The reachability problem for branching vector addition systems requires doubly-exponential space

Ranko Lazić

DIMAP, Department of Computer Science, University of Warwick, UK

Abstract

Branching vector addition systems are an extension of vector addition systems where new reachable vectors may be obtained by summing two reachable vectors and adding an integral vector from a fixed finite set. The reachability problem for them is shown hard for doubly-exponential space. For an alternative extension of vector addition systems, where reachable vectors may be combined by subtraction, most decision problems of interest are shown undecidable.

Key words: program correctness, theory of computation

1. Introduction

Vector addition systems (shortly, VAS), or equivalently Petri nets (e.g., [1]), are a fundamental model of computation, which is more expressive than finite-state machines but not Turing-powerful. Decidability and complexity of a variety of problems have been extensively studied (Esparza and Nielsen's [2] is a comprehensive survey).

A k-dimensional VAS consists of an initial vector of non-negative integers, and a finite set of vectors of integers, all of dimension k. Let us call the initial vector *axiom*, and the other vectors *rules*. A computation can then be thought of as a *derivation*: it starts with the axiom, and at each step, the next vector is derived from the current one by adding a rule. The vectors of interest are the ones derived *admissibly*, i.e. at the end of a derivation which is such that none of the vectors derived during it contains a negative entry.

One of the most famous theoretical questions in program correctness is the complexity of the reachability problem for VAS. The latter decision problem asks, given a VAS $\mathcal{V}$ and a vector $\mathbf{v}$, whether $\mathcal{V}$ can reach (i.e., derive admissibly) $\mathbf{v}$. Hardness for exponential space was established by Lipton's simulation of a hierarchy of bounded counters [3], where the bound for each level is the square of the previous one. The simulation partly resembles Stockmeyer's landmark construction to show that first-order logic on linear orders with a unary predicate

is not elementary [4], in which the gaps are exponential rather than quadratic. Decidability of VAS reachability then had to wait for the ingenious proof of Mayr [5], subsequently simplified by Kosaraju [6]. Building on Lambert's presentation [7], Leroux's recent breakthrough [8] provides a trivial algorithm by proving that, in case of unreachability, $\mathcal{V}$ has an invariant which is definable in Presburger arithmetic and does not contain $\mathbf{v}$. However, no known algorithm is even primitive recursive, so a huge gap to Lipton's lower bound remains open.

The following is a natural extension of VAS: instead of linearly, computation proceeds from the leaves to the root of a tree. For each non-leaf node, its vector is derived by summing the vectors derived at its children (if there is more than one child) and adding a rule vector. The same condition of admissibility applies, i.e. no derived vector may contain a negative entry. This model of computation is called branching VAS (shortly, BVAS).

In recent years, it has turned out that BVAS have interesting connections to a number of formalisms:

- BVAS correspond to a class of linear index grammars in computational linguistics [9];
- reachability for BVAS is decidable iff provability in multiplicative exponential linear logic is decidable [10];
- Verma and Goubault-Larrecq have extended the computation of Karp and Miller trees [11] to BVAS, and used it to draw conclusions about a

class of equational tree automata which are useful for analysing cryptographic protocols [12];

- if first-order logic with 2 variables on finite data trees (which has applications to the XPath query language for XML) is decidable, then so is reachability for BVAS [13].

Our main result is that reachability for BVAS is 2ExpSpace-hard. The key part of the proof is showing that, by a careful combination of Lipton's simulation and branching, BVAS can compute triply-exponentially large numbers. We then show how that capability can be used to simulate increments, decrements and zero tests of triply-exponentially bounded counters. For zero tests, guesses are involved, whose accuracy is checked at the end using the fact that the reachability problem asks whether *exactly* the vector given is derivable admissibly. Hence, it does not follow that problems such as covering and boundedness for BVAS are also 2ExpSpace-hard. In fact, the latter two problems have recently been proved 2ExpTime-complete [14], where the lower bounds were obtained essentially by extending Lipton's simulation with alternation, which still involves only doubly-exponentially large numbers.

