Appears in: Proc. of Static Analysis Symposium, Glasgow, Lecture Notes in Computer Science, Springer-Verlag, September, 1995.

An Automata-Theoretic Approach to Presburger Arithmetic Constraints (Extended Abstract)

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 introduces a finite-automata based representation of Presburger arithmetic definable sets of integer vectors. The representation consists of concurrent automata operating on the binary encodings of the elements of the represented sets. This representation has several advantages. First, being automata-based it is operational in nature and hence leads directly to algorithms, for instance all usual operations on sets of integer vectors translate naturally to operations on automata. Second, the use of concurrent automata makes it compact. Third, it is insensitive to the representation size of integers. Our representation can be used whenever arithmetic constraints are needed. To illustrate its possibilities we show that it can handle integer programming optimally, and that it leads to a new original algorithm for the satisfiability of arithmetic inequalities.

1 Introduction

Making a case for the usefulness of arithmetic constraints in computer science is hardly necessary. Indeed, beyond the many common problems that can be reduced to the satisfaction of constraints, constraints have been adopted as a central formalism in a growing number of areas of computer science, for instance constraint databases [KKR90, KSW90], constraint programming languages [Hen89], and the analysis of real-time systems [AH90, ACD90].

The usual view of an arithmetic constraint is as a formula involving constants, variables, arithmetic operators (addition, multiplication), arithmetic predicates (equality, order), and logical operators (boolean, quantification). The constraints that can be expressed depend on the exact choice of constraint language, a typical example being conjunctions of linear inequalities, as used in integer programming. Algorithms for solving constraint problems start from this representation and use a variety of techniques [Sch86].

In this paper, we argue that a more operational description of arithmetic constraints based on finite automata has distinct advantages and leads to interesting algorithmic insights. Precisely, we consider finite automata that accept

* "Aspirant" (Research Assistant) for the National Fund for Scientific Research (Belgium).

the binary representations of integer vectors. Interestingly, results of Cobham and of Semenov show that the sets of integer vectors whose encodings in any base (other than unary) are recognizable by finite automata are exactly the Presburger definable sets [Cob69, Sem77, BHMV94]. More sets are definable in any particular base, for instance powers of 2 in binary, but the results of Cobham and Semenov tell us that Presburger definable sets are those that are robustly representable by automata operating on number encodings.

Also, as will not surprise hardware designers, the automata representing of elementary arithmetic operations and predicates (e.g., addition and order) can be designed to operate on encodings of any length and only require very few states (see Section 3)[2]. Furthermore, by using concurrent automata, we are able to preserve the compactness of the representation for a large class of constraint formulas. For instance, systems of linear inequalities with bounded constants can be translated linearly into concurrent automata.

A clear benefit of representing constraints by automata is that one can directly exploit algorithms from automata theory to obtain algorithms for constraint problems. For instance, obtaining a decision procedure for any first-order constraint language whose atomic formulas are representable by finite automata is immediate : represent the atomic formulas by automata; use the standard constructions on finite automata for conjunction, disjunction, negation and projection; test the result for nonemptiness. This general technique is well known in logic [BHMV94], but we believe that similarly to what has been done for modal and temporal logic [VW86b, VW86a, BVW94, VW94], automata-theoretic techniques for arithmetic can yield algorithmically interesting results for a variety of formalisms.

To make this concrete, we first consider the case of linear inequalities over the integers (integer programming). Checking for the existence of a solution to a system of linear inequalities amounts to checking the nonemptiness of a concurrent automaton, a PSPACE-complete problem. Since, as discussed above, the size of the automaton is polynomial in the size of the constraint system, this gives a PSPACE algorithm for solving the constraint problem. This does not match the well known NP upper bound for this problem [Pap81], but if one remembers that this bound is obtained by showing that linear inequalities always have solutions with polynomially bounded encodings, one notices that rather than checking general nonemptiness, one is checking nonemptiness over words of polynomial length, which can be done in NP.

