On the Construction of Automata from Linear Arithmetic Constraints*

Pierre Wolper and Bernard Boigelot

Université de Liège, Institut Montefiore, B28, 4000 Liège Sart-Tilman, Belgium.
Email : {pw,boigelot}@montefiore.ulg.ac.be

Abstract. This paper presents an overview of algorithms for constructing automata from linear arithmetic constraints. It identifies one case in which the special structure of the automata that are constructed allows a linear-time determinization procedure to be used. Furthermore, it shows through theoretical analysis and experiments that the special structure of the constructed automata does, in quite a general way, render the usual upper bounds on automata operations vastly overpessimistic.

1 Introduction

Model checking [CES86, QS81, VW86] is a now widespread technique for verifying temporal properties of reactive programs. There are several ways to develop the theory of model checking, a particularly attractive one being through the construction of automata from temporal logic formulas [VW86, BVW94]. As a result, there has been a fair amount of interest in the construction of automata from temporal logical formulas, the history of which is actually fairly interesting.

The starting point is clearly the work of Büchi on the decidability of the first and second-order monadic theories of one successor [Büc62]. These decidability results were obtained through a translation to infinite-word automata, for which Büchi had to prove a very nontrivial complementation lemma. The translation is nonelementary, but this is the best that can be done. It is quite obvious that linear-time temporal logic can be translated to the first-order theory of one successor and hence to infinite-word automata. From a logician's point of view, this could be seen as settling the question, but an interest in using temporal logic for computer science applications, in particular program synthesis [MW84, EC82] triggered a second look at the problem. Indeed, it was quite obvious that a nonelementary construction was not necessary to build an automaton from a temporal logic formula; it could be done within a single exponential by a direct construction [WVS83, VW94]. As originally presented, this construction was worst and best case exponential. Though it was fairly clear that it could be modified to operate more effectively on many instances, nothing was written about this, probably because the topic was thought to be rather trivial and had no bearing on general complexity results.

* This research was partially funded by a grant of the "Communauté française de Belgique - Direction de la recherche scientifique - Actions de recherche concertées".

Nevertheless, the idea that doing model checking through the construction of automata was taken seriously, at least by some, and attempts were made to incorporate automata-theoretic model checking into tools, notably into SPIN [Hol91, Hol97]. Of course, this required an effective implementation of the logic to automaton translation algorithm and the pragmatics of doing this are not entirely obvious. A description of such an implementation was given in [GPVW95] and "improved" algorithms have been proposed since [DGV99]. Note that there are some questions about how to measure such "improvements" since the worst-case complexity of the algorithms stays the same. Nevertheless, experiments show that, for the temporal logic formulas most frequently used in verification, the automata can be kept quite small. Thus, even though it is an intrinsically exponential process, building an automaton from a temporal logic formula appears to be perfectly feasible in practice. What is surprising is that it took quite a long time for the details of a usable algorithmic solution to be developed and codified.

Since building automata from temporal logic formulas turns out to be feasible, one might wonder if the same approach could work for other logics. This has been tried for the second-order monadic logic of one successor (*S1S*) in the MONA tool [HJJ+95]. Here, one is confronted with nonelementary complexity, but careful algorithm selection and coding as well as the fact that the practically useful formulas are not arbitrary make the tool unquestionably usable. Motivated by the need to represent sets of integer vectors in the context of the verification of infinite-state systems [BW94], an automata-based approach is being developed for linear integer (Presburger) arithmetic [WB95, Boi98]. The idea that Presburger arithmetic formulas can be represented by automata goes back at least to Büchi [Büc60], and has lead to nice characterization results for the finite-state representable sets of integer vectors [Cob69, Sem77, BHMV94]. The attractiveness of the approach is not so much for single-shot arithmetic decision problems for which more traditional decision procedures perform well [Pug92], but for situations in which represented sets are repeatedly manipulated and compared, as is necessary in verification. Indeed, minimized deterministic finite automata are a convenient normal form for arithmetic formulas, in a way similar to BDDs [Bry92] being a normal form for Boolean formulas.

Nevertheless, attempts to make a pragmatic use of automata representing arithmetic formulas are fairly recent [WB95, BC96] and one now needs to delve into the details of the automata constructions. Indeed, a straightforward approach to building the automata is quite unworkable and a crude complexity analysis leads only to a nonelementary upper bound, which is unsatisfactory since Presburger arithmetic is know to be decidable in double exponential space. Fortunately, one can do better. In [WB95] it was suggested to use concurrent automata as a representation. This indeed reduces the size of the automata, but pushes up the complexity of manipulating them. An important step was made in [BC96] where it was showed that there is a simple construction for obtaining a deterministic automaton corresponding to an equation or an inequation. That paper even goes further and claims that a triple exponential deterministic automaton can be built for an arbitrary Presburger formula. Unfortunately, though the result itself might not be false, the argument used to substantiate this claim is intrinsically incorrect as we will discuss in this paper. In [TRS98] an encouraging experiment with an automaton-based Presburger implementation is described. Finally, the LASH tool [LASH] is a comprehensive implementation of arithmetic through automata.

This paper aims at presenting and improving on the basics of the pragmatics of constructing automata from Presburger formulas. It starts with a detailed exposition of the construction of automata for linear equations and inequations. The fundamental idea of the construction is that of [BC96], which we extend and improve. First, we deal with signed integers using 2's complement notation (see also [BBR97, BRW98]). Second, we aim at obtaining automata for both directions of reading number encodings. For equations, this is not problematic since the constructed automaton is immediately deterministic in both directions. For inequations, the construction of [BC96] gives an automaton that is deterministic in one direction, but nondeterministic in the other. However, we show that the automaton, taken in its nondeterministic direction, has a special structure that allows the use of a linear-time determinization procedure of possibly independent interest. Furthermore, this result shows that at least in this special case, the general exponential upper bound on determinization is vastly pessimistic.

