Counting the Solutions of Presburger Equations without Enumerating Them*

Bernard Boigelot and Louis Latour

Institut Montefiore, B28
Université de Liège
B-4000 Liège Sart-Tilman, Belgium
{boigelot,latour}@montefiore.ulg.ac.be
http://www.montefiore.ulg.ac.be/~{boigelot,latour}/

Abstract. The *Number Decision Diagram (NDD)* has recently been proposed as a powerful representation system for sets of integer vectors. In particular, NDDs can be used for representing the sets of solutions of arbitrary Presburger formulas, or the set of reachable states of some systems using unbounded integer variables. In this paper, we address the problem of counting the number of distinct elements in a set of vectors represented as an NDD. We give an algorithm that is able to perform an exact count without enumerating explicitly the vectors, which makes it capable of handling very large sets. As an auxiliary result, we also develop an efficient projection method that allows to construct efficiently NDDs from quantified formulas, and thus makes it possible to apply our counting technique to sets specified by formulas. Our algorithms have been implemented in the verification tool LASH, and applied successfully to various counting problems.

1 Introduction

Presburger arithmetic [Pre29], i.e., the first-order additive theory of integers, is a powerful formalism for solving problems that involve integer variables. The manipulation of sets defined in Presburger arithmetic is central to many kinds of applications, including integer programming problems [Sch86,PR96], compiler optimization techniques [Pug92], temporal database queries [KSW95], and program analysis tools [FO97,SKR98].

The more direct way of handling algorithmically Presburger-definable sets consists of using a formula-based representation system. This approach has been successfully implemented in the *Omega package* [Pug92], which is probably the most widely used Presburger tool at the present time. Unfortunately, formula-based representations suffer from a serious drawback : They lack canonicity, which implies that sets with a simple structure are in some situations represented by very complex formulas; this notably happens when these formulas are

* This work was partially funded by a grant of the "Communauté française de Belgique — Direction de la recherche scientifique — Actions de recherche concertées", and by the European Commission (FET project ADVANCE, contract No IST-1999-29082).

obtained as the result of lengthy sequences of operations. Moreover, the absence of a canonical representation hinders the efficient implementation of usually essential decision procedures, such as testing whether two sets are equal to each other.

In order to alleviate these problems, an alternative representation of Presburger-definable sets has been developed, based on finite-state automata. The *Number Decision Diagram (NDD)* [WB95,Boi99] is, sketchily, a finite-state machine recognizing the encodings of the integer vectors belonging to the set that it represents. Its main advantage are that most of the usual set-theory operations can be performed by simply carrying out the corresponding task on the languages accepted by the automata, and that a canonical representation of a set can easily be obtained by minimizing its associated automaton. Among its applications, the NDD has made it possible to develop a tool for computing automatically the set of reachable states of programs using unbounded integer variables [LASH].

The problem of counting how many elements belong to a Presburger-definable set has been solved for formula-based representations [Pug94] of Presburger sets. Though of broad scope, this problem has interesting applications related to verification and program analysis. First, it can be used in order to quantify precisely the performances of some systems. In particular, one can estimate the computation time of code fragments or the amount of resources that they consume wherever these quantities can be expressed as Presburger formulas. Furthermore, counting the number of reachable data values at some control locations makes it possible to detect quickly some inconsistencies between different releases of a program, without requiring to write down explicit properties. For instance, it can promptly alert the developer, although without any guarantee of always catching such errors, that a local modification had an unwanted influence on some remote part of the program. Finally, studying the evolution of the number of reachable states with respect to the value of system parameters can also help to detect unsuspected errors.

The main goal of this paper is to present a method for counting the number of elements belonging to a Presburger-definable set represented by an NDD. Intuitively, our approach is based on the idea that one can easily compute the number of distinct paths of a directed acyclic graph without enumerating them. The actual algorithm is however more intricate, due to the fact that the vectors belonging to a set and the accepting paths of its representing NDD are not linked to each other by a one-to-one relationship.

