Efficient Symbolic Representations for Arithmetic Constraints in Verification *

Constantinos Bartzis and Tevfik Bultan

Department of Computer Science
University of California
Santa Barbara CA 93106, USA
{bar,bultan}@cs.ucsb.edu

Abstract. In this paper we discuss efficient symbolic representations for infinite-state systems specified using linear arithmetic constraints. We give new algorithms for constructing finite automata which represent integer sets that satisfy linear constraints. These automata can represent either signed or unsigned integers and have a lower number of states compared to other similar approaches. We experimentally compare different symbolic representations by using them to verify non-trivial specification examples. In many cases symbolic representations based on our construction algorithms outperform the polyhedral representation used in Omega Library, or the automata representation used in LASH.

1 Introduction

Symbolic representations enable verification of systems with large state spaces which cannot be analyzed using enumerative approaches. Recently, symbolic model checking has been applied to verification of infinite-state systems using symbolic representations that can encode infinite sets [13, 6, 9]. One class of infinite-state systems are systems that can be specified using linear arithmetic formulas on unbounded integer variables. Verification of such systems have many interesting applications such as monitor specifications, mutual exclusion protocols [6, 9], parameterized cache coherence protocols [8], and static analysis of access errors in dynamically allocated memory locations (buffer overflows) [10]. In this paper we present new symbolic representations for linear arithmetic formulas and experimental results on efficiency of different symbolic representations.

There are two basic approaches to symbolic representation of linear arithmetic constraints in verification: 1) *Polyhedral representation*: In this approach linear arithmetic formulas are represented in a disjunctive form where each disjunct corresponds to a convex polyhedron. Each polyhedron corresponds to a conjunction of linear constraints [11, 13, 9]. This approach can be extended to

* This work is supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822.

full Presburger arithmetic by including divisibility constraints (which can be represented as an equality constraint with an existentially quantified variable) [6, 15]. 2) *Automata representation*: An arithmetic constraint on v integer variables can be represented by a v-track automaton that accepts a string if it corresponds to a v-dimensional integer vector (in binary representation) that satisfies the corresponding arithmetic constraint [4, 17]. For both of these symbolic representations one can implement algorithms for intersection, union, complement, existential quantifier elimination operations, and subsumption, emptiness and equivalence tests, and therefore use them in model checking.

In this paper we present new construction algorithms for the automata representation and also experimentally compare these different approaches. We give a new algorithm for constructing a finite automata representation for sets satisfying Presburger arithmetic formulas. The size of the resulting automaton in our construction has the same upper bound as the construction given in [4], however, our construction is also able to handle negative integers. The size of the resulting automaton in our construction is different than the construction given in [17]. We implemented our construction algorithm using the MONA tool [14] and integrated it to a set of tools for infinite-state model checking [18]. We experimented with a large set of examples. To compare the performance of our construction algorithm to other approaches we also integrated the LASH tool [1] which uses the automata construction given in [17], and Omega Library [2] which uses a polyhedral representation to the same set of tools and ran them on the same set of examples.

Our experimental results show that our construction algorithm produces more compact representations than the construction algorithm given in [17]. Also automata representation is more efficient compared to the polyhedral representation used in [2]. The rest of the paper is organized as follows. In Sections 2 and 3 we give our automata construction algorithms for Presburger arithmetic formulas. In Section 4 we summarize the complexity results and discuss the related work. In Section 5 we discuss our implementation and experimental results. Finally, we state our conclusions in Section 6.

2 Finite Automata Representation for Presburger Formulas

In this section we describe our algorithm for constructing a finite automaton that accepts the set of natural number tuples that satisfy a Presburger arithmetic formula on v variables. The same problem has been solved in [4]. Here we give a similar approach that enables us to prove a tighter bound on the size of the resulting automaton and can also be easily generalized to unsigned integers as we will show in Section 3.

We encode numbers using their binary representation. A v-tuple of natural numbers $(n_1, n_2, ..., n_v)$ is encoded as a word over the alphabet $\{0,1\}^v$, where the i_{th} letter in the word is $(b_{i1}, b_{i2}, ..., b_{iv})$ and b_{ij} is the i_{th} least significant bit of number n_j. Given a Presburger formula ϕ, we construct a finite automaton

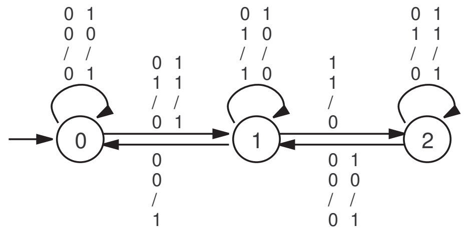


Fig. 1. A Basic State Machine for $x + 2y$

FA(ϕ) that accepts the language L(ϕ) over the alphabet $\{0, 1\}^v$, which contains all the encodings of the natural number tuples that satisfy the formula.

