Reachability in Fixed VASS: Expressiveness and Lower Bounds

Andrei Draghici[0009−0000−9308−1169], Christoph Haase[0000−0002−5452−936X], and Andrew Ryzhikov[0000−0002−2031−2488]

Department of Computer Science, University of Oxford, Oxford, UK

Abstract. The recent years have seen remarkable progress in establishing the complexity of the reachability problem for vector addition systems with states (VASS), equivalently known as Petri nets. Existing work primarily considers the case in which both the VASS as well as the initial and target configurations are part of the input. In this paper, we investigate the reachability problem in the setting where the VASS and the final configuration are fixed and only the initial configuration is variable. We show that fixed VASS fully express arithmetic with counting on initial segments of the natural numbers. It follows that there is a very weak reduction from any fixed such number-theoretic predicate (e.g. square-freeness or "N_1 is the number of primes smaller than N_2") to reachability in fixed VASS where configurations are presented in unary. If configurations are given in binary, we show that there is a fixed VASS with five counters whose reachability problem is PSPACE-hard.

1 Introduction

Vector addition systems with states (VASS), equivalently known as Petri nets, are a fundamental model of computation. A VASS comprises a finite-state controller with a finite number of counters ranging over the non-negative integers. When a transition is taken, counters can be updated by adding an integer, provided that the resulting counter values are all non-negative; otherwise the transition blocks. Given two configurations of a VASS, each consisting of a control state and an assignment of values to the counters, the reachability problem asks whether there is a path connecting the two configurations in the infinite transition system induced by the VASS. The VASS reachability problem has been one of the most intriguing problems in theoretical computer science and studied for more than fifty years. In the 1970s, Lipton showed this problem EXPSPACE-hard [18]. Ever since the 1980s [19, 14, 16], the reachability problem has been known to be decidable, albeit with non-elementary complexity. This wide gap between the EXPSPACE lower bound and a non-elementary upper bound persisted for many years, until a recent series of papers established various non-elementary lower bounds [5, 6, 15], and resulted in matching a recently established upper bound [17], showing the VASS reachability problem Ackermann-complete. The lower bounds for this result require an unbounded number of counters, but even for a fixed number of counters, the Petri net reachability problem requires non-elementary time [6, 7, 15].

Main results. The main focus of this paper is to investigate the reachability problem for *fixed* VASS, where the VASS under consideration and the final configuration are fixed and only the initial configuration forms the input to a reachability query. Here, it is crucial to distinguish between the encoding of numbers used to represent counter values in configurations: in *unary encoding*, the representation length of a natural number $n \in \mathbb{N}$ is its magnitude n whereas in *binary encoding* the bit length of $n \in \mathbb{N}$ is $\lceil \log n \rceil + 1$. It turns out that establishing meaningful lower bounds under unary encoding of configurations is a rather delicate issue; a full discussion is deferred to Section 4. As a first step, we establish a tight correspondence between reachability in VASS and the first-order theory of initial segments of $\mathbb{N}$ with the arithmetical relations addition ($+$), multiplication ($\times$) and counting quantifiers. An initial segment in $\mathbb{N}$ is a set $\underline{N} = \{0, \ldots, N\}$ for some arbitrary but fixed $N \in \mathbb{N} \setminus \{0\}$. Relations definable in this family of structures are known as *rudimentary relations* and contain many important number-theoretic relations, cf. [9] and the references therein. For instance, the fixed formula $\mathrm{PRIME}(x) \equiv \neg(x = 0) \wedge \neg(x = 1) \wedge \forall y < x \, \forall z < x \, \neg(x = y \times z)$ evaluates to true in $\underline{N}$ precisely for all prime numbers up to N. The formula $\exists^{=z} y \ (y < x) \wedge \mathrm{PRIME}(y)$ evaluates to true if and only if there exist exactly z prime numbers smaller than x.

Given a fixed rudimentary relation $\Phi(x_1, \ldots, x_k)$, we show how to construct a fixed VASS $\mathcal{V}$ and fixed polynomials $p_1, \ldots, p_m$ such that $\Phi(n_1, \ldots, n_k)$ evaluates to true in $\underline{N}$ if and only if there is a run in $\mathcal{V}$ starting in $(p_1(N, n_1, \ldots, n_k), \ldots, p_m(N, n_1, \ldots, n_k))$ and ending in a zero vector. It thus follows that reachability in fixed VASS under unary encoding of configurations is at least as hard as evaluating any rudimentary relation under unary encoding of numbers. Hence, reachability queries in fixed VASS can, e.g., determine primality and square-freeness of a number given in unary. From those developments, it is already possible to infer that reachability in fixed VASS with configurations encoded in binary is hard for every level of the polynomial hierarchy by a reduction from the validity problem for short Presburger arithmetic [21]. In fact, we can establish a PSPACE lower bound for reachability in a fixed VASS with five counters with configurations encoded in binary, by a generic reduction allowing to simulate space-bounded computations of arbitrary Turing machines encoded as natural numbers. A recent conjecture of Jecker [13] states that for every VASS $\mathcal{V}$, there exists a fixed constant C such that if a target configuration is reachable from an initial configuration, then there exists a witnessing path whose length is bounded by $C \cdot m$, where m is the maximum constant appearing in the initial and final configurations. Thus, assuming Jecker's conjecture, reachability in fixed VASS under binary encoding of configurations would be PSPACE-complete. In the course of our work, we were not able to find any evidence that this conjecture is false. It is also worth noting that while all our results assume that the final configuration is fixed to a zero vector, we did not find any stronger lower bounds for the case where the final configuration is variable, and only the VASS is fixed.

Related work. To the best of our knowledge, the reachability problem for fixed VASS has not yet been systematically explored. Closest to the topics of this paper

is the work by Rosier and Yen [22], who conducted a multi-parameter analysis of the complexity of the boundedness and coverability problems for VASS.

However, the study of the computation power of other fixed machines has a long history in the theory of computation. The two classical decision problems for a computation model are *membership* (also called the *word problem*) and *reachability*. Membership asks whether a given machine accepts a given input; the (generic) reachability problem asks whether given an initial and a target configuration, there is a path in the transition system induced by a given machine from the initial configuration to the target configuration. The most prominent example of a reachability problem is the halting problem for different kinds of machines. Classically, the computational complexity of such problems assumes that both the computational model and its input word (for membership) or configurations (for reachability) are part of the input. However, these are two separate parameters. For example, in database theory, the database size and the query size are often considered separately, since the complexity of algorithms may depend very differently on these two parameters, and the sizes of these two parameters in applications can also vary a lot [26]. One approach to study such phenomena is to fix either the database or the query. More generally, the field of parameterised complexity studies the computational difficulty of a problem with respect to multiple parameters of the input.

Returning to our setting, this means fixing either the machine or its input. In this paper, we concentrate on the former. The question can then be seen as follows: in relation to a problem such as membership or reachability, which machine is the hardest one in the given computation model? For some models, the answer easily follows from the existence of universal machines, i.e., machines which are able to simulate any other machine from their class. A classical example here is a universal Turing machine. Sometimes the ability to simulate all other machines has to be relaxed, for example as for Greibach's hardest context-free language [11]. Greibach showed that there exists a fixed context-free grammar such that a membership query for any other context-free grammar can be efficiently reduced to a membership query for this grammar. Similar results are known for two-way non-deterministic pushdown languages [23, 4].

2 Preliminaries

We denote by $\mathbb{Z}$ and $\mathbb{N}$ the set of integers and non-negative integers, respectively. For $N \in \mathbb{N}$ we write $\underline{N}$ to denote the set $\{0, \ldots, N\}$. By $[n, m]$ we define the set of integers between n and m: $[n, m] = \{k \in \mathbb{Z} \mid n \leq k \leq m\}$. By $\mathbf{0}$ we denote the zero vector $(0, 0 \ldots, 0)$ whose dimension is clear from the context.

