Using NDD to represent constraints on integers in TREX *

Stéphanie Delaune and Mihaela Sighireanu

LIAFA - University of Paris 7, 2 place Jussieu, 75251 Paris France

Abstract. TREX is a tool for reachability analysis of systems using variables with infinite domains (*e.g.*, clocks, counters) and parameters (i.e., uninstantiated constants). In order to represent the sets of reachable configurations for these variables, TREX uses symbolic representations. For example, to represent (linear or non-linear) constraints over the (integer or real) parameters, we used a library developed in TREX based on syntactic tree representation. In order to solve these constraints, external solvers are called on the string representing the constraint. In this report we describe the integration in TREX of an other symbolic representation for the linear constraints over integer parameters. This representation is Number Decision Diagrams (NDD), for which a library is available within the LASH tool. NDD are finite automata representing Presburger arithmetic definable sets of integer vectors. The NDD library provides also solvers for such constraints. The conclusion of this work is positive: the performances of TREX are improved for some examples and we point out interesting suggestions for the improvement of the NDD library.

1 Introduction

Symbolic reachability analysis is a powerful paradigm used in the verification of infinite-state systems, such as extended automata, communicating automata through unbounded queues, parameterized systems, etc. It consists in using finite structures to represent infinite sets of configurations, and iterative exploration procedures to compute (a finite representation of) the set of all reachable configurations, or an upper approximation of this set.
Several verification tools implement the symbolic reachability, for example LASH [WB00], TREX [BCAS01], ALV [Bul98], etc. Each of them developed specific data structures for the symbolic representation of infinite sets of configurations. For example, LASH focuses on the symbolic representations based on automata which may represent sets of configurations over integer and real variables. ALV uses linear programming over integers. TREX uses extensions of DBMs [Dil89], syntactic trees, and regular expressions [AAB99] to represent configurations over clocks, counters, and lossy FIFO queues.
The comparison of performances for these symbolic representations is actually difficult for at least two reasons. First, the performances reported for each tool consider different examples or the same example tuned to give best results with the tool. Second, the tools are not designed to support easy change of the symbolic structures used.
This report explains how we deal with this problem by allowing the use of a symbolic representation provided by LASH in TREX. The work consists in implementing an interface between TREX and the NDD library of LASH for representation of Presburger definable sets of integers vectors. This interface is used to represent the constraints between integer parameters. Although the NDD representation may also replace the DBMs for the configurations over counters, we conserve the DBMs in TREX. The reason is that the extrapolation procedures implemented in TREX work only on DBMs. Replacing DBMs by NDDs asks for importing the extrapolation procedures of LASH.
This experience is useful for at least three reasons. First, we may compare the performances of different representations. Second, the use we made for the NDD library is special, so it may give some hints about the improvement of this library. Third, from the point of view of TREX, it may allow to obtain better performances.

* This work was supported in part by the European Commission (FET project ADVANCE, contract No IST-1999-29082).

2 Overview of TREX

TREX[1] allows to analyze automatically automata-based models equipped with variables of different kinds of *infinite*-domain data structures and with *parameters* (i.e., uninstantiated constants). These models are, at the present time, parametric (continuous-time) timed automata, extended with integer counters and communicating through unbounded lossy FIFO queues.
The techniques used in TREX are based on symbolic reachability analysis. In the current version, TREX provides packages for symbolic representation of configurations of lossy FIFO channels and parametric timed automata and counter automata. For parametric timed and counter automata, the symbolic structures used are *Constrained Parametric Difference Bound Matrices* (CPDBM) [AAB00]. A CPDBM is a pair (M, Φ) where M is a DBM [Dil89] which bounds may be arithmetical expressions over a set P of parameters and Φ is a constraint relating elements of P. The use of CPDBMs allows to deal in a uniform way with counter/clock automata, parametric/non-parametric models, and systems generating linear or nonlinear arithmetical constraints.
We implemented in C++ a package for manipulating CPDBMs. The package provides compact representation of CPDBMs and efficient methods for operations used during symbolic analysis (*e.g.*, emptiness check, intersection, and inclusion test). The package has three parts: the first part represents terms over a given set of parameters P, the second part represents the PDBMs and uses the first part for the bounds of matrices, and the third part represents constraints over the set P. A special effort has been devoted to develop efficient representation of terms and formulas and simplification techniques on these objects. For this, we implemented a package, called FOAF (*First-Order Arithmetical Formulas*), which also gives the kind (linear or non-linear) of terms and formulas. This analysis is needed in order to apply the right decision procedure for the satisfiability of formulas.
The external decision procedures used actually by TREX are those offered by OMEGA [Ome96] for formulas over integers and by the REDLOG package of REDUCE [Hea99] for formulas over reals. Moreover, we implemented in the FOAF package the Fourier-Motzkin procedure [ER92] for elimination of quantifiers over real variables.