Addition of v ***variables*** $\sum_{i=1}^{v} x_i$ As a basis for all following construction we will use a state machine that performs linear arithmetic over natural numbers, which we will refer to as BSM (for basic state machine) from now on. When adding v variables, all the BSM has to remember is all the possible values for the carry, which in particular are all the integers between 0 and $v - 1$. Thus a BSM with v states is sufficient. At any point, the BSM adds up all the bits of the current letter plus the carry value of the current state, writes the resulting bit to the output, and moves to a new state according to the value of the new carry. The entry state is the one with carry value zero.

Linear functions $\sum_{i=1}^{v} a_i \cdot x_i$ Now each variable is multiplied with a positive (negative) coefficient. The only change to the above addition process is that now the bit of the i_{th} variable is added (subtracted) a_i times. For the moment, we assume that there is no constant term. Obviously the number of possible values of the carry (and thus the number of states) is now equal to $\sum_{i=1}^{v} |a_i|$. This is true if the GCD of all the coefficients is 1. If this is not the case, we can divide all the coefficients with their GCD and round the quotient of the division of the constant term with the GCD properly. An example BSM for calculating $x + 2y$ is shown in Figure 1. The transitions are labeled with the current bit of x and y and also the bit written to the output.

Checking for equality with zero $\sum_{i=1}^{v} x_i = 0$ From now on we describe Finite Automata (FA) based on the BSM. This means that we now characterize the states as accepting and rejecting and there is no output. This FA operates similarly to the BSM , except that after computing every bit, instead of printing it to the output, it compares it against zero. While the comparison is successful (the resulting bit is indeed zero), the FA continues its normal operation. Otherwise it jumps to an extra sink state. The size of this FA is at most one more than the size of the BSM. The accepting state is the entry state (carry is zero) and all the others are rejecting. If we are checking for non-equality, all we need to do is make the accepting states non-accepting and vice-versa.

FA Algorithm for equations $\sum_{i=1}^{v} a_i \cdot x_i = c$

Create a new state labeled $-c$, make it the initial state and mark it as expandable
Create a new state labeled sink, point all transitions from it to itself
Mark it as non-expandable
While there exists an expandable state labeled s do:
    For each transition $b_1 b_2 ... b_v$ from this state do
        If $s' = (\sum_{i=1}^{v} a_i \cdot b_i + s)/2$ is an integer then
            If there is no state labeled s'
                Create a new state labeled s' and mark it as expandable
            Make that state the destination of the transition
        Else
            Make state labeled sink the destination of the transition
    Mark state labeled s as non-expandable
Make state labeled 0 an accepting state
Make all other states rejecting states

Fig. 2. FA Algorithm for equations

Checking inequalities The situation now is simpler. If we are checking for < 0, the FA is identical to the BSM. The accepting states are all the states representing negative carries. All the other states are rejecting. If we are checking for ≥ 0, we take the negation of the above described FA. For the rest of the cases $(> 0, \leq 0)$, we change the signs of all coefficients and then we construct the FA like before.

Constant term So far we assumed no constant term in the equations and inequations or, in other words, the constant term was zero. In the case where the linear combination of the variables is compared against a non-zero integer constant term c, all we have to do is make the state corresponding to a carry value of $-c$ the initial state of the FA, that is, we begin with a carry of $-c$ and by gradually adding the linear combination of the variables, we compare the result against zero, as we did before. If no such state exists, we introduce one. Of course we need to introduce more states corresponding to carry values between $-c$ and the carry value closest to $-c$. Thus the number of states now becomes at most: $S = |\min(-c, \sum_{a_i<0} a_i)| + |\max(-c, \sum_{a_i>0} a_i)|$. Note that this upper bound on the number of states is tighter that the one given in [4], even though the construction algorithms are similar. The algorithms for constructing FA from equations and inequations are given in Figures 2 and refineq-al, respectively.

