





# Checking Refinement of Asynchronous Programs Against Context-Free Specifications

Pascal Baumann  



Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Moses Ganardi  

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Rupak Majumdar  

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Ramanathan S. Thinniyam  

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Georg Zetsche  

Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

---

## Abstract

---

In the language-theoretic approach to refinement verification, we check that the language of traces of an implementation all belong to the language of a specification. We consider the refinement verification problem for asynchronous programs against specifications given by a Dyck language. We show that this problem is  $\text{EXPSPACE}$ -complete – the same complexity as that of language emptiness and for refinement verification against a regular specification. Our algorithm uses several technical ingredients. First, we show that checking if the coverability language of a succinctly described vector addition system with states (VASS) is contained in a Dyck language is  $\text{EXPSPACE}$ -complete. Second, in the more technical part of the proof, we define an ordering on words and show a downward closure construction that allows replacing the (context-free) language of each task in an asynchronous program by a regular language. Unlike downward closure operations usually considered in infinite-state verification, our ordering is not a well-quasi-ordering, and we have to construct the regular language ab initio. Once the tasks can be replaced, we show a reduction to an appropriate VASS and use our first ingredient. In addition to the inherent theoretical interest, refinement verification with Dyck specifications captures common practical resource usage patterns based on reference counting, for which few algorithmic techniques were known.

**2012 ACM Subject Classification** Theory of computation → Concurrency; Software and its engineering → Software verification

**Keywords and phrases** Asynchronous programs, VASS, Dyck languages, Language inclusion, Refinement verification

**Digital Object Identifier** 10.4230/LIPIcs.ICALP.2023.110

**Category** Track B: Automata, Logic, Semantics, and Theory of Programming

**Related Version** *Full Version:* <https://arxiv.org/abs/2306.13058>

**Funding** Funded by the European Union (ERC, FINABIS, 101077902). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. Partially funded by the DFG project 389792660 TRR 248-CPEC.



© Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetsche; licensed under Creative Commons License CC-BY 4.0

50th International Colloquium on Automata, Languages, and Programming (ICALP 2023).  
Editors: Kousha Etessami, Uriel Feige, and Gabriele Puppis; Article No. 110; pp. 110:1–110:20



Leibniz International Proceedings in Informatics  
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



## 1 Introduction

Asynchronous programs are a common programming idiom for multithreaded shared memory concurrency. An asynchronous program executes tasks atomically; each task is a sequential recursive program that can read or write some shared state, emit events (such as calling an API), and, in addition, can spawn an arbitrary number of new tasks for future execution. A cooperative scheduler iteratively picks a previously spawned task and executes it atomically to completion. Asynchronous programs occur in many software systems with stringent correctness requirements. At the same time, they form a robustly decidable class of infinite-state systems closely aligned with other concurrency models. Thus, algorithmic verification of asynchronous programs has received a lot of attention from both theoretical and applied perspectives [25, 13, 10, 8, 9, 15, 11, 16, 20].

We work in the language-theoretic setting, where we treat asynchronous programs as generators of languages, and reduce verification questions to decision problems on these languages. Thus, an execution of a task yields a word over the alphabet of its events and task names. An execution of the asynchronous program concatenates the words of executing tasks and further ensures that any task executing in the concatenation was spawned before and not already executed. The trace of an execution projects the word to the alphabet of events and the language of the program is the set of all traces. With this view, reachability or safety verification questions reduce to language emptiness, and refinement verification reduces to language inclusion of a program in a given specification language over the alphabet of events.

We consider the language inclusion problem for asynchronous programs when the specification language is given by a Dyck language. Our main result shows that this problem is EXPSPACE-complete. The language emptiness problem for asynchronous programs, as well as language inclusion in a regular language, are already EXPSPACE-complete [10]. Thus, there is no increase in complexity even when the specifications are Dyck languages. However, as we shall see below, our proof of membership in EXPSPACE requires several new ingredients.

In addition to the inherent language-theoretic interest, the problem is motivated by the practical “design pattern” of reference counting and barrier synchronization in concurrent event-driven programs. In this pattern, each global shared resource maintains a counter of *how many* processes have access to it. Before working with the shared resource, a task acquires access to the resource by incrementing a counter (the reference count). Later, a possibly different task can release the resource by decrementing the reference count. When the count is zero, the system can garbage collect the resource. For example, device drivers in the kernel maintain such reference counts, and there are known bugs arising out of incorrect handling of reference counts [21]. Here is a small snippet that shows the pattern in asynchronous code:

```

start :   { t := inc(); if (t) spawn(work); }
           // arbitrarily many requests may start concurrently
work :   { in this code, we can assert that the reference count is positive ;
           spawn(cleanup); }
cleanup : { dec(); if zeroref() { garbage collect the resource }}

```

Here, **inc** and **dec** increment and decrement the reference count associated with a shared resource, **inc** succeeds if the resource has not been garbage collected. **spawn** starts a new task, and **zeroref** checks if the reference count is zero. There are three tasks, **start**, **work**, and **cleanup**; each invocation of a task executes atomically. Initially, an arbitrary number of **start** tasks are spawned.

Our goal is to ensure the device is not garbage collected while some instance of **work** is pending. Intuitively, the reason for this is clear: each **work** is spawned by a previous **start** that takes a reference count and this reference is held until a later **cleanup** runs. However,

it is difficult for automated model checking tools to perform this reasoning, and existing techniques require manual annotations of invariants [11, 15]. Dyck languages allow specifying correct handling of reference counts [1], and our algorithm provides as a special case an algorithmic analysis of correct reference counting for asynchronous programs.

Since there is a simple reduction from language emptiness to inclusion, we immediately inherit EXPSPACE-hardness. Let us therefore focus on the challenges in obtaining an EXPSPACE upper bound. The EXPSPACE algorithm for language emptiness proceeds as follows (see [10, 20]). First, we can ignore the alphabet of events and only consider words over the alphabet of task names. Second, we notice that (non-)emptiness is preserved if we “lose” some spawns along an execution; this allows us to replace the language of each task by its downward closure. By general results about well-quasi orderings, the downward closure is a regular language which, moreover, has a succinct representation. Thus, we can reduce the language emptiness problem to checking (coverability) language emptiness of an associated vector addition system with states (VASS). This problem can be solved in EXPSPACE, by a result of Rackoff [22].

Unfortunately, this outline is not sufficient in our setting. First, unlike for language emptiness or regular language inclusion, we cannot simply replace tasks with their downward closures (w.r.t. the subword ordering). While we can drop spawns as before, dropping letters from the event alphabet does not preserve membership in a Dyck language. Second, even if each handler is regular, we are left with checking if a VASS language is contained in a Dyck language. We provide new constructions to handle these challenges.

Our starting point is the characterization of inclusion in Dyck languages [23]: A language  $L$  is not included in a Dyck language if and only if there is a word  $w \in L$  with either an *offset violation* (number of open brackets does not match the number of closed brackets), a *dip violation* (some prefix with more closed brackets than open ones), or a *mismatch violation* (an open bracket of one kind matched with a closed bracket of a different kind).

**Checking VASS Language Inclusion.** Our first technical construction shows how to check language inclusion of a VASS coverability language in a Dyck language in EXPSPACE. (In a *coverability language*, acceptance is defined by reaching a final control state.) In fact, our result carries over when the control states of the VASS are *succinctly represented*, for example by using transducers and binary encodings of numbers.

We first check that the VASS language is *offset-uniform*, that is, every word in the language has exactly the same offset (difference between open brackets and closed brackets), and that this offset is actually zero. (If this condition is not true, there is already an offset violation.) We show that the offset of every prefix of a word in any offset-uniform VASS language is bounded by a doubly exponential number, and therefore, this number can be tracked by adding double exponentially bounded counters (as in Lipton’s construction [18]) in the VASS itself. Moreover, we can reduce the checking of dip or mismatch violations to finding a *marked Dyck factor*: an infix of the form  $\#w\bar{\#}$  for a Dyck word  $w$ . Finally, for offset-uniform VASS, finding a marked Dyck factor reduces to coverability in succinctly represented VASS, which can be checked in EXPSPACE [2]. Offset uniformity is important – finding a marked Dyck factor in an arbitrary VASS language is equivalent to VASS reachability, which is Ackermann-complete [7, 17]. In fact, checking whether a given VASS language is included in the set of *prefixes* of the one-letter Dyck language is already equivalent to VASS reachability (see the long version of the paper for a proof).

A consequence of our result is that given a VASS coverability language  $K$  and a *reachability language* (i.e. acceptance requires all counters to be zero in the end)  $L$  of a *deterministic* VASS, deciding whether  $K \subseteq L$  is EXPSPACE-complete. This is in contrast (but not in contradiction<sup>1</sup>) to recent Ackermann-completeness results for settings where both  $K$  and  $L$  are drawn from subclasses of VASS coverability languages [6].

**Downward Closure of Tasks.** Next, we move to asynchronous programs. We define a composite ordering on words that is a combination of two different orderings: the subword ordering for task names, and the syntactic preorder on the events projected to a single set  $\{x, \bar{x}\}$  of Dyck letters. In our case, the latter means a word  $u$  is less than  $v$  iff they both have the same offset, but  $v$  has at most the dip of  $u$ . The composite order is defined so as to preserve the existence of marked Dyck factors. In contrast to the subword ordering, this (composite) ordering is not a well-quasi-ordering (since, e.g.,  $\bar{x}x, \bar{x}\bar{x}xx, \bar{x}\bar{x}\bar{x}xxx, \dots$  forms an infinite descending chain). Nevertheless, our most difficult technical construction shows that for any context-free language (satisfying an assumption, which we call tame-pumping) there exists a regular language with the same downward closure in this ordering. The case of general context-free languages reduces to this special case since the presence of a non-tame pump immediately results in a Dyck-violation and can easily be detected in PSPACE. For the tame-pumping grammars, a succinct description of the corresponding automaton can be computed in PSPACE. This key observation allows us to replace the context-free languages of tasks with regular sets, and thereby reduce the problem to checking VASS language inclusion.

**Related Work.** Language inclusion in Dyck languages is a well-studied problem. For example, inclusion in a Dyck language can be checked in polynomial time for context-free languages [26] or for ranges of two-copy tree-to-string transducers [19]. Our work extends the recent result that the language noninclusion problem for context-bounded multi-pushdown systems in Dyck languages is NP-complete [1]. Our result is complementary to that of [1]: their model considers a *fixed* number of threads but allows the threads to be interrupted and context-switched a fixed number of times. In contrast, we allow dynamic spawning of threads but assume each thread is atomically run to completion. A natural open question is whether our results continue to hold if threads can be interrupted up to a fixed number of times.

Inclusion problems have recently also been studied when both input languages are given as VASS coverability languages [6]. Since in our setting, the supposedly larger language is always a Dyck language (which is not a coverability VASS language), those results are orthogonal.