In order to apply our counting technique to the set of solutions of a given Presburger formula, one needs first to build an NDD from that formula. This problem has been solved in [BC96,Boi99], but only in the form of a construction algorithm that is exponentially costly in the number of variables involved in the formula. As an auxiliary contribution of this paper, we describe an improved algorithm for handling the problematic projection operation. The resulting construction procedure has been implemented and successfully applied to problems involving large numbers of variables.

2 Basic Notions

We here explain how finite-state machines can represent sets of integer vectors. The main idea consists of establishing a mapping between vectors and words. Our encoding scheme for vectors is based on the classical expression of numbers in a base $r > 1$, according to which an encoding of a positive integer z is a word $a_{p-1}a_{p-2}\cdots a_1a_0$ such that each *digit* a_i belongs to the finite alphabet $\{0, 1, \ldots, r-1\}$ and $z = \sum_{i=0}^{p-1} a_i r^i$. Negative numbers z have the same p-digit encoding as their r's complement $r^p + z$. The number p of digits is not fixed, but must be large enough for the condition $-r^{p-1} \leq z < r^{p-1}$ to hold. As a corollary, the first digit of the encodings is 0 for positive numbers and $r-1$ for negative ones, hence that digit is referred to as the *sign digit* of the encodings.

In order to encode a vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$, one simply reads repeatedly and in turn one digit from the encodings of all its components, under the additional restriction that these encodings must share the same length. In other words, an encoding of $\boldsymbol{v}$ is a word $d_{p-1,1}d_{p-1,2}\ldots d_{p-1,n}d_{p-2,1}d_{p-2,2}\ldots d_{0,n-1}d_{0,n}$ such that for every $i \in \{1, \ldots, n\}$, $d_{p-1,i}d_{p-2,i}\ldots d_{0,i}$ is an encoding of v_i. An encoding of a vector of dimension n has thus n sign digits — each associated to one vector component — the group of which forms a *sign header*.

Let $S \subseteq \mathbf{Z}^n$ be a set of integer vectors. If the language $L(S)$ containing all the encodings of all the vectors in S is regular, then any finite-state automaton accepting $L(S)$ is a *Number Decision Diagram (NDD)* representing S. It is worth noticing that, according to this definition, not all automata defined over the alphabet $\{0, 1, \ldots, r-1\}$ are valid NDDs. Indeed, an NDD must accept only valid encodings of vectors that share the same dimension, and must accept all the encodings of the vectors that it recognizes. Note that the vector encoding scheme that we use here is slightly different from the one proposed in [BHMV94,Boi99], in which the digits related to all the vector components are read simultaneously rather than successively. It is easy to see that both representation methods are equivalent from the theoretical point of view, the advantage of our present choice being that it produces considerably more compact finite-state representations. For instance, a minimal NDD representing $\mathbf{Z}^n$ is of size $O(2^n)$ if it reads component digits simultaneously, which limits the practical use of that approach to small values of n. On the other hand, our improved encoding scheme yields an automaton of size $O(n)$.

It is known for a long time [Cob69,Sem77] that the sets that can be represented by finite-state automata in every base $r > 1$ are exactly those that are definable in *Presburger arithmetic*, i.e., the first-order theory $\langle \mathbf{Z}, +, \leq \rangle$. One direction of the proof of this result is constructive, and translates into a algorithm for constructing an NDD representing an arbitrary Presburger formula [BHMV94]. Sketchily, the idea is to start from elementary NDDs corresponding to the formula atoms, and to combine them by means of set operators and quantifiers. It is easily shown that computing the union, intersection, difference or Cartesian product of two sets represented by NDDs is equivalent to carrying out similar operations on the languages accepted by the underlying automata. Quantifying existentially a set with respect to a vector component, which amounts to *pro-*

jecting this set along this component, is more tedious. We discuss this problem in the next section.