Finding a somewhat different algorithm for a well known NP problem might appear as a mere curiosity. However, it is interesting to analyze the nature of the algorithm that is obtained. The algorithm is a search through a bounded depth state-space. Since we represent integers with their most significant bit first, a depth-first search through this state space has the nature of a space partitioning algorithm coupled with backtracking. Furthermore, the algorithm

[2] Of course, multiplication cannot be represented, but it is not finite-state in any base and is not Presburger definable. Multiplication by a constant, on the other hand, is not problematic.

is still usable if one extends the class of constraints that are considered, for instance to include periodicity constraints. Interestingly, it is the difficulty of handling periodic sets of integer vectors bounded by linear constraints in the context of a program verification application [BW94] that led us to explore the representation of sets of integers by automata. We were also inspired by the growing use of BDDs [Bry86, Bry92] for representing boolean constraints and by their frequent inadequacy for representing integer constraints.

Since solving a constraint problem reduces to exploring a state-space, this opens up the possibility of using the techniques that have been developed for the latter problem. Possibilities (which we have not tested out) are partial-order techniques [GW93] and symbolic techniques [BCM+92]. Another idea is that if, for some restricted case, the state space can be searched without backtracking then we would obtain an efficient polynomial-time algorithm. Interestingly, this happens for systems of arithmetic inequalities [Klu88]. For this problem we obtain a quadratic algorithm, which matches the complexity of the best known algorithm [SW94] but has a very different nature.

2 Encoding Integer Vectors

We consider integer vectors $\mathbf{v} = (v_1, \ldots, v_n)$ where each $v_i \in \mathbf{Z}$. We encode integers in binary (most significant bit first) using 2's complement for negative numbers. We do not fix the length of encodings, but we do require that the encoding of an integer x such that $-2^{p-1} \leq x < 2^{p-1}$ has at least p bits. Hence, the most significant bit of a positive number will always be 0 and that of a negative number 1. By convention, the empty word encodes 0.

To represent a vector of integers, we encode each of the component integers with an identical number of bits. A vector of integers thus has an infinite number of possible encodings, the shortest of which having the length required by the component with the largest magnitude. All other encodings can be obtained from the shortest one by repeating the initial bit of each component an identical number of times. An encoding of a vector of integers $\mathbf{v} = (v_1, \ldots, v_n)$ can indifferently be viewed either as a tuple $(w_1, \ldots, w_n)$ of words of identical length over the alphabet $\{0, 1\}$, or as a single word $\mathbf{w}$ over the alphabet $\{0, 1\}^n$.

3 Automata Accepting Integer Vector Encodings

Since a integer vector has several possible encodings of different length, we have to choose which of these the automata we define will recognize. A natural choice would be to accept the shortest possible encoding, another to accept all encodings. However, both these choices make some operations on our automata somewhat more complicated. So, we make a third choice for which all operations, except complementation (which in any case is costly) are simple. The requirement is that, if an automaton accepts some encoding of a vector, there is some length ℓ such that it accepts all encodings of that vector that are longer than ℓ.

Automata accepting the encodings of integer vectors could simply be classical finite-state automata over the alphabet $\{0,1\}^n$. However, with this representation, the size of the automata would quickly blow up. We thus choose a more powerful type of automata, namely a particular kind of concurrent automata.

Our *Concurrent Number Automata* (*CNA*) are an adaptation of the automata used to model concurrent systems where components synchronize on like-labeled transitions and interleave on others [Mil89]. A concurrent number automaton recognizing encodings of integer vectors with n elements (a n-*CNA*) has n *number component* (one per element of the vector), plus an arbitrary number (possibly 0) of *synchronization components*. Each component is defined by a *synchronization alphabet* and by a finite-state automaton. The transitions of the automaton of a component are labeled by a subset of its synchronization alphabet (*synchronization transitions*), or by a bit and a subset of its synchronization alphabet (*bit transitions* of number components). The synchronization rule is that a transition with a label containing an element a of the synchronization alphabet must synchronize with a transition containing a in its label for each of the components whose synchronization alphabet contains a. Furthermore, bits are read synchronously, meaning that bits from each of the n integers of the vector are always read simultaneously.