## 2 Language-Theoretic Preliminaries

**General Definitions.** We assume familiarity with basic language theory, see the textbook [14] for more details. For an alphabet  $\Sigma \subseteq \Theta$ , let  $\pi_\Sigma: \Theta^* \rightarrow \Sigma^*$  denote the projection onto  $\Sigma^*$ . In other words, for  $w \in \Theta^*$ , the word  $\pi_\Sigma(w)$  is obtained from  $w$  by deleting every occurrence of a letter in  $\Theta \setminus \Sigma$ . If  $\Sigma$  contains few elements, e.g.  $\Sigma = \{x, y\}$ , then instead of writing  $\pi_{\{x,y\}}$  we also write  $\pi_{x,y}$ , leaving out the set brackets. We write  $|w|_\Sigma$  for the number of occurrences of letters  $x \in \Sigma$  in  $w$ , and similarly  $|w|_x$  if  $\Sigma = \{x\}$ .

<sup>1</sup> For general VASS, every coverability language is also a reachability language. However, *deterministic* VASS with reachability acceptance cannot accept all coverability languages.

**Context-Free Languages.** A *context-free grammar* (CFG)  $\mathcal{G} = (N, \Theta, P, S)$  consists of an alphabet of *nonterminals*  $N$ , an alphabet of *terminals*  $\Theta$  with  $N \cap \Theta = \emptyset$ , a finite set of *productions*  $P \subseteq N \times (N \cup \Theta)^*$ , and the start symbol  $S \in N$ . We usually write  $A \rightarrow v$  to denote a production  $(A, v) \in P$ . The size of the CFG  $\mathcal{G}$  is defined as  $|\mathcal{G}| = \sum_{A \rightarrow v \in P} (|v| + 1)$ . We denote the *derivation relation* by  $\Rightarrow_{\mathcal{G}}$  and its reflexive, transitive closure by  $\stackrel{*}{\Rightarrow}_{\mathcal{G}}$ . We drop the subscript  $\mathcal{G}$  if it is clear from the context. We also use *derivation trees* labelled by  $N \cup \Theta$  for derivations of the form  $A \stackrel{*}{\Rightarrow} w$  for some  $A \in N$ . Here we start with the root labelled by  $A$ , and whenever we apply a production  $B \rightarrow v$  with  $v = a_1 \dots a_n$ , we add  $n$  children labelled by  $a_1, \dots, a_n$  (in that order from left to right) to a leaf labelled by  $B$ . A *pump* is a derivation of the form  $A \stackrel{*}{\Rightarrow} uAv$  for some nonterminal  $A$ . A derivation tree which is pumpfree, i.e., in which no path contains multiple occurrences of the same nonterminal, is referred to as a *skeleton*. We will often see an arbitrary derivation tree as one which is obtained by inserting pumps into a skeleton.

The *language*  $L(\mathcal{G}, A)$  of  $\mathcal{G}$  starting from nonterminal  $A \in N$  contains all words  $w \in \Theta^*$  such that there exists a derivation  $A \stackrel{*}{\Rightarrow}_{\mathcal{G}} w$ . The language of  $\mathcal{G}$  is  $L(\mathcal{G}) = L(\mathcal{G}, S)$ . A *context-free language* (CFL)  $L$  is a language for which there exists a CFG  $\mathcal{G}$  with  $L = L(\mathcal{G})$ .

A CFG  $\mathcal{G} = (N, \Theta, P, S)$  is said to be in *Chomsky normal form* if all of its productions have one of the forms  $A \rightarrow BC$ ,  $A \rightarrow a$ , or  $S \rightarrow \varepsilon$ , where  $B, C \in N \setminus \{S\}$ ,  $a \in \Theta$ , and the last form only occurs if  $\varepsilon \in L(\mathcal{G})$ . It is well known that every CFG can be transformed in polynomial time into one in Chomsky normal form with the same language.

An *extended context-free grammar* (ECFG)  $\mathcal{G} = (N, \Theta, P, S)$  is a CFG, which may additionally have productions of the form  $A \rightarrow \Gamma^* \in P$  for some alphabet  $\Gamma \subseteq \Theta$ . Productions of this form induce derivations  $uAs \Rightarrow_{\mathcal{G}} uvs$ , where  $u, s \in (N \cup \Theta)^*$  and  $v \in \Gamma^*$ . Chomsky normal form for ECFG is defined as for CFG, but also allows productions of the form  $A \rightarrow \Gamma^*$ . An ECFG can still be transformed into Chomsky normal form using the same algorithm as for a CFG, treating expressions  $\Gamma^*$  like single terminal symbols. Since the extended productions can be simulated by conventional CFG productions, the language of an ECFG is still a CFL.

**Dyck Language.** Let  $X$  be an alphabet and let  $\bar{X} = \{\bar{x} \mid x \in X\}$  be a disjoint copy of  $X$ . The *Dyck language (over  $X$ )*  $\text{Dyck}_X \subseteq (X \cup \bar{X})^*$  is defined by the following context-free grammar:

$$S \rightarrow \varepsilon \mid S \rightarrow SS \mid S \rightarrow xS\bar{x} \quad \text{for } x \in X.$$

Let  $\Theta \supseteq X \cup \bar{X}$  be an alphabet. For  $w \in \Theta^*$  we define  $\text{offset}(w) = |w|_X - |w|_{\bar{X}}$ . A language  $L \subseteq \Theta^*$  is called *offset-uniform* if for any  $u, v \in L$ , we have  $\text{offset}(u) = \text{offset}(v)$ .

The *dip* of  $w \in \Theta^*$  is defined as  $\text{dip}(w) = \max\{-\text{offset}(u) \mid u \text{ is a prefix of } w\}$ . We define  $e(w) = (\text{dip}(w), \text{offset}(w))$ . Observe that for  $w \in (X \cup \bar{X})^*$  with  $|X| = 1$  we have  $w \in \text{Dyck}_X$  if and only if  $e(w) = (0, 0)$ .

A language  $L \subseteq (X \cup \bar{X})^*$  is *not* included in  $\text{Dyck}_X$  if and only if there exists a word  $w \in L$  that satisfies one of the following violation conditions [23]:

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

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

### 3 Asynchronous Programs

An *asynchronous program* [10], henceforth simply called a *program*, is a tuple  $\mathcal{P} = (Q, \Sigma, \Gamma, \mathcal{G}, \Delta, q_0, q_f, \gamma_0)$ , where  $Q$  is a finite set of *global states*,  $\Sigma$  is an alphabet of *event letters*,  $\Gamma$  is an alphabet of *handler names* with  $\Sigma \cap \Gamma = \emptyset$ ,  $\mathcal{G}$  is a CFG over the terminal symbols  $\Sigma \cup \Gamma$ ,  $\Delta$  is a finite set of transition rules (described below),  $q_0 \in Q$  is the *initial state*,  $q_f \in Q$  is the *final state*, and  $\gamma_0$  is the *initial handler*.

Transition rules in  $\Delta$  are of the form  $q \xrightarrow{a,A} q'$ , where  $q, q' \in Q$  are global states,  $a \in \Gamma$  is a handler name, and  $A$  is a nonterminal symbol in  $\mathcal{G}$ .

Let  $\mathbb{M}[S]$  denote the set of all multisets of elements from the set  $S$ . A *configuration*  $(q, \mathbf{m}) \in Q \times \mathbb{M}[\Gamma]$  of  $\mathcal{P}$  consists of a global state  $q$  and a multiset  $\mathbf{m} : \Gamma \rightarrow \mathbb{N}$  of pending handler instances. The *initial* configuration of  $\mathcal{P}$  is  $c_0 = (q_0, \llbracket \gamma_0 \rrbracket)$ , where  $\llbracket \gamma_0 \rrbracket$  denotes the singleton multiset containing  $\gamma_0$ . A configuration is considered *final* if its global state is  $q_f$ . The rules in  $\Delta$  induce a transition relation on configurations of  $\mathcal{P}$ : We have  $(q, \mathbf{m}) \xrightarrow{w} (q', \mathbf{m}')$  iff there is a rule  $q \xrightarrow{a,A} q' \in \Delta$  and a word  $u \in L(\mathcal{G}, A)$  such that  $\pi_\Sigma(u) = w$  and  $\mathbf{m}' = (\mathbf{m} \ominus \llbracket a \rrbracket) \oplus \text{Parikh}(\pi_\Gamma(u))$ , where  $\mathbf{m}'' = \mathbf{m} \oplus \mathbf{m}'$  is the multiset which satisfies  $\mathbf{m}''(a) = \mathbf{m}'(a) + \mathbf{m}(a)$  for each  $a \in \Gamma$ . Similarly  $\mathbf{m}'' = \mathbf{m} \ominus \mathbf{m}'$  is the multiset which satisfies  $\mathbf{m}''(a) = \mathbf{m}'(a) - \mathbf{m}(a)$  for each  $a \in \Gamma$  with the implicit assumption that  $\mathbf{m}'(a) \geq \mathbf{m}(a)$ . Here,  $\text{Parikh}(w) : \Gamma \rightarrow \mathbb{N}$  is the *Parikh image* of  $w$  that maps each handler in  $\Gamma$  to its number of occurrences in  $w$ . Note that the transition is feasible only if  $\mathbf{m}$  contains at least one instance of the handler  $a$ .

Intuitively, a program consists of a set of atomic event handlers that communicate over a shared global state  $Q$ . Each handler is a piece of sequential code that generates a word over a set of events  $\Sigma$  and, in addition, posts new instances of handlers from  $\Gamma$ . A configuration  $(q, \mathbf{m})$  represents the current value of the shared state  $q$  and a task buffer  $\mathbf{m}$  containing the posted, but not yet executed, handlers. At each step, a scheduler non-deterministically picks and removes a handler from the multiset of posted handlers and “runs” it. Running a handler changes the global state and produces a sequence of events over  $\Sigma$  as well as a multiset of newly posted handlers. The newly posted handlers are added to the task buffer.

We consider asynchronous programs as generators of words over the set of events. A *run* of  $\mathcal{P}$  is a finite sequence of configurations  $c_0 = (q_0, \llbracket \gamma_0 \rrbracket) \xrightarrow{w_1} c_1 \xrightarrow{w_2} \dots \xrightarrow{w_\ell} c_\ell$ . It is an *accepting* run if it ends in a final configuration.

The *language* of  $\mathcal{P}$  is defined as

$$L(\mathcal{P}) = \{w \in \Sigma^* \mid w = w_1 \cdots w_\ell, \text{ there is an accepting run } c_0 \xrightarrow{w_1} \dots \xrightarrow{w_\ell} c_\ell\}.$$

The size of the program  $\mathcal{P}$  is defined as  $|\mathcal{P}| = |Q| + |\mathcal{G}| + |\Delta|$ , i.e., the combined size of states, grammar, and transitions.

The *Dyck inclusion problem* for programs asks, given a program  $\mathcal{P}$  over a set  $(X \cup \bar{X})$  of events, whether every word in  $L(\mathcal{P})$  belongs to the Dyck language  $\text{Dyck}_X$ . We show the following main result.

► **Theorem 3.1** (Main Theorem). *Given a program  $\mathcal{P}$  with  $L(\mathcal{P}) \subseteq (X \cup \bar{X})^*$ , deciding if  $L(\mathcal{P}) \subseteq \text{Dyck}_X$  is EXPSPACE-complete.*

EXPSPACE-hardness follows easily from the following result on language emptiness (by simply adding a loop with a letter  $\bar{x} \in \bar{X}$  at the final state). Therefore, the rest of the paper focuses on the EXPSPACE upper bound.