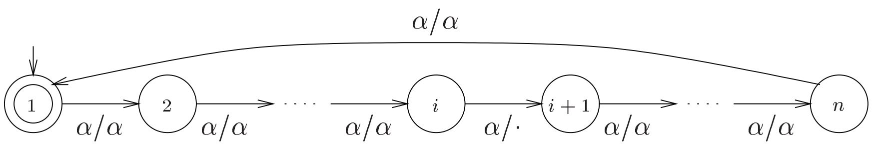
At this time, one could wonder why we did not opt for defining NDDs as automata accepting only one encoding (for instance the shortest one) of each vector, and for encoding negative numbers as their sign followed by the encoding or their absolute value. It turns out that these alternate choices complicate substantially some elementary manipulation algorithms, such as computing the Cartesian product or the difference of two sets, as well as the construction of the automata representing atomic formulas, such as linear equations or inequations. On the other hand, our present choices lead to simple manipulation algorithms, with the only exceptions of projection and counting.

3 Projecting NDDs

The projection problem can be stated in the following way. Given an NDD $\mathcal{A}$ representing a set $S \subseteq \mathbf{Z}^n$, with $n > 0$, and a component number $i \in \{1, \ldots, n\}$, the goal is to construct an NDD $\mathcal{A}'$ representing the set

$$\exists_i S = \{(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \mid (v_1, \ldots, v_n) \in S\}.$$

For every accepting path of $\mathcal{A}$, there must exist a matching path of $\mathcal{A}'$, from the label of which the digits corresponding to the i-th vector component are excluded. Thus, one could be tempted to compute $\mathcal{A}'$ as the direct result of applying to $\mathcal{A}$ the transducer depicted at Figure 1.



For all transitions, $\alpha \in \{0, \ldots, r-1\}$.

Fig. 1. Projection transducer.

Unfortunately, this method produces an automaton $\mathcal{A}|_{\neq i}$ that, even though it accepts valid encodings of all the elements of $\exists_i S$, is generally not an NDD. Indeed, for some vectors, the automaton may only recognize their encodings if they are of sufficient length, think for instance of $\exists_1\{(4, 1)\}$. In order to build $\mathcal{A}'$ from $\mathcal{A}|_{\neq i}$, one thus has to transform the automaton so as to make it also accept the shorter encodings of the vectors that it recognizes.

Clearly, two encodings of the same vector only differ in the number of times that their sign header is repeated. We can thus restate the previous problem in the following way: Given a finite-state automaton $\mathcal{A}_1$ of alphabet Σ accepting the language L_1, and a dimension $n \geq 0$, construct an automaton $\mathcal{A}_2$ accepting $L_2 = \{u^i w \mid u \in \{0, r-1\}^n \wedge w \in \Sigma^* \wedge i \in \mathbf{N} \wedge (\exists k > 0)(k \geq i \wedge u^k w \in L_1)\}$.

In [Boi99], this problem is solved by considering explicitly every potential value u of the sign header, and then exploring $\mathcal{A}_1$ in order to know what states can be reached by a prefix of the form u^i, with $i > 0$. It is then sufficient to make each of these states reachable after reading a single occurrence of u, which can be done by a simple construction, and to repeat the process for other u. Although satisfactory from a theoretical point of view, this solution exhibits a systematic cost in $O(2^n)$ which limits its practical use to problems with a very small vector dimension.

The main idea behind our improved solution consists of handling simultaneously sign headers that cannot be distinguished from each other by the automaton $\mathcal{A}_1$, i.e., sign headers $u_1, u_2 \in \{0, r-1\}^n$ such that for every $k > 0$, reading u_1^k leads to the same automaton states as reading u_2^k. For simplicity, we assume $\mathcal{A}_1$ to be deterministic[1].

