International Journal of Foundations of Computer Science
© World Scientific Publishing Company

EFFICIENT SYMBOLIC REPRESENTATIONS FOR ARITHMETIC CONSTRAINTS IN VERIFICATION[*]

CONSTANTINOS BARTZIS and TEVFİK BULTAN
Department of Computer Science, University of California, Santa Barbara CA 93106, USA

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

In this paper we discuss efficient symbolic representations for infinite-state systems specified using linear arithmetic constraints. We give 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 present efficient storage techniques for the transition function of the automata and extend the construction algorithms to formulas on both boolean and integer variables. We also derive conditions which guarantee that the pre-condition computations used in symbolic verification algorithms do not cause an exponential increase in the automata size. We experimentally compare different symbolic representations by using them to verify non-trivial concurrent systems. Experimental results show that the symbolic representations based on our construction algorithms outperform the polyhedral representation used in Omega Library, and the automata representation used in LASH.

Keywords: automata, Presburger arithmetic, symbolic model checking

1. Introduction

Symbolic representations enable verification of systems with large state spaces which cannot be analyzed using enumerative approaches [1]. Recently, symbolic model checking has been applied to verification of infinite-state systems using symbolic representations that can encode infinite sets [2, 3, 4]. One class of infinite-state systems is systems that can be specified using linear arithmetic formulas on unbounded integer variables. Verification of such systems has many interesting applications such as monitor specifications [5], mutual exclusion protocols [3, 4], parameterized cache coherence protocols [6], and static analysis of access errors in dynamically allocated memory locations (buffer overflows) [7]. In this paper we present new symbolic representations for linear arithmetic formulas and experimental results on efficiency of different symbolic representations.

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

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 [8, 2, 4]. This approach can be extended to full Presburger arithmetic by including divisibility constraints (which can be represented as an equality constraint with an existentially quantified variable) [3, 9]. 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 [10, 11, 12, 13]. 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 construction algorithms for the automata representation of sets satisfying Presburger arithmetic formulas. We also experimentally compare two alternative automata representations and the polyhedra representation. Our construction algorithms are based on finite state transducers which compute linear functions (used in the arithmetic constraints that are being translated to FA) by processing the bits of the variables starting with the least significant bits. The input encoding is the same as the one used in [10] and the resulting FA are equivalent, however, our construction is also able to handle negative integers. Also we are able to prove tighter bounds on the sizes of the generated automata. The sizes of the generated automata and the input encoding in our construction are different than the construction given in [13]. We present automata construction for divisibility constraints which can be used as an alternative to projection operation required for Presburger formulas with quantification. We also show that the pre-condition computations required in symbolic verification algorithms do not cause an exponential increase in the automata representation if certain realistic conditions are satisfied.

We implemented our construction algorithm using the MONA tool [14] and integrated it to a set of tools for infinite-state model checking [15]. 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 [16] which uses the automata construction given in [13], and Omega Library [17] 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 [13]. Also automata representation is more efficient compared to the polyhedral representation used in [17].

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 present an alternative encoding and discuss efficient storage techniques for the transition function of the FA and extend the construction algorithms to formulas on both boolean and integer variables. In Section 5 we summarize the complexity results and discuss the related work. In Section 6 we derive conditions which guarantee that the pre-condition computations do not cause an exponential increase in the FA size. In Section 7 we discuss our implementation and experimental results. Finally, we state our conclusions in Section 8.

2. Finite Automata Representation for Linear Constraints on Natural Numbers

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 [10]. 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 include negative 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 FA(ϕ)=$(K, \Sigma, \delta, e, F)$ that accepts the language L(ϕ) over the alphabet $\Sigma = \{0,1\}^v$, which contains all the encodings of the natural number tuples that satisfy the formula. K is the set of automaton states, Σ is the input alphabet, $\delta : K \times \Sigma \to K$ is the transition function, $e \in K$ is the initial state, and $F \subseteq K$ is the set of final or accepting states.

2.1. Addition of v variables $\sum_{i=1}^{v} x_i$

As a basis for all following construction algorithms 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. Formally, the BSM is a Mealy machine or a finite state transducer $(K, \Sigma, \Lambda, \delta, e)$, where K is a set of states, Σ is the input alphabet, Λ is the output alphabet, $e \in K$ is the initial state, and $\delta : K \times \Sigma \to K \times \Lambda$ is the transition function from pairs of states and input symbols to pairs of states and output symbols.

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 initial state is the one with carry value zero. Formally, $K = \{0, 1, ..., v-1\}$, $\Sigma = \{0,1\}^v$, $\Lambda = \{0,1\}$, $e = 0$, and $\delta(k, (b_1, ..., b_v)) = (\lfloor (k + \sum_{i=1}^{v} b_i)/2 \rfloor, (k + \sum_{i=1}^{v} b_i) \bmod 2)$, where $a \bmod b$ is the remainder of the division of a by b.

2.2. 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

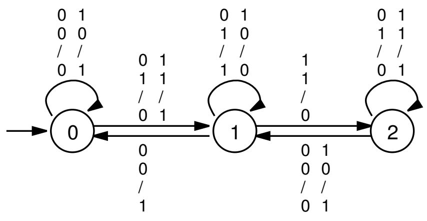

Fig. 1. A Basic State Machine for $x + 2y$

(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. Formally, Σ, Λ and e are the same as before, $K = \{k \mid \sum\limits_{a_i<0} a_i \leq k \leq \sum\limits_{a_i>0} a_i\}$ and $\delta(k, (b_1, ..., b_v)) = (\lfloor (k + \sum_{i=1}^{v} a_i \cdot b_i)/2 \rfloor, (k + \sum_{i=1}^{v} a_i \cdot b_i) \bmod 2)$.

An example BSM for calculating $x + 2y$ is shown in Figure 1. The transitions are labeled with the current bits of x and y and also the bit written to the output.

