

Efficient BDDs for bounded arithmetic constraints

Constantinos Bartzis, Tevfik Bultan

Department of Computer Science, University of California, Santa Barbara, CA 93106, USA
e-mail: bar@cs.ucsb.edu

Published online: 7 October 2005 – © Springer-Verlag 2005

Abstract. Symbolic model checkers use BDDs to represent arithmetic constraints over bounded integer variables. The size of such BDDs can in the worst case be exponential in the number and size (in bits) of the integer variables. In this paper we show how to construct linear-sized BDDs for linear integer arithmetic constraints. We present basic constructions for atomic equality and inequality constraints and generalize our complexity results for arbitrary linear arithmetic formulas. We also present three alternative ways of handling out-of-bounds transitions and discuss heterogeneous bounds on integer variables. We experimentally compare our approach to other BDD-based symbolic model checkers and demonstrate that the algorithms presented in this paper can be used to improve their performance significantly.

Keywords: BDD – Model checking – Integer arithmetic – SMV

1 Introduction

The performance of a symbolic model checker depends on the efficiency of the algorithms for the BDD construction and the sizes of the generated BDD representations. In this paper we address both these issues for linear arithmetic constraints on bounded integer variables. BDD-based model checkers represent bounded integer variables by mapping them to a set of Boolean variables using a binary encoding. Our experiments show that the state-of-the-art BDD-based model checkers [1–3, 12, 16] use inefficient algorithms for BDD construction

from linear arithmetic constraints and fail to generate compact BDD representations for them. Handling linear arithmetic constraints efficiently is an important problem since such constraints are common in reactive system specifications. For example, the distribution files for the BDD-based model checker NuSMV [2, 12] contain specifications with linear arithmetic constraints; however, the verification time for these specifications for NuSMV does not scale when the bounds on integer variables are increased. The algorithms and complexity results presented in this paper demonstrate that this inefficiency is not inherent to the BDD data structure and can be avoided.

We present algorithms for constructing efficient BDD representations from atomic arithmetic constraints of the form $\sum_{i=1}^v a_i \cdot x_i \# a_0$, where $\# \in \{=, \neq, >, \geq, \leq, <\}$. We show that the size of the resulting BDD is linear in the number of variables and the number of bits used to encode each variable. We also show that the time complexity of the construction algorithm is the same. We also give bounds for BDDs for linear arithmetic formulas that can be obtained by combining atomic arithmetic constraints with Boolean connectives. We show that the resulting BDDs for linear arithmetic formulas are still linear in the number of variables and the number of bits used to encode each variable.

We extend the construction algorithms to handle transitions that can take the bounded integer variables out of bounds. We present three different approaches for handling out-of-bounds transitions and show that all of them preserve our complexity results. We also generalize the construction algorithms to multiple bounds on integer variables. We show that as long as all the bounds are powers of two, the complexity results are preserved. One interesting result is that multiple bounds that are not powers of two cause the BDD size to be exponential in the number of variables in the worst case.

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

The preliminary results from this paper were presented in the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003) [6].

The rest of the paper is organized as follows. In Sect. 1.1 we discuss the related work. In Sects. 2 and 3 we give the BDD construction algorithm for atomic equality and inequality constraints, respectively, and prove its time and space complexity. In Sect. 4 we generalize our results for arbitrary linear arithmetic formulas consisting of atomic constraints and Boolean connectives. In Sects. 5–7 we discuss modifications to the basic construction algorithm in order to handle constraints that involve multiplication of variables, transitions that result in out-of-bounds errors, and the presence of distinct arbitrary bounds on integer variables, respectively. Finally, in Sect. 8 we present experimental results that demonstrate the advantages of our approach.

1.1 Related work

The problem of inefficient BDD representation of arithmetic constraints in symbolic model checkers has been pointed out in [11, 21]. In [11], the problem for SMV is handled by writing a preprocessor and fixing the BDD variable order. However, as we show in this paper, this extra step is not necessary since efficient BDDs can be directly constructed from a set of linear arithmetic constraints. In [21], the problem is solved only for constraints of the form $x + y = z$, where x , y , and z can be variables or constants. Even though such constraints arise very often in practice, our algorithms are more general without sacrificing efficiency.

Multiterminal BDDs (MTBDDs) [13], or Arithmetic Decision Diagrams (ADDs) [18], are structures similar to BDDs but with multiple terminal nodes and are used to represent functions with integer ranges. Given an equation or inequation, one can first construct ADDs for the right- and left-hand side of the (in)equation and then transform them to BDDs by matching terminal nodes. It is known that ADDs are very inefficient for representing functions with large ranges. In fact, this is the method used in NuSMV, and our experiments demonstrate its inefficiency.

Binary Moment Diagrams (BMDs) [9] and Hybrid Decision Diagrams (HDDs) [14] are data structures designed to represent arithmetic expressions and handle arithmetic operations in word-level verification where an array of binary bits can be referred to as an integer variable. These data structures can also be used to construct linear-sized BDDs from linear arithmetic constraints. However, in this paper, we show that one can construct linear-sized BDDs from linear arithmetic constraints directly, without using these data structures. Hence, the algorithms we present can be easily integrated into a BDD-based model checker.

The problem of constructing finite-state automata to represent linear arithmetic constraints on unbounded integer variables has been studied in [5, 7, 20]. We use similar ideas to construct BDDs for constraints on bounded variables. In fact, BDDs can be seen as finite-state auto-

mata with a binary alphabet, whose only accepting state is the terminal node **1**.

2 Atomic equality constraints

In this section we discuss the construction of BDDs for atomic equality constraints on bounded integer variables. Given a set of v integer variables x_i , $1 \leq i \leq v$ such that $0 \leq x_i < 2^b$ and a linear equation of the form

$$\sum_{i=1}^v a_i \cdot x_i = a_0,$$

we construct a BDD with $v \cdot b$ Boolean variables $x_{i,j}$, $1 \leq i \leq v$, $0 \leq j < b$, which evaluates to **1** iff

$$\sum_{i=1}^v a_i \cdot \left(\sum_{j=0}^{b-1} x_{i,j} \cdot 2^j \right) = a_0.$$

In other words, the BDD variables $x_{i,j}$ represent the binary digits of the integer variables and the BDD evaluates to **1** iff the equation is satisfied by the valuation $x_i = \sum_{j=0}^{b-1} x_{i,j} \cdot 2^j$ for $1 \leq i \leq v$. We show that such a BDD has $O(v \cdot b \cdot \sum_{i=1}^v |a_i|)$ nodes, i.e., the size of the BDD is linear in the number of Boolean variables. Note that in general the size of a BDD can be exponential in the number of Boolean variables, and experimental results show

