Jan Długosz University in Częstochowa
Scientific Issues, Mathematics XV, Częstochowa 2010

A BOOLEAN ENCODING OF ARITHMETIC OPERATIONS

Andrzej Zbrzezny

Institute of Mathematics and Computer Science
Jan Długosz University in Częstochowa
al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland
e-mail: a.zbrzezny@ajd.czest.pl

Abstract. In this paper we present algorithms for a Boolean encoding of four basic arithmetic operations on integer numbers: addition, subtraction, multiplication and division. Integer numbers are encoded in two's complement system as vectors of Boolean formulae, and arithmetic operations are faithfully encoded as operations on vectors of Boolean formulae.

1. Introduction

Boolean encoding of arithmetic operations is an important issue in some areas of symbolic model checking, for example, in SAT-based model checking for timed automata with discrete data (TADD), i.e. timed automata augmented with integer variables. The first attempt to develop bounded model checking for TADD was undertaken in [9]. However, the set of arithmetic operations considered in this paper was limited to addition and subtraction of integer variables.

In Saturn [7], the system for static analysis of programs that was developed at Stanford University, Boolean encoding of arithmetic operations is also limited. In an unpublished technical report [1] there are listed many operations for constructing and manipulating vectors of Boolean formulae, among others, addition and subtraction. But for multiplication and division there are mentioned only restricted versions of operations on Boolean formulae: namely, multiplication and division of a Boolean vector by a constant integer number and division of a Boolean vector by a constant integer number.

There are many tools which make use of a Boolean encoding of arithmetic operations. One of them is C32SAT – a tool for checking C expressions by means of satisfiability testing [2]. C32SAT parses the input expression and builds a parse tree, which is transformed into an And-Inverter Graph. Afterwards, the graph is transformed into conjunctive normal form and passed to a SAT solver.

In this paper we show how to encode faithfully the four basic arithmetic operations for integer numbers: addition, subtraction, multiplication and division. Our algorithms for Boolean encoding of the operations in question are based on standard algorithms well-known in the theory of computer arithmetic.

In the technical report [8] we have also provided algorithms for a Boolean encoding of the operation of calculating integer square root and of the operation of exponentiation with nonnegative integer exponent.

2. Basic notions and notations

Definition 1. *Let $\mathcal{V}$ be a nonempty set of propositional variables. The set $\mathcal{F}(\mathcal{V})$ of* Boolean formulae *over $\mathcal{V}$ is defined by the following grammar:*

$$f \ ::= \ \boldsymbol{false} \mid \boldsymbol{true} \mid p \mid \neg f \mid f \ \vee \ f \mid f \ \wedge \ f$$

The propositional variables and the constants $\boldsymbol{false}$ and $\boldsymbol{true}$ are called *atomic Boolean formulae.* In order to enhance readability of Boolean encoding of arithmetic operations we shall use two auxiliary propositional connectives: $\oplus$ (*exclusive disjunction*) and $\equiv$ (*biconditional*), defined in the standard way:

$$f \oplus g \ = \ (f \wedge \neg g) \vee (\neg f \wedge g) \qquad f \equiv g \ = \ (f \wedge g) \vee (\neg f \wedge \neg g)$$

We assume that, from greatest to lowest priority, the priority order is as follows: $\neg$, $\wedge$, $\vee$, $\oplus$, $\equiv$.

Definition 2. *Let $\mathcal{B}_2 = \langle \{0,\ 1\},\ -,\ \cup,\ \cap\ ,\ 0,\ 1\rangle$ be the two element Boolean algebra. A* valuation *v is a mapping from the set of atomic Boolean formulae to the universe of the Boolean algebra $\mathcal{B}_2$ satisfying the condition $v(\boldsymbol{false}) = 0$ and $v(\boldsymbol{true}) = 1$. The set of all the valuations will be denoted by $\mathcal{V}al(\mathcal{V})$.*

It is well known that each valuation v can be uniquely extended to a homomorphism h^v from the algebra of formulae $\langle \mathcal{F}(\mathcal{V}),\ \neg,\ \vee,\ \wedge\ \boldsymbol{false},\ \boldsymbol{true}\rangle$ to the Boolean algebra $\mathcal{B}_2$.

From now on we shall write $\mathcal{F}$ and $\mathcal{V}al$ instead of $\mathcal{F}(\mathcal{V})$ and $\mathcal{V}al(\mathcal{V})$ respectively, as we assume that the set $\mathcal{V}$ of the propositional variables is fixed.