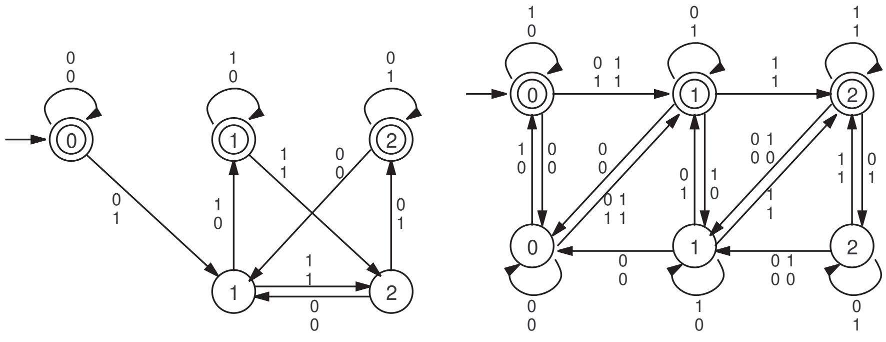
FA Algorithm for inequations $\sum_{i=1}^{v} a_i \cdot x_i < c$

Create a new state labeled $-c$, make it the initial state and mark it as expandable
While there exists an expandable state labeled s do:
    For each transition $b_1 b_2 ... b_v$ from this state do
        $s' = \left\lfloor (\sum_{i=1}^{v} a_i \cdot b_i + s)/2 \right\rfloor$
        If there is no state labeled s'
            Create a new state labeled s' and mark it as expandable
        Make that state the destination of the transition
    Mark state labeled s as non-expandable
Make states with negative labels accepting states
Make all other states rejecting states

Fig. 3. FA Algorithm for inequations

There is an alternative way to cope with the constant term. If $l = \log_2 c$ is the length of the binary representation of the constant c, we can stack $l+1$ BSMs in $l+1$ layers. For equations, any transition from layer $1 \leq i \leq l$, whose output is the same as the i_{th} bit of c goes to the appropriate state of layer $i+1$, if the output is different, it goes to the sink state. Entry state is the entry state of the first BSM. Accepting states are the accepting states of the final BSM. For inequations ($< c$)), all transitions from any layer except the last go to the appropriate state of the next layer and accepting states are those representing a negative carry. The total number of states is at most $(l+1) \cdot \sum_{i=1}^{v} |a_i|$. Note that the two approaches described will result in the same automaton, when minimized. We can choose which one to use depending on the expected upper bound on the number of states.

Presburger formulas Presburger formulas contain Boolean connectives between atomic linear constraints (equations and inequations) and quantifiers. Given two linear constraints and the FA representing them (with n and m states respectively), we can easily construct a new FA that accepts the Boolean AND (OR) between them by computing the intersection (union) of the two FA. It is known that such a FA contains at most $n \cdot m$ states. Similarly we can deal with NOT, by FA complementation. If some variable is existentially quantified, we can compute a non-deterministic FA accepting the projection of the initial FA on the remaining variables and then determinize it. The resulting automaton may not accept *all* satisfying encodings (with any number of leading zeros). We can overcome this by identifying all rejecting states from which the all-zero transition leads to an accepting state, and make them accepting. Universal quantification can be similarly implemented by the use of negation.



(a) $x + 2y = 0$ **(b)** $x + 2y < 0$

Fig. 4. Example automata for all integers: **(a)** equalities and **(b)** inequalities

3 Finite Automata for Linear Constraints on *all* Integers

So far we have described the construction of automata for linear constraints on natural variables. Here we show how to build finite automata which accept linear constraints on all integers, including negative, using 2's complement arithmetic. These new FA are only twice as large as the former ones. The construction, again, is based on the basic state machine which calculates linear formulas. As an example we use the constraint $x + 2y = 0$, the BSM of which is shown in Figure 1. Again we describe automata for constraints with no constant terms. The addition of such terms is done in exactly the same way as in the previous section.

Finite automata for equality with zero The procedure is based on the fact that in order for the FA to accept the encoding of a tuple of numbers, it must also accept the encoding of the same numbers with arbitrarily many sign bits (i.e. the most significant bit of each number repeated arbitrarily many times). This means that the according BSM must write only 0s when reading such an arbitrarily long word. The only way for the BSM to continue writing 0s by repeatedly receiving the same combination of bits is by following a looping transition. The FA contains two clones of each state of the BSM, one accepting and one rejecting. It also contains a rejecting sink state. Looping transitions in the BSM that write 0 go to the according accepting clone. All other transitions that write 0 go to the rejecting clone. All transitions that write 1 in the BSM go to the sink state. Obviously such an automaton has twice as many states and transitions as the BSM, thus its size is at most $2S+1$, where S was defined in the previous section. One can easily verify that this is the minimum possible. The example automaton is shown in Figure 4(a). The sink state and all transitions to it have been omitted.

