X)$ in polynomial time, and we define $\Delta_-(X) = d(X)$ and $\Delta_+(X) = \Delta(X) + d(X)$. Let us describe intuitively how the 2-VASS works. Consider a derivation tree of $\mathcal{G}$ for a word $w^{(1)}\#w^{(2)}$ and consider the path from the root to the leaf labeled by the separator #, as shown in Fig. 4. The 2-VASS traverses this path from the root to the #-leaf. While doing so, it applies the effects of the subtrees to the sides of the path (shown in blue and red in Fig. 4) to its counters. More precisely, it either adds the effect of a left-branching nonterminal (blue in Fig. 4) to the first counter, or subtracts the effect of a right-branching nonterminal (red in Fig. 4) from the second counter. Note that although the offset of a nonterminal X is unique, namely $\Delta(X)$, its *dip* is not uniquely determined. The 2-VASS will always assume the minimal dip $d(X)$, which is enough to capture the union of the reachability relations $\rho(w^{(1)})$ and $\rho(w^{(2)})$ when w ranges over all words in L.

More formally, let $N_\# \subseteq N$ be the set of nonterminals with $L(N_\#) \subseteq \Sigma^*\#\Sigma^*$. The 2-VASS has state set $N_\# \cup \{\#\}$ (and some auxiliary states) such that for all $X \in N_\#$ we have

$$\bigcup_{w \in L(X)} \rho(w^{(1)}) \times \rho(w^{(2)}) = \{(n_1, m_1, m_2, n_2) \mid X(n_1, n_2) \xrightarrow{*} \#(m_1, m_2)\}.$$

Note that here, the entries of $\rho(w^{(1)}) \times \rho(w^{(2)})$ require m_1 and m_2 to be in the middle, because the 2-VASS simulates the word $w^{(2)}$ from right to left. Then the statement follows from Proposition 3.10. The 2-VASS contains the following transitions

$$X \xrightarrow{(-\Delta_-(Y),0)} \bullet \xrightarrow{(\Delta_+(Y),0)} Z \qquad \text{for each } X \to YZ \text{ with } Z \in N_\#$$

$$X \xrightarrow{(0,-\Delta_+(Z))} \bullet \xrightarrow{(0,\Delta_-(Z))} Y \qquad \text{for each } X \to YZ \text{ with } Y \in N_\#$$

$$X \xrightarrow{(0,0)} \# \qquad \text{for all } X \to \#.$$

Here, $\bullet$ represents a fresh state introduced for each production. Offset-uniformity is inherited from $\mathcal{G}$. Correctness can be verified by induction. □

3.9 Proposition 3.8: General Case

It remains to prove Proposition 3.8. This means, we need to lift Proposition 3.11, i.e. the case of $L \subseteq \Sigma^*\#\Sigma^*$, to the more general case of $L \subseteq (\Sigma^*\#)^k\Sigma^*$. Having established Proposition 3.11, the general case is conceptually simple and follows.

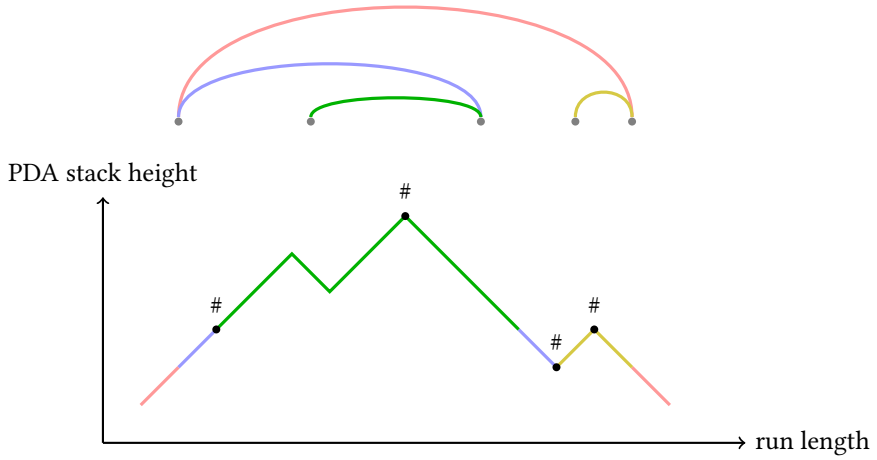


Fig. 5. Decomposing a segmented PDA run into 2-segmented runs (bottom) via cancellation graph (top).

Decomposing runs. The key idea is that each run of our PDA for $L \subseteq (\Sigma^*\#)^k\Sigma^*$ reading a word $u_0\#u_1 \cdots \#u_k$ can be decomposed into pairs (π_1, π_2) of sub-runs π_1 and π_2 such that

(1) each sub-run π_1 and π_2 reads an infix of some u_i, $i \in [0, k]$, and
(2) π_1 produces a stack content w which is completely consumed by π_2. Here, we refer to the stack of the PDA itself, not the input letters (which can also be seen as stack operations).

This is illustrated in Fig. 5 (bottom). We decompose as follows. We go from left to right through the run for $u_0\#u_1 \cdots \#u_k$, beginning at u_0. We pick the longest prefix π_1 of the run on u_0 so that there is an $i \in [0, k]$ such that each push in π_1 has its corresponding pop either (i) still within the run on u_0 or (ii) in the run on the segment u_i. In Fig. 5, the first pair (π_1, π_2) is the one in red, and we have $i = 4$, because the red sub-run on the left cancels with the red sub-run on the right.

One can then notice that π_1 corresponds to a sub-run π_2 of the run on u_i so that all pushes within π_1 that are not popped within the run on u_0, are popped in π_2. Now we have found our first pair of sub-runs. To find the next pair, we continue along u_0 until we encounter a push that is popped outside of u_0 and u_i. In Fig. 5, this would be the blue pair, which has pushes within u_0 matched by pops in u_2. This then yields a new pair (π_1', π_2'), etc. At some point we will also finish decomposing the run on u_0, at which point we move on to u_1 and so on. In Fig. 5, each pair (π_1, π_2) corresponds to one edge in the graph above the PDA run.