2.3. Checking for equality with zero $\sum_{i=1}^{v} a_i \cdot x_i = 0$

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 initial 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. Formally, $\mathrm{FA}(\sum_{i=1}^{v} a_i \cdot x_i = 0) = (K, \Sigma, \delta, e, F)$, where $K = \{k \mid \sum\limits_{a_i<0} a_i \leq k \leq \sum\limits_{a_i>0} a_i\} \cup \{sink\}$, $\Sigma = \{0,1\}^v$, $e = 0$, $F = \{0\}$, and the transition function δ is defined as:

$$\delta(k, (b_1, ..., b_n)) = \begin{cases} (k + \sum\limits_{i=1}^{v} a_i \cdot b_1)/2 & \text{if } k + \sum_{i=1}^{v} a_i \cdot b_1 \text{ is even and } k \neq sink \\ sink & \text{otherwise} \end{cases}$$

2.4. Checking inequalities

The situation now is simpler. If we are checking for < 0, the FA is identical to the BSM, except that it does not produce any output. 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 complement of the above described FA, by making

```
Input equation: $\sum_{i=1}^{v} a_i \cdot x_i = c$   Output FA: $(\Sigma, K, \delta, e, F)$

$\Sigma := \{0,1\}^v$
$K := \{-c, sink\}$
$e := -c$
FOR ALL $\sigma \in \Sigma$  $\delta(sink, \sigma) := sink$
WHILE $\exists k \in K, \sigma \in \Sigma$ s.t. $\delta(k, \sigma) = null$ DO
    FOR ALL $\sigma = (b_1, ..., b_v) \in \Sigma$ DO
        IF $k' := (\sum_{i=1}^{v} a_i \cdot b_i + k)/2 \in \mathbb{Z}$ THEN
            $K := K \cup \{k'\}$
            $\delta(k, \sigma) := k'$
        ELSE
            $\delta(k, \sigma) := sink$
$F := \{0\}$
```

Fig. 2. FA Algorithm for equations on natural numbers

```
Input inequation: $\sum_{i=1}^{v} a_i \cdot x_i < c$   Output FA: $(\Sigma, K, \delta, e, F)$

$\Sigma := \{0,1\}^v$
$K := \{-c\}$
$e := -c$
WHILE $\exists k \in K, \sigma \in \Sigma$ s.t. $\delta(k, \sigma) = null$ DO
    FOR ALL $\sigma = (b_1, ..., b_v) \in \Sigma$ DO
        $k' := \lfloor (\sum_{i=1}^{v} a_i \cdot b_i + k)/2 \rfloor$
        $K := K \cup \{k'\}$
        $\delta(k, \sigma) := k'$
$F := \emptyset$
FOR ALL $k \in K$
    IF $k < 0$ THEN $F := F \cup \{k\}$
```

Fig. 3. FA Algorithm for inequations on natural numbers

the accepting states rejecting and vice versa. For the rest of the cases ($> 0, \leq 0$), we change the signs of all coefficients and then we construct the FA like before. Formally, FA($\sum_{i=1}^{v} a_i \cdot x_i < 0$) $= (K, \Sigma, \delta, e, F)$, where $K = \{k \mid \sum\limits_{a_i<0} a_i \leq k \leq \sum\limits_{a_i>0} a_i\}$ is the set of states, $\Sigma = \{0,1\}^v$ is the alphabet, $e = 0$ is the initial state, $F = \{k \mid k \in K \wedge k < 0\}$ is the set of accepting states, and the transition function is $\delta(k, (b_1, ..., b_n)) = \lfloor (k + \sum_{i=1}^{v} a_i \cdot b_1)/2 \rfloor$.

2.5. Constant term

So far we assumed no constant term in the equations and inequations or, in other words, the constant term was zero. Now suppose that the linear combination of the variables is compared against a non-zero integer constant term c, i.e. we are constructing a FA for a constraint of the form $\sum_{i=1}^{v} a_i \cdot x_i \sim c$, where $\sim \in \{=, \neq, >, \geq, \leq, <\}$. All we have to do is make the state corresponding to a carry value of $-c$ the initial state of the FA ($e = -c$), 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 values between $-c$ and the carry value closest to $-c$. Specifically, for inequalities, the set of states becomes $K = \{k \mid \sum_{a_i<0} a_i \le k \le \sum_{a_i>0} a_i \vee 0 \le k \le -c \vee -c \le k \le 0\}$. Thus the number of states now becomes $|K| \le S =_{def} |\min(-c, \sum_{a_i<0} a_i)| + |\max(-c, \sum_{a_i>0} a_i)|$. For equalities, there is one extra sink state. Note that this upper bound on the number of states is tighter that the one given in [10], even though the construction algorithms are similar. The algorithms for constructing FA from equations and inequations are given in Figures 2 and 3, respectively.

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 $1 \le i \le l$, 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. Initial state is the initial 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 $(l+1) \cdot \sum_{i=1}^{v} |a_i|$, i.e. $|K| \le (\log_2 c + 1) \cdot \sum_{i=1}^{v} |a_i|$. 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.

2.6. Presburger formulas

Presburger formulas contain Boolean connectives between atomic linear constraints (equations and inequations) and quantifiers. Given two linear constraints ϕ_1 and ϕ_2 and the automata FA(ϕ_1) and FA(ϕ_2) representing them (with sizes |FA(ϕ_1)| and |FA(ϕ_2)| 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 $|\mathrm{FA}(\phi_1)| \cdot |\mathrm{FA}(\phi_2)|$ states. Similarly we can deal with NOT, by FA complementation. Given a formula ϕ containing l atomic linear constraints of the form $\phi_j : \sum_{i=1}^{v} a_{i,j} \cdot x_i \sim c_j, 1 \le j \le l$, where $\sim \in \{=, \neq, >, \ge, \le, <\}$, and Boolean connectives, one would expect the size of FA(ϕ) to be $\prod_{j=1}^{l} |\mathrm{FA}(\phi_\mathrm{j})|$. Fortunately, |FA($\phi$)| can be much less. Suppose ϕ contains m atomic linear constraints $\psi_1, ...\psi_m$ with distinct coefficients and FA(ψ_j)=$(K_j, \Sigma, \delta_j, e_j, F_j)$ is defined as described earlier. Now FA(ϕ)=$(K, \Sigma, \delta, e, F)$ can be constructed as follows. Each state of FA(ϕ) will correspond to a carry value for each of the distinct constraints (or sink for equalities). Transitions are defined in a similar way to those for single constraints. Finally, any state can be characterized as accepting iff ϕ evaluates to true according to the carries stored in that state. Formally, $\Sigma = \{0,1\}^v$, $K = \prod_{j=1}^{m} K_j$, $\delta((k_1, ..., k_m), \sigma) = (\delta(k_1, \sigma), ..., \delta(k_m, \sigma))$, and $e = (e_1, e_2, ..., e_m)$. Clearly the size of FA(ϕ) is $O(\prod_{j=1}^{m} \sum_{i=0}^{v} |a_{i,j}|)$. Hence, the size of the FA(ϕ) depends on the number of distinct atomic constraints in ϕ, i.e., repetition of an atomic constraint does not increase the size of the FA(ϕ).