3 FOAF library

The FOAF library represents the arithmetical terms and constraints on parameters. The representation is based on syntactic trees. The advantages of this solutions are:

- It is very general since it can represent linear and non-linear constraints between integer and real parameters.
- It is explicit, so easy to be debugged and to be read by a user or a solver.

However, it raises the following problems:

- The representation by syntactic trees has no canonical form, so the simplification of constraints is difficult.
  To deal with this problem we implemented in FOAF some heuristic methods. First, we define a normal form for terms and constraints which is based on a lexicographical order for terms and atomic constraints, and on the DNF for more complex constraints. Then, we use this normal form to simplify terms, to find equalities allowing substitutions, and to reduce formulas containing tautologies.
- It may be expensive in memory consumption for particular classes of constraints.
  We tried to reduce this consumption by hashing all terms and formulas in order to share the common sub-terms and sub-formulas. The hashing uses the normal form described above to quickly compare terms and formulas.

[1] *Tool for REachability analysis of CompleX Systems*, available at `http://www.liafa.jussieu.fr/~sighirea/trex`

– It uses external decision procedures, which call is time consuming.
  Compared with the decision procedures implemented in Reduce, our Fourier-Motzkin procedure is slower for big formulas. Moreover, for all decision procedure on integer constraint, we don't have local procedures. The connection with external decision procedures is done by Unix pipes.

To evaluate the impact of these disadvantages on the performances of TReX, we integrated the Ndd library of Lash in order to represent linear constraints between integers.

4 Ndd library of Lash

The Lash[2] [WB00] tool set is developed at University of Liège for representing infinite sets and exploring infinite state spaces. It is based on finite-state representations, which rely on finite-state automata for representing and manipulating infinite sets of values over various data domains.
The Ndd library provides functions and data types for manipulating sets of unbounded integer vectors. The sets of vectors that can be represented by Ndds include those that can be expressed in Presburger arithmetic (i.e., the first-order arithmetic theory of integers without multiplication).
The representation used is based on finite automata accepting the binary coding of sets of integers. This representation has a canonic form, so the simplifications of a set of constraints is natural.
The Ndd library provides functions for representing elementary sets and for performing various operations (including decision procedures) over represented sets. In the remainder of this section, we present data structures and operations which are useful for the interface with TReX.

4.1 Types and data structures

The Lash core package defines (file `lash-types.h`) the elementary types of Table 1 which are used also by the Ndd library.

Table 1. Base types of Lash.

uint1	unsigned char	sint1	signed char
uint4	unsigned int	sint4	signed int
uint8	unsigned long long	sint8	signed long long

The main data structure of the Ndd library is the `ndd` type defined by:

```
typedef struct ndd {
        uint4     dim;
        automate *automaton;
        uint1     base, properties;
} ndd;
```

where

– `dim` denotes the dimension of the vectors in the set encoded by the Ndd.
– `automaton` denotes the deterministic finite automaton representing the encoding of the constraint; the type `automate` is definite in the file `lash-auto.h`.
– `base` denotes the base used for the encoding of integers; this base is usually 2, but other bases > 1 may be chosen.

[2] `http://www.montefiore.ulg.ac.be/~boigelot/research/lash`

– `properties` gives some specific information on the encoding, *e.g.*, if the encoding of integers is *msdf*[3] or *lsdf*[4]. For example, the vector (5,3,2) is binary encoded by (0101,011,010) and represented by the word (0,0,0)(1,0,0)(0,1,1)(1,1,0) which corresponds to 000100011110 in *msdf* encoding; for the *lsdf* encoding the word representing the vector is 110011100000.