As in Fig. 5 (top), the pairs (π_1, π_2) are edges in a graph, which we call "cancellation graph". Slightly more formally, a *cancellation graph* is a graph $G = (V, E, \lambda)$, where $V = [0, k]$, $E \subseteq V \times V$ is a set of edges with $i < j$ for each $(i, j) \in E$, and λ is a labeling function that maps nodes and edges to the states occupied by the PDA at the beginning and end at each subrun.

Each cancellation graph requires polynomially many bits to store. Therefore, the algorithm for Proposition 3.8 can guess a cancellation graph and then computes in NP an $\exists$PA formula for the ρ relation covering all runs that decompose using this cancellation graph. To this end, it applies Proposition 3.8 to each pair (π_1, π_2) and then combines the formulas using Lemma 3.9.

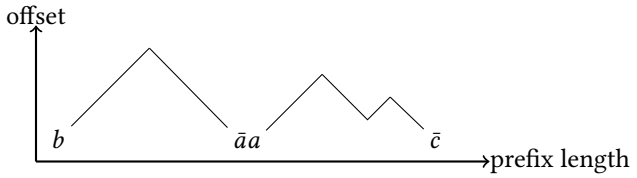


Fig. 6. Illustration of Lemma 4.1

4 COMPUTING ANNOTATIONS

In this section, we prove Proposition 3.1. As mentioned before, the language L_{ann} is a superset of the input language L. We will argue that $L_{\text{ann}} \subseteq D_A$ if and only if $L \subseteq D_A$ by observing that all words in L_{ann} are obtained from words in L by *free group reductions*. Let us give some context on this. Recall that $\Sigma = A \cup \bar{A}$. It is a classical fact (and follows easily from the definition of D_A) that for $u, v \in \Sigma^*$, we have $uv \in D_A$ if and only if $ua\bar{a}v \in D_A$. If we flip a and $\bar{a}$, the same is not true—inserting $\bar{a}a$ can spoil membership in D_A: We have $\varepsilon \in D_A$, but $\bar{a}a \notin D_A$.

In fact, allowing $\bar{a}a$ to be deleted in addition to $a\bar{a}$ leads to *free group reductions*: For $w, w' \in \Gamma^*$, we write $w \rightarrowtail w'$ if there are $u, v \in \Gamma^*$ and $a \in A$ with $w' = uv$ and either (i) $w = ua\bar{a}v$ or (ii) $w = u\bar{a}av$. Moreover, $\rightarrowtail^*$ denotes the reflexive transitive closure of $\rightarrowtail$. Then, the language of all $w \in \Sigma^*$ with $w \rightarrowtail^* \varepsilon$ is called the *word problem of the free group (over A)* and checking inclusion there is often algorithmically easier than inclusion in D_A.

We observe here, somewhat surprisingly, that applying free group reductions preserves membership in D_A, *in one direction*:

Lemma 4.1. *Let $u, v \in \Sigma^*$ and $a \in A$. If $u\bar{a}av \in D_A$, then also $uv \in D_A$.*

Proof. For each type of violation, we can check that if uv has a violation, then $u\bar{a}av$ has one as well. For dip and offset violations, this is obvious. Finally, suppose uv has a mismatch violation, say with a factor $bw\bar{c}$, where $e(w) = (0, 0)$ and $b \neq c$. If both b and $\bar{c}$ are in the same factor (u or v) of uv, then this mismatch violation also exists in $u\bar{a}av$. Otherwise, b occurs in u, $\bar{c}$ occurs in v, and $\bar{a}a$ is inserted inside of w. Let w' be the word obtained by inserting $\bar{a}a$ into w.

If $e(w') = (0, 0)$, then the same b and $\bar{c}$ constitute a mismatch violation for $u\bar{a}av$. Otherwise, we must have $d(w') = 1$, because the offsets in w can only sink by at most 1 if we merely insert $\bar{a}a$. But then we are in a situation as in Fig. 6: all four letters, b, $\bar{a}$, a, and $\bar{c}$, are on the same level. Since $b \neq c$, we must have $a \neq b$ or $a \neq c$ (otherwise, $b = a = c$). Therefore, there must be a mismatch violation in $u\bar{a}av$ between b and $\bar{a}$ or between a and $\bar{c}$. □

Lemma 4.1 motivates the following definition. For a language $L \subseteq \Gamma^*$, we define

$$L^\diamond := \{w' \in \Gamma^* \mid \exists w \in L \colon w \rightarrowtail^* w'\}.$$

Lemma 4.1 tells us that applying free group reductions preserves membership in D_A:

Corollary 4.2. *For $L \subseteq \Sigma^*$, we have $L \subseteq D_A$ if and only if $L^\diamond \subseteq D_A$.*

Proof. The "if" follows from $L \subseteq L^\diamond$ and the "only if" follows by induction from Lemma 4.1 and from the fact that for any $u, v \in \Sigma^*$ and $a \in A$, we have $ua\bar{a}v \in D_A$ if and only if $uv \in D_A$. □

Corollary 4.2 will let us argue that $L \subseteq L_{\text{ann}} \subseteq L^\diamond$, which implies $L_{\text{ann}} \subseteq D_A$ if and only if $L \subseteq D_A$.

Compressed Words and Straight Line Programs. Our annotation procedure will also employ known algorithms on compressed words, represented by context-free grammars accepting exactly one word. A *straight-line program (SLP)* is a context-free grammar $\mathbb{A}$ where every nonterminal X

produces exactly one word, denoted by $\mathsf{eval}(X)$. The word produced by the starting nonterminal is denoted by $\mathsf{eval}(\mathbb{A})$. In the following we summarize a few results that we will need. For more details, we refer to the survey on algorithms on SLP-compressed words by Lohrey [Lohrey 2012]. We stress that in (2), the mentioned algorithm computes an SLP for a *single arbitrary* word derivable from X (among a potentially infinite set). We employ this fact in algorithms where it does not matter which word is picked.

