Utilisation et interface avec une librairie de représentation des contraintes linéaires par des automates

Stéphanie Delaune

8 juillet 2002

Résumé

La complexité croissante des logiciels ainsi que leur plus large utilisation dans des équipements critiques (avions, trains, voitures,...) demandent de s'assurer de leur bon fonctionnement. Pour cela, des techniques de vérification automatique de programmes ont été développées ces dernieres années. Ces techniques consistent à explorer toutes les exécutions possibles des logiciels afin de contrôler que leurs calculs ou réactions satisfont certaines propriétés. Les logiciels qui implémentent ces techniques utilisent souvent une représentation symbolique des contraintes sur les variables, et c'est le cas du logiciel TReX[1] développé au LIAFA[2] . Mais il est également possible d'utiliser une représentation plus compacte, sous forme d'automate, appelée NDD[3].

[1] Tool for Reachability Analysis of CompleX System
[2] Laboratoire d'Informatique Algorithmique : Fondements et Applications
[3] Number Decision Diagramms

Table des matières

1 Introduction

Dans TReX, les contraintes sont représentées à l'aide de la librairie Foaf[4]. Elle permet de représenter et de simplifier les contraintes linéaires et non linéaires sur des variables réelles ou entières. Pour le travail sur les variables entières, simplifications des contraintes, procédures de décision,... on utilise actuellement le logiciel Omega avec lequel on communique par des fichiers. C'est ce dernier point que l'on souhaite améliorer, en remplacant la représentation actuelle des contraintes sur les entiers par celle offerte par LASH[5]. Dans LASH, les contraintes sont représentées par des automates finis appelés NDD, opérant sur un codage (binaire ou autres) des élements de l'ensemble. Grâce à cette représentation, les opérations et les algorithmes de simplification utilisés sur les automates peuvent être repris pour la simplification et la manipulation des contraintes. Une NDD va permettre de représenter toutes les contraintes pouvant s'exprimer dans l'arithmétique de Presburger, c'est-à-dire toute les contraintes linéaires sur les entiers. Le but de ce TER est de remplacer dans TReX la représentation actuelle des contraintes linéaires sur les entiers avec celle offerte par LASH. Il faut donc implémenter, sous forme d'une librairie, une interface avec LASH, offrant les mêmes fonctionnalités que la librairie Foaf, afin de pouvoir, suivant le type de la contrainte, linéaire ou non, utiliser sa représentation symbolique *FoafFormula*, ou sa représentation sous forme d'automate *NddFormula*. Après une présentation de LASH et une description détaillée du contenu de la librairie *Ndd*, on s'intéressera à l'implémentation de la librairie Nddmodel.C avant d'effectuer une comparaison avec la librairie existante Foafmodel.C.

2 Présentation de LASH

LASH est un outil de vérification symbolique développé à l'Université de Liège. Il permet de représenter les ensembles infinis d'entiers et d'explorer les ensembles infinis d'états, grâce à une représentation par automate fini.

2.1 Architecture

Actuellement, LASH se compose d'une librairie écrite en C, qui fournit les fonctions et les structures permettant de réaliser les tâches suivantes :

la librairie principale : est composée des fichiers nommés lash-xxx. La fonction *lash_init()* permet d'initialiser LASH, et doit être appelée avant toutes autres fonctions. De même, la fonction *lash_end()* devra etre appelée avant de sortir de l'application. Dans le fichier *lash-types.h*, on trouve la définition des types de la Table 1.

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

TAB. 1 – Les types de base de LASH.

[4]First Order Arithmetical Formulas
[5]Liege Automata-based Symbolic Handle

la librairie automate : est composée des fichiers nommés auto-xxx et permet la manipulation des automates finis. Nous ne manipulerons pas ces fonctions directement, mais par l'intermédiaire des fonctions implementées dans la librairie ndd. Seule la fonction *auto_serialize_write_dot_file*, declarée dans auto-io-dots.h, et permettant l'affichage d'un automate au format *dot* est utilisée dans notre librairie.

la librairie ndd : est composée des fichiers nommés ndd-xxx et fournit les fonctions et les structures pour manipuler des ensembles de vecteurs. Les ensembles de vecteurs qui peuvent être représentés par des ndd incluent ceux qui peuvent s'exprimer dans l'arithmétique de Presburger. La représentation par ndd est en réalité légèrement plus expressive que l'arithmétique de Presburger. C'est cette librairie qui nous intéresse plus particulierement et que nous étudierons plus en détails dans la section suivante.

la librairie d'exploration de l'espace d'états : fournit une structure pour représenter les ensembles d'etats et la relation de transition d'une machine à états ; diverses fonctions sont fournies pour calculer les successeurs d'un ensemble d'etats.

2.2 Librairie NDD

On s'intérresse au contenu de la librairie ndd et en particulier aux fonctions utilisées pour l'écriture de l'interface avec TReX dans la librairie Nddmodel.C.