4.2 Building NDDs

The methods for building NDDs allow as depart point boolean constants and linear (in)equations. In the file `ndd-sets.h` are provided the following methods:

`ndd * ndd_create_empty(uint1 r, uint4 n, uint1 msdf)`
Creates a NDD representing the empty constraint (*false*) for a set of vectors of dimension `n`, encoded in base `r` following the coding specified by the `msdf` parameter. The automaton of the NDD created does not recognize any word.

`ndd * ndd_create_zn_msdf(uint1 r, uint4 n)`
Creates a NDD representing the universal set $Z^{\mathtt{n}}$ for a set of vectors of dimension `n`, encoded in base `r` following the *msdf* coding. The automaton of the NDD created recognizes any word. A similar method exists for the *lsdf* encoding.

Remark 1. These two methods build only the empty and universal constraint for a given dimension. In TREX, the dimension of constraints is dynamic because the number of parameters may change during the analysis. Then, an efficiant coding of boolean constants is those using NDD with dimension `n`=0 because the associated automaton is the smaller one. As explained in Section 5, we follow this remark by coding the *false* and *true* constants with NDD of dimension 0. The automaton of *false* recognizes the empty language, and the automaton of *true* recognizes the empty word.

In the file `ndd-equations.h` are provided the following methods:

`ndd* ndd_create_equation_msdf(uint1 r, uint4 n, sint4 *c, sint4 b)`
Creates the NDD representing the set of integer solutions of the following equation :

$$c[0] * x_0 + c[1] * x_1 + c[2] * x_2 + ... + c[n-1] * x_{n-1} = b$$

The parameters `n` and `r` have the same meaning as above. A similar method exists for the *lsdf* encoding.

`ndd* ndd_create_inequation_msdf(uint1 r, uint4 n, sint4 *c, sint4 b)`
Creates the NDD representing the set of integer solutions of the following inequation :

$$c[0] * x_0 + c[1] * x_1 + c[2] * x_2 + ... + c[n-1] * x_{n-1} \leq b$$

The parameters `n` and `r` have the same meaning as above. A similar method exists for the *lsdf* encoding.

Remark 2. If these methods are called with `n`=0, they returns NULL pointers. For our use, it would be useful to change this behavior and to return:

– the NDD of dimension 0 and automaton recognizing the empty word (*i.e.*, *true*) if `b`=0 *resp.* `b`≥0, and
– the NDD of dimension 0 and automaton recognizing the empty language (*i.e.*, *false*) if `b`≠0 *resp.* `b`<0, and

[3] Most Significant Digit First
[4] Low Significant Digit First

4.3 Tests on NDDs

We use intensively the equality test and the empty test on NDD provided by the following methods:

`int ndd_is_empty(ndd * nd)`
Return 1 if the set represented by the NDD `nd` is empty (*i.e.*, the constraint represented is invalid) and 0 otherwise.

`int ndd_equality(ndd * nd1, ndd * nd2)`
Returns 1 if `nd1` and `nd2` represent the same set of vectors. The two NDD must have the same dimension. The test corresponds to a equality of languages recognized by the automatons.

4.4 Operations on NDD

The operations available on NDD are similar to those on automata.
In the file `ndd-operations.h` are defined the following methods:

`ndd *ndd_product(ndd *nd1, ndd *nd2)`
Computes the Cartesian product of the two sets represented by the arguments. Indeed, if $\xi(\texttt{nd}i)$ and n_i ($i \in 1..2$) are the set *resp.* the dimension of vectors encoded by the NDD `nd`i, then the result of this method is a NDD having the dimension $n_1 + n_2$ and representing the set:

$$\xi(\texttt{ndr}) = \{(x_1, ..., x_{n_1}, y_1, ..., y_{n_2}) \mid (x_1, ..., x_{n_1}) \in \xi(\texttt{nd1}) \wedge (y_1, ..., y_{n_2}) \in \xi(\texttt{nd2})\}$$

Remark 3. In the context of dynamic dimension for NDD, this method allows to add dimensions to the NDD, but only in front or at the end.
When one of the arguments has dimension 0, the method returns the other NDD. Or, if the NDD of dimension 0 represents the empty set, the result should also represent the empty set.