Lemma 4.3. *(1) Given two SLPs $\mathbb{A}$ and $\mathbb{A}'$, there is a polynomial-time procedure to check if* $\mathsf{eval}(\mathbb{A}) = \mathsf{eval}(\mathbb{A}')$.

(2) Given a context-free grammar $\mathcal{G}$, we can compute in polynomial time for each productive nonterminal X of $\mathcal{G}$ an SLP $\mathbb{A}_X$ such that $\mathsf{eval}(\mathbb{A}_X)$ *is one of the words that can be derived from X in $\mathcal{G}$. Formally,* $\mathsf{eval}(\mathbb{A}_X) \in L_{\mathcal{G}}(X)$.

(3) Given an SLP $\mathbb{A}$ for a word $a_1 \dots a_n$ and two positions $1 \le i \le j \le n$, one can compute an SLP for the factor $a_i \dots a_j$ in polynomial time.

(4) Given SLPs $\mathbb{A}_1, \mathbb{A}_2$, one can compute in polynomial time an SLP $\mathbb{A}$ such that $\mathsf{eval}(\mathbb{A})$ *is the longest common prefix of* $\mathsf{eval}(\mathbb{A}_1)$ *and* $\mathsf{eval}(\mathbb{A}_2)$.

(5) Given an SLP $\mathbb{A}$ with $|\mathsf{eval}(\mathbb{A})|_{\#} = k$ for some k given in unary, we can compute in polynomial time SLPs $\mathbb{A}_0, \dots, \mathbb{A}_k$ such that $\mathsf{eval}(\mathbb{A}) = \mathsf{eval}(\mathbb{A}_0)\#\mathsf{eval}(\mathbb{A}_1) \cdots \#\mathsf{eval}(\mathbb{A}_k)$.

(6) Given an SLP $\mathbb{A}$ with $\mathsf{eval}(\mathbb{A}) \in \bar{A}^* A^*$, *we can compute in polynomial time SLPs $\mathbb{A}_1, \mathbb{A}_2$ such that* $\mathsf{eval}(\mathbb{A}) = \overline{\mathsf{eval}(\mathbb{A}_1)}\mathsf{eval}(\mathbb{A}_2)$.

Proof. Statement (1) is a well-known result for SLPs [Plandowski 1994]. For statement (2) we compute all productive nonterminals using a propagation algorithm, similar to solving satisfiability for Horn formulas. Every time a new productive nonterminal is identified, we additionally pick a witnessing production and include it in the SLP. Statement (3) can be found in [Schleimer 2008, Theorem 2.9], in more generality. To compute the longest common prefix (statement (4)), we do a binary search over the set of prefixes with the help of statement (3) and the fact that equality of two SLPs can be tested in polynomial time (statement (1)).

For statement (5), for every nonterminal X in $\mathbb{A}$ we compute the set of positions P_X of the #-symbols in $\mathsf{eval}(X)$. Assuming Chomsky normal form, this can be done in polynomial time by a bottom-up computation. For a rule $X \to \#$ we set $P_X = \{1\}$ and for a rule $X \to a$ with $a \neq \#$ we set $P_X = \emptyset$. For a rule $X \to YZ$ we set $P_X = P_Y \cup (P_Z + |\mathsf{eval}(Y)|)$. With the positions for the starting nonterminal in hand, we can decompose $\mathbb{A}$ into $k+1$ SLPs using statement (4).

For statement (6), again, we perform a bottom-up computation. We compute for every nonterminal X whether $\mathsf{eval}(X)$ contains symbols from A, symbols from $\bar{A}$, and if it contains both, we compute the position of the rightmost $\bar{A}$-symbol. Finally, we split the SLP along the position computed for the starting nonterminal using statement (4). □

Lemma 4.4. *Given an SLP $\mathbb{A}$, one can compute an annotated PDA for* $\{\mathsf{eval}(\mathbb{A})\}$ *in polynomial time.*

Proof. We can assume that $\mathbb{A} = (N, \Sigma, P, S)$ is in Chomsky normal form. For any nonterminal X in $\mathbb{A}$, let Δ_X and d_X denote $\Delta(\mathsf{eval}(X))$ and $d(\mathsf{eval}(X))$. These values can be computed in polynomial time for all nonterminals using the following equations:

$$\begin{aligned} &\Delta_X = \Delta(a), && d_X = d(a) && \text{for all productions } X \to a \\ &\Delta_X = \Delta_Y + \Delta_Z, && d_X = \max\{d_Y, d_Z - \Delta_Y\} && \text{for all productions } X \to YZ. \end{aligned}$$

In particular we can compute $d(\mathsf{eval}(\mathbb{A})) = d_S$. Let $P' \subseteq P$ be the set of all productions of $\mathbb{A}$ of the form $X \to YZ$. The PDA $\mathcal{P}$ has states $Q = \{r, q\} \cup \{q_{p,1}, q_{p,2} \mid p \in P'\}$ and its stack alphabet is the set N of nonterminals of $\mathbb{A}$. Furthermore, its transitions are

$$r \xrightarrow[\mathsf{push}(S)]{\varepsilon} q$$

$$q \xrightarrow[\mathsf{pop}(X)]{a} q \qquad \text{for all productions } X \to a$$

$$q \xrightarrow[\mathsf{pop}(X)]{\varepsilon} q_{p,1} \xrightarrow[\mathsf{push}(Z)]{\varepsilon} q_{p,2} \xrightarrow[\mathsf{push}(Y)]{\varepsilon} q \qquad \text{for all productions } p = X \to YZ.$$

It remains to construct a valid annotation d for $\mathcal{P}$. To this end we also construct an auxiliary function $\Delta : Q \times Q \to \mathbb{Z} \cup \{\infty\}$, which we will discard at the end. We begin by setting some initial values based on matching push- and pop-transitions:

- $d(r, q) = d(\mathsf{eval}(\mathbb{A}))$, $\Delta(r, q) = \Delta(\mathsf{eval}(\mathbb{A}))$,
- if there is a production $p_S = S \to YZ \in P'$ then $d(r, q_{p_S,1}) = 0 = \Delta(r, q_{p_S,1})$,
- for all $p = X \to YZ \in P'$: $d(q_{p,1}, q) = d_X$, $d(q_{p,2}, q) = d_Y$, $\Delta(q_{p,1}, q) = \Delta_X$, $\Delta(q_{p,2}, q) = \Delta_Y$,
  - if there is a production $p_Y = Y \to Y'Z' \in P'$ then $d(q_{p,1}, q_{p_Y,1}) = d_Z$, $\Delta(q_{p,1}, q_{p_Y,1}) = \Delta_Z$,
  - if there is a production $p_Z = Z \to Y'Z' \in P'$ then $d(q_{p,2}, q_{p_Z,1}) = 0 = \Delta(q_{p,2}, q_{p_Z,1})$,
- d and Δ initially set to ∞ on all other inputs.

