# How Big is the Automaton? Certified Lower Bounds on the Size of Presburger DFAs

Nicolas Amat
DTIS, ONERA, Université de Toulouse, Toulouse, France
Email: nicolas.amat@onera.fr

Pierre Ganty and Alessio Mansutti
IMDEA Software Institute, Madrid, Spain
Email: {pierre.ganty,alessio.mansutti}@imdea.org

***Abstract*—Lower bounds provide essential insights into the minimal computational resources required for algorithm execution. This paper focuses on logical theories, a domain where estimating resources is particularly difficult, and provides a novel, fully-automated method for computing lower bounds on memory usage, serving as a proxy for the computational resources required to perform logical reasoning. Specifically, the paper focuses on computing lower bounds on the size of the minimal deterministic finite automaton that encodes the solution set of a given Presburger arithmetic (also known as linear integer arithmetic) formula. The lower bounds are accompanied by independently verifiable certificates which also support a union-like operation that can be used to increase the computed bounds.**

**We conducted an extensive empirical evaluation of our method using over 5 000 formulae from the quantifier-free fragment of Presburger arithmetic, sourced from the SMT-LIB repository. The results show that our method often produces lower bounds that are close to the actual size of the minimal deterministic finite automaton. Moreover, it succeeds in computing non-trivial bounds even for instances that are out of reach (by several orders of magnitude) for the existing state-of-the-art automata-based tools for solving Presburger arithmetic.**

***Index Terms*—Presburger arithmetic, Finite automata, SMT.**

## I. Introduction

*On the use of lower bounds in Software Engineering:* Knowing a lower bound on the computing resources that an algorithm will require for a given input can be highly valuable. Such bounds are useful in a variety of contexts:

1) They specify a minimum of resources to be allocated (for instance, by cloud providers or system administrators).
2) Comparing the resource requirements of different algorithms for solving the same problem can help determine which algorithm is preferable.
3) Developers can use lower bounds as a sanity check, detecting bugs when an implementation consumes less resources than required by the lower bound.

This paper tackles the challenge of computing non-trivial lower bounds in the context of logical reasoning. Logical reasoning plays a prominent role in many Software Engineering tools. First-order reasoners, such as SMT solvers [8], are widely used in applications like bounded model checking (CBMC [2]), concolic testing (KLEE [3]), and verification of TLA+ specifications (Apalache [1]). Due to the inherent complexity of logical reasoning (even the simplest theories are NP-hard [28]), the performance of these tools is highly unpredictable. As a result, estimating the resources required for logical reasoning is a challenging and essential task.

Among the several logical theories of interest, a prominent one is *Presburger arithmetic*, also known as *linear integer arithmetic* [21]. This theory finds applications in several domains of computer science, such as polyhedral compilation [32], [31], formal methods [15], [5], [13] and theorem proving for combinatorics on words [29]. Its quantifier-free formulae constitute a significant portion of the SMT-LIB repository [8] which is used for the SMT-COMP [7]. Presburger arithmetic can be solved using a range of techniques, including symbolic computation (quantifier elimination) [33], geometric methods [16] and automata-based approaches [21]. In the worst case, all these techniques are highly space inefficient, making it often more practical to devote all computational resources to a single method rather than running multiple methods in parallel. Consequently, selecting which algorithm to run by comparing their resource requirements for a given query is a fundamental yet largely unexplored problem. Automata-based approaches construct a deterministic automaton encoding the set of solutions to a given formula. In this context, a key indicator of the computational resources required to solve the query is the size of the minimal deterministic finite automaton (minimal DFA) encoding this set.

*Our contribution:* In this paper, we formulate a fully automated method to compute non-trivial lower bounds on the size of minimal DFAs for quantifier-free Presburger formulae. Our method offers the following features:

**Scalability:** It computes lower bounds without computing the minimal DFAs themselves, enabling our method to scale beyond state-of-the-art automata-based approaches.

**Efficiency:** It is both space efficient, through dedicated data-structures, and time efficient, by allowing users to specify a time budget and obtain bounds following this constraint.

**Certification:** It provides verifiable lower bounds by generating independently checkable certificates (simply outputting a number is of little use). Multiple bounds can be combined by merging their certificates into a single, verifiable one, where the sum of the bounds represents the best-case scenario in the merged certificate.

*The gist of our method:* Let us start by recalling the basics the automata-based techniques for Presburger arithmetic. Each solution to the input quantifier-free Presburger formula $\varphi$ has

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00219

a corresponding encoding in the form of a word in a formal language. Each alphabet letter is a $d$-dimensional vector, one dimension per variable in $\varphi$, with entries in $\{0, 1\}$. That formal language represents all the encodings for all the solutions to $\varphi$, and is known to be regular [14]. Therefore, that language is accepted by a minimal DFA whose number of states we are trying to estimate from below. The states from the minimal DFA are in one-to-one correspondence with the equivalence classes of the so-called Nerode's congruence, following a classical result form language theory. The existence of a set of $n$ words each of which belongs to a distinct equivalence class of the Nerode's congruence is a proof that the minimal DFA has at least $n$ states. In a nutshell, our method leverages an SMT solver to get solutions to $\varphi$; the encodings as words of these solutions are then split and recombined following carefully designed heuristics to produce witnesses of the difference equivalence classes in the Nerode's congruence.

*Evaluation:* To empirically assess the strengths of our approach, we consider **all** the QF_LIA benchmarks from the SMT-LIB repository with a suitable "complexity measure" of at most 140 (see Section IV). This sums to a total of 5 684 quantifier-free Presburger arithmetic formulae. To put the computed lower bounds into perspective, we compare them with the size of the minimal DFAs returned by two state-of-the-art tools, whenever they succeed in computing them. These tools take as input a Presburger arithmetic formula and compute a minimal DFA whose language encodes all the solutions of the formula. The tools are Amaya [22] and Walnut [29].[1] At the time of writing, Walnut is the most established tool implementing automata-based procedures for Presburger arithmetic and its extensions. With almost a decade of development Walnut has a proven track record of automatically proving theorems in word combinatorics.

The bar graph of Fig. 1 gives a quick summary supporting the claim that our method scales beyond state-of-the-art automata-based approaches. This is clearly illustrated by the fading of the blue bars well before the orange ones. While most of the automata computed by Amaya and Walnut have hundreds of states, our method shows that the majority of the queries of the benchmark we consider have minimal automata with more than ten thousand states; and more than four hundred queries have a number of states in the millions.

Last but not least, returning to our initial discussion on the use of lower bounds, our tool allowed us to uncover subtle bugs in Amaya. These bugs were identified by observing that Amaya occasionally constructs automata smaller than our computed lower bounds, and their presence can be confirmed using either SMT solvers or our certificates. This highlights a practical utility of our method as a testing and diagnostic tool.

[1]Older automata-based tools for Presburger arithmetic are available, such as MONA [24], LASH [10] and Composite [9]. We focus on Amaya and Walnut as they build upon these older tools, and they are currently actively developed.

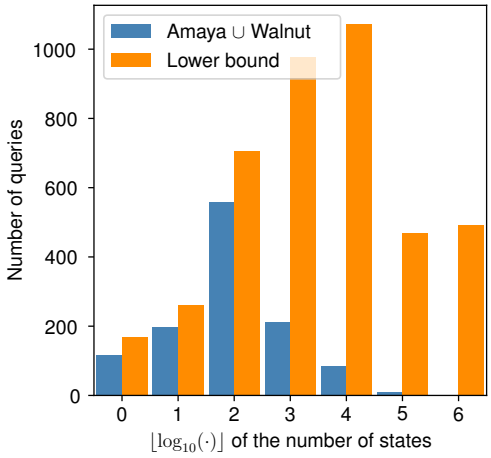



Fig. 1: Plot of our method (orange) and the tools Amaya and Walnut (blue). For our method, a bar of height $h$ in the $i$th column depicts that for $h$ formulae our tool reported a lower bound of $n \in [10^i..10^{i+1}-1]$ states. For Amaya and Walnut, a bar of height $h$ in the $i$th column depicts that for $h$ formulae at least one of these two tools managed to construct an equivalent automaton of $n \in [10^i..10^{i+1}-1]$ states.

## II. Presburger Arithmetic and Automata Thereof

In this section, we review the basics of Presburger arithmetic and automata constructions thereof. We also fix our notation.

*Presburger arithmetic (PA)* is the first-order theory of the structure $(\mathbb{Z}, +, \leq)$. We are interested in the *quantifier-free fragment* of PA, whose formulae are built from the grammar

$$\varphi, \psi \ ::= \ \bot \ \mid \ \top \ \mid \ \vec{a}^\intercal \cdot \vec{x} \leq c \ \mid \ \vec{a}^\intercal \cdot \vec{x} = c \\ \mid \ \varphi \wedge \psi \ \mid \ \varphi \vee \psi \ \mid \ \neg\varphi \ ,$$

where $\vec{a}^\intercal$ is a row vector of integer *coefficients*, $\vec{x}$ is a (column) vector of variables, taken from a countable set, and $c$ is an integer *constant*. (Here $.^\intercal$ denotes the operation of transposing a matrix or vector.) We write $\varphi(\vec{x})$ to denote the fact that the variables of the formula $\varphi$ are taken from the vector $\vec{x}$. A *solution* to $\varphi(x_1, \ldots, x_n)$ is any assignment $\sigma\colon \{x_1, \ldots, x_n\} \to \mathbb{Z}$ such that the (variable-free) sentence $\varphi(\sigma(x_1), \ldots, \sigma(x_n))$, obtained by replacing each $x_i$ with its value $\sigma(x_i)$, is valid. The validity of such sentences can be easily checked in polynomial time; whereas finding a solution is a canonical NP-complete problem [11].

