The Reachability Problem for Petri Nets Is Not Elementary*

WOJCIECH CZERWIŃSKI, University of Warsaw, Poland
SŁAWOMIR LASOTA, University of Warsaw, Poland
RANKO LAZIĆ, University of Warwick, UK
JÉRÔME LEROUX, Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, Talence, France
FILIP MAZOWIECKI, Max Planck Institute for Software Systems, Germany

Petri nets, also known as vector addition systems, are a long established model of concurrency with extensive applications in modelling and analysis of hardware, software and database systems, as well as chemical, biological and business processes. The central algorithmic problem for Petri nets is reachability: whether from the given initial configuration there exists a sequence of valid execution steps that reaches the given final configuration. The complexity of the problem has remained unsettled since the 1960s, and it is one of the most prominent open questions in the theory of verification. Decidability was proved by Mayr in his seminal STOC 1981 work, and the currently best published upper bound is non-primitive recursive Ackermannian of Leroux and Schmitz from LICS 2019. We establish a non-elementary lower bound, i.e. that the reachability problem needs a tower of exponentials of time and space. Until this work, the best lower bound has been exponential space, due to Lipton in 1976. The new lower bound is a major breakthrough for several reasons. Firstly, it shows that the reachability problem is much harder than the coverability (i.e., state reachability) problem, which is also ubiquitous but has been known to be complete for exponential space since the late 1970s. Secondly, it implies that a plethora of problems from formal languages, logic, concurrent systems, process calculi and other areas, that are known to admit reductions from the Petri nets reachability problem, are also not elementary. Thirdly, it makes obsolete the currently best lower bounds for the reachability problems for two key extensions of Petri nets: with branching and with a pushdown stack.

We develop a construction that uses arbitrarily large pairs of values with ratio R to provide zero testable counters that are bounded by R. At the heart of our proof is then a novel gadget, the so-called factorial amplifier that, assuming availability of counters that are zero testable and bounded by k, guarantees to produce arbitrarily large pairs of values whose ratio is exactly the factorial of k. Repeatedly composing the factorial amplifier with itself by means of the former construction enables us to compute in linear time Petri nets that simulate Minsky machines whose counters are bounded by a tower of exponentials, which yields the non-elementary lower bound. By refining this scheme further, we in fact establish hardness for h-exponential space already for Petri nets with $h + 13$ counters.

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

Additional Key Words and Phrases: Petri nets, vector addition systems, reachability problems

*This research has been supported by the ERC project 'Lipa' within the EU Horizon 2020 research and innovation programme (No. 683080), NCN grants 'Separation problems in automata theory' (2016/21/D/ST6/01376) and 'Automatic analysis of concurrent systems' (2017/27/B/ST6/02093), ANR programmes IdEx Bordeaux (ANR-10-IDEX-03-02) and BraVAS (ANR-17-CE40-0028), and the Leverhulme Trust Research Fellowship 'Petri Net Reachability Conjecture' (RF-2017-579).

Authors' addresses: Wojciech Czerwiński, University of Warsaw, Poland, wczerwin@mimuw.edu.pl; Sławomir Lasota, University of Warsaw, Poland, sl@mimuw.edu.pl; Ranko Lazić, University of Warwick, UK, R.S.Lazic@warwick.ac.uk; Jérôme Leroux, Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, Talence, France, jerome.leroux@labri.fr; Filip Mazowiecki, Max Planck Institute for Software Systems, Germany, filipm@mpi-sws.org.

Manuscript submitted to ACM

ACM Reference Format:
Wojciech Czerwiński, Sławomir Lasota, Ranko Lazić, Jérôme Leroux, and Filip Mazowiecki. 2021. The Reachability Problem for Petri Nets Is Not Elementary. 1, 1 (November 2021), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Petri nets [47], also known as vector addition systems [25], [19, cf. Section 5.1], [21], are a long established model of concurrency with extensive applications in modelling and analysis of hardware [8, 30], software [6, 18, 23] and database [4, 5] systems, as well as chemical [1], biological [2, 46] and business [38, 52] processes (the references on applications are illustrative). The central algorithmic problem for Petri nets is reachability: whether from the given initial configuration there exists a sequence of valid execution steps that reaches the given final configuration.

There are several presentations of Petri nets, and a number of variants of their reachability problem, all of which are equivalent. One simple way to state the problem is: given a finite set T of integer vectors in d-dimensional space and two d-dimensional vectors $\mathbf{v}$ and $\mathbf{w}$ of nonnegative integers, does there exist a walk from $\mathbf{v}$ to $\mathbf{w}$ such that it stays within the nonnegative orthant, and its every step modifies the current position by adding some vector from T?

Brief History of the Problem. Over the past half century, the complexity of the Petri nets reachability problem has remained unsettled. The late 1970s and the early 1980s saw the initial burst of activity. After an incomplete proof by Sacerdote and Tenney [49], decidability of the problem was established by Mayr [41, 42], whose proof was then simplified by Kosaraju [26]. Building on the further refinements made by Lambert in the 1990s [27], there has been substantial progress over the past ten years [31–33], culminating in the first upper bound on the complexity [34], recently improved to Ackermannian [35].

In contrast to the progress on refining the proof of decidability and obtaining an upper bound on the complexity, Lipton's landmark result that the Petri nets reachability problem requires exponential space [39] has remained the state of the art on lower bounds for over 40 years. Moreover, in conjunction with an apparent tightness of Lipton's construction, this has led to the conjecture that the problem is ExpSpace-complete becoming common in the community.[1]

Main Result and Its Significance. We show that the Petri nets reachability problem is not elementary, more precisely that it is hard for the class Tower of all decision problems that are solvable in time or space bounded by a tower of exponentials whose height is an elementary function of the input size [50, Section 2.3]. We see this result as important for several reasons:

- It refutes the conjecture of ExpSpace-completeness, establishing that the reachability problem is much harder than the coverability (i.e., state reachability) problem; the latter is also ubiquitous but has been known to be ExpSpace-complete since the late 1970s [39, 48].[2]
- It narrows significantly the gap to the best known upper bound [35] in terms of the Ackermannian function, which is among the slowest-growing functions that dominate all primitive recursive functions.
- It implies that a plethora of problems from formal languages [10], logic [9, 11, 12, 24], concurrent systems [15, 17], process calculi [44], linear algebra [20] and other areas (the references are again illustrative), that are known to admit reductions from the Petri nets reachability problem, are also not elementary; for more such problems and a wider discussion, we refer to Schmitz's survey [51].

[1] For an interesting post by Lipton about his exponential space hardness result, we refer the reader to https://rjlipton.wordpress.com/2009/04/08/an-expspace-lower-bound/.
[2] In terms of the statement of the reachability problem above, coverability is obtained by relaxing the requirement that the walk hits exactly $\mathbf{w}$ to hitting any vector pointwise greater than or equal to $\mathbf{w}$.

- It makes obsolete the Tower lower bounds for the reachability problems for two key extensions of Petri nets: branching vector addition systems [28] and pushdown vector addition systems [29].
- It shows that the proposed proof of an earlier claim that the Petri nets reachability problem is decidable in doubly exponential space [7], which was subsequently shown incorrect [22], cannot be repaired.

Petri Nets and Exponential Space Hardness. Before we present the main ideas involved in the proof of the non-elementary lower bound for the reachability problem, let us introduce some key aspects of Petri nets by recalling the crux of Lipton's construction for the exponential space hardness.

Minsky machines, which can be thought of as deterministic finite-state machines equipped with several registers, are one of the classical universal models of computation [45, Chapter 14]. The registers, which are called counters, store natural numbers (initially 0) and can be manipulated by only two simple operations: increments (x += 1), and conditionals that either jump if a counter is zero or decrement it otherwise (**if** $\mathsf{x} = 0$ **then goto** L **else** x −= 1). With appropriate restrictions, the halting problem for Minsky machines is complete for various time and space complexity classes. Lipton's proof proceeds by reducing from the following ExpSpace-complete problem (cf. [16, Theorem 3.1]): given a Minsky machine of size n with 3 counters, does it halt after a run in which the counters remain bounded by 2^{2^n}?

Petri nets can be construed as similar to Minsky machines, but with two important differences. Firstly, Petri nets can increment a counter always, and can decrement a counter if positive, but cannot test whether a counter is zero. Secondly, Petri nets are nondeterministic. Thus, a decrement of a counter either succeeds and the run continues (if the counter was positive), or fails and the current nondeterministic branch is blocked (if the counter was zero). It is the lack of zero tests that makes decidable [42] the reachability problem: given a Petri net and a subset of its counters, does it halt in a configuration where all the counters from the subset are zero?

To construct a Petri net that simulates the given Minsky machine of size n as long as its 3 counters are bounded by 2^{2^n}, the main task is therefore checking that such a counter x is zero. Lipton observed that it suffices to introduce a counter $\hat{\mathsf{x}}$, set up and maintain the invariant $\mathsf{x} + \hat{\mathsf{x}} = 2^{2^n}$, and implement a macro $\mathbf{Dec}_n\ \hat{\mathsf{x}}$ that decrements $\hat{\mathsf{x}}$ and increments x (i.e., performs the code $\hat{\mathsf{x}}$ −= 1 x += 1) exactly 2^{2^n} times. That is because the code $\mathbf{Dec}_n\ \hat{\mathsf{x}}$ $\mathbf{Dec}_n\ \mathsf{x}$ (where, in the latter instance of the macro, x and $\hat{\mathsf{x}}$ are swapped) then checks that counter x is zero: it either succeeds and leaves x and $\hat{\mathsf{x}}$ unchanged if x was zero (i.e. $\hat{\mathsf{x}}$ was 2^{2^n}), or fails otherwise.

Lipton's construction meets that goal inductively, by setting up pairs of counters such that $\mathsf{x}_i + \hat{\mathsf{x}}_i = 2^{2^i} = \mathsf{y}_i + \hat{\mathsf{y}}_i$ and implementing a macro $\mathbf{Dec}_i$ that decrements a counter and increments its complement exactly 2^{2^i} times, for $i = 0, 1, \ldots, n$. Doing it for $i = 0$ is easy. To step from i to $i+1$, consider the following code, where the loops are repeated nondeterministic numbers of times:

loop
  x_i += 1 $\hat{\mathsf{x}}_i$ −= 1
  loop
    y_i += 1 $\hat{\mathsf{y}}_i$ −= 1
    $\hat{\mathsf{x}}_{i+1}$ −= 1 x_{i+1} += 1
  $\mathbf{Dec}_i\ \mathsf{y}_i$
$\mathbf{Dec}_i\ \mathsf{x}_i$.

Assuming that x_i and y_i are zero at the start, there is a unique nondeterministic branch that runs the code completely (without getting blocked): it repeats the outer loop 2^{2^i} times with the final $\mathbf{Dec}_i\ \mathsf{x}_i$ both checking that x_i equals 2^{2^i} (i.e.

$\hat{x}_i$ equals zero) and resetting it to zero, and in each iteration similarly the inner loop is repeated also 2^{2^i} times. Hence, by squaring 2^{2^i}, the code decrements the counter $\hat{x}_{i+1}$ and increments the counter x_{i+1} exactly $2^{2^{i+1}}$ times, as required for an implementation of $\mathbf{Dec}_{i+1}\ \hat{x}_{i+1}$.