From here we iteratively compute further values similarly to the equations above. Let $p', q', r' \in Q$ be states with $d(p', q') \neq \infty$, $\Delta(p', q') \neq \infty$, $d(q', r') \neq \infty$, $\Delta(q', r') \neq \infty$, and $d(p', r') = \infty = \Delta(p', r')$. Then we set $d(p', r') = \max\{d(p', q'), d(q', r') - \Delta(p', q')\}$, and $\Delta(p', r') = \Delta(p', q') + \Delta(q', r')$. It suffices to compute these values only once, since for states $p', q' \in Q$ there is a unique word $w \in \Sigma^*$ with $(p', \varepsilon) \xrightarrow{w} (q', \varepsilon)$. This is because $\mathcal{P}$ was constructed to simulate an SLP. □

Lemma 4.5. *Suppose $L \subseteq \Sigma^*$ and there are $x, y \in \Sigma^*$ such that $xLy \subseteq D_A$. Moreover, suppose $\bar{u}\bar{t}tv \in \mathsf{DyckNF}(L)$ such that $u, v \in A^*$ and t is the longest common prefix of tu and tv. Then for every $w \in L$, we have $d(w) \geq |u|$.*

Proof. Since $x\bar{u}\bar{t}tvy \in D_A$ we can conclude $\mathsf{DyckNF}(x) = eu$ and $\mathsf{DyckNF}(y) = \bar{v}\bar{f}$ for some $e, f \in A^*$. Now e and f must have the same length because $\Delta(e\bar{f}) = \Delta(eu\bar{u}\bar{t}tv\bar{v}\bar{f}) = 0$. Suppose there is a word $w \in L$ with $d(w) < |u|$. We claim that $uw\bar{v} \in D_A$: Since $uw\bar{v}$ is a factor of $\mathsf{DyckNF}(x)w\mathsf{DyckNF}(y) = euw\bar{v}\bar{f} \in D_A$ it contains no mismatched letters. Furthermore $\Delta(uw\bar{v}) = 0$ since $\Delta(euw\bar{v}\bar{f}) = 0$ and $\Delta(e\bar{f}) = 0$, and $d(uw\bar{v}) = 0$ since $d(w) < |u|$, proving the claim.

Since $d(w) < |u|$ the normal form $\mathsf{DyckNF}(uw)$ has a prefix which is a proper prefix u_1 of u. Since $uw\bar{v} \in D_A$, u_1 must also be a prefix of v. This contradicts the assumption that t is the longest common prefix of tu and tv. □

For a PDA $\mathcal{P}$ over $\Sigma \cup \{\#\}$ and states p, q let $L_{p,q}(\mathcal{P})$ be the set of all words $w \in \Sigma^*$ with $(p, \varepsilon) \xrightarrow{w} (q, \varepsilon)$. A state q of $\mathcal{P}$ is *productive* if it occurs on some accepting run of $\mathcal{P}$.

Lemma 4.6. *Given a shuffle $L \subseteq \Sigma^*$ and productive states p, q in a PDA $\mathcal{P}$ for L, one can compute in polynomial time SLPs for words $x, y \in \Sigma^*$ such that $xL_{p,q}(\mathcal{P})y \subseteq L$.*

Proof. Let $\mathcal{P}_\$$ be a PDA which accepts all words $x\$y$ such that $\mathcal{P}$ has runs of the form $(q_0, \varepsilon) \xrightarrow{x} (p, s)$, $(p, \varepsilon) \xrightarrow{z} (q, \varepsilon)$, and $(q, s) \xrightarrow{y} (q_f, \varepsilon)$ where q_f is a final state, $s \in \Gamma^*$, $x, y \in (\Sigma \cup \{\#\})^*$, and $z \in \Sigma^*$. Such a PDA $\mathcal{P}_\$$ works by guessing a subrun between the states p and q, in which it reads a single \$-symbol instead of the input symbols. Furthermore, it can make sure that the subrun corresponds to a run $(p, \varepsilon) \xrightarrow{z} (q, \varepsilon)$ by initially pushing a special marker and finally popping it.

Let $L = \mathrm{shuf}_\sigma(L_1, \ldots, L_m)$, and let $L_\$$ be the modified shuffle where the PDA $\mathcal{P}$ is replaced by $\mathcal{P}_\$$. Then each word $x\$y \in L_\$$ satisfies $xL_{p,q}(\mathcal{P})y \subseteq L$. To compute SLPs for such words x, y, we compute for each context-free language L_i an SLP $\mathbb{A}_i$ with $w_i := \mathrm{eval}(\mathbb{A}_i) \in L_i$. Then $\mathrm{shuf}_\sigma(w_1, \ldots, w_m)$ is of the form $x\$y$. Using Lemma 4.3(5) we can compute SLPs for the segments of each w_i. By concatenating these SLPs in the proper order we obtain an SLP for $x\$y$. Finally, we obtain SLPs for x and y by again applying Lemma 4.3(5). □

We need a final ingredient: computing the Dyck normal form of a compressed word in polynomial time. This is a key step in [Tozawa and Minamide 2007a].