Formally, an n-*Concurrent Number Automaton* (n-*CNA*) is a tuple $A = (C_1, \ldots C_n, C_{n+1}, \ldots C_{n+m})$ of $n \geq 1$ *number components* and $m \geq 0$ *synchronization components*.

- A number component C_k is defined by a *synchronization alphabet* Σ_k and a finite automaton $A_k = (2^{\Sigma_k} \cup (\{0,1\} \times 2^{\Sigma_k}), S_k, s_{0k}, T_k, F_k)$ where
  - $2^{\Sigma_k} \cup (\{0,1\} \times 2^{\Sigma_k})$ is the alphabet,
  - S_k is a set of states,
  - s_{0k} is an initial state,
  - $T_k \subseteq S_k \times (2^{\Sigma_k} \cup (\{0,1\} \times 2^{\Sigma_k})) \times S_k$ is a set of transitions, and
  - F_k is a set of accepting states.
- A synchronization component is defined exactly as a number component, except that its alphabet is simply 2^{Σ_k} (no bit may appear in the label of transitions).

To give precise semantics to concurrent number automata, we define the sequential automaton $Seq(A)$ corresponding to a CNA $A = (C_1, \ldots C_n, C_{n+1}, \ldots C_{n+m})$. We have $Seq(A) = (\overline{\Sigma}, \mathbf{S}, \mathbf{s_0}, \mathbf{T}, \mathbf{F})$ where

- $\overline{\Sigma} = 2^{\Sigma} \cup (\{0,1\}^n \times 2^{\Sigma})$ with $\Sigma = \bigcup \Sigma_k$ being the *synchronization alphabet* of A;
- $\mathbf{S} = S_1 \times \cdots \times S_{n+m}$;
- $\mathbf{s}_0 = (s_{01}, \ldots, s_{0(n+m)})$;
- $((s_1, \ldots, s_{(n+m)}), \alpha, (s'_1, \ldots, s'_{(n+m)})) \in \mathbf{T}$ iff either
  - $\alpha \in 2^{\Sigma}$, for all $1 \leq k \leq n+m$ such that $\alpha \cap \Sigma_k \neq \emptyset$ $(s_k, \alpha \cap \Sigma_k, s'_k) \in T_k$, and $s'_k = s_k$ for all other k, or

- $\alpha \in \{0,1\}^n \times 2^{\Sigma}$ (i.e., $\alpha = ([b_1, \ldots b_n], \alpha_s)$), for all $1 \leq k \leq n$ $(s_k, (b_k, \alpha_s \cap \Sigma_k), s'_k) \in T_k$, for all $1 \leq k \leq n+m$ such that $\alpha_s \cap \Sigma_k \neq \emptyset$ $(s_k, \alpha_s \cap \Sigma_k, s'_k) \in T_k$, and $s'_k = s_k$ for all other k;

$\mathbf{F} = F_1 \times \cdots \times F_{n+m}$.

A binary encoding of an integer vector is then accepted by a concurrent number automaton A if it is the projection on $\{0,1\}^n$ of a word accepted by the sequential automaton $Seq(A)$ (elements of 2^{Σ} project to the empty word ϵ). By extension, we say that an integer vector is accepted by a CNA if at least one of its encodings is accepted.

A CNA is *well-formed* if whenever it accepts an encoding of an integer vector, there is some length ℓ such that it accepts all encodings of that vector that are longer than ℓ. In what follows we only consider well-formed CNA, unless stated otherwise. Finally, we say that a CNA A is deterministic if $Seq(A)$ is deterministic' when projected on the alphabet $\{0,1\}^n$.