Finally, we turn to the problem of building automata for arbitrary Presburger formulas. Here, the interesting question is whether an unbounded alternation of quantifiers leads or not to a nonelementary blowup in the size of the automaton. This of course can be the case for arbitrary automata, but we show, with the help of a logic-based argument, that it is not the case for the automata obtained from Presburger formulas. We further substantiate this by giving the results of a number of experiments done with the LASH tool.

2 Preliminaries

Presburger arithmetic is the first-order theory of the structure $\langle \mathbf{N}, 0, \leq, + \rangle$, i.e. the natural numbers with the $\leq$ predicate as well as the 0-ary function 0 and the binary function $+$, all interpreted in the standard way. A Presburger formula with free variables thus represents a set of natural number vectors. In what follows, we will also refer to the theory of the related structure $\langle \mathbf{Z}, 0, \leq, + \rangle$, i.e. the additive theory of the integers, as Presburger arithmetic. Context will remove any ambiguity.

When encoded in a base $r \geq 2$, a natural number is a word over the alphabet $\{0, \ldots r-1\}$. A language or set of words thus represents a set of natural numbers. An obvious question to ask then is which sets of natural numbers correspond to the regular languages under this representation. The question was answered by Cobham who showed that the sets representable in at least two relatively prime bases are exactly those definable in Presburger arithmetic [Cob69]. If one limits oneself to a specific base, say base 2, slightly more is representable. Precisely, one can add to Presburger arithmetic the function $V_r(n)$ giving the largest power of the base r dividing its argument n (see [BHMV94]).

Similar results exist for vectors of natural numbers. To encode an n-dimensional vector $\mathbf{x} = (x_1, \ldots, x_n)$, one encodes each of its components in base r. The length of the encoding of the components is then made uniform by adding leading 0s to the shorter components. The result is then viewed as a word over the alphabet r^n by considering together the first digits of all the vector components, then the second digits, and so on.

Example 1. The vector $(4, 3)$ is encoded in binary by $(100, 011)$, which is viewed as the word $(1, 0)(0, 1)(0, 1)$ over the alphabet 2^2.

Cobham's result on the sets representable by regular languages was extended to natural number vectors by Semenov [Sem77].

In many situations, it is useful to deal with integers rather than with natural numbers. There are several ways to extend the encoding we just introduced to integers. An obvious one is to add a sign bit, but this leads to the need to constantly distinguish the cases of positive and negative numbers. If one works in base 2, which will be our choice from now on, things can be made more uniform, exactly as is done in computer arithmetic, by using 2's complement notation as proposed in [WB95, BRW98, Boi98].

In this notation, a number $b_k b_{k-1} \ldots b_1 b_0$ of length $k+1$ written in base 2 is interpreted as $-b_k 2^k + \sum_{0 \leq i \leq k-1} b_i 2^i$. It is thus positive if b_k is 0 and negative if this bit is 1. There is one slight difficulty that comes from the fact that there is no bound on the size of the integers we consider and that thus we are dealing with variable-length encodings of integers, as opposed to the fixed length usually used in computer arithmetic. This is not problematic if we require that the leading bit of a number is always a sign bit, i.e. it is 0 is the number is positive and 1 if the number is negative[2]. Indeed, there is then no ambiguity on the interpretation of the first bit of a number and repeating the sign bit, whether it is 0 or 1, has no incidence on the value of the number interpreted according to 2's complement's rule since $-2^k + 2^{k-1} = -2^{k-1}$. We can thus still easily make the lengths of the encodings of the components of a vector equal.

Example 2. The vector $(-2, 12)$ can be encoded as $(11110, 01100)$ or as the word $(1, 0)(1, 1)(1, 1)(1, 0)(0, 0)$.

Our goal here is to use finite automata to represent Presburger definable sets of integers. The advantages of this representation are that it is easy to compute with and that it makes questions about the represented sets, for instance nonemptiness, easy to decide. Furthermore, by using minimal deterministic automata, one even obtains a convenient normal form for Presburger definable sets of integer vectors. We will thus consider the problem of building automata corresponding to Presburger formulas. There are however two questions we have to deal with before doing so.

[2] More formally, this means that to represent an integer x, we use a number of bits $k > 0$ large enough to satisfy $-2^{k-1} \leq x < 2^{k-1}$.

Since sign bits can be repeated any number of times, an integer vector has an infinite number of representations. The question then is, which representations should the automata accept. It turns out that the most convenient answer is *all valid representations*, a representation of a vector being valid if its length is sufficient to allow its largest magnitude component to start with a sign bit. Indeed, representing an integer vector by all its encodings allows the Boolean operations on sets of vectors to correspond exactly with the matching language operation on the encodings. The same is unfortunately not true of projection, which is the automaton operation that allows us to handle existential quantification. Indeed, if for example one projects out the largest component of a vector by using language projection on the encodings, one can be left with an automaton that accepts only encodings beyond an unnecessarily long minimum inherited from the component that was eliminated. This problem can nevertheless be solved by using a specific projection operation that allows skipping the repetition of the initial symbol of a word.

The second question is whether our automata will read encodings starting with the most significant or with the least significant bit. One can see advantages to using either directions, and the constructions we give allow automata to be built for either direction. However, our default choice, and the one we will use in examples, is to start with the most significant bit, this order often making the search for a vector accepted by an automaton more effective.

3 Building Automata for Presburger Formulas

We now turn to the problem of building an automaton accepting all encodings of the elements of a set defined by a Presburger formula. We could begin with a construction of automata for addition, equality and inequality, but there are interesting constructions that can deal directly with linear equations and inequations. We thus start with these.

