

20 Juin 2008

---

---

**Implémentation d'un algorithme d'unification pour les termes  
schématisés à exposants entiers**

---

---

Rapport de projet de spécialité  
Mathieu Guillame-Bert et Gabriel Synnaeve



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Conception et Architecture</b>	<b>5</b>
2.1	Technologies . . . . .	5
2.2	Architecture du code . . . . .	5
2.3	Parsing . . . . .	6
<b>3</b>	<b>Unification des termes avec exposants entiers</b>	<b>7</b>
3.1	Article de Hubert Comon (On unification of terms with integer exponents) . . .	7
3.2	Validation du fonctionnement . . . . .	7
<b>4</b>	<b>Résolution des contraintes numériques</b>	<b>9</b>
4.1	Présentation du problème . . . . .	9
4.2	Développement de notre propre algorithme de résolution . . . . .	9
4.3	Automates pour l'arithmétique de Presburger . . . . .	11
<b>5</b>	<b>Conclusions</b>	<b>14</b>
<b>6</b>	<b>Annexes</b>	<b>15</b>
6.1	Manuel Utilisateur - man . . . . .	15
6.2	Diagramme de classes . . . . .	17
6.3	Exemple de sortie générée avec l'option -latex . . . . .	18
6.4	Code source . . . . .	19
6.5	Fichiers d'entête . . . . .	19
6.5.1	automata.h . . . . .	19
6.5.2	config.h . . . . .	20
6.5.3	display.h . . . . .	20
6.5.4	exception.h . . . . .	20
6.5.5	main.h . . . . .	21
6.5.6	synnaeve.guillame.bert.h . . . . .	21
6.5.7	unification.h . . . . .	22
6.6	Fichiers de code . . . . .	23
6.6.1	automata.h . . . . .	23
6.6.2	config.h . . . . .	26
6.6.3	display.h . . . . .	28
6.6.4	exception.h . . . . .	30
6.6.5	main.h . . . . .	31
6.6.6	parser.h . . . . .	31
6.6.7	parser.ll . . . . .	46
6.6.8	parser.yy . . . . .	48
6.6.9	synnaeve.guillame.bert.h . . . . .	50
6.6.10	unification.h . . . . .	56
6.7	En chiffres . . . . .	78

# 1 Introduction

L'unification est l'un des concepts centraux de la logique des prédicats (logique du premier ordre).

Elle consiste à trouver, pour deux termes  $t_1$  et  $t_2$ , la substitution  $\sigma$  (unificateur le plus général) tel que  $\sigma(t_1) = \sigma(t_2)$ . De manière un peu plus intuitive, la résolution d'équation est une forme d'unification. Il est à noter que, contrairement à la plupart des équations que l'on a l'habitude d'utiliser, l'unification considère les variables et les fonctions qu'elle traite indépendamment de leurs natures (on ne considère donc pas d'éventuelles propriétés sur ces objets). Dans ce sens, l'unification ne pourra pas assimiler 1 à *division*(2, 2).

L'unification "de base" ne traite que les variables, les fonctions d'arité fixe et les constantes (fonctions d'arité nulle). Sur cette grammaire, l'unification est décidable : il est mécaniquement possible de calculer en temps fini s'il existe ou non un unificateur, et, dans le premier cas, de le calculer. Le but de notre stage a été d'implémenter une extension de cette grammaire par l'ajout et le support d'une syntaxe avec "exposants" (cette notion sera précisée plus loin), ceci, bien sûr, en gardant la décidabilité. Cette forme d'extension de l'unification a déjà été étudiée, mais n'a jamais été implémentée. Du moins, il n'y a pas de travaux d'implémentation connus ni de Nicolas Peltier, ni de nous-même.

Pour cela, nous avons utilisé le travail réalisé par *Hubert Comon en 1993 [1]* sous le nom "On unification of terms with integer exponents" dans lesquels il définit et analyse cette extension grammaticale, les *i-terms*. De manière pratique, notre travail a été de reprendre son article, d'implémenter ce qu'il y a décrit et de proposer une interface pour l'utilisateur (les résultats de l'unification n'étant pas toujours utilisables directement - utilisation des automates de Presburger pour la résolution de contraintes sur les variables numériques). Le besoin de cette implémentation est venu du domaine de la preuve automatique.

Voici quelques exemples simples d'unification et d'unification avec termes à exposants entiers.

Note : les variables sont en majuscules et les constantes en minuscules.

$$f(X, a) = f(b, Y)$$

L'unification est possible avec  $X = b$  et  $Y = a$ .

$$f(X, a) = g(b, Y)$$

L'unification est impossible car rien ne nous dit que  $f = g$ .

$$f(X, Z) = g(f(Y), h(X))$$

L'unification est possible avec  $X = f(Y)$ ,  $Z = h(X)$  et  $Y$  libre.

L'ajout des termes avec exposants entiers augmente la puissance du langage mais rend plus complexe la recherche de solution d'unification.

## Notation à exposant

La notation suivante sera employée pour désigner certaines formes :

- $u[v]_p$  représentera une séquence  $u$  ayant une sous-séquence  $v$  à la position  $p$ .

- $u[\diamond]^N.v$  représente le “dépliage” de  $N$  fois  $u$  sur  $v$ .

### Exemple 1

La forme  $f(a, g(h(b)))$  pourra être assimilée à  $u[v]_p$  avec  $u = f(a, g(\diamond))$ ,  $v = h(b)$  et  $p = \{2, 1\}$ .

### Exemple 2

Si  $N = 1$ ,  $u[\diamond]^N.v$  revient à remplacer  $\diamond$  par  $v$  dans  $u$ .

Pour un  $N > 1$ ,  $u[\diamond]^N.v = u[\diamond]^{N-1}.(u[\diamond]^1.v)$

$$f(\diamond, a)^2.X = f(f(\diamond, a), a)^1.X = f(f(f(\diamond, a), a), a)^0.X = f(f(f(X, a), a), a)$$

**Note :** Les  $N$  ne sont pas forcément définis ie. ce sont des variables numériques libres (pour l’algorithme : dans  $\mathbb{N}^*$ ). Les contraintes qui apparaîtront au cours de l’unification permettront de définir, dans le cas où il en existe, les valeurs possibles qu’elles peuvent prendre.

$$f(\diamond)^{N1}.Y = f(\diamond)^{N2}.Y$$

L’unification est possible avec  $N1 = N2$  et  $Y$  libre.

$$f(\diamond)^N.a = f(f(\diamond))^M.a$$

L’unification est possible avec  $N = 2.M$ .

$$f(g(f(\diamond))^N.g(a)) = f(g(f(g(\diamond))))^M.a$$

L’unification est possible avec  $N = M = 1$  ou avec  $N = 2.H + 1$  et  $M = H + 1$ , ou  $H$  est une variable libre (dans  $\mathbb{N}^*$ ) .

## 2 Conception et Architecture

Avant-propos : Il peut être très intéressant pour le futur développeur de se documenter via la documentation HTML Doxygen du projet qui contient nos commentaires sur les attributs et méthodes des classes et qui donne des diagrammes de classe clickables.

### 2.1 Technologies

Pour l'implémentation de l'algorithme, nous avons choisi d'utiliser le langage C++ pour : sa vitesse, le support de la programmation orientée objet, la disponibilité d'une bibliothèque d'analyse lexicale et syntaxique (flex/bison) simple à utiliser et, surtout, pour l'expérience que nous en avons déjà.

L'application produite se présente donc sous la forme d'un unique exécutable, fonctionnant en mode console, supportant une syntaxe d'option à la unix et n'utilisant aucune fonctionnalité non portable. Notre programme a été testé sous GNU/Linux (noyau 2.6), Mac OS X (10.5), et Windows (XP, Vista). L'injection et la récupération de données se font par l'entrée et la sortie standards, dans un format textuel facilement exploitable (utilisation facile par des codes externes). L'ensemble des options disponibles sont décrites dans la page du man et par l'appel du binaire avec l'option -help (-h).

Un autre mode de sortie a également été mis en place : avec l'option -latex (-l), le programme donne les résultats avec la syntaxe latex (utilisation simplifiée pour les publications). Un exemple complet est donné en fin de rapport.

### 2.2 Architecture du code

De manière pratique, le code est divisé comme suit :

- Gestion des options
- Affichage
- Erreurs
- Parseur
- Unificateur
- Analyseur d'arithmétique de Presburger (Automate)

L'architecture modulaire est basée sur le concept d'objet. Ceux-ci sont imbriqués de façon à garantir le fonctionnement du programme. Un développeur extérieur au projet pourra donc reprendre notre librairie et interfacer les modules à sa convenance. Par exemple, il pourra ne pas acquérir les données à partir du parser mais directement construire les équations grâce aux fonctions prévues à cet effet. On trouve un diagramme de classes du projet en annexe.

Pour illustrer ce propos, voici le contenu du fichier “main.cpp”, on pourra utiliser la documentation pour plus d'information sur l'utilisation des fonctions.

```
/**
 * Main file of the unificator
 * @author Mathieu Guilleme-Bert and Gabriel Synnaeve
 */
#include <stdio.h>
#include <iostream>
#include <fstream>
#include "config.h"
#include "exception.h"
#include "display.h"
#include "parser.h"
#include "unification.h"
```

```

int main(int argc, char *argv[])
{
    try
    {
        Configuration *config = globconfig = new Configuration(argc, argv);
        Display::Head(config);
        Parser *parser = new Parser(config);
        Unification *unification = new Unification(config, parser);
        unification->run();
        Display::displayResult(unification);
        Display::Foot(config);
    }
    catch (exception& e)
    {
        cerr << endl << e.what() << endl;
    }
    return 0;
}

```

Une modification plus profonde de l'algorithme peut aussi se faire une fois que l'architecture intime du programme comprise. Par exemple, si le programmeur veut ajouter une règle de réécriture, il lui suffira d'écrire sa fonction (qui travaillera avec la structure d'arbre) et de l'ajouter dans l'unificateur aux règles qui sont vérifiées. (unificateur.cpp - L.1857)

```

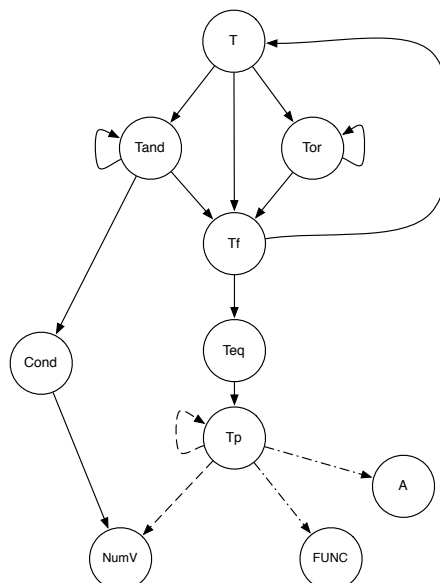
if(!tryApplyR1())
if(!tryApplyR2())
if(!tryApplyR3())
if(!tryApplyR4())
if(!tryApplyR5())
    break;

```

De manière générale, l'ensemble du code est ainsi extensible, par un bon découpage et de par sa conception où nous avons ajouté les fonctionnalités pas à pas (intégration continue) après avoir conçu une structure évolutive.