### *Automata for Presburger Arithmetic*

The fact that Presburger arithmetic is an automatic structure, that is, a structure whose formulae admit a presentation by finite automata, is a classical result firstly proved by Büchi [14]. Several papers revisit this result [34], [12], [20], [22]. Below, we give an informal account of this construction, highlighting those properties that matter for our lower bound method.

As a start, one can describe an automaton for a (not necessarily quantifier-free) formula $\varphi(x_1, \ldots, x_n)$ of Presburger

arithmetic as a *deterministic*[2], *finite-state* machine that upon reading a word $w$ from the alphabet $\mathbb{B}^n \coloneqq \{0,1\}^n$, accepts if and only if the vector of integers encoded by $w$ is a solution to $\varphi$. In this respect, the first thing we need to clarify is in what sense does $w$ encode a vector of integers.

*How to encode the integers:* In the context of automata, the most natural way of encoding integers (and vectors thereof) is given by the two's complement representation. Both least-significant-digit-first (LSD) and most-significant-digit-first (MSD) encodings can be used. In this paper we focus on the LSD encoding, which is the encoding used, e.g., by the tool Amaya [22]. Our technique can be easily adapted to the MSD encoding. In the two's complement LSD representation, the non-empty word of binary digits $w = d_0 d_1 \cdots d_m \in \mathbb{B}^+$ encodes the integer $-d_m \cdot 2^m + \sum_{i=0}^{m-1} d_i \cdot 2^i$. A key property of this representation is that padding $w$ to the right by repeating its most significant digit does not change the encoded number. That is, $d_0 \cdots d_m$ and $d_0 \cdots d_m d_m \cdots d_m$, where $d_m$ is repeated an arbitrary amount of times, represent the same integer. Because of this property, we can encode any vector of $\mathbb{Z}^n$ as a word over the alphabet $\mathbb{B}^n$:

$$w = \begin{bmatrix} d_{1,0} \\ \vdots \\ d_{n,0} \end{bmatrix} \begin{bmatrix} d_{1,1} \\ \vdots \\ d_{n,1} \end{bmatrix} \begin{bmatrix} d_{1,m} \\ \vdots \\ d_{n,m} \end{bmatrix}$$

encodes

$$\rho_{\mathbb{Z}}(w) \coloneqq \begin{bmatrix} -d_{1,m}{\cdot}2^m + \sum_{i=0}^{m-1} d_{1,i}{\cdot}2^i \\ \vdots \\ -d_{n,m}{\cdot}2^m + \sum_{i=0}^{m-1} d_{n,i}{\cdot}2^i \end{bmatrix}.$$

The map $\rho_{\mathbb{Z}}$ is a surjection to $\mathbb{Z}^n$. Later in the paper, we will also need a surjection $\rho_{\mathbb{N}}$ from words to $\mathbb{N}^n$. For a word $w$ as the one above, the $j$th entry (with $j \in [1..n]$) of $\rho_{\mathbb{N}}(w)$ is defined as $\sum_{i=0}^{m} d_{j,i} \cdot 2^i$. Note that $\rho_{\mathbb{Z}}$ is not defined on the empty word $\varepsilon$, and indeed the automata does not accept the empty word. For $\rho_{\mathbb{N}}$, we postulate $\rho_{\mathbb{N}}(\varepsilon) \coloneqq \vec{0}$ instead.

We can now make our previous statement more precise: an automaton for a formula $\varphi(x_1, \ldots, x_n)$ reads non-empty words $w$ from the alphabet $\mathbb{B}^n$, and accepts if and only if $\rho_{\mathbb{Z}}(w)$ is a solution to $\varphi$.

*Where to place the acceptance condition:* A second point that needs to be clarified is whether we define the acceptance condition of the automaton using its states or its transitions. Because $\varepsilon$ does not encode any integer, the most natural choice is to use a transition-based acceptance condition: transitions of the automaton are marked as *final* or *non-final*, and the automaton accepts a word whenever the last transition taken is a final one. We remark that, by using the transition-based acceptance condition, no accepting run recognizes $\varepsilon$ regardless of the starting state of the run.

Previous research [26], [22] has shown that considering a transition-based acceptance condition is beneficial in terms of the sizes of the constructed automata. On one side, it is easy to see that for any automaton $A$ with a state-based acceptance condition and not accepting the empty word, there is a language-equivalent automaton $A'$ with a transition-based acceptance condition and the same number of states as $A$. One obtains $A'$ from $A$ by marking as accepting all transitions entering an accepting state of $A$. On the other side, $A'$ may also be reducible to a smaller automaton. This occurs in particular when $A$ features two states $q$ and $q'$ starting from which the automaton accepts the same language, except for the empty word (that is, only one among $q$ and $q'$ is a final state). In this case, $q$ and $q'$ can be merged in $A'$.

*Partial transition relation:* We consider automata with a transition function that is partial, i.e., the automata have at most one transition for each state and each alphabet symbol. We can thus assume that the automata have no sink state, as all transitions leading to such a state can be omitted.

*Assigning meaning to the states:* In the case of automata for quantifier-free Presburger arithmetic (which are the ones we are ultimately interested in), one last important property is that it is simple to *associate* to each state of the automata (reachable from the initial one) a formula that characterizes the words accepted by starting the automaton from that state. Let $\varphi(\vec{x})$ be a quantifier-free formula from PA, with $\vec{x} = (x_1, \ldots, x_n)$. Consider an enumeration

$$(\vec{a}_1^{\intercal} \cdot \vec{x} = c_1), \ldots, (\vec{a}_j^{\intercal} \cdot \vec{x} = c_j), \\ (\vec{a}_{j+1}^{\intercal} \cdot \vec{x} \leq c_{j+1}), \ldots, (\vec{a}_k^{\intercal} \cdot \vec{x} \leq c_k)$$

of all the equalities and inequalities occurring in $\varphi$. Given a vector $\vec{g} \coloneqq (g_1, \ldots, g_k)$ from $(\mathbb{Z} \cup \{\bot\})^j \times \mathbb{Z}^{k-j}$, we write $\varphi\langle\!\langle \vec{g} \rangle\!\rangle$ for the formula obtained from $\varphi$ as follows:

1) $\forall i \in [1..j]$ with $g_i \in \mathbb{Z}$, replace $\vec{a}_i^{\intercal} \cdot \vec{x} = c_i$ by $\vec{a}_i^{\intercal} \cdot \vec{x} = g_i$,
2) $\forall i \in [1..j]$ with $g_i = \bot$, replace $\vec{a}_i^{\intercal} \cdot \vec{x} = c_i$ by $\bot$, and
3) $\forall i \in [j+1..k]$, replace $\vec{a}_i^{\intercal} \cdot \vec{x} \leq c_i$ with $\vec{a}_i^{\intercal} \cdot \vec{x} \leq g_i$.

We *associate* to the initial state of the automaton the formula $\varphi$, which we note being syntactically equal to $\varphi\langle\!\langle c_1, \ldots, c_k \rangle\!\rangle$. Consider now a state $q$, and let $\varphi\langle\!\langle g_1, \ldots, g_k \rangle\!\rangle$ be one of the formulae associated to it. Also consider a word $w$ of length $\ell \geq 1$, and let $q'$ be the state reached from $q$ by reading $w$. We associate to $q'$ the formula $\varphi\langle\!\langle h_1, \ldots, h_k \rangle\!\rangle$, where

1) for every $i \in [1..j]$, $h_i \coloneqq \frac{g_i - \vec{a}_i^{\intercal} \cdot \rho_{\mathbb{N}}(w)}{2^\ell}$ only if $g_i \neq \bot$ and $g_i - \vec{a}_i^{\intercal} \cdot \rho_{\mathbb{N}}(w)$ is divisible by $2^\ell$, otherwise $h_i \coloneqq \bot$,
2) for every $i \in [j+1..k]$, $h_i \coloneqq \lfloor \frac{g_i - \vec{a}_i^{\intercal} \cdot \rho_{\mathbb{N}}(w)}{2^\ell} \rfloor$.

Multiple formulae can be associated to a single state, but all such formulae are logically equivalent. This is because the solutions to every formula associated to a state $q$ characterize the set of words accepted by the automaton starting from $q$. We denote this set as $L_A(q)$, where $A$ is the automaton.

**Proposition 1** ([12])**.** *Let $\varphi(x_1, \ldots, x_n)$ be a quantifier-free Presburger formula, and $A$ be an automaton for $\varphi$. Let $\psi$ be a formula associated to a state $q$ of $A$. For every $w \in (\mathbb{B}^n)^+$, $w \in L_A(q)$ if and only if $\rho_{\mathbb{Z}}(w)$ is a solution to $\psi$.*

As an example, Fig. 2 shows an automaton for $2 \cdot x - y \leq 1$.