Finite automata for inequalities Like before we will only describe the < 0 case. All other types of inequalities can be easily and effectively derived from it. Now we want the result of the addition to be negative, thus having an 1 as a sign bit. Unfortunately, the bit most recently written by the BSM is not always the sign bit of the result, due to possible overflow. Thus, a tuple of integers renders a negative result iff by repeating their most significant bits an adequate number of times, the BSM will eventually enter a looping transition that writes 1. Any number greater or equal to the binary logarithm of the sum of the absolute values of the coefficients is adequate. Thats how many extra bits of the result one would need to compute in order to avoid an overflow. Again we create one accepting and one rejecting clone of each state of the BSM and there is no sink state. Looping transitions that write 1 go to the accepting clone and those which write 0 go to the rejecting clone. Any other transition goes to the appropriate accepting clone, iff by repeatedly receiving the same combination of bits the BSM will eventually enter a loop which writes 1. Otherwise it goes to the rejecting clone. Obviously this FA has $2S$ states. See Figure 4(b).

Note that for both equations and inequations, the automata resulting from our construction are deterministic, unlike those described in [17] which require (linear time) determinization. Secondly, once a state has been created, all transitions originating from it can be computed immediately (as opposed to [17]), which is more convenient when transitions are stored using BDDs. Moreover, even though we can prove an upper bound on the number of states different (and in many cases better) than the one given in [17], if we follow the BSM stacking method for dealing with the constant term, described in Section 2 we can achieve the same bound of $O(\log_2 c \cdot \sum_{i=1}^{v} |a_i|)$. Finally, our construction and the one in [17] result in different automata, because the accepted languages are different (one is the reverse of the other).

Alternative FA Here we will describe an alternative FA construction. Alternative FA accepts the same values for the variables as before, but the encoding is different. There is only one track containing the bits of all the variables interleaved. The procedure is similar but now we have to keep all the intermediate results of the addition of the i_{th} bit of the variables, thus we need the appropriate states to represent these results and to tell which variable's bit is coming next. States will be labeled with a variable's name and a value for an intermediate result. For the first variable there are as many states as the different possible values of the carry which are at most $2S$. For the next variable there are at most $|a_1|$ more states and so on. The total number of states in the automaton is $2vS + \sum_{i=1}^{v} (v-i)|a_i|$ which is less than $3vS$. Interestingly this leads us to the conclusion that the size of the automaton is minimized if the variables are ordered in increasing order of the absolute values of their coefficients. The transition graph of this alternative FA is isomorphic to a multi-rooted, multi-terminal BDD that stores the transitions of the default FA described earlier. This is the technique used in MONA. Furthermore, if one unrolls the alternative FA n times, the resulting graph will be isomorphic to a traditional BDD that represents the same formula as the FA but for n-bit variables. Such BDDs are useful in the verification of finite state systems.

4 Complexity Outline and Related Work

The following table summarizes the time complexity of the construction algorithms for linear integer arithmetic constraints, as well as the sizes of the resulting structures. The entries of the tables are worst-case asymptotic estimations. The size of a FA is the number of its states. Like before, v denotes the number of integer variables, s is the sum of the absolute values of the coefficients and c is the constant term.

	FA for non-negative integers	FA for all integers	LASH
Construction Time	$2^v(s+\|c\|)$	$2^{v+1}(s+\|c\|)$	$2^{v+1}\log(\|c\|)s$
Construction Size	$s+\|c\|$	$2(s+\|c\|)$	$2\log(\|c\|)s$

The reader can find similarities between our FA construction algorithms and those proposed in [4] and [17]. Here we briefly explain the advantages of our approach. In [4] only constraints over non-negative integers are considered. We describe FA for constraints over all integers as well, by only doubling the complexity. Also, by using the preprocessing technique described in Section 2 we guarantee FA of minimal size.

On the other hand, in [17] the complexity of $O(2 \cdot \log_2 c \cdot ||a||_1)$, where c is the constant term and $||a||_1$ is the sum of the absolute values of the coefficients, is worse than ours for most practical cases. Note that by slightly modifying our construction we can also achieve that complexity, whenever needed. Moreover in [17] no algorithm is given for constraints on non-negative integers. There are cases where all variables range over the non-negative integers, and adding extra constraints to denote that, would be wasteful. Finally in Section 3 we give a detailed complexity analysis for an alternative automaton that is equivalent to the "sequentialized" FA described in [17].