3.1 Building Automata for Equations

The construction we present here is in essentially the one given in [BC96] adapted to handle negative numbers represented using 2's complement, as well as to reading numbers starting with the most significant bit first. The construction is based on the following simple observation. Consider a representation of a vector $\mathbf{x} = (x_1, \ldots, x_n)$ that is k bits long, and imagine adding the bits[3] $(b_1, \ldots, b_n) = \mathbf{b}$ respectively to the encodings of $(x_1, \ldots, x_n)$. The value $\mathbf{x}'$ of the $(k+1)$-bit long encoding thus obtained is given by $\mathbf{x}' = 2\mathbf{x} + \mathbf{b}$ where addition is component-wise. This rule holds for every bit-tuple added except for the

[3] Note that since there is a unique integer vector corresponding to an encoding, we will quite liberally talk about the vector defined by an encoding and, when appropriate use vector notation for encodings. In particular, we will always write elements $(b_1, \ldots, b_n)$ of 2^n as bit vectors $\mathbf{b}$.

first one, in which 1s have to be interpreted as -1. Thus, the value of a one bit long vector $(b_1, \ldots, b_n) = \mathbf{b}$ is simply $-\mathbf{b}$.

Given this, it is very simple to construct an automaton for a linear equation $a_1x_1 + \cdots + a_nx_n = c$ which we write $\mathbf{a}.\mathbf{x} = c$. Indeed, the idea is to keep track of the value of the left-hand side of the equation as successive bits are read. Thus, except for a special initial state, each state of the automaton corresponds to an integer that represents the current value of the left-hand side. From a state corresponding to an integer $\gamma = \mathbf{a}.\mathbf{x}$ for the vector $\mathbf{x}$ that has been read so far, there is a single transition for each bit vector $\mathbf{b}$ leading to the state $\gamma' = \mathbf{a}.(2\mathbf{x} + \mathbf{b}) = 2\mathbf{a}.\mathbf{x} + \mathbf{a}.\mathbf{b} = 2\gamma + \mathbf{a}.\mathbf{b}$. From the special initial state, the transition labeled $\mathbf{b}$ simply leads to the state $\mathbf{a}.(-\mathbf{b})$. The only accepting state is the one whose value is c. Formally, the automaton corresponding to an n-variable equation $\mathbf{a}.\mathbf{x} = c$ is $A = (S, 2^n, \delta, s_i, c)$ where

- $S = \mathbf{Z} \cup \{s_i\}$, i.e. the states are the integers plus a special state s_i;
- the alphabet 2^n is the set of n-bit vectors;
- the transition function δ is defined by
  - $\delta(s_i, \mathbf{b}) = -\mathbf{a}.\mathbf{b}$ and
  - $\delta(\gamma, \mathbf{b}) = 2\gamma + \mathbf{a}.\mathbf{b}$, for $\gamma \neq s_i$;
- the initial state is the special state s_i;
- the only accepting state is c, the value of the right-hand side of the equation.

As defined, the automaton is infinite, but there are only a finite number of states from which the accepting state is reachable. Indeed, if $\|\mathbf{a}\|_1$ represents the norm of a vector $\mathbf{a} = (a_1, \ldots, a_n)$ defined by $\|\mathbf{a}\|_1 = \sum_{i=1}^{n} |a_i|$, we have that from any state γ such that $|\gamma| > \|\mathbf{a}\|_1$, any transition leads to a state γ' with $|\gamma'| > |\gamma|$. So, if furthermore $|\gamma| > |c|$, c can never be reached from such a state. Hence, all states γ such that $|\gamma| > \|\mathbf{a}\|_1$ and $|\gamma| > |c|$ can be collapsed into a single nonaccepting state.

Example 3. The automaton for the equation $x - y = 2$ is given in Figure 3.1. Note that, according to the criterion we have just given, the states beyond the solid line cannot be reached from the accepting state and thus can be collapsed into a single nonaccepting state. Furthermore, looking more carefully at this particular automaton, one sees that the states to be collapsed can in fact include those beyond the dotted line.

The rule we have given for identifying unnecessary states is only approximative. It can be refined, but a more effective approach of identifying the necessary states is actually to construct the automaton backwards, starting from the accepting state. If this construction is limited to reachable states, only necessary states will be constructed. The exact construction is given in Figure 2.

When limited to its reachable states, the automaton obtained by this construction is exactly the useful part of the automaton given by the forward construction. One can complete it by directing all missing transitions to a single nonaccepting sink state. It is deterministic since it is a part of the forward automaton we constructed initially and furthermore, it is minimal, since the sets

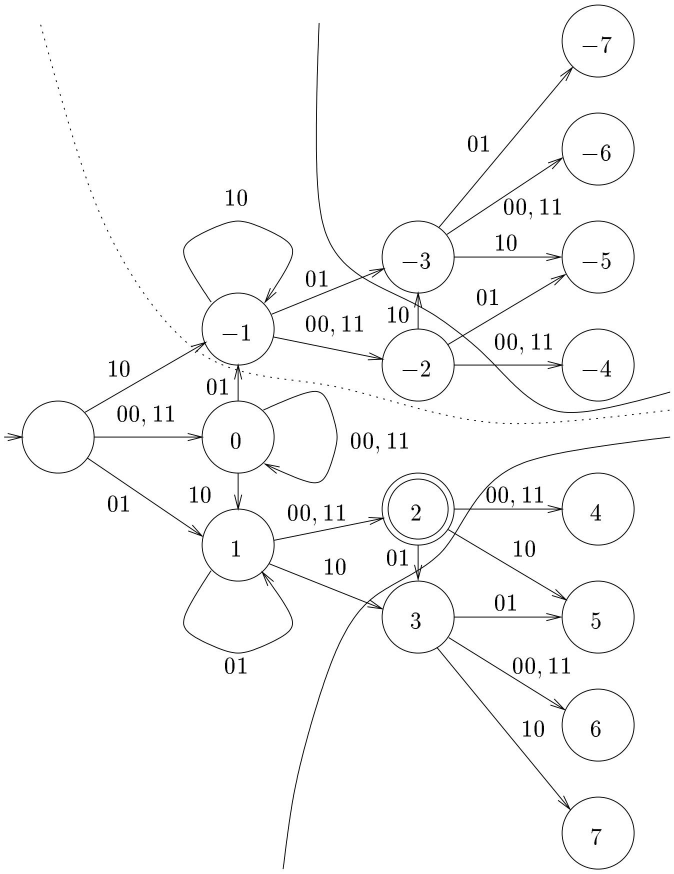

Fig. 1. The automaton for $x - y = 2$.