2.2.1 Structures de données

Dans cette librairie, plusieurs structures de données sont déclarées, mais on s'intéresse uniquement à la structure ndd :

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

- attribut *dim* : permet de connaître la dimension de la ndd, c'est-à-dire la dimension des vecteurs encodés.
- attribut *automaton* : est un automate fini déterministe représentant la contrainte. Le type automate est défini dans *lash-auto.h*.
- attribut *base* : permet de connaître la base dans laquelle les entiers sont codés. On utilisera la représentation binaire, mais une autre base, supérieur à 1, est envisageable.
- attribut *properties* : permet de stocker quelques propriétés de la ndd.

NDD_PROP_NOTHING	0x00	rien de particulier
NDD_PROP_SERIAL	0x01	mode de codage des vecteurs
NDD_PROP_MSDF	0x02	les vecteurs sont codés *msdf*[6]

Les deux derniers flags seront systématiquement positionnnés. Ainsi le vecteur (5,3,2), codé en binaire par (0101,011,010) est représenté par le mot (0,0,0)(1,0,0)(0,1,1) (1,1,0) soit 000100011110 dans le codage *msdf*. On peut cependant choisir de coder les vecteurs avec le bit le moins significatif en tête, codage *lsdf*[7]. Le vecteur (5,3,2) sera alors représenté par le mot 110011100000.

[6]Most Signifiant Digit First
[7]Low Signifiant Digit First

2.2.2 Construction de Ndd

Les ndd permettent de représenter des contraintes simples, les constantes *true* et *false*, mais aussi des contraintes plus complexes exprimant des liens entre les différentes variables.

La construction de Ndd simples *(ndd-sets.h)*

- la fonction *ndd * ndd_create_empty(uint1 r, uint4 n, uint1 msdf)* : permet de créer une ndd représentant la formule *false* à n variables. La ndd créée a pour dimension n, les entiers sont codés en base r, et l'automate associé à cette ndd ne reconnaît aucun mot.
- les fonctions *ndd * ndd_create_zn_Xsdf(uint1 r, uint4 n)* : (avec $X \in \{m, l\}$) permettent de créer une ndd représentant l'ensemble Z^n. La ndd créée a pour dimension n, les entiers sont codés en base r selon le codage *msdf* respectivement *lsdf.* L'automate associé à cette ndd reconnaît tous les vecteurs d'entiers.

Remarque 2.1 *Pour représenter la constante* true*, il nous faut une ndd reconnaissant tous les vecteurs d'entiers. On préferera la ndd de dimension 0 dans la mesure où l'automate associé est le plus petit. Il s'agit de l'automate reconnaissant le mot vide.*

La construction de Ndd plus complexes *(ndd-equations.h)*

- les fonctions *ndd* ndd_create_equation_Xsdf(uint1 r, uint4 n, sint4 * c, sint4 b)* : (avec $X \in \{m, l\}$) permettent de construire une ndd représentant l'ensemble des solutions d'une équation. La ndd créée a pour dimension n, les entiers sont codés en base r selon le codage *msdf* respectivement *lsdf.*
  L'automate associé à la ndd reconnaît les vecteurs solution de l'équation :

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

  Remarque 2.2 *Lorsqu'il s'agit d'une équation à 0 variables (*n *= 0), ces deux fonctions retournent NULL. Il serait préférable que la ndd soit créée avec avec une dimension égale à 0 et que l'automate associé soit :*
  – l'automate reconnaissant le mot vide si $b = 0$
  – l'automate reconnaissant le langage vide si $b \neq 0$
- les fonctions *ndd* ndd_create_inequation_Xsdf(uint1 r,uint4 n,sint4 * c,sint4 b)* : (avec $X \in \{m, l\}$) permettent de construire une ndd représentant l'ensemble des solutions d'une inéquation. La ndd créée a pour dimension n, les entiers sont codés en base r selon le codage *msdf* respectivement *lsdf.*
  L'automate associé à la ndd reconnaît les vecteurs solution de l'inéquation :

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

  Remarque 2.3 *Lorsqu'il s'agit d'une inéquation à 0 variables (*n *= 0), ces deux fonctions retournent NULL. Il serait préférable que la ndd soit créée avec* dim *= 0, et que l'automate associé soit :*
  – l'automate reconnaissant le mot vide si $b \geq 0$
  – l'automate vide si $b < 0$

Ces quatre fonctions nous seront particulièrement utiles pour traduire une comparaison (classe *FoafComp*) en une *NddFormula.*

2.2.3 Tests sur les Ndd

Les seuls tests utilisés pour l'écriture de la librairie *Nddmodel* sont le test d'égalité et le test du vide.

– la fonction *int ndd_is_empty(ndd * nd)* : teste si l'ensemble représenté par la ndd passée en argument est l'ensemble vide, *i.e.* si la formule représentée par la ndd est une antilogie. Si oui, la valeur de retour est 1.
– la fonction *ndd_equality(ndd * nd1, ndd * nd2)* : teste l'égalité des ensembles représentés par les deux ndd. Pour ce faire, il faut que les deux ndd codent le même type de vecteurs et qu'elles aient donc la même dimension. Si les deux ndd représentent le même ensemble de vecteurs alors la valeur de retour est 1. Il s'agit donc d'une égalité des langages.

2.2.4 Opérations sur les Ndd

Les différentes opérations que l'on peut réaliser sur les ndd sont les mêmes que celles effectuées sur des automates.