## 2.3 Parsing

Le programme dispose de sa propre analyse lexicale et syntaxique. Le lexique est décrit dans le fichier *parser.ll* et la syntaxe dans *parser.yy*. La syntaxe reconnue en entrée respecte au mieux la littérature existante sur le sujet. Cependant, certaines formes non admises sont détectées et évincées lors du processus d'unification, comme cela est fait dans l'algorithme d'unification pour les i-termes. Ci-dessous un document de conception de la grammaire reconnue pour les termes à exposants entier.



### 3 Unification des termes avec exposants entiers

#### 3.1 Article de Hubert Comon (On unification of terms with integer exponents)

Notre travail a commencé avec l'article d'Hubert Comon "On unification of terms with integer exponents" [1], lequel nous a fourni l'ensemble des règles pour cette extension de l'unification ainsi que les preuves de leurs complétude, correction et arrêt.

Le principe des ces règles est de décomposer/dérouler les expressions pour retrouver une forme résolue du type  $\exists \vec{n}. N_1 = E_1 \wedge \dots \wedge N_k = E_k \wedge x_1 = t_1 \wedge \dots \wedge x_n = t_n$  qui serait la solution. Encore faut-il vérifier, par les automates dans l'arithmétique de Presburger, que ces contraintes sont satisfaisables.

Ces règles, ont été, pour certaines plus que pour d'autres, relativement dures à mettre en oeuvre de par leur complexité. Bien que cette difficulté n'ait été qu'algorithmique, c'est sur elle que nous avons passé le plus de temps.

##### Exemple de règle : Unfold 1

$$s = t[\diamond]_p^N . u \rightarrow (N = 1 \wedge s = t[u]_p) \vee (\exists M. N = M + 1 \wedge s = t[t^M . u]_p)$$

De manière assez informelle : cette règle cherche les séquences de la forme  $s = t[\diamond]_p^N . u$  et les remplace par les cas particuliers  $N = 1$  et  $N > 1$ .

Les règles ne seront pas toutes décrites ici (pour plus d'informations à leur sujet, se référer à l'article d'Hubert Comon) [1].

Pour vous donner une idée du genre de règle que nous avons eu à mettre en place, voici une de celles que nous avons considérées comme complexes.

##### Decompose 2

$$\begin{aligned} & s[(v_1[w_1[\diamond]_{p'}]^{N_1} . u_1]_p = (v_2[w_2[\diamond]_{q'}]^{N_2} . u_2) \rightarrow \\ & s[a]_p = w_1[a]_p \wedge s[a]_p = v_2[a]_p \wedge v_1[a]_{q'} = w_2[a]_{q'} \\ & \wedge ((N_1 = N_2 \wedge s[u_1]_p = u_2) \\ & \vee (\exists M_1. N_1 = N_2 + M_1 \wedge s[(v_1[w_1[\diamond]_{p'}]^{M_1} . u_1]_p = u_2) \\ & \vee (\exists M_2. N_2 = N_1 + M_2 \wedge w_1[u_1]_p = (v_2[w_2[\diamond]_{q'}]^{M_2} . u_2) \\ & ) \end{aligned}$$

#### 3.2 Validation du fonctionnement

Le processus de validation s'est effectué en trois temps.

Premièrement, après l'implémentation de chacune des règles, nous avons effectué un test sur les exemples donnés par l'auteur de l'article. Par la suite, nous avons initié une base de tests pour automatiser les vérifications et prévenir les régressions. Celle-ci a repris les tests de base



de l'étape précédente et a été enrichie par de nouvelles entrées tout au long du projet. Enfin, notre tuteur Nicolas Peltier a mis en relief certains “bugs”, dûs majoritairement à de mauvaises interprétations que nous avons eues du sujet. Nous les avons corrigés et ajoutés leurs tests à notre base.

De manière pratique, la base de tests se lance par le script shell “test.sh” :

```
snippy:unfication snippy$ bash test.sh
Lancement des tests
test/invalid/double.uni [OK]
test/invalid/simple.uni [OK]
test/invalid/simple2.uni [OK]
test/invalid/syntax.uni [OK]
test/invalid/syntax2.uni [OK]
test/invalid/syntax3.uni [OK]
test/invalid/syntax4.uni [OK]
test/valid/failed/simple1.uni [OK]
test/valid/failed/simple2.uni [OK]
test/valid/succeeds/all.uni [OK]
test/valid/succeeds/bigone.uni [OK]
test/valid/succeeds/decompose2.uni [OK]
test/valid/succeeds/empty.uni [OK]
test/valid/succeeds/simple2.uni [OK]
test/valid/succeeds/simple3.uni [OK]
test/valid/succeeds/simple4.uni [OK]
test/valid/succeeds/simple5.uni [OK]
test/valid/succeeds/simple6.uni [OK]
test/valid/succeeds/unfold2.uni [OK]
test/valid/succeeds/unfold3.uni [OK]
Nombre de tests ayant réussi: 20
Nombre de tests ayant echoue: 0
```

## 4 Résolution des contraintes numériques

### 4.1 Présentation du problème

On se retrouve en sortie de notre unification avec un ensemble de solutions générées qui sont chacune un système d'équation linéaires sur les variables numériques, avec parfois de nouvelles variables introduites. Nous avons utilisé pour ces variables la notation de prolog :  $\_N_i$ . Il ne faut pas renvoyer que l'unification est possible et donner les ensembles de solutions si ceux-ci sont vides. C'est pourquoi nous avons besoin de vérifier que les contraintes d'une solution donnée (un système d'équations linéaires) sont bien réalisables.

### 4.2 Développement de notre propre algorithme de résolution

Nous avons commencé à développer un algorithme qui permettrait de résoudre les systèmes d'équations à contraintes linéaires. La motivation venait principalement de 3 idées :

- Les algorithmes existants avaient un coût en temps et surtout en espace très important car il résolvait ces contraintes dans l'ensemble de l'arithmétique de Presburger. Nous n'avons pas besoin du quantificateur universel  $\forall$ , donc le problème que nous avons à traiter est bien plus simple : pourquoi écraser une mouche avec un marteau ? La borne inférieure de complexité de la résolution d'un problème de décision dans l'arithmétique de Presburger est de  $2^{2^{O(n)}}$  (*Berman 80*[2]) et notre problème en est clairement un sous-problème.
- Les algorithmes existants implémentés ne nous convenaient pas. Celui implémenté dans *Coq* [3] ne traite que le quantificateur universel  $\forall$ , ce qui est incompatible avec notre résolution de contraintes ("existe-t-il une solution?"). Une *librarie prolog* [4] que nous avons trouvée nous obligeait, pour l'utiliser, à ajouter la librairie dynamique (shared object) de Prolog à notre projet et le problème des performances mentionné plus haut restait entier. Nous l'avons utilisée et intégrée avec succès mais retiré dans la version finale du programme.
- L'affichage des solutions : en effet, on peut souvent réduire le nombre d'équation grâce à un "bon paramétrage" de certaines variables numériques en fonction des autres. Nous avons remarqué qu'un bon outil pour chercher se paramétrage était une structure de graphe reliant les variables en relation.

#### Ébauche de notre algorithme :

Vocabulaire : Noeud = Sommet

On s'intéresse uniquement aux conditions en sortie de l'algorithme d'unification sur les Variables Numériques dans N, ie :

- $N = a.M + b$  avec  $(a, b) \in \mathbb{Z}^2$
- $P = N + M$

(sachant que l'on peut faire beaucoup plus, récursivement avec ces 2 formes)

But : Dire si un ensemble de conditions sur les Variables Numériques est satisfaisable, puis, si oui : trouver le "meilleur moyen de paramétrer" les VN en tenant compte du nombre de "degrés de libertés".

Initialisation : On crée un graphe où chaque Variable Numérique N est un sommet portant le couple de valeur  $[\min; \max]$  encadrant les valeurs que N peut prendre. On relie ces sommets comme suit :

- $N = a.M + b$ 
  - Si  $a < 0$  et  $b < 0$  : échec.
  - Si  $a < 0$  et  $b \geq 0$  : un arc de M vers N portant les valeurs  $[a; b]$  avec  $N \leftarrow [1; b]$  et  $M \leftarrow [1; -(b-1)/a]$
  - Si  $a > 0$  et  $b < 0$  : un arc de M vers N portant les valeurs  $[a; b]$  avec  $N \leftarrow [1; +\infty]$  et  $M \leftarrow [1; +\infty]$
  - Si  $a > 0$  et  $b \geq 0$  : un arc de M vers N portant les valeurs  $[a; b]$  avec  $N \leftarrow [1; +\infty]$  et  $M \leftarrow [1; +\infty]$
  - Si  $a = 0$  : si  $b \geq 0 \Rightarrow N = b$ , sinon échec.
- $P = N + M$ , 3 arcs *cond* un peu spéciaux : on crée un noeud  $\circ$  et on fait :
  - un arc de N vers  $\circ$  portant  $[1; 0]$
  - un arc de M vers  $\circ$  portant  $[1; 0]$
  - un arc de  $\circ$  vers P portant  $[1; 0]$

Nota : cela peut être vu comme un “arc à 3 extrémités”.

Quand on a des valeurs non entières dans  $[\min; \max]$  on prend  $\min = E(\min)+1$  et  $\max = E(\max)$ .

Itération :

Chercher puis traiter les cycles

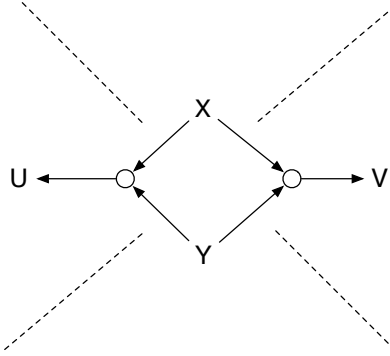
1. On recherche les cycles et dans le cas d'un cycle
2. On écrit l'équation qu'il faut respecter pour que le cycle soit réalisable
3. On la résoud, 2 cas :
  - si elle a des solutions : on les considère (mise à jour des  $[\min; \max]$  des sommets du cycle), on élimine le cycle en le réduisant aux seuls points qui ont un contact avec le reste du graphe.
  - si elle n'a pas de solution : échec.

Propapagation des contraintes

On propage les contraintes en regardant pour chaque noeud les contraintes qui lui arrivent par les arcs. On n'a plus de cycles (ils ont été traités ci-dessus), on ajuste donc maintenant les  $[\min; \max]$  de chaque noeud pour respecter les noeuds qui y arrivent.

Par un parcours particulier de l'arbre et un système de marquage spécifique pour les noeuds “équation”, on peut ensuite déterminer quelles variables choisir pour articuler l'ensemble de solution (et n'en oublier aucune). On en déduit ainsi un sous-ensemble de variable avec pour chacune ses contraintes propres qui définiront toutes les autres variables du système d'équation. On obtient de plus une idée de la dimension de la solution (nombre de variables dans ce sous ensemble).

Exemple bloquant :



qui correspond à

$$\begin{cases} U = X + Y \\ V = a.X + b.Y \end{cases}$$

sans aucune relation entre  $X, Y, U, V$  deux à deux (symbolisé par - - -)

Explication : Si il y a une relation entre 2 membres de ce type de cycle mais qu'elle n'est pas encore une fois de ce type là, on peut commencer par éliminer ce cycle de type "traitable" et cela changera la forme de ce cycle. Si on se retrouve avec une forme qui correspond vraiment à ce schéma (donc sans relations 2 à 2) avec une alternance de noeuds normaux et  $\circ$ , nous n'avons pas trouvé de moyen de réduire ce cycle. Au mieux on retourne le problème.