4 Automata for Elementary Predicates

Well-formed concurrent number automata accepting sets of (the encodings of) integer vectors satisfying elementary predicates are easy to obtain. We show here some typical examples. To represent a CNA, we give the transition graphs of its number and synchronization (when present) components. The initial state of a number component is labeled by its index (position of the corresponding number in the vector), whereas the initial states of synchronization components are unlabeled. The synchronization alphabet of each component is written in square brackets near its initial state. The labels $\{a, b, \ldots\}$ of synchronization transitions are simply written $ab\ldots$, and the labels $(\mathbf{b}, \{a, b, \ldots\})$ of bit transitions are written $\mathbf{b}$ $ab\ldots$.

A well formed CNA accepting $\mathbf{Z}^n$ is presented in Figure 1. For any constant k, one easily obtains a CNA with one component having $O(\log(k))$ states. Figure 2 represents a well-formed CNA accepting the set $\{(x_1, x_2) \mid x_1 \leq x_2\}$. When examining this automaton remember that the most significant bits are read first. A CNA accepting $\{(x_1, x_2, x_3) \mid x_1 + x_2 = x_3\}$ is given in Figure 3. Note that the synchronization between the components of this CNA is such that the corresponding sequential automaton only has 3 reachable states. More importantly, all the CNA for elementary predicates are deterministic (the corresponding sequential automata projected on $\{0,1\}^n$ are deterministic).

5 Operations

We consider operations on sets of integer vectors and study the implementation of these operations by operations on the CNA representing these sets. Given sets VS_1 and VS_2 of vectors of respective arities (number of components per vector) n_1 and n_2, the operations of interest are

- Union $(VS_1 \cup VS_2)$ and intersection $(VS_1 \cap VS_2)$ provided that $n_1 = n_2$;

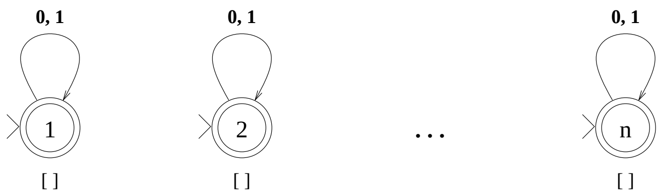

Fig. 1. $A_{\mathbf{Z}^n}$.

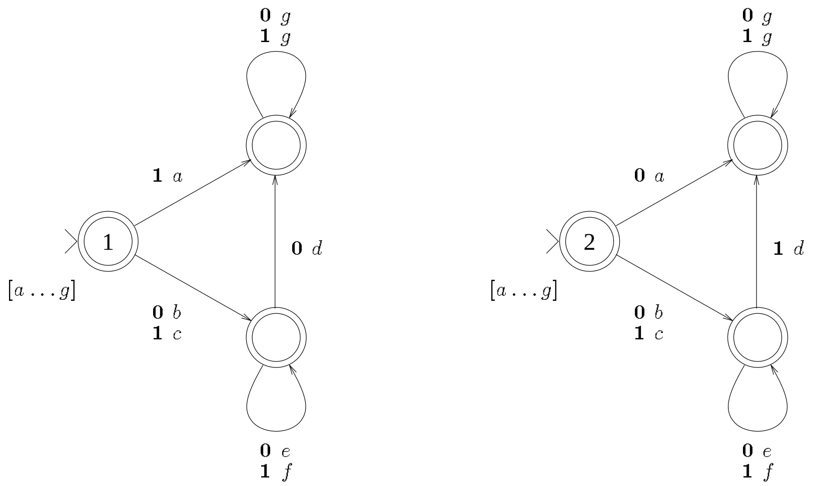

Fig. 2. $A_{\{(x_1,x_2)|x_1 \le x_2\}}$.