► **Proposition 3.2** (Theorem 6.2, Ganty and Majumdar [10]). *Given a program  $\mathcal{P}$ , checking if  $L(\mathcal{P}) = \emptyset$  is EXPSPACE-complete.*



A nonterminal  $B$  in the grammar  $\mathcal{G}$  of a program  $\mathcal{P}$  is called *useful* if there exists a run  $\rho$  of  $\mathcal{P}$  reaching  $q_f$  in which there exists a derivation tree containing  $B$ . More precisely, there are two successive configurations  $(q, \mathbf{m}) \xrightarrow{w} (q', \mathbf{m}')$  in  $\rho$  such that there is a rule  $q \xrightarrow{a,A} q'$  and a word  $u \in L(\mathcal{G}, A)$  with  $\pi_\Sigma(u) = w$ ,  $\mathbf{m}' = (\mathbf{m} \ominus \llbracket a \rrbracket) \oplus \text{Parikh}(\pi_\Gamma(u))$ , and  $B$  occurs in some derivation tree with root  $A$  and yield  $u$ . There is a simple reduction from checking if a nonterminal is useful to checking language emptiness (see the full version) so we can check if a nonterminal is useful also in EXPSPACE. Therefore, in the following, we shall assume that all nonterminals are useful.

## 4 Checking Dyck Inclusion for VASS Coverability Languages

As a first technical construction, we show how to check Dyck inclusion for (succinctly defined) VASS languages. We shall reduce the problem for programs to this case.

### 4.1 Models: VASS and Succinct Versions

**Vector Addition Systems with States.** A *vector addition system with states* (VASS) is a tuple  $\mathcal{V} = (Q, \Sigma, I, E, q_0, q_f)$  where  $Q$  is a finite set of *states*,  $\Sigma$  is a finite alphabet of *input letters*,  $I$  is a finite set of *counters*,  $q_0 \in Q$  is the *initial state*,  $q_f \in Q$  is the *final state*, and  $E$  is a finite set of *edges* of the form  $q \xrightarrow{x,\delta} q'$ , where  $q, q' \in Q$ ,  $x \in \Sigma \cup \{\varepsilon\}$ , and  $\delta \in \{-1, 0, 1\}^I$ .<sup>2</sup>

A *configuration* of  $\mathcal{V}$  is a pair  $(q, \mathbf{u}) \in Q \times \mathbb{M}[I]$ . The elements of  $\mathbb{M}[I]$  and  $\{-1, 0, 1\}^I$  can also be seen as vectors of length  $|I|$  over  $\mathbb{N}$  and  $\{-1, 0, 1\}$ , respectively, and we sometimes denote them as such. The edges in  $E$  induce a transition relation on configurations: there is a transition  $(q, \mathbf{u}) \xrightarrow{x} (q', \mathbf{u}')$  if there is an edge  $q \xrightarrow{x,\delta} q'$  in  $E$  such that  $\mathbf{u}'(i) = \mathbf{u}(i) + \delta(i) \geq 0$  for all  $i \in I$ . A *run* of the VASS is a finite sequence of configurations  $c_0 \xrightarrow{x_1} c_1 \xrightarrow{x_2} \dots \xrightarrow{x_\ell} c_\ell$  where  $c_0 = (q_0, \mathbf{0})$ . A run is said to reach a state  $q \in Q$  if the last configuration in the run is of the form  $(q, \mathbf{m})$  for some multiset  $\mathbf{m}$ . An *accepting* run is a run whose final configuration has state  $q_f$ . The (*coverability*) *language* of  $\mathcal{V}$  is defined as

$$L(\mathcal{V}) = \{w \in \Sigma^* \mid \text{there exists a run } (q_0, \mathbf{0}) = c_0 \xrightarrow{x_1} \dots \xrightarrow{x_\ell} c_\ell = (q_f, \mathbf{u}) \text{ with } w = x_1 \dots x_\ell\}.$$

The size of the VASS  $\mathcal{V}$  is defined as  $|\mathcal{V}| = |I| \cdot |E|$ .

**Models with Succinct Control.** In this paper we need various models with *doubly succinct* control, i.e., models with doubly exponentially many states. Informally speaking, a machine with finite control  $\mathcal{B}$ , e.g. an NFA or a VASS, is doubly succinct if its set of control states is  $\Lambda^M$  where  $M \in \mathbb{N}$  is an exponential number given in binary encoding, and  $\Lambda$  is a finite alphabet. The initial and final state of  $\mathcal{B}$  are the states  $0^M$  and  $1^M$  for some letters  $0, 1 \in \Lambda$ . Finally, the transitions of  $\mathcal{B}$  are given by *finite-state transducers*  $\mathcal{T}$ , i.e., asynchronous multitape automata recognizing relations  $R \subseteq (\Lambda^M)^k$ . For example, a *doubly succinct* NFA (dsNFA in short) contains binary transducers  $\mathcal{T}_a$  for each  $a \in \Sigma \cup \{\varepsilon\}$  where  $\Sigma$  is the input alphabet, and  $\mathcal{B}$  contains a transition  $p \xrightarrow{x} q$  if and only if  $(p, q)$  is accepted by  $\mathcal{T}_x$ . A *doubly succinct* VASS (dsVASS, for short) contains binary transducers  $\mathcal{T}_{x,i}, \mathcal{T}_{x,\bar{i}}, \mathcal{T}_{x,\varepsilon}$  for each  $x \in \Sigma \cup \{\varepsilon\}$  and  $i \in I$ , where  $I$  is the set of counters. A state pair  $(p, q)$  accepted by  $\mathcal{T}_{x,i}$  specifies

<sup>2</sup> A more general definition of VASS would allow each transition to add an arbitrary vector over the integers. We instead restrict ourselves to the set  $\{-1, 0, 1\}$ , since this suffices for our purposes, and the EXPSPACE-hardness result by Lipton [18] already holds for VASS of this form.

a transition  $p \xrightarrow{x, \mathbf{e}_i} q$  in  $\mathcal{B}$ , where  $\mathbf{e}_i$  only increments counter  $i$  and leaves other counters the same. Similarly  $\mathcal{T}_{x, \bar{i}}$  and  $\mathcal{T}_{x, \varepsilon}$  specify decrementing transitions and transitions without counter updates.

Later we will also use (*singly succinct*) ECFGs, which are extended context-free grammars whose set of nonterminals is  $\Lambda^M$  where  $M$  is a unary encoded number. The set of productions is given in a suitable fashion by transducers. Let us remark that the precise definition of (doubly) succinct automata or grammars is not important for our paper, e.g. one could also use circuits instead of transducers to specify the transitions/productions.

## 4.2 Checking Dyck Inclusion for dsVASS

We prove our first technical contribution: an EXPSPACE procedure to check non-inclusion of a VASS language in a Dyck language. This involves checking if one of (OV), (DV), or (MV) occurs. We begin by showing how these violations can be detected for a (non-succinct) VASS.

To this end, first we show that offset-uniformity of a VASS language implies a doubly exponential bound  $B$  on the offset values for prefixes of accepted words (Theorem 4.1). Given an alphabet  $X$  and a number  $k \in \mathbb{N}$ , we define the language

$$\mathcal{B}(X, k) = \{w \in (X \cup \bar{X})^* \mid \text{for every prefix } v \text{ of } w: |\text{offset}(v)| \leq k\}.$$

► **Theorem 4.1.** *Let  $\mathcal{V}$  be a VASS with  $L(\mathcal{V}) \subseteq (X \cup \bar{X})^*$ . If  $L(\mathcal{V})$  is offset-uniform, then  $L(\mathcal{V}) \subseteq \mathcal{B}(X, 2^{2^{p(|\mathcal{V}|)}}$ ) for some polynomial function  $p$ .*

**Proof.** Let  $\mathcal{V} = (Q, X \cup \bar{X}, I, E, q_0, q_f)$  be a VASS where  $L(\mathcal{V}) \neq \emptyset$  is offset-uniform. The unique offset of  $L(\mathcal{V})$  is bounded double exponentially in  $|\mathcal{V}|$  since  $L(\mathcal{V})$  contains some word that is at most double exponentially long, a fact that follows from Rackoff's bound on covering runs [22]. Let  $C \subseteq Q \times \mathbb{M}[I]$  be the set of configurations that are reachable from  $(q_0, \mathbf{0})$  and from which the final state can be reached. Observe that for any configuration  $c \in C$  the language  $L(c) = \{w \in (X \cup \bar{X})^* \mid \exists \mathbf{u}: c \xrightarrow{w} (q_f, \mathbf{u})\}$  is also offset-uniform since  $L(c) \subseteq \{w \in (X \cup \bar{X})^* \mid vw \in L(\mathcal{V})\}$  where  $v \in (X \cup \bar{X})^*$  is any word with  $(q_0, \mathbf{0}) \xrightarrow{v} c$ . Define the function  $f: C \rightarrow \mathbb{Z}$  where  $f(c)$  is the unique offset of the words in  $L(c)$ . It remains to show that  $|f(c)|$  is bounded double exponentially for all  $c \in C$ .

Let  $M$  be the set of all configurations from which the final state can be reached (hence  $C \subseteq M$ ). Consider the following order on VASS configurations  $Q \times \mathbb{M}[I]$ :  $(q, \mathbf{u}) \leq (q', \mathbf{u}')$  iff  $q = q'$  and  $\mathbf{u}(i) \leq \mathbf{u}'(i)$  for each  $i \in I$ . The cardinality of the set  $\min(M)$  of minimal elements in  $M$  with respect to this order is bounded doubly exponentially in the size of  $\mathcal{V}$ . This follows directly from the fact that Rackoff's doubly-exponential bound [22] on the length of a covering run does not depend on the start configuration (but only the size of the VASS and the final configuration). An explicit bound for  $|\min(M)|$  is given in [4, Theorem 2].

Observe that if  $c_1 \in M$  and  $c_2 \in C$  with  $c_1 \leq c_2$  then  $L(c_1) \subseteq L(c_2)$  and therefore  $L(c_1)$  is also offset-uniform, having the same offset as  $L(c_2)$ . Hence, if for two configurations  $c_1, c_2 \in C$  there exists a configuration  $c \in M$  with  $c \leq c_1$  and  $c \leq c_2$ , then  $f(c_1) = f(c_2)$ . Since for every  $c_2 \in C$  there exists  $c_1 \in \min(M)$  with  $c_1 \leq c_2$ , the function  $f$  can only assume doubly exponentially many values on  $C$ .

Finally, we claim that  $f(C) \subseteq \mathbb{Z}$  is an interval containing 0, which proves that the norms of elements in  $f(C)$  are bounded by the number of different values, i.e., double exponentially. Since we assumed  $L(\mathcal{V}) \neq \emptyset$ , some final configuration  $(q_f, \mathbf{u}) \in C$  is reachable from  $(q_0, \mathbf{0})$ , and therefore  $0 \in f(C)$  since  $\varepsilon \in L((q_f, \mathbf{u}))$ . Consider the configuration graph  $\mathcal{C}$  of  $\mathcal{V}$  restricted to  $C$ . For any edge  $c_1 \rightarrow c_2$  in  $\mathcal{C}$  we have  $|f(c_1) - f(c_2)| \leq 1$  since VASS transitions consume at most one input symbol. Moreover, the underlying undirected graph of  $\mathcal{C}$  is connected since any configuration is reachable from  $(q_0, \mathbf{0}) \in C$ . Therefore  $f(C)$  is an interval, which concludes the proof. ◀



Note that although  $\mathcal{B}(X, k)$  is a regular language for each  $X$  and  $k$ , Theorem 4.1 does not imply that every offset-uniform VASS language is regular. For example, the VASS language  $\{(x\bar{x})^m(y\bar{y})^n \mid m \geq n\}$  is offset-uniform, but it is not regular. This is because Theorem 4.1 only implies boundedness of the number of occurrences of letters in the input words, but the VASS's own counters might be unbounded.