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 size of the resulting deterministic FA can be exponential on the size of the initial FA, i.e. $|\mathrm{FA}(\exists x_i.\phi)| = 2^{|\mathrm{FA}(\phi)|}$ in the worst case. The resulting FA may not accept *all* satisfying encodings (with any number of leading zeros). We can overcome this by identifying all rejecting states k such that $\delta(k, (0, 0, ..., 0)) \in F$, and make them accepting. Clearly, this modification does not increase the size of the FA. Universal quantification can be similarly implemented by the use of the FA complementation.

It is known that any Presburger formula can be expressed in a disjunctive form, where each disjunct is a conjunction of equality, inequality and divisibility constraints [18]. Divisibility constraints are of the form $\sum_{i=1}^{v} a_i \cdot x_i + c \equiv_d 0$, where d is a positive integer constant, and $\equiv_d$ means equivalent modulo d. Given a Presburger arithmetic formula in the form $\exists x_i.\phi$ where ϕ is a conjunction of atomic equality and inequality constraints, one can construct a formula ϕ' which is equivalent to $\exists x_i.\phi$ and ϕ' consists of atomic equality, inequality, and divisibility constraints and x_i does not appear in ϕ' [18]. Hence, if we can construct FA for arithmetic formulas with divisibility constraints we would have an alternative approach to constructing FA for quantified formulas. Having shown how to construct FA for equations and inequations, here we describe the construction of FA for divisibility constraints.

First we describe a FA $(K, \Sigma, \delta, e, F)$ that accepts an integer if it is divisible by d, by reading its bits in most-significant to least-significant bit order, i.e. $\Sigma = \{0, 1\}$. There are d states, each representing the remainder of the division of the input with d, i.e., $K = \{0, 1, ..., d-1\}$. State 0 is the entry state and also the only accepting state. Finally, $\delta(k, 0) = (2 \cdot k) \bmod d$ and $\delta(k, 1) = (2 \cdot k + 1) \bmod d$. In order to make this FA "compatible" with those for equations and inequations described earlier, we need to reverse the bit order. That is easy when d is odd. We just reverse all the transitions. One can easily verify that the resulting FA is deterministic. This is because in the initial FA the entry state is the same with the unique accepting state and also there are no same-labeled transitions originating in distinct states and going to the same state. If d is even, then it can be written as $2^n \cdot m$ where m is odd and $n > 0$. We can build a FA for divisibility with m as described above and add $n + 1$ states to check if the first n least significant bits of the input are zero. The resulting FA will have at most d states. Now we can define $\mathrm{FA}(\sum_{i=1}^{v} a_i \cdot x_i + c \equiv_d 0)=(K, \Sigma, \delta, e, F)$, based on $\mathrm{FA}(y \equiv_d 0)=(K_{div}, \{0, 1\}, \delta_{div}, 0, \{0\})$ and $\mathrm{BSM}(\sum_{i=1}^{v} a_i \cdot x_i + c)=(K_{BSM}, \Sigma_{BSM}, \Lambda, \delta_{BSM}, e_{BSM})$. In particular, $K = K_{BSM} \times K_{div}$, $\Sigma = \Sigma_{BSM}$, $e = (e_{BSM}, 0)$, $F = F_{BSM} \times \{0\}$ and finally $\delta((k_1, k_2), \sigma) = (k_1', k_2')$ iff $\delta_{BSM}(k_1, \sigma) = (k_1', b) \wedge \delta_{div}(k_2, b) = k_2'$. The size of this FA is $S \cdot d$, where S was defined in Section 2.5.

3. Finite Automata for Linear Constraints on 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,

```
Input equation: $\sum_{i=1}^{v} a_i \cdot x_i = c$   Output FA: $(\Sigma, K, \delta, e, F)$

$\Sigma := \{0,1\}^v$
$K := \{(-c,a),(-c,r), sink\}$
$e := (-c,a)$
FOR ALL $\sigma \in \Sigma$  $\delta(sink,\sigma) := sink$
WHILE $\exists (k,a) \in K, \sigma \in \Sigma$ s.t. $\delta((k,a),\sigma) = null$ DO
    FOR ALL $\sigma = (b_1,...,b_v) \in \Sigma$ DO
        IF $k' := (\sum_{i=1}^{v} a_i \cdot b_i + k)/2 \in \mathbb{Z}$ THEN
            IF $k' = k$ THEN
                $\delta((k,a),\sigma) := (k,a)$
            ELSE
                $K := K \cup \{(k',a),(k',r)\}$
                $\delta((k,a),\sigma) := (k',r)$
        ELSE
            $\delta((k,a),\sigma) := sink$
        $\delta((k,r),\sigma) := \delta((k,a),\sigma)$
$F := \emptyset$
FOR ALL $(k,a) \in K$ DO
    $F := F \cup \{(k,a)\}$
```

Fig. 4. FA Algorithm for equations on integers

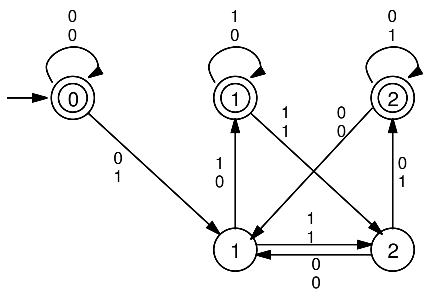

Fig. 5. Example automaton for $x + 2y = 0$

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.