Dans *ndd_operations.h* :

– la fonction *ndd * ndd_product(ndd * nd1, ndd * nd2)* : permet de calculer le produit cartésien des deux ensembles représentés par les ndd passées en argument. Cette fonction va nous permettre d'ajouter de nouvelles dimensions à une ndd, mais uniquement en tête ou en queue.
Soient $\xi(ndi)$ (*resp.* n_i), l'ensemble (*resp.* la dimension) des vecteurs codés par la ndd *ndi*. La ndd obtenue par le produit a une dimension égale à la somme des dimensions des deux ndd passées en argument, elle permet de représenter l'ensemble suivant :

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

Remarque 2.4 *Lorqu'une des deux ndd est de dimension 0, cette fonction retourne systématiquement l'autre ndd. Or, s'il s'agit de la ndd représentant l'ensemble vide, la ndd retournée doit également représentée l'ensemble vide.*

– la fonction *ndd *ndd_difference(ndd *nd1, ndd *nd2)* : permet de calculer la différence entre les deux ensembles représentés par les ndd passées en argument. Pour réaliser cette opération, les deux ndd doivent avoir la même dimension, la ndd retournée représente alors l'ensemble suivant :

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

– la fonction *ndd *ndd_union(ndd *nd1, ndd *nd2)* : permet de calculer l'union des ensembles représentés par les deux ndd passées en argument Pour réaliser cette opération, les deux ndds doivent avoir la même dimension, la ndd retournée représente alors l'ensemble suivant :

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

– la fonction *ndd *ndd_intersection(ndd *nd1, ndd *nd2)* : permet de calculer l'intersection des ensembles représentés par les deux ndd passées en argument. Pour réaliser cette opération, les deux ndd doivent avoir la même dimension, la ndd retournée représente alors l'ensemble suivant :

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

– la fonction *ndd *ndd_interleave_z(ndd *nd, uint4 nl, uint4 nr)* : permet d'insérer de nouvelles variables au milieu des composantes d'une ndd. Dans notre librarie *Nddmodel*, l'appel à cette fonction se fait toujours avec $nl = 1$ et $nr = 0$ (voir 3.2.2). La ndd retournée représente alors l'ensemble suivant :

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

Dans *ndd_projection.h* :
- la fonction *ndd * ndd_projection(ndd * nd, uint4 p)* : retourne une ndd représentant la projection de la p^{ieme} composante. La ndd retournée a donc une composante de moins que la ndd passée en argument, et représente l'ensemble suivant :

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

2.3 Sources de LASH

Les sources de LASH sont disponibles gratuitement pour l'évaluation et l'usage éducatif. Une copie de la version 0.8 peut être téléchargée sur la page de LASH :

http ://www.montefiore.ulg.ac.be/~boigelot/research/lash

2.3.1 Installation

Taper les commandes suivantes :

```
gunzip lash-v0.8.tar.gz
tar xvf lash-v0.8.tar
cd lash-v0.8
make
```

Le répertoire lib contient alors les 3 fichiers ndd-package.o, qdd-package.o et lash-core.o. Le répertoire bin contient les 3 exécutables presburger, siflash et splash.

2.3.2 Commande presburger

```
Usage : presburger [ options ] <source-file>
                 -m <name>  :  Outputs expression (string).
                 -ndd <name>  : Outputs expression (ndd).
                 -dot <name>  : Outputs expression (dot).
> presburger -m test1.str -ndd test1 -dot test1.dot test1.pr
```

Cette commande prend en entrée un fichier contenant une formule, le fichier `test1.pr`, et permet d'obtenir :
- dans `test1..dot` la description de l'automate acceptant le codage binaire des vecteurs satisfaisant la formule
- dans `test1.ndd` la description de l'automate en format ndd
- dans `test1.str` la formule transformée en format interne NDD. Les transformations effectuées permettent d'obtenir des formules dans lesquelles :
  - les variables ont été renommées suivant leur ordre d'apparition
  - seuls les opérateurs $\leq$ et $=$ sont utilisés
  - les quantificateurs FORALL ont été supprimés

Exemples de formules syntaxiquement correctes :
- (x1 - x2 = 2)
- x - y <= 2
- (forall z : (2 < z) or (0 >= z))
- exists x : (x > 2) and (x < 4)
- forall x : (exists (y : ((x = 2*y) or (x = 2*y +1))))

Les transformations effectuées permettent donc d'obtenir des formules utilisant uniquement :
- des variables nommées xi où $i \in N$
- les opérateurs $\leq$ et $=$
- les connecteurs AND, OR et NOT

– le quantificateur EXISTS

On constate quelques problèmes lors de cette transformation.