BDD construction algorithm for equations $\sum_{i=1}^v a_i \cdot x_i = a_0$
on b -bit variables

```

1 Procedure: node( $i, j, c$ )
  //Constructs a node in level  $i$  of layer  $j$  with label  $c$ 
2 BDDnode  $n$ 
3    $n.index := j \cdot v + i$ 
4   if ( $i = v$  and  $j = b - 1$ ) //Very last level
     //Accept iff final carry is 0
5     if ( $c = 0$ )  $n.low := 1$ 
6     else  $n.low := 0$ 
7     if ( $c + a_v = 0$ )  $n.high := 1$ 
8     else  $n.high := 0$ 
9   else if ( $i = v$ ) //Last level of layer  $j$ 
     //Compute carry or reject
10    if ( $c$  is even)  $n.low := node(1, j + 1, c/2)$ 
11    else  $n.low := 0$ 
12    if ( $c + a_v$  is even)
        $n.high := node(1, j + 1, (c + a_v)/2)$ 
13    else  $n.high := 0$ 
14  else //All other levels. Compute intermediate result
15     $n.low := node(i + 1, j, c)$ 
16     $n.high := node(i + 1, j, c + a_v)$ 
17  return  $n$ 

```

18 Main: return node($1, 0, -a_0$)

Fig. 1. BDD construction algorithm for equations

that state-of-the-art model checkers often produce exponentially large BDDs.

The construction algorithm is given in Fig. 1. The constructed BDD consists of b layers of v levels each. The j th layer corresponds to the j th least significant bit of each integer variable and the i th level in a layer corresponds to the i th integer variable. Every node in a level is labeled with an integer c between $-\sum_{i=0}^v |a_i|$ and $\sum_{i=0}^v |a_i|$. In particular, the label of a node in the first level of the j th layer corresponds to a value of the carry c resulting from the computation of the expression

$$\sum_{i=1}^v a_i \cdot \left(\sum_{n=0}^{j-1} x_{i,j} \cdot 2^n \right) - a_0,$$

where $x_{i,j}$ s are the values of the BDD variables along one of the paths from the root to that node. Furthermore, the label of a node in the k th level, $2 \leq k \leq v$, of the j th layer is the value $c + \sum_{i=1}^{k-1} a_i \cdot x_{i,j}$, where $x_{i,j}$ s are the values of the BDD variables along one of the paths from the node in the first level of the j th layer with label c to that node.

As an example consider the linear equation $2x - 3y = 1$. Figure 2 shows the structure of a complete intermediate layer (inside the dashed rectangle) of the corresponding BDD, regardless of the length of x and y . The nodes outside the rectangle comprise the first level of the next layer. The complete BDD, when x and y are 4 bits long, is shown in Fig. 3. For all figures, edges not shown point to the 0 terminal node. Note that the BDDs constructed by the algorithms in this paper are not necessarily reduced. Standard BDD reduction needs to be applied after the construction. It is known that BDD reduction can be done in linear time [19].

Theorem 1. *The algorithm given in Fig. 1 constructs a BDD representing the linear equation $\sum_{i=1}^v a_i \cdot x_i = a_0$ on b -bit nonnegative integer variables. The time complexity of the algorithm and the size of the resulting BDD is $O(v \cdot b \cdot \sum_{i=1}^v |a_i|)$.*

Proof. For the purposes of the proof we can think of a BDD as a bit-serial processor as described in [8]. Such a processor computes a Boolean function by examining the arguments x_1, x_2 , and so on in order, producing output 0 or 1 after the last bit has been read. It requires internal storage to store enough information about the

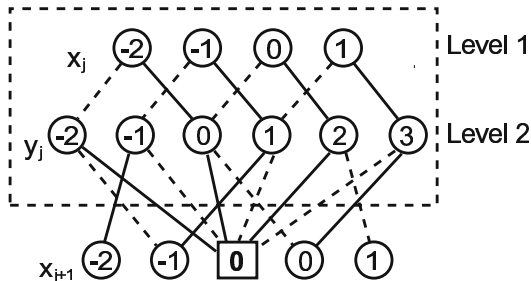


Fig. 2. Layer of a BDD for $2x - 3y = 1$

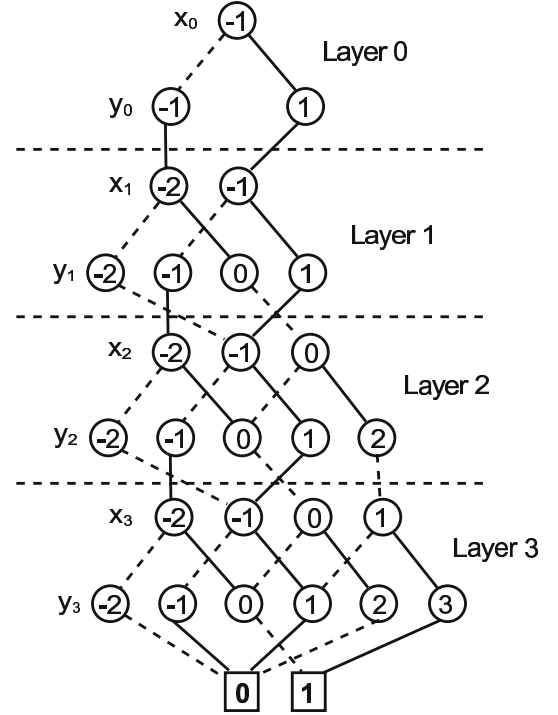


Fig. 3. BDD for $2x - 3y = 1$ for 4-bit variables

arguments it has already seen to correctly deduce the value of the function from the values of the remaining arguments. Trivially it can store all the values of the arguments it has already seen by using exponentially large storage. In our case we can show that linear storage is needed. The size of the storage consumed by the processor translates to the number of nodes in the BDD.