As is the case with Lipton's result, we in fact establish 2ExpSpace-hardness of reachability for BVAS whose axioms and rules contain only entries -1, 0 or 1. Inspecting the translations to multiplicative exponential linear logic [10] and first-order logic with 2 variables on finite data trees [13] reveals that they use only logarithmic space for such BVAS, so 2ExpSpace-hardness for the respective provability and satisfiability problems follows.

Verma and Goubault-Larrecq have proposed a variant of BVAS in which binary rules subtract rather than add vectors, and asked about their algorithmic properties [12]. We provide a negative answer, showing that already problems such as covering are undecidable.

2. Preliminaries

We write $\mathbb{N}$ and $\mathbb{Z}$ for the sets of all non-negative and arbitrary integers, respectively.

BVAS. For the systems we consider, we adopt the stateless formalisation of Demri et al. [14], which is equivalent to the branching vector addition systems with states [12] and the vector addition tree automata [10, 13]. A simple way to encode states is to increase the dimension by the number of states and use unit vectors; further explanations can be found in the former paper [14].

A *branching vector addition system* ($BVAS$) is a tuple $\mathcal{B} = \langle k, A_0, R_1, R_2 \rangle$, where:

- $k \in \mathbb{N}$ is the dimension;
- $A_0 \subseteq \mathbb{N}^k$ is a non-empty finite set of axioms;
- $R_1, R_2 \subseteq \mathbb{Z}^k$ are finite sets of unary and binary rules, respectively.

We say that $\mathbf{v} \in \mathbb{N}^k$ is *reachable* by $\mathcal{B}$ iff there exists a labelled finite binary tree $\mathcal{D}$ such that:

- each leaf is labelled by an axiom, each node with one child by a unary rule, and each node with two children by a binary rule;
- for each node ν, the vector $\widehat{\mathcal{D}}(\nu)$ derived at it, which is the sum of all labels in the subtree rooted at ν, is in $\mathbb{N}^k$;
- $\mathbf{v} = \widehat{\mathcal{D}}(\varepsilon)$, the vector derived at the root.

We call a BVAS *normal* iff its axioms and rules contain only entries -1, 0 or 1.

Programs with counters. As in Esparza's nice presentation of Lipton's proof [15, §7], it will be convenient to work with programs which operate on non-negative counters. The following are three kinds of such programs.

A *counter program* is a finite sequence of commands which may be labelled. A command is one of: an increment of a counter ($x := x + 1$), a decrement of a counter ($x := x - 1$), a jump to a labelled command (goto L), a zero test (if $x = 0$ then L else L'), or termination (halt). Initially, all counters have value 0. Whenever a decrement of a counter with value 0 is attempted, the program aborts. In every program, halt occurs only as the last command.

Net programs are defined like counter programs, except that we are not allowed zero tests, but we can use calls of subroutines (gosub L), returns from subroutines (return), and non-deterministic jumps (goto L or L'). The stack involved is bounded, since we require that each subroutine can be assigned a level so that subroutines of level i can only call subroutines of level $i + 1$. That includes the main program, whose level is 0. Only jumps to commands in the same subroutine are permitted, and in every subroutine which is not the main program, return occurs only as the last command.

A *branching net program* can have one or more subroutines at the same level as the main program

(i.e. 0). Such a subroutine L can be used only by the command `gojoin` L at level 0. The effect is to launch a new copy of subroutine L, which starts with all counters having value 0, and if it returns, to add the finishing value of each counter to its value at the point of the launch.

Decision problems. We consider the following ones:

BVAS reachability Given a BVAS $\mathcal{B}$ and a non-negative vector $\mathbf{v}$ of the same dimension, can $\mathcal{B}$ reach $\mathbf{v}$?

Halting Given a program $\mathcal{P}$, does it have a halting computation?