fichier .pr	fichier .str	ce qu'il faudrait obtenir
x - y = 2	x0 - x1 == -2	$x_0 - x_1 = 2$
x - y <= 2	x0 - x1 <= -2	$x_0 - x_1 \leq 2$
x - y < 2	x0 - x1 <= -3	$x_0 - x_1 \leq 1$
x - 2*y < -2	x0 - 2 x1 <= 1.	$x_0 - 2x_1 \leq -3$
x -2*y < 2	x0 - 2 x1 <= -3	$x_0 - 2x_1 \leq 1$
x -3*y < -5	x0 - 3 x1 <= 4.	$x_0 - 2x_1 \leq -6$
x -3*y < 5	x0 - 3 x1 <= -6.	$x_0 - 3x_1 \leq 4$
x - 2*y <= -2	x0 - 2 x1 <= 2.	$x_0 - 2x_1 \leq -2$
x - 2*y <= 2	x0 - 2 x1 <= -2.	$x_0 - 2x_1 \leq 2$

2.3.3 Commande siflash

```
Usage : siflash [ options ] <source-file>
    Options are  -c          :  No state-space exploration.
                 -m <name>   :  Outputs state machines.
                 -v          :  Verbose
    output.
                 -ndd <name>   :  Outputs ndd (variables).
                 -dot <name>   :  Outputs ndd (variables) (dot-format).
                 -ss           :  save ndd at each step of  exploration.
```

Cette commande prend en entrée un `fichier.if`, et permet d'obtenir :

- dans .dot la description de l'automate représentant les configurations atteignables, *i.e.* les valeurs qui peuvent être prises par les différentes variables du système que l'on a modélisé
- dans .ndd la description de l'automate en format ndd

Exemple : ***contenu du fichier test1.if***

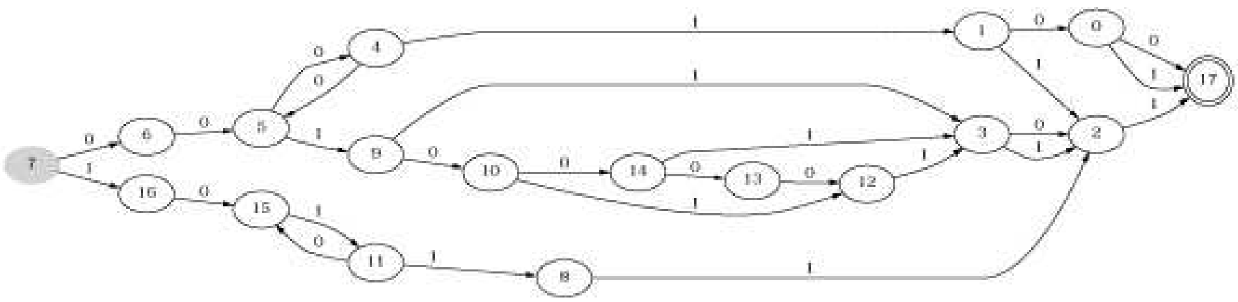
```
system test;
var
x(0) : int;
y(2) : int;
process one;
state
s0 :init;
s1 ;
s2 ;
transition
from s0 do x := 4, y := 3 to s1;
from s1
if x <=  5 do x := x + 5 to s2;
from s2
if  x  >= 5 do x := x - 6 to s1;
count(global.x);
count(global.y);
count(global.x,global.y);

> siflash -m test1.str -dot test1.dot test1.if
*** Program validated.
COUNT(2): global.x global.y :
```

```
                 12
COUNT(1): global.y :
                 2
COUNT(0): global.x :
                 11
```

Ces premiers resultats nous informent que 12 valeurs sont possibles pour le vecteur (x,y), la variable x peut prendre 11 valeurs, et y peut prendre 2 valeurs différentes.

L'automate décrit dans le fichier `test1.dot` est le suivant :



Remarque 2.5 *Cet automate reconnaît une infinité de mots dans la mesure où il contient des cycles. Cependant, comme plusieurs mots permettent de coder le même vecteur, cet automate représente un ensemble fini de configurations possibles. En effet, tous les mots de la forme (00)*000100 représentent un codage du vecteur (0,2).*

Voici le tableau des mots reconnus par cet automate :

mots reconnus	x en base 2	y en base 2	x en base 10	y en base 10
000100	000	010	0	2
000101	000	011	0	3
000111	001	011	1	3
0010000101	01000	00011	8	3
0010000111	01001	00011	9	3
00100101	0100	0011	4	3
00100111	0101	0011	5	3
00101101	0110	0011	6	3
00101111	0111	0011	7	3
001101	010	011	2	3
001111	011	011	3	3
101111	111	011	-1	3

Un vecteur admet toujours une infinité de codages. Il suffit en effet de répéter le (ou les) bit(s) de signe. Dans ce tableau, seuls les mots sans répétition inutile des bits de signe ont été conservés.

Un petit travail réalisé à la main a permis de calculer les configurations accessibles de ce système. Les résultats obtenues sont présentés dans le tableau suivant :

état	s_0	s_1	s_2	s_1	s_2	s_1	s_2	s_1	s_2	s_1	s_2	s_1	s_2
x	0	4	9	3	8	2	7	1	6	0	5	-1	4
y	2	3	3	3	3	3	3	3	3	3	3	3	3

2.3.4 Commande splash

```
Usage : spflash [ options ] <source-file>
    Options are  -c          :  No state-space exploration.
                 -m <name>   :  Outputs state machines.
                 -v          :  Verbose
```

3 Présentation de la librairie NddFormula

La librairie *NddFormula* doit permettre de représenter des contraintes linéaires sur les entiers, comme le faisait déjà la librairie *FoafFormula*. La plupart des méthodes implementées dans la classe *FoafFormula* doivent donc être reprises dans la classe *NddFormula*. TReX pourra ainsi appeler de façon transparente les méthodes de l'une ou l'autre des librairies.