3.1. 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 (k,a) and (k,r) of each state

```
Input inequation: $\sum_{i=1}^{v} a_i \cdot x_i < c$   Output FA: $(\Sigma, K, \delta, e, F)$

$\Sigma := \{0,1\}^v$
$K := \{(-c,a),(-c,r)\}$
$e := (-c,a)$
WHILE $\exists(k,a) \in K, \sigma \in \Sigma$ s.t. $\delta((k,a),\sigma) = null$ DO
    FOR ALL $\sigma = (b_1,...,b_v) \in \Sigma$ DO
        $k' := \lfloor(\sum_{i=1}^{v} a_i \cdot b_i + k)/2\rfloor$
        $temp1 := k$
        $temp2 := k'$
        WHILE $temp1 \neq temp2$ DO
            $temp1 := temp2$
            $temp2 := \lfloor(\sum_{i=1}^{v} a_i \cdot b_i + temp1)/2\rfloor$
        IF $(\sum_{i=1}^{v} a_i \cdot b_i + temp1)/2 \in \mathbb{Z}$ THEN
            $\delta((k,a),\sigma) := (k',r)$
        ELSE
            $\delta((k,a),\sigma) := (k',a)$
        $\delta((k,r),\sigma) := \delta((k,a),\sigma)$
$F := \emptyset$
FOR ALL $(k,a) \in K$ DO
    $F := F \cup \{(k,a)\}$
```

Fig. 6. FA Algorithm for inequations on integers

k of the BSM, one accepting and one rejecting respectively. 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. Formally, one can define the FA$(\sum_{i=1}^{v} a_i \cdot x_i = 0) = (K_{FA}, \Sigma, \delta_{FA}, e_{FA}, F)$ based on the BSM$(\sum_{i=1}^{v} a_i \cdot x_i)$=$(K_{BSM}, \Sigma, \Lambda, \delta_{BSM}, e_{BSM})$. In particular, $K_{FA} = \{(k,a),(k,r) \mid k \in K_{BSM}\} \cup \{sink\}$, $e_{FA} = (e_{BSM}, a)$, $F = \{(k,a) \mid k \in K_{BSM}\}$ and

$$\delta_{FA}((k,a),\sigma) = \delta_{FA}((k,r),\sigma) = \begin{cases} (k,a) & \text{iff } \delta_{BSM}(k,\sigma) = (k,0) \\ (k',r) & \text{iff } \delta_{BSM}(k,\sigma) = (k',0) \wedge k' \neq k \\ sink & \text{iff } \delta_{BSM}(k,\sigma) = (k',1) \end{cases}$$

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 Section 2.5. One can easily verify that this is the minimum possible. The algorithm for constructing FA for equations on integers is shown in Figure 4. The example automaton for $x + 2y = 0$ is shown in Figure 5. The sink state and all transitions to it have been omitted.

3.2. Finite automata for inequalities

Like before we will only describe the < 0 case. All other types of inequalities can be 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

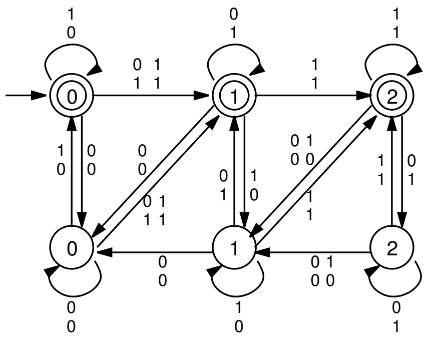

Fig. 7. Example automaton for $x + 2y < 0$

writes 1. Any number of repetitions greater or equal to the binary logarithm of the sum of the absolute values of the coefficients is adequate. That is 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. This condition is checked in the second WHILE loop in Figure 6. As explained earlier the loop will finish in at most $\log_2(\sum_{i=1}^{v} |a_i|)$ steps. Again, one can formally define the FA$(\sum_{i=1}^{v} a_i \cdot x_i < 0)$ based on the BSM$(\sum_{i=1}^{v} a_i \cdot x_i)$. Obviously this FA has $2S$ states. The algorithm for constructing FA for inequations on integers is shown in Figure 6. The example automaton for $x + 2y < 0$ is shown in Figure 7.

Note that for both equations and inequations, the automata resulting from our construction are deterministic, unlike those described in [13] which require (linear time) determinization. Secondly, once a state has been created, all transitions originating from it can be computed immediately (as opposed to [13]), 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 [13], if we follow the BSM stacking method for dealing with the constant term, described in Section 2.5 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 [13] result in different automata, because the accepted languages are different (one is the reverse of the other).

4. Practical issues and extensions

In all FA construction algorithms given so far, v-tuples of integers have been encoded as words over the alphabet $\{0,1\}^v$. While this encoding is convenient for describing the algorithms and proving their complexity, it is inefficient in practice.

The reason is that the size of the transition function is proportional to the size of the alphabet. Thus, an alphabet with size exponential in the number of variables v is problematic. In this section we discuss two ways of dealing with this problem: 1) changing the encoding and 2) storing the transition function more efficiently. Finally, we use these ideas to construct FA for combined Presburger and Boolean formulas.

4.1. 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. Particularly, a v-tuple of natural numbers $(n_1, n_2, ..., n_v)$ is encoded as a word over the alphabet $\{0, 1\}$, where the i_{th} least significant bit of number n_j appears in position $(i-1) \cdot v + j$ of the word.

The construction 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.

4.2. Storing the transition function δ

A problem with the FA representation for arithmetic constraints is the size of the transition function, since the number of transitions from each state is exponential on v, the number of integer variables. Hence, it is impractical to store the transition function as a table. Actual implementations use different solutions to this problem. LASH [16] follows a technique similar to the Alternative FA described earlier. The input is serialized by interleaving the bits of each integer variable in some fixed order. This way there are (linearly) more states but only two transitions from each state. The weakness of this approach is that the number of states in the FA is proportional to the total number of variables and not only those with non-zero coefficients. On the other hand, MONA is an automata package [14] that uses BDDs [19] to store the transition function. In particular, for each FA state n, there is a BDD representing the function $\delta(n, (b_1, ..., b_v))$. The terminal nodes are also FA states and internal nodes can be shared. Since BDDs are a canonical representation for Boolean functions, given a fixed variable ordering, the size of the transition relation can be kept minimal, e.g., variables with zero coefficients do not appear in the BDD representing the transition function. Note that this BDD is isomorphic to the transition graph of the Alternative FA of the previous paragraph, thus its size is linear on the number of integer variables v.