$f(n)$**-bounded halting** Given a program $\mathcal{P}$ with n commands, does it have a halting computation with all counter values at most $f(n)$ (throughout)?

Halting-with-zero Given a program $\mathcal{P}$ and a counter x, does it have a halting computation in which the final value of x is zero?

By basic constructions of computability theory (encoding a tape by two stacks, simulating a stack by two counters, and padding the input program), we have:

Proposition 1. *The $2^{2^{2^n}}$-bounded halting problem for counter programs is* 2ExpSpace*-hard.*

3. Hardness for doubly-exponential space

The main part of Lipton's proof is showing that, given a counter program $\mathcal{C}$ with n commands, a net program $\mathcal{N}(\mathcal{C})$ with $O(n)$ commands is computable in time polynomial in n such that $\mathcal{C}$ has a 2^{2^n}-bounded halting computation iff $\mathcal{N}(\mathcal{C})$ has a halting computation. (Recall that counter programs are deterministic, but net programs may be non-deterministic.) In particular, since 2^{2^n}-bounded halting for counter programs is ExpSpace-hard, we have the same lower bound for the halting problem for net programs.

After sketching the construction of $\mathcal{N}(\mathcal{C})$ (further details can be found in Esparza's article [15, §7]), we shall show how its components can be used to obtain a reduction from the $2^{2^{2^n}}$-bounded halting problem for counter programs to the halting-with-zero problem for branching net programs. It will then remain to reduce the latter problem to the reachability problem for normal BVAS.

From counter programs to net programs. Let x_1, $\ldots$, x_l be the counters of $\mathcal{C}$. Then $\mathcal{N}(\mathcal{C})$ has counters x_j and $\bar{x}_j$ for each $1 \leq j \leq l$, s_i and $\bar{s}_i$ for each $0 \leq i \leq n$, and y_i, $\bar{y}_i$, z_i and $\bar{z}_i$ for each $0 \leq i < n$, where n is the number of commands in $\mathcal{C}$. At the beginning, $\mathcal{N}(\mathcal{C})$ performs a subroutine $Init_n(x_1, \ldots, x_l)$ which ensures:

(1) for all $1 \leq j \leq l$, $x_j = 0$ and $\bar{x}_j = 2^{2^n}$;

(2) for all $0 \leq i \leq n$, $s_i = 0$ and $\bar{s}_i = 2^{2^i}$;

(3) for all $0 \leq i < n$, $y_i = 2^{2^i} = z_i$ and $\bar{y}_i = 0 = \bar{z}_i$.

The rest of the main program in $\mathcal{N}(\mathcal{C})$ is obtained by translating each command in $\mathcal{C}$. The translations are such that the sum of each pair of complementary counters is maintained: $x_j + \bar{x}_j = 2^{2^n}$ for all $1 \leq j \leq l$, $s_i + \bar{s}_i = 2^{2^i}$ for all $0 \leq i \leq n$, and $y_i + \bar{y}_i = 2^{2^i} = z_i + \bar{z}_i$ for all $0 \leq i < n$. Thus, each increment of x_j in $\mathcal{C}$ is translated to $x_j := x_j + 1$; $\bar{x}_j := \bar{x}_j - 1$ in $\mathcal{N}(\mathcal{C})$, and correspondingly for decrements.

The key component in translating zero tests is a subroutine Dec_n which performs $s_n := s_n - 1$; $\bar{s}_n := \bar{s}_n + 1$ exactly 2^{2^n} times. Given counters c and $\bar{c}$, and labels L and L', let $Test_n(c, \bar{c}, L, L')$ stand for the code below. It starts with a guess of whether c is zero or not, where checking the latter choice is easy. To check that c is zero, the code transfers a non-deterministic part of $\bar{c}$ to s_n (while maintaining $c + \bar{c}$ and $s_n + \bar{s}_n$), and then calls Dec_n. Assuming that $c + \bar{c} = 2^{2^n}$ and $s_n = 0$ (and hence $\bar{s}_n = 2^{2^n}$) at the start, that check succeeds (i.e. does not abort) indeed only for $c = 0$, in which case it finishes with $c = 2^{2^n}$ and $s_n = 0$.

```
          goto zero or nonzero;
nonzero:  c := c − 1; c := c + 1; goto L′;
   zero:  c := c + 1; c̄ := c̄ − 1;
          sₙ := sₙ + 1; s̄ₙ := s̄ₙ − 1;
          goto exit or zero;
   exit:  gosub Decₙ; goto L
```