The ordering of the Boolean variables $x_{i,j}$ is lexicographical primarily on j and secondarily on i , or, equivalently, the index of variable $x_{i,j}$ is $j \cdot v + i$. The index of the root is 1. One can easily verify that any internal node with index $index$ points to a node with index $index + 1$, except for the nodes with index $b \cdot v$ that point to the terminal nodes. Thus the constructed BDD is consistent with the ordering mentioned above. The bit-serial processor corresponding to the BDD first processes the least significant bit of the integer variables x_1, x_2, \dots, x_v in this order, then it processes the second least significant bits, and so on. In the end the processor needs to verify whether or not $\sum_{i=1}^v a_i \cdot x_i = a_0$ or, equivalently, $\sum_{i=1}^v a_i \cdot \left(\sum_{j=0}^{b-1} x_{i,j} \cdot 2^j \right) = a_0$, or

$$-a_0 + \sum_{j=0}^{b-1} 2^j \cdot \left(\sum_{i=1}^v a_i \cdot x_{i,j} \right) = 0. \quad (1)$$

To accomplish this the processor gradually computes the left-hand side of (1) bit by bit and compares it against zero. If at any point the comparison fails, it immediately evaluates to 0; otherwise it continues. It starts with an initial value of $-a_0$ and then gradually adds to it

$a_i \cdot x_{i,0}$ as it reads the values of $x_{1,0}$ up to $x_{v,0}$ and stores the intermediate result $-a_0 + \sum_{i=1}^l a_i \cdot x_{i,0}$ every time (lines 15–16 of the algorithm). Note that the value stored is shown as the label of each BDD node in the figures. At the end of processing layer 0, if the result is an odd number (i.e., the resulting bit is 1), the processor immediately evaluates to 0, since what remains to add, namely, $\sum_{j=1}^{b-1} 2^j \cdot (\sum_{i=1}^v a_i \cdot x_{i,j})$, is an even number; therefore, the final result cannot be zero. Otherwise, the intermediate result at this point divided by 2 is equal to the remaining carry c , i.e., $c = (-a_0 + \sum_{i=1}^v a_i \cdot x_{i,0}) / 2$. The value of the carry is the only piece of information that needs to be stored at this point (lines 9–13). If we divide both sides of (1) by 2, we will get:

$$c + \sum_{j=1}^{b-1} 2^{j-1} \cdot \left(\sum_{i=1}^v a_i \cdot x_{i,j} \right) = 0. \quad (2)$$

Now the processor needs to verify (2), and this task is similar to the initial one, so the processor continues to operate in a similar manner. In the end, in order for the final result to be 0, the final carry also has to be 0. In that case the processor evaluates to 1; otherwise it evaluates to 0 (lines 4–8). This concludes the proof of correctness of our construction algorithm.

For the proof of termination and complexity, the fundamental question that needs to be answered is how many different intermediate results need to be stored at any point during the operation of the bit-serial processor or, in other words, how many BDD nodes there are at any level. The number of nodes at any level is bounded by the size of the range defined by the least and the greatest label in that level. If the labels of the nodes at level $j \cdot v + 1$ belong to a range of size $n_{1,j}$, then level $j \cdot v + 2$ has at most $n_{2,j} = n_{1,j} + |a_1|$ nodes, level $j \cdot v + 3$ has at most $n_{3,j} = n_{1,j} + |a_1| + |a_2|$ nodes, and so on. Finally, level $j \cdot v + v + 1 = (j+1) \cdot v + 1$ has at most $n_{1,j+1} = (n_{1,j} + \sum_{k=1}^v |a_k|) / 2$ nodes because that many are the different values of the carry that need to be stored, as described earlier. Initially $n_{1,0} = 1$, and by induction one can prove that no $n_{1,j}$ is larger than $\sum_{k=1}^v |a_k|$. As mentioned above, $n_{2,j} = n_{1,j} + |a_1|$. Therefore, all $n_{2,j}$ are at most $\sum_{k=1}^v |a_k| + |a_1|$, and in general $n_{i,j} \leq \sum_{k=1}^v |a_k| + \sum_{k=1}^{i-1} |a_k|$. In total, the number of nodes in layer j is at most

$$\sum_{i=1}^v n_{i,j} = \sum_{i=1}^v |a_i| \cdot (2v - i). \quad (3)$$

There are b layers, so the total number of nodes in the BDD is at most $b \cdot \sum_{i=1}^v |a_i| \cdot (2v - i)$ or $O(v \cdot b \cdot \sum_{i=1}^v |a_i|)$, i.e., the size of the constructed BDD is linear on both v and b . Each node is created once if we store each of them in a hash table indexed by i, j , and c , and the creation of a node requires a fixed amount of work, so the complexity of our algorithm is $O(v \cdot b \cdot \sum_{i=1}^v |a_i|)$.

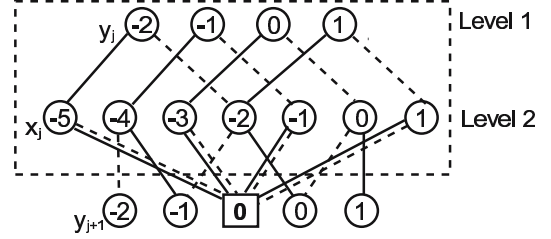


Fig. 4. Layer of a BDD for $2x - 3y = 1$, when the order of x and y is reversed

The bound on the number of nodes in any layer shown in (3) leads us to another interesting conclusion. The size of the constructed BDD is minimized if the integer variables are ordered in increasing order of the absolute values of their coefficients. For example, consider the BDD for the equation $2x - 3y = 1$, a layer of which is shown in Fig. 2. If we change the ordering of the variables so that y_i appears before x_i , each layer will have one more node, as shown in Fig. 4.

3 Atomic inequality constraints

Next, we show how to construct BDDs for inequations of the form

$$\sum_{i=1}^v a_i \cdot x_i < a_0, \quad 0 \leq x_i < 2^b.$$

BDD construction algorithm for inequations $\sum_{i=1}^v a_i \cdot x_i < a_0$ on b -bit variables

```

1 Procedure: node( $i, j, c$ )
  //Constructs a node in level  $i$  of layer  $j$  with label  $c$ 
2 BDDnode  $n$ 
3    $n.index := j \cdot v + i$ 
4   if ( $i = v$  and  $j = b - 1$ ) //Very last level
     //Accept iff final carry is  $< 0$ 
5     if ( $c < 0$ )  $n.low := 1$ 
6     else  $n.low := 0$ 
7     if ( $c + a_v < 0$ )  $n.high := 1$ 
8     else  $n.high := 0$ 
9   else if ( $i = v$ ) //Last level of layer  $j$ . Compute carry
10    if ( $c$  is even)  $n.low := node(1, j + 1, c/2)$ 
11    else  $n.low := node(1, j + 1, (c - 1)/2)$ 
12    if ( $c + a_v$  is even)
        $n.high := node(1, j + 1, (c + a_v)/2)$ 
13    else  $n.high := node(1, j + 1, (c + a_v - 1)/2)$ 
14   else //All other levels. Compute intermediate result
15      $n.low := node(i + 1, j, c)$ 
16      $n.high := node(i + 1, j, c + a_v)$ 
17   return  $n$ 