Counter automata. A *d-counter automaton* is a tuple $\mathcal{A} = (Q, \Delta, \zeta, q_0, q_f)$, where Q is a finite set of states, $\Delta \subseteq Q \times \mathbb{Z}^d \times Q$ is the transition relation, $\zeta : \Delta \to [1, d] \cup \{\top\}$ is a function indicating which counter is tested for zero along a transition ($\top$ meaning no counter is tested), $q_0 \in Q$ is the initial state, and $q_f \in Q$ is the final state. We assume that q_f does not have outgoing transitions.

The set of configurations of $\mathcal{A}$ is $C(\mathcal{A}) := \{(q, n_1, \ldots, n_d) : q \in Q, n_i \in \mathbb{N}, 1 \leq i \leq n\}$. A run ϱ of a counter automaton $\mathcal{A}$ from a configuration $c_1 \in C(\mathcal{A})$ to $c_{n+1} \in C(\mathcal{A})$ is a sequence of configurations interleaved with transitions

$$\varrho = c_1 \xrightarrow{t_1} c_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} c_{n+1}$$

such that for all $1 \leq i \leq n$, $c_i = (q, m_1, \ldots, m_d)$ and $c_{i+1} = (r, m'_1, \ldots, m'_d)$,

- $t_i = (q, (z_1, \ldots, z_d), r)$ with $m'_j = m_j + z_j$ for all $1 \leq j \leq d$; and
- $m_j = 0$ if $\zeta(t_i) = j$.

Observe that we can without loss of generality assume that each transition $t \in \Delta$ is of one of the two types:

- either no counter is tested for zero along t, that is, $\zeta(t) = \top$, in which case we call it *an update transition*;
- or t does not change the values of the counters, that is, $\zeta(t) = j$ for some $1 \leq j \leq d$ and $t = (q, \mathbf{0}, r)$, in which case we call it *a zero-test transition.*

We say that $\mathcal{A}$ is a *vector addition system with states of dimension d (d-VASS)* if $\mathcal{A}$ cannot perform any zero tests, i.e., ζ is the constant function assigning $\top$ to all transitions. We can now formally define the main decision problem we study in this paper.

Problem 1. Fixed VASS zero-reachability
Fixed: d-VASS $\mathcal{A}$.
Input: A vector $\boldsymbol{x} \in \mathbb{N}^d$ of initial values of the counters.
Output: YES if and only if $\mathcal{A}$ has a run from $(q_0, \boldsymbol{x})$ to $(q_f, \mathbf{0})$.

Counter programs. For ease of presentation, we use the notion of counter programs presented e.g. in [5], which are equivalent to VASS, and allow for presenting VASS (and counter automata) in a serialised way. A counter program is a primitive imperative program that executes arithmetic operations on a finite number of counter variables. Formally, a *counter program* consists of a finite set $\mathcal{X}$ of global counter variables (called *counters* subsequently for brevity) ranging over the natural numbers, and a finite sequence $1, \ldots, m$ of line numbers (subsequently *lines* for brevity), each associated with an instruction manipulating the values of the counters or a control flow operation. Each instruction is in of one the following forms:

```
1: goto 2 or 4
2:    x -= 3
3:    goto 1
4: x += 1
5: halt
```

Fig. 1. Example of a counter program.

- $x \mathrel{+}= c$ (increment counter x by constant $c \in \mathbb{N}$),
- $x \mathrel{-}= c$ (decrement counter x by constant $c \in \mathbb{N}$),
- **goto** L_1 **or** L_2 (non-deterministically jump to the instruction labelled by L_1 or L_2),
- **skip** (no operation).

We write **goto** L as an abbreviation for **goto** L **or** L, and also allow statements of the form **goto** L_1 **or** L_2 **or** ... **or** L_k. Moreover, the line with the largest number is a special instruction **halt**. In our examples of counter programs, we usually omit this last line if it is not referenced explicitly.

An example of a counter program is given in Figure 1. This counter program uses a single counter x and consists of five lines. Starting in line 1, the program non-deterministically loops and decrements the counter x by three every time, until it increments x by one and terminates.

To be able to compose counter programs, we describe the operation of substitution, which substitutes a given line (which we always assume to have a **skip** instruction) of a counter program with the "code" of another counter program. Formally, let C_1, C_2 be counter programs with m_1 and m_2 lines respectively. The result of substituting line k, $1 \leq k \leq m_1 - 1$, of C_1 with C_2 is a counter program C_1' with $m_1 + m_2 - 1$ lines obtained, intuitively, by calling C_2 as a sub-routine in this line and when it halts returning control back to C_1. Formally, the instruction corresponding to a line L, $1 \leq L < m_1 + m_2$, is defined as follows:

- if $L < k$, it is the instruction of line L in C_1,
- if $k \leq L < m_2 + k - 1$, it is the instruction of line $L - k + 1$ in C_2,
- if $L = m_2 + k - 1$, it is the instruction **skip**,
- if $m_2 + k \leq L$, it is the instruction of line $L - m_2$ in C_1.

The line numbers in **goto** instructions are changed accordingly. We also consider a substitution of several counter programs. When specifying counter programs, to denote substitution of another counter program we just write its name instead of an instruction in a line. Also, we write $C_1; C_2$ for

1: C_1
2: C_2

and C_1 **or** C_2 as syntactic sugar for the counter program:

1: **goto** 2 **or** 4
2: C_1
3: **goto** 5
4: C_2

When C is a counter program, we write **loop** C as an abbreviation for the counter program

1: **goto** 2 **or** 4
2: C
3: **goto** 1

Hence, the counter program in Figure 1 corresponds to

1: **loop**
2: $\quad x -= 3$
3: $x += 1$

We use indentation to mark the scope of the **loop** instruction. We also assume that if several instructions share the same line and are separated by a semicolon, they all belong to the scope of a loop.

Runs of counter programs. Exactly as in the case of VASS, a *configuration* of a counter program is an element $(L, f) \in \mathbb{N} \times \mathbb{N}^{\mathcal{X}}$, where $L \in \mathbb{N}$ is a program line with a corresponding instruction, and $f\colon \mathcal{X} \to \mathbb{N}$ is a counter valuation. The semantics of counter programs are defined in a natural way: after executing the instructions on the line L, we either non-deterministically go to one of the specified lines (if the instruction on line L is a **goto** instruction), or, otherwise, we go to the line $L + 1$. After executing the last line, we stop.

One can view a counter program as a VASS by treating line numbers as states and defining transitions as specified by the counter program, each labelled with the respective instruction. It is also easy to see how to convert a VASS into a counter program.

A *run* of a counter program is a sequence $\varrho\colon (L_1, f_1) \to (L_2, f_2) \to \ldots \to (L_n, f_n)$ of configurations defined naturally according to the described semantics. For example, $(1, \{x \mapsto 7\}) \to (4, \{x \mapsto 7\}) \to (5, \{x \mapsto 8\})$ is a run of the counter program in Figure 1. Given a run $\varrho\colon (L_1, f_1) \to (L_2, f_2) \to \ldots \to (L_n, f_n)$, we say that ϱ is *terminating* if $L_1 = 1$ and the instruction on line L_n is **halt**, and *zero-terminating* if additionally $f_n(x) = 0$ for all $x \in \mathcal{X}$. We denote by $\mathrm{val}_{\mathrm{end}}(\varrho, x) := f_n(x)$ the value of the counter x at the end of a terminating run. Sometimes, we also want to talk about the value of a counter at a specific point during the execution of a run and define $\mathrm{val}_i(\varrho, x)$ to be the value of the counter x right before we execute the instruction on line i in the run ϱ for the first time, i.e. $\mathrm{val}_i(\varrho, x) := f_k(x)$, where k is the smallest index such that $L_k = i$. For instance, in the example above, we have $\mathrm{val}_{\mathrm{end}}(\varrho, x) = 8$ and $\mathrm{val}_4(\varrho, x) = 7$. We often construct counter programs that admit exactly one run ϱ from a given initial configuration to a target configuration. In such a setting, we may omit the reference to ϱ and simply write $\mathrm{val}_{\mathrm{end}}(x)$ and $\mathrm{val}_i(x)$. The effect $\mathrm{eff}(\varrho)\colon \mathcal{X} \to \mathbb{Z}$ of a run ϱ starting in $(1, f_1)$ and ending in (n, f_n) is a map such that $\mathrm{eff}(\varrho, x) = f_n(x) - f_1(x)$ for all $x \in \mathcal{X}$.