Each `if` $x_j = 0$ `then` L `else` L' is then translated to the following code. If $x_j = 0$, the first instance of $Test_n$ will finish with $x_j = 2^{2^n}$, and the second instance will undo that side effect and jump to L. Otherwise, already the former will jump to L'.

$$Test_n(x_j, \bar{x}_j, continue, L');$$
$$continue: \quad Test_n(\bar{x}_j, x_j, L, L')$$

To implement Dec_n, we also implement Dec_i for each $0 \leq i < n$, whose specification is to perform $s_i := s_i - 1$; $\bar{s}_i := \bar{s}_i + 1$ exactly 2^{2^i} times (and

abort if that is not possible). The code for Dec_0 is trivial. For Dec_{i+1}, the auxiliary counters y_i, $\bar{y}_i$, z_i and $\bar{z}_i$ are used to provide two nested loops that count from 2^{2^i} to 0 each, and so iterate $2^{2^i} \cdot 2^{2^i} = 2^{2^{i+1}}$ times together. Whether a loop counter has reached zero is checked by the $Test_i$ code defined above, which may call Dec_i. Its side effect, that the counter is complemented if it is zero, ensures that Dec_{i+1} finishes with $y_i = 2^{2^i} = z_i$ and $\bar{y}_i = 0 = \bar{z}_i$, which are also assumed at the start.

$$
\begin{aligned}
Dec_{i+1} : \quad & y_i := y_i - 1;\ \ \bar{y}_i := \bar{y}_i + 1; \\
Dec'_{i+1} : \quad & z_i := z_i - 1;\ \ \bar{z}_i := \bar{z}_i + 1; \\
& s_{i+1} := s_{i+1} - 1;\ \ \bar{s}_{i+1} := \bar{s}_{i+1} + 1; \\
& Test_i(z_i, \bar{z}_i, exit'_{i+1}, Dec'_{i+1}); \\
exit'_{i+1} : \quad & Test_i(y_i, \bar{y}_i, exit_{i+1}, Dec_{i+1}); \\
exit_{i+1} : \quad & \texttt{return}
\end{aligned}
$$

Observe that, for each command in $\mathcal{C}$, if its translation in $\mathcal{N}(\mathcal{C})$ does not abort, then it maintains properties (2) and (3) of the auxiliary counters.

From counter programs to branching net programs. From the building blocks of $\mathcal{N}(\mathcal{C})$, it is straightforward to implement the following subroutines, which assume (2) and (3). Branching is not required.

- $Move_n(h, \bar{h}, h', \bar{h}')$, which also assumes $h + \bar{h} = 2^{2^n}$ and $h' = 0 = \bar{h}'$, and transfers $h, \bar{h}$ to $h', \bar{h}'$;
- $Zero_n(h', \bar{h}')$, which also assumes $h' + \bar{h}' = 2^{2^n}$, and empties (i.e. sets to zero) h' and $\bar{h}'$;
- Fin_n, which empties all counters s_i and $\bar{s}_i$ ($0 \le i \le n$), and $y_i, \bar{y}_i, z_i$ and $\bar{z}_i$ ($0 \le i < n$).