`ndd *ndd_difference(ndd *nd1, ndd *nd2)`
Computes the difference between the sets represented by the arguments. If the two NDD have the same dimension `n`, the resulting NDD represents the set:

$$\xi(\texttt{ndr}) = \{(x_1, ..., x_n) \mid (x_1, ..., x_n) \in \xi(\texttt{nd1}) \wedge (x_1, ..., x_n) \notin \xi(\texttt{nd2})\}$$

`ndd *ndd_union(ndd *nd1, ndd *nd2)`
Computes the union of sets represented by the arguments. If the two NDD have the same dimension `n`, the resulting NDD represents the set:

$$\xi(\texttt{ndr}) = \{(x_1, ..., x_n) \mid (x_1, ..., x_n) \in \xi(\texttt{nd1}) \vee (x_1, ..., x_n) \in \xi(\texttt{nd2})\}$$

`ndd *ndd_intersection(ndd *nd1, ndd *nd2)`
Computes the intersection of sets represented by the arguments. If the two NDD have the same dimension `n`, the resulting NDD represents the set:

$$\xi(ndr) = \{(x_1, ..., x_n) \mid (x_1, ..., x_n) \in \xi(\texttt{nd1}) \wedge (x_1, ..., x_n) \in \xi(\texttt{nd2})\}$$

`ndd *ndd_interleave_z(ndd *nd, uint4 nl, uint4 nr)`
Allows to insert a new dimension (variable) in the middle of an NDD. In our interface (see 6.2), this method is always called with `nl`=1 and `nr`=0. In this case, the resulting NDD represents the set:

$$\xi(\texttt{ndr}) = \{(x_1, y_1, x_2, y_2, ..., x_n, y_n) \mid (x_1, ..., x_n) \in \xi(\texttt{nd}) \wedge (y_1, ..., y_n) \in Z^n\}$$

In the file `ndd-projection.h` is defined the following method:

`ndd *ndd_projection(ndd * nd, uint4 p)`
Returns the NDD representing the projection of the `p`th component of `nd`, which denotes the following set :

$$\xi(\texttt{ndr}) = \{(x_1, ..., x_{p-1}, x_{p+1}, ..., x_n) \mid \exists x_p \in Z \;.\; (x_1, ..., x_{p-1}, x_p, x_{p+1}, ..., x_n) \in \xi(\texttt{nd})\}$$

5 Integration of the NDD library

Before starting to discuss the integration issues, we note that NDD cannot be a better solution than the syntactic representation of FOAF because:

– NDD represents only linear integer constraints, so it is less general.
– The representation is not explicit and its translation into a readable formula is difficult.
– The finite automata encoding may consume a lot of memory when big integer constants are used in constraints (FOAF representation is not sensible to this problem).

For this reason, the user should be free to choose one or other of representations depending on the kind of system (s)he has to analyze with TREX. Moreover, when non-linear constraints appear during the exploration, it would be useful to be able to change the NDD representation into a FOAF representation. This means that the representation of constraints should be transparent to the exploration algorithm.

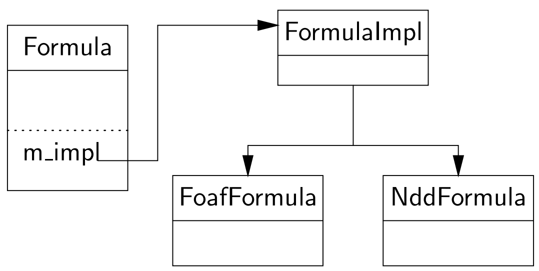

Fig. 1. Solution for formula implementation in TREX.

The solution we chosen for implementation is given on Figure 1. We split the FOAF library in two parts: the first part concerns only terms (used in bounds of PDBMs) and to build formulas, and the second part concerns formulas implemented as syntactic trees. Then, we use the design pattern model of *bridge* [GHJV94]. We define an abstract interface for formulas implementation, the abstract C++ class `FormulaImpl`. Two implementations are now available for this interface: the syntactic tree implementation provided by the FOAF library and the NDD implementation. The call of different implementations is done transparently by the use of class `Formula` which contains as private member the implementation.
The integration of NDD is then reduced to the implementation of the interface `FormulaImpl`. The whole specification of this interface is given on Annex A. In the remainder of this section we give some details about the implementation.