Obtaining the Tower *Lower Bound.* To prove that the Petri nets reachability problem requires a tower of exponentials of time and space, we have to tackle two major obstacles:

(1) The fact that Lipton's construction applies also to the coverability problem, which has an ExpSpace upper bound [48], means that a construction that achieves a lower bound beyond ExpSpace cannot follow the same pattern. For example, we cannot hope to implement a macro whose unique complete execution performs some given counter operations exactly a triply exponential number of times.
(2) It has been known for many years how Petri nets can compute various functions weakly, in the sense that the result may be nondeterministically either correct or smaller [36, 43][3]. Most notably, for all natural numbers n, Grzegorczyk's function [40] F_n is computable weakly by a Petri net of size $O(n)$. However, even supposing that we have means of simulating zero tests of several counters bounded by some k, it has been unknown how to compute exactly a value exponential in k without using $\Omega(k)$ extra counters.

To overcome the ExpSpace barrier, we devise a novel construction for simulating zero tests of counters bounded by some R: instead of relying on an ability to repeat some counter operations exactly R times, it assumes that a pair of counters have been set to sufficiently large values whose ratio is exactly R, and it ensures that the simulations of zero tests are correct by testing that one of the two auxiliary counters is zero in the final configuration of the reachability problem instance.

In overcoming the second obstacle, surprisingly a central role is played by the simple identity $\prod_{i=1}^{k-1}(i+1)/i = k$. We devise a gadget, the so-called factorial amplifier, that sets two counters c and d to arbitrarily large values such that $d = c \cdot k!$ as follows. After initialising both counters to the same nondeterministic value y, the main loop uses some machinery and a constant number of auxiliary counters to attempt to multiply c and d by each of the fractions $1/i$ and $(i+1)/i$ (respectively) for $i = 1, \ldots, k-1$. Since the multiplications are implemented by repeated additions and subtractions, and since the factorial amplifier cannot zero-test counters that are not bounded by k but instead makes nondeterministic guesses, we have that the resulting values of c and d are not necessarily correct. Nevertheless, the gadget is programmed in such a way that, provided the initial nondeterministic value y has appropriate divisibility properties, there is a unique accurate computation in which all the multiplications by the fractions $1/i$ and $(i+1)/i$ are performed exactly. Moreover, all inaccurate computations produce final values of d that are smaller than $y \cdot k$. In other words, if an accurate computation exists (i.e. if y has the divisibility properties), then it can be forced by checking at the end of the gadget that d has been multiplied at least (i.e. exactly) by k. In that case, the accurate computation has also divided c by $(k-1)!$, yielding the ratio $k!$ between counters c and d as required.

For readers familiar with programming weak computations using Petri nets, or desiring further intuition at this stage, we remark that the properties we have outlined of the factorial amplifier gadget rely firstly on the fact that the fractions $(i+1)/i$ are greater than 1 and so any nondeterministic deviation from guessing the relevant zero tests correctly results in a smaller final value of counter d than required. However, using the identity $\prod_{i=1}^{k-1}(i+1)/i = k$ in that way just to effect a weak multiplication of d by k would be pointless since the latter can be done directly trivially. Rather, and this is one of the key ideas in this work, the gadget involves another counter b that is initially 1, and is

[3] In their article, Mayr and Meyer establish that the containment problem between finite sets of reachable configurations of two given Petri nets is 'the first uncontrived decidable problem which is not primitive recursive'.

then multiplied by $i + 1$ during iterations $i = 1, \ldots, k - 1$ of the main loop. Instead of attempting to multiply c by $1/i$ separately, the steps that implement the multiplication of d by $(i + 1)/i$ are used as a backbone where the current value of b determines the frequency of inserting among them the steps needed on c. Thus, success of the final check on d in the gadget necessitates that the operations on d have been correct and therefore performed the maximal possible numbers of times, which in turn necessitates that the same is true of the operations on c and b because the products of their numbers in each iteration of the main loop are upper bounds on the numbers of operations on d, and so in particular the final value of c must be correct too. In other words, the gadget is programmed so that correctness of the potentially weak computation on d, which we can check at the end because the function computed is multiplication by a small constant k, implies correctness of the intertwined computation on c, that we cannot check directly like for d because it is division by an exponentially large quantity $(k - 1)!$.

Organisation of the Paper. After the preliminaries in Section 2, our scheme for simulating zero tests of bounded counters is developed in Section 3, and the factorial amplifier for setting up arbitrarily large pairs of Petri net counters with ratio $k!$ is programmed in Section 4.

In Section 5, we put the pieces together to obtain the main result, and then also show how the construction can be refined to establish that, for each positive integer h, we have h-ExpSpace-hardness (tower of exponentials of height h) of the reachability problem already for Petri nets with $h + 13$ counters. In addition, we formulate a generalised Petri net reachability problem (corresponding to the reachability in a generalised vector addition system with states of Kosaraju and Mayr [26, 41, 42]) and show that it is not elementary already for a fixed number of 13 counters.

2 COUNTER PROGRAMS

Proving the main result of this paper, namely that solving the Petri nets reachability problem requires a tower of exponentials of time and space, involves some intricate programming. For ease of presentation, instead of working directly with Petri nets or vector addition systems, our primary language will be imperative programs that operate on variables which are called counters, and that range over the naturals (i.e. the nonnegative integers).

To streamline the main constructions and proofs, it will be useful to allow the programs to have two types of counters:

tested counters are bounded by a fixed positive integer B and may be tested for equality with the end points of their range, i.e. 0 and B;

untested counters are unbounded and the testing commands may not be applied to them.

We remark that the availability of the testing commands will not make counter programs more expressive than Petri nets, because the finiteness of the range of tested counters means that their values can be seen as components of net places (or of states in vector addition systems). However, such an enumerative translation involves a blow up proportional to the bound B.

Definition 1. A counter program is a sequence of commands, each of which is of one of the following five kinds:

x += 1	(increment counter x)
x −= 1	(decrement counter x)
goto L **or** L'	(jump to either line L or line L')
zero? x	(continue if counter x equals 0),
max? x	(continue if counter x equals B),

except that the last command is of the form:

halt if $x_1, \ldots, x_l = 0$ (terminate provided all the listed counters are zero).

Note that the two types of counters are not declared explicitly: without loss of generality, a counter x is regarded as tested (and thus has the range $\{0, \ldots, B\}$) if and only if it occurs in a **zero?** x or **max?** x command in the program.

We use a shorthand **halt** when no counter is required to be zero at termination. □

To illustrate how the available commands can be used to express further constructs, addition x += m and substraction x −= m of a natural constant m can be written as m consecutive increments x += 1 and decrements x −= 1 (respectively). As another illustration, conditional jumps **if** x = 0 **then goto** L **else** x −= 1 which feature in common definitions of Minsky machines can be written as:

1: **goto** 2 **or** 4
2: **zero?** x
3: **goto** L
4: x −= 1,

where **goto** L is a shorthand for the deterministic jump **goto** L **or** L.

We emphasise that counters (both tested and untested) are not permitted to have negative values. In the example we have just seen, that is why the decrement in line 4 works also as a non-zero test.

Three more remarks may be useful.

(1) As we shall see shortly, our focus will be on complete runs, and so for example the nondeterminism introduced in the coding of the conditional jumps above is not an issue since one of the two branches necessarily gets blocked.
(2) Our notion of counter programs only serves as a convenient medium for presenting both Petri nets and Minsky machines with bounded counters, and the exact syntax is not important; we were inspired here by Esparza's presentation [14, Section 7] of Lipton's lower bound [39].
(3) Although the **halt if** $x_1, \ldots, x_l = 0$ commands could be expressed by zero tests followed by just **halt**, having them as atomic commands makes it possible to require untested counters to be zero at termination. The latter feature makes untested counters correspond to Petri net counters, which are unbounded, and can be zero tested only at the start and finish of runs by specifying initial and final configurations in instances of the reachability problem.

2.1 Runs and Computed Relations

A *B-run* of a program from an initial valuation of all its counters is a run in which all values of all counters are at least 0, all values of all tested counters are at most B, and the max tests are interpreted as checks for equality with B.

We say that such a run is *halted* if and only if it has successfully executed its **halt** command (which is necessarily the program's last); otherwise, the run is either *partial* or *infinite*. Observe that, due to a decrement that would cause a counter to become negative, or due to an increment that would exceed the bound of a tested counter, or due to an unsuccesful zero or max test, or due to an unsuccessful terminal check for zero, a partial run may be maximal because it is blocked from further execution. Moreover, due to nondeterministic jumps, the same program from the same initial valuation may have various B-runs in each of the three categories: halted runs, maximal partial runs, and infinite runs. We are mostly going to be interested in final counter valuations that are reached by halted runs.

We regard a run as *complete* if and only if it is halted and its initial valuation assigns zero to every counter. Let $x_1, \ldots, x_l$ be some (not necessarily all) of the counters in the program. We say that the *relation B-computed in* $x_1, \ldots, x_l$

by a program is the set of all tuples $\langle v_1, \ldots, v_l \rangle$ such that the program has a complete B-run whose final valuation assigns to every counter x_i the natural number v_i.

We may consider the same program with more than one bound for its tested counters. When the bound B is clear, or when it is not important because there are no tested counters, we may write simply 'run' and 'computed' instead of 'B-run' and 'B-computed' (respectively).

2.2 Examples

Example 1. Consider the following program, where C is a natural constant, and we observe that all the counters are untested:

1: $\mathsf{x}' \mathrel{+}= C$
2: **goto** 6 **or** 3
3: $\mathsf{x} \mathrel{+}= 1 \quad \mathsf{x}' \mathrel{-}= 1$
4: $\mathsf{y} \mathrel{+}= 2$
5: **goto** 2
6: **halt if** $\mathsf{x}' = 0$.

It repeats the block of three commands in lines 3–4 some number of times chosen nondeterministically (possibly zero, a priori possibly infinite although not here because x' is decreasing) and then halts provided counter x' is zero. Replacing the two jumps by more readable syntactic sugar, we may write this code as:

1: $\mathsf{x}' \mathrel{+}= C$
2: **loop**
3: $\quad \mathsf{x} \mathrel{+}= 1 \quad \mathsf{x}' \mathrel{-}= 1$
4: $\quad \mathsf{y} \mathrel{+}= 2$
5: **halt if** $\mathsf{x}' = 0$.

It is easy to see that there is a unique complete run (and there are no infinite runs), in which the loop is iterated exactly C times. Thus, the relation computed in x, y is the set with the single tuple $\langle C, 2C \rangle$. □

Example 2. We shall need to reason about properties of counter valuations at certain points in programs. As an example which will be useful later for simulating tested counters by untested ones, consider a fixed positive integer B and assume that

$$\mathsf{x} + \hat{\mathsf{x}} \leq B \text{ and } \mathsf{d} \geq \mathsf{c} \cdot B \tag{1}$$

holds in a run at the entry to (i.e., just before executing) the following program fragment.

loop
$\quad \mathsf{x} \mathrel{+}= 1 \quad \hat{\mathsf{x}} \mathrel{-}= 1$
$\quad \mathsf{d} \mathrel{-}= 1$
$\mathsf{c} \mathrel{-}= 1$

The number of times the loop has been iterated by a run that also exits (i.e., completes executing) the program fragment is nondeterministic, so let us denote it by K. It is easy to see that property (1) necessarily also holds at the exit, since:

- the sum $\mathsf{x} + \hat{\mathsf{x}}$ is maintained by each iteration of the loop,
- we have that $K \leq B$, and

- counters d and c have been decreased by K and 1 (respectively).