1. Create a table H of automata states and a list L of "active" states. Both are initialized to $\{c\}$.
2. Repeat the following step until $L = \emptyset$.
3. Remove a state γ from L, and for every $\mathbf{b} \in 2^n$:
   - If $\gamma_o = (\gamma - \mathbf{a}.\mathbf{b})/2$ is an integer, then
     - If γ_o is not in H, add it to H and L;
     - Add a transition labeled $\mathbf{b}$ from γ_o to γ;
   - If $\gamma = -\mathbf{a}.\mathbf{b}$, then
     - Add a transition labeled by $\mathbf{b}$ from the initial state s_i to γ.

Fig. 2. The automaton construction algorithm for an equation.

of words accepted from two distinct states cannot be identical. Indeed, the automaton is also deterministic when going backwards, except for transitions from the initial state, which is not a problem since the language accepted from the initial state is never equal to the one accepted from any other state. How many states does the automaton have? Clearly, any state (except for the initial one) is obtained from c by repeatedly applying the transformation $T(\gamma) = (\gamma - \mathbf{a}.\mathbf{b})/2$. The states obtained by applying this transformation i times, i.e. those in $T^i(c)$ are in the range

$$[\frac{c}{2^i} \pm \sum_{j=1}^{i} \frac{\|\mathbf{a}\|_1}{2^j}]$$

which is always included in

$$[\frac{c}{2^i} \pm \|\mathbf{a}\|_1]. \tag{1}$$

Equation (1), implies that for $i > \log_2 c$, the range reduces to $[-\|\mathbf{a}\|_1, \|\mathbf{a}\|_1]$. Thus the states can be found in at most $\log_2 c + 1$ ranges each of size bounded by $2\|\mathbf{a}\|_1 + 1$. The total number of constructed states is thus $O(\log_2 c \times \|\mathbf{a}\|_1)$, which is only logarithmic in the value of the additive constant c and hence linear in its number of bits.

3.2 Building Automata for Inequations

Consider now an inequation $\mathbf{a}.\mathbf{x} \leq c$. Note that since we are dealing with integers, a strict inequation $\mathbf{a}.\mathbf{x} < c$ is equivalent to the nonstrict inequation $\mathbf{a}.\mathbf{x} \leq c - 1$. The forward construction we gave in the previous section can still be used to build an automaton for the inequation, the only difference being that now the set of accepting states is the set $F = \{\gamma \mid \gamma \leq c\}$. Again, the automaton can be limited to a finite number of states. Indeed, starting with a positive γ such that $\gamma > \|\mathbf{a}\|_1$, all transitions will lead to a $\gamma' > \gamma$ and hence if $\gamma > c$, the inequation will never be satisfied. Similarly, if γ is negative and $-\gamma > \|\mathbf{a}\|_1$, all transitions will always lead to a $\gamma' < \gamma$ and thus if $\gamma \leq c$, the inequation is satisfied.

Again, the analysis above is somewhat coarse and a backwards construction can yield an automaton with less states. However, we have to take into account the fact that we are dealing with an inequation and not an equation, which leads us to construct an automaton somewhat different from the forward automaton. The main point is that, when computing the transitions leading to a state γ, we can no longer dismiss transitions for which $\gamma_o = (\gamma - \mathbf{a}.\mathbf{b})/2$ is not an integer. Indeed, interpreting the fact that a state γ is reached to mean that the inequation $\mathbf{a}.\mathbf{x} \leq \gamma$ is satisfied by the word $\mathbf{x}$ read so far, the condition that has to be satisfied in a state γ_o from which γ is reached by a $\mathbf{b}$ transition is $\gamma_o \leq (\gamma - \mathbf{a}.\mathbf{b})/2$. An infinite number of states satisfy this condition, but it is sufficient to keep the largest since it corresponds to the weakest condition. Thus, as origin of a $\mathbf{b}$ transition to a state γ, we choose $\gamma_o = \lfloor(\gamma - \mathbf{a}.\mathbf{b})/2\rfloor$. Finally, we have to add the possibility of transitions originating in the initial state. Thus, if $-\mathbf{a}.\mathbf{b} \leq \gamma$, we also add a $\mathbf{b}$ transition from the initial state to γ.

1. Create a table H of automata states and a list L of "active" states. Both are initialized to $\{c\}$;
2. Repeat the following step until $L = \emptyset$:
3. Remove a state γ from L, and for every $\mathbf{b} \in 2^n$:
   - Let $\gamma_o = \lfloor(\gamma - \mathbf{a}.\mathbf{b})/2\rfloor$, then
     - If γ_o is not in H, add it to H and L;
     - Add a transition labeled $\mathbf{b}$ from γ_o to γ;
   - If $-\mathbf{a}.\mathbf{b} \leq \gamma$, then
     - Add a transition labeled by $\mathbf{b}$ from the initial state s_i to γ.

Fig. 3. The automaton construction algorithm for an inequation

The exact construction of the automaton is given in Figure 3, the initial state being s_i and the accepting state being c.