Definition 3. *A* vector of Boolean values *is a finite, nonempty sequence of Boolean values* 0 *and* 1*.*

As the Boolean values 0 and 1 can be identified with binary digits, from now on, vectors of Boolean values will be called *bit vectors*. Every bit vector will be interpreted as an integer encoded in the two's-complement system. Namely, let $\mathtt{a} = \langle \mathtt{a}_{n-1}, \ldots, \mathtt{a}_0 \rangle$ be a bit vector of length n. Define the interpretation $\mathcal{I}(\mathtt{a})$ in the following standard way:

$$\mathcal{I}(\mathtt{a}) = \left(\sum_{i=0}^{n-1} \mathtt{a}_i \cdot 2^i \right) - (\mathtt{a}_{n-1} \cdot 2^n) .$$

Definition 4. *A* vector of Boolean formulae *(a* Boolean vector *for short) is a finite, nonempty sequence of Boolean formulae. A set of all the Boolean vectors of length n will be denoted by $\mathcal{BV}_n$.*

Let $\mathtt{x} = \langle \mathtt{x}_{n-1}, \ldots, \mathtt{x}_0 \rangle$ be a Boolean vector and v be a valuation. Then a sequence $H^v(\mathtt{x}) = \langle h^v(\mathtt{x}_{n-1}), \ldots, h^v(\mathtt{x}_0) \rangle$ is a bit vector that will be interpreted as a number $\mathcal{I}(H^v(\mathtt{x}))$. From now on we shall write $\mathcal{I}^v(\mathtt{x})$ instead of $\mathcal{I}(H^v(\mathtt{x}))$.

It it well known from computer arithmetic that in two's complement representation of a number b the most significant bit is equal to 1 if and only if the number b is negative. Recall also that for every bit vector $\mathtt{a}$ of length n, the following hold:

$$-2^{n-1} \leq \mathcal{I}(\mathtt{a}) \leq 2^{n-1} - 1. \tag{1}$$

3. Encoding of arithmetic relations and operations

We start with an obvious observation that the result of an arithmetic operation may not fit in the two's complement representation of a given length n. This is clear for addition, subtracting and multiplication. There is also one particular case for division. Namely, when a dividend is equal to $-2^{n-1} - 1$ and a divisor is equal to -1, the result, which is equal to 2^{n-1}, does not fit into n bits. Such a situation is called an *overflow*. This motivates the following notion of faithful encoding.

Let $\circ$ be a binary arithmetic operation and let $\boxdot$ be a binary operation on Boolean vectors. We say that the operation $\boxdot$ *encodes* the operation $\circ$ *faithfully* if and only if for every $\mathtt{x}, \mathtt{y} \in \mathcal{BV}_n$ and every $v \in \mathcal{V}al$,

$$-2^{n-1} \leq \mathcal{I}^v(\mathtt{x}) \circ \mathcal{I}^v(\mathtt{y}) \leq 2^{n-1} - 1 \Longrightarrow \mathcal{I}^v(\mathtt{x} \boxdot \mathtt{y}) = \mathcal{I}^v(\mathtt{x}) \circ I^v(\mathtt{y}).$$

The definition of the faithfully encoding of a unary arithmetic operation is analogous.

In what follows we assume that there is a global variable `overflow` initially set to ***false*** in which the Boolean formula expressing a possible overflow is computed.

Let $\sim$ be a two argument arithmetic relation. We say that a two argument operation $\bowtie : \mathcal{BV}_n \times \mathcal{BV}_n \longrightarrow \mathcal{F}$ *faithfully encodes* the relation $\sim$ if and only if for every $\mathtt{x}, \mathtt{y} \in \mathcal{BV}_n$ and every $v \in \mathcal{V}al$,

$$\mathtt{h}^v(\mathtt{x} \bowtie \mathtt{y}) = 1 \iff \mathcal{I}^v(\mathtt{x}) \sim I^v(\mathtt{y}).$$

3.1. Encoding of the relation "equal to"

In order to find a Boolean formula that faithfully encodes the equality relation assume that v is an arbitrary but fixed valuation, and observe that $\mathcal{I}^v(\mathtt{x}) = \mathcal{I}^v(\mathtt{y})$ iff $h^v\left(\forall_{j=0}^{n-1}(\mathtt{x}_j \equiv \mathtt{y}_j)\right) = 1$. Thus, Algorithm 1 constructs a Boolean formula Equal$(\mathtt{x}, \mathtt{y})$ that is the conjunction of all the formulae of the form $\mathtt{x}_j \equiv \mathtt{y}_j$.