Continuing the example, if we additionally assume that the exit counter valuation satisfies $\mathsf{d} = \mathsf{c} \cdot B$, then we deduce that:

- necessarily $K = B$,
- $\mathsf{d} = \mathsf{c} \cdot B$ also held at the entry, and
- $\mathsf{x} = 0$ and $\hat{\mathsf{x}} = B$ at the entry, and their values at the exit are swapped.

We have thus seen two small arguments, one based on propagating properties of counter valuations forwards through executions of program fragments, and the other backwards. Both kinds will feature in the sequel. □

2.3 Petri Nets Reachability Problem

It is well known that Petri nets [47], vector addition systems [25], and vector addition systems with states [19, cf. Section 5.1], [21] are alternative presentations of the same model of concurrent processes, in the sense that between each pair there exist straightforward translations that run in polynomial time and preserve the reachability problem; for further details, see e.g. the survey [51, Section 2.1].

Since counter programs without tested counters can be seen as presentations of vector addition systems with states, where the latter are required to start with all vector components zero and to finish with vector components zero as specified by the **halt** command, the Petri nets reachability problem can be stated as:

Petri Nets Reachability
Input A counter program without tested counters.
Question Does it have a complete run?

We remark that restricting further to programs where no counter is required to be zero finally (i.e., where the last command is just **halt**) turns this problem into the Petri nets *coverability* problem. In the terminology of vector addition systems with states, the latter problem is concerned with reachability of just a state, with no requirement on the final vector components. Lipton's ExpSpace lower bound [39] holds already for the coverability problem, which is in fact ExpSpace-complete [48].

2.4 Canonical Problems for Towers of Exponentials

Let us write $!^k$ for the k^{th} iterate of factorial, so that $a!^k = a\overbrace{!\cdots!}^{k}$.

To prove that the Petri nets reachability problem is not elementary, we shall provide a linear-time reduction from the following problem. It is complete for the class Tower of all decision problems that are solvable in time or space bounded by a tower of exponentials whose height is an elementary function of the input size [50, Section 2.3], with respect to elementary reductions.

Tower-Bounded 3-Counter Machines Halting
Input A counter program of size n with 3 counters which are all tested.
Question Does it have a complete $3!^n$-run?

For confirming that this problem is Tower-complete, the reader is referred to [16, Theorem 3.1] for translations between space-bounded 3-counter machines and Turing machines, and to [50, Section 4.1] for the robustness of the class with respect to the choice of the fast-growing function hierarchy (here based on the factorial operation).

We shall also make use of the following problem, which is similarly h-ExpSpace-complete for any positive integer h:

h-Tower-Bounded 3-Counter Machines Halting

Input A counter program of size n with 3 counters which are all tested.

Question Does it have a complete $n!^{h+1}$-run?

3 SIMULATING TESTS

We now introduce our central notion of amplifier for a ratio, and define a special operator for composing them with programs. Provided the ratio of the amplifier is the same as the bound of the program's tested counters, the resulting composition will be an equivalent program in which those counters have become untested. That is accomplished through eliminating the original program's zero and max tests by simulating them and using the amplifier to check that the simulations are correct, where a price to pay is introducing an extra untested counter for each of the original tested ones.

The amplifiers themselves may have tested counters. An amplifier whose tested counters are bounded by B and whose ratio is a larger number R, called a B-amplifier by R, can then be seen, in conjunction with the composition operator, as a means for transforming programs whose tested counters are bounded by R into equivalent programs whose tested counters are bounded by B. In the special case when the amplifier has no tested counters, the same will be true of the resulting programs.

Another feature, which will be key in Section 5, is that more powerful amplifiers will be obtainable by composition: applying the operator to a B-amplifier by B', and B'-amplifier by B'', will produce a B-amplifier by B''.

3.1 Construction

Definition 2. For positive integers B and R, a *B-amplifier by R* is a program such that the relation it B-computes in counters b, c, d is

$$\{\langle b, c, d\rangle \ : \ b = R,\ c > 0,\ d = c \cdot b\}.$$

When the bound B is understood or irrelevant, we may omit it and write *amplifier*. □

Example 3. As an example to be used later, when R is sufficiently small to write R consecutive increments explicitly, it is very easy to code an amplifier by R:

```
1: b += R        → set b to constant R
2: c += 1    d += R
3: loop
4:     c += 1    d += R
5: halt.
```

Observe that this amplifier does not have any tested counters and so, for every positive integer B, it is a B-amplifier by R. □

Assuming $\mathcal{A}$ is a B-amplifier by R and $\mathcal{P}$ is a program, we now define a construction of a program $\mathcal{A} \triangleright \mathcal{P}$ which B-computes any relation that is R-computed by $\mathcal{P}$. The idea is to turn each tested counter x of $\mathcal{P}$ into an untested one through supplementing it by a new counter $\hat{\text{x}}$ and ensuring that the invariant $\text{x} + \hat{\text{x}} = R$ is maintained, so that zero tests of x can be replaced by loops that R times increment x and then R times decrement x, and similarly for max tests. Counter b provided by $\mathcal{A}$ is employed to initialise each complement counter $\hat{\text{x}}$, whereas c and d are used to ensure

that if d is zero at the end of the run then all the loops in the simulations of the zero and max tests iterated R times as required. Concretely, the program $\mathcal{A} \triangleright \mathcal{P}$ consists of the code of $\mathcal{A}$ without its **halt** command followed by the code of $\mathcal{P}$ modified as follows:

(i) counters are renamed if necessary so that no counter occurs in both $\mathcal{A}$ and $\mathcal{P}$;

(ii) letting $x_1, \ldots, x_l$ be the tested counters of $\mathcal{P}$, new counters $\hat{x}_1, \ldots, \hat{x}_l$ are introduced and the following code is inserted at the beginning of $\mathcal{P}$:

```
loop
    x̂1 += 1   ···   x̂l += 1
    b −= 1   d −= 1
c −= 1
```

(we shall show that complete runs necessarily iterate this loop R times, i.e. until counter b becomes zero);

(iii) every x_i += 1 command in $\mathcal{P}$ is replaced by two commands

x_i += 1 $\hat{x}_i$ −= 1;

(iv) every x_i −= 1 command in $\mathcal{P}$ is replaced by two commands

x_i −= 1 $\hat{x}_i$ += 1;

(v) every **zero?** x_i command in $\mathcal{P}$ is replaced by the following code:

```
loop
    xi += 1   x̂i −= 1
    d −= 1
c −= 1
loop
    xi −= 1   x̂i += 1
    d −= 1
c −= 1
```

(we shall show that complete runs necessarily iterate each of the two loops R times, i.e. they check that x_i equals 0 through checking that $\hat{x}_i$ equals R by transferring R from $\hat{x}_i$ to x_i and then back);

(vi) every **max?** x_i command in $\mathcal{P}$ is replaced analogously, i.e. by the code as for **zero?** x_i but with the increments and decrements of x_i and $\hat{x}_i$ swapped;

(vii) letting $y_1, \ldots, y_m$ (respectively, $z_1, \ldots, z_h$) be the counters that are required to be zero at termination of $\mathcal{A}$ (respectively, $\mathcal{P}$), the **halt** command of $\mathcal{P}$ is replaced by

halt if $d, y_1, \ldots, y_m, z_1, \ldots, z_h = 0$.

We remark that simulating zero tests of counters bounded by some R using transfers from and to their complements is a well-known technique that can be found already in Lipton [39]; the novelty here is the cumulative verification of such simulations, through decreasing appropriately the two counters d and c whose ratio is R, and checking that d is zero finally.

3.2 Correctness

Correctness. The next proposition states that the construction of $\mathcal{A} \triangleright \mathcal{P}$ is correct in the sense that its B-computed relations in counters of $\mathcal{P}$ are the same as those R-computed by $\mathcal{P}$. (We shall treat any renamings of counters in step (i) of the construction as implicit.) In one direction, the proof proceeds by observing that $\mathcal{A} \triangleright \mathcal{P}$ can simulate faithfully

any complete R-run of $\mathcal{P}$. In the other direction we argue that although some of the loops introduced in steps (v) and (vi) may iterate fewer than R times and hence erroneously validate a test, the ways in which counters c and d are set up by $\mathcal{A}$ and used in the construction ensure that no such run can continue to a complete one. Informally, as soon as a loop in a simulation of a test iterates fewer than R times, the equality $\mathsf{d} = \mathsf{c} \cdot R$ turns into the strict inequality $\mathsf{d} > \mathsf{c} \cdot R$ which remains for the rest of the run, preventing counter d from reaching zero.

Proposition 3. *For every valuation of counters of $\mathcal{P}$, it occurs after a complete B-run of $\mathcal{A} \triangleright \mathcal{P}$ if and only if it occurs after a complete R-run of $\mathcal{P}$.*

Proof. The 'if' direction is straightforward: from a complete R-run of $\mathcal{P}$ with a total of q zero and max tests, obtain a complete B-run of $\mathcal{A} \triangleright \mathcal{P}$ with the same final valuation of counters of $\mathcal{P}$ by

- running $\mathcal{A}$ to termination with $\mathsf{b} = R$, $\mathsf{c} = 2q + 1$, $\mathsf{d} = \mathsf{c} \cdot R$ and all of $\mathsf{y}_1, \ldots, \mathsf{y}_m$ equal to 0, where the latter counters will remain untouched for the rest of the run and hence satisfy the requirement to be zero finally (cf. step (vii) of the construction),
- iterating the loop in step (ii) R times to initialise each complement counter $\hat{\mathsf{x}}_i$ to R, which also subtracts R and 1 from d and c (respectively) as well as decreases b to 0, and
- in place of every zero or max test in $\mathcal{P}$, iterating both loops in step (v) or (vi) (respectively) R times, which subtracts $2R$ and 2 from d and c (again respectively), eventually decreasing them both to 0.

For the 'only if' direction, consider a complete B-run of $\mathcal{A} \triangleright \mathcal{P}$. Extracting from it a complete R-run of $\mathcal{P}$ with the same final valuation of counters of $\mathcal{P}$ is easy once we show that, for each simulation of a **zero?** x_i or **max?** x_i command by the code in step (v) or (vi) of the construction, the values of x_i at the start and at the finish of the code are 0 or R (respectively).

Firstly, by step (vii) and the fact that counters $\mathsf{y}_1, \ldots, \mathsf{y}_m$ are not used after executing the part of code from $\mathcal{A}$, we have that the values of b, c and d that have been provided by $\mathcal{A}$ satisfy $\mathsf{b} = R$ and $\mathsf{d} = \mathsf{c} \cdot R$. After the code in step (ii) we therefore have that $\mathsf{x}_i + \hat{\mathsf{x}}_i \leq R$ for all i. Recalling the reasoning in Example 2 and arguing forwards through the run, we infer that

$$\mathsf{x}_i + \hat{\mathsf{x}}_i \leq R \text{ for all } i, \text{ and } \mathsf{d} \geq \mathsf{c} \cdot R$$

is an invariant that is maintained by the rest of the run.