3.1 Données membres

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

      ....
 };
```

Une NddFormula est une contrainte linéaire sur des variables entières. Ces variables peuvent être :

- des paramètres, stockés dans le tableau Parameters
- des paramètres d'accélération, repérés uniquement par un numéro
- des compteurs, repérés également par un numéro dans le tableau global des compteurs

Afin de pouvoir associer à chacune des dimensions de la ndd la variable à laquelle elle se rapporte, il nous faut stocker ces informations. On choisit de stocker :

- les indices des paramètres utilisés dans un vecteur d'entiers, appelé *m_params*
- les numéros des paramètres d'accélération dans un vecteur d'entiers, appelé *m_acc_params*
- les indices des compteurs dans un vecteur d'entiers, appelé *m_compteurs*.

Dans la librairie ndd, aucune fonction ne permet de permuter les composantes d'un vecteur de manière efficace. On utilise donc un ordre sur les variables qui est le suivant :

$$\text{paramètres} < \text{paramètres accélération} < \text{compteurs}$$

A l'intérieur d'un même type, les variables sont classées suivants leurs indices ou numéros.

Cas particuliers :

- La formule *true* sera représentée par une ndd de dimension 0 et l'automate associé ne comportera que un seul état à la fois initial et final. Il reconaîtra donc le mot vide, *i.e.* tous les vecteurs de dimension 0. Les vecteurs *m_params, m_acc_params* et *m_compteurs* seront vides.
- La formule *false* sera représentée par une ndd de dimension 0, l'automate associé n'aura aucun état. Il reconnaîtra donc le langage vide. Les vecteurs *m_params*, *m_acc_params* et *m_compteurs* seront vides.

3.2 Suggestions de modifications pour la librarie Ndd

3.2.1 Fonctions à modifier

Certaines fonctions dans la librairie ndd de Lash ne sont pas utilisables pour des ndd de dimension 0. C'est le cas des fonctions suivantes :

- ndd* ndd_create_zn_msdf(uint1 r, uint4 n)
- ndd* ndd_create_zn_lsdf(uint1 r, uint4 n)
- ndd* ndd_create_equation_msdf(uint1 r, uint4 n, sint4 * c, sint4 b)
- ndd* ndd_create_equation_lsdf(uint1 r, uint4 n, sint4 * c, sint4 b)
- ndd* ndd_create_inequation_msdf(uint1 r, uint4 n, sint4 * c, sint4 b)
- ndd* ndd_create_inequation_lsdf(uint1 r, uint4 n, sint4 * c, sint4 b)

Ces différentes fonctions renvoient un pointeur NULL lorqu'elles sont appelées avec $n = 0$.

La fonction :

- ndd * ndd_product(ndd * nd1, ndd * nd2)

retourne systématiquement l'autre ndd lorqu'une des deux ndd est de dimension 0. Or, si la ndd de dimension 0 représente la constante *false*, la ndd retournée doit être une ndd dont l'automate associé est vide.

Pour contourner ces problèmes, nous avons implementé deux fonctions :

- ndd * mon_ndd_create_zn(int r, int n)
- ndd * mon_ndd_product(ndd * nd1, ndd * nd2)

Ces deux fonctions traitent les cas particuliers où la dimension d'une des ndd est 0, puis appellent la fonction Lash correspondante.

3.2.2 Une fonction sur les ndd à implémenter efficacement

Pour faire le *or* ou le *and* entre deux NddFormula, il nous faut réaliser l'union ou l'intersection des deux ndd. Mais avant de réaliser ces opérations, il faut s'assurer que les deux ndd soient de même dimension et que chacune des composantes se réfère à la même variable. Pour cela, on procède à un redimensionnement des deux ndd en ajoutant des composantes arbitaires. Il nous faut donc pouvoir ajouter une composante arbitraire à une position précise, c'est exactement ce que la fonction *AddDimensionAtPos* permet de faire.

*ndd * AddDimensionAtPos(ndd * nd, uint4 pos)*

Une composante arbitraire est ajoutée à la position *pos* ($pos = 0$ correspond à un ajout en tête). La ndd retournée à une dimension égale à $nd \rightarrow dim + 1$.

Cette fonction, définie dans *Nddmodel.C*, fait appel à la fonction *ndd_interleave_z* de la librairie *ndd* pour intercaler une composante arbitraire entre chacune des composantes de la ndd, puis par des appels à *ndd_projection*, toutes les composantes introduites à une position différente de celle souhaitée sont supprimées. Cette opération est donc assez coûteuse et doit pouvoir être améliorée.

3.3 Constructeurs de la classe NddFormula

Il nous faut pouvoir contruire :

- les formules de base, *true* et *false*
- une formule à partir d'une *FoafComp* (formule atomique n'utilisant aucun opérateur logique)
- une formule à partir d'autres formules et d'un opérateur logique comme *and*, *or*, *not*

3.3.1 Constructeurs pour *false* et *true*

`NddFormula(FoafOperation O)`

La ndd crée a pour dimension 0, les vecteurs *m_params*, *m_acc_params* et *m_compteurs* sont initialisés à vide. Si `O` vaut :