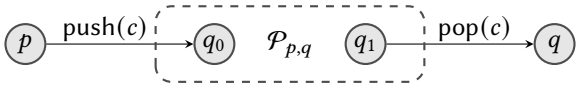
Lemma 4.7 ([Tozawa and Minamide 2007a, Proposition 3(iii)]). *Given an SLP $\mathbb{A}$ with $\mathrm{eval}(\mathbb{A}) \in \Sigma^*$, we can compute in polynomial time an SLP $\mathbb{A}'$ with $\mathrm{eval}(\mathbb{A}') = \mathsf{DyckNF}(\mathrm{eval}(\mathbb{A}))$.*

We are now ready to prove Proposition 3.1.

Proof. As a first step, for each participating PDA $\mathcal{P}$ and any states p, q in $\mathcal{P}$, we use Lemma 4.6 to compute in polynomial time words $x_{p,q}$ and $y_{p,q}$ as SLPs such that the language $K_{p,q} := x_{p,q}L_{p,q}(\mathcal{P})y_{p,q}$ is a subset of L. Since $K_{p,q}$ is a context-free language, for which we can easily compute a PDA, we can use Theorem 2.4 to check $K_{p,q} \subseteq D_A$ in polynomial time for each pair p, q. If for some states p, q, this inclusion does not hold, then clearly $L \not\subseteq D_A$ and our algorithm can just return the shuffle $\{\bar{a}\}$ for some $a \in A$, for which we can clearly compute an annotated shuffle.

Thus, suppose $K_{p,q} \subseteq D_A$ for every p, q. Now for each p, q, we check in polynomial time whether $L_{p,q}(\mathcal{P})$ is empty. If it is, then we can just set $d(p, q) = \infty$. Otherwise, $L_{p,q}(\mathcal{P})$ is not empty and we can compute a member in the form of an SLP by converting into a context-free grammar and applying Lemma 4.3(2). Furthermore, using Lemma 4.7 and Lemma 4.3(6) we can compute SLPs for $u_{p,q}, v_{p,q} \in A^*$ such that $\bar{u}_{p,q}v_{p,q} \in \mathsf{DyckNF}(L_{p,q}(\mathcal{P}))$. By Lemma 4.3(4), we can compute SLPs for words $r_{p,q}, s_{p,q}, t_{p,q} \in A^*$ such that $t_{p,q}$ is the longest common prefix of $u_{p,q}, v_{p,q}$ and $u_{p,q} = t_{p,q}r_{p,q}$ and $v_{p,q} = t_{p,q}s_{p,q}$.

Now we modify $\mathcal{P}$ as follows. Between p and q, we add a gadget: We build an annotated PDA $\mathcal{P}_{p,q}$ for $\{\bar{r}_{p,q}s_{p,q}\}$ using Lemma 4.4 and glue it between p and q as follows:



(3)

Here, the dashed box represents the PDA $\mathcal{P}_{p,q}$ and q_0 and q_1 are its initial and final state, respectively. Moreover, c is a fresh letter not used in $\mathcal{P}$ or $\mathcal{P}_{p,q}$. After performing this addition for every state pair (p, q), we arrive at the new PDA $\mathcal{P}'$. Furthermore, by replacing each PDA $\mathcal{P}$ in L with $\mathcal{P}'$, we obtain the new shuffle $L' \subseteq \Sigma^*$. We claim that:

(i) $L' \subseteq D_A$ if and only if $L \subseteq D_A$ and
(ii) $d_{\mathcal{P}'}(p, q) = |r_{p,q}|$ for states p, q appearing in $\mathcal{P}$

We begin with (ii). First, since $\bar{r}_{p,q}s_{p,q}$ belongs to $L_{p,q}(\mathcal{P}')$ by construction, the value $|r_{p,q}|$ is actually achieved as $d(w)$ for some $w \in L_{p,q}(\mathcal{P}')$. Hence, $d_{\mathcal{P}'}(p, q) \leq |r_{p,q}|$. Conversely, let w be any word in $L_{p,q}(\mathcal{P}')$. Then Lemma 4.5 tells us that $d(w) \geq |r_{p,q}|$. Therefore, $d_{\mathcal{P}'}(p, q) = |r_{p,q}|$.

For (i), we claim that $L \subseteq L' \subseteq L^\diamond$ and conclude the equivalence by Corollary 4.2. For the inclusion $L' \subseteq L^\diamond$, observe that any word in L' is obtained from a word in L by replacing factors from $L_{p,q}(\mathcal{P})$ by words $\bar{r}_{p,q}s_{p,q}$ as constructed above. Moreover, the word $\bar{r}_{p,q}s_{p,q}$ can be obtained from $\bar{u}_{p,q}v_{p,q} = \bar{r}_{p,q}\bar{t}_{p,q}t_{p,q}s_{p,q}$ by deleting $\bar{t}_{p,q}t_{p,q}$. Hence, we have $\bar{r}_{p,q}s_{p,q} \in L_{p,q}(\mathcal{P})^\diamond$. Therefore, every word $w' \in L$ can be written as

$$w' = v_0u_1'v_1 \cdots u_n'v_n,$$

where each u_i' satisfies $u_i \rightarrowtail^* u_i'$ for some $u_i \in \Sigma^*$ with

$$w = v_0 u_1 v_1 \cdots u_n v_n \in L.$$

In particular, we have $w \rightarrowtail^* w'$ and thus $w' \in L^\diamond$. Hence $L' \subseteq L^\diamond$, which yields the claim.