Now, due to step (vii) again, d is zero finally, and so the inequality $\mathsf{d} \geq \mathsf{c} \cdot R$ is finally an equality. Therefore, c is zero finally as well. Recalling again the reasoning in Example 2 and arguing backwards through the run, we conclude that in fact $\mathsf{d} = \mathsf{c} \cdot R$ has been maintained and that, for each simulation of a **zero?** x_i or **max?** x_i command, each of the two loops has been iterated exactly R times, and hence the values of x_i at its start and at its finish have been as required. Also, the loop introduced in step (ii) has been iterated R times, and b is zero finally. □

4 FACTORIAL AMPLIFIER

This section is the technical core of the paper. It provides a single program $\mathcal{F}$ called the factorial amplifier which is, for any positive integer k, a k-amplifier by $k!$. Together with the composition operator from Section 3, we shall then have all the tools needed for obtaining our main result in Section 5: chains of compositions of $\mathcal{F}$ with itself will yield amplifiers by ratios which are towers of exponentials.

4.1 A Simpler Program

As a warm up for the presentation of the main program and the proof of its correctness, let us consider a simpler program $\mathcal{E}$ specified in Algorithm I. Two macros are used to aid readability, and we now expand them, noting that hidden within them is another counter i':

$\mathsf{x} \mathrel{-}= \mathsf{i}$**:** To subtract the current value of counter i, we employ the auxiliary counter i' to which the value of i is transferred and then transferred back. At the start of the code, i' is assumed to be zero, and the same is guaranteed at the finish.

loop
  $\mathsf{i} \mathrel{-}= 1 \quad \mathsf{i}' \mathrel{+}= 1$
zero? i
loop
  $\mathsf{i} \mathrel{+}= 1 \quad \mathsf{i}' \mathrel{-}= 1 \quad \mathsf{x} \mathrel{-}= 1$
zero? i'

$\mathsf{x}' \mathrel{+}= \mathsf{i} + 1$**:** This is very similar, except for the extra increment of x'.

$\mathsf{x}' \mathrel{+}= 1$
loop
  $\mathsf{i} \mathrel{-}= 1 \quad \mathsf{i}' \mathrel{+}= 1$
zero? i
loop
  $\mathsf{i} \mathrel{+}= 1 \quad \mathsf{i}' \mathrel{-}= 1 \quad \mathsf{x}' \mathrel{+}= 1$
zero? i'

Algorithm I Counter program $\mathcal{E}$.

//*Untested counters*: x, y, x'
//*Tested counters*: i, i'
1: $\mathsf{i} \mathrel{+}= 1 \quad \mathsf{x} \mathrel{+}= 1 \quad \mathsf{y} \mathrel{+}= 1$
2: **loop**
3: $\mathsf{x} \mathrel{+}= 1 \quad \mathsf{y} \mathrel{+}= 1$
4: **loop**
5: **loop**
6: $\mathsf{x} \mathrel{-}= \mathsf{i} \quad \mathsf{x}' \mathrel{+}= \mathsf{i} + 1$
7: **loop**
8: $\mathsf{x}' \mathrel{-}= 1 \quad \mathsf{x} \mathrel{+}= 1$
9: $\mathsf{i} \mathrel{+}= 1$
10: **max?** i
11: **loop**
12: $\mathsf{x} \mathrel{-}= \mathsf{i} \quad \mathsf{y} \mathrel{-}= 1$
13: **halt if** $\mathsf{y} = 0$.

Program $\mathcal{E}$ has untested counters x, x' and y, and tested counters i and i'. Assuming that the bound for the tested counters is a positive integer k, the program does the following:

- initialises x and y to some positive integer a chosen nondeterministically, which will be kept unchanged in counter y until the final loop;

- in each iteration of the main loop, uses counter x′ to attempt to multiply counter x by the fraction (i + 1)/i;
- by the final loop and the terminal check that counter y is zero, halts provided the value of x is at least $a \cdot k$ (in which case it will be exactly $a \cdot k$).

The first and easier part of the exercise is to show that, for any positive a, there exists a complete k-run of $\mathcal{E}$ that initialises x and y to a, then multiplies x exactly by all the fractions $(i+1)/i$ for $i = 1, \ldots, k-1$, and finally checks that x equals $a \cdot k$.

The second part is to show the converse, i.e. that any complete k-run of $\mathcal{E}$ is of that form. As a hint, we remark that this is the case because as soon as a multiplication of x by a fraction $(i+1)/i$ does not complete accurately (because either the first inner loop does not decrease x to exactly zero, or the second inner loop does not decrease x′ to exactly zero), it will not be possible to repair that error in the rest of the run, in the sense that the value of x at the end of the main loop will necessarily be strictly smaller than $a \cdot k$ and thus it will be impossible to complete the run. We also remark that this vitally depends on the fact that all the fractions $(i+1)/i$ are greater than 1.

4.2 The Amplifier

The definition of $\mathcal{F}$ in Algorithm II is presented at a high level for readability. In addition to the two macros for subtracting i and adding i + 1 presented in the previous subsection, one further macro is used:

loop at most b **times** <*body*>**:** To express this construct, we employ the auxiliary counter b′ to which the value of b is transferred and then transferred back. Provided b′ is zero at the start, the body is indeed performed at most b times.

loop
  b −= 1 b′ += 1
loop
  b += 1 b′ −= 1
  <*body*>

Observe that the untested counters of program $\mathcal{F}$ are b, b′, c, c′, d, d′, x and y, and the tested ones are i and i′ (counter i′ is hidden in the macros).

4.3 Correctness

Lemma 4. *For any positive integer k, the program $\mathcal{F}$ is a k-amplifier by k!, i.e. the relation it k-computes in counters* b, c, d *is*

$$\{\langle b, c, d\rangle \ : \ b = k!,\ c > 0,\ d = c \cdot b\}.$$

Proof. Before presenting this main technical argument in the paper, we provide some intuitions:

- the counter d is used to preserve the value of x at the end of the main loop, since x is modified in the final loop;
- the counter d′ acts as the auxiliary counter for both d and x, so there is no need to have x′ as well;
- the counter c is initialised to the same positive integer a as d, x and y, whereas the counter b is initialised to 1;
- at the start of any iteration of the main loop in a complete run, the invariant d = c · b will hold, and so the first inner loop will divide c by i accurately;
- in order for the last inner loop to transfer d′ fully to d and x, the middle two inner loops will necessarily multiply b by i + 1 accurately;

Algorithm II Factorial amplifier $\mathcal{F}$.

```
   //Untested counters: b, b′, c, c′, d, d′, x, y
   //Tested counters: i, i′
1: i += 1   b += 1   c += 1   d += 1   x += 1   y += 1
2: loop
3:     c += 1   d += 1   x += 1   y += 1
4: loop
5:     loop
6:         c −= i   c′ += 1
7:         loop at most b times
8:             d −= i   x −= i   d′ += i + 1
9:     loop
10:        b −= 1   b′ += i + 1
11:    loop
12:        b′ −= 1   b += 1
13:    loop
14:        c′ −= 1   c += 1
15:        loop at most b times
16:            d′ −= 1   d += 1   x += 1
17:    i += 1
18: max? i
19: loop
20:    x −= i   y −= 1
21: halt if y = 0.
```

- at the end of the main loop, d, c and b will have values $a \cdot k$, $a/(k-1)!$ and $k!$ (respectively), and in particular a is necessarily divisible by $(k-1)!$.

To start the proof, we shall be considering k-runs of $\mathcal{F}$ whose initial valuation assigns zero to every counter, and which are either halted or blocked at the **halt** command because y is not zero. In particular, any such run will have completed the main loop, which runs for $\mathsf{i} = 1, \ldots, k-1$. Hence, we can introduce the following notations for counter values during the i^{th} iteration of the loop, where v is any of the counters b, b′, c, c′, d, d′:

$\bar{\mathsf{v}}_i$**:** the final value of v after lines 5–10;

v_i**:** the final value of v after lines 11–16.

It will also be convenient to write v_0 for the value of v at the start of the first iteration of the main loop. We emphasise that these notations are relative to the run under consideration, which for readability is not written explicitly.

The proof works for any positive integer k and consists of two parts, that establish the two inclusions between the relation k-computed by $\mathcal{F}$ in counters b, c, d and the relation in the statement of the lemma.

We first argue the easier direction, where we assume $b = k!$, $c > 0$ and $d = c \cdot b$, and show that $\mathcal{F}$ has a complete k-run whose final values of counters b, c, d are exactly b, c, d.

Claim 1. For any a divisible by $(k-1)!$, the program $\mathcal{F}$ has a complete k-run which satisfies the equalities in Table 1.

Proof of Claim 1. Such a run can be built by iterating each inner nondeterministic loop the maximum number of times. Namely, during iteration i of the main loop:

Table 1. Equalities for counter values in complete k-runs of program $\mathcal{F}$, for all $i = 0, \ldots, k-1$.

b_0	$=$	1		c_0	$=$	a		d_0	$=$	a
b'_0	$=$	0		c'_0	$=$	0		d'_0	$=$	0
$\bar{\mathsf{b}}_i$	$=$	0		$\bar{\mathsf{c}}_i$	$=$	0		$\bar{\mathsf{d}}_i$	$=$	0
$\bar{\mathsf{b}}'_i$	$=$	$\mathsf{b}_{i-1} \cdot (i+1)$		$\bar{\mathsf{c}}'_i$	$=$	c_{i-1}/i		$\bar{\mathsf{d}}'_i$	$=$	$\mathsf{d}_{i-1} \cdot (i+1)/i$
b_i	$=$	$\bar{\mathsf{b}}'_i$		c_i	$=$	$\bar{\mathsf{c}}'_i$		d_i	$=$	$\bar{\mathsf{d}}'_i$
b'_i	$=$	0		c'_i	$=$	0		d'_i	$=$	0

- the loop at line 5 is iterated c_{i-1}/i times and in each pass the loop at line 7 is iterated b_{i-1} times;
- the loop at line 9 is iterated b_{i-1} times;
- the loop at line 11 is iterated $\bar{\mathsf{b}}_i$ times;
- the loop at line 13 is iterated $\bar{\mathsf{c}}'_i$ times and in each pass the loop at line 15 is iterated b_i times.

The divisibility of a by $(k-1)!$ ensures that all divisions in the statement of the claim yield integers.

To see that the run thus obtained can be completed, observe that from the equalities in Table 1 it follows that

$$\mathsf{b}_{k-1} = \prod_{i=1}^{k-1} (i+1) = k! \qquad \mathsf{c}_{k-1} = a \cdot \prod_{i=1}^{k-1} \frac{1}{i} = \frac{a}{(k-1)!} \qquad \mathsf{d}_{k-1} = a \cdot \prod_{i=1}^{k-1} \frac{i+1}{i} = a \cdot k.$$

In particular, at the start of the final loop (at line 19), counter x equals counter d and hence has value $a \cdot k$, and counter y has value a. Iterating the final loop a times therefore reduces y (and x) to zero as required. □

To obtain b, c, d as the final values of counters b, c, d, we apply Claim 1 with $a = c \cdot (k-1)!$.

We now turn to the remaining second part of the proof of the lemma, where we consider any complete k-run and need to show that the final values b, c, d of counters b, c, d satisfy $b = k!$, $c > 0$ and $d = c \cdot b$.