- `FOAF_FALSE`, alors l'automate ne reconnaît aucun vecteur, c'est-à-dire le langage vide
- `FOAF_TRUE`, alors l'automate reconnait le mot vide, *i.e.*tous les vecteurs de dimension 0

Remarque 3.1 *Le constructeur par défaut,* NddFormula() *construit une NddFormula représentant la constante* true*, comme le ferait un appel à* NddFormula(FOAF_TRUE).

3.3.2 Constructeurs à partir d'un terme *Foaf*

`NddFormula(FoafComp * fc)`

Le principe est de récupérer les coefficients devant chacune des variables utilisées dans la FoafComp. On utilise pour cela les méthodes :
- `GetUsedVars(map<int,int> PL, set<int> AL)` permettant de récupérer les variables utilisées dans la FoafComp
- `GetSValProduct(FoafSum * fs)`, permettant de récupérer le coeffcient d'un produit dans une FoafSum
- `GetCConst()`, permettant de récupérer le membre gauche de la FoafComp
- `GetCOp()`, permettant de récupérer le signe, $=$, $<$ ou $\leq$, utilisé dans la FoafComp

On fait ensuite appel à l'une des quatre fonctions définies dans *ndd-equations.h* :
- ndd* ndd_create_equation_msdf(uint1 r, uint4 n, sint4 * c, sint4 b)
- ndd* ndd_create_equation_lsdf(uint1 r, uint4 n, sint4 * c, sint4 b)
- ndd* ndd_create_inequation_msdf(uint1 r, uint4 n, sint4 * c, sint4 b)
- ndd* ndd_create_inequation_msdf(uint1 r, uint4 n, sint4 * c, sint4 b)

Remarque 3.2 *Si les coefficients calculés pour chacune des variables est 0, la ndd aura alors une dimension égale à 0. Ce cas particulier est traité avant l'appel à l'une des quatre fonctions ci-dessus qui renverrait un pointeur NULL.*

Afin de réaliser des tests, deux autres constructeurs ont été implémentés :
- `NddFormula(FoafAndFormumla * faf)` : permet de construire une NddFormula à partir d'une FoafAndFormumla, ce constructeur fait appel au précédent.
- `NddFormula(FoafFormula * ff)` : permet de construire une NddFormula à partir d'une FoafFormula, ce constructeur fait également appel au constructeur précédent.

3.3.3 Le constructeur de copie

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

Ce constructeur permet de créer une NddFormula à partir de la valeur de ses différents attributs. Les différents attributs de la NddFormula sont initialisés de la façon suivante :
- m_ndd = nd
- m_params = p
- m_acc_params = a
- m_compteurs = c

3.3.4 Constructeurs à partir d'un opérateur logique

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

Ce constructeur fait appel à l'opérateur & ou | (voir 3.4.2).
- Si `O` = `FOAFOP_AND`, alors la ndd retournée est `*ndf1` & `*ndf2`
- Si `O` = `FOAFOP_OR`, alors la ndd retournée est `*ndf1` | `*ndf2`

3.4 Méthodes de la classe NddFormula

3.4.1 Concernant la table de hashage

Les tests réalisés avec la librairie Foaf ont montrés que les mêmes formules revenaient souvent. L'utilisation d'une table de hashage pour stocker les formules sans copie a donc permis de réaliser un gain en espace mémoire. Pour faire de même avec la librairie Ndd, il nous faut une fonction d'égalité et une fonction d'ordonancement sur les ndd. Pour l'égalité, on peut utiliser l'égalité des langages, mais une égalité structurelle serait suffisante et peut-être moins coûteuse. Pour mettre en place cette table de hashage, les fonctions à implémenter sont les suivantes :
- friend bool operator==(const NddFormula&, const NddFormula&)[8]
- friend bool operator<(const NddFormula&, const NddFormula&)
- friend bool operator<=(const NddFormula&, const NddFormula&)
- friend size_t Hash(const NddFormula&)
- friend NddFormula* GetUniqueForm(NddFormula*)

3.4.2 Opérations sur les formules

Afin de pouvoir combiner les NddFormula, les trois opérateurs logiques *not*, *and* et *or* ont été implémentés.
- *friend NddFormula Not(const NddFormula& ndf)*
  Cette fonction retourne une NddFormula représentant la négation de la NddFormula passée en argument. Pour cela, on calcule la ndd représentant la différence entre l'ensemble Z^n et l'ensemble représenté par la NddFormula passée en argument. Chacune des dimensions de la NddFormula se réfère toujours à la même variable, les vecteurs *m_params*, *m_acc_params* et *m_compteurs* restent inchangés.
- *friend NddFormula operator&(const NddFormula&, const NddFormula&)*
  La principale difficulté pour implémenter cette fonction est le redimmensionnement des deux NddFormula. En effet, pour réaliser l'intersection de deux ndd, il faut qu'elles aient même dimension et que chaque composante concerne la même variable. Autrement dit, les vecteurs *m_params*, *m_acc_params* et *m_compteurs* des deux NddFormula doivent être identiques. Pour cela, il faut introduire une composante arbitraire lorsque une variable apparaît dans une des deux NddFormula mais pas dans l'autre. Pour réaliser le redimenssionnement, on utilise la fonction *ndd * AddDimensionAtPos(ndd * nd, uint4 pos)* (voir 3.2.2).