Finally, notice that this means we can produce a complete annotation for L': For states p, q that are newly introduced (as part of the gadget in Eq. (3)), we already have $d_{\mathcal{P}'}(p, q)$, since $\mathcal{P}_{p,q}$ is already annotated. For state pairs consisting of one state from $\mathcal{P}$ and one state of some $\mathcal{P}_{p,q}$, we set d to ∞. This is correct because by construction, it is not possible to read a word from Σ^* between such a state pair (because of the push and pop of c in Eq. (3)). □

5 REACHABILITY RELATIONS OF OFFSET-UNIFORM 2-VASS

Let us now prove Proposition 3.10. The key idea is to construct a 1-VASS so that reachability in our 2-VASS can be expressed in terms of reachability in the 1-VASS, to which we apply the following:

Proposition 5.1 ([Li et al. 2020]). *Given a 1-VASS and states p, q, one can construct in polynomial time an $\exists$PA formula for the relation $\{(x, y) \in \mathbb{N}^2 \mid p(x) \xrightarrow{*} q(y)\}$.*

The 1-VASS essentially tracks one of the two counters. Here, we need to take extra precautions to make sure the corresponding run in the 2-VASS will stay non-negative even in the counter we do not track explicitly. To this end, we distinguish two cases: Roughly speaking, we distinguish whether the counter difference in the initial configuration is (in a certain sense)

(A) large, in which case offset-uniformity implies that the initially larger counter automatically stays non-negative, or
(B) small, in which case the counter difference will always stay small and we can add gadgets that make sure the untracked counter stays non-negative.

We need a simple lemma.

Lemma 5.2. *Given an offset-uniform 2-VASS and some distinguished state p, one can compute in polynomial time, a number π_q for each state q such that:*

$$p(n_1, n_2) \xrightarrow{*} q(m_1, m_2) \text{ implies } m_2 - m_1 = n_2 - n_1 + \pi_q$$

for any configurations $p(n_1, n_2), q(m_1, m_2)$.

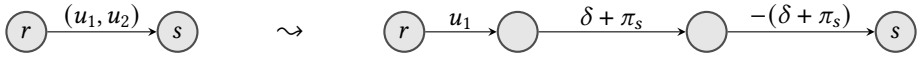
Proof. Given the 2-VASS, construct a directed graph G with vertices Q where for each transition $r \xrightarrow{(u_1,u_2)} q$ in the 2-VASS, there is an edge $r \to s$ labeled with the number $u_2 - u_1$. For each state q that is reachable from p in G, we take a path from p from to q and set π_q to be the sum of all labels on this path. For states q that are not reachable from p in G, we know that q can never be reached from p in the 2-VASS, so we can just set $\pi_q = 0$. It is easy to check that the π_q are as desired. □

Let us now formally prove Proposition 3.10.

Proof of Proposition 3.10. Let $V = (Q, T)$ be an offset-uniform 2-VASS. First, we use Lemma 5.2 on distinguished state p to compute a number $\pi_r \in \mathbb{Z}$ for each state r. Let $N = \max\{|\pi_r| \mid r \in Q\}$. Note that N might be exponential (because the π_r might be), but can be stored in binary.

Recall the two cases (A) and (B) described intuitively below Proposition 5.1. In order to simulate case (A), we simply have a 1-VASS that just tracks one of the counters. Thus, for $i = 1, 2$, we construct the 1-VASS V_i, which has a transition $r \xrightarrow{u_i} s$ for each transition $r \xrightarrow{(u_1,u_2)} s$ of our 2-VASS. In order to simulate case (B), we have a 1-VASS that depends on the small counter difference δ in the beginning. Here, "small" will mean $\delta \in [-N, N]$. Thus, for each $\delta \in [-N, N]$, we define

the 1-VASS V_δ as follows. Each transition $r \xrightarrow{(u_1,u_2)} s$ is transformed as follows into a sequence of transitions in V_δ:



where the unlabeled states are newly introduced. Here, the idea is that in each configuration $s(m_1, m_2)$ reachable from $p(n_1, n_1 + \delta)$, we must have $m_2 - m_1 = \delta + \pi_s$. Therefore, the 1-VASS V_δ simulates the first counter by first only adding u_1. Then, it temporarily switches to the counter value of the second counter in the original 2-VASS run, using the fact that this value is obtained by adding $\delta + \pi_s$. This way, it makes sure that also the second counter value remains non-negative. After this temporary switch, it switches back to the first counter by subtracting $\delta + \pi_s$.

We claim that then we have $p(n_1, n_2) \xrightarrow{*} q(m_1, m_2)$ if and only if $(m_2 - m_1) - (n_2 - n_1) = \pi_q$ and one of the following holds:

(1) $n_1 > n_2 + N$ and $p(n_2) \xrightarrow{*}_{V_2} q(m_2)$,
(2) $n_2 > n_1 + N$ and $p(n_1) \xrightarrow{*}_{V_1} q(m_1)$, or
(3) $n_2 - n_1 \in [-N, N]$ and for $\delta := n_2 - n_1$, we have $p(n_1) \xrightarrow{*}_{V_\delta} q(m_1)$.

In our claim, the "only if" direction follows directly by construction and offset-uniformity. For the "if" direction, we consider each case. In the first case, we start in a configuration $p(n_1, n_2)$ with $n_1 > n_2 + N$. By offset-unifomity and the choice of the numbers π_r, this implies that in any reachable configuration $r(m_1, m_2)$, we have $m_2 - m_1 = n_2 - n_1 + \pi_r < -N + \pi_r \leq 0$, which implies $m_1 \geq m_2$. Therefore, any transition sequence (starting in $p(n_1, n_2)$) in our 2-VASS that keeps the second counter non-negative will automatically keep the first one non-negative. This proves the "if" direction in the first case. The second case is analogous. For the third case, we have argued above that the 1-VASS V_δ guarantees that $p(n_1) \xrightarrow{*}_{V_\delta} q(m_1)$ implies that $p(n_1, n_1 + \delta) \xrightarrow{*} q(m_1, m_1 + \delta + \pi_q)$. This proves the claim. By our claim, the following formula defines the relation Eq. (2) (Page 18):

$$\left[\psi \wedge n_1 > n_2 + N \wedge \varphi_2(n_2, m_2)\right] \vee \left[\psi \wedge n_2 > n_1 + N \wedge \varphi_1(n_1, m_1)\right] \vee \bigvee_{\delta \in [-N,N]} \left[\psi \wedge n_2 - n_1 = \delta \wedge \varphi_\delta(n_1, m_1)\right] \tag{4}$$

Here, φ_i is an $\exists$PA formula for p-q-reachability in V_i for $i = 1, 2$ and φ_δ is an $\exists$PA formula for p-q-reachability in V_δ, and ψ is the $\exists$PA formula expressing $(m_2 - m_1) - (n_2 - n_1) = \pi_q$. The formulas φ_1, φ_2, and φ_δ can be constructed according to Proposition 5.1. We can clearly implement a non-deterministic polynomial-time algorithm that produces one of the (exponentially many) disjuncts of Eq. (4). □

6 CONCLUSION

We have described a coNP decision procedure for the context-bounded refinement problem for multithreaded shared memory programs against an expressive class of non-regular specifications. Our procedure provides the first automated technique for a number of common design patterns in concurrent systems, such as reference counted resources. We note that while our proof involves a number of subtle constructions, the final decision procedure is an encoding into existential Presburger arithmetic, for which efficient SMT solvers exist. Thus, we expect these checks to work well in practice, but leave empirical evaluation to future work.