As opposed to the case of equations, the automaton we have just built is quite different from our initial forward automaton and is no longer deterministic. Indeed, clearly transitions from the initial state are not deterministic and, furthermore, $\lfloor(\gamma - \mathbf{a}.\mathbf{b})/2\rfloor$ can be the same for two different values of γ, just think of $\gamma = 2$ and $\gamma = 3$ with $\mathbf{b} = \mathbf{0}$. The bound on the number of states we derived for the case equations still holds, but for a nondeterministic automaton. If a deterministic automaton is desired, one is now faced with a potentially exponential determinization cost. However, it would be quite surprising that the automaton for an inequation be so much bigger than the automaton for the corresponding equation. We show that this is not case since the automaton we have constructed has a special structure that allows it to be determinized without increasing its number of states.

The intuition behind the efficient determinization procedure is the following. Suppose that from a state γ, one has two $\mathbf{b}$ transitions leading respectively to states γ_1 and γ_2. One obviously has either $\gamma_1 < \gamma_2$ or $\gamma_2 < \gamma_1$ and one can assume without loss of generality that the former holds. If one reads being in a state γ as meaning that the inequation $\mathbf{a}.\mathbf{x} \leq \gamma$ is satisfied by what has been read so far, it is immediate that any $\mathbf{x}$ that satisfies $\mathbf{a}.\mathbf{x} \leq \gamma_1$ also satisfies $\mathbf{a}.\mathbf{x} \leq \gamma_2$. Hence only the stronger of the two conditions, i.e. $\mathbf{a}.\mathbf{x} \leq \gamma_1$ needs to be remembered in order to know if the word being read will end up being accepted, and the transition to the state γ_2 can be dropped. We now formalize this intuition.

Definition 1. Given a nondeterministic finite automaton $A = (S, \Sigma, \delta, s_0, F)$, let A_s be the automaton $A = (S, \Sigma, \delta, s, F)$, i.e. A where the initial state is s. The automaton A is then said to be *ordered* if there is an explicitly given, i.e. constant-time decidable, strict total order $\prec$ on its set S (possibly excluding the initial state if no transitions lead to it) of states and if for any pair of states satisfying $s_1 \prec s_2$, we have that $L(A_{s_1}) \subset L(A_{s_2})$.

Ordered automata can be determinized efficiently.

Lemma 2. *A nondeterministic ordered finite automaton can be determinized in linear time.*

Proof. Let $A = (S, \Sigma, \delta, s_0, F)$ be an ordered nondeterministic finite automaton, i.e its transition function is of the type $\delta : S \times \Sigma \to 2^S$. The corresponding deterministic automaton is $A' = (S, \Sigma, \delta', s_0, F)$, all components of which are identical to those of A, except for $\delta' : S \times \Sigma \to S$ which is defined by

$$\delta'(a, s) = \max(\delta(a, s)).$$

Thus, if several identically labeled transitions leave a state, they are replaced by a single transition to the largest of these states in the order defined on S. According to the definition of ordered automata, the language accepted from this largest state includes the language accepted from all smaller states and hence removing the transitions to smaller states does not change the language accepted by the automaton. Also note that if the initial state is not the target of any transition, it can safely be left out of the order. The determinization procedure just amounts to removing transitions and can be easily implemented in linear time. $\square$

We are aiming at applying Lemma 2 to the nondeterministic automata we have constructed for inequations. So we need to check if these automata are ordered. Let us look at the words accepted from a state γ of the automaton A constructed for an inequation $\mathbf{a}.\mathbf{x} \leq c$. These, will all be words w encoding a vector $\mathbf{x}_w$, which suffixed to any word w_0 encoding a vector $\mathbf{x}_{w_0}$ satisfying $\mathbf{a}.\mathbf{x}_{w_0} \leq \gamma$ form a word $w_0 w$ encoding a vector $\mathbf{x}_{w_0 w}$ that satisfies $\mathbf{a}.\mathbf{x}_{w_0 w} \leq c$. Thus all the words w accepted from a state γ are such that for all w_0 satisfying $\mathbf{a}.\mathbf{x}_{w_0} \leq \gamma$ one has

$$\mathbf{a}.\mathbf{x}_{w_0 w} = \mathbf{a}.\mathbf{x}_{w_0} 2^{length(w)} + \mathbf{a}.\mathbf{x}_w \leq c$$

and hence, since the above holds for any w_0 such that $\mathbf{a}.\mathbf{x}_{w_0} \leq \gamma$, w must satisfy

$$\gamma 2^{length(w)} + \mathbf{a}.\mathbf{x}_w \leq c. \qquad (2)$$

So, one expects that, if $\gamma_1 < \gamma_2$, a word w accepted from γ_2 will also be accepted from γ_1. In other words, one expects that $L(A_{\gamma_2}) \subset L(A_{\gamma_1})$ and that the automaton is ordered with respect to the relation $\prec$ which is the inverse of the numerical order. However, this is not quite so. Indeed, even though all words accepted from a state γ satisfy the relation expressed by Equation (2), it is not the case that all words satisfying Equation (2) are accepted. Fortunately, it is possible to "complete" the automaton we have constructed in such a way that the words accepted from a state γ are exactly those defined by Equation (2), and this can be done without adding states to the automaton.

The completion procedure just adds transitions and accepting states. Given the automaton $A = (S, 2^n, \delta, s_i, c)$ constructed by the algorithm of Figure 3 for an inequation $\mathbf{a}.\mathbf{x} \leq c$, it constructs an automaton $A' = (S, 2^n, \delta', s_i, F')$ as described in Figure 4.

1. The set of accepting states is $F' = \{\gamma \in S \mid \gamma \leq c\}$;
2. For every state γ, and bit vector $\mathbf{b} \in 2^n$, $\delta'(\gamma, \mathbf{b}) = \delta(\gamma, \mathbf{b}) \cup \{\gamma' \in S \mid \gamma' \geq 2\gamma + \mathbf{a}.\mathbf{b}\}$.