- *friend NddFormula operator|(const NddFormula&, const NddFormula&)*
  Comme pour la conjonction de deux NddFormula, la disjonction demande de faire un redimenssionnement des deux NddFormula avant de réaliser l'union des deux ndd.

3.4.3 Tests sur les formules

Dans la librairie Nddmodel.C, trois tests ont été implémentés :

[8] égalité structurelle

- le test *bool IsTrue()* : permet de savoir si une NddFormula est valide, c'est-à-dire si elle est vraie pour toutes les assignations des variables la concernant.
- le test *bool IsFalse()* : permet de savoir si une NddFormula est une antilogie, c'est-à-dire si elle est fausse pour toutes les assignations. On dira aussi que cette formule n'est pas satisfaisable.
- le test d'inclusion, *bool IsInclude(const NddFormula * ndf)* : permet de savoir si l'ensemble représenté par *this* est inclus dans l'ensemble représenté par *ndf*. Pour réaliser cette opération, il faut après redimensionnement des ndd, tester si la formule $Not(ndf) \wedge this$ est une antilogie, grâce à la méthode *IsFalse()*.

3.4.4 Elimination des quantificateurs existentiels

L'élimination des quantificateurs existentiels dans une formule revient à faire une projection pour éliminer les variables quantifiées. On obtient alors une ndd de dimension plus petite. Deux types de projection ont été implémentés :

- la méthode *friend NddFormula * ProjectAll(NddFormula * ndf)*
  On réalise une projection sur toutes les variables, la NddFomula retounée est soit la constante *true*, soit la constante *false*. Si l'ensemble représenté par la ndd est vide alors cette formule n'est pas satisfaisable et la valeur de retour est *NddFormula(FOAF_FALSE)*, sinon la NddFormula retournée est *NddFormula(FOAF_ TRUE)*. On réalise donc un test de satisfaisabilité.
- la méthode *friend NddFormula * ProjectNP(NddFormula * ndf)*
  Cette méthode permet de projeter afin d'éliminer tous les paramètres d'accélération ayant un numéro supérieur à NB_ACC. Ces paramètres ont été crées lors d'un appel à *friend NddFormula* SubstituteAcc(NddFormula*)* (voir 3.4.7).

3.4.5 Affichage et saisie d'une NddFormula

La librairie *Foaf* offre les fonctions suivantes :

- *virtual void DumpDefinition(FILE*) const* permet d'obtenir la description de la FoafFormula dans un fichier
- *virtual char* ToString() const* permet d'obtenir la description de la FoafFormula sous forme d'une chaîne de caractères
  Autrement dit, ces deux premières fonctions permettent de reconstituer la formule. Or cette opération ne peut pas être implementée efficacement pour les ndd. Dans *Nddmodel.C*, ces deux fonctions seront implémentées mais permettront simplement d'obtenir dans un fichier la description de l'automate au format `.dot`.
- *friend FoafFormula* ToFormula(char *)* permet de parser une chaîne de caractère pour construire une FoafFormula. Cette fonction sera implémentée dans *Nddmodel.C* sous le nom de *ToNddFormula*.

La librairie *Nddmodel.C* doit donc offrir dans la mesure du possible les mêmes fonctionnalités :

- la fonction *virtual void DumpDefinition(FILE*) const* permet d'écrire dans le fichier passé en argument toutes les caractéristiques de la NddFromula, sa dimension, la variable associée à chacune des composantes, et la description de l'automate au format `.dot`.
- la fonction *friend NddFormula* ToNddFormula(char*)* permet de parser une chaîne de caractère afin de construire une NddFormula. Un lexeur *ndd.lex.l* et un parseur *ndd.yacc.y* ont donc été réalisés sur le même modèle que le lexeur et le parseur pour les FoafFormula.
  Il est également possible de parser le contenu d'un fichier en utilisant la fonction *NddFormula* NddParserLoadSystem(char*)* qui prend le nom du fichier à parser en

paramètre. Comme pour Foaf, ces deux fonctions ne peuvent être utilisés simultanément. Pour l'utilisation de la librairie par TReX, l'option -D_NDD_INSTALL doit être positionné lors de la compilation afin que la fonction *ToNddFormula* soit utilisable. Si l'option n'est pas positionnée au moment de la compilation, seul le parsing d'une formule à partir d'un fichier sera utilisable.
– la fonction *virtual char* ToString() const* fait un simple appel à *DumpDefinition(FILE*)*, et écrit dans un nouveau fichier `autNN` à chaque appel (`NN` est un entier).

3.4.6 Forme normale d'une formule

L'interface *Foafmodel.h* proposait la mise sous forme normale d'une formule par l'intermédiaire des fonctions :