Algorithm 1 Equal

Input: Boolean vectors x, y of length n.

Output: A Boolean formula f such that $\forall v \in \mathcal{V}al$, $\mathcal{I}^v(\mathtt{f}) = 1 \iff \mathcal{I}^v(\mathtt{x}) = \mathcal{I}^v(\mathtt{y})$.

```
1: function Equal(x, y)
2:     f ← true
3:     for $j \leftarrow 0$ to $n-1$ do
4:         f ← f ∧ (x[$j$] ≡ y[$j$])
5:     end for
6:     return f
7: end function
```

3.2. Addition

To define the addition of two Boolean vectors we adapt the method of the addition of two bit vectors known from computer arithmetic. Let x, y be two Boolean vectors of length n, i.e. let $\mathtt{x} = \langle \mathtt{x}_{n-1}, \dots, \mathtt{x}_0 \rangle$ and $\mathtt{y} = \langle \mathtt{y}_{n-1}, \dots, \mathtt{y}_0 \rangle$, where for every $0 \leq k < n$, $\mathtt{x}_k$ and $\mathtt{y}_k$ are Boolean formulae. Define an ordered pair of Boolean vectors $\langle \mathtt{w}, \mathtt{c} \rangle \in \mathcal{BV}_n \times \mathcal{BV}_{n+1}$ as follows: first, let $\mathtt{c}_0 = 0$; next, for $0 \leq k < n$, let

$$\langle \mathtt{w}_k,\ \mathtt{c}_{k+1} \rangle = \langle \mathtt{x}_k \oplus \mathtt{y}_k \oplus \mathtt{c}_k,\ (\mathtt{x}_k \wedge \mathtt{y}_k) \vee (\mathtt{x}_k \wedge \mathtt{c}_k) \vee (\mathtt{y}_k \wedge \mathtt{c}_k) \rangle.$$

The vector `w` represents the sum of `x` and `y`, and the vector `c` represents the succeeding carry bits. Clearly, the sum of two bit vectors of length n may not fit into n bits. By (1), this happens if and only if the sum is less than -2^n or greater than $2^n - 1$. It is known from computer arithmetic that adding two integers cause an overflow exactly when the carry bits $\mathtt{c}_n$ and $\mathtt{c}_{n+1}$ are different.

Algorithm 2 ADD

Input: Boolean vectors `x`, `y` of length n.

Output: A Boolean vector `w` of length n such that $\forall v \in Val$, if $-2^{n-1} \leq \mathcal{I}^v(\mathtt{x}) + I^v(\mathtt{y}) \leq 2^{n-1} - 1$, then $\mathcal{I}^v(\mathtt{w}) = \mathcal{I}^v(\mathtt{x}) + I^v(\mathtt{y})$.

```
1: function ADD(x, y)
2:     c[0] ← $\boldsymbol{false}$
3:     for $k \leftarrow 0$ to $n-1$ do
4:         w[$k$] ← x[$k$] $\oplus$ y[$k$] $\oplus$ c[$k$]
5:         c[$k+1$] ← (x[$k$] $\wedge$ y[$k$]) $\vee$ (x[$k$] $\wedge$ c[$k$]) $\vee$ (y[$k$] $\wedge$ c[$k$])
6:     end for
7:     overflow ← overflow $\vee$ (c[$n$] $\oplus$ c[$n+1$])
8:     return w
9: end function
```

3.3. Subtraction

Notice that in order to subtract two integers it is enough to add to the first number the additive inverse of the second number. Therefore, we need an operation on Boolean vectors that encodes additive inverse.

Recall that computing additive inverse for a two's complement number involves complementing each bit and then adding 1. It follows that we need an operation for creating a Boolean vector that represents the number 1. It is obvious that the number 1 is represented by the Boolean vector of the form $\langle \boldsymbol{false}, \ldots, \boldsymbol{false}, \boldsymbol{true} \rangle$, and an algorithm for creating this vector is trivial. Nevertheless, it will be useful to provide a more general Algorithm 3 that for a given integer creates a Boolean vector representing that number.