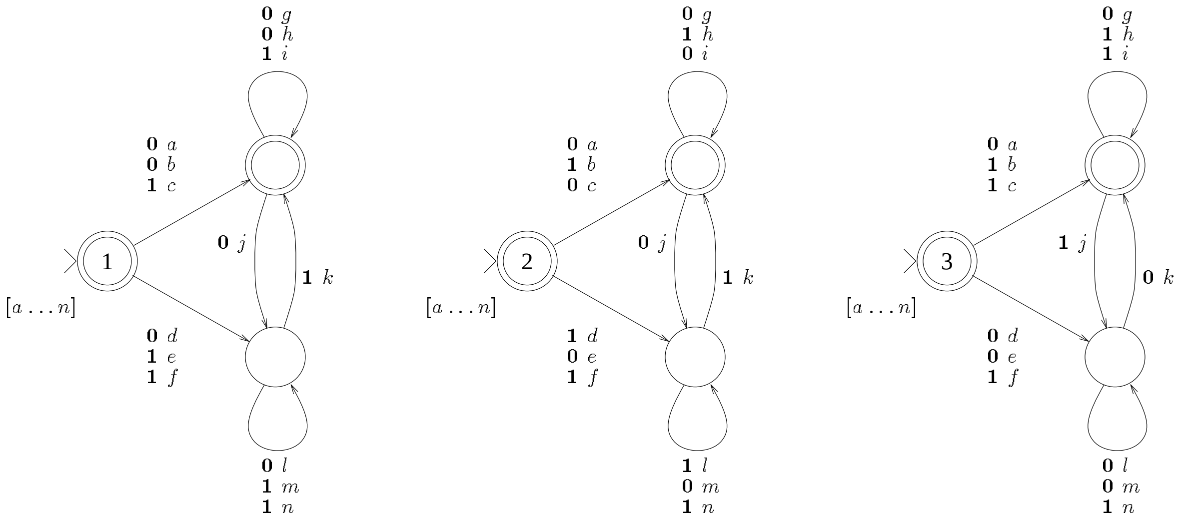

Fig. 3. $A_{\{(x_1,x_2,x_3)|x_1+x_2=x_3)\}}$.

– Complement : $\overline{VS_1} = \{\mathbf{v} \mid \mathbf{v} \notin VS_1\}$;
– Cartesian product : $VS_1 \times VS_2 = \{(\mathbf{v_1}, \mathbf{v_2}) \mid v_1 \in VS_1 \wedge v_2 \in VS_2\}$;
– Reordering : $\pi\, VS_1 = \{(x_{\pi(1)}, \ldots, x_{\pi(n_1)}) \mid (x_1, \ldots, x_{n_1}) \in VS_1\}$, where π is a permutation on $\{1, \ldots, n_1\}$;
– Projection : $\exists x_i\, VS_1 = \{(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n_1}) \mid \exists x_i (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n_1}) \in VS_1\}$.
– Restriction : $\sigma_{i,j}\, VS_1 = \{(x_1, \ldots, x_{n_1}) \mid (x_1, \ldots, x_{n_1}) \in VS_1 \wedge x_i = x_j\}$,

As stated by the theorem below, all operations other than complement can be computed in linear time and furthermore preserve determinism, except unsurprisingly for union and projection. Note that when we compute the size of a CNA, we do not consider the size of the alphabets of the automata defining the components of the CNA. Indeed, these alphabets are of the form $2^{\Sigma_k} \cup \{0,1\} \times 2^{\Sigma_k}$ and hence can be quite large. We only consider the size of Σ_k and, of course, of the elements of the alphabet actually used in transitions of the automaton.