Claim 2. For all $i = 1, \ldots, k-1$, we have:

- $\bar{\mathsf{d}}_i + \bar{\mathsf{d}}'_i \leq (\mathsf{d}_{i-1} + \mathsf{d}'_{i-1}) \cdot (i+1)/i$;
- $\bar{\mathsf{d}}_i + \bar{\mathsf{d}}'_i = (\mathsf{d}_{i-1} + \mathsf{d}'_{i-1}) \cdot (i+1)/i$ if and only if $\bar{\mathsf{d}}_i = \mathsf{d}'_{i-1} = 0$;
- $\mathsf{d}_i + \mathsf{d}'_i = \bar{\mathsf{d}}_i + \bar{\mathsf{d}}'_i$.

Proof of Claim 2. Straightforward calculation based on $(i+1)/i > 1$. □

Let a denote the value of counters c, d, x and y at the start of the main loop.

Claim 3. The equalities in Table 1 for the values of counters d and d′ are satisfied.

Proof of Claim 3. First, recall that at the start of the final loop (at line 19) counters x and d are equal, and by Claim 2 they have value at most $a \cdot k$. Since counter y has value a at that point and the run is complete, it must actually be the case that the value of x here equals $a \cdot k$. By Claim 2 again, we infer that for all $i = 1, \ldots, k-1$, we indeed have:

$$\bar{\mathsf{d}}_i = 0 \qquad \bar{\mathsf{d}}'_i = \mathsf{d}_{i-1} \cdot \frac{i+1}{i} \qquad \mathsf{d}_i = \bar{\mathsf{d}}'_i \qquad \mathsf{d}'_i = 0.$$ □

Claim 4. We have that a is divisible by $(k-1)!$ and that the equalities in Table 1 for the values of counters b, b′, c and c′ are satisfied.

Proof of Claim 4. That a is divisible by $(k-1)!$ will follow once we establish the equalities for the values of c and c′, since they involve dividing a by $(k-1)!$.

For the rest of the claim, we argue inductively, where the hypothesis is that the equalities for the values of b, b′, c and c′ are satisfied for all indices less than i. Consequently, recalling Claim 3, we have that

$$\mathrm{d}_{i-1} = \mathrm{c}_{i-1} \cdot \mathrm{b}_{i-1}. \tag{2}$$

Consider the iteration i of the main loop. We infer from Claim 3 that the commands in line 8 must have been performed d_{i-1}/i times. Hence, as the values of counters b and b′ remain unchanged until line 9, using equation (2) we deduce that the commands in line 6 must have been performed c_{i-1}/i times, and we have:

$$\bar{\mathrm{c}}_i = 0 \qquad \bar{\mathrm{c}}'_i = \mathrm{c}_{i-1}/i.$$

Also by Claim 3, the commands in line 16 must have been performed $\bar{\mathrm{d}}'_i = \mathrm{d}_{i-1} \cdot (i+1)/i$ times. From what we have just shown, that number equals $\bar{\mathrm{c}}'_i \cdot \mathrm{b}_{i-1} \cdot (i+1)$, and so we conclude that indeed:

$$\begin{array}{lllllllll} \bar{\mathrm{b}}_i & = & 0 & & \mathrm{b}_i & = & \bar{\mathrm{b}}'_i & \mathrm{c}_i & = & \bar{\mathrm{c}}'_i \\ \bar{\mathrm{b}}'_i & = & \mathrm{b}_{i-1} \cdot (i+1) & & \mathrm{b}'_i & = & 0 & \mathrm{c}'_i & = & 0. \end{array}$$ □

As in the first part, we now conclude that the final values b, c, d of counters b, c, d are $k!, a/(k-1)!, a \cdot k$, and in particular $c \cdot b = a \cdot k!/(k-1)! = d$. □

5 MAIN RESULTS

As already indicated, the bulk of the work for our headline result, namely Tower-hardness of the reachability problem for Petri nets, is showing how to construct an amplifier without tested counters and for a ratio which is a tower of exponentials. Most of the pieces have already been developed in Sections 3 and 4, and here we put them together to obtain a linear-time construction (although any elementary complexity of the reduction would suffice for the Tower-hardness).

Lemma 5. *An amplifier by $3!^n$ without tested counters is computable in time $O(n)$.*

Proof. Letting $\mathcal{A}$ be a trivial amplifier by 3 (cf. Example 3), the program

$$\overbrace{((\mathcal{A} \rhd \mathcal{F}) \rhd \mathcal{F}) \rhd \cdots \mathcal{F}}^{n \text{ compositions}}$$

is an amplifier by $3!^n$ without tested counters by Proposition 3 and Lemma 4, and it is computable in time $O(n)$ by the definition of the composition operator (cf. Section 3). □

Theorem 6. *The Petri nets reachability problem is* Tower*-hard.*

Proof. We reduce in linear time from the Tower-complete halting problem for counter programs of size n with 3 counters which are all tested and bounded by $3!^n$ (cf. Section 2).

Let $\mathcal{M}$ be such a program, and let $\mathcal{T}$ be an amplifier by $3!^n$ without tested counters which is computable in time $O(n)$ by Lemma 5. We have that the composite program $\mathcal{T} \rhd \mathcal{M}$ is without tested counters, and that by Proposition 3 it has a complete run if and only if the given program $\mathcal{M}$ does. □

Our further reductions below build on the following refinement of Lemma 5, whose proof is delegated to Section 6:

Lemma 7. *For every $n \geq 1$ and $h \geq 0$, an amplifier $\mathcal{T}$ by $n!^{h+1}$ without tested counters and with $h + 13$ untested ones of the form*

$\mathcal{H}_0$
$\mathcal{H}_1$
$\dots$
$\mathcal{H}_{h+1}$
halt if $\mathrm{d}_0, \mathrm{d}_1, \dots, \mathrm{d}_h = 0$

(for some program fragments $\mathcal{H}_0, \dots, \mathcal{H}_{h+1}$) is computable in time $O(n + h)$ such that:

(I) *in* 3 *out of* 12 *counters not appearing in the* ***halt*** *command (say* $\mathrm{b}, \mathrm{c}, \mathrm{d}_{h+1}$*) the program $\mathcal{T}$ computes its output relation, and the remaining* 9 *counters are zero at termination of every complete run of $\mathcal{T}$,*[4]

(II) *counter d_j is only modified in program fragments $\mathcal{H}_j$ and $\mathcal{H}_{j+1}$, for $j = 0, \dots, h$, and counter d_{h+1} only in $\mathcal{H}_{h+1}$.*

Theorem 8. *For any positive integer h, the Petri nets reachability problem with $h + 13$ counters is h-*ExpSpace*-hard.*[5]

Proof. We reduce in linear time from the h-ExpSpace-complete halting problem for counter programs of size n with 3 counters which are all tested and bounded by $n!^{h+1}$ (cf. Section 2), using the amplifier $\mathcal{T}$ given by Lemma 7.

Given such a program $\mathcal{M}$, the reduction builds the composite program $\mathcal{T} \triangleright \mathcal{M}$, analogously as in the proof of Theorem 6. In order to keep the number of counters in $\mathcal{T} \triangleright \mathcal{M}$ not greater than $h + 13$, we reuse 6 out of the 9 counters not appearing in the **halt** command of $\mathcal{T}$ (and forced to be zero at termination of $\mathcal{T}$) for simulation of the 3 counters of $\mathcal{M}$. □

5.1 Generalised Petri Nets Reachability Problem

The first proof of decidability of the reachability problem by Mayr and Kosaraju [26, 41, 42] applies actually to a more general model of *generalised vector addition system with states* that is, roughly, a sequence of Petri nets (or vector addition systems, or counter programs without tested counters) interconnected by zero-test transitions. In the subsequent simplified proofs essentially the same generalised model appeared under different names: marked graph-transition sequences in [27], marked witness graph sequences in [34], and KLM sequences in [35]. As worked out by Leroux and Schmitz in [34], the generalised model is not ad-hoc as it tightly corresponds to ideals of pre-runs.

In this section we show a sharper lower bound for the generalised model. To this aim we consider the following extensions of counter programs, amplifiers and associated basic notions:[6]

Definition 9. A *generalised counter program* is a concatenated sequence of counter programs, with notions of tested counter and untested counter the same as in Section 2. We write all the **halt** commands except at the end of the last component counter program as **pass**.

Runs are defined as in Section 2.1, where the **pass** commands act as zero tests, but without causing the counters that appear in them to be classified as tested. Similarly, halted runs are those that successfully execute the **halt** command (which is at the end of the last component counter program), and complete runs in addition have all counters that occur in the generalised counter program initially zero.

Computed relations are defined as in Section 2.1, in terms of valuations at the ends of complete runs.

Generalised amplifiers, and their compositions with generalised counter programs, are then defined as in Section 3. □

[4] Hence, 9 counters of $\mathcal{T}$ are *forced* to be zero at termination of every complete run, without being tested to be so.
[5] We remark that, in the terminology of the classical definition of Petri nets [47] (or vector addition systems [25]), the number of places (or dimension) will be $h + 16$ due to 3 extra places for encoding the control of counter programs.
[6] An alternative approach to obtaining a counterpart to generalised VASS in terms of counter programs is to replace **goto** commands by **loop** constructs and then allow only zero tests that occur outside loops.

We can now state the generalised Petri nets reachability problem as:

GENERALISED PETRI NETS REACHABILITY
Input A generalised counter program without tested counters.
Question Does it have a complete run?

Theorem 10. *The generalised Petri nets reachability problem with 13 counters is* TOWER*-hard.*[7]

PROOF. We proceed analogously to the proofs of Theorems 6 and 8, reducing in linear time from the TOWER-complete halting problem for counter programs of size n with 3 counters which are all tested and bounded by $3!^n$ (cf. Section 2).

Let $\mathcal{M}$ be such a program, and let $\mathcal{T}$ be an amplifier by $3!^n$ without tested counters and with $n+12$ untested ones from Lemma 7 (with 3 and n in places of n and $h+1$, respectively), computable in time $O(n)$. As every counter d_j of $\mathcal{T}$ is only modified in program fragments $\mathcal{H}_j$ and $\mathcal{H}_{j+1}$, the idea is to perform a zero test on every d_j as soon as execution of $\mathcal{H}_{j+1}$ is completed, and reuse d_j instead of d_{j+2} in the sequel. We transform $\mathcal{T}$ into an equivalent program that uses only 2 counters d_0 and d_1 instead of the $n+1$ counters $\mathsf{d}_0, \ldots, \mathsf{d}_n$, by

- replacing every counter d_j by $\mathsf{d}_{j \bmod 2}$, for $j = 2, \ldots, n$,
- inserting after every fragment $\mathcal{H}_{j+1}$ the command **pass if** $\mathsf{d}_{j \bmod 2} = 0$, for $j = 0 \ldots n-2$, and
- adjusting the final **halt** command to test only counter $\mathsf{d}_{n-1 \bmod 2}$.

This yields a generalised counter program $\mathcal{T}'$ of the following form, whose output relation is computed in counters $\mathsf{b}, \mathsf{c}, \mathsf{d}_{n \bmod 2}$:

$\mathcal{H}'_0$
$\mathcal{H}'_1$
pass if $\mathsf{d}_0 = 0$
$\mathcal{H}'_2$
pass if $\mathsf{d}_1 = 0$
$\ldots$
$\mathcal{H}'_{n-1}$
pass if $\mathsf{d}_{n-2 \bmod 2} = 0$
$\mathcal{H}'_n$
halt if $\mathsf{d}_{n-1 \bmod 2} = 0$

which is a generalised amplifier by $3!^n$ using only 13 counters.