For counter programs, the zero-reachability problem is as follows.

Problem 2. Fixed counter program zero-reachability
Fixed: Counter program $\mathcal{C}$.
Input: A vector $\boldsymbol{x} \in \mathbb{N}^d$ of initial values of the counters.
Output: YES if and only if $\mathcal{C}$ has a zero-terminating run from $\boldsymbol{x}$.

3 Implementation of zero tests

The structure of runs in arbitrary counter programs is very complicated and hard to analyse, and hence it is difficult to force a counter program to have a prescribed behaviour. One of the common ways to deal with this issue is to introduce some restricted zero tests, that is, some gadgets that guarantee that if a run reaches a certain configuration, then along this run, the values of some counters are zero at prescribed positions. In this section, summarising [5], we describe such a gadget in the case where the values of counters are bounded by a given number. The number of zero tests that can be performed this way is also bounded. For a counter v, we call this gadget **zero-test**(v), and later on we will

use it as a single instruction to test that the value of v is zero before executing it.

In Section 4, the assumption that the values of the counters are bounded comes from the the fact that the corresponding values of the variables in rudimentary arithmetic are bounded. In Section 5, we enforce this property for more powerful models of computation and show how to simulate them with VASS.

Let $N \in \mathbb{N}$ be an upper bound on the value of a counter v. Then, we can introduce a counter $\hat{v}$ and enforce the invariant $f(v)+f(\hat{v}) = N$ to hold in all the configurations of any run of our counter program. We achieve this by ensuring that every line containing an instruction of type $v \mathrel{+}= c$ must be followed by a line with a $\hat{v} \mathrel{-}= c$ instruction. From now on, we make the convention that the instruction $v \mathrel{+}= c$ is an abbreviation for $v \mathrel{+}= c; \hat{v} \mathrel{-}= c$. This allows us to remove the hatted counters from our future counter programs whenever it is convenient for us, which will ease readability. So, if we choose an initial configuration in which $f(v) + f(\hat{v}) = N$, we have that this invariant holds whenever the zero-test gadget is invoked.

We introduce auxiliary counters u_1, u_2 that will be tested for zero only in the final configuration, and hence have no hat counterpart. In the following, the instruction **zero-test**(v) denotes the following gadget:

1: **loop**
2: $\quad v \mathrel{+}= 1; \hat{v} \mathrel{-}= 1; u_2 \mathrel{-}= 1$
3: **loop**
4: $\quad v \mathrel{-}= 1; \hat{v} \mathrel{+}= 1; u_2 \mathrel{-}= 1$
5: $u_1 \mathrel{-}= 2$

Consider an initial configuration in which $f(u_1) = 2n$ and $f(u_2) = 2n \cdot N$ for some $n > 0$. Initially, it is true that $f(u_2) = f(u_1) \cdot N$.

Lemma 1 ([5]). *There exists a run of the counter program* ***zero-test***(v) *that starts in a configuration with $f(u_2) \geq 2$, $f(u_2) = f(u_1) \cdot N$, and ends in a configuration with $f(u_2) = f(u_1) \cdot N$ if and only if $f(v) = 0$ in the initial configuration.*

Proof. The invariant $f(v)+f(\hat{v}) = N$ ensures that the loops on line 1 and line 3 can each decrease the value of u_2 by at most N. Moreover, this can only happen if $f(v) = 0$ in the initial configuration. □

From a configuration with $f(u_2) = f(u_1) \cdot N$, a run "incorrectly" executing the **zero-test**(v) subroutine can only reach a configuration with $f(u_2) > f(u_1) \cdot N$. Observe that from such a configuration, we can never reach a configuration respecting the invariant $f(u_2) = f(u_1) \cdot N$ if the values of u_1, u_2 are only changed by **zero-test**(v) instructions. Now, consider a counter v and a counter program C that modifies the values of counters u_1 and u_2 only through the **zero-test**(v) instruction. If we start in a configuration in which $f(u_1) = 2n$ and $f(u_2) = 2n \cdot N$ for some $n > 0$, and we are guaranteed that any run of C cannot execute more than n **zero-test**(v) instructions, then after any run of C, we have that $f(u_2) = f(u_1) \cdot N$ only if the value of the counter v was zero at the beginning

of every **zero-test**(v) instruction. If all the counters that we are interested in are bounded by the same value N, we can use a single pair of counters u_1, u_2 to perform zero tests on all our counters. We subsequently call the counters u_1 and u_2 *testing counters*. To summarise, using this technique, we can perform n zero tests on counters bounded by N via a reachability query in a VASS.

Given a configuration (L, f), we say that (L, f) is a *valid configuration* if f respects the condition that $f(u_2) = f(u_1) \cdot N$. A *valid run* is a run that starts in a valid configuration and ends in a valid configuration. Also, a counter program *admits* a valid run if there exists a valid run that reaches the terminal instruction **halt**. Observe that in every valid run the **zero-test**() subroutine does not change the value of the counter which is tested for zero, that is, this value remains zero. Only the values of the testing counters are changed.

We now introduce components. Informally, a component is a counter program acting as a subroutine such that, if it is invoked in a configuration fulfilling the invariants required for valid runs, upon returning, those invariants still hold. Formally, a *component* is a counter program such that:

- there is a polynomial p such that every valid run performs at most $p(N)$ calls of **zero-test**() on all counters; and
- the values of u_1 and u_2 are updated only by **zero-test**() instructions.

We conclude this section with Lemma 2, which states that sequential composition and non-deterministic branching of components yields components. We will subsequently implicitly make use of this obvious lemma without referring to it.

Lemma 2. *If C_1, C_2 are components then both $C_1; C_2$ and C_1 **or** C_2 are also components.*

Remark 1. Let $\mathcal{V}$ be a fixed VASS, and $s = (q_0, \boldsymbol{n}), t = (q_f, \boldsymbol{m})$ be a pair of its configurations. Given s and t, the Fixed VASS coverability problem asks where there exists a run in $\mathcal{V}$ from s to a configuration $t' = (q_f, \boldsymbol{m'})$ such that $\boldsymbol{m'} \geq \boldsymbol{m}$ componentwise. Note that when simulating zero tests as described above, for each counter x except u_1, u_2, we have a counter $\hat{x}$ such that the sum of the values of x and $\hat{x}$ is always the same and is known in advance. Since the values of u_1, u_2 are never increased, we can introduce in the same way the counters $\hat{u}_1, \hat{u}_2$, initially set to zero, so that $u_i + \hat{u}_i$ is constant for $i = 1, 2$. Hence, by requiring that the final value of $\hat{x}$ is at least the initial value of x, we make sure that the final value of x is equal to zero. Thus, in this setting, reachability queries reduce to coverability queries.

4 Rudimentary arithmetic and unary VASS

In this section, we provide a lower bound for the zero-reachability problem for a VASS when the input configuration is encoded in unary. We observe that there is a close relationship between this problem and deciding validity of a formula of first-order arithmetic with counting, addition, and multiplication on an initial segment of $\mathbb{N}$, also known as rudimentary arithmetic with counting [9].