[2]Unless stated otherwise, by "automaton" we mean a deterministic one.

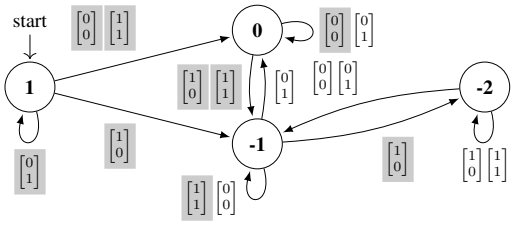



Fig. 2: An automaton for the formula $2 \cdot x - y \leq 1$. Accepting transitions are highlighted with a grey background. An integer $c$ appearing inside a state indicates that the formula $2 \cdot x - y \leq c$ is associated to that state.

By adapting the results in [20, Proposition 5] to the two's complement LSD encoding instead of the standard binary encoding of natural numbers, one can also show that, in the minimal automaton for a formula $\varphi$, every state has a corresponding formula $\varphi\langle\langle \vec{g} \rangle\rangle$ where the bit length of $\vec{g}$ is polynomially bounded in the bit length of $\varphi$. This implies an upper bound on the size of the minimal automaton for $\varphi$ (below, $|a|$ stands for the absolute value of an integer $a$):

**Proposition 2** ([20])**.** *Let $\varphi(x_1, \ldots, x_n)$ be a Boolean combination of $k$ (in)equalities $\{(a_{i,1}, \ldots, a_{i,n})^\intercal \cdot \vec{x} \sim_i c_i\}_{i=1}^k$, where $\sim_i$ in $\{=, \leq\}$. The minimal DFA for $\varphi$ has at most $2^k \prod_{i=1}^k \left(|c_i| + \sum_{j=1}^n |a_{i,j}| + 2\right)$ states.*

For our experimental evaluation (Section IV), we will use (the logarithm of) the upper bound given in Proposition 2 as a way of measuring the complexity of a query. We found no other good measure for predicting the performance of our method (just focusing on a simple parameter such as number of variables or number of atomic formulae yields no correlation).

## III. Computing Lower Bounds

In this section, we describe our method for computing lower bounds on the size of minimal (deterministic) automaton for quantifier-free Presburger arithmetic formulae. Parts of the method generalize to any automatic structure.

### A. Data Structures, Abstractions and Certificates

Given a regular language $L$, it follows from the MyHill-Nerode's theorem that the states of the minimal DFA for $L$ are in one-to-one correspondence with the equivalence classes of the following equivalence on words (given for automata with transition-based acceptance condition): $u \equiv v$ exactly when $u \circ w \in L$ iff $v \circ w \in L$ for all $w \in \Sigma^+$ — where $\circ$ stands for words concatenation. Classical textbooks often refer to this equivalence as the *Nerode's congruence*. Paraphrasing the formal definition of Nerode's congruence we have that two words $u$ and $v$ are *inequivalent* if there is a word $w$ we can append to them such that $u \circ w$ is in $L$ and $v \circ w$ is not or *vice versa*. Then, any set of $n$ pairwise inequivalent words contains representatives for $n$ distinct equivalence classes of the Nerode's congruence; meaning that the minimal DFA for $L$ must have at least $n$ states.

From the above, we can devise a simple method for computing lower bounds on the sizes of minimal automata for quantifier-free Presburger arithmetic formulae: find as many inequivalent words (in the Nerode's sense) as possible. To meet our desiderata to be checkable we can keep, for each pair of inequivalent words $u$ and $v$, a word $w$ that witnesses their inequivalence. We will discuss how to find these words in the next section. For now, assume we are given a set of "possibly inequivalent" words, which for simplicity we call *prefixes*, and a set of "possible witnesses", which we call *suffixes*. Inspired by Angluin's $L^*$ algorithm for learning DFAs [6] we organize prefixes and suffixes in a table. Rows and columns are indexed by these words, such that for a row indexed by a prefix $u$ and a column indexed by the suffix $v$, the table entry at position $(u, v)$ is 1 if $u \circ v \in L$, and 0 otherwise. Then, if the table has $n$ pairwise distinct rows (for each two prefixes $u$ and $v$ there is a suffix $w$ such that the table entries at $(u, w)$ and $(v, w)$ differ) then the minimal DFA for $L$ has at least $n$ states.

We could directly use these tables as both the main data structure and the certificate for our algorithm, but they come with two major drawbacks:

1) As we anticipate minimal automata, hence our lower bounds, to be large, data structures requiring $O(p \cdot s)$ memory, where $p$ is the number of prefixes and $s$ is the number of suffixes, may fail to scale up in practice. We should rather aim for a data structure of size $O(p + s)$.
2) Simply using words as indices for rows and columns may create a significant overhead. This is because, while searching for prefixes and suffixes, if two rows coincide on all entries, we might want to store them both anyway: we may later discover that there is a suffix that, when added to the table, differentiate the two. A clear drawback is that we may end up storing many prefixes that are in the same Nerode's congruence class. We need a way to recognize (as often as possible) these equivalent prefixes, so as to remove all but one from the table.

We solve the first drawback by only storing the prefixes $P$ and the suffixes $S$ (as lists of words), but not the table itself. Given these two lists, we can still compute a lower bound on the number of states of the minimal DFA in space $O(p+s)$ and time $O(p \cdot s)$. This is done by performing a *partition refinement* procedure on the list $P$ against the list $S$. In a nutshell, we start by taking an element $v$ from $S$, and split $P$ (in-place) into two blocks, $P_1$ and $P_2$, based on whether an element $u \in P$ is such that $u \circ v \in L$ or not. We recursively apply partition refinement on $P_1$ and $P_2$, against the list obtained from $S$ by removing $v$. The lower bound on the number of states of the minimal DFA corresponds then to the number of non-empty blocks generated by this process. Observe that partition refinement also intrinsically avoids superfluous membership queries: for example, if a block $P_1$ comprises a single element $u$, we do not need to refine it further. We will thus avoid checking $u \circ w \in L$ for all the remaining suffixes $w$ in $S$. The tree of recursions generated by partition refinement resembles the tree-like data structure described in [25, Sec. 8.3] in order to

store the $L^*$ table. That data structure has size $O(p \cdot s)$, which is the running time of partition refinement. However, partition refinement space usage is in $O(p+s)$, because of the in-place splitting of the list $P$.

Once the partition refinement has completed, we consider the set $R$ containing one representative prefix for each block; the words in $R$ are pairwise inequivalent (and so the cardinality of $R$ corresponds to the computed lower bound). We also consider the subset $W$ of those suffixes that produced non-trivial partitions (when $v$ splits $P$ into $P_1$ and $P_2$, the partition is non-trivial if both $P_1$ and $P_2$ are non-empty). The pair $(R, W)$ is our certificate. To check the lower bound, we can simply run again partition refinement on $R$ against $W$.

Let us now discuss how to solve the second drawback. In a nutshell, we will define two abstractions $\alpha_{\mathrm{p}}$ and $\alpha_{\mathrm{s}}$, one on prefixes and one on suffixes, such that

I. if $\alpha_{\mathrm{p}}(u) = \alpha_{\mathrm{p}}(v)$, then the two prefixes $u$ and $v$ belong to the same Nerode's congruence class;
II. if $\alpha_{\mathrm{s}}(u) = \alpha_{\mathrm{s}}(v)$, then the two suffixes $u$ and $v$ agree on all possible prefixes (that is, $w \circ u \in L$ if and only if $w \circ v \in L$, for every word $w$);
III. both $\alpha_{\mathrm{p}}$ and $\alpha_{\mathrm{s}}$ are polynomial-time computable from the prefix (resp. suffix) and the input quantifier-free Presburger arithmetic formula $\varphi$.

Because of Properties I and II, we skip adding a prefix to the table if its abstraction collides with that of a prefix in the table (and the same for suffixes). These pairs of abstractions rely on a crucial way on the structure of the automata for quantifier-free Presburger arithmetic described in Section II.

Let $\varphi(\vec{x})$, with $\vec{x} = (x_1, \ldots, x_n)$, be a quantifier-free Presburger arithmetic formula. As done in Section II, let us consider an enumeration

$$(\vec{a}_1^{\intercal} \cdot \vec{x} = c_1), \ldots, (\vec{a}_j^{\intercal} \cdot \vec{x} = c_j), \\ (\vec{a}_{j+1}^{\intercal} \cdot \vec{x} \leq c_{j+1}), \ldots, (\vec{a}_k^{\intercal} \cdot \vec{x} \leq c_k)$$

of all the equalities and inequalities occurring in $\varphi$. For a word $w$ of length $m$ over the alphabet $\mathbb{B}^n$, encoding $\vec{w} \coloneqq \rho_{\mathbb{N}}(w) \in \mathbb{N}^n$, we define $\alpha_{\mathrm{p}}(w)$ as follows:

$$\alpha_{\mathrm{p}}(w) \coloneqq \Big( \big(\tfrac{c_1 - \vec{a}_1^{\intercal} \cdot \vec{w}}{2^m}\big)_{\perp}, \ldots, \big(\tfrac{c_j - \vec{a}_j^{\intercal} \cdot \vec{w}}{2^m}\big)_{\perp}, \\ \lfloor \frac{c_{j+1} - \vec{a}_{j+1}^{\intercal} \cdot \vec{w}}{2^m} \rfloor, \ldots, \lfloor \frac{c_k - \vec{a}_k^{\intercal} \cdot \vec{w}}{2^m} \rfloor \Big);$$

where $(r)_{\perp} \coloneqq r$ whenever $r \in \mathbb{Z}$, and otherwise $(r)_{\perp} \coloneqq \perp$. Remark that if $w = \varepsilon$, then $\alpha_{\mathrm{p}}(w) = (c_1, \ldots, c_k)$.