In this algorithm we use the operation >> of arithmetic right shift also known as signed shift. Recall that in `Java` the operator >> designates signed shift, whereas in `C++` a meaning of the operator >> is implementation-defined. Note that in `gcc` compiler, i.e the compiler we use, the operator >> is implemented as signed shift. In order to ensure that an implementation of Algorithm 3 in the language `C++` is independent of an used compiler, one should use a proper implementation of signed shift instead of the operator >>. Now

Algorithm 3 BoolVec

Input: A number of bits n and an integer a.

Output: A Boolean vector w of length n such that $\forall v \in \mathcal{V}al$, if $-2n - 1 \leq a \leq 2^{n-1} - 1$, then $\mathcal{I}^v(\mathtt{w}) = a$.

```
function BoolVec(n, a)
    if $a < 0$ then
        $\mathtt{w}[n-1] \leftarrow$ true
    else
        $\mathtt{w}[n-1] \leftarrow$ false
    end if
    for $k \leftarrow 0$ to $n - 2$ do
        if $a$ mod $2 = 0$ then
            $\mathtt{w}[k] \leftarrow$ false
        else
            $\mathtt{w}[k] \leftarrow$ true
        end if
        $a \leftarrow a >> 1$
    end for
    overflow $\leftarrow$ overflow $\vee (a \neq 0 \wedge a \neq -1)$
    return w
end function
```

we are able to write down an algorithm that computes the additive inverse of a Boolean vector x, also called the opposite of x.

Note that in two's complement arithmetic adding the number 1 to a n-bit number b generates overflow if and only if $b = 2^n - 1$. As the two's complement representation of the number $2^n - 1$ is of the form $\langle 0, 1, \ldots, 1\rangle$, which is the result of complementing each bit in the vector $\langle 1, 0, \ldots, 0\rangle$ that represents the number -2^{n-1}, it follows that taking additive inverse of a given number a generates overflow exactly when $a = -2^{n-1}$.

In the algorithm for subtracting we need some auxiliary operations on Boolean vectors.

For a given Boolean vector x of length n and an integer $m \geq n$, the auxiliary operation Extend implemented in Algorithm 5 creates a Boolean vector w of length m that represents the same integers as the vector x. This is done by copying all the elements of x to the corresponding elements of y and then copying the sign bit of x to the most significant $m - n$ elements of y. The algorithm described reflects the known operation of extension that consists in increasing the number of bits of a binary number while preserving the number's sign and value. For example, if 8 bits are used to represent the value

Algorithm 4 OPP

Input: A Boolean vector x of length n.

Output: A Boolean vector w of length n such that $\forall v \in \mathcal{V}al$, if $-2^{n-1} < \mathcal{I}^v(\mathtt{x}) \leq 2^{n-1} - 1$, then $\mathcal{I}^v(\mathtt{w}) = -\mathcal{I}^v(\mathtt{x})$.

```
function Opp(x)
    for $k \leftarrow 0$ to $n-1$ do
        $\mathtt{w}[k] \leftarrow \neg\mathtt{x}[k]$
    end for
    $\mathtt{w} \leftarrow$ Add(w, BoolVec($n$, 1))
    return w
end function
```

Algorithm 5 EXTEND

Input: A Boolean vector x of length n and a positive number $m \geq n$.

Output: A Boolean vector w of length m such that $\forall v \in \mathcal{V}al$, $\mathcal{I}^v(\mathtt{w}) = \mathcal{I}^v(\mathtt{x})$.

```
function Extend(x, $m$)
    for $k \leftarrow 0$ to $n-1$ do
        $\mathtt{w}[k] \leftarrow \mathtt{x}[k]$
    end for
    for $k \leftarrow n$ to $m-1$ do
        $\mathtt{w}[k] \leftarrow \mathtt{x}[n-1]$
    end for
    return w
end function
```

-15 using two's complement $\langle 1,1,1,1,0,0,0,1\rangle$, and sign extend to 16 bits is used, the new representation would be $\langle 1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1\rangle$.

For a given Boolean vector x of length m, the auxiliary operation REDUCE implemented in Algorithm 6 creates a Boolean vector w of length $n \leq m$ such that every integer represented by the Boolean w is also represented by the Boolean vector x.