5.1 Data structure

The class `NddFormula` implements, using NDD the `FormulaImpl` interface:

```
class NddFormula : public FormulaImpl {
      ....
  private:
      ndd * m_ndd;
      vector<int> * m_params;
      vector<int> * m_acc_params;
      ....
};
```

A `NddFormula` is a constraint over integer parameters. These parameters are of two kinds:

– initial parameters of the model, stored in the `Parameters` array and
– acceleration parameters introduced during the extrapolation of loops, and identified by a unique number.

In order to know for each dimension of the NDD the corresponding variable, we include in class `NddFormula` two fields:

– `m_params` is the list of indexes of the parameter in the global array `Parameters`, and
– `m_acc_params` is the list of unique numbers of acceleration parameters.

Since the NDD library does not provide methods to swap the dimensions of the vectors represented, we fixed the following order:

$$\text{parameters} < \text{acceleration parameters}$$

and inside each kind of parameters, the variables are ordered on their index or number.

Remark 4. One may also use NDDs to represent also the constraints over the counters. In this case, our implementation should be extended with a new dimension vector corresponding to counters. The same should be done with the order proposed above.

5.2 Basic constraints and their constructor

The constraint *true* is represented by an NDD of dimension 0 which automaton has only one state, representing at the same time the initial and the final state. This automaton recognizes the empty word, *i.e.*, all the integer vectors of dimension 0. The vectors `m_params` and `m_acc_params` are empty.

The constraint *false* is represented by an NDD of dimension 0 which automaton has no state. This automaton recognizes the empty language. The vectors `m_params` and `m_acc_params` are empty.

We choose these representation of the basic constraints in order to reduce the memory space used for this kind of constraints and to be able to combine them with constraints of any dimension.

The constructor used to create the basic constraints is:

```
NddFormula(FoafOperation O)
```

which creates a NDD of dimension 0 and fields `m_params` and `m_acc_params` empty. If `O` is:

– `FOAF_TRUE`, the constraint represented is *true*
– `FOAF_FALSE`, the constraint represented is *false*

For the last case, the implementation of this constructor is simple since the method `ndd_create_empty` (see Section 4.2) is called with dimension 0. For the first case, we need to call one of the methods `ndd_create_zn_Xsdf` (see Section 4.2) with dimension 0, but they return the NULL pointer. In order to build the NDD representing the *true* constraint, we write a new method:

```
ndd* trex_ndd_create_zn(uint1 r, uint4 n)
```

which deals with the particular case of dimension 0.

5.3 Atomic constraints and their constructor

Atomic constraints are build from a comparison by the following constructor:

```
NddFormula(FoafComp* fc)
```

Its implementation tests first that the comparison is linear. Then, it calls the methods `GetUsedVars`, `GetSValProduct`, `GetCConst`, and `GetCOp` of FOAF to obtain the list of variables used in the comparison, the coefficients of each variable, the constant part and the operator ($=$, $\leq$, $<$) of the comparison. With these informations, it calls one of the methods defined in the file `ndd-equations.h` (see Section 4.2).

If all the coefficients obtained for the comparison are 0, the analysis explained at Remark 2 is applied before the call of NDD library methods. In this way, we avoid to return the NULL pointer.

5.4 Complex constraints and their constructors

The following constructor:

```
NddFormula(FoafOperation O, NddFormula* ndf1, NddFormula* ndf2)
```

builds the NDD representing the constraint `O(ndf1, ndf2)` where `O` may be `FOAF_AND` or `FOAF_OR` which corresponds to `and` and `or` logic operators. The constructor calls the methods described on Section 5.5.
Other constructors are provided to:

- build a copy of a constraint:

```
NddFormula(ndd* nd,vector<int>* p, vector<int>* a, vector<int>* c)
```

- build a NDD implementation fro a FOAF implementation:

```
NddFormula(FoafFormula* ff)
```

5.5 Operations on constraints

Logical operations Three logical operators are implemented on `NddFormula`:

`friend NddFormula Not(const NddFormula& ndf)`
Returns the `NddFormula` representing the negation of `ndf`. The resulting NDD is the difference between the NDD representing the universal set Z^n and the NDD of `ndf`. The dimensions of the resulting NDD points to the same variables than in `ndf`.