4.3. Adding Boolean formulas

Clearly, a Boolean formula ϕ with n boolean variables can be represented by a FA that accepts the valuations of the Boolean variables that satisfy the formula. Using the alphabet $\{0,1\}$, any word encoding a satisfying valuation has a fixed length n, hence the according language $\mathrm{L}(\phi)$ is regular. Now suppose that ϕ is a general formula containing both Boolean subformulas $(B_1, ..., B_n)$ and Presburger subformulas $(P_1, ..., P_m)$ combined with Boolean connectives ($\wedge, \vee, \neg$, etc). Obviously one can construct a FA for ϕ, with size $\prod_{i=1}^{n} |\mathrm{FA}(B_i)| \cdot \prod_{i=1}^{m} |\mathrm{FA}(P_i)|$. Fortunately, one can take advantage of the BDD representation for the transition function described in the previous paragraph to reduce the size of the FA for general formulas. The idea is to put the Boolean variables first in the total variable ordering. This way, the Boolean variables appear only in the first BDD representing the transition function from the initial state of $\mathrm{FA}(\phi)$. If one can find a good variable ordering for the Boolean variables, this BDD can be kept small. In the worst case the size of this BDD is at most $\prod_{i=1}^{n} |\mathrm{FA}(B_i)|$. The rest of the BDDs that represent the transitions in the $\mathrm{FA}(P_i)$ for $1 \leq i \leq m$ are independent of the Boolean variables and thus their total size is at most $\prod_{i=1}^{m} |\mathrm{FA}(P_i)|$. Finally the total size of the BDD representing the transition function of $\mathrm{FA}(\phi)$ is $\prod_{i=1}^{n} |\mathrm{FA}(B_i)| + \prod_{i=1}^{m} |\mathrm{FA}(P_i)|$.

5. Complexity Outline

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, a_i are the coefficients, c is the constant term and $S = |\min(-c, \sum\limits_{a_i<0} a_i)| + |\max(-c, \sum\limits_{a_i>0} a_i)|$.

	FA for non-negative integers	FA for all integers	LASH
Construction Time	$2^v S$	$2^{v+1} S \log_2 S$	$2^{v+1} \log(\lvert c\rvert) \sum_{i=1}^{v} \lvert a_i\rvert$
Construction Size	S	$2S$	$2 \log(\lvert c\rvert) \sum_{i=1}^{v} \lvert a_i\rvert$

The reader can find similarities between our FA construction algorithms and those proposed in [10] and [13]. Here we briefly explain the advantages of our approach. In [10] only constraints over non-negative integers are considered. We describe FA for constraints over all integers as well, by only doubling the complexity.

On the other hand, in [13] the complexity of $O(2 \cdot \log_2 |c| \cdot \sum_{i=1}^{v} |a_i|)$, where c is the constant term and $|a_i|$ are 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 [13] 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. Also, in Section 4.1 we give a detailed complexity analysis for an alternative automaton that is equivalent to the "sequentialized" FA described in [13].

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

6. Pre-condition computations

One of the fundamental operations in symbolic verification algorithms is computing the pre-condition of a set of states (configurations) of a system. One interesting issue is investigating the sizes of the FA that would be generated by the pre-condition operation if one uses the FA encoding described in the previous sections as a symbolic representation.

Given a set of states $S \subseteq \mathbb{Z}^v$ of a system as a relation on the state variables $x_1, \ldots, x_v$ and the transition relation $R \subseteq \mathbb{Z}^{2v}$ of the system as a relation on the current state and next state variables $x_1, \ldots, x_v, x'_1, \ldots, x'_v$, we would like to compute the pre-condition of S with respect to R, $pre(S, R) \subseteq \mathbb{Z}^v$. We consider systems where S and R can be represented as Presburger arithmetic formulas, i.e., $S = \{(x_1, \ldots, x_v) \mid \phi_S(x_1, \ldots, x_v)\}$ and $R = \{(x_1, \ldots, x_v, x'_1, \ldots, x'_v) \mid \phi_R(x_1, \ldots, x_v, x'_1, \ldots, x'_v)\}$, where ϕ_S and ϕ_R are Presburger arithmetic formulas. For example, consider a system with three integer variables x_1, x_2 and x_3. Let the current set of states be $S = \{(x_1, x_2, x_3) \mid x_1 + x_2 = x_3\}$ and the transition relation be $R = \{(x_1, x_2, x_3, x'_1, x'_2, x'_3) \mid x_1 > 0 \wedge x'_1 = x_1 - 1 \wedge x'_2 = x_2 \wedge x'_3 = x_3\}$. Then the pre-condition of S with respect to R is $pre(S, R) = \{(x_1, x_2, x_3) \mid x_1 > 0 \wedge x_1 + x_2 = x_3 + 1\}$.

One can compute $pre(S, R)$ by first renaming the variables in ϕ_S, then conjoining it with ϕ_R, and then existentially eliminating the next state variables, i.e., $\phi_{pre(S,R)}$ is equivalent to $\exists x'_1 \ldots \exists x'_v.(\phi_{S[x'_1 \leftarrow x_1, \ldots, x'_v \leftarrow x_v]} \wedge \phi_R)$ where $\psi_{[y \leftarrow z]}$ is the formula generated by substituting z for y in ψ. Hence, to compute $pre(S, R)$ we need three operations: conjunction, existential variable elimination and renaming. As stated in Section 2.6, given FA(ϕ) representing the set of solutions of ϕ, FA($\exists x_1, \ldots, \exists x_n.\phi$) can be computed using projection, and the size of the resulting FA is at most $O(2^{|\mathrm{FA}(\phi)|})$. Note that, existential quantification of more than one variable does not increase the worst case complexity since the determinization can be done once at the end, after all the projections are done. As discussed earlier, conjunction operation can be computed by generating the product automaton, and the renaming operation can be implemented as a linear time transformation of the transition function. Hence, given formulas ϕ_S and ϕ_R representing S and R, and corresponding FA, FA(ϕ_S) and FA(ϕ_R), the size of the automaton FA($\phi_{pre(S,R)}$) is $O(2^{|\mathrm{FA}(\phi_S)| \cdot |\mathrm{FA}(\phi_R)|})$ in the worst case. Below, we show that under some realistic assumptions, the size of the automaton resulting from the pre-condition computation can be much better.