For the abstraction on suffixes, given a *non-empty* word $w$ over the alphabet $\mathbb{B}^n$, encoding $\vec{w} \coloneqq \rho_{\mathbb{Z}}(w) \in \mathbb{Z}^n$, we define $\alpha_{\mathrm{s}}(w) \coloneqq (\vec{a}_1^{\intercal} \cdot \vec{w}, \ldots, \vec{a}_k^{\intercal} \cdot \vec{w})$.

To understand the intuition behind $\alpha_{\mathrm{p}}$ and $\alpha_{\mathrm{s}}$, consider a single constraint $\vec{a}^{\intercal} \cdot \vec{x} \sim c$, where $\sim$ is either $=$ or $\leq$. Also consider a non-empty word $w = d_1 d_2 \ldots d_m$ where each $d_i$ belongs to $\mathbb{B}^n$. Split $w$ into a prefix $u = d_1 \ldots d_i$ and a non-empty suffix $v = d_{i+1} \ldots d_m$. We have

$$\begin{aligned} \vec{a}^{\intercal} \cdot \rho_{\mathbb{Z}}(w) \sim c \iff & \vec{a}^{\intercal} \cdot (\rho_{\mathbb{N}}(u) + 2^i \cdot \rho_{\mathbb{Z}}(v)) \sim c \\ \iff & \vec{a}^{\intercal} \cdot (2^i \cdot \rho_{\mathbb{Z}}(v)) \sim c - \vec{a}^{\intercal} \cdot \rho_{\mathbb{N}}(u) \\ \iff & \vec{a}^{\intercal} \cdot \rho_{\mathbb{Z}}(v) \sim \frac{c - \vec{a}^{\intercal} \cdot \rho_{\mathbb{N}}(u)}{2^i}. \end{aligned} \quad (1)$$

Now, if $\sim$ stands for $=$, since the left-hand side of the last constraint is an integer, in the right-hand side $c - \vec{a}^{\intercal} \cdot \rho_{\mathbb{N}}(u)$ must be divisible by $2^i$. If instead $\sim$ stands for $\leq$, we can safely modify the right-hand side as $\lfloor \frac{c - \vec{a}^{\intercal} \cdot \rho_{\mathbb{N}}(u)}{2^i} \rfloor$. We then see that the left-hand side $\vec{a}^{\intercal} \cdot \rho_{\mathbb{Z}}(v)$ corresponds to the abstraction $\alpha_{\mathrm{s}}(v)$, while the right-hand side corresponds to the abstraction $\alpha_{\mathrm{p}}(u)$ (w.r.t. the constraint $\vec{a}^{\intercal} \cdot \vec{x} \sim c$).

Regarding the equivalences in Equation (1), two comments are in order. First, observe that our abstractions satisfy the properties I–III (Proposition 3), and they allow to quickly test if a prefix $u$ and a suffix $v$ are such that $u \circ v \in L$. It suffices to first compute the Boolean vector $\vec{b} \in \mathbb{B}^k$ given by[3]

- $\vec{b}[i] = 1$ iff $\alpha_{\mathrm{p}}(u)[i] = \alpha_{\mathrm{s}}(v)[i]$, for every $i \in [1..j]$,
- $\vec{b}[i] = 1$ iff $\alpha_{\mathrm{p}}(u)[i] \leq \alpha_{\mathrm{s}}(v)[i]$, for every $i \in [j+1..k]$.

Afterwards, we evaluate the Boolean structure of the formula $\varphi$ by replacing the $i$th atomic formula (with respect to the enumeration) with $\vec{b}[i]$. If this evaluation returns 1 (that is, true), then $u \circ v \in L$; else $u \circ v \notin L$. Whenever our procedure requires testing membership in $L$ (e.g, during partition refinement) it will do so in this way, using the abstractions.

The second remark is that we are constraining $v$ to be a non-empty suffix (and $\alpha_{\mathrm{s}}$ is undefined on the empty word). This is an effect of considering automata having a transition-based acceptance condition: since the empty word $\varepsilon$ is not accepted by any state of the automaton, we cannot consider it a suffix.

**Proposition 3.** *Properties I–III hold for $\alpha_{\mathrm{p}}$ and $\alpha_{\mathrm{s}}$.*

*Example:* Consider the formula $2 \cdot x - y \leq 1$ from Fig. 2, and take for instance the word $w \coloneqq \left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$. Note that $\rho_{\mathbb{Z}}(w) = (-5, 1)$, so $w$ encodes a solution to the formula. When splitting $w$ into the prefix $u \coloneqq \left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and the suffix $v \coloneqq \left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$, we have $\rho_{\mathbb{N}}(u) = (3, 1)$ and $\rho_{\mathbb{Z}}(v) = (-2, 0)$. The abstraction corresponding to $u$ and $v$ are, respectively, $\alpha_{\mathrm{p}}(u) = \lfloor \frac{1 - (2 \cdot 3 - 1)}{4} \rfloor = -1$, and $\alpha_{\mathrm{s}}(v) = 2 \cdot (-2) - 0 = -4$. In order to check that $w$ is a solution to the formula, it suffices to verify that $-4 \leq -1$. Also note that the value of $\alpha_{\mathrm{p}}(u)$ corresponds, in Fig. 2, to the integer appearing in the state of the automaton after reading $u$.

To summarize, in the implementation of our procedure we store the list of prefixes and (non-empty) suffixes, together with their abstractions. When two prefixes have the same abstraction, we keep only one in the list; and the same applies to suffixes. Moreover, we can easily check whether the concatenation of a prefix and a suffix is in the language using their

[3]The notation $[i]$ is a shorthand for the $i$th entry of a vector.

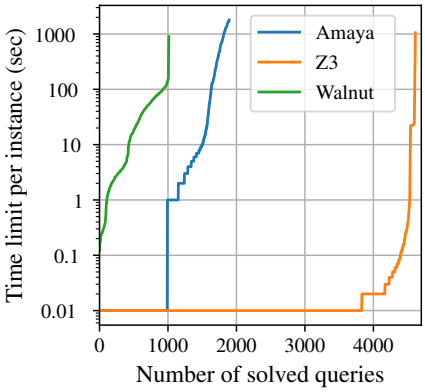



Fig. 3: Performance of Amaya, Walnut and z3 on some queries of the QF_LIA benchmark of the SMT-LIB repository. Details in Section IV.

abstractions. The certificates comprises inequivalent prefixes, together with the suffixes required to show the lower bound; the abstractions of these prefixes and suffixes is not stored, as they can be recomputed (property III). To "reduce" the list to inequivalent prefixes and suffixes, we use partition refinement.

### B. Finding Prefixes and Suffixes

Now that we have described our data structures, let us explain how they are populated; that is, how to pick prefixes and suffixes in practice.

As a first approximation, the procedure iteratively calls an SMT solver, asking for solutions to the input formula $\varphi(\vec{x})$, and populates the list of prefixes and suffixes using these solutions. Of course, the appeal to an SMT solver is very natural here: firstly because we cannot rely on automata procedure for Presburger arithmetic such as Amaya or Walnut (we are targeting queries that these tools cannot handle), and also because SMT solver are extremely efficient in solving quantifier-free Presburger arithmetic. The gap between these techniques is illustrated in Fig. 3. Ultimately, it is the existence of this gap in performances that will make our method work. *We stress that SMT solvers and automata-based procedures are not competing here, as they solve two different problems.* The former computes solutions to the input formula, whereas the latter solves the more computationally demanding problem of constructing an automaton representing *all* solutions.

The naïve solution of taking all prefixes and suffixes of a solution computed by the SMT solver and add them to the corresponding lists (if their abstraction is not already present) is not efficient. Going back to our initial organization of prefixes and suffixes in a table, we see that in the best-case scenario, a logarithmic number of suffixes compared to the number of prefixes suffices to certify a lower bound. Indeed, $s$ suffixes are in principle sufficient to obtain $2^s$ distinct rows of the table, and thus distinguish $2^s$ prefixes. Hence, a better strategy when it comes to populating the lists of prefixes and suffixes is, given a solution $w$ returned by the SMT, to add all the prefixes of $w$ to the list of prefixes, and add $w$ to the list of suffixes. (Again, these words are added to a list provided they do not share the same abstraction as other words already in the list; below we will avoid repeating this.) The decision of adding $w$ to the list of suffixes, instead of another of it suffixes, is not arbitrary (as we will see in the next paragraph) and works well in practice. With this strategy, our ratio of prefixes to suffixes is roughly quadratic: if the SMT solver returns $m$ solutions of length $m$, we will add $m^2$ words to the list of prefixes, and $m$ words to the list of suffixes. This is still very far from the (optimal) exponential ratio.