Nous avons quand même produit un peu de code pour cet algorithme : ce travail de recherche n'a pas été totalement inutile car cette transformation du système d'équation en graphe peut servir à chercher le meilleur affichage (dans le sens *paramétrage*) possible pour les solutions, quand elles existent.

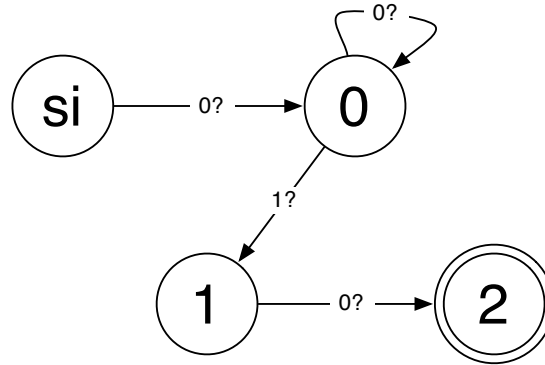
### 4.3 Automates pour l'arithmétique de Presburger

Nous avons donc arrêté nos recherches sur cet algorithme et nous avons décidé (en vu des contraintes présentées en introduction de cette partie) d'implémenter notre propre résolution de contraintes dans l'arithmétique de Presburger via ce qui existait algorithmiquement : les automates pour la résolution de contraintes linéaires. Nous nous sommes documentés ([5], [6], [7]) sur le sujet et nous sommes principalement basé sur un article de Pierre Wolper et Bernard Boigelot [8] décrivant la construction de tels automates.

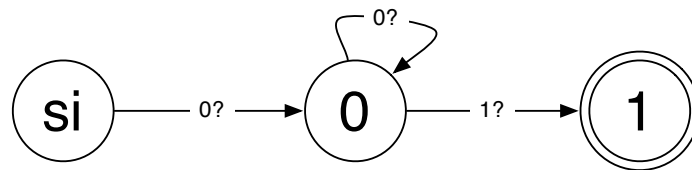
Nous avons des disjonctions de solutions qui sont des conjonctions d'équations, nous les traitons donc séparément (une peut être réalisable et pas une autre). Chaque équation numérique est mise sous la forme d'un automate qui permet très facilement de dire si cette équation est réalisable. Ensuite, on parcourt tous les automates "en parallèle" pour tester si les états finaux sont accessibles avec un ensemble de transitions commun. Si c'est le cas, la solution est réalisable. Sinon, on l'élimine de l'ensemble des solutions. Si cet ensemble se trouve vide à la fin de la résolution des contraintes, l'unification échoue. Nous ne détaillerons pas ici les détails de la construction des automates, ce travail est très bien effectué dans *On the Construction of Automata from Linear Arithmetic Constraints* [8] et documenté dans Doxygen. L'introduction au fonctionnement de notre implémentation de ces automates se fera par un exemple. Soit la solution (simplissime) :

$$\begin{cases} N = 2 \\ M = 1 \end{cases}$$

Elle entraine la construction de ces automates :



Automate correspondant à  $N = 2$

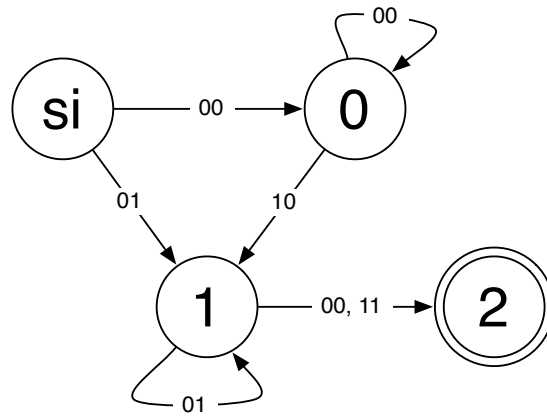


Automate correspondant à  $M = 1$

Les points d'interrogation remplacent un bit dont ne se soucie pas de la valeur. En effet, la construction des tableaux de bits de transition de taille égale au nombre de variables numériques permet une fusion (parcours en parallèle) beaucoup plus facile. Ici, leur parcours en parallèle (en utilisant les même transitions) permet d'arriver aux états finaux. Cette solution est donc réalisable.

En rajoutant la contrainte  $N = M + 2$ , on obtient ce système-solution et un nouvel automate.

$$\begin{cases} N = 2 \\ M = 1 \\ N = M + 2 \end{cases}$$



Automate correspondant à  $M = 1$

Le parcours en parallèle de ces 3 automates ne permet pas d'arriver à un état final pour tous à un moment commun. Il n'y a donc pas de solution à ce système-solution. Si c'est le seul résultat d'une unification, elle échoue.

## 5 Conclusions

L'implémentation de cette unification et des automates de résolution à donc été réalisée, testée et validée. Le résultat de notre travail se présente finalement sous la forme d'un exécutable et de son man, de ce rapport, du code source documenté et laissé disponible pour son évolution ou son utilisation en tant que librairie.

Le fait d'avoir pu faire un peu de travail de recherche nous à permis de ne pas nous enfermer uniquement dans l'implémentation et le suivi de travaux externes ; ce qui à grandi notre intérêt pour ce sujet.

L'ensemble du projet a donc été très agréable à mener, de par sa nature mais aussi grâce au cadre de travail.

Nous avons donc tiré de ce projet une expérience très enrichissante.

PS : Nous continuerons à maintenir le code fonctionnel.

## Références

- [1] H. Comon, On Unification of Terms with Integer Exponents, *Mathematical System Theory* **28**, 67 (1995).
- [2] L. Berman, The complexity of logical theories, *j-THEOR-COMP-SCI* **11**, 71 (1980).
- [3] <http://coq.inria.fr>, The coq proof assistant.
- [4] <http://stud4.tuwien.ac.at/e0225855/presprover/presprover.html>, Presprover - prove formulas of presburger arithmetic.
- [5] D. Cooper, Theorem proving in arithmetic without multiplication, in *Machine Intelligence* 7, edited by B. Meltzer and D. Michie, chap. 5, pp. 91–99, Edinburgh University Press, 1972.
- [6] P. Wolper and B. Boigelot, An automata-theoretic approach to presburger arithmetic constraints (extended abstract), in *Static Analysis Symposium*, Lecture Notes in Computer Science, pp. 21–32, Springer-Verlag, 1995.
- [7] F. Klaedtke, *ACM Trans. Comput. Logic* **9**, 1 (2008).
- [8] P. Wolper and B. Boigelot, On the construction of automata from linear arithmetic constraints, in *TACAS '00 : Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pp. 1–19, London, UK, 2000, Springer-Verlag.

## 6 Annexes

### 6.1 Manuel Utilisateur - man



**NAME**

unificator - unificator with Integer Exponents Terms

**SYNOPSIS**

unificator [OPTIONS] [FILE]

**DESCRIPTION**

A unificator which deal with integer exponents terms. If [FILE] is not given, the standard input is used to get data.

**OPTIONS**

- help, -h  
Print a usage help message.
- step, -s  
Pause on every step of the unification.
- nopbv Do not apply Presburger automata verification.
- beep, -b  
Emit a beep on every step of the unification and at the end
- anniversary  
Emit a special beep on every step of the unification and at the end.
- info, -i Print unification applied rules.
- latex, -l  
Print the steps and the final result in latex syntax.
- output, -o FILE  
Redirect standard output in a file.

**BUGS**

Reporting Bugs  
Email bug reports to <achoum@gmail.com>.

**SEE ALSO**

unificator -help

## 6.2 Diagramme de classes



### 6.3 Exemple de sortie générée avec l'option -latex

Initial set of equation

$$(a = X \vee b = f(X))$$

$$(a = X \vee b = f(X))$$

$$R1 : s = x \rightarrow x = s$$

$$(X = a \vee b = f(X))$$

$$R1 : f(x)=g(y) \rightarrow \perp$$

$$X = a$$

End of computation

Number of steps : 3

Computation time : 0 seconde(s)

$$X = a$$

Unification Succeeds

Number of main solution : 1

solution 1

$$X = a$$

## 6.4 Code source

## 6.5 Fichiers d'entête

### 6.5.1 automata.h

```
/*
 * File:    automata.h
 * Author:  snippy
 *
 * Created on June 12, 2008, 10:37 AM
 */

#ifndef _AUTOMATA_H
#define _AUTOMATA_H

#include <ostream>
#include <map>

class Transition;
class Automata;

class Groupe
{
public:
    list<Automata*> autos;
};

class State
{
public:
    Automata *automata;
    bool si;
    list<Transition*> step;
    int value;
    State(int v, bool s, Automata *a);
    void addTransition(Transition* t);
    ~State();
    void display(ostream &o);
};

class Transition
{
public:
    int* bits; // 0 ou 1 ou -1 (pour '?')
    State* st; // original state
    Transition(State* next, int* b);
    void display(int bits_size);
    ~Transition();
};

class StatePara
{
public:
    list<State*> actives;
};

class ExploPara
{
public:
    list<StatePara> statePara;
};

class Automata
{
public:
    State* si;
    int cvalue;
    int nvars;
    vector<int> a;
    int eqSize;
    bool isIntersect(Automata *au, int size);
    Automata(Condition* n, std::map<int, int>* numvars);
    Automata(Condition* n);
    ~Automata();
    void display(ostream &o);
};

#endif /* _AUTOMATA_H */
```

### 6.5.2 config.h

```
#ifndef CONFIG_H
#define CONFIG_H

#include <istream>
#include <ostream>

using namespace std;

class Configuration
{
public:
    enum Write_Mode
    {
        NORMAL ,
        LATEX ,
        XML ,
        HTML
    };

    Write_Mode writeMode;
    istream *inputdatas;
    bool pauseEveryStep;
    bool printRule;
    bool NcanbeZero;
    bool fileInput;
    bool presBurgerVerif;
    bool disjonction;
    int beep;
    static void printOptionInfo();
    static void printConsol();
    Configuration(int argc, char *argv[]);
    ~Configuration();
};

extern ostream *mycout;
extern Configuration *globconfig;

#endif
```

### 6.5.3 display.h

```
#ifndef DISPLAY_H
#define DISPLAY_H

#include "unification.h"

namespace Display
{
    void displayResult(Unification *unification);
    void displayTime(long atime);
    void Head(Configuration *config);
    void Foot(Configuration *config);
    void texEndl(std::ostream& s);
    void texHspace(std::ostream& s);
    void texDi(std::ostream& s);
    void beep(int freq,int time);
};

std::ostream& operator << (std::ostream& o, const list<unsigned int> &path);

#endif
```

### 6.5.4 exception.h

```
#ifndef EXCEPTION_H
#define EXCEPTION_H

#include <exception>
#include <string>
#include <assert.h>
using namespace std;

class OptionException: public exception
{
}
```

```

    string text;
public:
    const char* what() const throw();
    OptionException(string t);
    ~OptionException() throw();
};

class SytaxException: public exception
{
    string text;
public:
    const char* what() const throw();
    SytaxException(string t);
    ~SytaxException() throw();
};

#endif