At last, our transducer-based FA can be composed with FA for constraints beyond Presburger arithmetic, such as "x is a power of two".

In [16] polyhedral and automata representation for arithmetic constraints are compared experimentally for reachability analysis of several concurrent systems. The results show no clear winner. On some problem instances the polyhedral representation is superior, on some others automata representation is. Our experimental setup is more reliable compared to [16]. In [16] boolean variables are mapped to integer variables when polyhedral representation is used. This is an inefficient encoding which gives an unfair advantage to the automata representation. In our experiments boolean variables are not mapped to integers in any representation. Also, our tools perform full CTL model checking including liveness properties instead of just reachability analysis discussed in [16].

5 Implementation and Experiments

Before we explain our implementation and experiments we would like to explain the tools we used. Omega Library is a symbolic manipulator for Presburger arithmetic formulas [2]. Omega Library uses a disjunctive normal form to represent

Presburger arithmetic formulas where each disjunct corresponds to a conjunction of a set of equality, inequality or divisibility constraints (constraints in the form $\sum_{1 \le i \le v} a_i x_i + a_0$ is divisible by c where c is an integer constant). Any Presburger formula can be converted to such a disjunctive normal form. MONA is a tool that translates formulas to finite-state automata and implements decision procedures for the Weak Second-order Theory of One or Two successors [12]. We used MONA's automata package to implement our construction procedures and symbolic operations. We chose this specific package because its internal representation of state transitions (BDDs) makes most of the operations time efficient. Of course we had to make various modifications to meet our needs and also implement functions not included in MONA such as checking for automata equivalence and emptiness. LASH is a toolset for representing infinite sets and exploring infinite state spaces based on finite-state automata [1]. It includes a C library that provides FA construction functions for linear constraints as well as FA manipulation functions. In our experiments we examine their practical efficiency.

We integrated our construction algorithms to an infinite state CTL model checker called Action Language Verifier [5] built on top of the Composite Symbolic Library [18]. The Composite Symbolic Library uses an object-oriented design to combine different symbolic representations [18]. An abstract interface defines the operations used in symbolic verification: Boolean operations, equivalence and subsumption tests, and image computations (for the pre- and post-condition operators). To integrate a new symbolic representation to the Composite Symbolic Library one simply has to implement the abstract interface with specialized operations. Composite Symbolic Library supports a disjunctive composite representation for formulas on integer and boolean variables. A disjunctive composite representation is in the form $\bigvee_{i=1}^{n} \bigwedge_{t \in T} p_{it}$ where p_{it} denotes the formula of type t (which could be integer or boolean) in the ith disjunct, and n and T denote the number of disjuncts and the set of variable types ($T = \{integer, boolean\}$)), respectively. The methods such as intersection, union, complement, satisfiability check, subsumption test, which manipulate composite representations in the above form are implemented in the Composite Symbolic Library by calling the operations on integer and boolean formula representations [18].

We integrated five different symbolic representations to the Composite Symbolic Library. The first three use the disjunctive composite representation used in Composite Symbolic Library to combine formulas on integer and boolean variables. We used the BDD representation for boolean formulas. We implemented three different integer formula representations using LASH [1]) (version V3), Omega [2] (version V2), and our automata construction algorithm (version V1) which uses MONA automata package [14] as an automata manipulator. We also implemented two automata based representations using LASH (version V5) and our construction algorithms (version V4) again built on top of MONA automata package, for both boolean and integer variables without using the disjunctive composite representation. The states of both boolean and integer variables can be represented in an automata, hence one can avoid using the disjunctive composite representation.

We experimented with a large set of examples which are describe below. Each instance is labeled using NAME(number of processes)-(property number). Action Language specifications of all these examples and properties are available at: `http://www.cs.ucsb.edu/~bultan/composite/`
First set of examples are monitor specifications for the sleeping barber problem. We verified three properties for systems with 2, 3, and 4 customer processes and one barber process (BARBER(2,3,4)-(1,2,3)). We also verified the three properties (BARBERP-(1,2,3)) on the parameterized system for arbitrary number of customer processes. BAKERY(2,3,4) and TICKET(2,3,4) are mutual exclusion protocols (for (2,3,4) processes, respectively). We verified both mutual exclusion (BAKERY(2,3,4)-1, TICKET(2,3,4)-1) and starvation-freedom properties for these protocols (BAKERY(2,3,4)-2, TICKET(2,3,4)-2). We analyzed a parameterized cache coherence protocol specification given in [8]. We verified three properties given in [8], however the last property (COHERENCE-4) did not converge. In [8] this property is verified using widening technique. PC(5,10) is a bounded buffer producer consumer system with 5 or 10 control states [18]. RW(4,8,16,32,64) is a monitor specification for readers-writers problem for various number of processes [3]. LIGHTCONTROL and SIS are two reactive software specifications. LIGHTCONTROL is an office light control system specification written in statecharts [5]. SIS is specification of a safety injection system for a nuclear reactor [7] which uses three sensor readings and a majority vote to control water pressure level in the reactor. INSERTIONSORT is a specification from [9] for array bound checking of an implementation of insertion sort algorithm.