The main consequence of Theorem 4.1 is that in a VASS we can track the offset using a doubly succinct control state. Thus, we have the following corollary.

► **Corollary 4.2.** *The following problems can be decided in EXPSPACE: Given a VASS or dsVASS  $\mathcal{V}$ , does  $\text{offset}(w) = 0$  hold for all  $w \in L(\mathcal{V})$ ?*

**Proof.** First assume  $\mathcal{V}$  is a VASS. We show that the problem can be reduced to the intersection non-emptiness problem for a VASS and a doubly succinct NFA, i.e., given a VASS  $\mathcal{V}$  and a doubly succinct NFA  $\mathcal{A}$ , is the intersection  $L(\mathcal{V}) \cap L(\mathcal{A})$  nonempty? One can construct in polynomial time a doubly succinct VASS for  $L(\mathcal{V}) \cap L(\mathcal{A})$ , as a product construction between  $\mathcal{V}$  and  $\mathcal{A}$ . Since the emptiness problem for dsVASS is in EXPSPACE ([2, Theorem 5.1]), we can also decide emptiness of  $L(\mathcal{V}) \cap L(\mathcal{A})$  in EXPSPACE.

Define the number  $M = 2^{2^{p(|\mathcal{V}|)}}$  where  $p$  is the polynomial from Theorem 4.1. Let  $K_0 = \{w \in (X \cup \bar{X})^* \mid \text{offset}(w) = 0\}$ . According to Theorem 4.1, we have  $L(\mathcal{V}) \subseteq K_0$  if and only if  $L(\mathcal{V}) \subseteq K_0 \cap \mathcal{B}(X, M)$ . By the remarks above, it suffices to construct a doubly succinct NFA for the complement of  $K_0 \cap \mathcal{B}(X, M)$ . The following doubly succinct deterministic finite automaton  $\mathcal{A}$  recognizes  $K_0 \cap \mathcal{B}(X, M)$ : Given an input word over  $X \cup \bar{X}$ , the automaton tracks the current offset in the interval  $[-M, M]$ , stored in the control state as a binary encoding of length  $\log M = 2^{p(|\mathcal{V}|)}$  together with a bit indicating the sign. If the absolute value of the offset exceeds  $M$ , the automaton moves to a rejecting sink state. The state representing offset 0 is the initial and the only final state. Finally, we complement  $\mathcal{A}$  to obtain a doubly succinct NFA  $\bar{\mathcal{A}}$ , with a unique final state, for the complement of  $K_0 \cap \mathcal{B}(X, M)$ .

Now assume  $\mathcal{V}$  is a dsVASS. Using Lipton's construction simulating doubly exponential counter values [18], we can construct a (conventional) VASS  $\mathcal{V}'$ , size polynomial in  $|\mathcal{V}|$ , with the same language (similar to [2, Theorem 5.1]). We can now apply the above construction. ◀

Next, we check for (DV) or (MV), assuming offset uniformity. We will reduce both kinds of violations to the problem of searching for *marked Dyck factors*. A word of the form  $u\#v\#w$  is called a *marked Dyck factor* if  $u, v, w \in \{x, \bar{x}\}^*$  and  $v \in \text{Dyck}_x$ .

Intuitively, if a (DV) occurs in a word  $w$ , there is a first time that the offset reaches  $-1$ . Placing a  $\bar{\#}$  at the place where this happens, and a  $\#$  right at the beginning, we have a word of the form  $\#u\bar{\#}v$  where  $u \in \text{Dyck}_x$ . Similarly for (MV), we replace two letters  $z \in X$  and  $\bar{y} \in \bar{X}$  with  $z \neq y$  by  $\#$  and  $\bar{\#}$ , respectively, and look for a word  $u\#v\bar{\#}w$ , where  $v \in \text{Dyck}_x$ .

► **Proposition 4.3.** *The following problems can be decided in EXPSPACE: Given an offset-uniform VASS or dsVASS  $\mathcal{V}$ , does  $L(\mathcal{V})$  contain a marked Dyck factor?*

**Proof.** As in Corollary 4.2, given a dsVASS, we can convert to a polynomial-sized VASS with the same language and apply the following algorithm.

We again reduce to the intersection nonemptiness problem between a VASS and a doubly succinct NFA, and use the fact that nonemptiness of dsVASS is in EXPSPACE [2, Theorem 5.1]. As above, define the number  $M = 2^{2^{p(|\mathcal{V}|)}}$  where  $p$  is the polynomial from Theorem 4.1. The automaton keeps track of the offset and also verifies that the input has the correct format  $u\#v\bar{\#}w$  where  $u, v, w \in \{x, \bar{x}\}^*$ . Furthermore, upon reaching  $\#$  it starts tracking the current offset and verifies that (i) the offset stays nonnegative, (ii) the offset never exceeds  $2M$ ,

## 110:10 Checking Refinement of Asynchronous Programs

and (iii) the offset is zero when reaching  $\bar{\#}$ . If  $L(\mathcal{V})$  intersects  $L(\mathcal{A})$ , then clearly  $\mathcal{V}$  is a positive instance of the problem. Conversely, assume that  $L(\mathcal{V})$  contains a word  $u\#v\bar{\#}w$  with  $v \in \text{Dyck}_x$ . By offset-uniformity of  $\mathcal{V}$  and by Theorem 4.1, each prefix  $v'$  of  $v$  satisfies  $\text{offset}(v') = \text{offset}(uv') - \text{offset}(u) \leq M - (-M) = 2M$ . Therefore  $u\#v\bar{\#}w \in L(\mathcal{A})$ .  $\blacktriangleleft$

Let us put everything together. Let  $\rho: (X \cup \bar{X})^* \rightarrow \{x, \bar{x}\}^*$  be the morphism that replaces all letters from  $X$  (resp.,  $\bar{X}$ ) by the letter  $x$  (resp.,  $\bar{x}$ ). Given a dsVASS  $\mathcal{V}$  over  $X \cup \bar{X}$  we can construct in polynomial time three dsVASS  $\mathcal{V}_o, \mathcal{V}_d, \mathcal{V}_m$  where

$$\begin{aligned} L(\mathcal{V}_o) &= \rho(L(\mathcal{V})), \\ L(\mathcal{V}_d) &= \{\#\rho(v)\bar{\#}\rho(\bar{y}w) \mid v\bar{y}w \in L(\mathcal{V}) \text{ for some } v, w \in (X \cup \bar{X})^*, y \in X\}, \\ L(\mathcal{V}_m) &= \{\rho(u)\#\rho(v)\bar{\#}\rho(w) \mid u\bar{y}v\bar{z}w \in L(\mathcal{V}) \text{ for some } u, v, w \in (X \cup \bar{X})^*, y \neq z \in X\}. \end{aligned}$$

Observe that  $L(\mathcal{V}) \subseteq \text{Dyck}_x$  if and only if  $L(\mathcal{V}_o)$  has uniform offset 0 and  $L(\mathcal{V}_d)$  and  $L(\mathcal{V}_m)$  do not contain marked Dyck factors.

Hence, to decide whether  $L(\mathcal{V}) \subseteq \text{Dyck}_X$  we first test that  $L(\mathcal{V}_o)$  has uniform offset 0, using Corollary 4.2, rejecting if not. Otherwise, we can apply Proposition 4.3 to test whether  $L(\mathcal{V}_d)$  or  $L(\mathcal{V}_m)$  contain marked Dyck factors. If one of the tests is positive, we know  $L(\mathcal{V}) \not\subseteq \text{Dyck}_X$ , otherwise  $L(\mathcal{V}) \subseteq \text{Dyck}_X$ .

**► Theorem 4.4.** *Given a dsVASS  $\mathcal{V}$  over the alphabet  $X \cup \bar{X}$ , checking whether  $L(\mathcal{V}) \subseteq \text{Dyck}_X$  is EXPSPACE-complete.*

Let us remark that Theorem 4.4 can also be phrased slightly more generally. Above, we have defined the language of a VASS to be the set of input words for which a final state is reached. Such languages are also called *coverability languages*. Another well-studied notion is the *reachability language* of a VASS, which consists of those words for which a configuration  $(q_f, \mathbf{0})$  is reached. Moreover, a VASS is *deterministic* if for each input letter  $x$  and each state  $q$ , there is at most one  $x$ -labeled transition starting in  $q$  (and there are no  $\varepsilon$ -transitions). We can now phrase Theorem 4.4 as follows: Given a VASS coverability language  $K$  and a reachability language  $L$  of a deterministic VASS, it is EXPSPACE-complete to decide whether  $K \subseteq L$ . This is in contrast to inclusion problems where  $K$  is drawn from a subclass of the coverability languages: This quickly leads to Ackermann-completeness [6]. In fact, even if we replace  $\text{Dyck}_X$  in Theorem 4.4 with the set of prefixes of  $\text{Dyck}_{\{x\}}$ , the problem becomes Ackermann-complete (see the full version of this work).

## 5 Checking Dyck Inclusion for Programs

We now describe our algorithm for checking inclusion in  $\text{Dyck}_X$  for programs. Our argument is similar to the case of dsVASS: we first construct three auxiliary programs  $\mathcal{P}_o, \mathcal{P}_d$ , and  $\mathcal{P}_m$ , and then we use them to detect each type of violation in the original program. We construct the program  $\mathcal{P}_o$  for checking offset violation by projecting the Dyck letters to the one-dimensional Dyck alphabet  $\{x, \bar{x}\}$ . The programs  $\mathcal{P}_d$  and  $\mathcal{P}_m$  are constructed by first placing two markers like for VASS, and then projecting to  $\{x, \bar{x}\}$ .

As in the algorithm for VASS, we check whether  $L(\mathcal{P}_o)$  has uniform offset 0, and whether  $L(\mathcal{P}_d)$  and  $L(\mathcal{P}_m)$  contain marked Dyck factors. For these checks, we convert the three programs into dsVASS  $\mathcal{V}_o, \mathcal{V}_d$ , and  $\mathcal{V}_m$ , respectively, in such a way that violations are preserved. To be more precise, this conversion from programs to dsVASS will preserve the *downward closure* with respect to a specific order that we define below. The global downward closure procedure is obtained by composing a local downward closure procedure applied to each task.

On the task level, the order  $\sqsubseteq$  is a combination of the subword order on the handler names in  $\Gamma$  and the syntactic order of  $\text{Dyck}_X$  over the event letters. The core technical result is a transformation from context-free grammars into dsNFA which preserve the downward closure with respect to  $\sqsubseteq$ .

One key aspect of our downward closure construction is an important condition on the pumps that appear in the context-free grammar.