```

### 6.5.5 main.h

### 6.5.6 synnaeve.guillame.bert.h

```

#ifndef SOLVER_H
#define SOLVER_H

#include <list>
#include <vector>
#include "unification.h"
#include <ostream>
using namespace std;

namespace Solver
{
class Vertex;
class Edge;

    class VariableSolution
    {
    public:
        bool seeoutside;
        Node *definition;
    };

    class Edge
    {
    public:
        Vertex *n1;
        Vertex *n2;
        int a;
        int b;
    };

    class Vertex
    {
    public:
        list<Edge*> in;
        list<Edge*> out;
        int variable;
        int count;
        int min;
        int max;
        bool setBounds(int min,int max);
        bool setBoundMin(int min);
        bool setBoundMax(int max);

        Edge *father;
        bool visited;
    };

    class Contribution
    {
    public:
        Vertex *v;
        float a;
    };

    class SolverSolution : public Solution
    {

```

```

    public:
        Unification *unification;
        VariableSolution *variableSolution;
        unsigned int ndvar;
        bool success;
        void display(ostream &o);
        bool check();
    };

    class Graph
    {
    public:
        SolverSolution *solus;
        Unification *unification;
        bool success;
        list<Vertex*> vertexs;
        list<Edge*> edges;
        Graph(Node* node, Unification *unification);
        bool explo(Node* n);
        SolverSolution *run();
        ~Graph();
        bool seekCycle(list<Vertex*> &ver, list<Edge*> &ed);
        void resetVisited();
        void print();

        Vertex* addVertex(int id);
        Edge* addEdge(Vertex *v1, Vertex *v2, int a, int b);
    };

};

#endif

```

## 6.5.7 unification.h

```

#ifndef UNIFICATION_H
#define UNIFICATION_H

#include "parser.h"
#include "config.h"
#include <ostream>

class Unification;

class Solution
{
public:
    virtual void display(ostream &o) = 0;
    virtual bool check() = 0;
};

class VariableSolution
{
public:
    VariableSolution();
    bool seeoutside;
    Node *definition;
    list<Condition*> definitionnc;
};

class SimpleSolution : public Solution
{
public:
    Unification *unification;
    bool fail;
    unsigned int ndvar;
    unsigned int ndnumvar;
    VariableSolution *variableSolution;
    VariableSolution *numericalvariableSolution;
    SimpleSolution(unsigned int ndvar, unsigned int ndnumvar, Unification *unification);
    ~SimpleSolution();
    void display(ostream &o);
    bool check();
};

class Unification
{
public:
    Configuration *configuration;
    Parser *parser;
    bool result;

    list<Solution*> solutions;

```

```

Unification(Configuration *configuration, Parser *parser);
void run();
Node* ApplyR0(Node *start=NULL);
bool tryApplyR1();
bool tryApplyR2();
bool tryApplyR3();
bool tryApplyR4();
bool tryApplyR5();
Node* SetDisjunctiveSystem(Node *start = NULL);
bool checkDefinition();
void debugDisplay(ostream &s);
bool exploDefinition(Node *n, SimpleSolution *solution);
bool verifValidPresburger(Node *n);
~Unification();

static int gcd(unsigned int a, unsigned int b);
static bool prefix(const list<unsigned int> &l1, const list<unsigned int> &l2);
static list<unsigned int> append(const list<unsigned int> &l1, const list<unsigned int> &l2);

};

#endif

```

## 6.6 Fichiers de code

### 6.6.1 automata.h

```

#include "unification.h"
#include "exception.h"
#include "display.h"
#include "parser.h"
#include "config.h"
#include "automata.h"
#include <math.h>
#include <typeinfo>
// #include <ostream>

bool Automata::isIntersect(Automata *au, int size)
{
    int *mask1 = this->si->step.front()->bits;
    int *mask2 = au->si->step.front()->bits;
    int i;
    for(i=0; i<size; i++)
        if((mask1[i]!=-1)&&(mask2[i]!=-1))
            return true;
    return false;
}

State::State(int v, bool s, Automata *a)
{
    automata = a;
    value = v;
    si = s;
}

void State::addTransition(Transition* t)
{
    step.push_back(t);
}

void State::display(ostream &o)
{
    if (si)
        o << "INITIAL STATE" << endl;
    else
        o << "State: " << value << endl;
}

Transition::Transition(State* next, int* b)
{
    st = next;
    bits = b;
}

void Transition::display(int bits_size)
{
    (*mycout) << "BITS: ";
    for (int j=0; j < bits_size; j++) {
        if (bits[j] == -1) {
            (*mycout) << "?";
        } else {
            (*mycout) << bits[j];
        }
    }
}

```



```

    }
    (*mycout) << endl;
}

/**
 * A constructor that put the equation under the form :
 * N = cvalue (from N = a)
 * OR N - aM = cvalue (from N = aM + b)
 * OR N - M - P = 0 = cvalue (from N = M + P)
 * OR N - M = 0 = cvalue (from N = M)
 * and build the associated Automata
 */
Automata::Automata(Condition* n, std::map<int, int>* numvars) {
    // CondA, CondANplusB, CondNplusN, CondN
    if(typeid(*n)==typeid(CondA)) {
#ifdef DEBUG
        (*mycout) << "CondA" << "de_type:" << typeid(*n).name() << endl;
#endif
        cvalue = ((CondA*)n)->a;
        a.push_back(1);
        nvars = 1;
    } else if(typeid(*n)==typeid(CondANplusB)) {
#ifdef DEBUG
        (*mycout) << "CondANplusB" << "de_type:" << typeid(*n).name() << endl;
#endif
        cvalue = ((CondANplusB*)n)->b;
        nvars = 2;
        a.push_back(1);
        a.push_back(- ((CondANplusB*)n)->a );
    } else if(typeid(*n)==typeid(CondNplusN)) {
#ifdef DEBUG
        (*mycout) << "CondNplusN" << "de_type:" << typeid(*n).name() << endl;
#endif
        cvalue = 0;
        nvars = 3;
        a.push_back(1);
        a.push_back(-1);
        a.push_back(-1);
    } else { // CondN
#ifdef DEBUG
        (*mycout) << "CondN" << "de_type:" << typeid(*n).name() << endl;
#endif
        cvalue = 0;
        nvars = 2;
        a.push_back(1);
        a.push_back(-1);
    }
}

/** We're now in the middle of the constructor :
 * in a[] the coefficients of the NumVars,
 * in cvalue the result (right part of the equation),
 * in numvars[] the map between a NumericalVar* and its integer indice,
 * in numvars->size() the size of the bits[] table of the Transitions.
 */

si = new State(INT_MIN, true, this);
State* final_st = new State(cvalue, false, this);
vector<State*> table_of_st;
table_of_st.push_back(final_st);
list<State*> active_st;
active_st.push_back(final_st);
int bits_size = numvars->size();

while(!active_st.empty()) {
    State* tmpst = active_st.front();
    active_st.pop_front();
#ifdef DEBUG
    (*mycout) << "I_just_popped:";
    tmpst->display(*mycout);
#endif
    for (int i = 0 ; i < (pow(2.f, nvars)) ; i++){
        int decal = i;
        int* bits;
        bits = new int[bits_size];
        int* prebits = new int[nvars];
        for (int j=0 ; j < nvars ; j++){
            prebits[j] = (0x0001 & decal);
            // (*mycout) << "PREBITS : " << prebits[j] << " DE " << j << endl;
            decal = decal >> 1;
            // (*mycout) << "***PREBITS : " << prebits[j] << " DE " << j << endl;
        }

        for (int k=0 ; k < bits_size ; k++) bits[k] = -1;

        if(typeid(*n)==typeid(CondA)) {
            bits[((*numvars)[n->n1->id])] = prebits[0];
        } else if(typeid(*n)==typeid(CondANplusB)) {

```

```

        bits[(((numvars)[n->n1->id]))] = prebits[0];
        bits[(((numvars)[((CondANplusB*)n->n2->id]))] = prebits[1];
    } else if (typeid(*n) == typeid(CondNplusN)) {
        bits[(((numvars)[n->n1->id]))] = prebits[0];
        bits[(((numvars)[((CondNplusN*)n->n2->id]))] = prebits[1];
        bits[(((numvars)[((CondNplusN*)n->n3->id]))] = prebits[2];
    } else { // CondN
        bits[(((numvars)[n->n1->id]))] = prebits[0];
        bits[(((numvars)[((CondN*)n->n2->id]))] = prebits[1];
    }
}

#ifdef DEBUG
/*
(*mycout) << " BITS : ";
for (int j=0 ; j < bits_size ; j++) {
    if (bits[j] == -1 ) {
        (*mycout) << "?";
    } else {
        (*mycout) << bits[j];
    }
}
(*mycout) << endl;
(*mycout) << " PREBITS : ";
for (int j=0 ; j < nvars ; j++) (*mycout) << prebits[j];
(*mycout) << endl;
*/
#endif

// calculus of -a*bits
int ab;
if (typeid(*n) == typeid(CondA)) {
    ab = - (a[0]*bits[(((numvars)[n->n1->id]))];
} else if (typeid(*n) == typeid(CondANplusB)) {
    ab = - (a[0]*bits[(((numvars)[n->n1->id]))] +
            a[1]*bits[(((numvars)[((CondANplusB*)n->n2->id]))]);
} else if (typeid(*n) == typeid(CondNplusN)) {
    ab = - (a[0]*bits[(((numvars)[n->n1->id]))] +
            a[1]*bits[(((numvars)[((CondNplusN*)n->n2->id]))] +
            a[2]*bits[(((numvars)[((CondNplusN*)n->n3->id]))]);
} else { // CondN
    ab = - (a[0]*bits[(((numvars)[n->n1->id]))] +
            a[1]*bits[(((numvars)[((CondN*)n->n2->id]))]);
}
#ifdef DEBUG
/// (*mycout) << "Value of a*b : " << ab << endl;
/// (*mycout) << "Value of the current state : " << tmpst->value << endl;
#endif

if (ab == tmpst->value) {
    Transition* tr = new Transition(tmpst, bits);
    si->addTransition(tr);
#ifdef DEBUG
    (*mycout) << "Link to initial" << endl;
    (*mycout) << "si->" << tmpst->value << endl;
    tr->display(bits_size);
#endif
} // else {
// calculus of gamma 0 (g0)
double g0;
if (typeid(*n) == typeid(CondA)) {
    g0 = ((double)(tmpst->value - (a[0]*bits[(((numvars)[n->n1->id]))])) / 2;
} else if (typeid(*n) == typeid(CondANplusB)) {
    g0 = ((double)(tmpst->value - (a[0]*bits[(((numvars)[n->n1->id]))] +
            a[1]*bits[(((numvars)[((CondANplusB*)n->n2->id]))])) / 2;
} else if (typeid(*n) == typeid(CondNplusN)) {
    g0 = ((double)(tmpst->value - (a[0]*bits[(((numvars)[n->n1->id]))] +
            a[1]*bits[(((numvars)[((CondNplusN*)n->n2->id]))] +
            a[2]*bits[(((numvars)[((CondNplusN*)n->n3->id]))])) / 2;
} else { // CondN
    g0 = ((double)(tmpst->value - (a[0]*bits[(((numvars)[n->n1->id]))] +
            a[1]*bits[(((numvars)[((CondN*)n->n2->id]))])) / 2;
}
#ifdef DEBUG
/// (*mycout) << "Value of gamma0 : " << g0 << endl;
#endif
#ifdef DEBUG
// if (floor(g0) == floor(g0 + 0.6)) {
double double_g0;
if (!modf(g0, &double_g0)) {
    int int_g0 = int(double_g0);
    bool found = false;
    // search for a state with the value g0
    vector<State*>::iterator my_it = table_of_st.begin(); // MIAM
    while (my_it != table_of_st.end()) {
        if ((*my_it)->value == int_g0) {
            found = true;
            Transition* tr = new Transition(tmpst, bits);
            (*my_it)->addTransition(tr);
#ifdef DEBUG
            (*mycout) << " " << (*my_it)->value << " " << tmpst->value << endl;