To improve our "quadratic ratio", we are obliged to add prefixes without adding new suffixes. There is no reason to do this with a solution returned from the SMT solver: we have already spent considerable computational effort to compute it, and the overhead of adding a single suffix is comparatively negligible. We must thus find new prefixes outside those computed via the SMT solver. Importantly, it should be possible for these prefixes to be extended to solutions of the input formula $\varphi$. Recall that we consider automata with a *partial* transition function and no sink state, hence only these prefixes correspond to states of the automaton. Our approach is the following: instead of always asking the SMT solver for solutions to the formula $\varphi$, after a fixed number $N$ of calls to the solver, we pick a random prefix $p$ from the list of prefixes. We call this prefix a *pivot*. We compute a new quantifier-free formula $\psi(\vec{x})$ whose solutions are given by the set $\{\rho_{\mathbb{Z}}(s) : \varphi(\rho_{\mathbb{Z}}(p \circ s))\}$. Essentially, we will be asking the SMT solver to complete the prefix $p$ to a solution of $\varphi$. We do not have to look far in order to compute $\psi$: using the notation from Section II, this formula is simply $\varphi\langle\!\langle\, \alpha_{\mathrm{p}}(p)\,\rangle\!\rangle$.

**Proposition 4.** *Let $\varphi$ be a quantifier-free formula from Presburger arithmetic, and $p$ be a word. Then, $\{\rho_{\mathbb{Z}}(s) : \varphi(\rho_{\mathbb{Z}}(p \circ s))\}$ is the set of solutions to $\varphi\langle\!\langle\, \alpha_{\mathrm{p}}(p)\,\rangle\!\rangle$.*

After computing $\psi$, we iterate through the list of suffixes, to see if some of them are already solutions to this formula. The key observation is that if there is one such suffix $s$, then there is a good chance that some words of the form $p \circ w$, where $w$ is a prefix of $s$, are not in the list of prefixes. This is because $s$ may have been computed starting from a different pivot, and thus previous iterations of the procedure never encountered words of the form $p \circ w$. Naturally, these words also satisfy the requirement to be extensible to a solution of $\varphi$; this is done by adding a suitable suffix of $s$. After $N$ calls to the SMT solver using $\psi$, we change the pivot again. Our previous choice of adding the solutions returned by the SMT solver to the list of suffixes should now be clear: these are the longest suffixes we have discovered, and the ones that in principle generate the maximum number of prefixes $p \circ w$.

Fig. 4 illustrates the above strategy. As evidence of its effectiveness, let us report our tool's performance on the query `prime_cone_sat_5` from the QF_LIA benchmark of the SMT-LIB repository. This query has 5 variables and 11 inequalities. With a thirty-minutes limit $T$ for the *exploration time*, our tool calls the SMT solver z3 almost 4 000 times.

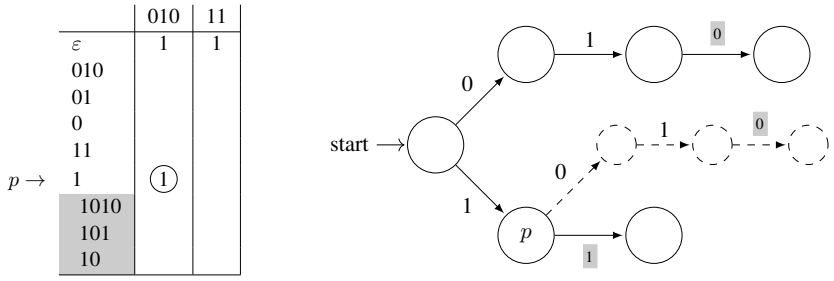



Fig. 4: Example of the strategy used to find new prefixes without appealing to the SMT solver. Starting from $\varepsilon$ as a pivot, suppose we have computed two solutions 010 and 11. Assume for simplicity that the abstraction $\alpha_{\mathrm{s}}$ is different for these solutions, and that $\alpha_{\mathrm{p}}$ is different from all their prefixes. In the minimal DFA, the runs for 010 and 11 thus visit distinct states (apart from the initial one); see figure on the right. The current state of the table is instead given on the left, excluding the rows highlighted in grey. Suppose the new pivot to be $p = 1$. Iterating through the suffixes, we discover that 1010 is a solution. We add to the table the new prefixes 1010, 101 and 10; highlighted in grey (if their abstractions $\alpha_{\mathrm{p}}$ is new). Likewise, we have discovered a new path ending with an accepting transition in the minimal DFA (dashed in the figure).

The bit size of each solution is at most 48. If we were only to consider the prefixes extracted from these solutions (the "quadratic ratio"), the tool would not compute more than 192 000 prefixes. Because of the approach described above, the tool computes instead more than 1.7 million prefixes *with pairwise distinct abstractions* $\alpha_{\mathrm{p}}$. After performing the partition refinement described in Section III-A, the tool produces a certificate showing that this query has at least 1.3 million states. We can prove this lower bound with 337 of the $\sim$4 000 suffixes. This is of course still very far from the exponential ratio of prefixes over suffixes: in principle, showing that a query has 1.3 million states may require as little as 21 suffixes. However, this ratio is most probably unattainable in practice, due to the properties of the underlying DFA.

Back to the algorithm, since we are now iterating through the list of suffixes in order to find solutions to $\psi$, we may as well make the best out of this computation. If we find that a suffix $s$ is a solution to $\psi$, we update $\psi$ by adding a constraint $\alpha_{\mathrm{s}}(\vec{x}) \neq \alpha_{\mathrm{s}}(s)$.[4] This constraint forces the SMT solver to find a solution whose abstraction differs from $\alpha_{\mathrm{s}}(s)$. We perform a similar update also when the SMT solver computes a new solution; so that solutions with previously unseen abstractions are found in the subsequent calls to the solver. An interesting benefit of this is that when the set $\{\alpha_{\mathrm{s}}(s) : s \text{ solution of } \psi\}$ is finite, the SMT solver will eventually return `unsat`. Our tool will then mark the pivot $p$ as *complete*, as to not consider it again when randomly selecting pivots. This concludes the overall description of our method. Fig. 5 provides a pseudocode of the algorithm, as a way of summarizing the content of this section. By default, 10 is the use count $N$ of each pivot and 30 min is the limit $T$ for the exploration time.

### C. Theoretical guarantees on the computed lower bound

By analysing the pseudocode in Fig. 5, we infer several properties of our procedure. First and foremost, the correctness

[4]Here, $\alpha_{\mathrm{s}}(\vec{x})$ is here short for $(\vec{a}_1^{\intercal} \cdot \vec{w}, \ldots, \vec{a}_k^{\intercal} \cdot \vec{w})$, where the indices $1, \ldots, k$ follow the enumeration on the constraints in $\varphi$ used to define $\alpha_{\mathrm{s}}$. It is then easy to encode $\alpha_{\mathrm{s}}(\vec{x}) \neq \alpha_{\mathrm{s}}(s)$ as a quantifier-free formula of PA.

**Fixed:** $N \geq 1$: use count of each pivot
**Fixed:** $T > 0$: the time limit for exploration time
**Input:** $\varphi(\vec{x})$: a quantifier-free Presburger formula
**Output:** A certificate for the lower bound on the number of states of the minimal DFA for $\varphi$

1: $P \leftarrow$ empty list of prefixes $u$ and their abstractions $\alpha_{\mathrm{p}}(u)$
2: $S \leftarrow$ empty list of prefixes $v$ and their abstractions $\alpha_{\mathrm{s}}(v)$
3: add to $P$ the pair $(\epsilon, \alpha_{\mathrm{p}}(\epsilon))$, where $\epsilon$ is the empty word
4: **while** $T$ is not exceeded and not all prefixes are marked as *complete* **do**
5: $\quad p \leftarrow$ random prefix in $P$ that is not marked as *complete*
6: $\quad \psi(\vec{x}) \leftarrow$ the formula $\varphi\langle\!\langle\, \alpha_{\mathrm{p}}(p) \,\rangle\!\rangle$
$\quad\quad \triangleright$ *characterizes* $\{\rho_{\mathbb{Z}}(s) : \varphi(\rho_{\mathbb{Z}}(p \circ s))\}$
7: $\quad$ **for** $(s, \alpha_{\mathrm{s}}(s))$ in $S$ **do**
8: $\quad\quad$ **if** $s$ is a solution to $\psi$ **then**
9: $\quad\quad\quad$ add the constraint $\alpha_{\mathrm{s}}(\vec{x}) \neq \alpha_{\mathrm{s}}(s)$ to $\psi$
$\quad\quad\quad\quad \triangleright$ *expressible in quantifier-free PA*
10: $\quad\quad\quad$ **for** $u$ prefix of $s$ **do**
11: $\quad\quad\quad\quad$ add $(p \circ u, \alpha_{\mathrm{p}}(p \circ u))$ to $P$ only if $\alpha_{\mathrm{p}}(p \circ u)$ is new
12: $\quad it \leftarrow 0$
13: $\quad$ **while** $it < N$ **do**
14: $\quad\quad v \leftarrow$ **try** GETSOLUTION$(\psi, T)$ **catch** (mark $p$ as *complete*; **break**)
15: $\quad\quad it \leftarrow it + 1$
16: $\quad\quad$ add the constraint $\alpha_{\mathrm{s}}(\vec{x}) \neq \alpha_{\mathrm{s}}(v)$ to $\psi$
17: $\quad\quad$ add $(v, \alpha_{\mathrm{s}}(v))$ to $S$ if $\alpha_{\mathrm{s}}(v)$ is new
18: $\quad\quad$ **for** $w$ prefix of $v$ **do**
19: $\quad\quad\quad$ add $(p \circ w, \alpha_{\mathrm{p}}(p \circ w))$ to $P$ if $\alpha_{\mathrm{p}}(p \circ w)$ is new
20: perform partition refinement on $P$ against $S$, obtaining a pair $(R, W)$ of inequivalent prefixes $R$, and witnesses $W$
21: **return** $(R, W)$ $\quad \triangleright$ *the minimal DFA has* $\geq |R|$ *states*

Fig. 5: Pseudocode of our method.

of the procedure follows from the MyHill-Nerode theorem.