4.1 Rudimentary arithmetic with counting

For the remainder of this section, all the structures we consider are relational. We denote by $\mathbf{FO}(+,\times)$ the first-order theory of the structure $\langle \mathbb{N}, +, \times \rangle$, where $+$ and $\times$ are the natural ternary addition and multiplication relations. When interpreted over initial segments of $\mathbb{N}$, i.e. sets $\{0, 1, \ldots, N\}$, for some fixed $N \in \mathbb{N}$, the family of the first-order theories is known as rudimentary arithmetic. Note that, in particular, for a predicate $x + y = z$ to hold, all of x, y, z must be at most N. It thus might seem that after we fix N, a formula $\Phi(\boldsymbol{x})$ can only express facts about numbers up to N. However, as discussed in [25] and [9], this can be improved to quantifying over variables up to N^d for any fixed d using $(N+1)$-ary representations of numbers. In other words, for any fixed d and formula $\Phi(\boldsymbol{x})$, there exists a formula $\Phi'(\boldsymbol{x})$ such that for any $N \in \mathbb{N}$ and $\boldsymbol{x} \in \underline{N}^n$, we have that $\langle \underline{N}, +, \times \rangle \models \Phi'(\boldsymbol{x})$ iff $\langle \underline{N}^d, +, \times \rangle \models \Phi(\boldsymbol{x})$.

Rudimentary arithmetic can be extended with counting quantifiers. As described in [25], let rudimentary $\mathbf{FOunC}(+,\times)$ be rudimentary $\mathbf{FO}(+,\times)$ extended with counting quantifiers of the form $\exists^{>x} y\ \varphi(y)$. In this expression, the variable x is free and the variable y is bounded by the quantifier. The semantics of this expression is that there exist more than x different values of y such that the formula $\varphi(y)$ is satisfied. The paper [25] actually uses the counting quantifier $\exists^{=x} y\ \varphi(y)$ to state that the number of such values is exactly x, which can be expressed as $(x = 0 \wedge \neg\exists y\ \varphi(y)) \vee ((\exists^{>x'} y\ \varphi(y)) \wedge (x' + 1 = x) \wedge \neg\exists^{>x} y\ \varphi(y))$.

Moreover, $\mathbf{FOunC}(+,\times)$ can be extended to $\mathbf{FO}k\text{-}\mathbf{aryC}(+,\times)$, $\mathbf{FO}(+,\times)$ with k-ary counting quantifiers $\exists^{=\boldsymbol{x}} \boldsymbol{y}\ \varphi(\boldsymbol{y})$. In this expression, $\boldsymbol{x}, \boldsymbol{y}$ are vectors of the same dimension, and similarly to the previous case, all the variables of $\boldsymbol{x}$ are free and all the variables of $\boldsymbol{y}$ are bounded by the quantifier. The semantics is that the k-tuple $\boldsymbol{x}$ is the $(N+1)$-ary representation of the number of k-tuples $\boldsymbol{y}$ that satisfy $\varphi(\boldsymbol{y})$. As shown in [3], rudimentary $\mathbf{FOunC}(+,\times)$ and rudimentary $\mathbf{FO}k\text{-}\mathbf{aryC}(+,\times)$ have the same expressive power. In order to have a meaningful reduction to *fixed* VASS, we are interested in the following decision problem:

Problem 3. Fixed rudimentary $\mathbf{FO}k\text{-}\mathbf{aryC}(+,\times)$ validity
Fixed: $\Phi(\boldsymbol{x}) \in \mathbf{FO}k\text{-}\mathbf{aryC}(+,\times)$.
Input: $N \in \mathbb{N}$ and $\boldsymbol{x} \in \underline{N}^n$ given in unary.
Output: YES if and only if $\langle \underline{N}, +, \times \rangle \models \Phi(\boldsymbol{x})$.

4.2 Reductions between unary languages

In order to study decision problems whose input is, for some constant k, a k-tuple of numbers presented in unary, and hence to analyse languages corresponding to them, we need a notion of reductions that are weaker compared to the standard ones that are widely used in computational complexity. The reason is that classical problems involving numbers represented in unary, such as Unary Subset Sum [8], have as an input a variable-length sequence of numbers given in unary. Hence, languages of such problems are in fact binary, as we need a delimiter symbol to separate the elements of the sequence. It is not clear how a reasonable

reduction from such a language to a language consisting of k-tuples of numbers for a *fixed* k would look like. In particular, note that unary FIXED VASS ZERO-REACHABILITY is not the unary "counterpart" of binary FIXED VASS ZERO-REACHABILITY in the classical sense. Conversely, arithmetic properties of a single number, e.g. primality or square-freeness, require very low computational resources if the input is represented in unary. Hence, the notion of a reduction between such "genuinely unary" languages has to be very weak.

In view of this discussion, we introduce the following kind of reduction. Given $k > 0$, a *k-tuple unary language* is a subset $L \subseteq \mathbb{N}^k$. We say that L is a tuple unary language if L is a k-tuple unary language for some $k > 0$. Let $L \subseteq \mathbb{N}^k$ and $M \subseteq \mathbb{N}^\ell$ be tuple unary languages, we say that L *arithmetically reduces* to M if there are fixed polynomials $p_1, \ldots, p_\ell \colon \mathbb{N}^k \to \mathbb{N}$ such that $(m_1, \ldots, m_k) \in L$ if and only if $(p_1(m_1, \ldots, m_k), \ldots, p_\ell(m_1, \ldots, m_k)) \in M$.

We believe that this reduction is sensible for the following informal reasons. Polynomials can be represented as arithmetic circuits. To the best of our knowledge, there are no known lower bounds for, e.g. comparing the output of two arithmetic circuits with all input gates having value one [1], suggesting that evaluating a polynomial is a computationally weak operation. Moreover, in the light of sets of numbers definable in rudimentary arithmetic, it seems implausible that applying a polynomial transformation makes, e.g. deciding primality of a number substantially easier.

For a formula Φ, let $\mathcal{L}_\Phi$ be the tuple unary language of yes-instances for FIXED RUDIMENTARY **FO**k-**aryC**$(+, \times)$ VALIDITY. Also, for a counter program C, define $\mathcal{L}_C$ as the tuple unary language of yes-instance for the FIXED COUNTER PROGRAM ZERO-REACHABILITY problem. The remainder of this section is devoted to proving the following theorem.

Theorem 1. *For every formula Φ of rudimentary* **FO**k-**aryC**$(+, \times)$*, there exists a counter program C such that $\mathcal{L}_\Phi$ arithmetically reduces to $\mathcal{L}_C$.*

This theorem can be viewed in two different contexts. On the one hand, it relates the computational complexity of the two problems using a very weak reduction as described above. On the other hand, it also relates the expressivity of two formalisms. Namely, the set of satisfying assignments for formulas of rudimentary arithmetic is at most as expressive as the composition of polynomial transformations with the sets of initial configurations for zero-reachable runs in counter programs. In particular, it shows that fixed VASS can, up to a polynomial transformation, decide number-theoretic properties such as primality, square-freeness, see [9] for further examples. Note that by Remark 1, an analogue of Theorem 1 holds for tuple unary languages of yes-instances of FIXED VASS COVERABILITY.

4.3 Components for arithmetic operations

Since there is no straightforward way to model negation with a counter program, we need to provide gadgets for both the predicates $+$ and $\times$ of rudimentary **FO**k-**aryC**$(+, \times)$ and their negations, and hence design a separate component

for each literal. However, these components may change the values of the counters representing first-order variables, and since a first-order variable might appear in multiple literals, we first provide a gadget to copy the value of a chosen counter to some auxiliary counter before it can be manipulated.

Copy. We provide a counter program $\textsc{Copy}[x, x']$ with the following properties:

1. it admits a valid run if and only if $\mathrm{val}_{\mathrm{end}}(x') = \mathrm{val}_{\mathrm{end}}(x) = \mathrm{val}_1(x)$; and
2. $\textsc{Copy}[x, x']$ is a component.