`friend NddFormula operator&(const NddFormula&, const NddFormula&)`
Return the conjunction of the two constraints. The difficult point in the implementation is change of dimension for each parameter. Indeed, the intersection of two NDD needs the same dimension for parameters. Moreover, in our implementation, the corresponding dimensions of each parameter must point the same variable, *i.e.* the vectors `m_params` and `m_acc_params` must be the same. To solve this problem we implemented a method to introduce a new dimension in an arbitrary position into a NDD (see Section 6.2):

```
ndd* AddDimensionAtPos(ndd* nd, uint4 p)
```

`friend NddFormula operator|(const NddFormula&, const NddFormula&)`
Return the disjunction of the two constraint. The implementation needs also to have the above of changing the dimension of the NDD used by the constraints.

Tests on constraints Three test operations are implemented:

`bool IsTrue()`
Return true if the constraint is valid.

`bool IsFalse()`
Return true if the constraint is not satisfiable.

`bool IsInclude(const NddFormula* ndf) const`
Return true if the current formula (*i.e.*, `this`) is included (implied) by `ndf`. The implementation computes the formula `Not(ndf) & this` and tests the result with `IsFalse`.

Existential quantifiers elimination This kind of operations is implemented as the projection on the dimensions corresponding to the quantified variables. Two operations have been implemented:

`friend NddFormula* ProjectAll(NddFormula* ndf)`
Returns the constraint obtained after the projection of all variables used, *i.e.*, the *true* or *false* constraint. After applying the projection method of NDD library on all variables, we test if the NDD obtained is empty. Then, the result is *false*, otherwise it is *true*.

`friend NddFormula* ProjectAcc(NddFormula* ndf)`
Returns the constraint obtained after the projection on the dimensions corresponding to the acceleration parameters which unique numbers are greater than `NB_ACC`. These parameters have been creating by the call of `SubstituteAcc` method below.

Conversions to/from strings To read and display formulas, we need to have a translation of `NddFormula` to and from strings. The problem is that the translation of a NDD into a string cannot be done efficiently. So, instead of printing the explicit form of the formula, we chose to print the NDD automaton. Three methods are implemented:

`void DumpDefinition(FILE*) const`
Writes into the file the characteristics of the `NddFormula`, *i.e.*, its dimension, the variables used by the constraint, and the description of the automaton into the `dot` format [5].

`char* ToString() const`
Returns the name of the file on which the description of the constraint is given. This solution has been chosen because of the difficulty of translation into an explicit form of NDD constraints. The methods call `DumpDefinition` above with a file which name is `autN` with `N` a unique number.

`friend NddFormula* ToNddFormula(char*)`
Returns the `NddFormula` corresponding to the entry string.

Specific operations The operations below are need by the exploration algorithm of TREX:

`friend NddFormula* SubstituteAcc(NddFormula* ndf)`
Returns a new `NddFormula` built from `ndf` where all acceleration parameters are renamed to obtain new parameters. This operation is done by adding to each number in the vector `m_acc_params` the maximal number used for acceleration parameters `NB_ACC`.

`void GetUsedVars(list<int>&, list<int>&) const`
Returns the lists of parameters (field `m_params`) *resp.*. of acceleration parameters (field `m_acc_params`).

6 Needs for better use of the NDD library

During the implementation of the NDD interface, we needed supplementary methods on NDDs both for writing the interface and for its improvement. We wrote some of the methods we needed using the existing methods of the NDD library. However, there are some methods that need more effort in order to have an efficient implementation. In the following we list the new methods we added and we make suggestions for the implementation of other methods.

6.1 Methods working on NDD of null dimension

The following methods provided by the NDD library do not work on NDDs of dimension 0:

– The methods for building NDDs presented in Section 4.2 (`ndd_create_*`) return NULL when they are asked to build an NDD of dimension 0.