**Proposition 5.** *Let $(R, W)$ be the output of the algorithm from Figure 5 on an input formula $\varphi$. Then, the number $|R|$ of prefixes in $R$ is a lower bound to the number of states in the minimal DFA for $\varphi$.*

A natural question is whether it is possible to evaluate the quality of our lower bounds from a purely theoretical perspective. First, note that if the input formula $\varphi$ is unsatisfiable or valid, the algorithm outputs $(R, W)$ with $|R| = 1$, as expected. For formulae that are satisfiable but not valid, this question is instead very challenging. The difficulty lies in the fact that, given sufficient time and mild assumptions, our algorithm can eventually determine the exact number of states of the automaton (except for the sink state, since we are only adding to the table prefixes that extend into a solution of the input formula). The assumptions are:

A1. The constraints $\alpha_{\mathrm{s}}(\vec{x}) \neq \alpha_{\mathrm{s}}(s)$ from lines 9 and 16 are replaced with $\vec{x} \neq s$. This replacement can be enabled in our tool using the `--concrete` option; in practice, we have not obtained better bounds using this option.

A2. The SMT solver used to find solutions (in line 14) is *enumerating*: for a given formula $\varphi$, (1) the user has a way to query the solver for the "next solution" to $\varphi$, if any, and (2) the user can obtain each solution $\sigma$ to $\varphi$ by invoking the "next-solution" mechanism a finite amount of times. In other words, for each $\sigma$ there is an $i \in \mathbb{N}$ such that the solver returns $\sigma$ after $i$ "next solutions".

**Proposition 6.** *Suppose (A1) and (A2) to hold. Let $\varphi$ be a quantifier-free Presburger formula that is neither unsatisfiable nor valid. There is a time limit $T$ with respect to which the algorithm in Figure 5 outputs $(R, W)$ such that the minimal DFA for $\varphi$ has $|R| + 1$ states.*

In the SMT community, *enumerating* SMT solvers are also referred to as *AllSMT* solvers. While very limited until a few years ago, AllSMT has become an active area of research in recent years; see for instance the recent papers [27], [30]. Some solvers, such as MathSAT [17], have an AllSMT mode. However, performances are for now inferior to standard SMT solving, and for this reason we still use z3 in the upcoming experimental evaluation. It is worth noticing that assumption A2 becomes unnecessary when the input formula $\varphi$ has only finitely many solutions. In that case, the blocking of solutions performed in lines 9 and 16 when using the `--concrete` option of our tool suffices to enumerate all solutions. Moreover, the finiteness of the set of solutions guarantees that all prefixes are eventually marked as complete, ensuring termination of the **while** loop of line 4, regardless of the time limit.

**Proposition 7.** *Suppose (A1) holds. Let $\varphi$ be a satisfiable quantifier-free Presburger formula having finitely many solutions. With $T = \infty$, the algorithm in Figure 5 outputs $(R, W)$ such that the minimal DFA for $\varphi$ has $|R| + 1$ states.*

A more practical approach to evaluating the quality of our lower bounds is to analyze, for all instances where automata-based tools successfully construct the minimal automaton of an input formula, how closely our computed lower bound approximates the actual size of the minimal DFA. This analysis is presented in the forthcoming experimental evaluation.

## IV. Experimental Results

We implemented our algorithm in a open-source tool called PATAPSCO (Presburger Arithmetic To Automata Prober of State COmplexity)[5]. The tool parses quantifier-free Presburger arithmetic formulae following the SMT-LIB format [8]; internally the parsing is managed the pySMT library [19]. We rely on the SMT solver z3 [18] to compute the solutions to be added to the lists of prefixes and suffixes, using the SMT-LIB format to exchange data with the solver. For numerical computations, such as the computation of the abstractions $\alpha_{\mathrm{p}}$ and $\alpha_{\mathrm{s}}$, we rely on the numPy library [23]. As the integers computed during the procedure may exceed 64 bits (this happens quite often when computing the abstractions) great care has been taken to make correctly handle integers of arbitrary length, in both our data structures and inside library invocations.

In this section, we analyze the performance our or tool on a benchmark of more than $5\,000$ queries from the SMT-LIB repository [8] (this repository includes all formulae used in the competition SMT-COMP [7]). To put the lower bounds we compute into perspective, we contrast the results of our tool with that of the state-of-the-art automata-based tools Amaya[6] [22] and Walnut[7] [29]. To replicate the experiments an artifact has been made available on Zenodo [4].

*Query complexity:* To properly benchmark our tool, we need a measure estimating the "complexity" of a quantifier-free Presburger formula. As mentioned at the end of Section II, we found no other sensible measure than the upper bound from Proposition 2 . Since this upper bound is usually very large, we take its logarithm as a measure of complexity. More precisely, if the number of states in the minimal automaton for a formula $\varphi$ has an upper bound $\alpha$, in the sense of Proposition 2, then we define the complexity of the formula $\varphi$ as $\lfloor \log_{10}(\alpha) \rfloor$.

*Description of the Benchmark Suite:* We selected **all** queries with complexity at most $140$ from the QF_LIA track of the SMT-LIB repository. As observed in Fig. 7, our tool currently performs poorly on queries with complexity above $120$. The threshold was selected to properly highlight this fact. This resulted in $5\,684$ quantifier-free Presburger formulae.

We computed the satisfiability status of the queries from our benchmark using z3, with a $30\,\mathrm{min}$ time limit. Beyond this time limit, we labelled queries as *Unknown* ($70$ among the $5\,684$ queries). Of the total queries, $4\,147$ are satisfiable. The distribution of these queries in terms of their complexity is shown in Fig. 6. As lower bounds only matters for satisfiable queries, in the remaining part of the section we will not discuss the unsatisfiable and unknown ones.

[5]https://patapsco.software.imdea.org

[6]Release: github.com/MichalHe/amaya/commit/f47bccd...

[7]Release: github.com/firetto/Walnut/commit/a4fe84e...

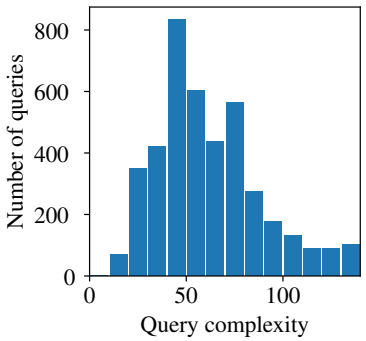
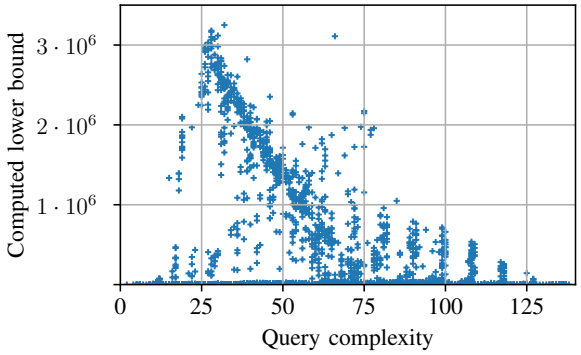



Fig. 6: Query complexity distribution for satisfiable queries.




MEAN: $2.8 \cdot 10^5$ – MEDIAN: $9.4 \cdot 10^3$ – MAX: $6.2 \cdot 10^6$

Fig. 7: Computed lower bounds vs. query complexity (only satisfiable queries are shown). MAX not plotted for clarity.

### A. Computed Lower Bounds

We now report the performance of our tool on the above benchmark. For the experiments, we imposed a 30 min limit for the exploration time $T$ computing the lists of prefixes and suffixes. After this limit, the tool proceeds to compute the lower bound using the partition refinement algorithm. We imposed an overall 12 h time limit for each query, which we hit a handful of times (see Section V for more information). The mean time for partition refinement is 17 min.