The results of our experimental evaluation of different representations for linear integer arithmetic constraints are shown in Table 1. We obtained the experimental results on a SUN ULTRA 10 work station with 768 Mbytes of memory, running SunOs 5.7. For each version of the verifier we recorded the following statistics: 1) Time elapsed during the construction of the symbolic representation of the transition system, shown in the table as CT. 2) Time elapsed during the verification process, shown as VT. It includes the time needed for forward or backward fixpoint computations, however, it excludes the construction time (CT). 3) The maximum amount of memory used by the verifier, shown as Mem. Also for V1, V3, V4 and V5 that use automata as a symbolic representation we recorded the size (number of states) of the automaton representing the transition system, shown as TRS, and the size of the largest automaton computed during the fixpoint computation, shown as MS. As discussed above our automata construction algorithm used in versions V1 and V4 uses MONA automata package. MONA automata package uses BDDs to store the transition relation of the automata. Therefore, to make the comparison with LASH fair, instead of giving the number of automaton states for versions V1 and V4, we give the total number of BDD nodes used in the MONA representation. For the barber and ticket problems we used forward fixpoint computation, while for all other problems backward fixpoint computations were used.

Table 1. Experimental results for performance of different symbolic representations for integer arithmetic constraints. Time measurements appear in seconds and memory measurements in Mbytes. For each problem instance the following values are shown: time elapsed during the construction of the symbolic representation of the transition system (CT), time elapsed during verification process (VT), the maximum amount of memory used by the verifier (Mem). For the automata representations we also give the total number of states in the automata representing the transition relation (TRS) and the total number of states in the automata representing the fixpoint iterate with the maximum size (MS).