We implement $\textsc{Copy}[x, x']$ as follows:

1: **loop**
2: $\quad x' -= 1$
3: **zero-test**(x')
4: **loop**
5: $\quad x -= 1; x' += 1; t += 1$
6: **zero-test**(x)
7: **loop**
8: $\quad t -= 1; x += 1$
9: **zero-test**(t)

The loop on line 1 ensures that $\mathrm{val}_4(x') = 0$. We do not do this for the auxiliary counter t because any valid run sets $\mathrm{val}_{\mathrm{end}}(t) = 0$. Observe that $\textsc{Copy}[x, x']$ admits a valid run if and only if the loop on line 4 is executed $\mathrm{val}_1(x)$ many times and the loop on line 7 is executed $\mathrm{val}_4(t) = \mathrm{val}_1(x)$ many times which happens if and only if $\mathrm{val}_{\mathrm{end}}(x') = \mathrm{val}_{\mathrm{end}}(x) = \mathrm{val}_1(x)$. Moreover, any valid run performs 3 calls to the **zero-test**() subroutine, so $\textsc{Copy}[x, x']$ is a component.

Addition. We define a counter program $\textsc{Addition}[x, y, z]$ that enables us to check whether the value stored in counter z is equal to the sum of the values stored in x, y. Formally, it has following properties:

1. $\textsc{Addition}[x, y, z]$ admits a valid run if and only if $\mathrm{val}_1(x) + \mathrm{val}_1(y) = \mathrm{val}_1(z)$;
2. $\textsc{Addition}[x, y, z]$ is a component; and
3. the effect of $\textsc{Addition}[x, y, z]$ is zero on counters x, y, z.

We implement $\textsc{Addition}[x, y, z]$ as follows:

1: $\textsc{Copy}[x, x']; \textsc{Copy}[y, y']; \textsc{Copy}[z, z']$
2: **loop**
3: $\quad z' -= 1$
4: $\quad x' -= 1$ **or** $y' -= 1$
5: **zero-test**(x'); **zero-test**(y'); **zero-test**(z')

It is easy to see that the first property is fulfilled by the counter program and that $\textsc{Addition}[x, y, z]$ is a component because any run performs exactly 12 calss to **zero-test**() (9 calls on line 1, and 3 calls on line 5). The last property is true based on the properties of Copy. The component for the negation of the addition predicate is defined similarly.

Multiplication. We now define a counter program MULTIPLICATION$[x, y, z]$ with the following properties:

1. it admits a valid run if and only if $\mathrm{val}_1(z) = \mathrm{val}_1(x) \cdot \mathrm{val}_1(y)$;
2. MULTIPLICATION$[x, y, z]$ is a component; and
3. the effect of MULTIPLICATION$[x, y, z]$ is zero on counters x, y, z.

We implement MULTIPLICATION$[x, y, z]$ as follows:

```
1:  COPY[x, x']; COPY[y, y']; COPY[z, z']
2:  loop
3:      loop
4:          x' -= 1; t += 1; z' -= 1
5:      zero-test(x')
6:      loop
7:          x' += 1; t -= 1;
8:      zero-test(t)
9:      y' -= 1
10: zero-test(y'); zero-test(z')
```

Observe that the loop on line 3 of any valid run must be executed $\mathrm{val}_1(x)$ $\mathrm{val}_1(x)$ many times in order to pass the zero test on line 5. The effect of this loop is then to decrease the value of z' by $\mathrm{val}_1(x)$ and to set the value of t to $\mathrm{val}_1(x)$. Next, the loop on line 6 must be executed $\mathrm{val}_5(t) = \mathrm{val}_1(x)$ many times to pass the zero test on line 8, so the value of x' is set to $\mathrm{val}_1(x)$ and the value of t is set again to zero. Hence, the effect of lines 3-8 is to subtract $\mathrm{val}_1(x)$ from the value of z' without changing the value of x'. Finally, any valid run passes the test on line 10 if and only if the loop on line 2 is executed $\mathrm{val}_1(y)$ many times, which happens if and only if $\mathrm{val}_1(z) = \mathrm{val}_1(x) \cdot \mathrm{val}_1(y)$. Since we argued that the loop on line 2 is executed $\mathrm{val}_1(y)$ many times, we conclude that any valid run of MULTIPLICATION$[x, y, z]$ performs at most $2N + 9$ calls to **zero-test**(), so MULTIPLICATION$[x, y, z]$ is a component. Again, the last property is ensured by the properties of COPY. The definition of $\neg$MULTIPLICATION$[x, y, z]$ is similar.

4.4 Components for quantification

We define the remaining components that we need in order to prove Theorem 1. These components allow us to existentially and universally quantify over variables in a bounded range.

Existential quantifiers. We start with a counter program EXISTS$[v]$ with the following properties:

1. for every $n \in \underline{N}$, EXISTS$[v]$ admits a valid run ϱ such that $\mathrm{val}_{\mathrm{end}}(\varrho, v) = n$;
2. EXISTS$[v]$ is a component.

We define EXISTS$[v]$ as follows:

```
1: loop v -= 1
```

2: **zero-test**(v)
3: **loop** $v \mathrel{+}= 1$

It is easy to see that both properties hold, since Exists[v] performs exactly one call to the **zero-test**() subroutine.

Universal quantifiers. While the component used for simulating existential quantification can be sequentially composed with a component for a subformula, universal quantification requires directly integrating the component over whose variable we universally quantify. Let $C[v]$ be a component that may access the counter v, test it for zero, and change its value on intermediate steps, but has overall effect zero on counter v. We write ForAll[v] : $C[v]$ for the following counter program:

1: **loop**
2: $\quad v \mathrel{-}= 1$
3: **zero-test**(v)
4: **loop**
5: $\quad C[v]$
6: $\quad v \mathrel{+}= 1$
7: **zero-test**($\hat{v}$)

The properties of ForAll[v] : $C[v]$ are as follows:

1. it admits a valid run if and only if for all $n \in \underline{N}$, C has a valid run with $\mathrm{val}_1(v) = n$; and
2. ForAll[v] : $C[v]$ is a component.

Notice that the instruction on line 7 tests if $\mathrm{val}_7(v) = N$. Thus, any valid run that passes the test on line 7 must be able to execute $C[v]$ for all values of $v \in \underline{N}$. Moreover, since $C[v]$ is a component, we know that the number of calls to **zero-test**() it makes is polynomial in N. Denote this number by B. Then ForAll[v] : $C[v]$ executes at most $N \cdot B + 1$ many calls to **zero-test**() and it is thus a component.

Counting quantifiers. Finally, we design a component which is an extension of the ForAll[v] : $C[v]$ component, where, as in the case of ForAll, $C[v]$ has overall effect zero on v. Formally, ExistsC[x, v] : $C[v]$ component has the following properties:

- it admits a valid run if and only if there exist more than $\mathrm{val}_1(x)$ different integers $n \in \underline{N}$ such that C has a valid run with $\mathrm{val}_1(v) = n$
- the overall effect on counter x is zero; and
- ExistsC[x, v] : $C[v]$ is a component.

We write ExistsC[x, v] : $C[v]$ for the following counter program:

1: **loop**
2: $v \mathrel{-}= 1$
3: **zero-test**(v)
4: COPY$[x, x']$
5: **goto** 6 **or** 9
6: **zero-test**$(\hat{x'})$
7: FORALL$[v] : C[v]$
8: **goto** 15
9: $x \mathrel{+}= 1$
10: **loop**
11: $v \mathrel{+}= 1$
12: **goto** 13 **or** 10
13: $C[v]$; $x' \mathrel{-}= 1$
14: **zero-test**(x')
15: **halt**

The branching on line 5 checks whether $\mathrm{val}_1(x) = N$. If so, $C[v]$ must have a valid run for all values of v, which is checked on line 7. Otherwise, the instructions on line 13 ensure that the value of x' can be decremented if only if $C[v]$ admits at least one valid run with the current value of v. Moreover, the zero test on line 14 is passed if and only if $C[v]$ admitted a valid run for more than $\mathrm{val}_1(x)$ different values. Similarly to the FORALL case, since $C[v]$ is a component, we have that it makes at most a polynomial number of calls to **zero-test**(). If we denote this number by B, the maximum number of calls to **zero-test**() performed by EXISTSC$[x, v] : C[v]$ is bounded by $N \cdot B + 5$. Hence, it is indeed a component.