The composite generalised counter program $\mathcal{T}' \triangleright \mathcal{M}$ has a complete run if and only if the given program $\mathcal{M}$ does. Finally, as before we keep the number of counters in $\mathcal{T}' \triangleright \mathcal{M}$ not greater than 13 by reusing 6 out of the 9 counters forced to be zero at termination of $\mathcal{T}'$ for simulation of the 3 counters of $\mathcal{M}$. □

6 PROOF OF LEMMA 7

We proceed by analysing closely the composite amplifier constructed in the proof of Lemma 5, performing a careful sequence of gradual transformations that reduce the number of counters by optimisations such as merging and reusing, and along the way providing arguments for its correctness.

From the analysis of the factorial amplifier $\mathcal{F}$ in the proof of Lemma 4 we derive the following claim.

[7] As before, in the terminology of classical Petri nets (or vector addition systems), the number of places (or dimension) will be 16.

Claim 5. Let $k > 0$. At termination of every complete k-run of $\mathcal{F}$, the counters $\mathsf{x}, \mathsf{d}', \mathsf{c}', \mathsf{b}', \mathsf{i}'$ are all zero, and the counter i equals k.

Let $\mathcal{A}$ be the program obtained from a trivial amplifier by n (cf. Example 3):

1: b_0 += n
2: c_0 += 1 d_0 += n
3: **loop**
4: c_0 += 1 d_0 += n
5: **halt**.

Similarly as in the proof of Lemma 5, the composite program

$$\mathcal{T} = \overbrace{((\mathcal{A} \rhd \mathcal{F}) \rhd \mathcal{F}) \rhd \cdots \mathcal{F}}^{h+1 \text{ compositions}}$$

is an amplifier by $n!^{h+1}$ without tested counters, computable in time $O(n + h)$. Our aim is to gradually optimise $\mathcal{T}$ until it finally satisfies the requirements of Lemma 7. For defining the optimisations we expand the $h + 1$ composition operations (cf. Section 3), and rename the counters explicitly (the counters of $\mathcal{A}$ are indexed by 0, and counters of the jth program $\mathcal{F}$, for $j = 1, \ldots, h + 1$, are indexed by j); this yields the following expanded form of $\mathcal{T}$:

$\mathcal{A}'$
$\mathcal{H}_1$
...
$\mathcal{H}_{h+1}$
halt if $\mathsf{d}_0, \mathsf{y}_1, \mathsf{d}_1, \ldots, \mathsf{y}_h, \mathsf{d}_h, \mathsf{y}_{h+1} = 0$

where $\mathcal{A}'$ is the program fragment obtained by removing the last line 5 from $\mathcal{A}$, and $\mathcal{H}_j$ is the program fragment shown in Algorithm III (without the boxed commands, which will be used only later to reduce the number of counters). Except for the renaming of counters, the program fragment $\mathcal{H}_j$ differs from the factorial amplifier $\mathcal{F}$ only by explicit manipulation of the counters $\hat{\mathsf{i}}_j, \hat{\mathsf{i}}'_j$ (introduced by the composition operations).

The program $\mathcal{T}$ makes use of the following macros depending on j. The macro **setup** $\hat{\mathsf{i}}_j, \hat{\mathsf{i}}'_j$ initialises the variables $\hat{\mathsf{i}}_j, \hat{\mathsf{i}}'_j$ to at most b_{j-1} (cf. point (ii) in Section 3):

setup $\hat{\mathsf{i}}_j, \hat{\mathsf{i}}'_j$**:**
  loop
    $\hat{\mathsf{i}}_j$ += 1 $\hat{\mathsf{i}}'_j$ += 1
    b_{j-1} −= 1 d_{j-1} −= 1
  c_{j-1} −= 1.

The macro **ismax** i_j follows points (v) and (vi) in Section 3 (we omit the analogous macrros **iszero** i_j and **iszero** i'_j):

ismax i_j**:**
  loop
    i_j −= 1 $\hat{\mathsf{i}}_j$ += 1
    d_{j-1} −= 1
  c_{j-1} −= 1
  loop
    i_j += 1 $\hat{\mathsf{i}}_j$ −= 1
    d_{j-1} −= 1
  c_{j-1} −= 1.

The macro x_j −= i_j is an adaptation of the macro from Section 4.1 respecting the points (iii) and (iv) in Section 3 (we omit the analogous macros c_j −= i_j, d_j −= i_j, d'_j += $\mathsf{i}_j + 1$ and b'_j += $\mathsf{i}_j + 1$):

x_j −= i_j**:**
  loop
    i_j −= 1 $\hat{\mathsf{i}}_j$ += 1 i'_j += 1 $\hat{\mathsf{i}}'_j$ −= 1
  iszero i_j
  loop
    i_j += 1 $\hat{\mathsf{i}}_j$ −= 1 i'_j −= 1 $\hat{\mathsf{i}}'_j$ += 1 x_j −= 1
  iszero i'_j.

Finally, the macro **loop at most** b_j **times** <*body*> is essentially copied from Section 4:

loop at most b_j **times** <*body*>**:**
  loop
    b_j −= 1 b'_j += 1
  loop
    b_j += 1 b'_j −= 1
    <*body*>.

The counters $\mathsf{b}_j, \mathsf{c}_j, \mathsf{d}_j$, for $j = 0, \ldots, h+1$, we call *ratio* counters, and the remaining ones *non-ratio* counters. Here are two straightforward but crucial observations following from the above explicit construction of $\mathcal{T}$.

Claim 6. For $j = 1, \ldots, h+1$, the program fragment $\mathcal{H}_j$ only modifies the j-indexed counters, plus the three ratio counters $\mathsf{b}_{j-1}, \mathsf{c}_{j-1}, \mathsf{d}_{j-1}$.

Claim 7. In every complete run of $\mathcal{T}$, at the end of $\mathcal{H}_j$ for every $j = 1, \ldots, h+1$, all non-ratio j-indexed counters are zero, except for the two counters $\mathsf{i}_j, \hat{\mathsf{i}}'_j$ which are equal to the value of b_{j-1} at the beginning of $\mathcal{H}_j$.

Proof. By the analysis in the proof of Proposition 3 we deduce that the counters b_{j-1} and c_{j-1} are zero. By Claim 5, at the end of the program fragment $\mathcal{H}_j$, not only the counter y_j (which is later checked to be zero by the **halt** command — cf. Claim 6) but actually all j-indexed non-ratio counters are zero, except for the two counters $\mathsf{i}_j, \hat{\mathsf{i}}'_j$. It is readily verified that these two counters will be equal to k (the value of b_{j-1} at the beginning of $\mathcal{H}_j$) due to the invariants we keep along every complete run: $\mathsf{i} + \hat{\mathsf{i}} = k$ and $\mathsf{i}' + \hat{\mathsf{i}}' = k$. □

Algorithm III Program fragment $\mathcal{H}_j$ (without the boxed commands); program fragment $\mathcal{H}_j^{(1)}$ (with the boxed commands).

1: **setup** $\mathring{\mathsf{i}}_j, \mathring{\mathsf{i}}'_j$
2: i_j += 1 $\mathring{\mathsf{i}}_j$ −= 1 (cf. point (iii) in Section 3)
3: b_j += 1 c_j += 1 d_j += 1 x_j += 1 y_j += 1 $\boxed{\mathsf{d}_{j-1}\ \text{+= } 1}$
4: **loop**
5: c_j += 1 d_j += 1 x_j += 1 y_j += 1 $\boxed{\mathsf{d}_{j-1}\ \text{+= } 1}$
6: **loop**
7: **loop**
8: c_j −= i_j c'_j += 1
9: **loop at most** b_j **times**
10: d_j −= i_j x_j −= i_j d'_j += $\mathsf{i}_j + 1$
11: **loop**
12: b_j −= 1 b'_j += $\mathsf{i}_j + 1$
13: **loop**
14: b'_j −= 1 b_j += 1
15: **loop**
16: c'_j −= 1 c_j += 1
17: **loop at most** b_j **times**
18: d'_j −= 1 d_j += 1 x_j += 1
19: i_j += 1 $\mathring{\mathsf{i}}_j$ −= 1 (cf. point (iii) in Section 3)
20: **ismax** i_j
21: **loop**
22: x_j −= i_j y_j −= 1 $\boxed{\mathsf{d}_{j-1}\ \text{−= } 1}$
23: $\boxed{\textbf{reset } \mathsf{i}_j, \mathring{\mathsf{i}}'_j}$

The first optimisation of the program $\mathcal{T}$ builds on Claim 7. We enforce the counters $\mathsf{i}_j, \mathring{\mathsf{i}}'_j$ to be zero by adjoining at the end of $\mathcal{H}_j$ a macro defined by the following piece of code (depicted as a boxed command in line 23 in Algorithm III):

reset $\mathsf{i}_j, \mathring{\mathsf{i}}'_j$**:**
 loop
 i_j −= 1 $\mathring{\mathsf{i}}'_j$ −= 1
 d_{j-1} −= 1
 c_{j-1} −= 1

Denoting by $\mathcal{H}_j^{(0)}$ the so modified program fragments, and by $\mathcal{T}^{(0)}$ the corresponding program

$\mathcal{A}'$
$\mathcal{H}_1^{(0)}$
...
$\mathcal{H}_{h+1}^{(0)}$
halt if $\mathsf{d}_0, \mathsf{y}_1, \mathsf{d}_1, \ldots, \mathsf{y}_h, \mathsf{d}_h, \mathsf{y}_{h+1} = 0$,

we summarise.

Claim 8. The program $\mathcal{T}^{(0)}$ is an amplifier by $n!^{h+1}$ without tested counters.

Recalling Claims 6 and 7, one easily demonstrates the following property of $\mathcal{T}^{(0)}$:

Claim 9. The program $\mathcal{T}^{(0)}$ is *cleaning*: in every complete run of $\mathcal{T}^{(0)}$, at the end of the program fragment $\mathcal{H}_j^{(0)}$ for every $j = 1, \ldots, h+1$, all counters are zero except for j-indexed ratio ones.

In the next optimisation, we reduce the number of counters that are required to be zero by the terminal command **halt** from $2h+2$ to $h+1$. Consider the following further modification of $\mathcal{H}_j^{(0)}$ depicted by boxed commands in lines 3, 5 and 22 in Algorithm III:

(1) insert the instruction d_{j-1} += 1 in lines 3 and 5 (which results in additional increasing the value of d_{j-1} by the value ultimately achieved by y_j), and
(2) insert the instruction d_{j-1} −= 1 in line 22 (since it is simultaneously decreased with y_j it results in decreasing the value of d_{j-1} by at most the value of the additional increase).

Observe that in complete runs, the decrease of d_{j-1} in (2) being equal to the increase in (1) is equivalent to y_j being zero at the end of $\mathcal{H}_j^{(1)}$. Denote by $\mathcal{H}_j^{(1)}$ the resulting program fragment, and by $\mathcal{T}^{(1)}$ the corresponding program that requires only counters $\mathsf{d}_0, \ldots, \mathsf{d}_h$ to be zero at termination:

$\mathcal{A}'$
$\mathcal{H}_1^{(1)}$
...
$\mathcal{H}_{h+1}^{(1)}$
halt if $\mathsf{d}_0, \mathsf{d}_1, \ldots, \mathsf{d}_h = 0$.