Problem Instance	Computed Fixpoints	Disjunctive Composite Representation																						
		V1 - Automata					V2 - Omega			V3 - LASH					V4 - Automata					V5 - LASH				
		CT	VT	Mem	TRS	MS	CT	VT	Mem	CT	VT	Mem	TRS	MS	CT	VT	Mem	TRS	MS	CT	VT	Mem	TRS	MS
BARBERM2-1	F(6)	0.13	0.07	8	184	26	0.22	0.11	7	0.31	0.23	6	198	25	0.05	0.01	0.45	115	40	1.71	0.22	0.36	276	46
BARBERM3-1	F(6)	0.14	0.08	9	241	26	0.2	0.11	7	0.39	0.29	6	257	25	0.07	0.01	0.46	124	42	2.39	0.26	0.4	311	52
BARBERM4-1	F(6)	0.15	0.09	9	298	26	0.21	0.13	7	0.44	0.36	6	316	25	0.07	0.02	0.5	133	44	3.1	0.28	0.43	346	58
BARBERM2-2	F(6)	0.12	0.08	8	184	26	0.18	0.11	7	0.31	0.25	6	198	25	0.05	0.01	0.45	115	40	1.72	0.22	0.36	276	46
BARBERM3-2	F(6)	0.13	0.09	9	241	26	0.19	0.12	7	0.37	0.31	6	257	25	0.06	0.02	0.46	124	42	2.3	0.26	0.4	311	52
BARBERM4-2	F(6)	0.15	0.1	9	298	26	0.21	0.13	7	0.44	0.36	6	316	25	0.07	0.02	0.5	133	44	3.09	0.29	0.43	346	58
BARBERM2-3	F(6)	0.12	0.08	8	184	26	0.19	0.11	7	0.31	0.25	6	198	25	0.06	0.01	0.45	115	40	1.71	0.22	0.36	276	46
BARBERM3-3	F(6)	0.14	0.08	9	241	26	0.19	0.12	7	0.37	0.31	6	257	25	0.06	0.02	0.46	124	42	2.39	0.26	0.4	311	52
BARBERM4-3	F(6)	0.14	0.1	10	298	26	0.22	0.13	7	0.45	0.37	6	316	25	0.07	0.02	0.5	133	44	3.1	0.28	0.04	346	58
BARBERMP-1	F(6)	0.2	0.22	9	1266	131	0.23	0.33	7	1.01	1.45	6	1282	139	0.2	0.12	0.7	1103	131	7.92	1.19	2.7	1544	156
BARBERMP-2	F(6)	0.21	0.21	9	1266	131	0.22	0.34	7	1.01	1.46	6	1282	139	0.2	0.12	0.7	1103	131	7.93	1.21	0.27	1544	156
BARBERMP-3	F(6)	0.2	0.22	9	1266	131	0.22	0.33	7	1.01	1.45	6	1282	139	0.19	0.12	0.7	1103	131	7.91	1.22	2.7	1544	156
BAKERY2-1	EF(4)	0.1	0.04	8	80	31	0.13	0.11	7	0.18	0.24	6	96	42	0.06	0.02	0.38	117	36	1.56	0.27	0.27	236	73
BAKERY2-2	EG(9), EF(1)	0.11	0.08	9	80	21	0.13	0.16	7	0.18	0.47	6	96	25	0.05	0.04	0.42	117	28	1.53	0.51	0.26	236	55
BAKERY3-1	EF(5)	0.17	1.52	33	558	306	0.3	3.92	10	0.64	10.39	6	569	424	0.37	0.77	0.64	646	382	6.22	6.5	1.83	1247	588
BAKERY3-2	EG(15), EF(4)	0.16	3.6	100	558	305	0.3	13.1	20	0.64	23.78	7	569	340	0.36	1.64	0.79	646	117	6.14	13.61	1.76	1247	230
BAKERY4-1	EF(6)	0.49	38.21	420	3284	2437	1.09	142.96	43	2.61	340.02	10	2691	3321	12.14	60.5	1.82	3444	6010	63.48	273.01	30.38	5542	5667
BAKERY4-2	EG(21), EF(6)	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	12.15	113	1.74	3444	869	63.45	546.81	29.75	5542	1027
TICKET2-1	F(4)	0.1	0.18	10	168	117	0.13	0.22	7	0.26	0.57	6	188	116	0.07	0.1	0.56	179	132	2	1.04	0.43	263	161
TICKET2-2	F(4), EG(5)	0.11	0.18	10	168	117	0.13	0.22	7	0.27	1.15	6	188	116	0.06	0.11	0.56	179	132	1.99	1.05	0.43	263	161
TICKET3-1	F(5)	0.13	0.91	33	315	107	0.17	1.99	10	0.46	4.51	7	354	627	0.16	1.36	0.93	333	562	4.78	8.93	1.93	492	839
TICKET3-2	F(5), EG(8)	0.14	1.67	33	315	107	0.18	1.99	13	0.45	12.34	8	354	627	0.16	1.3	0.93	333	562	4.78	8.93	1.93	492	839
TICKET4-1	F(6)	0.16	17.24	221	504	2775	0.23	27	27	0.7	116.97	8	568	2961	0.66	14.6	2.09	531	2336	11.86	74.72	10.2	789	4103
TICKET4-2	F(6), EG(11)	0.16	17.29	221	504	2775	0.23	26.94	27	0.71	116.79	17	568	2961	0.66	14.6	2.09	531	2336	11.86	74.74	10.2	789	4103
COHERENCE-1	EF(4)	0.19	0.25	10	1150	102	0.26	0.28	7	1.05	5.92	6	1203	263	0.11	0.33	0.7	730	86	3.56	4.51	1.52	1377	237
COHERENCE-2	EG(3), EF(5)	0.2	0.52	15	1150	51	0.27	1.99	10	1.04	8.49	7	1203	568	0.11	0.51	0.74	730	161	3.57	5.78	1.67	1377	472
COHERENCE-3	EG(3), EF(8)	0.21	2.46	30	1150	1013	0.26	7.3	14	1.02	47.16	7	1203	2019	0.11	2.2	0.98	730	544	3.55	25.27	3.8	1377	1216
COHERENCE-4	EG(150)	0.19	22.43	334	1150	448	0.27	42.91	39	1.04	206.13	12	1203	746	0.11	31.4	2	730	239	3.57	196.63	2.58	1377	641
PC5	EF(1)	0.13	0.03	8	440	54	0.15	0.05	7	0.43	0.92	6	587	74	0.09	0.04	0.55	244	61	4.26	0.61	0.94	680	144
PC10	EF(1)	0.19	0.06	8	700	54	0.22	0.08	7	0.62	1.55	7	917	74	0.19	0.06	0.79	283	61	11.9	0.7	1.26	811	166
RW4	EF(1)	0.09	0.01	7	50	3	0.12	0.01	6	0.12	0.01	6	40	3	0.02	0.01	0.03	71	5	0.63	0.03	0.17	136	18
RW8	EF(1)	0.13	0.01	8	100	3	0.18	0.01	7	0.2	0.02	6	80	3	0.04	0.01	0.31	143	5	2.61	0.05	0.24	256	30
RW16	EF(1)	0.23	0.02	10	200	3	0.36	0.02	7	0.42	0.05	6	160	3	0.12	0.01	0.04	287	5	13.01	0.09	0.41	496	54
RW32	EF(1)	0.65	0.03	19	400	3	1	0.03	8	1.2	0.09	7	320	3	0.38	0.02	0.07	575	5	76.28	0.17	0.76	976	102
RW64	EF(1)	2.83	0.06	49	800	3	3.86	0.06	13	4.61	0.18	10	640	3	1.82	0.07	1.33	1151	5	536.96	0.35	1.77	1936	198
SIS	EF(2)	694.87	0.34	19	107873	422	2.31	0.06	12	3.49	0.04	7	192	19	20.7	0.69	1.26	54568	431	152.85	1.92	39.22	95617	850
LIGHTCONTROL	EF(3)	0.2	0.08	10	108	12	0.18	0.07	7	0.28	0.2	6	72	12	0.07	0.02	0.04	187	27	2.41	0.3	0.34	301	70
INSERTIONSORT	EF(2)	0.11	0.01	7	140	13	0.11	0.15	7	0.21	0.67	6	173	86	0.06	0.02	0.46	127	19	1.5	0.77	0.43	231	101