Similarly, let $\texttt{if}_n\ h = h'\ \texttt{then}\ L\ \texttt{else}\ L'$ denote the code below. Assuming (2), (3) and $h + \bar{h} = 2^{2^n} = h' + \bar{h}'$, it tests equality of h and h'. It uses auxiliary counters h'' and $\bar{h}''$, which it initialises to $h'' = 0$ and $\bar{h}'' = 2^{2^n}$. It then transfers h and h' simultaneously to h'' until both are zero (they were equal) or only one is zero (they were unequal). Zero tests are implemented as in $\mathcal{N}(\mathcal{C})$. The subscript n emphasises that h and h' are "level n" counters.

$$
\begin{aligned}
init : \quad & s_n := s_n + 1;\ \ \bar{s}_n := \bar{s}_n - 1; \\
& \bar{h}'' := \bar{h}'' + 1;\ \texttt{goto}\ exit\ \texttt{or}\ init; \\
exit : \quad & \texttt{gosub}\ Dec_n; \\
transfer : \quad & \texttt{if}_n\ h = 0\ \texttt{then}\ zero\ \texttt{else}\ nonzero; \\
zero : \quad & \texttt{if}_n\ h' = 0\ \texttt{then}\ eq\ \texttt{else}\ uneq; \\
nonzero : \quad & \texttt{if}_n\ h' = 0\ \texttt{then}\ uneq\ \texttt{else}\ inc; \\
inc : \quad & h := h - 1;\ \ \bar{h} := \bar{h} + 1; \\
& h' := h' - 1;\ \ \bar{h}' := \bar{h}' + 1; \\
& h'' := h'' + 1;\ \ \bar{h}'' := \bar{h}'' - 1; \\
& \texttt{goto}\ transfer; \\
eq : \quad & \texttt{gosub}\ Restore;\ \texttt{goto}\ L; \\
uneq : \quad & \texttt{gosub}\ Restore;\ \texttt{goto}\ L'
\end{aligned}
$$

The subroutine restores h and h' by transferring back from h'', and then also empties $\bar{h}''$:

$$
\begin{aligned}
Restore : \quad & \texttt{if}_n\ h'' = 0\ \texttt{then}\ empty\ \texttt{else}\ dec; \\
dec : \quad & h := h + 1;\ \ \bar{h} := \bar{h} - 1; \\
& h' := h' + 1;\ \ \bar{h}' := \bar{h}' - 1; \\
& h'' := h'' - 1;\ \ \bar{h}'' := \bar{h}'' + 1; \\
& \texttt{goto}\ Restore; \\
empty : \quad & s_n := s_n + 1;\ \ \bar{s}_n := \bar{s}_n - 1; \\
& \bar{h}'' := \bar{h}'' - 1;\ \texttt{goto}\ done\ \texttt{or}\ empty; \\
done : \quad & \texttt{gosub}\ Dec_n;\ \texttt{return}
\end{aligned}
$$

We are now in a position to program our key subroutine $Tree_n(c)$, which involves branching. It is of the level of the main program, and its returned computations encode complete binary trees. In such a computation, the final value of h is the height of the tree, and the final value of c is its number of non-leaf nodes, so they satisfy $c = 2^h - 1$. The subroutine also uses counters s_i, $\bar{s}_i$ ($0 \le i \le n$), $y_i, \bar{y}_i, z_i, \bar{z}_i$ ($0 \le i < n$), $\bar{h}, h', \bar{h}', h''$ and $\bar{h}''$, whose final values satisfy properties (2) and (3), $\bar{h} = 2^{2^n} - h$, and $h' = \bar{h}' = h'' = \bar{h}'' = 0$.