Fig. 4. The completion algorithm.

The completion algorithm can add a number of transitions that is quadratic in the number of states and hence can require quadratic time. We will see how this can be improved, but first let us prove that the completion algorithm does produce an ordered automaton.

Lemma 3. *The completion algorithm of Figure 4 produces an ordered automaton that accepts the same language as the original automaton.*

Proof. The order with respect to which the completed automaton is ordered is the inverse of the numerical order. We thus have to prove that if $\gamma_1 < \gamma_2$ then $L(A_{\gamma_2}) \subset L(A_{\gamma_1})$. This is done by showing that the set of words accepted from any state γ is exactly the one satisfying the relation given in Equation (2), which at the same time shows that the language accepted by the completed automaton is unchanged since the original automaton already accepted all solutions of the inequation.

To show that any word satisfying Equation (2) in a state γ is accepted from that state, we proceed by induction on the length of words. For the induction to go through, we strengthen the property we are proving with the fact that for any word w of length k, the state $\gamma_w^{\max}$ which is the largest γ such that $\gamma 2^k + \mathbf{a}.\mathbf{x}_w \leq c$ is in S. If the word is of length 0 (the empty word ε), it must be accepted iff $\gamma \leq c$. This is guaranteed by the definition of the set of accepting states F'. Furthermore, $\gamma_\varepsilon^{\max}$ is simply c, which by construction is in S.

For the inductive step, let $w = \mathbf{b_1} w_1$, where w_1 is of length $k-1$. By inductive hypothesis, the state $\gamma_{w_1}^{\max}$ is in S. By construction, the state $\lfloor(\gamma_{w_1}^{\max} - \mathbf{a}.\mathbf{b_1})/2\rfloor$ is in S and is the state $\gamma_w^{\max}$. Since w satisfies the relation of Equation (2) in γ, one must have that $\gamma \leq \gamma_w^{\max}$. Hence, the completion procedure adds from γ a transition to $\gamma_{w_1}^{\max}$ given that

$$\gamma_{w_1}^{\max} \geq 2\gamma_w^{\max} + \mathbf{a}.\mathbf{b_1} \geq 2\gamma + \mathbf{a}.\mathbf{b_1}.$$

Hence w is accepted from γ. □

Note that the completion procedure adds transitions that will later be removed by the determinization procedure for the ordered automaton that is obtained. In fact from a state γ and for a bit vector $\mathbf{b}$, the determinization procedure only keeps the transition to the smallest γ' such that $\gamma' \geq \gamma + \mathbf{a}.\mathbf{b}$. Hence, in our completion procedure, we can add transitions according to the following rule:

1. For every state γ, and bit vector $\mathbf{b} \in 2^n$, $\delta'(\gamma, \mathbf{b}) = \delta(\gamma, \mathbf{b}) \cup \min\{\gamma' \in S \mid \gamma' \geq 2\gamma + \mathbf{a}.\mathbf{b}\}$.

This can be done in linear time and we have the following result.

Theorem 4. *The automaton constructed for an inequation by the algorithm of Figure 3 can be determinized in linear time.*

Example 4. The automaton produced by Algorithm 3 for the inequation $x-y \leq 2$ is given in Figure 5, with the elements added by the simplified completion procedure in boldface, and the transitions deleted by the determinization procedure underlined.

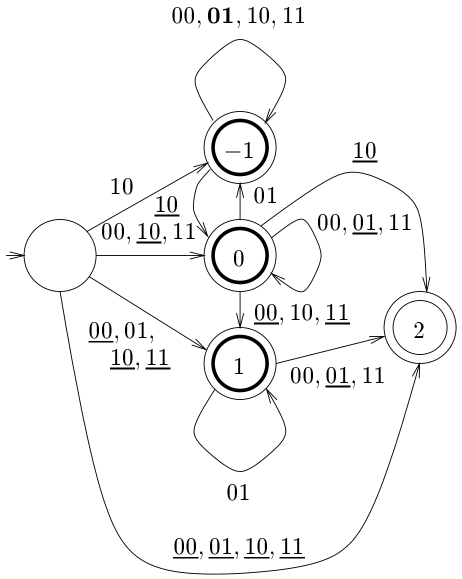

Fig. 5. The automaton for $x - y \leq 2$.

3.3 Building automata for arbitrary formulas

In an arbitrary Presburger formula, one can always move negations inwards and quantifications outwards. Doing so, one obtains a Boolean combination of linear (in)equations prefixed by a string of quantifiers, i.e. a formula of the form

$$Q_1 x_1 Q_2 x_2 \dots Q_n x_n \phi(x_1, \dots, x_n, y_1, \dots, y_m) \tag{3}$$

where each Q_i is either $\forall$ or $\exists$, ϕ is quantifier free and $y_1, \dots y_m$ are the free variables of the formula. The quantifier-free formula ϕ is a Boolean combination of linear equations and inequations ϕ_i. For each of the ϕ_i, we have seen how to build a deterministic automaton of size $O(2^{c|\phi_i|})$, where $|\phi_i|$ is the number of symbols needed to represent the (in)equation, coefficients being encoded in a base ≥ 2. The Boolean combination of these (in)equations can thus be represented by a deterministic automaton that is the product of the automata for the (in)equations, the accepting states being defined according to the given Boolean combination. This product is of size $O(\prod_i 2^{c|\phi_i|})$ or $O(2^{c\sum_i |\phi_i|})$, which is equal to $O(2^{c|\phi|})$. The size of this deterministic automaton is thus at most a single exponential in the size of the formula.

To handle quantification, one replaces $\forall$ by $\neg\exists\neg$, and uses projection as the automaton operation corresponding to existential quantification. There is however one slight problem in doing so, which is that the automaton obtained by standard projection does not accept all encodings of the projected set of integer vectors.