► **Definition 5.1.** *A context-free grammar  $\mathcal{G}$  is tame-pumping if for every pump  $A \xrightarrow{*} uAv$ , we have  $\text{offset}(u) \geq 0$  and  $\text{offset}(v) = -\text{offset}(u)$ . A derivation  $A \xrightarrow{*} uAv$  is called an increasing pump if  $\text{offset}(u) > 0$ , otherwise it is called a zero pump. An asynchronous program is tame-pumping if its grammar is tame-pumping.*

Note that while our definition of a tame-pumping grammar is syntactic, it actually only depends on the generated language, assuming every nonterminal occurs in a derivation: In that case, a grammar is tame-pumping if and only if (i) the set of offsets and (ii) the set of dips of words in its language are both finite.

The following lemma summarizes some properties of tame-pumping and why it is useful for our algorithm. The proof can be found in the full version.

► **Lemma 5.2.**

1. *We can check in coNP whether a given context-free grammar over  $\{x, \bar{x}\}$  is tame-pumping. Furthermore, given a nonterminal  $A_0$ , we can check in NP whether  $A_0$  has a zero pump (resp., increasing pump).*
2. *There exists a polynomial  $p$  such that, if  $\mathcal{G}$  is tame-pumping, then for every nonterminal  $A$  of  $\mathcal{G}$  and every  $w \in L(\mathcal{G}, A)$  we have  $\text{dip}(w) \leq 2^{p(|\mathcal{G}|)}$ .*
3. *If  $\mathcal{P}$  does not have tame-pumping, then  $L(\mathcal{P}) \not\subseteq \text{Dyck}_X$ .*

Thus, if  $\mathcal{P}$  is not tame-pumping, the refinement checking algorithm rejects immediately. From now on, we assume that  $\mathcal{P}$  is tame-pumping.

## 5.1 Combining the subword order and the syntactic order

Suppose  $\Gamma$  is an alphabet and let  $\Theta = \Gamma \cup \{x, \bar{x}\}$ . Define  $\bar{a} = a$  for  $a \in \Gamma$ . By  $\preceq$ , we denote the *subword ordering* on  $\Gamma^*$ , i.e.  $u \preceq v$  if and only if  $u$  can be obtained from  $v$  by deleting some letters. Formally there exist words  $u_1, \dots, u_n, v_0, \dots, v_n \in \Gamma^*$  such that  $u = u_1 \cdots u_n$  and  $v = v_0 u_1 v_1 \cdots u_n v_n$ . For  $u, v \in \{x, \bar{x}\}^*$ , we write  $u \trianglelefteq v$  if  $\text{offset}(u) = \text{offset}(v)$  and  $\text{dip}(u) \geq \text{dip}(v)$ . In fact,  $\trianglelefteq$  is the *syntactic order* with respect to the Dyck language, i.e. if  $u \trianglelefteq v$  and  $rus \in \text{Dyck}_x$  then  $rvs \in \text{Dyck}_x$  for all  $r, s$ . We define the ordering  $\sqsubseteq'$  on  $\Theta^*$  by  $z_1 \sqsubseteq' z_2$  if and only if  $\pi_{x, \bar{x}}(z_1) \trianglelefteq \pi_{x, \bar{x}}(z_2)$ , and  $\pi_\Gamma(z_1) \preceq \pi_\Gamma(z_2)$ . For example,  $a\bar{x}xc \sqsubseteq' abc\bar{x}$  because  $ac$  is a subword of  $abc$ , and both  $\bar{x}x$  and  $x\bar{x}$  have offset 0, but  $\bar{x}x$  has a larger dip.

Let  $\#, \bar{\#}$  be two fresh letters, called *markers*. The set of *marked words* is defined as

$$\mathcal{M} = \Theta^* \{\varepsilon, \#\} \Theta^* \{\varepsilon, \bar{\#}\} \Theta^*.$$

A marked word should be viewed as an infix of a larger word  $u\#v\bar{\#}w$ . The set of *admissible marked words*, denoted by  $\mathcal{A}$ , consists of those words  $z \in \mathcal{M}$  which are an infix of a word  $u\#v\bar{\#}w$  where  $v \in \text{Dyck}_x$ . For example, a marked word  $u\#v$  is admissible if  $v$  is a prefix of a Dyck word.

On the set of admissible marked words, we define an ordering  $\sqsubseteq$ . To do so, we first define for each marked word  $z \in \mathcal{M}$  two words  $\text{inside}(z)$  and  $\text{outside}(z)$  in  $\Theta^*$  as follows: Let  $u, v, w \in \Theta^*$  such that either  $z = v$ ,  $z = u\#v$ ,  $z = v\bar{\#}w$ , or  $z = u\#v\bar{\#}w$ . Then we define  $\text{inside}(z) = v$  and  $\text{outside}(z) = uw$  (here,  $u = \varepsilon$  if it is not part of  $z$ , same for  $w$ ). Given

## 110:12 Checking Refinement of Asynchronous Programs

two admissible marked words  $z_1, z_2 \in \mathcal{A}$  we define  $w \sqsubseteq w'$  if and only if  $z_1$  and  $z_2$  contain the same markers, and  $\text{inside}(z_1) \sqsubseteq' \text{inside}(z_2)$ , and  $\text{outside}(z_1) \sqsubseteq' \text{outside}(z_2)$ . For example,  $a\bar{x}xc\#a \sqsubseteq xabc\bar{x}\#ab$  because  $a\bar{x}xc \sqsubseteq' xabc\bar{x}$  and  $a \sqsubseteq' ab$ .

For a language  $L \subseteq \mathcal{M}$  we denote by  $L\downarrow$  the downward closure of  $L$  within  $\mathcal{A}$  with respect to the ordering  $\sqsubseteq$ . Thus, we define:

$$L\downarrow = \{u \in \mathcal{A} \mid \exists v \in L \cap \mathcal{A} : u \sqsubseteq v\}.$$

► **Theorem 5.3.** *Given a tame-pumping CFG  $\mathcal{G}$ , we can compute in polynomial space a doubly succinct NFA  $\mathcal{A}$  such that  $L(\mathcal{A})\downarrow = L(\mathcal{G})\downarrow$  and  $|\mathcal{A}|$  is polynomially bounded in  $|\mathcal{G}|$ .*

We explain how to prove Theorem 5.3 in Section 6. Let us make a few remarks. While downward closed sets with respect to the subword ordering are always regular, this does not hold for  $\sqsubseteq$ . Consider the language  $L = (ax)^*$  where  $a \in \Gamma$  is a handler name and  $x \in X$  is an event letter. Then  $L\downarrow$  consists of all words  $w \in \{a, x, \bar{x}\}^*$  where  $|w|_a \leq |w|_x - |w|_{\bar{x}}$ , which is not a regular language. Furthermore, the automaton in Theorem 5.3 may indeed require double exponentially many states. For example, given a number  $n$ , consider the language  $L = \{u\bar{x}^{2^n}\#x^{2^n}\bar{u} \mid u \in \{ax, bx\}^*\}$  where  $\Gamma = \{a, b\}$  is the set of handler names and  $X = \{x\}$ . Here we define  $\bar{a}_1\bar{a}_2\cdots\bar{a}_n = \bar{a}_n\cdots\bar{a}_2\bar{a}_1$  for a word  $a_1\cdots a_n \in \{a, b, x\}^*$  where  $\bar{a} = a$  and  $\bar{b} = b$ . This is generated by a tame-pumping context-free grammar of size linear in  $n$ . However, for any  $\mathcal{A}$  with  $L(\mathcal{A})\downarrow = L\downarrow$ , projecting to just  $a$  and  $b$  yields the language  $K = \{uu^{\text{rev}} \mid u \in \{a, b\}^*, |u| \leq 2^n\}\downarrow$ , for which an NFA requires at least  $2^{2^n}$  states.

Finally, note that the restriction to admissible words is crucial: If we defined the ordering  $\sqsubseteq$  on all words of  $\mathcal{M}$ , then for the tame-pumping language  $L = \{x^n\#\bar{x}^n \mid n \in \mathbb{N}\}$ , the downward closure would not be regular, because an NFA would be unable to preserve the unbounded offset at the separator  $\#$ . A key observation in this work is that in combination with tame pumping, admissibility guarantees that the offset at the borders  $\#$  and  $\bar{\#}$  is bounded (see Lemma 6.7), which enables a finite automaton to preserve it.

Given a tame-pumping asynchronous program  $\mathcal{P}$ , we can now compute a dsVASS  $\mathcal{V}$  with the same downward closure: Its counters are the handler names  $a \in \Gamma$  in  $\mathcal{P}$ . For each nonterminal  $A$  we apply Theorem 5.3 to  $\mathcal{G}_A$ , which is the grammar of  $\mathcal{P}$  with start symbol  $A$ , and obtain a dsNFA  $\mathcal{B}_A$ . We replace each transition  $q \xrightarrow{a, A} q'$  by the following gadget: First, it decrements the counter for the handler name  $a$ . Next, the gadget simulates the dsNFA  $\mathcal{B}_A$  where handlers  $b \in \Gamma$  are interpreted as counter increments. Finally, when reaching the final state of  $\mathcal{B}_A$  we can non-deterministically switch to  $q'$ .

► **Corollary 5.4.** *Given an asynchronous program  $\mathcal{P}$  with tame-pumping, we can compute in polynomial space a doubly succinct VASS  $\mathcal{V}$  such that  $L(\mathcal{P})\downarrow = L(\mathcal{V})\downarrow$  and  $|\mathcal{V}|$  is polynomially bounded in  $|\mathcal{P}|$ .*

The details of the proof are given in the full version.

## 5.2 The algorithm

We are now ready to explain the whole algorithm. Given an asynchronous program  $\mathcal{P} = (Q, X \cup \bar{X}, \Gamma, \mathcal{G}, \Delta, q_0, q_f, \gamma_0)$ , we want to check if  $L(\mathcal{P}) \subseteq \text{Dyck}_X$ . Recall that, wlog, we can assume all nonterminals are useful, meaning every nonterminal is involved in some accepting run. The algorithm is presented in Algorithm 1. As a first step, the algorithm verifies that  $\mathcal{P}$  is tame-pumping using Lemma 5.2. Next we construct the following auxiliary asynchronous programs  $\mathcal{P}_o, \mathcal{P}_d, \mathcal{P}_m$ , to detect offset, dip, and mismatch violations in  $L(\mathcal{P})$ .

■ **Algorithm 1** Checking non-inclusion of  $L(\mathcal{P})$  in the Dyck language  $\text{Dyck}_X$  in EXPSPACE.