[5] Format used by the GraphViz tool, `www.graphviz.org`

– The product method (`ndd_product`) of NDDs presented in Section 4.4 called with an operand of dimension 0 returns the other operand. Or, if the operand of dimension 0 represents the *false* constraint, the result should be also *false*.

We write two new methods to avoid the problems above:

`new_ndd_create_zn(int r, int n)` and
`new_ndd_product(ndd* nd1, ndd* nd2)`

These methods deal with the particular case of dimension 0 and then call the corresponding method of the NDD library.

6.2 Adding new dimension

In order to implement binary operations of constraints, we need to align dimension of each NDD and the variables that are pointed by each dimension. For this, we have to add a new component a a given position. We implemented a new method for this:

```
ndd* AddDimensionAtPos(ndd * nd, uint4 p)
```

It adds an arbitrary component at the position `p` (`p`=0 corresponds to add the component in front of the list). The resulting NDD has the dimension of `nd` plus 1.
To implement this method we call the method `ndd_interleave_z` (see Section 4.4) to interleave a new component between each component of `nd`. Then, by successive call to `ndd_projection`, all the components introduced at positions different from `p` are eliminated. This method is expensive and it has to be better implemented.

6.3 Sharing constraints

The experiments done with TREX using the FOAF library show that some constraints appear frequently in different symbolic configurations. For this reason the FOAF library does the hashing of constraints created during the analysis.
To obtain the hashing of NDDs, the following changes should be implemented:

– Although the NDD have a canonical form for a given dimension, this is no more true if the dimension is dynamic. Indeed, two NDDs of different dimensions may represent the same constraint. In order to store only one NDD by constraint, a "normal form" should be defined for NDDs. We propose the following definition for the normal form: "A NDD is in normal form iff its automaton contains a constraint strictly included in Z for each dimension." For example:
  - From all the NDDs representing the sets Z^n (with $n \geq 0$), only the one with $n = 0$ is in the normal form.
  - The constraints $x_1 = 0 \wedge x_2 \leq 0$ and $x_1 = 0 \wedge x_2 > 0$ supply informations about x_1 and x_2. However, the constraint $(x_1 = 0 \wedge x_2 \leq 0) \vee (x_1 = 0 \wedge x_2 > 0)$ built using the previous constraints does not supply any information about x_2.

– In order to do hashing of NDDs in normal form, we need to order NDDs (using, $=$, $<$, and $\leq$) and to obtain a hashing code for each NDD. For the test of equality, the NDD package provides a method testing the equality of languages. For hashing, a more simple test of equality, *e.g.*, the structural equality, may be implemented. This test should be more efficient.

7 Results

We done experiments with the two versions of representations on the example of lift controller. This example contains two processes (the motor of the lift and its controller) and three integer variables (the current floor, the goal floor, and the direction of moving). The number of floors N may be a constant, a parameter, or infinite.

Table 2. Results for lift controller example (1/2).

Nb. floors	2	3	4	5	6	7	8	9	10
with NDDs									
size of NDDs (min/max)	0/0	3/3	4/9	4/9	5/13	5/13	6/17	5/14	7/21
exec. time	0.08	1.97	3.26	3.27	3.57	3.71	3.98	3.94	4.39
with FOAF									
exec. time OMEGA	0.00	4.71	6.58	6.15	5.97	6.14	6.09	6.18	6.13
exec. time *total*	0.04	5.40	7.49	6.89	6.85	6.91	6.93	6.91	6.87

Table 3. Results for lift controller example (2/2)

Nb. floors	100	1000	10000	100000	1000000	∞
with NDDs						
size of NDDs (min/max)	14/53	20/80	27/111	34/143	40/170	3/24
exec. time	11.25	22.43	42.91	73.20	112.64	5.92
with FOAF						
exec. time OMEGA	5.96	6.15	6.17	6.13	6.22	6.62
exec. time *total*	6.87	6.89	6.96	6.90	6.97	7.70

Tables 2 and 3 provides the results of tests. For the version using NDDs, we give the minimal and maximal sizes of NDDs used (*i.e.*, the number of states of the automaton) and the execution time (in seconds). For the version using FOAF, we give the the execution times (in seconds) for the total execution and for the call of OMEGA solver.
Table 2 shows a gain of 30-40% in execution time when the NDD library is used. However, the results given by Table 3 show that in some cases the FOAF library is better. This may be explained by the fact that the size of NDDs increases when the number of floors is a huge constant because the binary coding of this constant (20 bits are needed to code 1000000).