```

18 Main: return node($1, 0, -a_0$)

Fig. 5. BDD construction algorithm for inequations

Note that we can transform all other kinds of linear inequations ($\leq, >, \geq$) to this form by changing the signs of the coefficients and/or adding 1 to the constant term a_0 . The algorithm is similar to the one for equations and is shown in Fig. 5. There are only two differences. First, after having processed an equal number of bits from all integer variables (lines 9–13 of the algorithm), we do not require the resulting bit to be 0. The bit-serial processor only computes the correct value of the remaining carry and proceeds to the next level. Second, in order for the inequality to hold after all bits have been processed (lines 4–8 of the algorithm), the remaining carry has to be negative. To see the correctness of the algorithm, let r be the b -bit number formed by the resulting bits and c be the final carry. Then $-a_0 + \sum_{i=1}^v a_i \cdot x_i = 2^b \cdot c + r$ and $0 \leq r < 2^b$. It follows that $\sum_{i=1}^v a_i \cdot x_i < a_0 \Leftrightarrow c < 0$. Obviously, these two modifications do not change the bound on the number of nodes in the BDD, which is again $O(v \cdot b \cdot \sum_{i=1}^v |a_i|)$. This proves the following theorem.

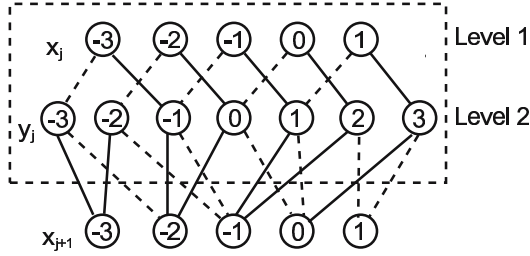


Fig. 6. Layer of a BDD for $2x - 3y < 1$

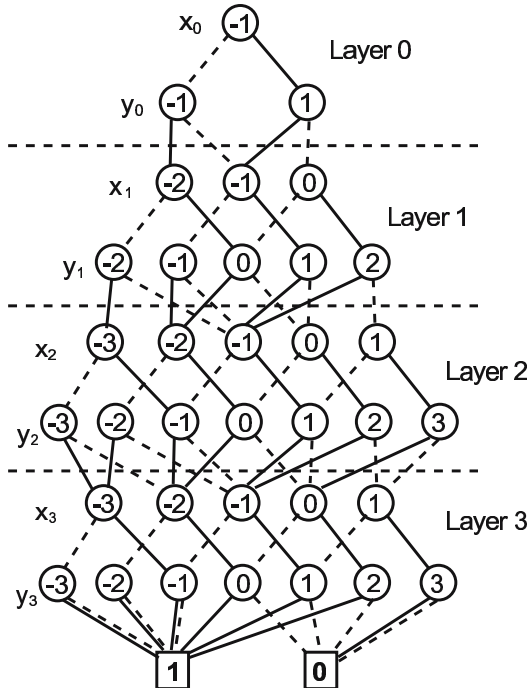


Fig. 7. BDD for $2x - 3y < 1$ for 4-bit variables

Theorem 2. *The algorithm given in Fig. 5 constructs a BDD representing the linear inequation $\sum_{i=1}^v a_i \cdot x_i < a_0$ on b -bit nonnegative integer variables. The time complexity of the algorithm and the size of the resulting BDD is $O(v \cdot b \cdot \sum_{i=1}^v |a_i|)$.*

As an example consider the linear inequation $2x - 3y < 1$, where x and y are 4 bits long. Figures 6 and 7 show the structure of an intermediate layer and the complete BDD before being reduced.

4 Linear arithmetic formulas

In symbolic model checking BDDs are subjected to operations such as intersection, union, negation, etc., as well as subsumption and equivalence tests. The time and space complexity of these operations depends on the size of the operands. The complexity of negation is $O(1)$, as it involves only swapping the terminal nodes 0 and 1, but the complexity of intersection and union, which are frequently used operations in symbolic model checking, is $O(n_1 \cdot n_2)$, where n_1 and n_2 are the sizes of the operands.

Suppose that one performs an intersection or union operation on two BDDs representing the constraints $\sum_{i=1}^v a_i \cdot x_i = a_0$ and $\sum_{i=1}^v b_i \cdot x_i = b_0$ whose sizes are $O(v \cdot b \cdot \sum_{i=1}^v |a_i|)$ and $O(v \cdot b \cdot \sum_{i=1}^v |b_i|)$, respectively, as proved earlier. One would expect the size of the resulting BDD to be $O(v^2 \cdot b^2 \cdot \sum_{i=1}^v |a_i| \cdot \sum_{i=1}^v |b_i|)$. Actually, this is a pessimistic estimation. The resulting BDD will have again $v \cdot b$ layers, corresponding to a bit-serial processor that examines each of the $x_{i,j}$ s one by one as before. The only difference is that now it needs to remember the intermediate results from both BDDs and thus every layer will have $O(\sum_{i=1}^v |a_i| \cdot \sum_{i=1}^v |b_i|)$ nodes and there will be $O(v \cdot b \cdot \sum_{i=1}^v |a_i| \cdot \sum_{i=1}^v |b_i|)$ nodes in total. Clearly the same argument holds for more than two linear constraints, which proves the following theorem.

Theorem 3. *Given a linear arithmetic formula on b -bit nonnegative integer variables consisting of n atomic constraints of the form $\sum_{i=1}^v a_{i,j} \cdot x_i = a_{0,j}$ or $\sum_{i=1}^v a_{i,j} \cdot x_i < a_{0,j}$, $1 \leq j \leq n$ and Boolean connectives \neg, \wedge, \vee , one can construct a BDD of size $O(v \cdot b \cdot \prod_{j=1}^n \sum_{i=1}^v |a_{i,j}|)$ representing the formula in time $O(v \cdot b \cdot \prod_{j=1}^n \sum_{i=1}^v |a_{i,j}|)$.*

The conclusion is that when basic operations are performed by a model checker on BDDs representing linear arithmetic constraints on bounded integers, the size of intermediate BDD representations remains linear in the number and size of the integer variables, i.e., the space and time complexity of operations does not blow up with respect to these two parameters. This is very important since such “blowups” are a common drawback of BDD-based model checking.

It is known that satisfiability checking for BDDs can be performed in constant time. Given an instance

of integer programming concerning the satisfiability of a conjunction of n linear constraints on v b -bit integer variables, one can solve the problem in the following manner. First, construct a BDD for those constraints by using our construction algorithm in time $O\left(v \cdot b \cdot \prod_{j=1}^n \sum_{i=1}^v |a_{i,j}|\right)$. Then check for satisfiability in constant time. Hence, such instances of integer programming are solvable in time polynomial in v , b , and $\max(|a_{i,j}|)$ but exponential in n . Note that integer programming is NP-complete in the strong sense even if $b = 1$ [15], which implies that there is no algorithm that is polynomial in v , b , and n , unless $P = NP$. On the other hand, in [17] it is shown that if n is fixed, then a pseudopolynomial algorithm exists. Our complexity results serve as an alternative proof for the same fact.