Our algorithm proceeds as follows. First, it extracts from $\mathcal{A}_1$ a *prefix automaton* $\mathcal{A}_P$ that reads only the first n symbols of words and associates one distinct end state to each group of undistinguished sign headers. Each end state of $\mathcal{A}_P$ is then matched to all the states of $\mathcal{A}_1$ that can be reached by reading the corresponding sign headers any number of times. At every time during this operation one detects two sign headers that are not yet distinguished but that lead to different automaton states, one refines the prefix automaton $\mathcal{A}_P$ so as to associate a different end state to each header. Finally, the automaton $\mathcal{A}_2$ is constructed in such a way that following one of its accepting paths amounts to reading n symbols in $\mathcal{A}_P$, which results in reaching an end state s of this automaton, and then following an accepting path of $\mathcal{A}_1$ starting from a state matched to s.

The algorithm is formally described in Appendix A. Its worst-case time complexity is not less than that of the simple solution [Boi99] outlined at the beginning of this section. However, in the context of state-space exploration applications, we observed that it succeeds most of the time, if not always, to avoid the exponential blowup experienced with the latter approach.

4 Counting elements of NDDs

We now address the problem of counting the number of vectors that belong to a set S represented by an NDD $\mathcal{A}$. Our solution proceeds in two steps: First, we check whether S is finite or infinite and, in the former case, we transform $\mathcal{A}$ into a deterministic automaton $\mathcal{A}'$ that accepts exactly one encoding of each vector that belongs to S. Second, we count the number of distinct accepting paths in $\mathcal{A}'$.

4.1 Transformation step

Let $\mathcal{A}$ be an NDD representing the set $S \subseteq \mathbf{Z}^n$. If S is not empty, then the language accepted by $\mathcal{A}$ is infinite, hence the transition graph of this automaton

[1] This is not problematic in practice, since the cost of determinizing an automaton built from an arithmetic formula is often moderate [WB00].

contains cycles. In order to check whether S is finite or not, we thus have to determine if these cycles are followed when reading different encodings of the same vectors, or if they can be iterated in order to recognize an infinite number of distinct vectors.

Assume that $\mathcal{A}$ does not contain unnecessary states, i.e., that all its states are reachable and that there is at least one accepting path starting from each state. We can classify the cycles of $\mathcal{A}$ in three categories:

- A *sign loop* is a cycle that can only be followed while reading the sign header of an encoding, or a repetition of that sign header;
- An *inflating loop* is a cycle that can never be followed while reading the sign header of an encoding or one of its repetitions;
- A *mixed loop* is a cycle that is neither a sign nor an inflating loop.

If $\mathcal{A}$ has at least one inflating or mixed loop, then one can find an accepting path in which one follows that loop while not reading a repetition of a sign header. By iterating the loop, one thus gets an infinite number of distinct vectors, which results in S being infinite. The problem thus reduces to checking whether $\mathcal{A}$ has non-sign (i.e., inflating or mixed) loops[2]. Thanks to the following result, this check can be carried out by inspecting the transition graph of $\mathcal{A}$ without paying attention to the transition labels.

Theorem 1. *Assume that $\mathcal{A}$ is a deterministic and minimal (with respect to language equivalence) NDD. A cycle λ of $\mathcal{A}$ is a sign loop if and only if it can only be reached by one path (not containing any occurrence of that cycle).*

Proof. Since $\mathcal{A}$ is an NDD, it can only accept words whose length is a multiple of n. The length of λ is thus a multiple of n.

- *If λ is reachable by only one path π.* Let $u \in \{0, r-1\}^n$ be the sign header that is read while following the n first transitions of the path $\pi\lambda$, and let s and s' be the states of $\mathcal{A}$ respectively reached after reading the words u and uu (starting from the initial state).
  Since $\mathcal{A}$ accepts all the encodings of the vectors in S, it accepts, for every $w \in \{0, 1, \ldots, r-1\}^*$, the word uw if and only if it accepts the word uuw. It follows that the languages accepted from the states s and s' are identical which implies, since $\mathcal{A}$ is minimal, that $s = s'$.
  Therefore, λ can only be visited while reading the sign header u or its repetition, and is thus a sign loop.
- *If λ is reachable by at least two paths π_1 and π_2.* Let kn, with $k \in \mathbf{N}$ be the length of λ. Since $\mathcal{A}$ only accepts words whose length is a multiple of n, there are exactly k states $s_1, s_2, \ldots, s_k$ that are reachable in λ from the initial state of $\mathcal{A}$ after following a multiple of n transitions.
  If the words read by following λ from s_1 to s_2, from s_2 to s_3, …, and from s_k to s_1 are not all identical, then λ is not a sign loop. Otherwise, let u^k, with $u \in \{0, 1, \ldots, r-1\}^n$, be the label of λ.