Example 5. The automaton for $x = 1 \wedge y = 4$ and the result of projecting out y from this automaton are given in Figure 6. The resulting automaton accepts the encodings of 1, but only those that are of length at least 4. The encodings 01 and 001 are not accepted.

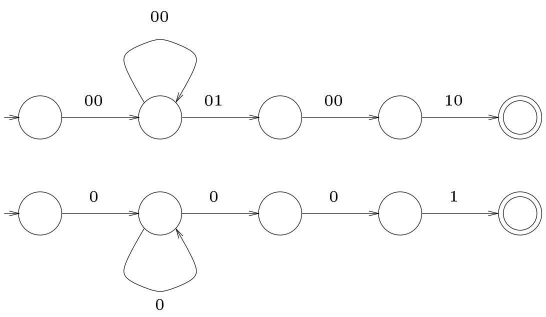

Fig. 6. Automata for $x = 1 \wedge y = 4$ and its projection.

The problem illustrated in Example 5 can be solved by modifying the automaton obtained from projection to ensure that when a word in $\mathbf{b}^+ w$ is accepted the word $\mathbf{b}w$ is also accepted. This is done by including in the set of states reachable from the initial state by $\mathbf{b}$ all states reachable from the initial state by $\mathbf{b}^+$.

The automaton obtained after a projection step is in general nondeterministic and one needs to determinize it in order to apply the complementation needed to handle universal quantification. One thus expects a exponential blowup in the size of the automaton for each quantifier alternation and thus an automaton whose size grows in a nonelementary way. In [BC96] it is argued that this is not the case, and that the size of the automaton is at most 3 exponentials in the size of the formula. Unfortunately, the argument used is false. Indeed, it essentially amounts to translating the string of alternating quantifiers to Boolean transitions, generalizing the translation to nondeterministic transitions done in the handling of projection. The result is thus an alternating automaton of size $O(2^{c|\phi|})$, which can be converted into a deterministic automaton two exponentials larger. The catch is that this implies that the quantifier prefix is handled bit-wise rather than number-wise. Explicitly, when moving from numbers to binary encodings, this implies that rather than translating (3) to

$$Q_1 b_{11} b_{12} \dots b_{1k} Q_2 b_{21} b_{22} \dots b_{2k} \dots Q_n b_{n1} b_{n2} \dots b_{nk} \phi,$$

one translates it to

$$Q_1 b_{11} Q_2 b_{21} \dots Q_n b_{n1} Q_1 b_{12} Q_2 b_{22} \dots Q_n b_{n2} \dots Q_1 b_{1k} Q_2 b_{2k} \dots Q_n b_{nk} \phi,$$

which has, of course, an entirely different meaning.

That the argument used in [BC96] is false does not mean that the size of the automaton for a Presburger formula will grow nonelementarily with respect to the number of quantifier alternations. Indeed, an analysis of the traditional quantifier elimination procedure for Presburger arithmetic [End72] shows the opposite. Looking at this procedure, one notices that the number of basic formulas that are generated stays elementary in the size of the initial formula. Whatever the quantifier prefix of the formula, the quantifier elimination procedure only generates a Boolean combination of this elementary number of formulas. Hence, the formula obtained by the quantifier elimination procedure is elementary and so will be the corresponding automaton.

3.4 Pragmatics

So far, we have tried to present in a fairly detailed way the algorithms used to build automata from Presburger formulas. However, there are still a a substantial number of "improvements" that can be added to what we have described in order to obtain a good implemented system. We discuss here one such important improvement. The reader is certainly aware of the fact that one of the drawbacks of the automata we are constructing is that their alphabet is exponential in the number of variables of the arithmetic formula. Thus, even very simple formulas involving many variables will lead to automata with a huge number of transitions. Fortunately, there is a way around this.

The idea is to sequentialize the reading of the bits of the vector components. That is, rather than reading a bit vector $\mathbf{b} = (b_1, b_2, \dots, b_n)$ as a single entity, one reads b_1, b_2, $\dots$, b_n one at a time in a fixed order. The size of the alphabet is now always 2, whatever the number of components of the integer vectors defined. Of course, the counterpart is that the number of states of the automaton is increased, but this increase can be more moderate than the explosion in the number of transitions that comes from a large number of variables. This can easily be understood by observing that using 2^n as alphabet amounts to representing the transitions from a state as a truth table, whereas sequentializing the reading of the bits corresponds to representing the transitions from a state as a decision diagram for a given bit order. Minimizing the automaton has the effect of minimizing this diagram and one is in fact representing the transitions from a state with a structure that is similar to an OBDD [Bry92]. This technique is used in the LASH package [LASH] as well as in the MONA tool [HJJ+95]. The construction algorithms presented in this paper can easily be adapted to the sequentialized encoding of vectors.

4 Experimental Results

As discussed above, each application of a projection and determinization construction to an automaton representing arithmetic constraints is not going to yield an exponential blowup in the size of the automaton. The question then is, what blowup does in fact occur? To attempt to answer this question, we turned to experiments performed with the help of the LASH tool.

The first experiment consists of applying an existential quantifier to the sets of solutions of random systems of linear inequalities. The results obtained for 100 systems of 8 inequations of dimension 4 with coefficients in the interval $[-5, \ldots, 5]$ are given in Figure 7. This figure depicts the number of states of the quantified automata, which are made deterministic and minimal, with respect to the size of the unquantified automata. Note that all the points fall below the dotted equality line, which means that the number of states always decreases.

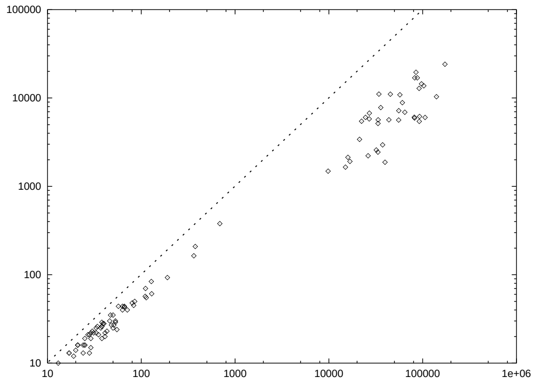

Fig. 7. Effect of quantification over systems of linear inequalities.