The initialisation ensures $h = 0$, $\bar{h} = 2^{2^n}$, (2) and (3). The remaining counters are initially zero by default. The non-deterministic choice is whether to increase the height by one or return. In the former case, before joining another returned computation of $Tree_n(c)$, h and $\bar{h}$ are moved to h' and $\bar{h}'$, and all other auxiliary counters are emptied (h'' and $\bar{h}''$ are already zero). Hence, after the join (which adds all counters pointwise), h contains the returned computation's final value, c contains the sum of its old value and the returned computation's final value, (2) and (3) are satisfied, and h'' and $\bar{h}''$ are zero. To ensure that the binary tree is complete, h and h' are tested for equality. The height is then increased by 1, h' and $\bar{h}'$ are emptied, the number of non-leaf nodes is also increased by 1, and the non-deterministic choice is repeated.

$$
\begin{aligned}
Tree_n(c) : \quad & \texttt{gosub}\ Init_n(h); \\
loop : \quad & \texttt{goto}\ next\ \texttt{or}\ end; \\
next : \quad & Move_n(h, \bar{h}, h', \bar{h}');\ \ Fin_n; \\
& \texttt{gojoin}\ Tree_n(c); \\
& \texttt{if}_n\ h = h'\ \texttt{then}\ next'\ \texttt{else}\ dead; \\
next' : \quad & h := h + 1;\ \ \bar{h} := \bar{h} - 1; \\
& Zero_n(h', \bar{h}');\ \ c := c + 1; \\
& \texttt{goto}\ loop; \\
dead : \quad & \texttt{goto}\ dead; \\
end : \quad & \texttt{return}
\end{aligned}
$$

Proposition 2. *The final valuations of $Tree_n(c)$ are all those where $h + \bar{h} = 2^{2^n}$, $c = 2^h - 1$, (2) and (3) hold, and all other counters are zero.*

To reduce from the $2^{2^{2^n}}$-bounded halting problem for counter programs, suppose $\mathcal{C}$ is a counter program with n commands and counters x_1, ..., x_l. We describe how to compute a branching net program $\mathcal{M}(\mathcal{C})$ whose counters are those of the net program $\mathcal{N}(\mathcal{C})$ plus h, $\bar{h}$, h', $\bar{h}'$, h'', $\bar{h}''$, and g.

For each $1 \le j \le l$, $\mathcal{M}(\mathcal{C})$ initialises $\bar{x}_j$ to $2^{2^{2^n}}$ by launching $Tree_n(\bar{x}_j)$, then checking that $\bar{h} = 0$ (i.e. $h = 2^{2^n}$), emptying h, $\bar{h}$ and the other auxiliary counters, and incrementing $\bar{x}_j$:

> $\quad$ `gojoin` $Tree_n(\bar{x}_j)$;
> $\quad$ $\text{if}_n\ \bar{h} = 0\ \text{then}\ next_j\ \text{else}\ dead'$;
> $next_j:\quad Zero_n(h, \bar{h});\ Fin_n;\ \bar{x}_j := \bar{x}_j + 1$;
> $\quad\quad\ \dots$
> $dead':\quad$ `goto` $dead'$;

As before, increments and decrements in $\mathcal{C}$ are performed in $\mathcal{M}(\mathcal{C})$ together with the opposite operation on the complementary counter. To check that x_j is zero, $\mathcal{M}(\mathcal{C})$ launches $Tree_n(g)$, then checks that $\bar{h} = 0$, empties h, $\bar{h}$ and the other auxiliary counters, increments g, performs

$$g := g - 1;\ x_j := x_j + 1;\ \bar{x}_j := \bar{x}_j - 1$$

a non-deterministic number of times, and repeats all that with x_j and $\bar{x}_j$ swapped. If the resulting value of g is zero, since $x_j + \bar{x}_j = 2^{2^{2^n}}$ is maintained and each of the two launches of $Tree_n(g)$ increased g by exactly $2^{2^{2^n}} - 1$, we have that x_j was indeed zero, and also that g was zero. By repeating that argument, each preceding check that a counter of $\mathcal{C}$ is zero was also accurate. To conclude the following, it remains to observe that, although the initialisation of each $\bar{x}_j$ uses a separate subroutine $Tree_n(\bar{x}_j)$, their numbers of commands are constant.