- 
- 1 Asynchronous program  $\mathcal{P}$  for a language  $L \subseteq (X \cup \bar{X})^*$
  - 2 **if**  $\mathcal{P}$  does not have tame-pumping (Lemma 5.2) **then return**  $L \not\subseteq \text{Dyck}_X$ ;
  - 3 Construct asynchronous programs  $\mathcal{P}_o, \mathcal{P}_d, \mathcal{P}_m$  (Equation (1)).
  - 4 Construct dsVASS  $\mathcal{V}_o, \mathcal{V}_d, \mathcal{V}_m$  with  $L(\mathcal{V}_x)\downarrow = L(\mathcal{P}_x)\downarrow$  for  $x \in \{o, d, m\}$  (Corollary 5.4).
  - 5 **if**  $\mathcal{V}_o$  does not have uniform offset 0 (Corollary 4.2) **then return**  $L \not\subseteq \text{Dyck}_X$ ;
  - 6 **if**  $L(\mathcal{V}_d)$  or  $L(\mathcal{V}_m)$  contains a marked Dyck factor (Proposition 4.3) **then return**  
 $L \not\subseteq \text{Dyck}_X$ ;
  - 7 **return**  $L \subseteq \text{Dyck}_X$
- 

Let  $\rho: (X \cup \bar{X})^* \rightarrow \{x, \bar{x}\}^*$  be the morphism which replaces all letters in  $X$  by unique letter  $x$  and all letters in  $\bar{X}$  by unique letter  $\bar{x}$ . The programs  $\mathcal{P}_o, \mathcal{P}_d, \mathcal{P}_m$  recognize the following languages over the alphabet  $\{x, \bar{x}, \#, \bar{\#}\}$ :

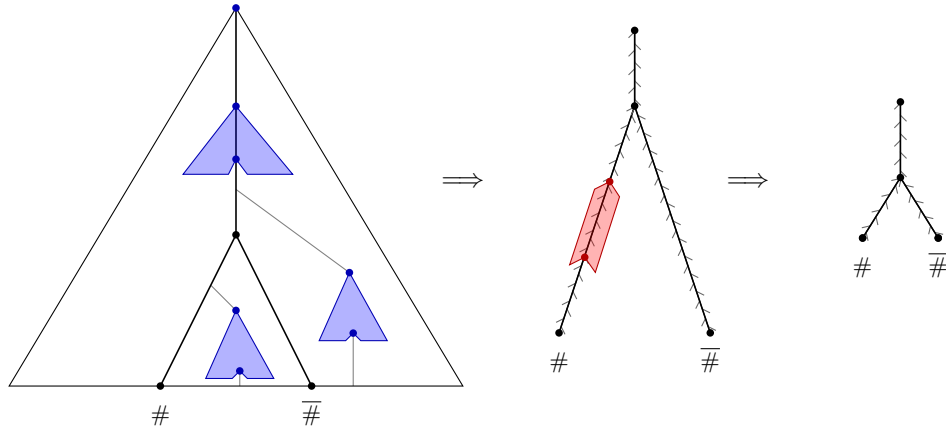
$$\begin{aligned}
 L(\mathcal{P}_o) &= \{\rho(w) \mid w \in L(\mathcal{P})\}, \\
 L(\mathcal{P}_d) &= \{\#\rho(v)\bar{\#}\rho(\bar{y}w) \mid v\bar{y}w \in L(\mathcal{P}) \text{ for some } v, w \in (X \cup \bar{X})^*, y \in X\}, \\
 L(\mathcal{P}_m) &= \{\rho(u)\bar{\#}\rho(v)\bar{\#}\rho(w) \mid uv\bar{z}w \in L(\mathcal{P}), \\
 &\quad \text{for some } u, v, w \in (X \cup \bar{X})^*, y \neq z \in X\}.
 \end{aligned} \tag{1}$$

In fact, if the original asynchronous program  $\mathcal{P}$  is tame-pumping, we can ensure that  $\mathcal{P}_o, \mathcal{P}_d, \mathcal{P}_m$  are also tame-pumping (see the full version for details).

It remains to verify whether  $L(\mathcal{P}_o)$  has uniform offset 0, and  $L(\mathcal{P}_d)$  and  $L(\mathcal{P}_m)$  do not contain marked Dyck factors. By Corollary 5.4 we can compute for each  $x \in \{o, d, m\}$  a dsVASS  $\mathcal{V}_x$  with  $L(\mathcal{V}_x)\downarrow = L(\mathcal{P}_x)\downarrow$ . Since  $\sqsubseteq$  preserves offsets we know that  $L(\mathcal{P}_o)$  has uniform offset 0 if and only if  $L(\mathcal{V}_o)$  has uniform offset 0, which can be decided in exponential space by Corollary 4.2. Finally, we check whether  $L(\mathcal{V}_d)$  or  $L(\mathcal{V}_m)$  contain a marked Dyck factor by Proposition 4.3. This is correct, because a language  $L$  contains a marked Dyck factor if and only if  $L\downarrow$  contains a marked Dyck factor: On the one hand, the “only if” direction is clear because  $L \subseteq L\downarrow$ . On the other hand, if  $u\bar{\#}v\bar{\#}w \in L\downarrow$  is a marked Dyck word then there exists a word  $u'\bar{\#}v'\bar{\#}w' \in L$  with  $v \preceq v'$ , and therefore  $v' \in \text{Dyck}_x$ .

## 6 Computing Downward Closures and the Proof of Theorem 5.3

It remains to show how the automaton  $\mathcal{A}$  for the downward closure in Theorem 5.3 is constructed. As a warm-up, let us illustrate how to construct from a context-free grammar  $\mathcal{G}$  an NFA  $\mathcal{A}$  for the subword closure of  $L(\mathcal{G})$ , cf. [5]. Here, *subword closure* refers to the downward closure with respect to the subword ordering  $\preceq$ . Notice that this is a special case of Theorem 5.3, namely where  $L(\mathcal{G}) \subseteq \Gamma^*$ . The basic idea is that every derivation tree of  $\mathcal{G}$  can be obtained by inserting pumps into a *skeleton* – a derivation tree without vertical repetitions of nonterminals. The skeleton can be guessed by an (exponentially large) automaton  $\mathcal{A}$  and the effects of pumps are abstracted as follows: For each nonterminal  $A$  one can compute the subalphabets  $\Gamma_{A,L}, \Gamma_{A,R} \subseteq \Gamma$  containing all letters occurring on the left side  $u$  and the right side  $v$  of a pump  $A \xrightarrow{*} uAv$ . Instead of inserting pumps, the automaton for the subword closure inserts arbitrary words  $u' \in \Gamma_{A,L}^*$  and  $v' \in \Gamma_{A,R}^*$  on the left or right side of  $A$ , respectively. This is sufficient because for any word  $w$ , the subword closure of the language  $w^*$  contains exactly those words that consist only of letters present in  $w$ .



■ **Figure 1** Abstracting undivided pumps (in blue) and divided pumps (in red).

The difficulty in proving Theorem 5.3 is to preserve, not only the subword closure, but also the downward closure with respect to the syntactic order  $\preceq$  on the letters  $\{x, \bar{x}\}$ . To do so, we need to distinguish between two types of pumps. Consider the derivation tree for a marked word  $z = u\#v\bar{\#}w$ , depicted left in Figure 1. Observe that removing one of the three pumps in blue does not change the offset of  $\text{inside}(z) = v$  or  $\text{outside}(z) = uw$ , because  $\mathcal{G}$  is tame-pumping. Such pumps, which are completely contained in  $\text{inside}(z)$  or  $\text{outside}(z)$ , will be called *undivided*. However, one needs to be more careful when removing *divided* pumps, e.g., the red pump in the second derivation tree of Figure 1. Removing the red pump decreases the offset of  $\text{outside}(z)$ , while increasing the offset of  $\text{inside}(z)$  by the same amount.

We will proceed in two transformations, which preserve the downward closure w.r.t.  $\sqsubseteq$ . In the first transformation we obtain a grammar whose derivation trees do not contain any undivided pumps. In the second step we additionally eliminate divided pumps.

## 6.1 Abstracting undivided pumps

Recall that  $\mathcal{M} = \Theta^*\{\#, \varepsilon\}\Theta^*\{\bar{\#}, \varepsilon\}\Theta^*$  where  $\Theta = \Gamma \cup \{x, \bar{x}\}$ . In the following we only consider *uniformly marked* grammars  $\mathcal{G}$ , that is, we assume  $L(\mathcal{G})$  is contained in one of the subsets  $\Theta^*\#\Theta^*\bar{\#}\Theta^*$ ,  $\Theta^*\#\Theta^*$ ,  $\Theta^*\bar{\#}\Theta^*$ , or  $\Theta^*$ . This is not a restriction since we can split the given grammar  $\mathcal{G}$  into four individual grammars, covering the four types of marked words, and treat them separately. This allows us to partition the set of nonterminals  $N$  into  $N_{\#\bar{\#}} \cup N_{\#} \cup N_{\bar{\#}} \cup N_0$  where  $N_{\#\bar{\#}}$ -nonterminals only produce marked words in  $\Theta^*\#\Theta^*\bar{\#}\Theta^*$ ,  $N_{\#}$ -nonterminals only produce marked words in  $\Theta^*\#\Theta^*$ , etc. A pump  $A \xrightarrow{*} uAv$  is *undivided* if  $A \in N_{\#\bar{\#}} \cup N_0$ , and *divided* otherwise. Our first goal will be to eliminate undivided pumps. A derivation tree without undivided pumps may still contain exponentially large subtrees below  $N_0$ -nonterminals. Such subtrees will also be “flattened” in this step, see the first transformation step in Figure 1.

► **Definition 6.1.** A context-free grammar  $\mathcal{G} = (N, \Theta \cup \{\#, \bar{\#}\}, P, S)$  is almost-pumpfree iff  
 (C1)  $\mathcal{G}$  does not have undivided pumps, and  
 (C2) for all productions  $A \rightarrow \alpha$  with  $A \in N_0$  either  $\alpha = a \in \Theta$  or  $\alpha = (\Gamma')^*$  for some  $\Gamma' \subseteq \Gamma$ .

We will now explain how to turn any uniformly marked CFG into an almost-pumpfree one. The resulting (extended) grammar will be exponentially large but can be represented succinctly. Recall that a *succinct ECFG* (sECFG) is an extended context-free grammar



$\mathcal{G}$  whose nonterminals are polynomially long strings and whose productions are given by finite-state transducers. For example, one of the transducers accepts the finite relation of all triples  $(A, B, C)$  such that there exists a production  $A \rightarrow BC$ . Productions either adhere to Chomsky normal form or have the form  $A \rightarrow B$ . The latter enables us to simulate PSPACE-computations in the grammar without side effects, see Observation 6.5 below.

► **Proposition 6.2.** *Given a uniformly marked tame-pumping CFG  $\mathcal{G}$ , one can compute in polynomial space a tame-pumping almost-pumpfree sECFG  $\mathcal{G}'$  such that  $L(\mathcal{G})\downarrow = L(\mathcal{G}')\downarrow$  and  $|\mathcal{G}'|$  is polynomially bounded in  $|\mathcal{G}|$ .*

To prove Proposition 6.2, we first need some auxiliary results, which are mainly concerned with computing the minimal dips and letter occurrences within undivided pumps of a grammar  $\mathcal{G}$ . Recall that for the subword closure we computed for each nonterminal  $A$  the subalphabets  $\Gamma_{L,A}$  and  $\Gamma_{R,A}$ , and inserted arbitrary words over  $\Gamma_{L,A}$  and  $\Gamma_{R,A}$  left and right to the nonterminal  $A$ . For the refined order  $\sqsubseteq$  we may only use a letter  $a \in \Gamma$  after simulating the minimal dip which is required to produce the letter  $a$ .

For a word  $w \in \Theta^*$  we define the set  $\psi(w)$  of all pairs  $(n, m) \in \mathbb{N}^2$  such that  $n \geq \text{dip}(w)$  and  $m = n + \text{offset}(w)$ . In other words,  $\psi(w)$  is the reachability relation induced by  $w$ , interpreted as counter instructions. Recall that *Presburger arithmetic* is the first-order theory of  $(\mathbb{N}, +, <, 0, 1)$ . As an auxiliary step, we will compute existential Presburger formulas capturing the relation  $\psi(u) \times \psi(v)$  for all pumps  $A \xrightarrow{\exists} uAv$  of a nonterminal  $A$ .