Now we are in a position to write down an algorithm for subtraction of two Boolean vectors representing integer numbers. At the beginning the algorithm enlarges both the arguments by one bit in order to avoid a possible overflow that may occur in the operation of taking the additive inverse. Then the algorithm adds the enlarged first argument to the additive inverse of the enlarged second argument and puts the result in an auxiliary Boolean vector w. Eventually, the overflow is computed and as the result of subtraction the algorithm returns the Boolean vector REDUCE(w, n).

Algorithm 6 Reduce

Input: A Boolean vector x of length m and a positive number $n \leq m$.

Output: A Boolean vector w of length n such that $\forall v \in \mathcal{V}al$, if $-2^n \leq \mathcal{I}^v(\mathtt{x}) \leq 2^n - 1$, then $\mathcal{I}^v(\mathtt{w}) = \mathcal{I}^v(\mathtt{x})$.

```
1: function Reduce(x, $n$)
2:     for $k \leftarrow 0$ to $n-2$ do
3:         w[$k$] $\leftarrow$ x[$k$]
4:     end for
5:     w[$n-1$] $\leftarrow$ x[$n-1$]
6:     return w
7: end function
```

Algorithm 7 Subtract

Input: Boolean vectors x, y of length n.

Output: A Boolean vector w of length n such that $\forall v \in \mathcal{V}al$, if $-2^{n-1} \leq \mathcal{I}^v(\mathtt{x}) - I^v(\mathtt{y}) \leq 2^{n-1} - 1$, then $\mathcal{I}^v(\mathtt{w}) = \mathcal{I}^v(\mathtt{x}) - I^v(\mathtt{y})$.

```
1: function Subtract(x, y)
2:     p $\leftarrow$ Extend(x, $n+1$); q $\leftarrow$ Extend(y, $n+1$)
3:     w $\leftarrow$ Add(p, Opp(q))
4:     overflow $\leftarrow$ overflow $\vee$ $\neg$(w[$n-1$] $\equiv$ w[$n$])
5:     return Reduce(w, $n$)
6: end function
```

3.4. Multiplication

In the algorithm for multiplication we need some additional auxiliary operations on Boolean vectors, namely, ShiftLeft, Conjunction and Abs.

The auxiliary operation ShiftLeft, which is implemented in Algorithm 8, is the operation of shifting a Boolean vector one bit left, and after shifting, filling in the least significant position of the vector with the Boolean formula ***false***. This operation corresponds to the well known operation called a logical shift. The operation of shifting is also used in the algorithm for dividing nonnegative integers.

The operation Conjunction implemented in Algorithm 9 creates the bitwise conjunction of a Boolean formula and a Boolean vector. This simple operation enables simulating a conditional execution in the algorithms of multiplying and dividing. The reason for this is the following obvious property:

$$(\forall v \in \mathcal{V}al)\ \ \mathcal{I}^v(\textsc{Conjunction}(f,\ \mathtt{x}) = \begin{cases} 0, & \text{if } \mathcal{I}^v(f) = 0 \\ \mathcal{I}^v(\mathtt{x}), & \text{if } \mathcal{I}^v(f) = 1. \end{cases}$$

Algorithm 8 SHIFTLEFT

Input: A Boolean vector x of length n.

Output: A Boolean vector x logically shifted left by one.

```
1: procedure ShiftLeft(x)
2:     for $k \leftarrow n-1$ down to 1 do
3:         $x[k] \leftarrow x[k-1]$
4:     end for
5:     $x[0] \leftarrow$ false
6: end procedure
```

Algorithm 9 CONJUNCTION

Input: A Boolean formula f and a Boolean vector x of length n.

Output: A Boolean vector w of length n such that for every $0 \le k < n$, $w[k] = f \wedge x[k]$.

```
1: function Conjunction($f$, x)
2:     for $k \leftarrow 0$ to $n-1$ do
3:         $w[k] \leftarrow f \wedge x[k]$
4:     end for
5:     return w
6: end function
```

For a given Boolean vector x of length n, the auxiliary operation ABS implemented in Algorithm 10 creates a Boolean vector w of length n such that w represents the absolute value of x.

Now we are in a position to write down the algorithm for multiplication of two Boolean vectors representing nonnegative integers. Algorithm 11 creates a Boolean vector that represents the result of multiplication of two Boolean vectors that represent nonnegative integers. We adapted the simplest method that computes the product one bit at a time, and is a symbolic version of the paper-and-pencil method.