Lemma 1. *Given a counter program $\mathcal{C}$ with n commands, a branching net program $\mathcal{M}(\mathcal{C})$ with $O(n)$ commands is computable in time polynomial in n such that $\mathcal{C}$ has a $2^{2^{2^n}}$-bounded halting computation iff $\mathcal{M}(\mathcal{C})$ has a halting computation in which the final value of g is zero.*

From branching net programs to normal BVAS. A simple translation from net programs with n commands to VAS of size $O(n^2)$ was described by Esparza [15, §7]. It operates in space logarithmic in n, and produces normal VAS which have a separate place (i.e. vector component) for each counter in the program and for each command in the program. The translation is easy to extend to branching net programs and BVAS: each `gojoin` L command produces a binary rule whose -1 entries ensure that the two derived vectors being summed correspond to the point where the command occurs and the point where L returns. By adding to the BVAS rules that can empty each counter except g provided that control has reached the `halt` command, and by ensuring that that command corresponds to the first vector component, we obtain:

Lemma 2. *Given a branching net program $\mathcal{M}$ with n commands and a counter g, a normal BVAS $\mathcal{B}(\mathcal{M})$ of size $O(n^2)$ is computable in space logarithmic in n such that $\mathcal{M}$ has a halting computation in which the final value of g is zero iff $\mathcal{B}(\mathcal{M})$ can reach the unit vector $\mathbf{e}_1 = \langle 1, 0, \dots, 0 \rangle$.*

By Proposition 1 and Lemmata 1 and 2, we infer:

Theorem 1. *The reachability problem for normal BVAS is* 2ExpSpace-*hard.*

4. Undecidability

The counterpart of the variant of BVAS in which binary rules join derived vectors by subtraction rather than addition [12] are net programs with `gojoin`$^-$ L commands that subtract the current value of each counter from its returning value from the subroutine L, aborting if negative values are produced. We are going to argue that already the halting problem for the latter programs is undecidable, and so also is covering and other problems for the BVAS variant to which the halting problem is reducible. (The covering problem is to decide, given a BVAS $\mathcal{B}$ and a non-negative vector $\mathbf{v}$ of the same dimension, whether $\mathcal{B}$ can reach a vector which is component-wise greater than or equal to $\mathbf{v}$.)

It suffices to show that the net programs with `gojoin`$^-$ L commands can simulate increments, decrements and zero tests of two integer-valued variables x_1 and x_2. Such a variable x_j is represented as the difference between counters x_j and $\hat{x}_j$. Its increments are performed as increments of x_j, and its decrements as increments of $\hat{x}_j$. In simulating zero tests, we use an auxiliary third variable x', which is represented by two counters x' and $\hat{x}'$ in the same way.

To check that x_j is zero, the program decrements x_j and $\hat{x}_j$ a non-deterministic number of times, and then performs `gojoin`$^-$ L_j twice:

$$\begin{aligned} loop_j: \quad & x_j := x_j - 1; \ \hat{x}_j := \hat{x}_j - 1; \\ & \texttt{goto } dec_j \texttt{ or } check_j; \\ check_j: \quad & \texttt{gojoin}^- \ L_j; \ \texttt{gojoin}^- \ L_j \end{aligned}$$

where L_j is a subroutine that just increments the counters for the other two variables a non-deterministic number of times:

$$\begin{aligned} L_j: \quad & x_{3-j} := x_{3-j} + 1; \ \hat{x}_{3-j} := \hat{x}_{3-j} + 1; \\ & x' := x' + 1; \ \hat{x}' := \hat{x}' + 1; \\ & \texttt{goto } L_j \texttt{ or } return_j; \\ return_j: \quad & \texttt{return} \end{aligned}$$

If the check succeeds, since the returning values of x_j and $\hat{x}_j$ from L_j are zero by default, their values before the two launches of L_j must have been also zero, so they must have been equal at the start of the check. Conversely, if $x_j = \hat{x}_j$ (i.e. variable x_j is zero), the check can succeed by reducing x_j and $\hat{x}_j$ to zero, and by ensuring that the returning values of x_{3-j}, $\hat{x}_{3-j}$, x' and $\hat{x}'$ are sufficiently large to avoid abortion. The purpose of the second launch of L_j is to undo the side effect that $x_{3-j} - \hat{x}_{3-j}$ and $x' - \hat{x}'$ (i.e. the values of the other two variables) are negated by the first launch.