[2] An example of a non-trivial instance of this problem can be obtained by building the minimal deterministic NDD representing the set $\{(x, y) \in \mathbf{Z}^2 \mid x + y \leq 0 \wedge x \geq 0\}$.

Since $\mathcal{A}$ is deterministic, at least one of the blocks of n consecutive digits read while following π_1 or π_2 up to reaching λ differs from u. Thus, λ can be visited while not reading a repetition of a sign header. $\square$

Provided that $\mathcal{A}$ has only sign loops, it can easily be transformed into an automaton $\mathcal{A}'$ that accepts exactly one encoding of each vector in S by performing a depth-first search in which one removes, for each detected cycle, the transition that gets back to a state that has already been visited in the current exploration path. This operation does not influence the set of vectors recognized by the automaton, since the deleted transitions can only be followed while reading a repeated occurrence of a sign header.

An algorithm that combines the classification of cycles with the transformation of $\mathcal{A}$ into $\mathcal{A}'$ is given in Appendix B. Since each state of $\mathcal{A}$ has to be visited at most once, the time and space costs of this algorithm – if suitably implemented – are linear in the number of states of $\mathcal{A}$.

4.2 Counting step

If S is finite, then the transition graph of the automaton $\mathcal{A}'$ produced by the algorithm given in the previous section is acyclic. The number of vectors in S corresponds to the number of accepting paths originating in the initial state of $\mathcal{A}'$.

For each state s of $\mathcal{A}'$, let $N(s)$ denote the number of paths of $\mathcal{A}'$ that start at s and end in an accepting state. Each of these paths either leaves s by one of its outgoing transitions, or has a zero length (which requires s to be accepting). Thus, we have at each state s

$$N(s) = \sum_{(s,d,s')\in\Delta} N(s') + acc(s),$$

where $acc(s)$ is equal to 1 if s is accepting, and to 0 otherwise.

Thanks to this rule, the value of $N(s)$ can easily be propagated from the states that have no successors to the initial state of $\mathcal{A}'$, following the transitions backwards. The number of additions that have to be performed is linear in the number of states of $\mathcal{A}'$.

5 Example of use

The projection and counting algorithms presented in Sections 3 and 4 have been implemented in the verification tool LASH [LASH], whose main purpose is to compute exactly the set of reachable configurations of a system with finite control and unbounded data. Sketchily, this tool handles finite and infinite sets of configurations with the help of finite-state representations suited for the corresponding data domains, and relies on *meta-transitions*, which capture the repeated effect of control loops, for exploring infinite state spaces in finite time. A description of the main techniques implemented by LASH is given in [Boi99].

In the context of this paper, we focus on systems based on unbounded integer variables, for which the set representation system used by LASH is the NDD. Our present results thus allow to count precisely the number of reachable system configurations that belong to a set computed by LASH.

Let us now describe an example of a state-space exploration experiment featuring the counting algorithm. We consider the simple lift controller originally presented in [Val89]. This system is composed of two processes modeling a lift panel and its motor actuator, communicating with each other by means of shared integer variables. A parameter N, whose value is either fixed in the model or left undetermined, defines the number of floors of the building. In the former case, one observes that the amount of time and of memory needed by LASH in order to compute the set of reachable configurations grows only logarithmically in N, despite the fact that the number of elements in this set is obviously at least $O(N^2)$. (Indeed, the behavior of the lift is controlled by two main variables modeling the current and the target floors, which are able to take any pair of values in $\{1, \ldots, N\}^2$.)