Note that at the beginning of the algorithm some preparatory steps are needed. First, both the arguments are copied to auxiliary variables p and q; next, the most significant bit of each of the auxiliary variables is set to ***false***; eventually, both the auxiliary variables are enlarged to size $2 \cdot n$, and w is set to $\langle$***false***$, \dots ,$ ***false***$\rangle$.

After these preparatory steps, the algorithm proceeds as follows: for every k from 0 to $n-1$ the conjunction of the multiplicand and the kth element of multiplier is added to w. This last step simulates the conditional addition of the multiplicand to the product: the multiplicand is added in the kth step if and only if the kth element of multiplier represents the binary value 1.

Algorithm 10 ABS

Input: A Boolean vector x of length n.

Output: A Boolean vector w of length n such that $\forall v \in \mathcal{V}al$, if $-2^{n-1} < \mathcal{I}^v(\mathtt{x}) \leq 2^{n-1} - 1$, then $\mathcal{I}^v(\mathtt{w}) = |\mathcal{I}^v(\mathtt{x})|$.

```
1: function ABS(x)
2:     y ← OPP($x$)
3:     for $k$ ← 0 to $n-2$ do
4:         w[$k$] ← (x[$n-1$] ∧ y[$k$]) ∨ (¬x[$n-1$] ∧ x[$k$])
5:     end for
6:     w[$n-1$] ← false
7:     return w
8: end function
```

Algorithm 11 MULTIPLYNONNEG

Input: Boolean vectors x, y of length n

Output: A Boolean vector w of length $2 \cdot n$ such that $\forall v \in \mathcal{V}al$, if $\mathcal{I}^v(\mathtt{x}) \geq 0$ and $\mathcal{I}^v(\mathtt{y}) \geq 0$, then $\mathcal{I}^v(\mathtt{w}) = \mathcal{I}^v(\mathtt{x}) \cdot I^v(\mathtt{y})$.

```
 1: function MULTIPLYNONNEG(x, y)
 2:     p ← x; p[$n-1$] ← false; p ← EXTEND(p, $2 \cdot n$)
 3:     q ← y; q[$n-1$] ← false; q ← EXTEND(q, $2 \cdot n$)
 4:     w ← BOOLVEC($2 \cdot n$, 0)
 5:     for $k$ ← 0 to $n-2$ do
 6:         w ← ADD(w, CONJUNCTION(q[$k$], p)
 7:         SHIFTLEFT(p)
 8:     end for
 9:     return w
10: end function
```

The following Algorithm 12 creates a Boolean vector that represents the result of multiplication of two Boolean vectors that represent signed integers. At the beginning, the algorithm enlarges both the arguments by one bit. Then, two cases are considered: the arguments are of the same sign (f_0) and the arguments have different signs (f_1). In each of the cases the algorithm symbolically converts the arguments to be nonnegative, does an unsigned multiplication, and for the case when the original arguments have different signs, negates the result. Next, from the two symbolic results, named $\mathtt{w}_0$ and $\mathtt{w}_1$, the final result is created in the following way: for every k such that $0 \leq k < 2 \cdot (n+1)$, the kth bit of the product is set to $f_0 \wedge \mathtt{w}_0[k] \vee f_1 \wedge \mathtt{w}_1[k]$. Eventually, the overflow is computed and the result is reduced to n bits.

Algorithm 12 Multiply

Input: Boolean vectors x, y of length n

Output: A Boolean vector w of length n such that $\forall v \in \mathcal{V}al$, if $-2^{n-1} \leq \mathcal{I}^v(\mathtt{x}) \cdot I^v(\mathtt{y}) \leq 2^{n-1}-1$, then $\mathcal{I}^v(\mathtt{w}) = \mathcal{I}^v(\mathtt{x}) \cdot I^v(\mathtt{y})$.

```
1:  function Multiply(x, y)
2:      $p \leftarrow$ Extend(x, $n+1$); $q \leftarrow$ Extend(y, $n+1$)
3:      $w_0 \leftarrow$ MultiplyNonNeg(Abs(p), Abs(q))
4:      $w_1 \leftarrow$ Opp($w_0$)
5:      $f_0 \leftarrow (\neg x[n-1] \wedge \neg y[n-1]) \vee (x[n-1] \wedge y[n-1])$
6:      $f_1 \leftarrow (\neg x[n-1] \wedge y[n-1]) \vee (x[n-1] \wedge \neg y[n-1])$
7:      $m \leftarrow 2 \cdot (n+1)$
8:      for $k \leftarrow 0$ to $m-1$ do
9:          $w[k] \leftarrow f_0 \wedge w_0[k] \vee f_1 \wedge w_1[k]$
10:     end for
11:     $of \leftarrow$ false
12:     for $k \leftarrow n-1$ to $m-2$ do
13:         of $\leftarrow$ of $\vee \neg(w[k] \equiv w[m-1])$
14:     end for
15:     overflow $\leftarrow$ overflow $\vee$ $of$
16:     return Reduce(w, $n$)
17: end function
```