Theorem 1. *Given well-formed concurrent number automata A_1 and A_2 accepting respectively the integer vector sets VS_1 and VS_2, one can construct in time $O(|A_1| + |A_2|)$ well formed CNA of size $O(|A_1| + |A_2|)$ that accept the following sets : $VS_1 \cup VS_2$, $VS_1 \cap VS_2$, $VS_1 \times VS_2$, $\pi\, VS_1$, $\exists x_i\, VS_1$, and $\sigma_{i,j}\, VS_1$ (take the size of the second automaton to be 0 for unary operations). Furthermore, if A_1 and A_2 are deterministic, then so are the CNA accepting $VS_1 \cap VS_2$, $VS_1 \times VS_2$, $\pi\, VS_1$, and $\sigma_{i,j}\, VS_1$.*

Proof : The constructions proving the theorem are quite direct and will be given in a full version of this paper. □

For complementation, the situation is much less favorable. Indeed, to complement, one needs a deterministic sequential automaton, and the fact that the automaton is concurrent costs us one exponential, whereas its nondeterminism can cost us another exponential.

Theorem 2. *Given a well-formed concurrent number automaton A accepting the integer vector set VS, one can construct in time $O(2^{2^{|A|}})$ a well-formed deterministic sequential automaton of size $O(2^{2^{|A|}})$ that accepts the set $\overline{VS}$. Furthermore, if A is deterministic, then the size of the automaton accepting $\overline{VS}$ and the time needed to construct it are $O(2^{|A|})$.*

While this result is rather negative, the following result shows that we cannot to do much better, at least if complementation is done by constructing a sequential deterministic automaton.

Theorem 3. *The equivalence problem (acceptance of the same integer vector set) for concurrent number automata is EXPSPACE-complete.*

Proof : This can be derived from the result in [Rab92] on the complexity of trace equivalence for concurrent programs. □

A final problem of major importance is determining the nonemptiness of a concurrent number automaton. This is a PSPACE-complete problem.

Theorem 4. *The nonemptiness problem for concurrent number automata (deterministic or not) is PSPACE-complete.*

Proof : To show membership in PSPACE, one shows that the sequential automaton corresponding to a CNA can be constructed and explored in a space efficient way (a similar construction can be found in [VW94]). To show PSPACE hardness, one uses a CNA to efficiently encode the computations of a polynomial space Turing machine. □

Note that the PSPACE hardness applies to general concurrent number automata, not necessarily to those obtained from any class of arithmetic formulas. Indeed, we will show in the next Section that we can get better complexity results for the CNA obtained from two common arithmetic problems.

6 Applications

6.1 Integer Programming

The *integer programming* problem consists of determining whether a system of constraints of the form

$$\begin{array}{ccccccc} a_{1,1}x_1 & + & a_{1,2}x_2 & + & \cdots & a_{1,n}x_n & \leq b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & + & \cdots & a_{2,n}x_n & \leq b_2 \\ \vdots & & \vdots & & & \vdots & \vdots \\ a_{p,1}x_1 & + & a_{p,2}x_2 & + & \cdots & a_{p,n}x_n & \leq b_p, \end{array}$$

where all the $a_{i,j}$ are integers, accepts an integer solution $(x_1, x_2, \ldots, x_n)$.

The representation of sets of integer vectors by concurrent number automata leads to an original solution to this problem : one simply constructs the CNA accepting all solutions of the system of equations and tests for its emptiness. The real issue is the complexity of this method. We show here that the size of the CNA is linear in the size of the system equations and that, since integer programming problems admit "small" solutions [Pap81], this leads to an NP algorithm which matches the well known lower and upper bounds for this problem.

Our first result is thus the following.