```

```

        tr->display(bits_size);
        #endif
    }
    my_it++;
}

// if not found, add it
if (!found) {
    State* newst = new State(int_g0, false, this);
    table_of_st.push_back(newst);
    active_st.push_back(newst);
    Transition* tr = new Transition(tmpst, bits);
    newst->addTransition(tr);
    #ifdef DEBUG
    (*mycout) << " " << newst->value << " -> " << tmpst->value << endl;
    tr->display(bits_size);
    #endif
}
}
//}
}
}

Automata::Automata(Condition* n) {
}

Automata::~Automata() {
}

void Automata::display(ostream &o) {
    o << "Automata of value: " << cvalue << " and with a: ";
    si->display(*mycout);
}

```

## 6.6.2 config.h

```

#include "config.h"
#include "exception.h"

#include <string.h>
#include <fstream>
#include <iostream>

#include <typeinfo>

ostream *mycout;
Configuration *globconfig;

/**
 * Print options help message
 */
void Configuration::printOptionInfo()
{
    (*mycout) << endl;
    (*mycout) << "UnificatorOptions" << endl;
    (*mycout) << endl;
    (*mycout) << "unificator[OPTIONS][FILE]" << endl;
    (*mycout) << endl;
    (*mycout) << "Options:" << endl;
    (*mycout) << "-help, -h: print this message" << endl;
    (*mycout) << "-step, -s: pause on every step of the unification" << endl;
    (*mycout) << "-beep, -b: emit a beep on every step of the unification and at the end (need \"
        beep\" for linux)" << endl;
    (*mycout) << "-nopbv: do not apply Presburger automata verification" << endl;
    (*mycout) << "-anniversary: emit a special beep on every step of the unification and at the end"
        << endl;
    (*mycout) << "-info, -i: print unification applied rules" << endl;
    //(*mycout) << " -zero, -z : allow numerical variable to be nil" << endl;
    (*mycout) << "-latex, -l: print the steps and the final result in latex" << endl;
    (*mycout) << "-output, -o: redirect standard output in a file" << endl;
    (*mycout) << endl;
    (*mycout) << "If [FILE] is not given, the standard input is used to get data." << endl;
    printConsol();
}

/**
 * Print command help message
 */
void Configuration::printConsol()
{
    (*mycout) << endl;
    (*mycout) << "UnificatorSyntax" << endl;
}

```

```

(*mycout) << endl;
(*mycout) << "Commands:" << endl;
(*mycout) << "\[TERM] : new term (see documentation for more details)" << endl;
(*mycout) << "\help : print this message" << endl;
(*mycout) << "\set [OPTION] : activate the option" << endl;
(*mycout) << "\set [OPTION] : activate the option" << endl;
(*mycout) << endl;
(*mycout) << "Options:" << endl;
(*mycout) << "\step : pause on every step of the unification" << endl;
(*mycout) << "\beep : emit a beep on every step of the unification" << endl;
(*mycout) << "\info : print unification applied rules" << endl;
//(*mycout) << " zero : allow numerical variable to be nil" << endl;
(*mycout) << endl;
}

/**
 * Parse configuration from options
 */
Configuration::Configuration(int argc, char *argv[])
{
    inputdatas = NULL;
    mycout = NULL;
    printRule = false;
    pauseEveryStep = false;
    NcanbeZero = false;
    fileInput = false;
    presBurgerVerif = true;
    disjonction = true;
    writeMode = Configuration::NORMAL;
    beep = 0;
    bool askhelp = false;

    for(int i=1; i<argc; i++)
    {
        if(argv[i][0] == '-')
        {
            if((!strcmp(argv[i], "-step")) || (!strcmp(argv[i], "-s")))
                pauseEveryStep = true;
            else if((!strcmp(argv[i], "-info")) || (!strcmp(argv[i], "-i")))
                printRule = true;
            /*else if((!strcmp(argv[i], "-zero")) || (!strcmp(argv[i], "-z")))
                NcanbeZero = true;*/
            else if((!strcmp(argv[i], "-beep")) || (!strcmp(argv[i], "-b")))
                beep = 1;
            else if((!strcmp(argv[i], "-invbeep")) || (!strcmp(argv[i], "-ib")))
                beep = 2;
            else if((!strcmp(argv[i], "-anniversary")) || (!strcmp(argv[i], "-ann")))
                beep = 3;
            else if((!strcmp(argv[i], "-latex")) || (!strcmp(argv[i], "-l")))
                writeMode = Configuration::LATEX;
            else if(!strcmp(argv[i], "-nopbv"))
                presBurgerVerif = false;
            else if(!strcmp(argv[i], "-nodisj"))
                disjonction = false;
            else if((!strcmp(argv[i], "-output")) || (!strcmp(argv[i], "-o")))
            {
                if(!mycout)
                {
                    i++;
                    if(i>=argc)
                        throw OptionException("no output file given");
                    mycout = new fstream(argv[i], ios::out);
                    if(mycout->fail())
                        throw OptionException("can't open the output file");
                }
                else
                    throw OptionException("several ouput files");
            }
            else if((!strcmp(argv[i], "-help")) || (!strcmp(argv[i], "-h")))
                askhelp = true;
            else
            {
                throw OptionException(string("unknown option \"") + argv[i] + "\"");
            }
        }
        else
        {
            if(!inputdatas)
            {
                fileInput = true;
                inputdatas = new fstream(argv[i], ios::in);
                if(inputdatas->fail())
                    throw OptionException("can't open the input file");
            }
            else
                throw OptionException("several input files");
        }
    }
}

```

```

    if(!mycout)
        mycout = &cout;
    if(!inputdatas)
        inputdatas = &cin;

    if(askhelp)
    {
        printOptionInfo();
        exit(0);
    }
}

Configuration::~Configuration()
{
}

```

### 6.6.3 display.h

```

#include "display.h"
#include "exception.h"
#include <iostream>

#ifdef WIN32
#include <windows.h>
#else

#endif

namespace Display
{
/**
 * Print a new line in latex syntax
 * @param s output stream
 */
void texEndl(std::ostream& s)
{
    if(globconfig->writeMode==Configuration::LATEX)
        s << "\\\\" << endl;
}

/**
 * Print a # in latex syntax
 * @param s output stream
 */
void texDi(std::ostream& s)
{
    if(globconfig->writeMode==Configuration::NORMAL)
        s << "#";
}

/**
 * Print a 20 pt space in latex syntax
 * @param s output stream
 */
void texHspace(std::ostream& s)
{
    if(globconfig->writeMode==Configuration::LATEX)
        s << "\\hspace{20pt}";
}

/**
 * Print the head of the latex
 */
void Head(Configuration *config)
{
    if(config->writeMode==Configuration::LATEX)
    {
        (*mycout) << "\\documentclass[a4paper,10pt]{article}" << endl;
        (*mycout) << "\\newcommand{\\noi}{\\noindent}" << endl;
        (*mycout) << "\\newcommand{\\II}{\\mbox{\\large I\\hspace{-0,353em}1}}" << endl;
        (*mycout) << "\\newcommand{\\MR}{\\mathcal{R}}" << endl;
        (*mycout) << "\\newcommand{\\RR}{\\mbox{I\\hspace{-0.55em}R}}" << endl;
        (*mycout) << "\\usepackage[utf8]{inputenc}" << endl;
        (*mycout) << "\\usepackage{amsmath,amssymb}" << endl;
        (*mycout) << "\\usepackage{fullpage}" << endl;
        (*mycout) << "\\usepackage{graphicx}" << endl;
        (*mycout) << "\\begin{document}" << endl;
    }
}
}

/**

```

```

* Print the foot of the latex
*/
void Foot(Configuration *config)
{
    if(config->writeMode==Configuration::LATEX)
    {
        (*mycout) << "\\end{document}" << endl;
    }
}

/**
* Print the result of the unification (and play sound if needed)
*/
void displayResult(Unification *unification)
{
    if(unification->result)
    {
        Display::texEndl(*mycout);
        (*mycout) << endl;
        Display::texDi(*mycout);
        (*mycout) << "Unification_␣Succeeds";
        Display::texDi(*mycout);
        Display::texEndl(*mycout);
        (*mycout) << endl << endl;

        if(unification->configuration->disjonction)
        {
            (*mycout) << "Number_␣of_␣main_␣solution_␣:" << unification->solutions.size();
            Display::texEndl(*mycout);
            (*mycout) << endl << endl;
            list<Solution*>::iterator it = unification->solutions.begin();
            int num = 0;
            while(it!=unification->solutions.end())
            {
                num++;
                (*mycout) << "solution_␣" << num ;
                Display::texEndl(*mycout);
                (*mycout) << endl;
                (*it)->display(*mycout);
                if(unification->configuration->writeMode==Configuration::LATEX)
                {
                    (*mycout) << "\\null";
                    Display::texEndl(*mycout);
                    (*mycout) << endl;
                    it++;
                }
            }

            if(unification->configuration->beep!=0)
            {
                if(unification->configuration->beep==3)
                {
                    beep(1046,250);beep(1046,250);
                    beep(1175,500);beep(1046,500);
                    beep(1397,500);beep(1318,1000);
                    beep(1046,250);beep(1046,250);
                    beep(1175,500);beep(1046,500);
                    beep(1568,500);beep(1397,1000);
                    beep(1046,250);beep(1046,250);
                    beep(2093,500);beep(1760,500);
                    beep(1397,500);beep(1318,500);
                    beep(1175,1000);beep(1868,250);
                    beep(1868,250);beep(1760,500);
                    beep(1397,500);beep(1568,500);
                    beep(1397,1500);
                }
                else
                {
                    beep(500,100);
                    beep(600,100);
                }
            }
        }
    }
    else
    {
        (*mycout) << endl;
        Display::texDi(*mycout);
        (*mycout) << "Unification_␣Failed";
        Display::texDi(*mycout);
        Display::texEndl(*mycout);
        (*mycout) << endl;
        if(unification->configuration->beep!=0)
        {
            beep(50,100);
            beep(40,100);
            beep(50,100);
        }
    }
}