We assume that the formula ϕ_R defining the transition relation R is a guarded-update of the form $guard(R) \wedge update(R)$, where $guard(R)$ is a Presburger formula on current state variables $x_1, ..., x_v$ and $update(R)$ is of the form $x'_i = f(x_1, ..., x_v) \wedge \bigwedge_{j \neq i} x'_j = x_j$ for some $1 \leq i \leq v$, where $f : Z^v \to Z$ is a linear function. This is a realistic assumption, since in asynchronous concurrent systems, the transition relation is usually defined as a disjunction of such guarded-updates. This holds for all the concurrent systems we experimented with in Section 7. Also, note that, the pre-condition of a transition relation which is a disjunction of guarded-updates is the union of the pre-conditions of individual guarded-updates, and can be computed by computing pre-condition of one guarded-update at a time.

First, we consider two kinds of updates: $x'_i = c$ and $x'_i = x_i + c$, where c is and integer constant. Given ϕ_S and ϕ_R, the formulas defining S and R, respectively, we can show that for these two cases the size of FA$(\phi_{pre(S,R)})$ can be less than or equal to the size of the FA$(\phi_S \wedge guard(R))$ in certain cases, and in the worst case it is exponential in the number of distinct atomic constraints in ϕ_S, which is a significant improvement over the upper bound of $O(2^{|\mathrm{FA}(\phi_S)| \cdot |\mathrm{FA}(\phi_R)|})$ given above.

Suppose ϕ_S consists of l atomic linear constraints of the form $\phi_j : \sum_{i=1}^{v} a_{i,j} \cdot x_i \sim c_j, 1 \leq j \leq l$, where $\sim \in \{=, \neq, >, \geq, \leq, <\}$, and Boolean connectives. If x_k is updated by $x'_k = c$, then $\phi_{pre(S,R)}$ is equivalent to $\phi_{S[x_k \leftarrow c]} \wedge guard(R))$. The FA$(\phi_j)$ can be constructed by the BSM stacking method described in Section 2.5 where the resulting FA will have $\log_2 c_j$ layers. The FA$(\phi_{j[x_k \leftarrow c]})$ can be constructed based on the BSM stacking method for FA(ϕ_j). The alphabet of FA(ϕ_j) includes a bit for variable x_k whereas the alphabet of FA$(\phi_{j[x_k \leftarrow c]})$ does not. Each layer in FA$(\phi_{j[x_k \leftarrow c]})$ has the same states as in FA(ϕ_j). Each transition in FA$(\phi_{j[x_k \leftarrow c]})$ corresponds to a transition in FA(ϕ_j) where the bit of x_k for that layer is equal to the corresponding bit of c. The number of layers now becomes $\max(\log_2 |c_j|, \log_2 |c|) + 1$. If $|c| \leq |c_j|$, then $|\mathrm{FA}(\phi_{j[x_k \leftarrow c]}| = |\mathrm{FA}(\phi_j)|$. If this holds for all j, the size of the FA$(\phi_{pre(S,R)})$ is equal to the size of the FA$(\phi_S \wedge guard(R))$. Otherwise, it may increase exponentially on the number of $|c_j|$s that are less than $|c|$.

If x_k is updated by $x'_k = x_k + c$, then $\phi_{pre(S,R)}$ can be derived from ϕ_S by replacing each atomic constraint $\sum_{i=1}^{v} a_{i,j} \cdot x_i \sim c_j$ with $\sum_{i=1}^{v} a_{i,j} \cdot x_i \sim c_j - a_{k,j} \cdot c$ and then conjuncting the result with $guard(R)$. The FA for both of these constraints can be constructed by the algorithms described in Section 2. If $\min(-c_j, \sum_{a_i < 0} a_i) \leq -(c_j - a_{k,j} \cdot c) \leq \max(-c_j, \sum_{a_i > 0} a_i)$, then $|\mathrm{FA}(\sum_{i=1}^{v} a_{i,j} \cdot x_i \sim c_j - a_{k,j} \cdot c)| \leq |\mathrm{FA}(\sum_{i=1}^{v} a_{i,j} \cdot x_i \sim c_j)|$. Again, if this condition holds for all j, the size of the FA$(\phi_{pre(S,R)})$ is less than or equal to the size of the FA$(\phi_S \wedge guard(R))$. Otherwise, it may increase exponentially on the number of atomic constraints, for which the condition is violated.

In the most general case, an update can be in the form $x'_i = \sum_{j=1}^{v} a_j \cdot x_j + c$. Even for such updates the number of atomic constraints in $\phi_{pre(S,R)}$ will be at most the number of atomic constraints in ϕ_S plus the number of atomic constraints in $guard(R)$. An update in the above form may change all the coefficients on all atomic constraints ϕ_j in ϕ_S. However, the size of the FA$(\phi_{pre(S,R)})$ will not be more than the size of the FA$(\phi_S \wedge guard(R))$ if the constant term and the sum of the absolute values of the coefficients in each individual atomic constraint ϕ_j do not increase.

7. Implementation and Experiments

In this section we discuss the implementation of the presented algorithms and experimentally compare two alternative automata representations and the polyhedra representation for arithmetic constraints. Earlier results from our experiments were reported in [20].

In [21] 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 [21]. In [21] 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 [21].

7.1. Implementation

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 [17]. 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.

MONA is an automata manipulation tool which also implements decision procedures for the Weak Second-order Theory of One or Two successors [22]. 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 using 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 [16]. 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 efficiency.

We integrated our construction algorithms to an infinite state CTL model checker called Action Language Verifier [23] built on top of the Composite Symbolic Library [15]. The Composite Symbolic Library uses an object-oriented design to combine different symbolic representations [15]. An abstract interface defines the operations used in symbolic verification: Boolean operations, equivalence and subsumption tests, and pre- and post-condition computations. 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 [15].

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 [16]) (version V3), Omega [17] (version V2), and our automata construction algorithms (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 automaton, hence one can avoid using the disjunctive composite representation.

7.2. Experiments

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 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 [6]. We verified three properties given in [6]. RW(4,16,64) is a monitor specification for readers-writers problem for various number of processes [24]. AIRPORT(2,4,8,16) is a monitor specification from [25] for airport ground traffic control simulation with various number of processes.

The results of our experimental evaluation of different representations are shown in Tables 1 and 2. We obtained the experimental results on a SUN ULTRA 10 work station with 768 Mbytes of memory, running SunOs 5.7. In Table 2, we show the types and the number of fixpoint iterations (F denotes the forward fixpoint computation, EG and EF denote the fixpoint computations for corresponding CTL operators), and the number of integer and boolean variables for each problem instance. For each version of the verifier we recorded the following statistics: 1) Time

Table 1. Experimental results for performance of different encodings of integer arithmetic constraints within the disjunctive composite representation. Time measurements appear in seconds.