Our simple experiment has two goals: Studying precisely the evolution of the number of reachable configurations with respect to increasing values of N, and evaluating the amount of acceleration induced by meta-transitions in the state-space exploration process.

The results are summarized in Figures 2 and 3. The former table gives, for several values of N, the size (in terms of automaton states) of the finite-state representation of the reachable configurations, the exact number of these configurations, and the total time needed to perform the exploration. These results clearly show an evolution in $O(N^2)$, as suspected. It is worth mentioning that, thanks to the fact that the cost of our counting algorithm is linear in the size of NDDs, its execution time (including the classification of loops) was negligible with respect to that of the exploration.

N	NDD states	Configurations	Time (s)
10	852	930	25
100	1782	99300	65
1000	2684	9993000	101
10000	3832	999930000	153
100000	4770	99999300000	196
1000000	5666	9999993000000	242

Fig. 2. Number of reachable configurations w.r.t. N.

The latter table shows, for $N = 10^9$, the evolution of the number of configurations reached after the successive steps of the exploration algorithm. Roughly speaking, the states are explored in a breadth-first fashion, starting from the initial configuration and following transitions as well as meta-transitions, until

Step	NDD states	Configurations
1	638	3
2	1044	1000000003
3	1461	3999999999
4	2709	500000005499999997
5	4596	1500000006499999995
6	6409	3500000004499999994
7	7020	6499999997499999999
8	7808	7999999995000000000
9	8655	8999999994000000000
10	8658	9499999993500000000
11	8663	9999999993000000000

Fig. 3. Number of reached configurations w.r.t. exploration steps.

a fixpoint is detected. In the present case, the impact of meta-transitions on the number of reached states is clearly visible at Steps 2 and 4.

6 Conclusions and comparison with other work

The main contribution of this paper is to provide an algorithm for counting the number of elements in a set represented by an NDD. As an auxiliary result, we also present an improved projection algorithm that makes it possible to build efficiently an NDD representing the set of solutions of a Presburger formula. Our algorithms have been implemented in the tool LASH.

The problem of counting the number of solutions of a Presburger equation has already been addressed in [Pug94], which follows a formula-based approach. More precisely, that solution proceeds by decomposing the original formula into an union of disjoint convex sums, each of them being a conjunction of linear inequalities. Then, all but one variable are projected out successively, by splitering the sums in such a way that the eliminated variables have one single and one upper bound. This eventually yields a finite union of simple formulas, on which the counting can be carried out by simple rules.

The main difference between this solution and ours is that, compared to the general problem of determining whether a Presburger formula is satisfiable, counting with a formula-based method incurs a significative additional cost. On the other hand, the automata-based counting method has no practical impact on the execution time once an NDD has been constructed. Our method is thus efficient for all the cases in which an NDD can be obtained quickly, which, as it has been observed in [BC96,WB00], happens mainly when the coefficients of the variables are small. In addition, since automata can be determinized and minimized after each manipulation, NDDs are especially suited for representing the results of complex sequences of operations producing simple sets, as in most state-space exploration applications. The main restriction of our approach is that it cannot be generalized in a simple way to the more complex counting problems, such as summing polynomials over Presburger-definable sets, that are addressed in [Pug94].

References

[BC96] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proceedings of CAAP'96*, number 1059 in Lecture Notes in Computer Science, pages 30–43. Springer-Verlag, 1996.

[BHMV94] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bulletin of the Belgian Mathematical Society*, 1(2):191–238, March 1994.

[Boi99] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. Collection des publications de la Faculté des Sciences Appliquées de l'Université de Liège, Liège, Belgium, 1999.

[Cob69] A. Cobham. On the base-dependence of sets of numbers recognizable by finite automata. *Mathematical Systems Theory*, 3:186–192, 1969.

[FO97] L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *Proceedings of CONCUR'97*, volume 1243, pages 213–227, Warsaw, Poland, July 1997. Springer-Verlag.

[KSW95] F. Kabanza, J.-M. Stevenne, and P. Wolper. Handling infinite temporal data. *Journal of computer and System Sciences*, 51(1):3–17, 1995.