3.5. Division

There are many possible algorithms for dividing nonnegative integers. We adapted the so called restoring radix-2 division algorithm described in Appendix H of [3]. Algorithm 13 is done by shifts, subtractions, additions and testing whether the number is negative. The algorithm needs four registers: one for the dividend x, one for the divisor y, one for the quotient q, and one for the remainder r. The registers r and q form a double-length register pair. The register q is initially set to the value of x and the register q is initially set to 0.

Algorithm 14 creates a Boolean vector that represents the result of division of two Boolean vectors that represent signed integers. There are the same cases to consider for arguments as in Algorithm 12. Also the method of computing the final result is nearly the same. There are only two differences. The first one is that the result is not reduced to the length of arguments, as in all the cases considered the results are of length n. The second one is the method of setting the overflow.

Algorithm 13 DivideNonNeg

Input: Boolean vectors x, y of length n

Output: Boolean vectors r, q such that $\forall v \in \mathcal{V}al$, if $\mathcal{I}^v(\mathtt{x}) \geq 0$ and $\mathcal{I}^v(\mathtt{y}) > 0$, then $\mathcal{I}^v(\mathtt{x}) = \mathcal{I}^v(\mathtt{q}) \cdot I^v(\mathtt{y}) + \mathcal{I}^v(\mathtt{r})$.

```
1: function DivideNonNeg(x, y)
2:     q ← x; r ← BoolVec($n$, 0)
3:     for $k \leftarrow 0$ to $n-1$ do
4:         ShiftLeft(r)
5:         r[0] ← q[$n-1$]
6:         ShiftLeft(q)
7:         r ← Subtract(r, y)
8:         q[0] ← ¬r[$n-1$]
9:         r ← Add(r, Conjunction(r[$n-1$], y))
10:    end for
11:    return ⟨q, r⟩
12: end function
```

We would also point out that the signs of the quotient and of the remainder for negative dividends and/or negatives divisors are computed in accordance with the following rules of `C++` and `Java`: the quotient is negative if and only if both the dividend and the divisor have different signs, and the remainder is negative if and only if the dividend is negative.

3.6. Encoding of the relation "less than"

Let us note that the relation "less than" can be encoded by using the operation of subtraction. The algorithm enlarges both the arguments by one bit in order to avoid a possible overflow that may occur in the operation of subtraction and then returns the most significant element of the Boolean vector representing the difference.

4. Implementation

We have implemented the described algorithms in the programming language `C++` by designing the following classes: the class `BoolForm` that implements basic logical operations on Boolean formulae; the class `BoolFormVect` that implements basic operations on Boolean vectors; and the class `Integer`, derived from `BoolFormVect`, that implements the Boolean encoding of arithmetic relations and operations as described in this paper.