5 Handling multiplication

An inherently unavoidable shortcoming of BDDs is their inability to efficiently represent arithmetic constraints involving multiplication between variables. In [8] it was proved that the size of such BDDs has a lower bound exponential on b , the size of the integer variables, regardless of the variable ordering.

The good news is that, by choosing the variable ordering we defined earlier and by slightly modifying our construction algorithm one can accommodate multiplication and keep the size of the produced BDD exponential only in b and the number of integer variables involved in multiplications, which is in many cases less than v , the total number of integer variables.

The idea supporting this argument is the following. Suppose we want to construct a BDD for an arithmetic formula on $v = m + l$ integer variables, in which only m variables are multiplied with other variables in the formula (which we will call m -variables) and the remaining l variables (which we will call l -variables) are only multiplied with constants, forming the “linear part” of the formula. In the worst case, the bit-serial processor corresponding to the BDD will need to remember the exact values of the m -variables and the intermediate results c of the computation of the “linear part”, as described earlier. Therefore, each node is now labeled by an $(m + 1)$ -tuple containing the current values of the m -variables and c . The number of levels remains the same $v \cdot b$. At any level, when an m -variable is processed, all nodes are doubled in the next level, thus remembering the new bit of the m -variable, and the various c s are propagated properly. When an l -variable is processed, the processor behaves exactly as in the linear case. At the end of layer k , the processor knows the first k bits of the m -variables and the carry from the “linear part” and is able to compute the corresponding bit of the result and the carry for the next layer. The number of nodes is doubled $m \cdot b$ times, and consequently the size of the BDD will be $O\left(2^{m \cdot b} \cdot l \cdot b \cdot \sum_{i=1}^v |a_i|\right)$. Of course, this is still an exponential bound but nevertheless indicates a complexity that

is exponentially dependent on m and not v . In many practical cases, if m is nonzero, it is at least much less than v . Note that by choosing a different variable ordering one can end up with BDDs of exponential size in both b and v .

6 Handling overflows

When constructing BDDs to represent the transition relation of a system, special care is needed in order to handle possible overflows. For example, consider a transition labeled by $x' = x + 1$, where x is the current state variable and x' is the next state variable. They are both 2-bit non-negative integers ranging between 0 and 3. When $x = 3$ and the transition is taken, what is an appropriate value of x in the next state, since it cannot be 4? We consider three alternatives:

1. The transition cannot be taken, i.e., there is no next state.
2. Modular arithmetic is performed and $x = 0$ in the next state.
3. An “out-of-bounds” error is detected and reported.

BDD construction for the transition relation depends on the choice of one of these three alternative approaches. For our example, an intermediate layer and the complete BDDs for all three approaches are shown in Fig. 8. The construction algorithm described earlier follows the first approach.

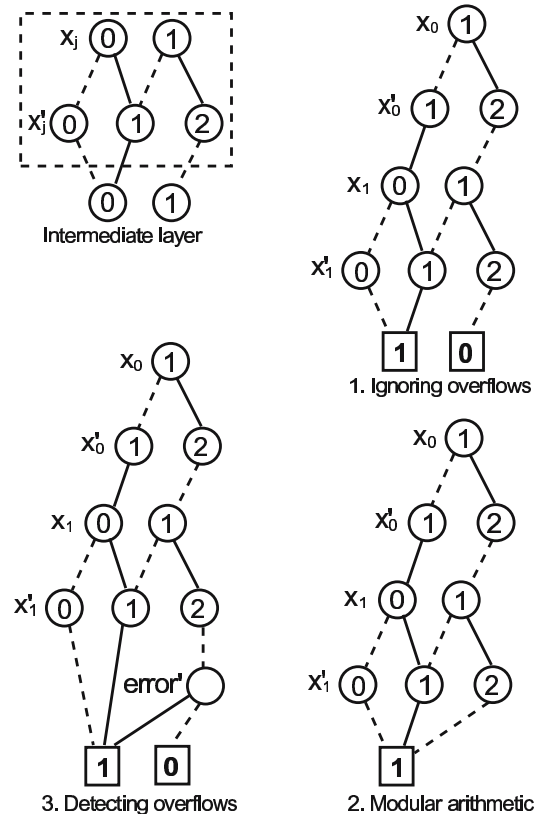


Fig. 8. Alternative ways to handle overflows

1. Ignoring overflows	
4	if ($i = v$ and $j = b - 1$)
5	if ($c = 0$) $n.low := 1$
6	else $n.low := 0$
7	if ($c + a_v = 0$) $n.high := 1$
8	else $n.high := 0$
2. Modular Arithmetic	
4	if ($i = v$ and $j = b - 1$)
5	if (c is even) $n.low := 1$
6	else $n.low := 0$
7	if ($c + a_v$ is even) $n.high := 1$
8	else $n.high := 0$
3. Detecting Overflows	
4	if ($i = v$ and $j = b - 1$)
5	if ($c = 0$) $n.low := 1$
	else if (c is even) $n.low = errornode$
6	else $n.low := 0$
7	if ($c + a_v = 0$) $n.high := 1$
	else if ($c + a_v$ is even) $n.high = errornode$
8	else $n.high := 0$

Fig. 9. Modifications of the BDD construction algorithm in order to handle overflows

The difference between the three approaches is in the edges generated by lines 4–8 of the construction algorithm in Fig. 1, which correspond to the case where the b least significant bits of the variables (in our example $b = 2$) satisfy the equation but there is a remaining nonzero carry indicating an overflow. Figure 9 shows the modifications to the algorithm for each of the three approaches. The first approach points all such edges to the **0** terminal node, thus making the BDD evaluate to **0** whenever an overflow occurs. The second approach points all such edges to **1**, thus ignoring overflows and performing modular arithmetic. The third approach is a bit more involved. There is an extra global Boolean variable *error* at the end of the variable ordering. All edges that indicate an overflow point to a node with the *index* of *error'* and *low* = **0** and *high* = **1**. Initially *error* is false. When an overflow occurs, *error* will become true in the next state. Note that for all three approaches Theorems 1, 2, and 3 still hold. The three alternative BDDs shown in Fig. 8 correspond to the following constraints, respectively:

1. $x < 3 \wedge x' = x + 1$
2. $x' = (x + 1) \bmod 4$
3. $x' = (x + 1) \bmod 4 \wedge ((x + 1) > 3 \Rightarrow error' = true)$

In all versions of SMV [1–3], out-of-bounds errors are checked statically. This means that even out-of-bound transitions that are not reachable are reported as out-

of-bounds errors. By using alternative 3 presented above one can check if an out-of-bounds error is reachable and report an out-of-bounds error only when one occurs on some execution path. One can also implement the static out-of-bounds error check used in SMV by reporting a potential out-of-bounds error if a node with the *index* of the Boolean variable *error'* appears in the transition relation BDD.