In the following lemma, when we say that we can *compute a formula for a relation*  $R \subseteq \mathbb{N}^k$  *in polynomial space*, we mean that there is non-deterministic polynomial-space algorithm, where each non-deterministic branch computes a polynomial-size formula for a relation  $R_i$  such that if  $R_1, \dots, R_n$  are the relations of all the branches, then  $R = \bigcup_{i=1}^n R_i$ . Here we tacitly use the fact that  $\text{NPSPACE} = \text{PSPACE}$  [24].

► **Lemma 6.3.** *Given an offset-uniform CFG with  $L(\mathcal{G}) \subseteq \Theta^*\$ \Theta^*$ , where  $\$ \notin \Theta$ , we can compute in polynomial space an existential Presburger formula for the relation*

$$\bigcup_{u\$v \in L} \psi(u) \times \psi(v) \subseteq \mathbb{N}^4.$$

**Proof sketch.** The result of Lemma 6.3 was already proved in [1, Proposition 3.8], under the additional assumption that the given context-free grammar  $\mathcal{G}$  for  $L$  is *annotated* (they even show that in this case the formula can be computed in NP). We call  $\mathcal{G}$  annotated if for every nonterminal  $A$  the minimal dip that can be achieved by a word in  $L(\mathcal{G}, A)$  is given as an input, denoted by  $\text{mindip}(A)$ . Hence, it remains to show how to compute the annotation of an offset-uniform grammar in PSPACE, which is possible using a simple saturation algorithm. For each nonterminal  $A$ , the algorithm stores a number  $D(A)$  satisfying  $D(A) \geq \text{mindip}(A)$ . Initially,  $D(A)$  is set to an upper bound for  $\text{mindip}(A)$ , which by Lemma 5.2 (2) can be chosen to be exponentially large in  $|\mathcal{G}|$ . In each round the function  $D$  is updated as follows: For each production  $A \rightarrow BC$  we set  $D(A)$  to the minimum of  $D(A)$  and  $\max\{D(B), D(C) - \text{offset}(B)\}$ , where  $\text{offset}(B)$  is the unique offset of  $L(\mathcal{G}, B)$ . Clearly, the algorithm can be implemented in polynomial space since the numbers are bounded exponentially. Termination of the algorithm is guaranteed since the numbers  $D(A)$  are non-increasing. ◀

With Lemma 6.3 in hand, we can now prove the following lemma, which allows us to check whether pumps with certain letter occurrences exist for certain minimal dips.

► **Lemma 6.4.** *Given a tame-pumping CFG  $\mathcal{G}$  such that  $L(\mathcal{G}) \subseteq \mathcal{M}$ , a nonterminal  $A$  in  $\mathcal{G}$ , a letter  $a \in \Gamma$  and two numbers  $d_L, d_R \in \mathbb{N}$ , we can decide in PSPACE if there exists a derivation  $A \xrightarrow{*} uAv$  such that  $u$  contains the letter  $a$  (or symmetrically, whether  $v$  contains the letter  $a$ ),  $\text{dip}(u) \leq d_L$ , and  $\text{dip}(v) \leq d_R$ . Furthermore, we can also decide in PSPACE whether a derivation with the above properties exists that also satisfies  $\text{offset}(u) > 0$ .*

**Proof sketch.** We first construct the CFG  $\mathcal{G}_A$  for the language of pumps of the nonterminal  $A$ , meaning for  $L(\mathcal{G}_A) = \{u\$v \mid A \xrightarrow{*}_{\mathcal{G}} uAv\}$ . Then we intersect with the regular language  $\Theta^*a\Theta^*\$ \Theta^*$ , and apply Lemma 6.3 to the resulting grammar. This is possible, because tame-pumping implies that the grammar for the pumps has a uniform offset of zero. We can modify the resulting Presburger formula from Lemma 6.3 to check for the required dips, and modify it further to check for the positive offset for  $u$ . Finally, we use the fact that testing satisfiability of an existential Presburger formula is in NP [3]. ◀

Now we are almost ready to prove Proposition 6.2. The last thing we need is for an sECFG to perform PSPACE-computations on paths in its derivation trees:

► **Observation 6.5.** *An sECFG can simulate PSPACE-computations on exponentially long paths in its derivation trees. This is because the nonterminals are polynomially long strings and can therefore act as polynomial space Turing tape configurations. Moreover, the transducers of the sECFG can easily be constructed to enforce the step-relation of a Turing machine. If we apply this enforcement to productions of the form  $A \rightarrow B$ , then the path that simulates the PSPACE-computation will not even have any additional side paths until after the computation is complete. Thus, only the result of the computation will affect the derived word.*

*Since grammars and transducers are non-deterministic (and  $\text{NPSPACE} = \text{PSPACE}$ ), we can even implement non-determinism and guessing within such computations.*

We are ready to present a proof sketch of Proposition 6.2. The main idea is that  $\mathcal{G}'$  simulates derivation trees of  $\mathcal{G}$  by keeping track of at most polynomially many nodes, and abstracting away pumps via the previous auxiliary results.

If a nonterminal  $A$  of  $\mathcal{G}$  does not belong to  $N_0$  (i.e., it produces a marker), then  $\mathcal{G}'$  guesses a production  $A \rightarrow BC$  to apply. If  $A$  furthermore belongs to  $N_{\#\#}$ , then  $\mathcal{G}'$  also guesses a pump to apply in the form of a 4-tuple consisting of two dip values  $d_L, d_R \in \mathbb{N}$  and two alphabets  $\Gamma_L, \Gamma_R \subseteq \Gamma$ . Guessing and storing the dip values is possible in PSPACE, since they are exponentially bounded by Lemma 5.2 (2). For each  $a \in \Gamma_L$ , Lemma 6.4 is used on input  $A, a, d_L, d_R$  to check in PSPACE whether a matching pump exists. A symmetric version of Lemma 6.4 is also used for each  $a \in \Gamma_R$ . Then, if all checks succeed,  $\mathcal{G}'$  simulates the pump as  $A \rightarrow \bar{x}^{d_L} x^{d_L} \Gamma_L^* BC \bar{x}^{d_R} x^{d_R} \Gamma_R^*$ . This simulation clearly preserves minimal dips and handler names, whereas by tame-pumping the combined offset of a pump is zero anyway, and therefore need not be computed.

If a nonterminal  $A$  belongs to  $N_0$ , then  $\mathcal{G}'$  abstracts away its entire subtree. To this end it generates a pumpfree subtree on-the-fly using depth-first search, which is possible in PSPACE since without pumps the tree has polynomial height. During this process pumps are simulated using the same strategy as before.

We also need to ensure that nonterminals of  $\mathcal{G}'$  in  $N_0$  only have productions that allow for a single leaf node below them. To this end  $\mathcal{G}'$  only ever derives letters and alphabets  $\Gamma^{/*}$  one at a time. Consider the up to two *main paths* in a derivation tree of  $\mathcal{G}'$ , by which we mean the paths leading from the root to a marker. Whenever  $\mathcal{G}'$  simulates a pump as  $A \rightarrow u'Av'$  in the above process, it extends the main path by  $|uv|$  and in each step only derives a single nonterminal from  $N_0$  to the left or right. When  $\mathcal{G}'$  abstracts an entire subtree of a nonterminal in  $N_0$ , then this subtree is also produced to the left or right of the main path, without leaving said path.

Additionally, whenever  $\mathcal{G}'$  simulates a pump of some  $A$ , then  $\mathcal{G}'$  assumes that this pump is the combination of all pumps that occur in the original derivation tree for that instance of  $A$ . Thus, below such a pump, it remembers in polynomial space, that  $A$  is not allowed to occur anymore. Finally, whenever  $\mathcal{G}'$  checks by Lemma 6.4 that a pump exists with  $\text{offset}(u) > 0$ , then this is a so-called increasing pump, and it can be repeated to achieve an infix with arbitrary high offset. Thus, dip values below this pump cannot make up for this offset and therefore will no longer be simulated.

## 6.2 Abstracting divided pumps

We have now removed all the undivided pumps and are left with derivation trees as in the middle picture of Figure 1. In this subsection, we will show the following:

► **Lemma 6.6.** *Given a tame-pumping almost-pumpfree sECFG  $\mathcal{G}$  with  $L(\mathcal{G}) \subseteq \mathcal{M}$ , one can construct in polynomial space a dsNFA  $\mathcal{B}$  such that  $L(\mathcal{B})\downarrow = L(\mathcal{G})\downarrow$  and  $|\mathcal{B}|$  is polynomially bounded in  $|\mathcal{G}|$ .*

We give a proof sketch here, the details can be found in the full version of the paper. Our starting point in the proof of Lemma 6.6 is the following key observation: The offsets which occur during the production of any *admissible* marked word  $w$  which contains exactly one marker are bounded. This allows us to keep track of the offset precisely, which is necessary for us to solve the marked Dyck factor (MDF) problem.

For a node  $t$  in a derivation tree  $T$ , let  $w(t)$  denote the word derived by the subtree rooted at  $t$  and let  $u(t) = \text{inside}(w(t))$ ,  $v(t) = \text{outside}(w(t))$ .

► **Lemma 6.7.** *There exists a polynomial  $p$  such that for any uniformly marked, tame-pumping, almost-pumpfree sECFG  $\mathcal{G}$  the following holds. Let  $T$  be a derivation tree of  $\mathcal{G}$  which produces an admissible marked word containing  $\#$  or  $\bar{\#}$ , but not both. Then we have  $|\text{offset}(u(t))|, |\text{offset}(v(t))| \leq 2^{p(|\mathcal{G}|)}$ .*

**Proof.** We consider the case when the word derived is of the form  $u\#v$ , the case for  $v\bar{\#}w$  being symmetric. Our derivation tree  $T$  has a skeleton  $T'$  into which pumps are inserted to form  $T$ . This means  $u\#v = u'_k \hat{u}_k \cdots u'_1 \hat{u}_1 u'_0 \# v'_0 \hat{v}_1 v'_1 \cdots \hat{v}_k v'_k$ , where  $u'_k \cdots u'_0 \# v'_0 \cdots v'_k$  is the word generated by  $T'$  and each pair  $(\hat{u}_i, \hat{v}_i)$  is derived using a pump. Then we have

$$\begin{aligned} \text{offset}(u) &= \underbrace{\text{offset}(u'_k \cdots u'_0)}_{=:U_0} + \sum_{i=1}^k \underbrace{\text{offset}(\hat{u}_i)}_{=:U_1}, \\ \text{offset}(v) &= \underbrace{\text{offset}(v'_0 \cdots v'_k)}_{=:V_0} + \sum_{i=1}^k \underbrace{\text{offset}(\hat{v}_i)}_{=:V_1}. \end{aligned}$$

We claim that each of the numbers  $|U_0|, |U_1|, |V_0|, |V_1|$  is bounded by  $n(\mathcal{G})$ , the number of nonterminals of  $\mathcal{G}$ . This clearly implies the lemma: Since  $\mathcal{G}$  is a succinct grammar, it has at most exponentially many nonterminals in the size of its description. We begin with  $U_0, V_0$ . The tree  $T'$  contains each nonterminal of  $\mathcal{G}$  at most once, and by property (C2) in Definition 6.1, we know that the subtree under each nonterminal in  $T'$  not containing  $\#$  has offset  $-1, 0$ , or  $1$ . Thus,  $|U_0|, |V_0| \leq n(\mathcal{G})$ . The bound on  $|U_1|, |V_1|$  is due to admissibility of  $u\#v$ : It yields  $V_0 + V_1 = \text{offset}(v) \geq 0$  and thus  $V_1 \geq -V_0$ . Moreover, by tame-pumping, we know that  $\text{offset}(\hat{v}_i) \leq 0$  for each  $i \in [1, k]$ , and thus  $V_1 \leq 0$ . Together, we obtain  $V_1 \in [-V_0, 0]$ . Finally, tame-pumping also implies  $\text{offset}(\hat{u}_i) = -\text{offset}(\hat{v}_i)$  for each  $i \in [1, k]$  and hence  $U_1 = -V_1$ . ◀

## 110:18 Checking Refinement of Asynchronous Programs

► Remark 6.8. Note that the bound only holds under the condition of admissibility. An easy counterexample is the tame-pumping language  $L = \{x^n \# \bar{x}^n \mid n \in \mathbb{N}\}$ .