4.5 Putting it all together

Having defined all the building blocks above, we now prove Theorem 1, which is a consequence of the following lemma.

Lemma 3. *For any formula $\Phi(\boldsymbol{x})$ of **FO**k-**aryC**$(+, \times)$, there exists a component C over k counters and polynomials $p_1, \ldots, p_k : \mathbb{N} \times \mathbb{N}^n \to \mathbb{N}$ such that for any $N \in \mathbb{N}$ and $\boldsymbol{x} \in \mathbb{N}^n$, $\langle \underline{N}, +, \times \rangle \models \Phi(\boldsymbol{x})$ if and only if C admits a valid run from the initial configuration $(p_1(N, \boldsymbol{x}), \ldots, p_k(N, \boldsymbol{x}))$.*

Proof. We prove this statement by structural induction on subformulas of Φ. As shown in [3], rudimentary **FOunC**$(+, \times)$ has the same expressive power as rudimentary **FO**k-**aryC**$(+, \times)$. Since in our setting the formula is fixed, we can thus assume that $\Phi \in$ **FOunC**$(+, \times)$. Moreover, it is easy to see that we can assume that only $\exists^{>x}$ is used as a counter quantifier, since $\exists^{=x}$ can easily be defined using it as described above. Finally, we can assume that negations appear in Φ only in front of arithmetic predicates. In particular, $\neg \exists^{>x} y\ \varphi(y)$ is equivalent to $(\exists^{>x'} y\ \neg\varphi(y)) \wedge (x + x' = N)$.

The counters of the component C are defined to be:

- a counter in vector $\boldsymbol{x}_C$ corresponding to every free variable of $\Phi(\boldsymbol{x})$;
- a counter in vector $\boldsymbol{y}_C$ corresponding to every quantified variable of $\Phi(\boldsymbol{x})$;
- a counter in vector $\boldsymbol{a}_C$ corresponding to every constant of $\Phi(\boldsymbol{x})$; and
- the auxiliary counters $t_C, \boldsymbol{x}'_C, \boldsymbol{y}'_C, \boldsymbol{c}'_C$ used inside the components for predicates and counting quantifiers described above.

We initialise them as follows:

- $f_1(x_C) = x$ and $f_1(\hat{x}_C) = N - x$ for each counter x_C corresponding to a variable x in $\boldsymbol{x}$;
- $f_1(v) = 0$ and $f_1(\hat{v}) = N$ for all the counters corresponding to quantified variables and constants, and auxiliary counters; and
- for the testing counters, $f_1(u_1) = 2N$ and $f_1(u_2) = 2N \cdot P(N)$, where the polynomial $P(N)$ will be defined later.

Assume first that a subformula φ of Φ consists of a single literal. Then, by using the previously defined components, we can construct a fixed component C' corresponding to this literal. In C', for every valid initial configuration (L, f), there exists a valid run starting in it if and only if φ is true under the assignment of the values of the counters in (L, f) to the corresponding variables in φ. If φ is a Boolean combination of multiple literals, by simulating conjunction via sequential composition and disjunction by non-deterministic branching, we can construct a component C_φ with the same property.

We now need to show how to simulate the quantifiers. Let C be the component constructed for φ. We then take

- for $\exists y\ \varphi$:
  1: EXISTS$[y_C]$
  2: $C[y_C]$
- for $\forall y\ \varphi$:
  1: FORALL$[y_C]$:
  2: $\quad C[y_C]$
- for $\exists^{>x} y\ \varphi$:
  1: EXISTSC$[x_C, y_C]$:
  2: $\quad C[y_C]$

As noted above, to be able to use these components, we need to make sure that $C[y_C]$ has overall zero effect on the value of y_C. This is indeed true, since the only place where the value of a counter y_C is changed by a subroutine is in the component corresponding to the quantifier bounding y.

The counter program C starts with a component C_0 that initialises the counters $\boldsymbol{a}$ corresponding to the constants of $\Phi(\boldsymbol{x})$ by a sequence of instruction of the type $a \mathrel{+}= c$ for a corresponding constant c appearing in $\Phi(\boldsymbol{x})$. Finally, we let $C = C_0; C_1$. By the properties established above, it is clear that C admits a valid run starting with f_1 defined above if and only if $\Phi(\boldsymbol{x})$ is valid. To see that C is a component, it remains to note that at every step of the structural induction the number of calls to **zero-test**() is polynomial in N. Hence, there exists a polynomial $P(N)$ such that the overall number calls to **zero-test**() performed by C is bounded by $P(N)$. We conclude by reminding that we use this polynomial to initialise the value of the testing counter u_2. □

To prove Theorem 1, add a loop repeating zero tests at the end of C, thus setting the values of the testing counters to zero if and only if the invariant described in Section 3 holds. After that, set to zero all the remaining counters (including the hatted counters) by decrementing them in loops. A run in thus constructed counter program is zero-accepting if and only if it is valid.

As proved in [3], rudimentary **FO**k-**aryC**$(<)$ has the same expressive power as **FO**k-**aryC**$(+, \times)$. Hence, an alternative proof for Theorem 1 is to express k-ary counting quantifiers without the need for components for addition and multiplication. However, this approach is more technical and less insightful.

5 A universal VASS for polynomial space computations

The goal of this section is to show that there is a fixed 5-VASS whose zero-reachability problem is PSPACE-hard, provided that the initial configuration is encoded in binary. Let us first remark that we can actually use the techniques developed in the previous section to prove that for every i, there exists a fixed VASS $\mathcal{V}_i$ such that deciding zero-reachability for $\mathcal{V}_i$ is Σ_i^{P}-hard. A result by Nguyen and Pak [21] shows that for every i, there is a formula Φ_i of so-called *short Presburger arithmetic* such that deciding Φ_i is Σ_i^{P}-hard. Applying bounds on quantifier elimination established in [27], it can be shown that quantification for formulas of short Presburger arithmetic relativises in a certain sense to an initial segment $\underline{N}$ for some $N \in N$ whose bit length is polynomial in the size of Φ_i. Hence, by combining the results from [21] with Lemma 3, it is possible to show that zero-reachability for fixed binary VASS is hard for the polynomial hierarchy. We do not explore this method further because we can actually construct a fixed binary VASS such that the zero-reachability problem is PSPACE-hard for it and which has a smaller number of counters than the fixed binary VASS obtained from showing NP-hardness via the reduction from short Presburger arithmetic outlined above.

We proceed with our construction as follows. We start with the halting problem for Turing machines (TMs) working in polynomial space and show that this problem is PSPACE-hard even if the space complexity of the TM is bounded by the length of its encoding and its input is empty. In Proposition 2, we then reformulate the halting problem as follows: given the encoding of such a machine as an input to a universal one-tape TM $\mathcal{U}$, does $\mathcal{U}$ accept?

We then use two consecutive simulation. First, we simulate $\mathcal{U}$ with a 3-counter automaton $\mathcal{A}$ (Proposition 3), and then simulate $\mathcal{A}$ with an 5-VASS $\mathcal{V}$ (Theorem 2). To be able to apply the technique described in Section 3, we make sure that the space complexity stays linear in the size of the input throughout these simulations. This implies that both the upper bound on the value of the counters and the required number of zero tests are polynomial in the size of the input, which enables us to establish a polynomial time reduction. As a result we obtain a VASS $\mathcal{V}$ which, in a certain sense, can simulate arbitrary polynomial-space computations.