When overflows are not considered errors, one can use alternative 1 or 2, depending on the nature of the system being modeled. For example, when creating a finite-state model of an infinite-state system by restricting unbounded variables to finite domains, alternative 1 is the most appropriate one.

7 Handling multiple bounds on variables

So far we have studied the construction of BDDs for linear arithmetic constraints on v integer variables x_i , $1 \leq i \leq v$ such that $0 \leq x_i < 2^b$, i.e., all variables were nonnegative and bounded by the same power of two. Now consider the case where each variable x_i has its own bounds $l_i \leq x_i < h_i$, where l_i and h_i are (possibly negative) integer constants that are not necessarily powers of 2. As a first step we can eliminate the lower bounds by replacing every variable x_i in every constraint by the variable $X_i = x_i - l_i$. Now, any constraint of the form $\sum_{i=1}^v a_i \cdot x_i \# a_0$, where $\# \in \{=, \neq, >, \geq, \leq, <\}$, becomes $\sum_{i=1}^v a_i \cdot X_i \# a_0 - \sum_{i=1}^v a_i \cdot l_i$ and $0 \leq X_i < h_i - l_i = d_i$. Now the lower bound of all variables is again 0 as it was initially, but the upper bounds are different and not necessarily powers of two.

Here we will show how to construct BDDs for equations of the form $\sum_{i=1}^v a_i \cdot x_i = a_0$, where $0 \leq x_i < d_i$ for $1 \leq i \leq v$. The construction of BDDs for inequations is similar. Since there are extra constraints $0 \leq x_i < d_i$ that have to be satisfied in order for the BDD to evaluate to **1**, extra information has to be “stored” in the nodes: the valid range for the part of every variable that has not yet been processed. Since the lower bound for every variable is 0, only the upper bound needs to be stored. At the root node, the upper bound for each variable x_i is d_i . After the least significant bit $x_{i,0}$ of variable x_i has been processed, the upper bound for the rest of x_i (i.e., the value of x_i with the least significant bit removed) becomes $\lceil (d_i - x_{i,0})/2 \rceil$. In general, if the upper bound u for x_i at a node n in level i of layer j is d , then at $n.low$ $u = \lceil d/2 \rceil$ and at $n.high$ $u = \lceil (d-1)/2 \rceil$. As an example, consider the equation $2x - 3y = 1$, where $0 \leq x < 11$ and $0 \leq y < 13$.

Figure 10 shows the bound information for x and y as described earlier. Figure 11 shows the complete BDD for the equation.

We can prove that at any level there are at most two different upper bounds, which differ by one, for each variable. Initially, there is only one bound for each variable. If at some level the two different upper bounds

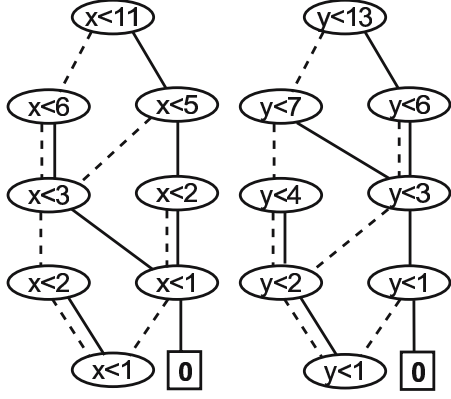


Fig. 10. Bounds information for $0 \leq x < 11$ and $0 \leq y < 13$

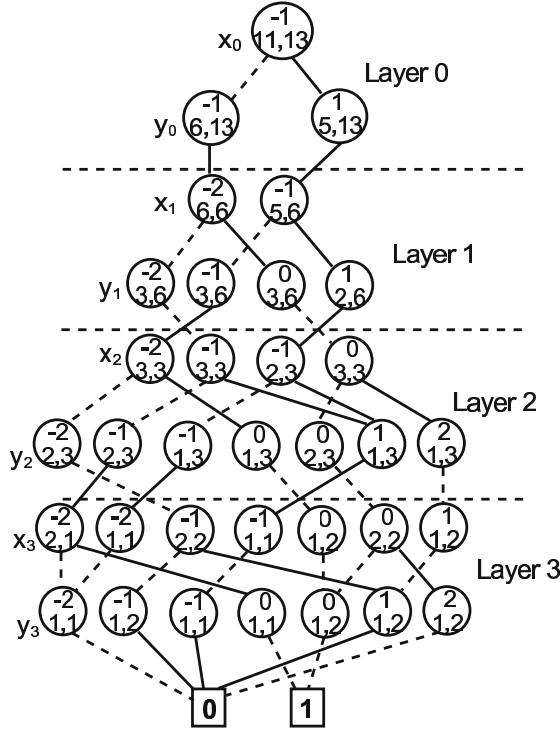


Fig. 11. BDD for $2x - 3y = 1$ when $0 \leq x < 11$ and $0 \leq y < 13$

for a variable are d and $d+1$, then in the next layer those bounds will become $\lceil (d-1)/2 \rceil$ and $\lceil (d+1)/2 \rceil$. In general, there can be at most 2^v different combinations of bounds for all v variables at any level. The maximum number of layers is $\log_2(\max(d_i))$. Consequently, the size of the BDD representing the equation is $O(v \cdot \log_2(\max(d_i)) \cdot 2^v \cdot \sum_{i=1}^v |a_i|)$, which is exponential in the number of variables v . Remember that when the upper bounds are powers of two, 2^b , the size of the BDD is only $O(v \cdot b \cdot \sum_{i=1}^v |a_i|)$. Interestingly, this indicates that when modeling a system and the choice of bounds for the integer variables is independent of the input specification, it is better to choose bounds that are powers of 2.

An equivalent way to handle multiple variable bounds would be to consider that all variables are bounded by the same power of 2, $2^{\lceil \log_2(\max(d_i)) \rceil}$, and conjoin the given

linear equation with the constraints $0 \leq x_i < d_i$. There are $v+1$ atomic constraints now and, as shown in Sect. 4, the size of a BDD is exponential in the number of atomic constraints in the worst case. However, in the trivial case where there is no arithmetic but only constraints of the form $l_i \leq x_i < h_i$, the variable ordering used in SMV (bits of the same variable are grouped together) yields BDDs of linear size.