Claim 10. The program $\mathcal{T}^{(1)}$ is a cleaning amplifier by $n!^{h+1}$ without tested counters.

Proof. To confirm the claim we need to demonstrate that every counter y_j is *forced* to be zero at the end of $\mathcal{H}_j^{(1)}$ in every complete run of $\mathcal{T}^{(1)}$. Indeed, observe that instead of the invariant $\mathsf{d}_{j-1} \geq \mathsf{c}_{j-1} \cdot k$ (cf. the proof of Proposition 3) we have now the invariant $\mathsf{d}_{j-1} \geq \mathsf{c}_{j-1} \cdot k + \mathsf{y}_{j-1}$. In consequence, when the counter d_{j-1} is zero, the counter y_j is necessarily zero too. □

We optimise further by collapsing all the counters $\mathsf{x}_1, \ldots, \mathsf{x}_{h+1}$ into one counter x, and analogously for all other non-ratio counters. In particular, since the counters $\mathsf{y}_1, \ldots, \mathsf{y}_{h+1}$ do not appear anymore in the terminal **halt** command of $\mathcal{T}^{(1)}$, we replace all these counters with one counter y. We thus obtain new program fragments $\mathcal{H}_j^{(2)}$, shown in Algorithm IV, each of them using the same 9 non-ratio counters $\mathsf{i}, \hat{\mathsf{i}}, \mathsf{i}', \hat{\mathsf{i}}', \mathsf{b}', \mathsf{c}', \mathsf{d}', \mathsf{x}, \mathsf{y}$, plus six ratio counters $\mathsf{b}_{j-1}, \mathsf{c}_{j-1}, \mathsf{d}_{j-1}, \mathsf{b}_j, \mathsf{c}_j, \mathsf{d}_j$ (the macros used in $\mathcal{H}_j^{(2)}$ are adjusted accordingly). The optimisation results in a new program $\mathcal{T}^{(2)}$:

$\mathcal{A}'$
$\mathcal{H}_1^{(2)}$
...
$\mathcal{H}_{h+1}^{(2)}$
halt if $\mathsf{d}_0, \mathsf{d}_1, \ldots, \mathsf{d}_h = 0$

satisfying the following claim.

Claim 11. The program $\mathcal{T}^{(2)}$ is a cleaning amplifier by $n!^{h+1}$ without tested counters such that:

- it has $3(h+2)+9 = 3h+15$ untested counters;
- $h+1$ counters are required to be zero by the terminal **halt** command;
- all other counters except for $\mathsf{b}_{h+1}, \mathsf{c}_{h+1}, \mathsf{d}_{h+1}$ are forced to be zero at termination of every complete run.

The last condition follows since $\mathcal{T}^{(2)}$ is cleaning. In particular, we know that all the counters $\mathsf{b}_0, \ldots, \mathsf{b}_h$ and $\mathsf{c}_0, \ldots, \mathsf{c}_h$ are necessarily zero at the termination of every complete run of $\mathcal{T}^{(2)}$, which will underly our further (final) optimisations.

Algorithm IV Program fragment $\mathcal{H}_j^{(2)}$.

```
1:  setup i, i'
2:  i += 1    i -= 1
3:  b_j += 1    c_j += 1    d_j += 1    x += 1    y += 1    d_{j-1} += 1
4:  loop
5:      c_j += 1    d_j += 1    x += 1    y += 1    d_{j-1} += 1
6:  loop
7:      loop
8:          c_j -= i    c' += 1
9:          loop at most b_j times
10:             d_j -= i    x -= i    d' += i + 1
11:     loop
12:         b_j -= 1    b' += i + 1
13:     loop
14:         b' -= 1    b_j += 1
15:     loop
16:         c' -= 1    c_j += 1
17:         loop at most b_j times
18:             d' -= 1    d_j += 1    x += 1
19:     i += 1    i -= 1
20: ismax i
21: loop
22:     x -= i    y -= 1    d_{j-1} -= 1
23: reset i, i'
```

First, as every c_j is only used in $\mathcal{H}_j^{(2)}$ and $\mathcal{H}_{j+1}^{(2)}$, and is zero at termination (and thus also at the end of $\mathcal{H}_{j+1}^{(2)}$) in every complete run, instead of $h+2$ counters $\mathsf{c}_0, \ldots, \mathsf{c}_{h+1}$ we can use in an alternating manner just two, denoted c_0 and c_1.

Second, as every b_{j-1} is zero at termination in every complete run, and not modified after line 1 in $\mathcal{H}_j^{(2)}$, it becomes forcedly zero before the next counter b_j is modified by $\mathcal{H}_j^{(2)}$. Therefore, all $h+2$ counters $\mathsf{b}_0, \ldots, \mathsf{b}_{h+1}$ can be collapsed to just one counter b.

Applying these two optimisations, together with a further improvement in macros (to be expanded below) to eliminate one more counter, yields our final version of the program fragment $\mathcal{H}_j^{(3)}$ as shown in Algorithm V. The **loop at most** b **times** macro is exactly the same as in Section 4, and the **ismax** i macro appearing in $\mathcal{H}_j^{(3)}$ is as follows:

```
ismax i:
    loop
        i -= 1    i' += 1    d_{j-1} -= 1
    c_{j-1 mod 2} -= 1
```

loop
  $\mathsf{i} \mathrel{+}= 1 \quad \mathsf{i}' \mathrel{-}= 1 \quad \mathsf{d}_{j-1} \mathrel{-}= 1$
$\mathsf{c}_{j-1 \bmod 2} \mathrel{-}= 1$

We implement the $\mathsf{x} \mathrel{-}= \mathsf{i}$ macro succinctly, eliminating the counter $\hat{\mathsf{i}}'$, as follows:

$\mathsf{x} \mathrel{-}= \mathsf{i}$**:**
loop
  $\mathsf{i} \mathrel{-}= 1 \quad \mathsf{i}' \mathrel{+}= 1 \quad \mathsf{x} \mathrel{-}= 1$
  $\mathsf{d}_{j-1} \mathrel{-}= 1$
loop
  $\hat{\mathsf{i}} \mathrel{-}= 1 \quad \mathsf{i} \mathrel{+}= 1$
  $\mathsf{d}_{j-1} \mathrel{-}= 1$
$\mathsf{c}_{j-1 \bmod 2} \mathrel{-}= 1$
loop
  $\mathsf{i} \mathrel{-}= 1 \quad \hat{\mathsf{i}} \mathrel{+}= 1$
  $\mathsf{d}_{j-1} \mathrel{-}= 1$
loop
  $\mathsf{i}' \mathrel{-}= 1 \quad \mathsf{i} \mathrel{+}= 1$
  $\mathsf{d}_{j-1} \mathrel{-}= 1$
$\mathsf{c}_{j-1 \bmod 2} \mathrel{-}= 1$

Analogously, we optimise the other macros $\mathsf{c}_0 \mathrel{-}= \mathsf{i}$, $\mathsf{c}_1 \mathrel{-}= \mathsf{i}$, $\mathsf{d}_j \mathrel{-}= \mathsf{i}$, $\mathsf{d}' \mathrel{+}= \mathsf{i}+1$ and $\mathsf{b}' \mathrel{+}= \mathsf{i}+1$ featuring in $\mathcal{H}_j^{(3)}$. Finally, we deliberately remove $\hat{\mathsf{i}}'$ from **setup** $\hat{\mathsf{i}}$ and **reset** i macros, respectively:

setup $\hat{\mathsf{i}}$**:** (note that the $\hat{\mathsf{i}}' \mathrel{+}= 1$ command is not present)
loop
  $\hat{\mathsf{i}} \mathrel{+}= 1 \quad \mathsf{b} \mathrel{-}= 1$
  $\mathsf{d}_{j-1} \mathrel{-}= 1$
$\mathsf{c}_{j-1 \bmod 2} \mathrel{-}= 1$
reset i**:** (note that the $\hat{\mathsf{i}}' \mathrel{-}= 1$ command is not present)
loop
  $\mathsf{i} \mathrel{-}= 1$
  $\mathsf{d}_{j-1} \mathrel{-}= 1$
$\mathsf{c}_{j-1 \bmod 2} \mathrel{-}= 1$

Algorithm V Program fragment $\mathcal{H}_j^{(3)}$.

```
1:  setup î
2:  i += 1   î −= 1
3:  b += 1   c_{j mod 2} += 1   d_j += 1   x += 1   y += 1   d_{j−1} += 1
4:  loop
5:      c_{j mod 2} += 1   d_j += 1   x += 1   y += 1   d_{j−1} += 1
6:  loop
7:      loop
8:          c_{j mod 2} −= i   c′ += 1
9:          loop at most b times
10:             d_j −= i   x −= i   d′ += i + 1
11:     loop
12:         b −= 1   b′ += i + 1
13:     loop
14:         b′ −= 1   b += 1
15:     loop
16:         c′ −= 1   c_{j mod 2} += 1
17:         loop at most b times
18:             d′ −= 1   d_j += 1   x += 1
19:     i += 1   î −= 1
20: ismax i
21: loop
22:     x −= i   y −= 1   d_{j−1} −= 1
23: reset i
```

Claim 12. The corresponding program $\mathcal{T}^{(3)}$

$\mathcal{A}'$
$\mathcal{H}_1^{(3)}$
...
$\mathcal{H}_{h+1}^{(3)}$
halt if $\mathsf{d}_0, \mathsf{d}_1, \ldots, \mathsf{d}_h = 0$

is a cleaning amplifier by $n!^{h+1}$ without tested counters satisfying the conditions of Lemma 7.

The proof of Lemma 7 is thus completed.

7 CONCLUDING REMARKS

We have focussed on presenting clearly the result that the Petri nets reachability problem is not elementary, leaving several arising directions for future consideration. The latter include investigating implications for the reachability problem for fixed-dimension flat vector addition systems with states (cf. [3, 13, 37**?**]).

ACKNOWLEDGEMENTS

We thank Alain Finkel for promoting the study of one-dimensional Petri nets with a pushdown stack, and Matthias Englert, Piotr Hofman and Radosław Piórkowski for working with us on open questions about those systems. The latter efforts led us to discovering the non-elementary lower bound presented in this paper.

We are also grateful to Marthe Bonamy, Artur Czumaj, Javier Esparza, Marcin Pilipczuk, Michał Pilipczuk, Sylvain Schmitz and Philippe Schnoebelen for helpful comments.

Finally, many thanks to the reviewers of the conference version and the referees of this article, through whose feedback the presentation has been significantly improved.