To provide the reduction, we then show how to transform in polynomial time the input of the problem we started with, the halting problem for polynomial-space TMs, into a zero-reachability query for $\mathcal{V}$.

5.1 The halting problem for space-bounded TMs

The goal of this subsection is to show that there exists a *fixed* polynomial-space TM whose halting problem is PSPACE-complete. Note that using standard arguments, we can assume that $\mathcal{M}$ below always halts.

Proposition 1 ([2, Section 4.2]). *The following problem is PSPACE-complete: given a TM $\mathcal{M}$, an input word w and a number n encoded in unary, decide if $\mathcal{M}$ accepts w in at most n space.*

We fix some way of encoding, using an alphabet of size at least two, of Turing machines and we denote by $|\mathcal{M}|$ the length of the encoding of $\mathcal{M}$, which we call the *size* of $\mathcal{M}$. Given a TM $\mathcal{M}$, we say that it is $|\mathcal{M}|$-space-bounded if on every input it halts using at most $|\mathcal{M}|$ space. Given $\mathcal{M}$, an input word w and a number n encoded in unary, it is easy to construct a $|\mathcal{M}|$-space-bounded TM $\mathcal{M}'$ such that if $\mathcal{M}$ accepts w in space at most n, then $\mathcal{M}'$ accepts on the empty input, otherwise $\mathcal{M}'$ rejects on the empty input. Moreover, the size of $\mathcal{M}'$ is polynomial in $|\mathcal{M}|$, $|w|$ and 2^n.

Indeed, $\mathcal{M}'$ can be constructed as follows. When run on the empty input, it writes w on some tape, and then runs $\mathcal{M}$ treating this tape as the input tape. Additionally, it initialises another tape with n written in unary, and before each step of $\mathcal{M}$ it checks that the space used by the tape where $\mathcal{M}$ is simulated does not exceed n. If it does, it immediately rejects. It is easy to see that such a TM is $|\mathcal{M}'|$-space-bounded and satisfies the required conditions.

Hence we get that the following problem is PSPACE-complete: given a $|\mathcal{M}|$-space-bounded TM $\mathcal{M}$, does $\mathcal{M}$ accept on the empty input? Observe that from the construction above we can assume that $\mathcal{M}$ has a special representation such that the fact that it is $|\mathcal{M}|$-space-bounded can be checked in polynomial time.

Let $\mathcal{U}$ be a one-tape universal TM. This TM has a single read-write tape, which in the beginning contains the input, that is, a description of a TM $\mathcal{M}$ it is going to simulate. If $\mathcal{M}$ is $|\mathcal{M}|$-space-bounded (and represented as mentioned in the previous paragraph), $\mathcal{U}$ simulates $\mathcal{M}$ on the empty input in space linear in $|\mathcal{M}|$ [2, Claim 1.6], otherwise $\mathcal{U}$ rejects. That is, in this space, $\mathcal{U}$ accepts or rejects depending on whether $\mathcal{M}$ accepts or rejects the empty word. Hence we get the following proposition.

Proposition 2. *There exists a fixed linear-space TM $\mathcal{U}$ such that the question whether $\mathcal{U}$ halts on a given input is PSPACE-complete.*

5.2 From TMs to a counter automata

In the previous subsection, we obtained a PSPACE-complete problem which already resembles the form of the reachability problem for a fixed counter program: given a fixed linear-space TM $\mathcal{U}$, does it accept a given input? In this section we show how to simulate $\mathcal{U}$ with a fixed counter automaton $\mathcal{A}$, and in the next section we show how to simulate $\mathcal{A}$ with a fixed binary VASS $\mathcal{V}$.

Let $\mathcal{A}$ be a counter automaton. We say that $\mathcal{A}$ is *deterministic* if for every configuration $(q, n_1, \ldots, n_d)$ there is at most one transition that $\mathcal{A}$ can take from this configuration. Suppose that $\mathcal{A}$ is deterministic, and that its final state q_f does not have any outgoing transitions. Let $\boldsymbol{n} = (n_1, \ldots, n_d) \in \mathbb{N}^d$. We treat $\mathcal{A}$ as an acceptor for such vectors. We say that $\mathcal{A}$ works in time t and space s on $\boldsymbol{n}$ if the unique run starting in the configuration $(q_0, n_1, \ldots, n_d)$ ends in a state without outgoing transitions, has length t, and the bit length of the largest value of a counter along this run is s. If this run ends in q_f, we say that $\mathcal{A}$ *accepts* this vector, otherwise we say that it *rejects* it. In all our constructions we make sure that there are no infinite runs. Note that, as in the case of TMs,

we measure space complexity in the bit length of the values of the counters, and not in their actual values.

Let Σ be a finite alphabet. Let us bijectively assign a natural number to each word over Σ as follows. First, assign a natural number between 1 and $|\Sigma|$ to each symbol in Σ. Then w can be considered as a number in base $|\Sigma|+1$, with the least significant digit corresponding to the first letter of w. We denote this number by $\mathrm{num}(w)$.

Let $\mathcal{M}$ be a TM, and w be its input. We can transform w into a vector $(\mathrm{num}(w), 0, \ldots, 0)$, which will be the input of a deterministic counter automaton $\mathcal{A}$. We say that $\mathcal{A}$ *simulates* $\mathcal{M}$ if w is accepted by $\mathcal{M}$ if and only if the corresponding vector is accepted by $\mathcal{A}$. We say that this simulation is *in linear space* if there exists a constant c such that if the space complexity of $\mathcal{M}$ is s on some input, then the space complexity of $\mathcal{A}$ on the coresponding input is cs.

The proof of the following proposition uses the techniques described in the proofs of [10, Theorem 4.3(a)] and [12, Theorem 2.4].

Proposition 3. *For every one-tape TM $\mathcal{M}$, there exists a deterministic 3-counter automaton $\mathcal{A}$ that simulates it in linear space.*

Proof. The idea of the proof is as follows. Two counters of $\mathcal{A}$, call them ℓ and r, represent the content of the tape of $\mathcal{M}$ to the left and to the right of the reading head. They are encoded similarly to the way we encode the input word. Namely, let $w_1 a w_2$, where $w_1, w_2 \in \Sigma^*$ and $a \in \Sigma$, be the content of the tape at some moment of time, with the working head in the position of the letter a. Denote by w_1^R the reversal of the word w. Then ℓ stores $\mathrm{num}(w_1^R)$, r stores $\mathrm{num}(w_2)$, and a is stored in the finite memory of the underlying finite automaton.

Now, to make a step to the left, we do the following. First, we need to add a to the end of the word encoded by the value of r. This is done by multiplying the value of r by $|\Sigma|+1$ and adding $\mathrm{num}(a)$ to it. Next, we need to extract the last letter of the word encoded by the value of ℓ, and remove this letter. To do so, we do the opposite of what we did for r: this letter is the residue of dividing the value of ℓ by $|\Sigma|+1$, and the new value of ℓ is the result of this division.

The reason we need the third counter x is to perform these multiplications and divisions. Namely, to divide the value of a counter ℓ by a constant c, we repeat the following until it is no longer possible: subtract c from the value of ℓ and add one to the value of x. When the value of ℓ becomes smaller than c, we get the result of the division in the counter x, and the remainder in ℓ. Multiplication by a constant is done similarly. Observe that by construction the largest value of a counter of $\mathcal{A}$ at any moment of time is at most $(|\Sigma|+1)^S$, where S is the maximal amount of space $\mathcal{M}$ uses on given input. The bit length of this number is linear in S, hence $\mathcal{A}$ simulates $\mathcal{M}$ in linear space. □

By simulating $\mathcal{U}$ from Proposition 2 with a counter automaton $\mathcal{A}$, we get the following statement.