Theorem 5. *Given an integer programming problem P of size n, it is possible to build in time $O(n)$ a concurrent number automaton of size $O(n)$ that accepts exactly the integer vectors that are solutions of P.*

Proof sketch : The only real difficulty in the construction is to handle the multiplicative constants with only a linear blowup. The size of a constant a is taken to be the size r of its binary encoding $b_{r-1}\ldots b_1 b_0$. The problem is to represent the set $\{(x_1, x_2) \mid x_1 = ax_2\}$ with a CNA of size linear in r. To do this, we simply use the fact that $a = \sum_{0\leq i\leq r-1} 2^i b_i$. This leaves us with 2 problems : representing the sum of r integers and representing multiplication by powers of 2. By introducing auxiliary variables, sums of r terms can be represented by

$r-1$ binary sums. Since each binary sum has a constant size representation, this leads to a linear representation of the sum. For powers of 2, the idea is similar. One can represent $\{(y_0, y_1) \mid y_1 = 2y_0\}$, $\{(y_1, y_2) \mid y_2 = 2y_1\}$, ..., each with a fixed size CNA. Using $r-1$ such CNA's, one can thus represent multiplication by the necessary powers of 2. □.

To solve the integer programming problem, all that needs to be done is to test for nonemptiness of the CNA obtained by the construction of Theorem 5. By Theorem 4 this is a PSPACE-complete problem. However, we have here some additional information on the structure of the CNA : its is obtained from an integer programming problem. Given that if an integer programming problem P has a solution, it has a solution that can be written with a number of bits bounded by a polynomial in $|P|$ [Pap81], we only need to look for a polynomially bounded integer vector representation accepted by the CNA, and this can be done in NP. Our approach thus also yields an NP algorithm for integer programming.

From a more concrete algorithmic point of view, what we propose is to solve the integer programming problem by searching through the state space of the CNA we have constructed. Remembering that we represent most significant bits first, a depth-first exploration of this state space loosely corresponds to a space partitioning approach with backtracking. It is likely that how well this type of algorithm will perform will depend on how strong the constraints on a solution are. It is likely to perform well when solutions are loosely constrained and to get bogged down when the solutions are very constrained (and hence difficult to find).

6.2 Systems of Arithmetic Inequalities

The problem here consists of determining whether a conjunction P of inequalities of the form $x_i \,\theta\, x_j$, where $\theta \in \{=, \neq, <, \leq\}$ admits a solution $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. It is a problem that has been studied, for instance, in the context of database theory [Ull89, Klu88]. Building a linear size CNA for such a problem P is straightforward. What we show here is that testing the emptiness of this CNA can be done very efficiently.

To see this, we first notice that if a system of arithmetic inequalities has a solution, then it has a positive solution. We will take advantage of this by working from now on only with positive numbers, and we will thus no longer use 2's complement representation. We next prove a lemma about the solutions to conjunctions of arithmetic inequalities.

Lemma 6. *If a system of arithmetic inequalities has a solution, then for any strict inequality $x_1 < x_2$ or difference $x_1 \neq x_2$ constraint in the system, there is a solution in which x_1 and x_2 differ on their most significant bit.*

Proof : Take a positive solution of the system and prefix the most significant bit 0 to the value of x_1, and of all variables less than or equal to x_1 in the solution. Similarly, prefix the most significant bit 1 to the value of x_2 and to all variables greater than x_1 in the solution. □

Next, we examine the CNA obtained for the systems of constraints we are considering and examine their behavior on the encodings of positive numbers. The first important observation is that the automata representing the elementary constraints have the property that, each time a bit of the input is read, either no transition is possible, or the automaton stays in the same state, or it moves to an accepting state from which all inputs are accepted. The consequence of this is that whenever one moves within the reachable states of the CNA, as long as the CNA accepts a nonempty language, one can always complete the execution to an accepting one. In other words, one can always find a solution without backtracking. A second observation on the automata is that the only non-accepting states are the initial states of components corresponding to a strict inequality or difference constraint.

This leads to our algorithm which (sketchily) is the following.

1. Select arbitrarily the components corresponding to a strict inequality or difference constraint and that are not in an accepting state (this choice can be arbitrary given Lemma 6).
2. Check if some choice of input bits can force these components to move to an accepting state. This amounts to solving an instance of 2SAT which can be done in linear time [APT79]. If there is no such choice, the system of constraints is unsatisfiable.
3. Repeat steps 1 and 2 until all components are in accepting states.