Algorithm 14 Divide

Input: Boolean vectors x, y of length n

Output: Boolean vectors q, r such that $\forall v \in \mathcal{V}al$, if $\mathcal{I}^v(\mathtt{y}) \neq 0$, then $\mathcal{I}^v(\mathtt{x}) = \mathcal{I}^v(\mathtt{q}) \cdot I^v(\mathtt{y}) + \mathcal{I}^v(\mathtt{r})$, $sgn(\mathcal{I}^v(\mathtt{q})) = sgn(\mathcal{I}^v(\mathtt{x})) \cdot sgn(\mathcal{I}^v(\mathtt{y}))$, and $sgn(\mathcal{I}^v(\mathtt{r})) = sgn(\mathcal{I}^v(\mathtt{x}))$.

```
1:  function Divide(x, y)
2:      p ← Extend(x, $n+1$); q ← Extend(y, $n+1$)
3:      $\langle$q$_0$, r$_0\rangle$ ← DivideNonNeg(Abs(p), Abs(q))
4:      q$_1$ ← Opp(q$_0$); r$_1$ ← Opp(r$_0$)
5:      f$_{00}$ ← $\neg$x[$n-1$] $\wedge$ $\neg$y[$n-1$]); f$_{01}$ ← $\neg$x[$n-1$] $\wedge$ y[$n-1$])
6:      f$_{10}$ ← x[$n-1$] $\wedge$ $\neg$y[$n-1$]); f$_{11}$ ← x[$n-1$] $\wedge$ y[$n-1$])
7:      for $k$ ← 0 to $n$ do
8:          q[$k$] ← ((f$_{00}$ $\vee$ f$_{11}$) $\wedge$ q$_0$[$k$]) $\vee$ ((f$_{01}$ $\vee$ f$_{10}$) $\wedge$ q$_1$[$k$])
9:          r[$k$] ← ((f$_{00}$ $\vee$ f$_{01}$) $\wedge$ r$_0$[$k$]) $\vee$ ((f$_{10}$ $\vee$ f$_{11}$) $\wedge$ r$_1$[$k$])
10:     end for
11:     a ← BoolVec($n+1$, 0)
12:     b ← BoolVec($n+1$, 1)
13:     z ← BoolVec($n+1$, $2^{n-1}$)
14:     of ← Equal(p, Opp(z)) $\wedge$ Equal(q, Opp(b)) $\vee$ Equal(y, a)
15:     overflow ← overflow $\vee$ of
16:     return $\langle$q, r$\rangle$
17: end function
```

In order to test the above algorithms we have created testing programs for all the arithmetic operations considered. In every program some suitable formula φ is tested in the following way: at first, φ is converted to a set of clauses C in a way such that although the set C is not logically equivalent to the formula φ, it preserves satisfiability, i.e. C is satisfiable if and only if φ is satisfiable; then, we check satisfiability of C by using `MiniSat`. Some of experimental results for the programs mentioned above are provided in [8].

5. Final remarks

As a result of implementing our Boolean encoding of arithmetic operations we were able to extend the module `BMC4TADD` of the model checker Verics [4] in order to include multiplication and division in the set of the allowed operations. The module `BMC4TADD` serves for verification of properties of timed automata with discrete data. The formalism of timed automata with discrete data and basic arithmetic operations is now used in verification of Java programs (see [6, 10]). The Boolean encoding of arithmetic operations was also used in a new approach to model checking of systems specified in UML (see [5]).

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, P. Hawkins. The Saturn program analysis system. Technical Report, Stanford University, 2006.

[2] R. Brummayer, A. Biere. C32SAT: Checking C expressions. In: *Proc. CAV'2007*, LNCS 4590, pp. 294–297, Springer, Berlin, 2007.

[3] J.L. Hennessy, D.A. Patterson. *Computer Architecture: A Quantitative Approach*, 3rd edition. Morgan Kaufmann Publishers, San Francisco, CA, 2003.

[4] M. Kacprzak, W. Nabiałek, A. Niewiadomski, W. Penczek, A. Pólrola, M. Szreter, B. Woźna, A. Zbrzezny. VerICS 2007 - a model checker for knowledge and real-time. *Fund. Informaticae*, **85** (1-4), 313–328, 2008.

[5] A. Niewiadomski, W. Penczek, M. Szreter. A new approach to model checking of UML state machines. *Fund. Informaticae*, **93** (1-3), 289–303, 2009.

[6] A. Rataj, B. Woźna, A. Zbrzezny. A translator of Java programs to TADDs. *Fund. Informaticae*, **93** (1-3), 305–324, 2009.

[7] Y. Xie, A. Aiken. Saturn: A SAT-based tool for bug detection. In: *Proc. CAV'2005*, LNCS 3576, pp. 139–143. Springer, Berlin, 2005.

[8] A. Zbrzezny. A boolean encoding of arithmetic operations. Technical Report 999, ICS PAS, 2007.

[9] A. Zbrzezny, A. Pólrola. SAT-based reachability checking for timed automata with discrete data. *Fund. Informaticae*, **79** (3–4), 579–593, 2007.

[10] A. Zbrzezny, B. Woźna. Towards verification of Java programs in VerICS. *Fund. Informaticae*, **85** (1-4), 533–548, 2008.