8 Experimental results

We integrated our construction algorithms to the Composite Symbolic Library and the Action Language Verifier [10, 22]. The Action Language Verifier is an infinite-state CTL model checker, and it uses Composite Symbolic Library as its symbolic manipulator. We created a new version of the Action Language Verifier (ALV) by using BDDs as symbolic representations for bounded integers and integrating our BDD construction algorithms for linear arithmetic constraints.

We experimented with two specification examples, *Bakery* and *Barber*, from the ALV distribution, which is available at: <http://www.cs.ucsb.edu/~bultan/composite/>. *Bakery* is a mutual exclusion protocol for two processes. *Barber* is a monitor specification for the Sleeping Barber problem from [4]. We also verified three

Table 1. Bakery

Bits	CMU SMV	Cadence SMV	NuSMV	ALV
4	0.04	0.17	0.07	0.17
5	0.23	0.27	0.26	0.17
6	1.27	0.5	1.71	0.17
7	9.37	1.39	20.52	0.18
8	78.87	6.12	142.82	0.18
9	673.11	21.67	1186.45	0.18
10	↑	84.1	↑	0.19
11	↑	329.93	↑	0.19
12	↑	1503.83	↑	0.19
100	↑	↑	↑	0.31

Table 2. Barber

Bits	CMU SMV	Cadence SMV	NuSMV	ALV
4	0.15	0.36	0.3	0.23
5	0.46	0.86	1.09	0.25
6	2.03	2.97	13.47	0.29
7	14.14	10.42	1185.92	0.3
8	274.89	38.29	↑	0.35
9	↑	157.58	↑	0.39
10	↑	721.25	↑	0.42
11	↑	↑	↑	0.44
12	↑	↑	↑	0.48
100	↑	↑	↑	5.12

Table 3. Alternating bit protocol. The property checked is independent of the integer data field. Since Cadence SMV [1] can detect and verify data-independent properties efficiently, we also verified a data-dependent property. The verification time for the data-dependent property stays almost the same for the Action Language Verifier

Bits	CMU SMV	Cadence SMV		NuSMV	ALV	
		Data independent	Data dependent		Data independent	Data dependent
4	0.12	0.21	3.91	2.69	0.23	0.24
5	0.26	0.2	7.56	2.71	0.23	0.25
6	1.26	0.23	24.11	2.63	0.24	0.24
7	30.24	0.18	84.33	3.05	0.23	0.26
8	147.96	0.19	343.47	3.61	0.23	0.24
9	693.67	0.2	↑	6.6	0.25	0.26
10	3755.46	0.24	↑	24.12	0.23	0.27
11	↑	0.27	↑	87.62	0.27	0.27
12	↑	0.36	↑	342.89	0.24	0.27
100	↑	↑	↑	↑	0.69	0.72

specification examples, **abp**, **p-queue**, and **prod-cons**, included in the NuSMV distribution [2]. Example **abp** is an alternating bit protocol, and **p-queue** and **prod-cons** are two different implementations of a buffer where data are inserted, sorted, and consumed. We were able to verify safety and liveness properties for these examples. We run these experiments using three different implementations of the SMV [1–3], namely, CMU SMV (version 2.5.4.3), Cadence SMV (version 08-20-01p2), and NuSMV (version 2). We obtained the experimental results on a SUN ULTRA 10 workstation with 768 MB of memory, running SunOs 5.7.

We measured the time required to verify each of the examples for different sizes of the integer variables from 4 bits to 100 bits. The results are shown in Tables 1–5. Entries ↑ signify that the corresponding experiment did not finish in 4000 s. It is clear that for all current imple-

mentations of SMV, the recorded times are exponential in the size of the integer variables, while for ALV, which uses the construction algorithms presented earlier, the recorded times are linear in the size of the integer variables. Note that ALV is not a BDD-based model checker, hence SMV may be better optimized for BDD-based verification. However, our point is that the advantages of our construction algorithms can be exploited by integrating them to any BDD-based model checker.

Figure 12 illustrates the effect of arbitrary bounds on variables as described in Sect. 7. We used our construction algorithm to build a BDD for the equation $x_1 + x_2 - x_3 - x_4 + x_5 - x_6 = 7$, where $0 \leq x_1, x_2, x_3, x_4, x_5, x_6 < 2^8$ and recorded the size of the resulting reduced BDD. Then we gradually changed the upper bound of each of the variables to some arbitrary unique value less than 2^8 and recorded the size of the resulting reduced BDD each time. The results shown in Fig. 12 demonstrate the exponential growth of the size of the BDD described in Sect. 7.

We performed the same experiment, this time using the BDD variable ordering used in SMV, where all bits of the same integer variable are next to each other. The re-

Table 4. Queue

Bits	CMU SMV	Cadence SMV	NuSMV	ALV
4	0.3	0.57	0.33	3.21
5	1.75	1.23	1.21	5.63
6	24.47	5.37	12.07	8.14
7	2159.69	38.8	122.3	10.77
8	↑	318.39	1125.34	13.06
100	↑	↑	↑	440.87

Table 5. Producer – Consumer

Bits	CMU SMV	Cadence SMV	NuSMV	ALV
2	5.49	4.61	23.27	210.44
3	216.94	73.98	3264.97	698.93
4	↑	1430.54	↑	2600
5	↑	↑	↑	6062.34

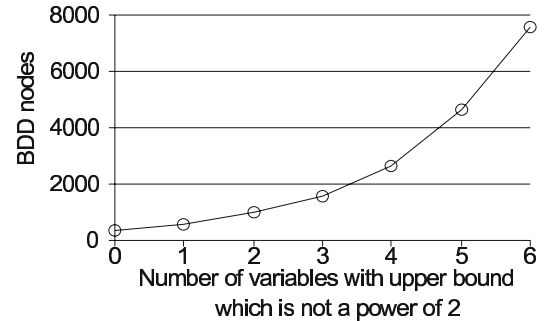


Fig. 12. Size of the BDD for $x_1 + x_2 - x_3 - x_4 + x_5 - x_6 = 7$ vs. the number of variables with upper bound different than 2^8 , using the variable ordering described in this paper

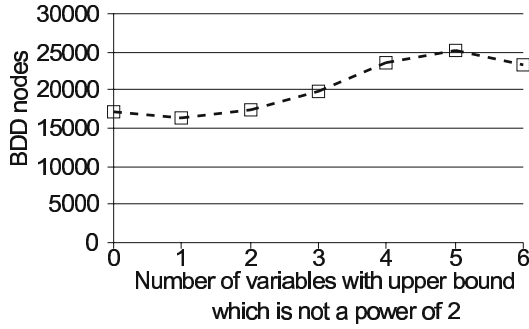


Fig. 13. Size of the BDD for $x_1 + x_2 - x_3 - x_4 + x_5 - x_6 = 7$ vs. the number of variables with upper bound different than 2^8 , using the variable ordering used in SMV