```

```

    }

/**
 * Print the duration
 */
void displayTime(long atime)
{
    if(atime > 1000*60*60*24)
    {
        (*mycout) << (atime / (1000 * 60*60*24));
        (*mycout) << "␣day(s)␣";
    }

    if(atime > 1000*60*60)
    {
        (*mycout) << (atime / (1000 * 60*60));
        (*mycout) << "␣hour(s)␣";
    }

    if(atime > 1000*60)
    {
        (*mycout) << (atime / (1000 * 60));
        (*mycout) << "␣minute(s)␣";
    }

    (*mycout) << (((float)(atime%(1000*60)))/1000);
    (*mycout) << "␣seconde(s)␣";

    }

    void beep(int freq,int time)
    {
#ifdef WIN32
        Beep(freq,time);
#else
        char tmp[100];
        sprintf(tmp,"beep␣-f␣%i␣-l␣%i",freq,time);
        //sprintf(tmp,"xset b 100 %i %i && echo -e \"\\a\\\" &",freq,time);
        //printf("%s\\n",tmp);
        system(tmp);

        //printf("\\a");
#endif
    }

    }

/**
 * Print a list of uint
 */
std::ostream& operator << (std::ostream& o, const list<unsigned int> &path)
{
    list<unsigned int>::const_iterator it = path.begin();
    while(it!=path.end())
    {
        o << (*it) << "␣";
        it++;
    }
    return o;
}

```

## 6.6.4 exception.h

```

#include "exception.h"
#include <string.h>
#include <stdio.h>
#include <stdarg.h>

/**
 * Print a description of the exception
 */
const char* OptionException::what() const throw()
{
    return text.c_str();
}

OptionException::OptionException(string t)
{
    text = "Option␣Error␣:␣" + t;
}

OptionException::~OptionException() throw()

```

```

    {
    }

/**
 * Print a description of the exception
 */
const char* SytaxException::what() const throw()
{
    return text.c_str();
}

SytaxException::SytaxException(string t)
{
    text = "Syntax_␣Error_␣:␣" + t;
}

SytaxException::~SytaxException() throw()
{
}

```

### 6.6.5 main.h

```

/**
 * Main file of the unificator
 * @author Mathieu Guillame-Bert and Gabriel Synnaeve
 */

#include <stdio.h>
#include <iostream>
#include <fstream>
#include "config.h"
#include "exception.h"
#include "display.h"
#include "parser.h"
#include "unification.h"

int main(int argc, char *argv[])
{
    try
    {
        Configuration *config = globconfig = new Configuration(argc,argv);
        Display::Head(config);
        Parser *parser = new Parser(config);
        Unification *unification = new Unification(config,parser);
        unification->run();
        Display::displayResult(unification);
        Display::Foot(config);
    }
    catch (exception& e)
    {
        cerr << endl << e.what() << endl;
        //getc(stdin);
    }
    return 0;
}

```

### 6.6.6 parser.h

```

#include "FlexLexer.h"
#include "parser.h"
#include "exception.h"
#include <typeinfo>
#include <sstream>

// bon with the parser
extern int yylex();
extern void set_extParser(Parser* extP);
extern void set_result(Node **r);
extern yyFlexLexer* lexer;
extern int numline;
extern int yyparse (void);

int Node::ndterms;

/**
 * Parse input data and check soundness
 */
Parser::Parser(Configuration *config)
{

```



```

this->config = config;
autoNumericalValCur = 0;
autoContstantValCur = 0;

set_extParser(this);
this->result = NULL;
set_result(&this->result);

lexer = new yyFlexLexer(config->inputdatas, mycout);

numLine = 1;
if((yyvsparse()) || (!this->result))
    exit(1);

//=====test 1=====
// f(g(a,y),y) = f(x,b)

/*
this->result = new And();
Function *f1 = new Function("f",this);
Function *f2 = new Function("f",this);
Function *f3 = new Function("f",this);

f2->addArgument(new Function("a",this));
f2->addArgument(new Variable("y",this));

f1->addArgument(f2);
f1->addArgument(new Variable("y",this));

f3->addArgument(new Variable("x",this));
f3->addArgument(new Function("b",this));
((And*)(result))->addBranch(new Equal(f1,f3));
((And*)(result))->addBranch(new Equal(new Variable("z",this),new Variable("z",this)));
*/

//=====test 2=====
// f(@,a)^{N}.x = f(y,f(a,@)^{M}.z)

/*
this->result = new And();

Function *f2 = new Function("f",this);
f2->addArgument(new Hole());
f2->addArgument(new Function("a",this));

Function *f3 = new Function("f",this);
f3->addArgument(new Function("a",this));
f3->addArgument(new Hole());

Function *f1 = new Function("f",this);
f1->addArgument(new Variable("y",this));
f1->addArgument(new Expo(f3,new NumericalVar("M",this),new Variable("z",this)));

((And*)(result))->addBranch(new Equal(new Expo(f2,new NumericalVar("N",this),new Variable("x",this)),f1));
*/

//=====test 3=====
// f(@,a)^{2}.x = f(y,f(a,@)^{2}.z)
/*
this->result = new And();

Function *f2 = new Function("f",this);
f2->addArgument(new Hole());
f2->addArgument(new Function("a",this));

Function *f3 = new Function("f",this);
f3->addArgument(new Function("a",this));
f3->addArgument(new Hole());

Function *f1 = new Function("f",this);
f1->addArgument(new Variable("y",this));
f1->addArgument(new Expo(f3,2,new Variable("z",this)));

((And*)(result))->addBranch(new Equal(new Expo(f2,2,new Variable("x",this)),f1));
*/

//=====test 4=====
// f(@,a)^N.y = x
/*
this->result = new And();

Function *f2 = new Function("f",this);
f2->addArgument(new Hole());
f2->addArgument(new Function("a",this));

((And*)(result))->addBranch(new Equal(new Expo(f2,new NumericalVar("N",this),new Variable("y",this)),new Variable("x",this)));

```

```

*/
//=====test 5=====
// g(f(a,g(@))^{N1}.z) = g(f(x,g(@))^{N2}.y
/*
this->result = new And();

Function *f6 = new Function("g",this);
f6->addArgument(new Hole());

Function *f5 = new Function("f",this);
f5->addArgument(new Variable("x",this));
f5->addArgument(f6);

Function *f4 = new Function("g",this);
f4->addArgument(f5);

Function *f3 = new Function("g",this);
f3->addArgument(new Hole());

Function *f2 = new Function("f",this);
f2->addArgument(new Function("a",this));
f2->addArgument(f3);

Function *f1 = new Function("g",this);
f1->addArgument(new Expo(f2,new NumericalVar("N1",this),new Variable("z",this)));

((And*)(result))->addBranch(new Equal(f1,new Expo(f4,new NumericalVar("N2",this),new Variable("y",
this))));
*/
//=====
checkSoundness();
}

/**
 * generation of an intermediate name for numerical variable
 */
string Parser::genNumericalValName()
{
    ostreamstream text;
    if(this->config->writeMode==Configuration::LATEX)
        text << "\\_N";
    else
        text << "_N";
    text << autoNumericalValCur;
    autoNumericalValCur++;
    return text.str();
}

/**
 * generation of an intermediate name for variable
 */
string Parser::genContstantValName()
{
    ostreamstream text;
    if(this->config->writeMode==Configuration::LATEX)
        text << "\\_C";
    else
        text << "_C";
    text << autoContstantValCur;
    autoContstantValCur++;
    return text.str();
}

void Parser::checkSoundness()
{
    {
        Node::ndterms = 0;
        result->checkSoundnessAndCalcul();
    }
}

//=====

void Node::print(std::ostream& o)
{
    o << "<?>";
}

void Condition::print(std::ostream& o)
{
    o << "<?>";
}

void And::printNoBracket(std::ostream& o)
{
    list<Condition*>::iterator it = conditions.begin();
    while(it!=conditions.end())
    {

```

```

        (*it)->print(o);
        if ((*it!=conditions.back()) || (!branches.empty()))
            if (globconfig->writeMode==Configuration::LATEX)
                o << "\\wedge_";
            else
                o << "_and_";
            it++;
        }

        list<Node*>::iterator it2 = branches.begin();
        while (it2!=branches.end())
        {
            (*it2)->print(o);
            if (*it2!=branches.back())
                if (globconfig->writeMode==Configuration::LATEX)
                    o << "\\wedge_";
                else
                    o << "_and_";
            it2++;
        }
    }

    void And::print(std::ostream& o)
    {
        o << "_(" ;
        printNoBracket(o);
        o << ")_";
    }

    void Equal::print(std::ostream& o)
    {
        n1->print(o);
        o << "_=" ;
        n2->print(o);
    }

    void Variable::print(std::ostream& o)
    {
        o << parser->variabletab[id].name ;
    }

    void Function::print(std::ostream& o)
    {
        o << parser->fonctiontab[id].name ;
        if (!arguments.empty())
        {
            o << "(" ;
            vector<Node*>::iterator it = arguments.begin();
            while (it!=arguments.end())
            {
                (*it)->print(o);
                if ((*it)!=arguments.back())
                    o << "," ;
                it++;
            }
            o << ")";
        }
    }

    void NumericalVar::print(std::ostream& o)
    {
        o << parser->numericalvariabletab[id].name ;
    }

    void CondA::print(std::ostream& o)
    {
        n1->print(o);
        o << "=" << a ;
    }

    void CondANplusB::print(std::ostream& o)
    {
        n1->print(o);
        o << "=" ;
        if (a!=1)
            o << a << "*";
        n2->print(o);
        if (b!=0)
            o << "+" << b;
    }

    void CondNplusN::print(std::ostream& o)
    {
        n1->print(o);
        o << "=" ;
        n2->print(o);
        o << "+" ;
        n3->print(o);
    }

```

```

    }

void CondN::print(std::ostream& o)
{
    n1->print(o);
    o << "=";
    n2->print(o);
}

void Expo::print(std::ostream& o)
{
    if(typeid(*n1)==typeid(Expo))
        o << "(";
    n1->print(o);
    if(typeid(*n1)==typeid(Expo))
        o << ")";
    o << "{";
    if(!free)
        var->print(o);
    else
        o << expNum;
    o << "}. ";
    if(typeid(*n2)==typeid(Expo))
        o << "(";
    n2->print(o);
    if(typeid(*n2)==typeid(Expo))
        o << ")";
}

void Or::print(std::ostream& o)
{
    o << "⋃(" << endl;
    list<Node*>::iterator it = branches.begin();
    while(it!=branches.end())
    {
        (*it)->print(o);
        if(*it!=branches.back())
            if(globconfig->writeMode==Configuration::LATEX)
                o << "\\vee⋃";
        else
            o << "⋃or⋃" << endl;
        it++;
    }
    o << endl << "⋃";
}

//=====

Node::Node()
{
    rhoterm = false;
    ndHole = false;
    holePosition = NULL;
}

Node::~Node()
{
}

Hole::~Hole()
{
    if(holePosition)
        delete holePosition;
}

Hole::Hole()
: Node()
{
    holePosition = new list<unsigned int>();
}

Condition::~Condition()
{
}

NumericalVar::NumericalVar(const char* name, Parser *parser, bool automatic)
{
    this->parser = parser;
    vector<NumericalVariableTabElement>::iterator it = parser->numericalvariabletab.begin();
    int num=0;
    while(it!=parser->numericalvariabletab.end())
    {
        if(it->name==name)
        {
            this->id = num;
            if(!automatic!=(*it).automatic)
                throw SyntaxException("using⋃of⋃reserved⋃name⋃for⋃numerical⋃variable");
        }
    }
}