8 Conclusion

We report about the integration in TReX of the NDD library for the symbolic representation of constraints over integer parameters. The benefits of this work are:

- It does a step forward in the modularization of TReX code. The solution proposed for integration of NDDs allows also to integrate other symbolic representation of constraints, for example the RVA representation developed in LASH for constraints over integers and reals.
- We are able now to compare the performances of each representation and so to obtain the best results for a given example.
- The use of the NDD library in TReX point out some improvements and research direction to be explored for this library.

Acknowledgments We thanks Bernards Boigelot and Louis Latour for their help and careful reading of the preliminary version of this report.

References

[AAB99] P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In *Proceedings of 5th TACAS*, volume 1579 of *LNCS*. Springer Verlag, 1999.

[AAB00] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of the 12th CAV*, volume 1855 of *LNCS*, pages 419–434. Springer Verlag, July 2000.

[BCAS01] A. Bouajjani, A. Collomb-Annichini, and M. Sighireanu. Trex: A tool for reachability analysis of complex systems. In *Proceedings of CAV'01*, LNCS. Springer Verlag, 2001.

[Bul98] T. Bultan. *Automated symbolic analysis of reactive systems.* PhD thesis, University of Maryland, 1998.

[Dil89] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the 1st CAV*, volume 407 of *LNCS*, pages 197–212. Springer Verlag, 1989.

[ER92] B.C. Eaves and U.G. Rothblum. Dines-Fourier-Motzkin quantifier elimination and an application of corresponding transfer principles over ordered fields. *Mathematical Programming*, 53:307–321, 1992.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1994.

[Hea99] A.C. Hearn. *REDUCE — User's and Contributed Packages Manual.* Codemist Ltd., February 1999. version 3.7.

[Ome96] Omega Team. *The Omega Library*, November 1996. version 1.1.0.

[WB00] 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 Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *LNCS*. Springer Verlag, 2000.

A Interface for implementation of formulas

```
class FormulaImpl : public FoafTerm {

  friend bool operator==(const FormulaImpl&, const FormulaImpl&);
  friend bool operator<(const FormulaImpl&, const FormulaImpl&);
  friend bool operator<=(const FormulaImpl&, const FormulaImpl&);
  friend size_t Hash(const FormulaImpl&);
  friend FormulaImpl* GetUniqueForm(FormulaImpl*);

  friend FormulaImpl operator&(const FormulaImpl&, const FormulaImpl&);
  friend FormulaImpl operator|(const FormulaImpl&, const FormulaImpl&);
  friend FormulaImpl Not(const FormulaImpl&);

  friend FormulaImpl* ProjectAll(FormulaImpl*);
  friend FormulaImpl* ProjectAcc(FormulaImpl*);
  friend FormulaImpl* SubstituteAcc(FormulaImpl*);

public:
  FormulaImpl();
  FormulaImpl(FoafOperation);
  FormulaImpl(FoafComp*);
  FormulaImpl(FoafOperation, FormulaImpl*, FormulaImpl*);
  virtual ~FormulaImpl();

  /* predicates */
  bool IsTrue() const;
  bool IsFalse() const;
  bool IsInclude(const FormulaImpl&) const;

  /* logic operations */
  FormulaImpl& operator&=(const FormulaImpl&);
  FormulaImpl& operator|=(const FormulaImpl&);

  /* functions rewriting FoafTerm */
  virtual bool IsNormalForm() const;
```

```
  virtual FoafTerm* GetNormalForm();
  virtual void GetUsedVars(list<int>&, list<int>&) const;
  virtual FoafLinearKind GetLinearKind() const;
  virtual char* GetKind() const;
  virtual void DumpDefinition(FILE*) const;
  virtual char* ToString() const;

};
```