	Disjunctive Composite Representation											
	V1 - Automata				V2 - Omega				V3 - LASH			
Problem Instance	CT	VT	TRS	MS	CT	VT	TR AC	MAX AC	CT	VT	TRS	MS
BARBERM2-1	0.15	0.90	27(184)	86(224)	0.14	1.57	48	87	0.19	17.10	198	283
BARBERM3-1	0.13	1.53	35(241)	112(294)	0.16	2.69	60	105	0.22	40.55	257	365
BARBERM4-1	0.17	2.57	43(298)	142(34)	0.17	4.13	72	123	0.27	79.37	316	461
BARBERM2-2	0.14	0.12	27(184)	13(13)	0.14	0.12	48	12	0.19	2.16	198	51
BARBERM3-2	0.16	0.13	35(241)	13(22)	0.16	0.15	60	12	0.23	3.03	257	51
BARBERM4-2	0.17	0.16	43(298)	13(17)	0.17	0.18	72	12	0.28	3.86	316	51
BARBERM2-3	0.16	0.24	27(184)	36(71)	0.14	0.31	48	30	0.19	8.60	198	125
BARBERM3-3	0.16	0.65	35(241)	58(120)	0.15	0.77	60	42	0.23	22.64	257	202
BARBERM4-3	0.15	1.42	43(298)	82(174)	0.16	1.51	72	54	0.27	49.85	316	281
BARBERMP-1	0.16	0.39	82(1266)	26(131)	0.16	0.45	72	36	0.42	1.29	1282	139
BARBERMP-2	0.14	0.37	82(1266)	26(131)	0.14	0.45	72	36	0.42	1.28	1282	139
BARBERMP-3	0.14	0.39	82(1266)	26(131)	0.15	0.45	72	36	0.43	1.29	1282	139
BAKERY2-1	0.12	0.02	18(80)	15(31)	0.14	0.07	32	20	0.12	0.22	96	42
BAKERY2-2	0.12	0.09	18(80)	6(13)	0.13	0.16	32	8	0.12	0.42	96	25
BAKERY3-1	0.15	0.62	63(558)	97(306)	0.19	2.06	126	183	0.41	9.31	569	424
BAKERY3-2	0.20	3.06	63(558)	63(232)	0.19	12.06	126	139	0.42	22.66	569	340
BAKERY4-1	0.32	12.58	228(3284)	551(2437)	0.34	58.95	396	1820	1.51	298.82	2691	3321
BAKERY4-2	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
TICKET2-1	0.14	0.13	18(168)	24(117)	0.15	0.14	38	24	0.17	0.52	188	116
TICKET2-2	0.14	0.13	18(168)	24(117)	0.14	0.22	38	24	0.27	1.15	188	116
TICKET3-1	0.18	0.79	27(315)	99(611)	0.17	0.98	66	119	0.28	4.12	354	627
TICKET3-2	0.15	1.83	27(315)	99(611)	0.16	1.91	66	119	0.45	12.34	354	627
TICKET4-1	0.18	6.28	36(504)	365(2775)	0.19	13.11	100	638	0.42	30.30	568	2961
TICKET4-2	0.19	17.95	36(504)	365(2775)	0.20	25.78	100	638	0.71	116.79	568	2961
COHERENCE-1	0.18	0.19	74(1150)	39(23)	0.19	0.12	120	42	0.53	5.50	1203	263
COHERENCE-2	0.19	0.89	74(1150)	165(573)	0.19	0.62	120	105	1.04	8.49	1203	568
COHERENCE-3	0.18	2.55	74(1150)	323(1595)	0.18	2.17	120	189	1.02	47.16	1203	2019
RW4	0.12	0.01	20(50)	2(3)	0.12	0.01	16	1	0.09	0.01	40	3
RW16	0.31	0.01	80(200)	2(3)	0.33	0.02	64	1	0.28	0.04	160	3
RW64	3.37	0.07	320(800)	2(3)	3.76	0.08	256	1	3.47	0.18	640	3
AIRPORT2	0.70	0.26	148(3958)	3(15)	0.96	4.48	1094	247	4.49	65.44	4454	52
AIRPORT4	1.48	0.37	296(7916)	3(15)	1.93	8.34	2188	247	9.09	68.15	8908	52
AIRPORT8	3.76	0.60	592(15832)	3(15)	4.95	16.05	4376	247	20.00	74.50	17816	52
AIRPORT16	12.70	1.12	1184(31664)	3(15)	13.09	31.16	8752	247	48.35	87.31	35632	52

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) 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, for versions V1 and V4, we also give the total number of BDD nodes used in the MONA representation in parentheses. For V2 which uses a polyhedral representation we give the number of total atomic arithmetic constraints in the transition relation (TR AC) and the largest fixpoint iterate (MAX AC). For the instances marked with symbol $\uparrow$, the verification tool did not converge in an hour.

By inspecting the Tables 1 and 2 one can make the following observations. The number of states of the automata used in V1 is about an order of magnitude less that the number of states of the automata used in V3. This is due to the different encodings used in V1 and V3. Note that the transition function in V3 requires a constant amount of memory per state of the automata whereas in V1 the transitions from each state are encoded using BDDs. Hence, a better caparison would be to compare the number of states of the automata in V3 to the number of

Table 2. Experimental results for performance of different automata-based encodings of integer arithmetic constraints and boolean formulas. Time measurements appear in seconds.