```

```

        return;
    }
    it++;
    num++;
}
NumericalVariableTabElement add;
add.name = name;
add.automatic = automatic;
parser->numericalvariabletab.push_back(add);
this->id = num;
}

Function::Function(int id, Parser *parser)
: Node()
{
    this->id = id;
    this->parser = parser;
}

Function::Function()
: Node()
{
    this->id = -1;
    this->parser = NULL;
}

void Function::setName(const char* name, Parser *parser)
{
    this->parser = parser;
    vector<FunctionTabElement>::iterator it = parser->fonctiontab.begin();
    int num=0;
    while(it!=parser->fonctiontab.end())
    {
        if(it->name==name)
        {
            this->id = num;
            return;
        }
        it++;
        num++;
    }
    FunctionTabElement add;
    add.name = name;
    add.arity = -1;
    parser->fonctiontab.push_back(add);
    this->id = num;
}

Or::Or()
: Node()
{
}

void Or::addBranch(Node *n)
{
    branches.push_back(n);
}

CondA::CondA(NumericalVar* n1, int a)
{
    this->n1 = n1;
    this->a = a;
}

CondANplusB::CondANplusB(NumericalVar* n1, int a, NumericalVar* n2, int b)
{
    this->n1 = n1;
    this->n2 = n2;
    this->a = a;
    this->b = b;
}

CondNplusN::CondNplusN(NumericalVar* n1, NumericalVar* n2, NumericalVar* n3)
{
    this->n1 = n1;
    this->n2 = n2;
    this->n3 = n3;
}

CondN::CondN(NumericalVar* n1, NumericalVar* n2)
{
    this->n1 = n1;
    this->n2 = n2;
}

And::And()
: Node()

```

```

    {
    }

void And::addBranch(Node *n)
{
    branches.push_back(n);
}

void And::addCondition(Condition *c)
{
    conditions.push_back(c);
}

Equal::Equal(Node *n1, Node *n2)
: Node()
{
    this->n1 = n1;
    this->n2 = n2;
}

Variable::Variable(const char *name, Parser *parser)
: Node()
{
    this->parser = parser;
    vector<VariableTabElement>::iterator it = parser->variabletab.begin();
    int num=0;
    while(it!=parser->variabletab.end())
    {
        if(it->name==name)
        {
            this->id = num;
            return;
        }
        it++;
        num++;
    }
    VariableTabElement add;
    add.name = name;
    parser->variabletab.push_back(add);
    this->id = num;
}

Function::Function(const char* name, Parser *parser)
: Node()
{
    this->parser = parser;
    vector<FonctionTabElement>::iterator it = parser->fonctiontab.begin();
    int num=0;
    while(it!=parser->fonctiontab.end())
    {
        if(it->name==name)
        {
            this->id = num;
            return;
        }
        it++;
        num++;
    }
    FonctionTabElement add;
    add.name = name;
    add.arity = -1;
    parser->fonctiontab.push_back(add);
    this->id = num;
}

void Function::addArgument(Node *n)
{
    arguments.push_back(n);
}

Expo::Expo(Node *n1, int num, Node *n2)
: Node()
{
    this->n1 = n1;
    this->n2 = n2;
    this->expNum = num;
    free = false;
}

Expo::Expo(Node *n1, NumericalVar *num, Node *n2)
: Node()
{
    this->n1 = n1;
    this->n2 = n2;
    this->var = num;
    free = true;
}

```

```

void Hole::print(std::ostream& o)
{
    if(globconfig->writeMode==Configuration::LATEX)
        o << "\\diamond_";
    else
        o << "@";
}

void Clash::print(std::ostream& o)
{
    if(globconfig->writeMode==Configuration::LATEX)
        o << "\\bot_";
    else
        o << "<Clash>";
}

void Trivial::print(std::ostream& o)
{
    if(globconfig->writeMode==Configuration::LATEX)
        o << "\\top_";
    else
        o << "<Triv>";
}

void Node::checkSoundnessAndCalcul()
{
    Node::ndterms++;
    this->ndHole = 0;
    this->rhoterm = false;
}

void Hole::checkSoundnessAndCalcul()
{
    Node::ndterms++;
    this->ndHole = 1;
    this->rhoterm = false;
    this->holePosition->clear();
}

void And::checkSoundnessAndCalcul()
{
    Node::ndterms++;
    list<Node*>::iterator it = branches.begin();
    rhoterm = false;
    ndHole = 0;
    while(it!=branches.end())
    {
        (*it)->checkSoundnessAndCalcul();
        if((*it)->rhoterm)
            rhoterm = true;
        ndHole += (*it)->ndHole;
        it++;
    }
}

void Or::checkSoundnessAndCalcul()
{
    Node::ndterms++;
    list<Node*>::iterator it = branches.begin();
    rhoterm = false;
    ndHole = 0;
    while(it!=branches.end())
    {
        (*it)->checkSoundnessAndCalcul();
        if((*it)->rhoterm)
            rhoterm = true;
        ndHole += (*it)->ndHole;
        it++;
    }
}

void Function::checkSoundnessAndCalcul()
{
    Node::ndterms++;
    rhoterm = false;
    ndHole = 0;
    if(parser->fonctiontab[id].arity==-1)
        parser->fonctiontab[id].arity = (int)arguments.size();
    else
        if(arguments.size()!=(unsigned int)parser->fonctiontab[id].arity)
            throw SyntaxException("fonction_\" + parser->fonctiontab[id].name + "\"_with_several_
            arities");

    vector<Node*>::iterator it = arguments.begin();
    int numArg = 0;
    while(it!=arguments.end())
    {

```

```

        (*it)->checkSoundnessAndCalcul();
        if((*it)->rhoterm)
            rhoterm = true;
        ndHole += (*it)->ndHole;
        if((*it)->ndHole > 0)
        {
            this->holePosition = (*it)->holePosition;
            this->holePosition->push_front(numArg);
        }
        it++;
        numArg++;
    }
}

void Variable::checkSoundnessAndCalcul()
{
    Node::ndterms++;
    rhoterm = false;
    ndHole = 0;
}

void Equal::checkSoundnessAndCalcul()
{
    Node::ndterms++;
    rhoterm = false;
    ndHole = 0;
    n1->checkSoundnessAndCalcul();
    n2->checkSoundnessAndCalcul();

    if(n1->rhoterm)
        rhoterm = true;
    ndHole += n1->ndHole;

    if(n2->rhoterm)
        rhoterm = true;
    ndHole += n2->ndHole;
}

void Expo::checkSoundnessAndCalcul()
{
    Node::ndterms++;
    rhoterm = true;
    ndHole = 0;
    n1->checkSoundnessAndCalcul();
    n2->checkSoundnessAndCalcul();
    this->holePosition = n1->holePosition;

    if(n1->ndHole == 0)
    {
        this->print(cerr);
        throw SyntaxException("Hole not found");
    }
    if(n1->ndHole > 1)
    {
        this->print(cerr);
        throw SyntaxException("Too much hole");
    }
    if(n1->rhoterm)
        //if(n2->rhoterm)
    {
        this->print(cerr);
        throw SyntaxException("Do not tolerate rho-terms in iterated part");
    }

    if(n1->holePosition->empty())
    {
        this->print(cerr);
        throw SyntaxException("Do not tolerate single hole in iterated part");
    }
}

NodeIterator::NodeIterator(Node *n, Node **direct, Mode mode)
{
    this->mode = mode;
    this->n = n;
    this->direct = direct;
    StackElement e(n, direct);
    stack.push_back(e);
    allowUnderEqual = false;
}

void NodeIterator::setUnderEqual(bool u)
{
    allowUnderEqual = u;
}

const list<unsigned int> &NodeIterator::getPosition()

```



```

    {
        return last.position;
    }
Node* NodeIterator::next()
{
    if(stack.empty())
        return NULL;

    StackElement e;
    if(mode==NodeIterator::BREADTH_FIRST)
    {
        e = stack.front();
        stack.pop_front();
    }
    else
    {
        e = stack.back();
        stack.pop_back();
    }

    last = e;

    if(typeid(*e.n)==typeid(And))
    {
        list<Node*>::iterator it = ((And*)(e.n))->branches.begin();
        while(it!=((And*)(e.n))->branches.end())
        {
            stack.push_back(StackElement(*it,&*it));
            it++;
        }
    }
    else if(typeid(*e.n)==typeid(Or))
    {
        list<Node*>::iterator it = ((Or*)(e.n))->branches.begin();
        while(it!=((Or*)(e.n))->branches.end())
        {
            stack.push_back(StackElement(*it,&*it));
            it++;
        }
    }
    else if(typeid(*e.n)==typeid(Function))
    {
        vector<Node*>::iterator it = ((Function*)(e.n))->arguments.begin();
        int num = 0;
        while(it!=((Function*)(e.n))->arguments.end())
        {
            stack.push_back(StackElement(*it,&*it,e.position,num));
            num++;
            it++;
        }
    }
    else if(typeid(*e.n)==typeid(Equal))
    {
        if(allowUnderEqual)
        {
            stack.push_back(StackElement(((Equal*)(e.n))->n1,&((Equal*)(e.n))->n1));
            stack.push_back(StackElement(((Equal*)(e.n))->n2,&((Equal*)(e.n))->n2));
        }
    }
    return e.n;
}

StackElement::StackElement(Node *n,Node **direct)
{
    this->n = n;
    this->direct = direct;
}

StackElement::StackElement(Node *n,Node **direct,list<unsigned int> &parentpos,int me)
{
    this->n = n;
    this->direct = direct;
    position = parentpos;
    position.push_back(me);
}

StackElement::StackElement()
{
}

/**
 * replace the current element by n (you can't use next anymore)
 */
void NodeIterator::replaceBy(Node *n)
{
    if(last.direct)
        (*last.direct) = n;
}

```

```

        else
            assert(false); // Can't change node
    }

/**
 * come back to the initial position
 */
void NodeIterator::reset()
{
    stack.clear();
    StackElement e(n,direct);
    stack.push_back(e);
}

/**
 * the search for a variable
 */
bool Node::containVar(int idvar)
{
    return false;
}

bool Variable::containVar(int idvar)
{
    return id==idvar;
}

bool Function::containVar(int idvar)
{
    vector<Node*>::iterator it = arguments.begin();
    while(it!=arguments.end())
    {
        if((*it)->containVar(idvar))
            return true;
        it++;
    }
    return false;
}

bool And::containVar(int idvar)
{
    list<Node*>::iterator it = branches.begin();
    while(it!=branches.end())
    {
        if((*it)->containVar(idvar))
            return true;
        it++;
    }
    return false;
}

bool Or::containVar(int idvar)
{
    list<Node*>::iterator it = branches.begin();
    while(it!=branches.end())
    {
        if((*it)->containVar(idvar))
            return true;
        it++;
    }
    return false;
}

bool Equal::containVar(int idvar)
{
    return n1->containVar(idvar) || n2->containVar(idvar);
}

bool Expo::containVar(int idvar)
{
    return n1->containVar(idvar) || n2->containVar(idvar);
}

/**
 * repalce a variable by a generic node
 */
void Node::replace(int idvar,Node *n)
{
}

void Function::replace(int idvar,Node *n)
{
    vector<Node*>::iterator it = arguments.begin();
    while(it!=arguments.end())
    {
        if((typeid(*it)==typeid(Variable))&&(((Variable*)(*it))->id==idvar))
            *it = n->duplicate();
        (*it)->replace(idvar,n);
    }
}