Given that for a system of size n, there are at most n strict inequality or difference constraints, and that for each the algorithm requires time $O(n)$, the global time complexity of the method is $O(n^2)$. Note that once it has been obtained automata-theoretically, it is possible to rewrite this algorithm to avoid the explicit construction of the automaton.

7 Conclusions

The goal of this paper is to establish that using automata operating on binary encodings of integer vectors is a fruitful approach for representing and manipulating Presburger arithmetic constraints. We have made our case by showing that this use of automata could lead to interesting algorithmic insights. Even though we have not improved on known worst-case complexity bounds, we believe that the algorithms derived from our automata-theoretic approach can have interesting behaviors in practice. Of course in such a well studied area as integer programming, this should be said with caution and needs to be shown by experiments. An advantage of our approach though is that, since it is based on the very general pattern of a state-space search, it does not break down when one moves beyond the exact class of problems for which the algorithm is designed. Extensions can be incorporated gracefully, but of course will usually lead to a performance penalty.

We have introduced a particular representation of constraints (concurrent number automata) that is especially adequate for the applications we considered.

In fact, one can imagine a whole spectrum of representations, ranging from the traditional formulas to more and more restricted types of automata. For instance, deterministic sequential automata might be a very reasonable representation if complementation operations have to be performed frequently. Of course, the deterministic automata could be much larger than other representations. This would amount to paying upfront the cost of the processing that will be required, often a reasonable approach.

Possible applications of our approach are numerous : whenever arithmetic constraints are used, it could be useful to think about them automata-theoretically. There are probably many more algorithms that can be obtained from the automata-theoretic approach, and more work to be done on analyzing and testing those we have obtained. As far as extensions, just one idea would be to handle the rationals with automata on infinite words [Büc62, Tho90].

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, June 1990.

[AH90] R. Alur and T. Henzinger. Real-time logics: complexity and expressiveness. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 390–401, Philadelphia, June 1990.

[APT79] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BHMV94] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. In *Bulletin of the Belgian Mathematical Society*, volume 1, pages 191–238, Mar 1994.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

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

[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] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Workshop*, Stanford, California, June 1994. Lecture Notes in Computer Science, Springer-Verlag. full version available from authors.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Computer Aided Verification, Proc. 6th Int. Workshop*, Stanford, California, June 1994. Lecture Notes in Computer Science, Springer-Verlag.

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

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[Hen89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[KKR90] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Revesz. Constraint query languages. In *Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.

[Klu88] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1), January 1988.

[KSW90] F. Kabanza, J.-M. Stévenne, and P. Wolper. Handling infinite temporal data. In *Proc. of the 9th ACM Symposium on Principles of Database Systems*, pages 392–403, Nashville Tennessee, 1990.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Pap81] C. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28:765–768, Oct 1981.

[Rab92] A. Rabinovich. Checking equivalences between concurrent systems of finite agents. In *International Colloquium on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 696–707. Springer-Verlag, 1992.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* A Wiley-Interscience publication. Wiley, Chichester, New York, 1986.

[Sem77] A. L. Semenov. The presburger nature of predicates that are regular in two number systems. *Siberian Math. J.*, 18(2):289–299, 1977.

[SW94] W. Sun and M. A. Weiss. An improved algorithm for implication testing involving arithmetic inequalities. In *IEEE Transactions on Knowledge and Data Engineering*, volume 6, pages 997–1001, Dec 1994.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science – Volume B: Formal Models and Semantics*, chapter 4, pages 133–191. Elsevier Science Publishers, Amsterdam, 1990.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems – Volume II: The New Technologies.* Computer Science Press, 1989.

[VW86a] M.Y. Vardi and P. 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.

[VW86b] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–21, April 1986.

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

This article was processed using the LaTeX macro package with LLNCS style