The dsNFA  $\mathcal{B}$  of Lemma 6.6 can now be constructed in three steps as follows:

**Step I: Tracking counter effects.** We first observe that since  $\mathcal{G}$  is almost-pumpfree, its pumps  $A \xrightarrow{*} uAv$  can be simulated by a transducer that traverses the derivation tree bottom-up. Thus, we can construct a *singly* succinct finite-state transducer  $\mathcal{T}_A$  with size polynomial in  $|\mathcal{G}|$  that captures all pumps  $A \xrightarrow{*} uAv$ . To be precise,  $\mathcal{T}_A$  accepts exactly those pairs  $(u, v)$  for which  $A \xrightarrow{*} u^{\text{rev}}Av$ . The transducer  $\mathcal{T}_A$  has one state for each nonterminal of  $\mathcal{G}$ .

Since  $\mathcal{B}$  will need to preserve offset and dip, we need to expand  $\mathcal{T}_A$  to track them as well. Here, it is crucial that we only need to do this for  $A \in N_{\#} \cup N_{\bar{\#}}$  and pumps  $A \xrightarrow{*} uAv$  that are used to derive an admissible word. According to Lemma 6.7 tells us that in such a pump, the absolute values of offsets and dips of  $u$  and  $v$  are bounded by  $2^{q(|\mathcal{G}|)}$  for some polynomial  $q$ . Thus, we can modify  $\mathcal{T}_A$  so as to track the dip and offset of the two words it reads. Therefore, for each  $A \in N_{\#} \cup N_{\bar{\#}}$  and each quadruple  $\mathbf{x} = (d_L, \delta_L, d_R, \delta_R)$  of numbers with absolute value at most  $2^{q(|\mathcal{G}|)}$ , we can construct in PSPACE a transducer  $\mathcal{T}_{A,\mathbf{x}}$  with

$$(u, v) \text{ is accepted by } \mathcal{T}_{A,\mathbf{x}} \quad \text{iff} \quad A \xrightarrow{*} u^{\text{rev}}Av \text{ and } e(u^{\text{rev}}) = (d_L, \delta_L), \text{ and } e(v) = (d_R, \delta_R).$$

Moreover,  $\mathcal{T}_{A,\mathbf{x}}$  is singly succinct, polynomial-size, and can be computed in PSPACE. Observe that by Lemma 6.7, if a pump  $A \xrightarrow{*} uAv$  is used in a derivation of an admissible word, then for some quadruple  $\mathbf{x}$ , the pair  $(u^{\text{rev}}, v)$  is accepted by  $\mathcal{T}_{A,\mathbf{x}}$ .

**Step II: Skeleton runs.** The automaton  $\mathcal{B}$  has to read words from left to right, rather than two factors in parallel as  $\mathcal{T}_A$  and  $\mathcal{T}_{A,\mathbf{x}}$  do. To this end, it will guess a run of  $\mathcal{T}_{A,\mathbf{x}}$  without state repetitions; such a run is called a *skeleton run*. For a fixed skeleton run  $\rho$ , the set of words read in each component of  $\mathcal{T}_{A,\mathbf{x}}$  is of the shape  $\Gamma_0^* \{a_1, \varepsilon\} \Gamma_1^* \cdots \{a_k, \varepsilon\} \Gamma_k^*$ , where each  $a_i$  is read in a single step of  $\rho$  and  $\Gamma_i$  is the set of letters from  $\Gamma$  seen in cycles in a state visited in  $\rho$ . Sets of this shape are called *ideals* [12]. The ideal for the left (right) component is called the *left (right) ideal* of the skeleton run. Note that since  $\mathcal{T}_{A,\mathbf{x}}$  has exponentially many states, the skeleton run is at most exponentially long.

**Step III: Putting it together.** The dsNFA  $\mathcal{B}$  guesses and verifies an exponential size skeleton  $T$  of the sECFG  $\mathcal{G}$ . Moreover, for each node  $t$  that is above  $\#$  or  $\bar{\#}$ —but not both—it guesses a quadruple  $\mathbf{x} = (d_L, \delta_L, d_R, \delta_R)$  with  $d_L, d_R \in [0, 2^{q(|\mathcal{G}|)}]$ ,  $\delta_L, \delta_R \in [-2^{q(|\mathcal{G}|)}, 2^{q(|\mathcal{G}|)}]$  and a skeleton run  $\rho_t$  of the transducer  $\mathcal{T}_{A,\mathbf{x}}$ , where  $A$  is  $t$ 's label. The automaton  $\mathcal{B}$  then traverses the skeleton  $T$  in-order; i.e. node, left subtree, right subtree, node; meaning each inner node is visited exactly twice. Whenever  $\mathcal{B}$  visits a node  $t$  as above, it produces an arbitrary word from an ideal of  $\rho_t$ : For the first (resp. second) visit of  $t$ , it uses the left (resp. right) ideal of  $\rho_t$ . Moreover, in addition to the word from the left ideal,  $\mathcal{B}$  outputs a string  $w \in \{x, \bar{x}\}^*$  with  $e(w) = (d_L, \delta_L)$ , where  $\mathbf{x} = (d_L, \delta_L, d_R, \delta_R)$  is the quadruple guessed for  $t$  (and similarly for the right ideal). This way, it preserves offset and dip at the separators  $\#$  and  $\bar{\#}$ .

Since the skeleton  $T$  has exponentially many nodes (in  $|\mathcal{G}|$ ) and each skeleton run  $\rho_t$  requires exponentially many bits, the total number of bits that  $\mathcal{B}$  has to keep in memory is also bounded by an exponential in  $|\mathcal{G}|$ .

---

**References**

---

- 1 Pascal Baumann, Moses Ganardi, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetsche. Context-bounded verification of context-free specifications. *Proc. ACM Program. Lang.*, 7(POPL):2141–2170, 2023. doi:10.1145/3571266.
- 2 Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetsche. Context-bounded verification of thread pools. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi:10.1145/3498678.
- 3 I. Borosh and L. B. Treybig. Bounds on positive integral solutions of linear diophantine equations. *Proceedings of the American Mathematical Society*, 55(2):299–304, 1976.
- 4 Laura Bozzelli and Pierre Ganty. Complexity analysis of the backward coverability algorithm for VASS. In Giorgio Delzanno and Igor Potapov, editors, *Reachability Problems – 5th International Workshop, RP 2011, Genoa, Italy, September 28-30, 2011. Proceedings*, volume 6945 of *Lecture Notes in Computer Science*, pages 96–109. Springer, 2011. doi:10.1007/978-3-642-24288-5\_10.
- 5 Bruno Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, 44:178–186, 1991.
- 6 Wojciech Czerwinski and Piotr Hofman. Language inclusion for boundedly-ambiguous vector addition systems is decidable. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.16.
- 7 Wojciech Czerwiński and Łukasz Orlikowski. 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*, pages 1229–1240. IEEE, 2021. doi:10.1109/FOCS52979.2021.00120.
- 8 Ankush Desai, Pranav Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 709–725. ACM, 2014. doi:10.1145/2660193.2660211.
- 9 Ankush Desai and Shaz Qadeer. P: modular and safe asynchronous programming. In Shuvendu K. Lahiri and Giles Reger, editors, *Runtime Verification – 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, volume 10548 of *Lecture Notes in Computer Science*, pages 3–7. Springer, 2017. doi:10.1007/978-3-319-67531-2\_1.
- 10 Pierre Ganty and Rupak Majumdar. Algorithmic verification of asynchronous programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):6, 2012. doi:10.1145/2160910.2160915.
- 11 Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. Rely/guarantee reasoning for asynchronous programs. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015*, volume 42 of *LIPIcs*, pages 483–496. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.CONCUR.2015.483.
- 12 Jean Goubault-Larrecq, Simon Halfon, P. Karandikar, K. Narayan Kumar, and Philippe Schnoebelen. The ideal approach to computing closed subsets in well-quasi-orderings. In Peter M. Schuster, Monika Seisenberger, and Andreas Weiermann, editors, *Well-Quasi Orders in Computation, Logic, Language and Reasoning*, volume 53 of *Trends In Logic*, pages 55–105. Springer, 2020. doi:10.1007/978-3-030-30229-0\_3.
- 13 Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. In *POPL '07: Proc. 34th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, pages 339–350. ACM Press, 2007.
- 14 Dexter Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997.

- 15 Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 227–242. ACM, 2020. doi:10.1145/3385412.3385980.
- 16 Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Synchronizing the asynchronous. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPICs*, pages 21:1–21:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.CONCUR.2018.21.
- 17 Jérôme Leroux. The Reachability Problem for Petri Nets is Not Primitive Recursive. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1241–1252, February 2022. doi:10.1109/FOCS52979.2021.00121.
- 18 Richard Lipton. The reachability problem is exponential-space hard. *Yale University, Department of Computer Science, Report*, 62, 1976.
- 19 Raphaela Löbel. *Linear Tree Transducers: From Equivalence to Balancedness*. PhD thesis, Technical University of Munich, Germany, 2020. URL: <https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20201127-1552125-1-5>.
- 20 Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetsche. General decidability results for asynchronous shared-memory programs: Higher-order and beyond. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 449–467. Springer, 2021. doi:10.1007/978-3-030-72016-2\_24.
- 21 Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In William W. Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 14–24. ACM, 2004. doi:10.1145/996841.996845.
- 22 Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.
- 23 Robert W Ritchie and Frederick N Springsteel. Language recognition by marking automata. *Information and Control*, 20(4):313–330, 1972.
- 24 Walter J Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4(2):177–192, 1970.
- 25 Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV '06: Proc. 18th Int. Conf. on Computer Aided Verification*, volume 4144 of *LNCS*, pages 300–314. Springer, 2006.
- 26 Akihiko Tozawa and Yasuhiko Minamide. Complexity results on balanced context-free languages. In Helmut Seidl, editor, *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*, volume 4423 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2007. doi:10.1007/978-3-540-71389-0\_25.