Each satisfiable query is a point in Fig. 7 whose position relates the query complexity and the lower bound computed. Our tool performs best in the complexity range 20–40. With a few notable exceptions, all lower bounds of at least 2 million fall within this range, and conversely, the majority of queries in this complexity range have more than 2 million states. In a nutshell, in this range automata are large, yet z3 is efficient enough to produce many solutions within the 30 min limit specified by the exploration time $T$. Beyond a complexity of 40, our tool's performance degrades and, above complexity 120, the tool computes a handful of interesting lower bounds. Notably, the tool computes lower bounds above 500 000 states for queries with complexity around 100. Under the (perhaps strong) assumption that the upper bound in Proposition 2 is reasonably tight (asymptotically, this is the case for infinitely many queries), these queries are beyond what any method based on computing DFAs can achieve, as the automata may have close to $10^{100}$ states.

Since it is difficult to grasp how queries having a certain complexity look like, TABLE I provides more statistics on queries in the "optimal" complexity range 20–40 as well as in the "last viable" complexity range (100–120). Since the most interesting statistics are from queries where the tool excels, this table comprises queries with lower bounds above 100 000 states. We conclude from the statistics on the average value of the coefficients, that the atomic formulae of these queries are "sparse", i.e., not many variables feature non-zero coefficients.

### B. Validation using State-of-the-art Tools

We now contrast our lower bounds with the sizes of the (minimal) automata constructed by Amaya and Walnut. We recall some important features of these tools:

- Walnut assumes variables range over $\mathbb{N}$ and offers several encodings, including LSD binary, which we use. Furthermore, Walnut uses a state-based acceptance condition. This implies that the lower bounds computed by our tool cannot exceed the number of states of the automata Walnut returns. Indeed, consider one such automaton for a formula $\varphi$; and assume it is minimal. Because of the LSD binary encoding, each accepting state has a transition labelled with the vector $\vec{0}$ ending in an accepting state (as appending 0s does not change the vector of numbers a word encodes). It turns out that making such transition accepting results in an automaton for $\varphi$ respecting the definition given in Section II. All statistics we give for Walnut are for queries having all their variables in $\mathbb{N}$ (4 630 out of a total of 5 684 queries).
- Amaya uses the automata described in Section II, with binary decision diagrams to encode transitions. Observe that both the description given in Section II, and our lower bounds, are agnostic of how transitions are represented.

In a first experiment, we ran Walnut, Amaya (and z3) on queries with variables over $\mathbb{N}$. Fig. 3 plots, for each tool, the number of solved queries with a 30 min time limit per query. As Amaya scaled up to larger automata than Walnut, we increased its time limit to 1 h and 2 h per query (results in TABLE II). Observe that the increased time limit benefits Amaya marginally: only 8% more queries are solved, none of which improved the maximum automaton constructed.

Returning to Fig. 1, our tool infers lower bounds for the sizes of automata that are much larger compared to what state-of-the-art automata-based tools can handle. Fig. 7 and TABLE II convey a similar message: across all runs of Amaya and Walnut, the average number of states of the automata is at most 4 340 (Amaya, 2 h time limit); the average lower bound our tool computes is of 280 000 states.

### C. Coverage

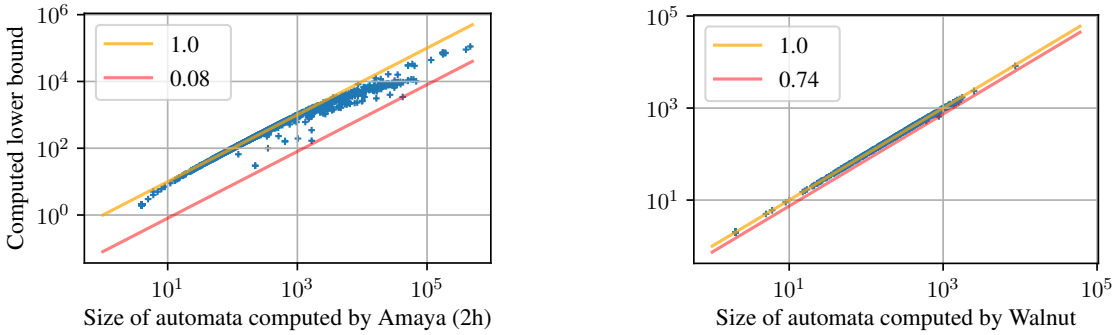
Completing the discussion from Section III-C on the quality of the computed bounds, let us look at the *coverage* of the

TABLE I: Statistics for the queries with lower bounds above 100 000, for the "optimal" complexity range (20–40) and the "last viable" complexity range (100–120). |COEFF.| is the absolute value of the maximum coefficient appearing in the formula.

| COMPLEXITY RANGE | AVG(# VARIABLES) | AVG(# ATOMIC FORMULAE) | AVG(\|COEFF.\|) |
|---|---|---|---|
| 20–40 | 26.2 | 23.9 | 0.5 |
| 100–120 | 83 | 106.8 | 0.1 |

TABLE II: Queries solved by Amaya and Walnut and statistics on the computed automata (for the satisfiable queries).

| TOOL (TIMEOUT) | #ANSWERS | #SAT QUERIES | #STATES | | |
|---|---|---|---|---|---|
| | | | MEAN | MEDIAN | MAX |
| *Walnut* (30 min) | 1 001 | 630 | 317 | 216 | 8 659 |
| *Amaya* (30 min) | 1 959 | 1 014 | 2 798 | 234 | 464 775 |
| *Amaya* (1 h) | 2 041 | 1 092 | 3 438 | 261 | 464 775 |
| *Amaya* (2 h) | 2 129 | 1 173 | 4 328 | 305 | 464 775 |




MEAN: 0.82 – MEDIAN: 0.97 – MIN: 0.08

MEAN: 0.99 – MEDIAN: 0.97 – MIN: 0.74

Fig. 8: Plots of the coverage of the computed lower bounds (for the satisfiable queries). The left plot does not depict instances where Amaya behaves incorrectly (see Section IV-D).

lower bounds for the satisfiable queries for which Amaya or Walnut successfully construct the minimal automaton. Since ours is the first tool for computing these bounds, this is the only possible analysis to assess their quality in practice.

For a given query, by *coverage* we mean the ratio between the computed lower bound and the size of the minimal automaton. It is depicted in Fig. 8. Notably, both the mean and median values exceed 0.8 for Amaya, indicating that in the majority of cases our lower bound is at least 80% of the actual size of the minimal automaton. Moreover, since the mean (0.82) is below the median (0.97), we infer that only for a few queries our tool did not manage to significantly "cover" the states of the automaton; indeed, for over 75% of the queries the coverage exceeds 0.64. More in details, our coverage is close to 1 for automata with fewer than 1 000 states. This is particularly evident in the plot corresponding to Walnut (whose computed automata rarely surpass 1 000 states): the mean and median are in this case above 0.95.

### D. Finding bugs

As discussed in the introduction, computing lower bounds provides a simple correctness check for automata-based tools: if a tool constructs a DFA smaller than the lower bound computed by our tool, this indicates a bug. We conducted a "bug-finding campaign" that, rather than focusing on the performance of our tool, aimed at identifying implementation bugs in Amaya and Walnut. For these experiments, we considered queries beyond the SMT-LIB, and ran multiple parallel instances of our tool on the same query, later combining the computed certificates to obtain tighter lower bounds (this methodology is explained in the next section). While the results of Walnut were always consistent with our bounds, we have encountered implementation bugs in Amaya. The contradictions we observed fall into two types:

- For ten queries, Amaya returned a minimal DFA with fewer states than our lower bound. On nine of these queries, Amaya in fact returned a DFA for an empty language, as if the query is UNSAT. Our lower bound implies instead that the query is SAT. In these cases, the bug in Amaya was independently confirmed by solving the query using the SMT solver cvc5, which returned SAT. In the remaining case, Amaya returned a non-empty DFA having five states less than our lower bound. In this case the bug could not be confirmed using SMT solvers (all tools agree on the satisfiability of the query), but it

can still be confirmed using our certificates, as explained at the end of the section. Remarkably, this is the unique query of our bug-finding campaign that is not part of the benchmark suite. It is also an outlier in terms of the number of variables (100) and (in)equalities (338), compared to the other queries Amaya can solve.
- Amaya and Walnut return minimal DFAs with different number of states. This occurred for one query, where the DFA returned by Amaya has approximately twice the size of that returned by Walnut. On this query, our tool computed a lower bound that matched the number of states of the DFA returned by Walnut, making it very unlikely that Walnut is incorrect.

We reported the first type of bugs to the Amaya developers, who attributed them preprocessing heuristics. An update in summer 2025 fixed one of the nine UNSAT errors; the rest remain. We nevertheless decided to keep Amaya in our performance evaluation because it outperformed Walnut, and we believe its underlying theory to be sound. We do not expect fixing these bugs to significantly affect our empirical results.