There are two immediate directions for future work. The first expands the class of programs to include dynamic spawning of threads [Atig et al. 2011] (in our current model, the number of

threads is fixed). The second expands the class of specifications. There is a procedure (albeit doubly exponential) to decide if a CFL is included in a superdeterministic CFL [Greibach and Friedman 1980]. We do not know if our algorithm generalizes to this class of specifications. On the other hand, we do not know of natural specifications that lie outside our class but are superdeterministic.

ACKNOWLEDGMENTS

This research was sponsored in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248–CPEC and by the European Research Council under the Grant Agreement 610150 (http://www.impact-erc.eu/) (ERC Synergy Grant ImPACT).

REFERENCES

Rajeev Alur, Swarat Chaudhuri, and P. Madhusudan. 2011. Software model checking using languages of nested trees. *ACM Trans. Program. Lang. Syst.* 33, 5 (2011), 15:1–15:45. https://doi.org/10.1145/2039346.2039347

Rajeev Alur, Kousha Etessami, and P. Madhusudan. 2004. A Temporal Logic of Nested Calls and Returns. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 467–481. https://doi.org/10.1007/978-3-540-24730-2_35

Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, László Babai (Ed.). ACM, 202–211. https://doi.org/10.1145/1007352.1007390

Rajeev Alur and P. Madhusudan. 2009. Adding nesting structure to words. *J. ACM* 56, 3 (2009), 16:1–16:43. https://doi.org/10.1145/1516512.1516518

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

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)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 111:1–111:16. https://doi.org/10.4230/LIPIcs.ICALP.2020.111

Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2021. Context-Bounded Verification of Liveness Properties for Multithreaded Shared-Memory Programs. *Proceedings of the ACM on Programming Languages (PACMPL)* 5, POPL, Article 44 (Jan. 2021), 31 pages. https://doi.org/10.1145/3434325

Jean Berstel and Luc Boasson. 2002. Formal properties of XML grammars and languages. *Acta Informatica* 38, 9 (Aug 2002), 649–671. https://doi.org/10.1007/s00236-002-0085-4

Alberto Bertoni, Christian Choffrut, and Roberto Radicioni. 2011. The Inclusion Problem of Context-Free Languages: Some Tractable Cases. *Int. J. Found. Comput. Sci.* 22, 2 (2011), 289–299. https://doi.org/10.1142/S0129054111008040

Michael Blondin, Matthias Englert, Alain Finkel, Stefan Göller, Christoph Haase, Ranko Lazic, Pierre McKenzie, and Patrick Totzke. 2021. The Reachability Problem for Two-Dimensional Vector Addition Systems with States. *J. ACM* 68, 5 (2021), 34:1–34:43. https://doi.org/10.1145/3464794

I. Borosh and L. B. Treybig. 1976. Bounds on Positive Integral Solutions of Linear Diophantine Equations. *Proc. Amer. Math. Soc.* 55, 2 (1976), 299–304.

Wojciech Czerwiński and Lukasz Orlikowski. 2021. Reachability in Vector Addition Systems is Ackermann-complete. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. IEEE, 1229–1240. https://doi.org/10.1109/FOCS52979.2021.00120

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. 2009. Verifying Reference Counting Implementations. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5505)*, Stefan Kowalewski and Anna Philippou (Eds.). Springer, 352–367. https://doi.org/10.1007/978-3-642-00768-2_30

Matthias Englert, Piotr Hofman, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Juliusz Straszynski. 2021. A lower bound for the coverability problem in acyclic pushdown VAS. *Inf. Process. Lett.* 167 (2021), 106079. https://doi.org/10.1016/j.ipl.2020.106079

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2014. Proofs that count. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 151–164. https://doi.org/10.1145/2535838.2535885

Kostas Ferles, Jon Stephens, and Isil Dillig. 2021. Verifying correct usage of context-free API protocols. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434298

Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, and Jean-Marc Talbot. 2018. Visibly pushdown transducers. *J. Comput. Syst. Sci.* 97 (2018), 147–181. https://doi.org/10.1016/j.jcss.2018.05.002

Emily P. Friedman. 1976. The inclusion problem for simple languages. *Theoretical Computer Science* 1:4 (1976), 297–316.

Sheila A Greibach and Emily P Friedman. 1980. Superdeterministic PDAs: A subcase with a decidable inclusion problem. *Journal of the ACM (JACM)* 27, 4 (1980), 675–700.

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

Donald E. Knuth. 1967. A Characterization of Parenthesis Languages. *Information and Control* 11, 3 (Sept. 1967), 269–289. https://doi.org/10.1016/S0019-9958(67)90564-5

Naoki Kobayashi. 2019. Inclusion between the frontier language of a non-deterministic recursive program scheme and the Dyck language is undecidable. *Theoretical Computer Science* 777 (2019), 409–416.

Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2009. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 477–492. https://doi.org/10.1007/978-3-642-02658-4_36

Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. 2010. The Language Theory of Bounded Context-Switching. In *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010, Proceedings (Lecture Notes in Computer Science, Vol. 6034)*. Springer, 96–107. https://doi.org/10.1007/978-3-642-12200-2_10

Akash Lal and Thomas W. Reps. 2009. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design* 35, 1 (2009), 73–97. https://doi.org/10.1007/s10703-009-0078-9

Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. 2008. Interprocedural Analysis of Concurrent Programs under a Context Bound. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 282–298.