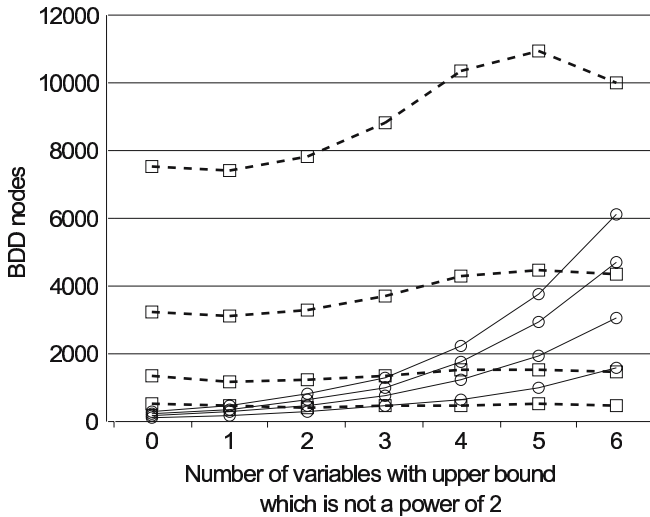


Fig. 14. Size of the BDD for $x_1 + x_2 - x_3 - x_4 + x_5 - x_6 = 7$ vs. the number of variables with upper bound different than 2^b , for different lengths of variables $b = 4, 5, 6, 7$, using our algorithms (solid lines) and SMV (dotted lines)

sults are shown in Fig. 13. Interestingly, for this ordering the number of BDD nodes does not grow exponentially with respect to the number of variables with arbitrary bounds. However, the lowest value in Fig. 13 is more than double the maximum value in Fig. 12. Note that the approach used in SMV produces BDDs with size exponential in b , the number of bits of each integer variable. To illustrate the effect of both increasing the length of the integer variables and the presence of arbitrary bounds, for both approaches, we repeated the same experiment with $b = 4, 5, 6$, and 7 . The results are shown in Fig. 14. Solid lines correspond to our method, whereas dotted lines correspond to SMV.

9 Conclusions

In this paper we have shown experimentally that current implementations of BDD-based symbolic model checkers are inefficient in representing linear arithmetic con-

straints on bounded integer variables. We solved this problem using efficient BDD construction algorithms, proving their complexity and experimentally demonstrating their efficiency. These algorithms can be used to improve the performance of existing BDD-based symbolic model checkers. Finally, we have shown that powers of 2 are a good choice for variable bounds, and choosing arbitrary bounds for integer variables can cause exponential blowup in the BDD size.

References

1. Cadence SMV. <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv>
2. NuSMV. <http://nusmv.irst.itc.it/>
3. SMV. www.cs.cmu.edu/~modelcheck/smv.html
4. Andrews GR (1991) Concurrent programming: principles and practice. Benjamin/Cummings, Redwood City, CA
5. Bartzis C, Bultan T (2003) Efficient symbolic representations for arithmetic constraints in verification. *Int J Found Comput Sci* 14(4):605–624
6. Bartzis C, Bultan T (2003) Construction of efficient BDDs for bounded arithmetic constraints. In: *Proceedings of the 9th international conference on tools and algorithms for the construction and analysis of systems. Lecture notes in computer science*, vol 2619. Springer, Berlin Heidelberg New York, pp 394–408
7. Boudet A, Comon H (1996) Diophantine equations, Presburger arithmetic and finite automata. In: *Kirchner H (ed) Proceedings of the 21st international colloquium on trees in algebra and programming (CAAP'96)*, April 1996. *Lecture notes in computer science*, vol 1059. Springer, Berlin Heidelberg New York, pp 30–43
8. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. *IEEE Trans Comput* 35(8):677–691
9. Bryant RE, Chen YA (1995) Verification of arithmetic functions with binary moment diagrams. In: *Proceedings of the 32nd ACM/IEEE conference on design automation*, June 1995
10. Bultan T, Yavuz-Kahveci T (2001) Action Language Verifier. In: *Proceedings of the 16th IEEE international conference on automated software engineering*
11. Chan W, Anderson RJ, Beame P, Burns S, Modugno F, Notkin D, Reese JD (1998) Model checking large software specifications. *IEEE Trans Softw Eng* 24(7):498–520
12. Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) Nusmv 2: An opensource tool for symbolic model checking. In: *Proceedings of the international conference on computer-aided verification*
13. Clarke EM, McMillan KL, Zhao X, Fujita M, Yang J (1993) Spectral transforms for large boolean functions with applications to technology mapping. In: *Proceedings of the 30th international conference on design automation*. ACM Press, New York, pp 54–60
14. Clarke EM, Fujita M, Zhao X (2000) Hybrid decision diagrams – overcoming the limitations of mtbdds and bmds. In: *Proceedings of the international conference on computer-aided design*, pp 159–163
15. Garey M, Jonson D (1979) *Computers and intractability: a guide to the theory of NP-completeness*. Freeman, New York
16. McMillan KL (1993) *Symbolic model checking*. Kluwer, Norwell, MA
17. Papadimitriou CH (1981) On the complexity of integer programming. *J ACM* 28(4):765–768
18. Bahar RI, Frohm EA, Gaona CM, Hachtel GD, Macii E, Pardo A, Somenzi F (1993) *Algebraic Decision Diagrams and their applications*. In: *IEEE /ACM international conference on computer-aided design*, pp 188–191. IEEE Press, New York

19. Sieling D, Wegener I (1993) Reduction of OBDDs in linear time. *Inf Process Lett* 48(3):139–144
20. Wolper P, Boigelot B (2000) On the construction of automata from linear arithmetic constraints. In: Graf S, Schwartzbach M (eds) *Proceedings of the 6th international conference on tools and algorithms for the construction and analysis of systems*, April 2000. *Lecture notes in computer science*, vol 1785. Springer, Berlin Heidelberg New York, pp 1–19
21. Yang J, Mok AK, Wang F (1997) Symbolic model checking for event-driven real-time systems. *ACM Trans Programm Lang Syst* 19(2):386–412
22. Yavuz-Kahveci T, Tuncer M, Bultan T (2001) Composite symbolic library. In: *Proceedings of the 7th international conference on tools and algorithms for the construction and analysis of systems*, April 2001. *Lecture notes in computer science*, vol 2031. Springer, Berlin Heidelberg New York, pp 335–344