[LASH] The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[PR96] M. Padberg and M. Rijal. *Location, Scheduling, Design and Integer Programming*. Kluwer Academic Publishers, Massachusetts, 1996.

[Pre29] M. Presburger. Über die Volständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves*, pages 92–101, Warsaw, Poland, 1929.

[Pug92] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, pages 102–114, August 1992.

[Pug94] W. Pugh. Counting solutions to Presburger formulas: How and why. *SIGPLAN*, 94-6/94:121–134, 1994.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & sons, Chichester, 1986.

[Sem77] A. L. Semenov. Presburgerness of predicates regular in two number systems. *Siberian Mathematical Journal*, 18:289–299, 1977.

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Proceedings of the 10th Intl. Conf. on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 280–292, Vancouver, June/July 1998. Springer-Verlag.

[Val89] A. Valmari. State space generation with induction. In *Proceedings of the SCAI'89*, pages 99–115, Tampere, Finland, June 1989.

[WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proceedings of Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32, Glasgow, September 1995. Springer-Verlag.

[WB00] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.

A Projection algorithm

Let $(\Sigma, Q, s^{(0)}, \Delta, F)$ be the automaton $\mathcal{A}_1$, where Σ is the alphabet $\{0, \ldots, r-1\}$, Q is a finite set of states, $s^{(0)} \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ is a set of accepting states.

1. Let $\mathcal{A}_P = (\Sigma, Q_P, s_P^{(0)}, \Delta_P, F_P)$, with $s_P^{(0)} = (s^{(0)}, 0)$, $Q_P = \{s_P^{(0)}\}$, and $\Delta_P = F_P = \emptyset$. Each state (s, i) of $\mathcal{A}_P$ is composed of a state s of $\mathcal{A}_1$ associated with an index i ranging from 0 to n. The index n corresponds to the end states.
2. For $i = 1, \ldots, n$ and for each $(s, \alpha, s') \in \Delta$ such that $(s, i-1) \in Q_P$, add (s', i) to Q_P and $((s, i-1), \alpha, (s', i))$ to Δ_P.
3. For each $s \in Q$ such that $(s, n) \in Q_P$, let $\mathit{matches}[(s, n)] = \{s\}$.
4. Let $\mathit{remaining} = \{(s, s) \mid (s, n) \in Q_P\}$.
5. For each $(s, s') \in \mathit{remaining}$:
   - If there does not exist $s'' \in Q \setminus \mathit{matches}[(s, n)]$ and $u \in \Sigma^n$ such that $(s_P^{(0)}, u, (s, n)) \in \Delta_P^*$ and $(s', u, s'') \in \Delta^*$, then remove (s, s') from *remaining*.
   - If there exists $s'' \in Q \setminus \mathit{matches}[(s, n)]$ such that for every $u \in \Sigma^n$ for which $(s_P^{(0)}, u, (s, n)) \in \Delta_P^*$, $(s', u, s'') \in \Delta^*$, then add s'' to the set $\mathit{matches}[(s, n)]$, add (s, s'') to *remaining*, and remove (s, s') from *remaining*.
   - Otherwise, find $u, u' \in \Sigma^n$ such that $(s_P^{(0)}, u, (s, n)) \in \Delta_P^*$, $(s_P^{(0)}, u', (s, n)) \in \Delta_P^*$ and either
     - there exists $s'', s''' \in Q$, $s'' \neq s'''$, such that $(s', u, s'') \in \Delta^*$ and $(s', u', s''') \in \Delta^*$, or
     - there exists $s'' \in Q$ such that $(s', u, s'') \in \Delta^*$ but no $s''' \in Q$ such that $(s', u', s''') \in \Delta^*$,

     then refine $\mathcal{A}_P$ with respect to the state s' and the headers u and u' (this operation will be described separately).
6. Let $\mathcal{A}_2 = (\Sigma, Q_2, s_2^{(0)}, \Delta_2, F_2)$, with $Q_2 = Q \cup Q_P$, $s_2^{(0)} = s_P^{(0)}$, $\Delta_2 = \Delta \cup \Delta_P \cup \{((s, n), \varepsilon, s') \mid s' \in \mathit{matches}[(s, n)]\}$, and $F_2 = F$.