– *virtual bool IsNormalForm() const*
– *virtual FoafTerm* GetNormalForm()*
Les automates associés aux ndd sont minimisés et déterminisés de façon systématique par Lash, mais deux NddFormula ayant des dimensions différentes peuvent représenter la même formule. On dira qu'une NddFormula est sous forme normale si toutes les dimensions apportent une information.
Exemples :
Toutes les NddFormula représentant l'ensemble Z^n sont des représentantes de la formule *true*, seule celle de dimension 0 est sous forme normale.

La formule $x_1 = 0 \wedge x_2 \leq 0$ apportent des informations sur x_1 et x_2 ; de même pour la formule $x_1 = 0 \wedge x_2 > 0$. En revanche, la fomule $(x_1 = 0 \wedge x_2 \leq 0) \vee (x_1 = 0 \wedge x_2 > 0)$ construites à partir des deux formules précédentes n'apportent aucune information sur x_2.

La mise sous forme normale doit donc permettre la suppression de ces composantes inutiles.

3.4.7 Autres fonctions

– la fonction *friend NddFormula* SubstituteAcc(NddFormula*)* permet de créer une nouvelle NddFormula à partir d'une autre, seuls les paramètres d'accélération sont renommés. Il suffit donc de changer le contenu du vecteur *m_acc_params*.
– la fonction *virtual void GetUsedVars(list<int>&, list<int>&) const* récupère dans la première liste les indices des paramètres stockés dans le vecteur *m_params* et dans la seconde liste les numéros des paramètres d'accélération stockés dans le vecteur *m_acc_params*

4 Comparaison des performances

Contrairement à la librairie *NddFormula*, *FoafFormula* est très générale dans la mesure où elle permet de traiter :
– les variables entières et réelles
– les contraintes linéaires et non linéaires
Cependant elle fait appel à des solveurs extérieurs comme les logiciels *Omega* et *Reduce* pour résoudre les formules et faire les simplifications. La communication s'établit alors par l'intermédiaire de fichiers et ralentit donc le déroulement de l'exécution. Une analyse en temps sur un exemple qui modélise un ascenseur montre que 80% du temps est utilisé par *Omega* (processus child).

Remarque 4.1 *La taille d'une ndd correspond au nombre d'états que possède l'automate qui lui est associé.*

nombre d'étages	2	3	4	5	6	7	8	9	10
lib Ndd									
taille des ndd (min/max)	0/0	3/3	4/9	4/9	5/13	5/13	6/17	5/14	7/21
temps d'exécution	0.08	1.97	3.26	3.27	3.57	3.71	3.98	3.94	4.39
lib Foaf									
temps d'exécution *self*	0.04	0.69	0.91	0.74	0.88	0.77	0.84	0.73	0.74
temps d'exécution *child*	0.00	4.71	6.58	6.15	5.97	6.14	6.09	6.18	6.13
temps d'exécution *total*	0.04	5.40	7.49	6.89	6.85	6.91	6.93	6.91	6.87

Tab. 2 – Démonstration sur la modélisation du fonctionnement d'un ascenceur (1/2)

nombre d'étages	100	1000	10000	100000	1000000	∞
lib Ndd						
taille des ndd (min/max)	14/53	20/80	27/111	34/143	40/170	3/24
temps d'exécution	11.25	22.43	42.91	73.20	112.64	5.92
lib Foaf						
temps d'exécution *self*	0.91	0.74	0.79	0.77	0.75	1.08
temps d'exécution *child*	5.96	6.15	6.17	6.13	6.22	6.62
temps d'exécution *total*	6.87	6.89	6.96	6.90	6.97	7.70

Tab. 3 – Démonstration sur la modélisation du fonctionnement d'un ascenceur (2/2)

L'utilisation de la librairie *NddFormula* a permis sur cet exemple de réaliser un gain en temps de l'ordre de 30 à 40%.

Pour les valeurs de la table 3, la librairie *NddFormula* ne permet pas une exécution plus rapide, ces résultats peuvent s'expliquer par :

- la taille des ndd : les automates associés aux ndd ont un nombre d'états de plus en plus important en raison de la taille du codage binaire du nombre d'étage. En effet, si 4 bits suffisent pour coder l'entier 7, il faut environ 20 bits pour coder l'entier 1 000 000.
- De plus, la mise en place d'une table de hashage pour stocker les ndd permettra de stocker les formules sans copie et donc d'améliorer le temps d'exécution.

5 Conclusion

Le but de ce TER a été d'utiliser le pouvoir des automates pour la simplification et la manipulation des contraintes. Ces automates, appelée NDD, opérent sur le codage binaire des éléments de l'ensemble et permettent de représenter toutes les contraintes pouvant s'exprimer dans l'arithmétique de Presburger, c'est-à-dire toute les contraintes linéaires sur les entiers. Il serait également intéressant d'utiliser une représentation similaire pour les contraintes sur les réels, cette fonctionnalité est offerte par la librairie RVA de Lash.

Pour avoir des informations supplémentaires, consulter :

- la page de LASH : http ://www.montefiore.ulg.ac.be/˜boigelot/research/lash
- la page de TReX : http ://www.liafa.jussieu.fr/˜sighirea/trex