A second test consists of repeatedly applying an existential quantification to the automata of the previous experiment, until only a single free variable remains. Figure 8 gives the number of states of the automata obtained during, and as a result of, this process, relative to the size of the automaton obtained prior to the application of the last quantification operation.

Finally, Figure 9 illustrates the effect of applying existential quantification to non-convex sets obtained by joining together the sets of solutions of two random systems of linear inequalities.

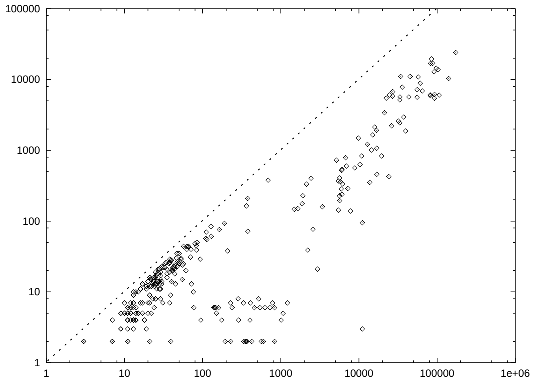

Fig. 8. Effect of repeated quantification over systems of linear inequalities.

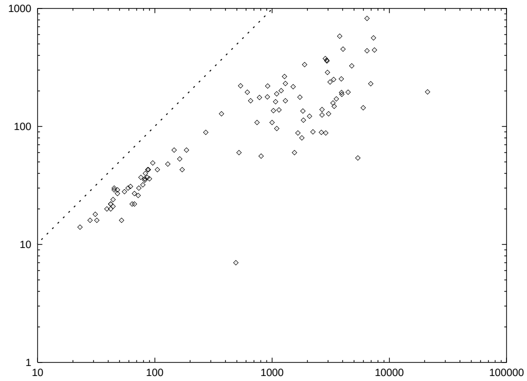

Fig. 9. Effect of quantification over non-convex sets.

It is rather surprising that these experiments show that every projection-determinization step in fact decreases the size of the automaton, whereas an exponential blowup could have been feared. This raises interesting questions, for instance, what exact bound can be proved on the size increase resulting from projecting and determinizing an arithmetic automaton? What structural properties of such automata explain this bound? These are still open questions.

5 Conclusions

There are two sets of conclusions that can be drawn from this paper. The first concerns the use of finite automata as a tool for handling Presburger arithmetic. The initial construction of an automaton from a quantifier-free formula can be exponentially expensive, either as the result of the interaction of many constraints or as a consequence of the presence of large multiplicative constants in formulas. It is easy to construct examples where this explosion occurs, but also to construct examples where things are much tamer. There is however, an important benefit linked to this potentially high cost: the automaton is a structure in which much of the information contained in the formula is explicit. For instance, satisfiability becomes decidable in linear time and inclusion between represented sets is, at worst, quadratic. Furthermore, as shown by our experiments, subsequent manipulation of the automaton need not be very costly. This indicates, that if one needs to repeatedly work with and transform a Presburger formula, as is often the case in verification applications, adopting the automata-based approach might very well be an excellent choice. On the other hand, if one is interested in a one shot satisfiability check, traditional approaches have the edge since building the automaton involves doing substantially more than just checking for the possibility of satisfying the given formula. Of course, only the accumulation of experiments coupled with the fine-tuning of tools will give the final word on the value of the approach.

The second set of conclusions is about computing with automata and the corresponding complexity bounds. Our special determinization procedure for inequation automata as well as our discussion of projection-determinization operations indicate that the general complexity bounds for automata operations do not tell the full story when dealing with automata corresponding to linear constraints. For inequation automata, we were able to identify the structure that explained the absence of blowup while determinizing. For the determinization of the result of a projection operation, our only arguments for the absence of blowup comes from a logic-based analysis of the represented sets. It would, however, be much more satisfactory to explain the absence of blowup in purely automata-theoretic terms, which could lead to more direct and efficient algorithms, just as in the case of inequation automata. But, this remains an open problem.

References

[BBR97] B. Boigelot, L. Bronne, and S. Rassart. An improved reachability analysis method for strongly linear hybrid systems. In *Proc. 9th Int. Conf.on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 167–178, Haifa, June 1997. Springer-Verlag.

[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.

[Boi98] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1998.

[BRW98] Bernard Boigelot, Stéphane Rassart, and Pierre Wolper. On the expressiveness of real and integer arithmetic automata. In *Proc. 25th Colloq. on Automata, Programming, and Languages (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 152–163. Springer-Verlag, July 1998.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift Math. Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[BVW94] Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Workshop*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, California, June 1994. Springer-Verlag.

[BW94] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In *Computer Aided Verification, Proc. 6th Int. Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67, Stanford, California, June 1994. Springer-Verlag.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

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

[DGV99] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Computer-Aided Verification, Proc. 11th Int. Conference*, volume 1633, pages 249–260, July 1999.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[End72] H. B. Enderton. *A mathematical introduction to logic.* Academic Press, 1972.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.

[HJJ+95] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, 1995.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editions, 1991.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

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

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[Pug92] W. Pugh. A practical algorithm for exact array dependency analysis. *Comm. of the ACM*, 35(8):102, August 1992.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th Int'l Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

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

[TRS98] R. K. Ranjan T. R. Shiple, J. H. Kukula. A comparison of Presburger engines for EFSM reachability. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 280–292, Vancouver, July 1998. Springer-Verlag.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

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

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, Tucson, 1983.