It is worth mentioning that the test performed at Line 5 can be carried out efficiently by a search in the transition graph of the automata. The details of this operation are omitted from this short description.

A central step of the algorithm consists of refining the prefix automaton $\mathcal{A}_P$ in order to associate different end states to two sign headers u and u' read from the state s' of $\mathcal{A}_1$. This operation is performed as follows:

1. Let $k \in \{1, \dots, n\}$ be the smallest integer such that the paths reading u and u' from the state $s_P^{(0)}$ of $\mathcal{A}_P$ reach the same state after having followed k transitions, and the paths reading u and u' from the state s' of $\mathcal{A}_1$ reach two distinct states after the same number k of transitions.
2. Let $((s_1, k-1), d, (s_2, k))$ and $((s'_1, k-1), d', (s_2, k))$ be the k-th transitions of the paths reading (respectively) u and u' in $\mathcal{A}_P$.
3. For each $q \in Q_P$ such that $((s_2, k), w, q) \in \Delta_P^*$ for some $w \in \Sigma^*$, add a new state q' to Q_P and set $\mathit{split}[q] = q'$.
4. For each transition $(q, d, q') \in \Delta_P$ such that $\mathit{split}[q]$ is defined, add the transition $(\mathit{split}[q], d, \mathit{split}[q'])$ to Δ_P.
5. Replace the transition $((s'_1, k-1), d', (s_2, k))$ by $((s'_1, k-1), d', \mathit{split}[(s_2, k)])$ in Δ_P.
6. For each $q \in Q_P$ such that $\mathit{split}[q]$ exists, let $\mathit{matches}[\mathit{split}[q]] = \mathit{matches}[q]$.
7. For each $(s, s') \in \mathit{remaining}$ such that $\mathit{split}[(s, n)]$ is defined, add the pair $(\mathit{split}[(s, n)],\, s')$ to *remaining*.

B Cycle classification and removal algorithm

1. Let $\mathcal{A} = (\Sigma, Q, s^{(0)}, \Delta, F)$, let $\mathit{visited} = \emptyset$, and for each state $s \in Q$, let $\mathit{leads\text{-}to\text{-}cycle}[s] = \mathbf{F}$;
2. If $\mathit{explore}(s^{(0)}, 0) = \mathbf{F}$, then the set represented by $\mathcal{A}$ is infinite. Otherwise, the automaton $\mathcal{A}'$ is given by $(\Sigma, Q, s^{(0)}, \Delta, F)$.

Subroutine $\mathit{explore}(s, k)$:

1. Let $\mathit{visited} = \mathit{visited} \cup \{s\}$, and let $\mathit{history}[k] = s$;
2. For each $(s', d, s'') \in \Delta$ such that $s' = s$:
   - If $s'' \notin \mathit{visited}$, then
     (a) If $\mathit{explore}(s'', k+1) = \mathbf{F}$ then return $\mathbf{F}$;
     (b) If $\mathit{leads\text{-}to\text{-}cycle}[s'']$ then let $\mathit{leads\text{-}to\text{-}cycle}[s] = \mathbf{T}$;
   - If $(\exists i < k)(\mathit{history}[i] = s'')$, then
     (a) If $\mathit{leads\text{-}to\text{-}cycle}[s]$ then return $\mathbf{F}$;
     (b) Let $\mathit{leads\text{-}to\text{-}cycle}[s] = \mathbf{T}$, and remove (s', d, s'') from Δ;
   - If $s'' \in \mathit{visited}$ and $(\forall i < k)(\mathit{history}[i] \neq s'')$, then
     (a) If $\mathit{leads\text{-}to\text{-}cycle}[s'']$ then return $\mathbf{F}$;
3. Return $\mathbf{T}$.