To check that x_j is non-zero, the program non-deterministically:

- either checks that x_j is greater than zero, by decrementing x_j and incrementing x' one or more times, then checking that x_j is zero (as above), and finally repeating with x_j and x' swapped in order to restore x_j and reset x' to zero;
- or checks that x_j is smaller than zero, by incrementing x_j and decrementing x' one or more times, then checking that x_j is zero (as above), and finally repeating with x_j and x' swapped in order to restore x_j and reset x' to zero.

5. Concluding remarks

For VAS reachability, the highest known lower bound is Lipton's EXPSPACE-hardness [3], the same as for the EXPSPACE-complete problems of covering and boundedness for VAS [16]. Although BVAS reachability has so far resisted attempts to prove it decidable, the 2EXPSPACE-hardness shown in this letter, together with the 2EXPTIME-completeness of covering and boundedness for BVAS [14], indicate that BVAS are not a trivial extension of VAS.

References

[1] W. Reisig, Petri Nets: An Introduction, Vol. 4 of Monographs in Theor. Comput. Sci. An EATCS Series, Springer, 1985.

[2] J. Esparza, M. Nielsen, Decidability issues for Petri nets — a survey, Bull. EATCS 52 (1994) 244–262.

[3] R. J. Lipton, The reachability problem requires exponential space, Tech. Rep. 62, Dep. Comput. Sci., Yale Univ. (Jan. 1976).

[4] L. J. Stockmeyer, The complexity of decision problems in automata theory and logic, Ph.D. thesis, MIT, TR-133, Lab. Comput. Sci. (1974).

[5] E. W. Mayr, An algorithm for the general Petri net reachability problem, SIAM J. Comput. 13 (3) (1984) 441–460.

[6] R. Kosaraju, Decidability of reachability in vector addition systems, in: STOC, 1982, pp. 267–281.

[7] J.-L. Lambert, A structure to decide reachability in Petri nets, Theoretical Comput. Sci. 99 (1) (1992) 79–104.

[8] J. Leroux, The general vector addition system reachability problem by Presburger inductive invariants, in: LICS, IEEE Comput. Soc., 2009, pp. 4–13.

[9] O. Rambow, Multiset-valued linear index grammars: imposing dominance constraints on derivations, in: ACL, Morgan Kaufmann, 1994, pp. 263–270.

[10] P. de Groote, B. Guillaume, S. Salvati, Vector addition tree automata, in: LICS, IEEE Comput. Soc., 2004, pp. 64–73.

[11] R. M. Karp, R. E. Miller, Parallel program schemata, J. Comput. Syst. Sci. 3 (2) (1969) 147–195.

[12] K. N. Verma, J. Goubault-Larrecq, Karp-Miller trees for a branching extension of VASS, Discr. Math. and Theor. Comput. Sci. 7 (2005) 217–230.

[13] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, Two-variable logic on data trees and XML reasoning, J. ACM 56 (3).

[14] S. Demri, M. Jurdziński, O. Lachish, R. Lazić, The covering and boundedness problems for branching vector addition systems, in: FSTTCS, Vol. 4 of LIPIcs, Schloss Dagstuhl, 2009, pp. 181–192.

[15] J. Esparza, Decidability and complexity of Petri net problems — an introduction, in: Lectures on Petri Nets I: Basic Models, Vol. 1491 of Lect. Notes Comput. Sci., Springer, 1998, pp. 374–428.

[16] C. Rackoff, The covering and boundedness problems for vector addition systems, Theoretical Comput. Sci. 6 (2) (1978) 223–231.