By carefully inspecting Table 1 one can make the following remarks. Versions V4 and V5 that use only automata as a single representation for both integer and boolean variables require less memory, both for the transition relation and the fixpoint iterates, than V1, V2 and V3 that use disjunctive representations. This can be explained by the fact that deterministic minimal automata are canonical, while the composite symbolic representation is not. For most of the examples but not all, V4 performs better than V5 both in construction and verification times and memory requirements. This is in accordance to the measured sizes of the two automata implementations. On the other hand, by inspecting the data for V1, V2, V3, we can see a clear manifestation of time-memory tradeoff. For most examples V1 is faster than V2 and V3, but uses more memory. V3 consumes less memory, but is usually slower than the other two. V2 appears to be a good compromise between time and memory efficiency. Now, when comparing the composite versions V1 and V3 with their "pure automata" counterparts V4 and V5 respectively, we can conclude that the later generally perform better than the earlier, with the exception of most construction time measurements for V5 being greater that those for V3.

Based on these results we conclude that the versions based on our automata construction algorithms for linear integer arithmetic formulas and implemented using MONA automata package (V1 and V4) are the most time efficient of all, with V4 being more memory efficient than V1. If a composite representation must be used, Omega library based V2 can be a good time-memory compromise. Finally LASH based V5 is a close competitor to V4, but suffers from high construction times.

6 Conclusions

In this work we have presented new algorithms for constructing Finite Automata which represent integer sets that satisfy linear constraints. These automata can represent either signed or unsigned integers and have a lower number of states compared to other similar approaches. We have also showed how to construct minimal BDDs for any linear constraint on bounded variables. Finally we have tested these representations by using them to verify non-trivial specification examples and showed that in many cases they perform better than the polyhedral representation used in Omega Library, or the automata representation used in LASH.

References

1. The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.
2. The Omega project. `http://www.cs.umd.edu/projects/omega/`.
3. G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

4. A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Proceedings of the 21st International Colloquium on Trees in Algebra and Programming - CAAP'96*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, April 1996.
5. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.
6. Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.
7. P. J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proceedings of the 15th International Conference on Software Engineering*, pages 315–323, May 1993.
8. G. Delzanno and T. Bultan. Constraint-based verification of client-server protocols. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, 2001.
9. Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *Journal of Software Tools and Technology Transfer*, 3(3):250–270, 2001.
10. N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Proceedings of the 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer-Verlag, 2001.
11. N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.
12. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996. Also available through http://www.brics.dk/~klarlund/Mona/main.html.
13. T. A. Henzinger, P. Ho, and H. Wong-Toi. Hytech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
14. Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.
15. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis, 1992.
16. T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Proceedings of the 10th International Conference on Computer-Aided Verification*, 1998.
17. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 1–19. Springer, April 2000.
18. T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. Composite symbolic library. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.