Corollary 1. *There exists a fixed 3-counter automaton $\mathcal{A}$ working in linear space such that the zero-reachability problem for it is PSPACE-complete.*

For 2-counter automata, no such result is known. Informally speaking, such automata are exponentially slower than 3-counter automata: the known simulation requires storing the values of the three counters x, y, z as $2^x 3^y 5^z$ [20]. They are also less expressive: for example, 2-counter automata cannot compute the function 2^n [24], while for 3-counter automata this is trivial. It is worth noting the developments of the next subsection imply that a lower bound for fixed 2-counter automata translates into a lower bound for fixed 4-VASS.

5.3 From counter automata to VASS

To go from a counter automaton to a VASS, we need to simulate zero tests with a VASS. In general, this is not possible. However, the space complexity of the counter automaton in Corollary 1 is linear, so the values of all its counters are bounded by a polynomial in the bit length of the input. The number of zero tests $\mathcal{A}$ performs does not exceed its time complexity, which is at most exponential in the space complexity. However, this is not a problem, since all the values are provided and stored in binary. The bit length of the number of zero tests is thus polynomial in the input, and hence the testing counters described in Section 3 can be initialised with a polynomial time reduction, hence obtaining PSPACE-hardness of the zero-reachability problem in fixed 8-VASS.

Moreover, a more advanced technique of quadratic pairs described in [7] allows to deduce the same result for 5-VASS. Namely, a slight variation of [7, Lemma 2.7] states that given a 3-counter automaton $\mathcal{A}$ working in linear space, one can construct a 5-VASS $\mathcal{V}$ such that fixed zero-reachability in $\mathcal{A}$ can be reduced in polynomial time to fixed zero-reachability in $\mathcal{V}$. The same reasoning as before shows that we can initialise the counters of $\mathcal{V}$ to account for enough zero tests. Hence we get the main result of this section.

Theorem 2. *There exists a fixed* 5*-VASS such that the* Fixed VASS zero-reachability *problem for it is PSPACE-hard assuming that the input configuration is given in binary.*

By Remark 1 and by further inspecting the construction in [7, Lemma 2.7], together with the PSPACE upper bound for coverability in fixed VASS with configurations given in binary established in [22], we moreover obtain the following corollary.

Corollary 2. *There exists a fixed* 6*-VASS such that the* Fixed VASS coverability *problem for it is PSPACE-complete assuming that the input configurations are given in binary.*

Acknowledgements. We would like to thank anonymous reviewers for their useful comments on the content and presentation of the paper. This work is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 852769, ARiAT).

References

1. Allender, E., Bürgisser, P., Kjeldgaard-Pedersen, J., Miltersen, P.B.: On the complexity of numerical analysis. SIAM Journal on Computing **38**(5), 1987–2006 (2009). https://doi.org/10.1137/070697926
2. Arora, S., Barak, B.: Computational Complexity – A Modern Approach. Cambridge University Press (2009)
3. Barrington, D.A.M., Immerman, N., Straubing, H.: On uniformity within NC^1. Journal of Computer and System Sciences **41**(3), 274–306 (1990). https://doi.org/10.1016/0022-0000(90)90022-D
4. Chistikov, D., Majumdar, R., Schepper, P.: Subcubic certificates for CFL reachability. Proceedings of the ACM on Programming Languages **6**(POPL) (2022). https://doi.org/10.1145/3498702
5. Czerwiński, W., Lasota, S., Lazić, R., Leroux, J., Mazowiecki, F.: The reachability problem for petri nets is not elementary. Journal of the ACM **68**(1), 1–28 (2020). https://doi.org/10.1145/3313276.3316369
6. Czerwinski, W., Orlikowski, L.: Reachability in vector addition systems is Ackermann-complete. In: Annual Symposium on Foundations of Computer Science, FOCS. pp. 1229–1240. IEEE (2021). https://doi.org/10.1109/FOCS52979.2021.00120
7. Czerwinski, W., Orlikowski, L.: Lower bounds for the reachability problem in fixed dimensional VASSes. In: Symposium on Logic in Computer Science, LICS. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3531130.3533357
8. Elberfeld, M., Jakoby, A., Tantau, T.: Logspace versions of the theorems of Bodlaender and Courcelle. In: Annual Symposium on Foundations of Computer Science, FOCS. pp. 143–152 (2010). https://doi.org/10.1109/FOCS.2010.21
9. Esbelin, H.A., More, M.: Rudimentary relations and primitive recursion: A toolbox. Theoretical Computer Science **193**(1), 129–148 (1998). https://doi.org/https://doi.org/10.1016/S0304-3975(97)00002-9
10. Fischer, P.C., Meyer, A.R., Rosenberg, A.L.: Counter machines and counter languages. Mathematical systems theory **2**, 265–283 (1968). https://doi.org/10.1007/BF01694011
11. Greibach, S.A.: The hardest context-free language. SIAM Journal on Computing **2**(4), 304–310 (1973). https://doi.org/10.1137/0202025
12. Greibach, S.A.: Remarks on the complexity of nondeterministic counter languages. Theoretical Computer Science **1**(4), 269–288 (1976). https://doi.org/10.1016/0304-3975(76)90072-4
13. Jecker, I.: 22.1 complexity of fixed vas reachability. https://autoboz.org/open-problems (2023), accessed: 2023-10-12
14. Kosaraju, S.R.: Decidability of reachability in vector addition systems (preliminary version). In: Symposium on Theory of Computing, STOC. pp. 267–281. ACM (1982). https://doi.org/10.1145/800070.802201
15. Leroux, J.: The reachability problem for petri nets is not primitive recursive. In: Annual Symposium on Foundations of Computer Science, FOCS. pp. 1241–1252. IEEE (2021). https://doi.org/10.1109/FOCS52979.2021.00121
16. Leroux, J., Schmitz, S.: Demystifying reachability in vector addition systems. In: Symposium on Logic in Computer Science, LICS. pp. 56–67. IEEE Computer Society (2015). https://doi.org/10.1109/LICS.2015.16

17. Leroux, J., Schmitz, S.: Reachability in vector addition systems is primitive-recursive in fixed dimension. In: Symposium on Logic in Computer Science (LICS). pp. 1–13 (2019). https://doi.org/10.1109/LICS.2019.8785796
18. Lipton, R.J.: The reachability problem requires exponential space. Research report (Yale University. Department of Computer Science), Department of Computer Science, Yale University (1976)
19. Mayr, E.W.: An algorithm for the general petri net reachability problem. SIAM Journal on Computing **13**(3), 441–460 (1984). https://doi.org/10.1137/0213029
20. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, USA (1967)
21. Nguyen, D., Pak, I.: Short Presburger arithmetic is hard. SIAM Journal on Computing **51**(2), 17:1–30 (2022). https://doi.org/10.1137/17M1151146
22. Rosier, L.E., Yen, H.C.: A multiparameter analysis of the boundedness problem for vector addition systems. Journal of Computer and System Sciences **32**(1), 105–135 (1986). https://doi.org/10.1016/0022-0000(86)90006-1
23. Rytter, W.: A hardest language recognized by two-way nondeterministic pushdown automata. Information Processing Letters **13**(4), 145–146 (1981). https://doi.org/10.1016/0020-0190(81)90045-4
24. Schroeppel, R.: A two counter machine cannot calculate 2^N. Artificial Intelligence Memo 257, Massachusetts Institute of Technology (1972)
25. Schweikardt, N.: Arithmetic, first-order logic, and counting quantifiers. ACM Transactions on Computational Logic (TOCL) **6**(3), 634–671 (2005). https://doi.org/10.1145/1071596.1071602
26. Vardi, M.Y.: The complexity of relational query languages (extended abstract). In: Symposium on Theory of Computing, STOC. pp. 137–146. Association for Computing Machinery, New York, NY, USA (1982). https://doi.org/10.1145/800070.802186
27. Weispfenning, V.: The complexity of almost linear diophantine problems. Journal of Symbolic Computation **10**(5), 395–403 (1990). https://doi.org/10.1016/S0747-7171(08)80051-X