				V4 - Automata				V5 - LASH			
Problem Instance	Computed Fixpoints	int var	bool var	CT	VT	TRS	MS	CT	VT	TRS	MS
BARBERM2-1	EF(9)	3	4	0.04	0.13	11(115)	7(51)	1.41	4.85	276	177
BARBERM3-1	EF(10)	3	5	0.04	0.19	11(124)	9(63)	2.08	6.54	311	230
BARBERM4-1	EF(11)	3	6	0.05	0.26	11(133)	9(76)	2.7	8.42	346	287
BARBERM2-2	EF(5)	3	4	0.04	0.05	11(115)	5(19)	1.41	2.48	276	61
BARBERM3-2	EF(5)	3	5	0.05	0.06	11(124)	5(20)	2.08	2.73	311	69
BARBERM4-2	EF(5)	3	6	0.05	0.07	11(133)	5(21)	2.69	3.04	346	76
BARBERM2-3	EF(11)	3	4	0.04	0.16	11(115)	6(37)	1.44	5.64	276	165
BARBERM3-3	EF(12)	3	5	0.04	0.23	11(124)	7(54)	2.07	8.22	311	305
BARBERM4-3	EF(13)	3	6	0.05	0.37	11(133)	8(71)	2.69	11.72	346	479
BARBERMP-1	F(7)	6	2	0.05	0.11	68(1103)	21(131)	1.33	1.02	1544	156
BARBERMP-2	F(7)	6	2	0.04	0.11	68(1103)	21(131)	1.32	1.02	1544	156
BARBERMP-3	F(7)	6	2	0.04	0.11	68(1103)	21(131)	1.32	1.02	1544	156
BAKERY2-1	EF(4)	2	4	0.04	0.02	14(117)	12(36)	1.43	0.25	236	73
BAKERY2-2	EG(9), EF(1)	2	4	0.04	0.04	14(117)	9(28)	1.57	0.52	236	55
BAKERY3-1	EF(5)	3	6	0.08	0.62	56(646)	91(382)	4.04	6.39	1247	588
BAKERY3-2	EG(15), EF(1)	3	6	0.32	1.43	56(646)	30(117)	6.24	13.80	1247	230
BAKERY4-1	EF(6)	4	8	0.22	56.00	218(3444)	964(6010)	11.05	226.52	5542	5667
BAKERY4-2	EG(21), EF(6)	4	8	11.67	106.01	218(3444)	142(869)	64.33	546.90	5542	1027
TICKET2-1	F(4)	4	4	0.04	0.04	12(179)	19(132)	1.80	0.54	263	161
TICKET2-2	F(4), EG(5), EF(1)	4	4	0.06	0.09	12(179)	19(132)	2.02	1.07	263	161
TICKET3-1	F(5)	5	6	0.06	0.48	16(333)	64(562)	3.83	3.51	492	839
TICKET3-2	F(5), EG(8), EF(1)	5	6	0.14	1.29	16(333)	64(562)	4.85	9.04	492	839
TICKET4-1	F(6)	6	8	0.09	4.67	20(531)	232(2336)	6.63	23.13	789	4103
TICKET4-2	F(6), EG(11), EF(1)	6	8	0.62	13.10	20(531)	232(2336)	12.03	75.68	789	4103
COHERENCE-1	EF(4)	6	4	0.07	0.31	40(730)	18(86)	3.13	4.49	1377	237
COHERENCE-2	EG(6), EF(5)	6	4	0.11	0.49	40(730)	36(161)	3.60	5.71	1377	472
COHERENCE-3	EG(9), EF(8)	6	4	0.10	2.42	40(730)	94(544)	3.59	24.88	1377	1216
RW4	EF(1)	1	5	0.02	0.01	6(71)	4(5)	0.55	0.04	136	18
RW16	EF(1)	1	17	0.08	0.01	6(287)	4(5)	12.62	0.10	496	54
RW64	EF(1)	1	65	1.14	0.06	6(1151)	4(5)	531.55	0.35	1936	198
AIRPORT2	EF(1)	13	8	0.70	3.24	34(2070)	3(15)	53.41	177.77	3058	76
AIRPORT4	EF(1)	13	16	1.41	65.82	34(2710)	3(15)	↑	↑	↑	↑
AIRPORT8	EF(1)	13	32	↑	↑	↑	↑	↑	↑	↑	↑
AIRPORT16	EF(1)	13	64	↑	↑	↑	↑	↑	↑	↑	↑

BDD nodes used to encode the transition function of the automata in V1 (shown in parenthesis under TRS and MS). We see that these numbers are comparable for each problem instance. However, V1 is much faster than V3. Hence, although the BDD encoding requires same amount of space as the encoding used in V3, it improves the verification time significantly. The amount of memory required for the polyhedra based approach used in V2 is proportional to the number of atomic constraints in the polyhedra which is shown under TR AC and MAX AC columns under V2. The increase in the memory requirement of V2 with the increasing problem size seems to be similar to the one observed for the automata based approaches V1 and V3. The verification time for V2 is between the verification time required for V1 and V3.

In general the memory requirement for version V4 seems to be less than that of V5. Also verification times for V4 are significantly faster than those of V5. V4 and V5 are both able to verify the problem instance BAKERY4-2 whereas V1, V2, and V3 were not able to verify this property in an hour. However, for the AIRPORT example, V1, V2, and V3 scale better than V4 and V5. Both for V3 and V5 the symbolic representation construction time takes a significant amount of time for problem instances with large number of boolean variables. 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.

8. Conclusions

In this work we have presented algorithms for constructing Finite Automata (FA) 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 discussed efficient storage techniques for the transition function of the FA and extended the construction algorithms to formulas on both boolean and integer variables. We also derived conditions which guarantee that the pre-condition computations used in symbolic verification algorithms do not cause an exponential increase in the FA size. Finally, we have tested the presented representations by using them to verify non-trivial systems 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. K. L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Massachusetts, 1993.
2. 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.
3. T. Bultan, R. Gerber, and W. 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, July 1999.
4. Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *Journal of Software Tools and Technology Transfer*, 3(3):250–270, 2001.
5. T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 169–179, July 2002.
6. 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.
7. 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.
8. 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.
9. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis, 1992.
10. 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.
11. P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proceedings of the Static Analysis Symposium*, September 1995.
12. Bernard Boigelot, Stéphane Rassart, and Pierre Wolper. On the expressiveness of

real and integer arithmetic automata. *Lecture Notes in Computer Science*, 1443, 1998.

13. 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.
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. 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.
16. The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.
17. The Omega project. `http://www.cs.umd.edu/projects/omega/`.
18. D. C. Cooper. Programs for mechanical program verification. In *Machine Intelligence 6*, pages 43–59, New York, 1971. American Elsevier.
19. R. Bryant. Graph-based algorithms for boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.
20. C. Bartzis and T. Bultan. Automata-based representations for arithmetic constraints in automated verification. In *Proceedings of the Seventh International Conference on Implementation and Application of Automata*, June 2002.
21. 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.
22. 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.
23. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.
24. G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.
25. T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–179, 2002.