Jérôme Leroux. 2021. The Reachability Problem for Petri Nets is Not Primitive Recursive. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. IEEE, 1241–1252. https://doi.org/10.1109/FOCS52979.2021.00121

Jérôme Leroux and Sylvain Schmitz. 2019. Reachability in Vector Addition Systems is Primitive-Recursive in Fixed Dimension. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, Canada, June 24-27, 2019*. 1–13. https://doi.org/10.1109/LICS.2019.8785796

Jérôme Leroux and Grégoire Sutre. 2004. On Flatness for 2-Dimensional Vector Addition Systems with States. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3170)*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer, 402–416. https://doi.org/10.1007/978-3-540-28644-8_26

Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. 2015. On the Coverability Problem for Pushdown Vector Addition Systems in One Dimension. 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, 324–336. https://doi.org/10.1007/978-3-662-47666-6_26

Xie Li, Taolue Chen, Zhilin Wu, and Mingji Xia. 2020. Computing Linear Arithmetic Representation of Reachability Relation of One-Counter Automata. In *Dependable Software Engineering. Theories, Tools, and Applications - 6th International Symposium, SETTA 2020, Guangzhou, China, November 24-27, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12153)*, Jun Pang and Lijun Zhang (Eds.). Springer, 89–107. https://doi.org/10.1007/978-3-030-62822-2_6

Richard J Lipton and Yechezkel Zalcstein. 1977. Word problems solvable in logspace. *Journal of the ACM (JACM)* 24, 3 (1977), 522–526.

Raphaela Löbel, Michael Luttenberger, and Helmut Seidl. 2021. On the Balancedness of Tree-to-Word Transducers. *Int. J. Found. Comput. Sci.* 32, 6 (2021), 761–783. https://doi.org/10.1142/S0129054121420077

Markus Lohrey. 2012. Algorithmics on SLP-compressed strings: A survey. *Groups Complex. Cryptol.* 4, 2 (2012), 241–299. https://doi.org/10.1515/gcc-2012-0016

Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. 2015. Automating grammar comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 183–200. https://doi.org/10.1145/2814270.2814304

P. Madhusudan and Gennaro Parlato. 2011. The tree width of auxiliary storage. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 283–294. https://doi.org/10.1145/1926385.1926419

Sebastian Maneth and Helmut Seidl. 2018. Balancedness of MSO transductions in polynomial time. *Inform. Process. Lett.* 133 (2018), 26–32.

Roland Meyer, Sebastian Muskalla, and Georg Zetzsche. 2018. Bounded Context Switching for Valence Systems. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs, Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.12

Andrzej S. Murawski, C.-H. Luke Ong, and Igor Walukiewicz. 2005. Idealized Algol with Ground Recursion, and DPDA Equivalence. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3580)*, Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung (Eds.). Springer, 917–929. https://doi.org/10.1007/11523468_74

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

Tmima Olshansky and Amir Pnueli. 1977. A direct algorithm for checking equivalence of LL(k) grammars. *Theoretical Computer Science* 4, 3 (1977), 321–349. https://doi.org/10.1016/0304-3975(77)90016-0

C.-H. Luke Ong. 2002. Observational Equivalence of 3rd-Order Idealized Algol is Decidable. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 245–256. https://doi.org/10.1109/LICS.2002.1029833

Wojciech Plandowski. 1994. Testing Equivalence of Morphisms on Context-Free Languages. In *Algorithms - ESA '94, Second Annual European Symposium, Utrecht, The Netherlands, September 26-28, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 855)*, Jan van Leeuwen (Ed.). Springer, 460–470. https://doi.org/10.1007/BFb0049431

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)*. Springer, 93–107. https://doi.org/10.1007/978-3-540-31980-1_7

Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William W. Pugh and Craig Chambers (Eds.). ACM, 14–24. https://doi.org/10.1145/996841.996845

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 49–61. https://doi.org/10.1145/199448.199462

Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.* 58, 1-2 (2005), 206–263. https://doi.org/10.1016/j.scico.2005.02.009

Robert W Ritchie and Frederick N Springsteel. 1972. Language recognition by marking automata. *Information and Control* 20, 4 (1972), 313–330.

Daniel J. Rosenkrantz and Richard E. Stearns. 1970. Properties of deterministic top-down grammars. *Information and Control* 17, 3 (1970), 226–256. https://doi.org/10.1016/S0019-9958(70)90446-8

Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1&2 (1996), 131–170. https://doi.org/10.1016/0304-3975(96)00072-2

Saul Schleimer. 2008. Polynomial-time word problems. *Commentarii mathematici helvetici* 83, 4 (2008), 741–765.

Géraud Sénizergues. 1997. The equivalence problem for deterministic pushdown automata is decidable. In *International Colloquium on Automata, Languages, and Programming*. Springer, 671–681.

Aneesh K. Shetty, Shankara Narayanan Krishna, and Georg Zetzsche. 2021. Scope-Bounded Reachability in Valence Systems. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference (LIPIcs, Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:19. https://doi.org/10.4230/LIPIcs.CONCUR.2021.29

Michael Sipser. 2012. *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning, Inc.

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

Akihiko Tozawa and Yasuhiko Minamide. 2007a. Complexity Results on Balanced Context-Free Languages. In *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4423)*, Helmut Seidl (Ed.). Springer, 346–360. https://doi.org/10.1007/978-3-540-71389-0_25

Akihiko Tozawa and Yasuhiko Minamide. 2007b. *Complexity Results on Balanced Context-Free Languages*. Lecture Notes in Computer Science, Vol. 4423. Springer Berlin Heidelberg, 346–360. https://doi.org/10.1007/978-3-540-71389-0_25

Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. 2005. On the Complexity of Equational Horn Clauses. In *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3632)*, Robert Nieuwenhuis (Ed.). Springer, 337–352. https://doi.org/10.1007/11532231_25

Received 2022-07-07; accepted 2022-11-07