REFERENCES

[1] David Angeli, Patrick De Leenheer, and Eduardo D. Sontag. 2011. Persistence Results for Chemical Reaction Networks with Time-Dependent Kinetics and No Global Conservation Laws. *SIAM Journal of Applied Mathematics* 71, 1 (2011), 128–146. https://doi.org/10.1137/090779401

[2] Paolo Baldan, Nicoletta Cocco, Andrea Marin, and Marta Simeoni. 2010. Petri nets for modelling metabolic pathways: a survey. *Natural Computing* 9, 4 (2010), 955–989. https://doi.org/10.1007/s11047-010-9180-6

[3] Michael Blondin, Alain Finkel, Stefan Göller, Christoph Haase, and Pierre McKenzie. 2015. Reachability in Two-Dimensional Vector Addition Systems with States Is PSPACE-Complete. In *LICS*. IEEE Computer Society, 32–43. https://doi.org/10.1109/LICS.2015.14

[4] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2011. Two-variable logic on data words. *ACM Trans. Comput. Log.* 12, 4 (2011), 27:1–27:26. http://doi.acm.org/10.1145/1970398.1970403

[5] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2009. Two-variable logic on data trees and XML reasoning. *J. ACM* 56, 3 (2009), 13:1–13:48. http://doi.acm.org/10.1145/1516512.1516515

[6] Ahmed Bouajjani and Michael Emmi. 2013. Analysis of Recursively Parallel Programs. *ACM Trans. Program. Lang. Syst.* 35, 3 (2013), 10:1–10:49. http://doi.acm.org/10.1145/2518188

[7] Zakaria Bouziane. 1998. A Primitive Recursive Algorithm for the General Petri Net Reachability Problem. In *FOCS*. IEEE Computer Society, 130–136. https://doi.org/10.1109/SFCS.1998.743436

[8] Frank P. Burns, Albert Koelmans, and Alexandre Yakovlev. 2000. WCET Analysis of Superscalar Processors Using Simulation With Coloured Petri Nets. *Real-Time Systems* 18, 2/3 (2000), 275–288. https://doi.org/10.1023/A:1008101416758

[9] Thomas Colcombet and Amaldev Manuel. 2014. Generalized Data Automata and Fixpoint Logic. In *FSTTCS (LIPIcs)*, Vol. 29. Schloss Dagstuhl, 267–278. https://doi.org/10.4230/LIPIcs.FSTTCS.2014.267

[10] Stefano Crespi-Reghizzi and Dino Mandrioli. 1977. Petri Nets and Szilard Languages. *Information and Control* 33, 2 (1977), 177–192. https://doi.org/10.1016/S0019-9958(77)90558-7

[11] Normann Decker, Peter Habermehl, Martin Leucker, and Daniel Thoma. 2014. Ordered Navigation on Multi-attributed Data Words. In *CONCUR (LNCS)*, Vol. 8704. Springer, 497–511. https://doi.org/10.1007/978-3-662-44584-6_34

[12] Stéphane Demri, Diego Figueira, and M. Praveen. 2016. Reasoning about Data Repetitions with Counter Systems. *Logical Methods in Computer Science* 12, 3 (2016). https://doi.org/10.2168/LMCS-12(3:1)2016

[13] Matthias Englert, Ranko Lazić, and Patrick Totzke. 2016. Reachability in Two-Dimensional Unary Vector Addition Systems with States is NL-Complete. In *LICS*. ACM, 477–484. http://doi.acm.org/10.1145/2933575.2933577

[14] Javier Esparza. 1998. Decidability and Complexity of Petri Net Problems — An Introduction. In *Lectures on Petri Nets I (LNCS)*, Vol. 1491. Springer, 374–428. https://doi.org/10.1007/3-540-65306-6_20

[15] Javier Esparza, Pierre Ganty, Jérôme Leroux, and Rupak Majumdar. 2017. Verification of population protocols. *Acta Inf.* 54, 2 (2017), 191–215. https://doi.org/10.1007/s00236-016-0272-3

[16] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. 1968. Counter Machines and Counter Languages. *Mathematical Systems Theory* 2, 3 (1968), 265–283. https://doi.org/10.1007/BF01694011

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

[18] Steven M. German and A. Prasad Sistla. 1992. Reasoning about Systems with Many Processes. *J. ACM* 39, 3 (1992), 675–735. http://doi.acm.org/10.1145/146637.146681

[19] Sheila A. Greibach. 1978. Remarks on Blind and Partially Blind One-Way Multicounter Machines. *Theor. Comput. Sci.* 7 (1978), 311–324. https://doi.org/10.1016/0304-3975(78)90020-8

[20] Piotr Hofman and Sławomir Lasota. 2018. Linear Equations with Ordered Data. In *CONCUR (LIPIcs)*, Vol. 118. Schloss Dagstuhl, 24:1–24:17. https://doi.org/10.4230/LIPIcs.CONCUR.2018.24

[21] John E. Hopcroft and Jean-Jacques Pansiot. 1979. On the Reachability Problem for 5-Dimensional Vector Addition Systems. *Theor. Comput. Sci.* 8 (1979), 135–159. https://doi.org/10.1016/0304-3975(79)90041-0

[22] Petr Jancar. 2008. Bouziane's transformation of the Petri net reachability problem and incorrectness of the related algorithm. *Inf. Comput.* 206, 11 (2008), 1259–1263. https://doi.org/10.1016/j.ic.2008.06.003

[23] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2014. A Widening Approach to Multithreaded Program Verification. *ACM Trans. Program. Lang. Syst.* 36, 4 (2014), 14:1–14:29. http://doi.acm.org/10.1145/2629608

[24] Max I. Kanovich. 1995. Petri Nets, Horn Programs, Linear Logic and Vector Games. *Ann. Pure Appl. Logic* 75, 1–2 (1995), 107–135. https://doi.org/10.1016/0168-0072(94)00060-G

[25] Richard M. Karp and Raymond E. Miller. 1969. Parallel Program Schemata. *J. Comput. Syst. Sci.* 3, 2 (1969), 147–195. https://doi.org/10.1016/S0022-0000(69)80011-5

[26] S. Rao Kosaraju. 1982. Decidability of Reachability in Vector Addition Systems (Preliminary Version). In *STOC*. ACM, 267–281. http://doi.acm.org/10.1145/800070.802201

[27] Jean-Luc Lambert. 1992. A Structure to Decide Reachability in Petri Nets. *Theor. Comput. Sci.* 99, 1 (1992), 79–104. https://doi.org/10.1016/0304-3975(92)90173-D

[28] Ranko Lazić and Sylvain Schmitz. 2015. Nonelementary Complexities for Branching VASS, MELL, and Extensions. *ACM Trans. Comput. Log.* 16, 3 (2015), 20:1–20:30. http://doi.acm.org/10.1145/2733375

[29] Ranko Lazić and Patrick Totzke. 2017. What Makes Petri Nets Harder to Verify: Stack or Data?. In *Concurrency, Security, and Puzzles — Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday (LNCS)*, Vol. 10160. Springer, 144–161. https://doi.org/10.1007/978-3-319-51046-0_8

[30] Hélène Leroux, David Andreu, and Karen Godary-Dejean. 2015. Handling Exceptions in Petri Net-Based Digital Architecture: From Formalism to Implementation on FPGAs. *IEEE Trans. Industrial Informatics* 11, 4 (2015), 897–906. https://doi.org/10.1109/TII.2015.2435696

[31] Jérôme Leroux. 2010. The General Vector Addition System Reachability Problem by Presburger Inductive Invariants. *Logical Methods in Computer Science* 6, 3 (2010). https://doi.org/10.2168/LMCS-6(3:22)2010

[32] Jérôme Leroux. 2011. Vector addition system reachability problem: a short self-contained proof. In *POPL*. ACM, 307–316. http://doi.acm.org/10.1145/1926385.1926421

[33] Jérôme Leroux. 2012. Vector Addition Systems Reachability Problem (A Simpler Solution). In *Turing-100 (EPiC Series in Computing)*, Vol. 10. EasyChair, 214–228. http://www.easychair.org/publications/paper/106497

[34] Jérôme Leroux and Sylvain Schmitz. 2015. Demystifying Reachability in Vector Addition Systems. In *LICS*. IEEE Computer Society, 56–67. https://doi.org/10.1109/LICS.2015.16

[35] Jérôme Leroux and Sylvain Schmitz. 2019. Reachability in Vector Addition Systems is Primitive-Recursive in Fixed Dimension. In *LICS*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785796

[36] Jérôme Leroux and Philippe Schnoebelen. 2014. On Functions Weakly Computable by Petri Nets and Vector Addition Systems. In *RP (LNCS)*, Vol. 8762. Springer, 190–202. https://doi.org/10.1007/978-3-319-11439-2_15

[37] Jérôme Leroux and Grégoire Sutre. 2004. On Flatness for 2-Dimensional Vector Addition Systems with States. In *CONCUR (LNCS)*, Vol. 3170. Springer, 402–416. https://doi.org/10.1007/978-3-540-28644-8_26

[38] Yuliang Li, Alin Deutsch, and Victor Vianu. 2017. VERIFAS: A Practical Verifier for Artifact Systems. *PVLDB* 11, 3 (2017), 283–296. http://www.vldb.org/pvldb/vol11/p283-li.pdf

[39] Richard J. Lipton. 1976. *The reachability problem requires exponential space.* Technical Report 62. Yale University. http://cpsc.yale.edu/sites/default/files/files/tr63.pdf

[40] Martin H. Löb and Stanley S. Wainer. 1970. Hierarchies of number-theoretic functions. I. *Archiv für mathematische Logik und Grundlagenforschung* 13, 1–2 (1970), 39–51. https://doi.org/10.1007/BF01967649

[41] Ernst W. Mayr. 1981. An Algorithm for the General Petri Net Reachability Problem. In *STOC*. ACM, 238–246. http://doi.acm.org/10.1145/800076.802477

[42] Ernst W. Mayr. 1984. An Algorithm for the General Petri Net Reachability Problem. *SIAM J. Comput.* 13, 3 (1984), 441–460. https://doi.org/10.1137/0213029

[43] Ernst W. Mayr and Albert R. Meyer. 1981. The Complexity of the Finite Containment Problem for Petri Nets. *J. ACM* 28, 3 (1981), 561–576. http://doi.acm.org/10.1145/322261.322271

[44] Roland Meyer. 2009. A theory of structural stationarity in the *pi*-Calculus. *Acta Inf.* 46, 2 (2009), 87–137. https://doi.org/10.1007/s00236-009-0091-x

[45] Marvin L. Minsky. 1967. *Computation: finite and infinite machines.* Prentice-Hall, Inc. https://dl.acm.org/citation.cfm?id=1095587

[46] Mor Peleg, Daniel L. Rubin, and Russ B. Altman. 2005. Research Paper: Using Petri Net Tools to Study Properties and Dynamics of Biological Systems. *JAMIA* 12, 2 (2005), 181–199. https://doi.org/10.1197/jamia.M1637

[47] Carl Adam Petri. 1962. *Kommunikation mit Automaten.* Ph.D. Dissertation. Universität Hamburg. http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/

[48] Charles Rackoff. 1978. The Covering and Boundedness Problems for Vector Addition Systems. *Theor. Comput. Sci.* 6 (1978), 223–231. https://doi.org/10.1016/0304-3975(78)90036-1

[49] George S. Sacerdote and Richard L. Tenney. 1977. The Decidability of the Reachability Problem for Vector Addition Systems (Preliminary Version). In *STOC*. ACM, 61–76. http://doi.acm.org/10.1145/800105.803396

[50] Sylvain Schmitz. 2016. Complexity Hierarchies beyond Elementary. *TOCT* 8, 1 (2016), 3:1–3:36. http://doi.acm.org/10.1145/2858784

[51] Sylvain Schmitz. 2016. The complexity of reachability in vector addition systems. *SIGLOG News* 3, 1 (2016), 4–21. http://doi.acm.org/10.1145/2893582.2893585

[52] Wil M. P. van der Aalst. 2015. Business process management as the "Killer App" for Petri nets. *Software and System Modeling* 14, 2 (2015), 685–691. https://doi.org/10.1007/s10270-014-0424-2