Beyond establishing lower bounds, the certificates generated by our tool can aid in confirming the presence of bugs. Let us start with a simple observation: distinct prefixes from the certificate of a formula $\varphi$ must lead to distinct states in the minimal DFA for $\varphi$. To see why this holds, consider two distinct prefixes $u_1$ and $u_2$ from the certificate. Because they are part of the certificate, the certificate contains a suffix $v$ such that either $\rho_{\mathbb{Z}}(u_1 \cdot v)$ satisfies $\varphi$ while $\rho_{\mathbb{Z}}(u_2 \cdot v)$ does not, or vice versa. Hence, in any DFA for $\varphi$, the words $u_1$ and $u_2$ must lead to different states. This reasoning is applicable to all computed DFAs. Furthermore, if the DFA has fewer states than the number of prefixes in the certificate (as observed in the first type of bugs encountered in Amaya), there will always be two prefixes that end up in the same state, confirming the bug. Although this technique is simple to describe, its implementation requires instrumenting the automata-based tools (in our case Amaya or Walnut) to traverse the constructed DFA. Since our primary focus was on computing lower bounds rather than utilizing the certificates, we did not extend Amaya to support this form of bug detection.

## V. Large Lower Bounds by Combining Certificates

Previously, we mentioned that a few queries caused our tool to hit the $12\,\mathrm{h}$ time limit. On these queries (which are all in the complexity range 20–40), the number of computed prefixes exceeded $10^7$. This combined with a high number of suffixes prevented the partition refinement to complete within $12\,\mathrm{h}$.

In cases like these, one can obtain high lower bounds by setting a low solving time (e.g., $5\,\mathrm{min}$), run the tool several times, and then combine all certificates into a single one. Combining certificate is a simple operation: merge the lists of prefixes and suffixes from two or more certificates, and then run partition refinement on the resulting lists. For large automata this approach is more efficient than one single run, and leads to better coverage (it also allows for parallelization). Indeed, on one side, unneeded prefixes and suffixes are discarded during each partition refinement, speeding up the computation of the last, combined certificate. On the other side, this approach alleviates a down-side of our strategy of finding prefixes by using pivots. Consider an automaton in which, starting from the initial state, several strongly connected components are reachable. Suppose that, after calling the SMT solver $N$ times using the empty word $\varepsilon$ as a pivot, the tool computed $p$ prefixes (excluding $\varepsilon$), all of which correspond to states within a few of the strongly connected components. Discovering the other components requires re-selecting $\varepsilon$ as a pivot, which happens with probability $\frac{1}{p+1}$. The tool may thus fail to explore the other components in a reasonable time. In practice, things are not as extreme, but there are still parts of the automaton that, depending on the list of prefixes, have a lower probability of being discovered. Performing several runs, each giving a different random seed to the SMT solver, increases the chances of discovering more portions of the automaton.

As an evidence to support the combination of certificate let us put forward the query `prime_cone_sat_7` (complexity 23) for which we computed a combined certificate for a lower bound of $96$ million states, our highest lower bound yet.

## VI. Conclusion and Future Work

We have developed (and evaluated) a method for computing non-trivial lower bounds on the size of minimal DFAs for quantifier-free Presburger formulae. As demonstrated in our experiments involving the tool Amaya, a practical application of these lower bounds is bug detection. Beyond this, we believe that our method might turn out to be useful in other ways.

First, our tool can in principle be used to improve "portfolio approaches" for Presburger arithmetic. As explained in the introduction, all techniques for solving this theory (quantifier elimination, automata methods, and geometric procedures) are very inefficient in the worst case. Little is known regarding which of these three techniques one should appeal to, given a particular formula. Running our tool for a few minutes can give a quick way of excluding automata from the pool of options. Further experiments performed on the part of the QF_LIA benchmark considered in Section IV show that the average lower bound computed by our tool with a time limit of $3\,\mathrm{min}\,45\,\mathrm{sec}$ is above $18\,000$; more than four times the average size of the automata computed by Amaya in 2 hours.

Second, our results suggest that SMT solvers can be used to efficiently find states of minimal automata. In this respect, we believe that automata-based tools could incorporate queries to SMT solvers to improve their performances when it comes to constructing the automaton of a quantifier-free (sub)formula.


## Acknowledgments

This work was funded by MICIU/AEI (grant CEX2024-001471-M and PID2022-138072OB-I00) and FEDER, UE. Alessio Mansutti is co-funded by the European Union (GA 101154447). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the EU or European Commission. Neither the European Union nor the granting authority can be held responsible for them.

## References


[1] Apalache: A symbolic model checker for TLA+. https://github.com/apalache-mc/apalache. Accessed: October 3, 2025.

[2] CBMC: Bounded model checking for software. https://github.com/diffblue/cbmc. Accessed: October 3, 2025.

[3] KLEE symbolic execution engine. https://github.com/klee/klee. Accessed: October 3, 2025.

[4] N. Amat, P. Ganty, and A. Mansutti. Artifact for ASE 2025 Paper: "How Big is the Automaton? Certified Lower Bounds on the Size of Presburger DFAs". Zenodo, 2025. doi: 10.5281/zenodo.17235269.

[5] T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic timing verification of timing diagrams using presburger formulas. In *DAC*, 1997. doi: 10.1145/266021.266071.

[6] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 1987. doi: 10.1016/0890-5401(87)90052-6.

[7] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. In *CAV*, 2005. doi: 10.1007/11513988_4.

[8] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2016. url: smt-lib.org [Accessed: October 3, 2025].

[9] C. Bartzis and T. Bultan. Automata-based representations for arithmetic constraints in automated verification. In *CIAA*, 2002. doi: 10.1007/3-540-44977-9_30.

[10] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, Liège, Belgium, 1998. url: orbi.uliege.be/bitstream/2268/74874/1/Boigelot98.pdf [Accessed: October 3, 2025].

[11] I. Borosh and L. B. Treybig. Bounds on positive integral solutions of linear diophantine equations. *Proc. Am. Math. Soc.*, 1976. doi: 10.2307/2041711.

[12] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *CAAP*, 1996. doi: 10.1007/3-540-61064-2_27.

[13] M. Bozga, R. Iosif, and F. Konečný. Fast acceleration of ultimately periodic relations. In *CAV*, 2010. doi: 10.1007/978-3-642-14295-6_23.

[14] J. R. Büchi. Weak second-order arithmetic and finite automata. *Math. Logic Quart.*, 1960. doi: 10.1002/malq.19600060105.

[15] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *CAV*, 1997. doi: 10.1007/3-540-63166-6_39.

[16] D. Chistikov, C. Haase, and A. Mansutti. Geometric decision procedures and the VC dimension of linear arithmetic theories. In *LICS*, 2022. doi: 10.1145/3531130.3533372.

[17] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The mathsat5 SMT solver. In *TACAS*, 2013. doi: 10.1007/978-3-642-36742-7_7.

[18] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008. doi: https://doi.org/10.1007/978-3-540-78800-3_24.

[19] M. Gario and A. Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*, 2015. url: andrea.micheli.website/papers/pysmt.pdf [Accessed: October 3, 2025].

[20] F. Guépin, C. Haase, and J. Worrell. On the existential theories of Büchi arithmetic and linear p-adic fields. In *LICS*, 2019. doi: 10.1109/LICS.2019.8785681.

[21] C. Haase. A survival guide to Presburger arithmetic. *ACM SIGLOG News*, 5(3):67–82, 2018.

[22] P. Habermehl, V. Havlena, M. Hečko, L. Holík, and O. Lengál. Algebraic reasoning meets automata in solving linear integer arithmetic. In *CAV*, 2024. doi: 10.1007/978-3-031-65627-9_3.

[23] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 2020. doi: 10.1038/s41586-020-2649-2.

[24] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, 1995. doi: 10.1007/3-540-60630-0_5.

[25] M. J. Kearns and U. Vazirani. *An introduction to computational learning theory*. MIT press, 1994. doi: 10.7551/mitpress/3897.001.0001.

[26] A. Martin, E. Renault, and A. Duret-Lutz. Translation of semi-extended regular expressions using derivatives. In *Proc. of Int'l Conf. on Implementation and Application of Automata*, page 234–248. Springer, 2024. doi: 10.1007/978-3-031-71112-1_17.

[27] G. Masina, G. Spallitta, and R. Sebastiani. On CNF conversion for SAT and SMT enumeration. *J. Artif. Intell. Res.*, 83, 2025. doi: 10.1613/JAIR.1.16870.

[28] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, 1981. doi: 10.1145/322276.322287.

[29] J. Shallit. *The Logical Approach to Automatic Sequences: Exploring Combinatorics on Words with Walnut*. Number 482 in London Mathematical Society Lecture Note Series. Cambridge university press, 2023. doi: 10.1017/9781108775267.

[30] G. Spallitta, R. Sebastiani, and A. Biere. Disjoint projected enumeration for SAT and SMT without blocking clauses. *Artif. Intell.*, 345:104346, 2025. doi: 10.1016/J.ARTINT.2025.104346.

[31] S. Verdoolaege. Presburger formulas and polyhedral compilation, 2021. See libisl.sourceforge.io/tutorial.pdf [Accessed: October 3, 2025].

[32] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 2007. doi: 10.1007/s00453-006-1231-0.

[33] V. Weispfenning. The complexity of almost linear diophantine problems. *J. Symb. Comput.*, 1990. doi: 10.1016/S0747-7171(08)80051-X.

[34] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *SAS*, 1995. doi: 10.1007/3-540-60360-3_30.