```

```

        it++;
    }
}

void Or::replace(int idvar, Node *n)
{
    list<Node*>::iterator it = branches.begin();
    while(it!=branches.end())
    {
        if((typeid(**it)==typeid(Variable))&&(((Variable*)(*it))->id==idvar))
            *it = n->duplicate();
        (*it)->replace(idvar,n);
        it++;
    }
}

void And::replace(int idvar, Node *n)
{
    list<Node*>::iterator it = branches.begin();
    while(it!=branches.end())
    {
        if((typeid(**it)==typeid(Variable))&&(((Variable*)(*it))->id==idvar))
            *it = n->duplicate();
        (*it)->replace(idvar,n);
        it++;
    }
}

void Equal::replace(int idvar, Node *n)
{
    if((typeid(*n1)==typeid(Variable))&&(((Variable*)n1)->id==idvar))
        n1 = n->duplicate();

    if((typeid(*n2)==typeid(Variable))&&(((Variable*)n2)->id==idvar))
        n2 = n->duplicate();

    n1->replace(idvar,n);
    n2->replace(idvar,n);
}

void Expo::replace(int idvar, Node *n)
{
    if((typeid(*n1)==typeid(Variable))&&(((Variable*)n1)->id==idvar))
        n1 = n->duplicate();

    if((typeid(*n2)==typeid(Variable))&&(((Variable*)n2)->id==idvar))
        n2 = n->duplicate();

    n1->replace(idvar,n);
    n2->replace(idvar,n);
}

Node* Hole::duplicate()
{
    {
        return new Hole();
    }
}

Node* Clash::duplicate()
{
    {
        return new Clash();
    }
}

Node* Trivial::duplicate()
{
    {
        return new Trivial();
    }
}

NumericalVar* NumericalVar::duplicate()
{
    {
        return new NumericalVar(*this);
    }
}

Condition* CondANplusB::duplicate()
{
    {
        return new CondANplusB(n1->duplicate(),a,n2->duplicate(),b);
    }
}

Condition* CondA::duplicate()
{
    {
        return new CondA(n1->duplicate(),a);
    }
}

Condition* CondNplusN::duplicate()
{
    {
        return new CondNplusN(n1->duplicate(),n2->duplicate(),n3->duplicate());
    }
}

Condition* CondN::duplicate()

```

```

    {
        return new CondN(n1->duplicate(), n2->duplicate());
    }

Node* Variable::duplicate()
{
    return new Variable(*this);
}

Node* Function::duplicate()
{
    Function *dme= new Function(id, parser);
    vector<Node*>::iterator it = arguments.begin();
    while(it!=arguments.end())
    {
        dme->addArgument((*it)->duplicate());
        it++;
    }
    return dme;
}

Node* Or::duplicate()
{
    Or *dme= new Or();
    list<Node*>::iterator it = branches.begin();
    while(it!=branches.end())
    {
        dme->addBranch((*it)->duplicate());
        it++;
    }
    return dme;
}

Node* And::duplicate()
{
    And *dme= new And();
    list<Node*>::iterator it = branches.begin();
    while(it!=branches.end())
    {
        dme->addBranch((*it)->duplicate());
        it++;
    }
    return dme;
}

Node* Equal::duplicate()
{
    return new Equal(n1->duplicate(), n2->duplicate());
}

Node* Expo::duplicate()
{
    if (free)
        return new Expo(n1->duplicate(), var->duplicate(), n2->duplicate());
    else
        return new Expo(n1->duplicate(), expNum, n2->duplicate());
}

/**
 * unfold two time the exp term
 */
Node* Expo::buildTwo(Node *sub)
{
    return buildOne(buildOne(sub));
}

/**
 * unfold one time the exp term
 */
Node* Expo::buildOne(Node *sub)
{
    if (typeid(*n1)==typeid(Hole))
        return ((sub)?sub:(n2->duplicate()));

    Node *ret = n1->duplicate();

    Node *function = ret;
    Node **hole;
    list<unsigned int>::iterator it = holePosition->begin();
    while(it!=holePosition->end())
    {
        assert(typeid(*function)==typeid(Function)); // This node is not a function
        hole = &((Function*)function)->arguments[*it];
        function = ((Function*)function)->arguments[*it];
        it++;
    }
    assert(typeid(**hole)==typeid(Hole)); // This node is not a hole

```

```

        delete function;
        *hole = ((sub)?sub:(n2->duplicate()));
        return ret;
    }

/**
 * duplicate the tree and replace the node a the position path by sub
 */
Node* Node::buildOneAndReplace(list<unsigned int> &path, Node *sub)
{
    assert(path.empty());
    return sub;
}

Node* Function::buildOneAndReplace(list<unsigned int> &path, Node *sub)
{
    assert(typeid(*this)==typeid(Function));
    if(path.empty())
        return sub;

    unsigned int top = path.back();
    path.pop_back();

    Function *ret = new Function(id, parser);
    vector<Node*>::iterator it = arguments.begin();
    unsigned int num = 0;
    while(it!=arguments.end())
    {
        if(num==top)
        {
            if(path.empty())
                ret->addArgument(sub);
            else
                ret->addArgument(((Function*)(*it))->buildOneAndReplace(path, sub));
        }
        else
            ret->addArgument((*it)->duplicate());
        it++;
        num++;
    }
    return ret;
}

/**
 * search for the node at the position path
 */
Node* Node::find(list<unsigned int> &path)
{
    assert(path.empty());
    return this;
}

Node* Function::find(list<unsigned int> &path)
{
    assert(typeid(*this)==typeid(Function));

    if(path.empty())
        return this;

    unsigned int top = path.back();
    path.pop_back();

    assert(top<arguments.size());
    if(path.empty())
        return arguments[top];
    else
    {
        assert(typeid(*((Function*)arguments[top]))==typeid(Function));
        return ((Function*)arguments[top])->find(path);
    }
}

/**
 * equality of structure
 */
bool Node::isEqual(Node *n)
{
    return (typeid(*this)==typeid(*n));
}

bool NumericalVar::isEqual(Node *n)
{
    if(typeid(*this)!=typeid(*n))
        return false;
    return (this->id==((NumericalVar*)n)->id);
}

bool Variable::isEqual(Node *n)

```

```

    {
        if (typeid(*this) != typeid(*n))
            return false;
        return (this->id == ((Variable*)n)->id);
    }

bool Function::isEqual(Node *n)
{
    if (typeid(*this) != typeid(*n))
        return false;

    if (this->id != ((Function*)n)->id)
        return false;

    vector<Node*>::iterator it = arguments.begin();
    vector<Node*>::iterator it2 = ((Function*)n)->arguments.begin();
    while (it != arguments.end())
    {
        if (!(*it)->isEqual(*it2))
            return false;
        it2++;
        it++;
    }

    return true;
}

bool Or::isEqual(Node *n)
{
    if (typeid(*this) != typeid(*n))
        return false;

    list<Node*>::iterator it = branches.begin();
    list<Node*>::iterator it2 = ((Or*)n)->branches.begin();
    while (it != branches.end())
    {
        if (!(*it)->isEqual(*it2))
            return false;
        it2++;
        it++;
    }

    return true;
}

bool And::isEqual(Node *n)
{
    if (typeid(*this) != typeid(*n))
        return false;

    list<Node*>::iterator it = branches.begin();
    list<Node*>::iterator it2 = ((And*)n)->branches.begin();
    while (it != branches.end())
    {
        if (!(*it)->isEqual(*it2))
            return false;
        it2++;
        it++;
    }

    list<Condition*>::iterator it3 = conditions.begin();
    list<Condition*>::iterator it4 = ((And*)n)->conditions.begin();
    while (it3 != conditions.end())
    {
        if (!(*it3)->isEqual(*it4))
            return false;
        it3++;
        it4++;
    }

    return true;
}

bool Equal::isEqual(Node *n)
{
    if (typeid(*this) != typeid(*n))
        return false;
    if (!this->n1->isEqual(((Equal*)n)->n1))
        return false;
    if (!this->n2->isEqual(((Equal*)n)->n2))
        return false;
    return true;
}

bool Expo::isEqual(Node *n)
{
    if (typeid(*this) != typeid(*n))
        return false;
    if (!this->n1->isEqual(((Expo*)n)->n1))

```

```

        return false;
    if(!this->n2->isEqual(((Expo*)n)->n2))
        return false;

    if(this->free==((Expo*)n)->free)
    if(free)
        return this->var->id == ((Expo*)n)->var->id;
    else
        return this->expNum == ((Expo*)n)->expNum;

    return false;
}

bool CondA::isEqual(Condition *n)
{
    if(typeid(*this)!=typeid(*n))
        return false;
    return false;
}

bool CondANplusB::isEqual(Condition *n)
{
    if(typeid(*this)!=typeid(*n))
        return false;
    return false;
}

bool CondNplusN::isEqual(Condition *n)
{
    if(typeid(*this)!=typeid(*n))
        return false;
    return false;
}

bool CondN::isEqual(Condition *n)
{
    if(typeid(*this)!=typeid(*n))
        return false;
    return false;
}

Node* Expo::build(int num,Node *sub)
{
    Node *ret = sub;
    int i;
    for(i=0;i<num;i++)
        ret = buildOne(ret);
    return ret;
}

```

## 6.6.7 parser.ll

```

%{

#include <string>

#include "parser.h"
#include "parser.tab.hh"

#undef yywrap
#define yywrap() 1
extern int numLine;

%}

%option noyywrap
%option c++

letter      [a-z]
LETTER      [A-Z]
number      [0-9]
impr        [\040-\176]
Car_imp     [\040-\176]
comment     "#"({ Car_imp }|\t)*
blancs      [ \t]+
MotMaj      ({ LETTER }){ letter }| "-" |{ number })*
MotMin      ({ letter }){ letter }| "-" |{ number })*
MotNum      number+

%% /** Regular Expressions Part **/

{comment} { }

```

```

{blancs} { }

{number}+ {
    yylval.integerVal = atoi(yytext);
    return INT;
}

"run" return (RUN);

"set" return (SET);
"help" return (HELP);

"step" return (STEP);
"info" return (INFO);
"zero" return (ZERO);
"beep" return (BEEP);

"or" return (OR);

"and" return (AND);

"OR" return (OR);

"AND" return (AND);

{MotNum} {
    yylval.integerVal = atoi(yytext);
    return (INT);
}

{MotMaj} {
    yylval.stringVal = new std::string(yytext);
    return (MAJ);
}

{MotMin} {
    yylval.stringVal = new std::string(yytext);
    return (MIN);
}

"=" {
    return EQUAL;
}

"==" {
    return EQUAL2;
}

"(" {
    return LEFT_PAR;
}

")" {
    return RIGHT_PAR;
}

"{" {
    return LEFT_BRA;
}

"}" {
    return RIGHT_BRA;
}

"@" {
    return HOLE;
}

", " {
    return COMMA;
}

"." {
    return DOT;
}

"*" {
    return TIMES;
}

"+" {
    return PLUS;
}

"^" {
    return POWER;
}

```



```
[ \t\r]+ {
}

"\n" {
    numLine++;
    return EOL;
}
```

## 6.6.8 parser.yy

```
%{

#include "FlexLexer.h"
#include <stdio.h>
#include <string>
#include <vector>
#include "parser.h"
#include "config.h"

int yylex();

extern Node **result;
extern Parser* extParser;
extern void set_extParser(Parser* extP);
extern int yyerror(char *s);
extern void set_result(Node **r);
extern yyFlexLexer* lexer;
extern int numLine;

%}

%defines

%union {
    int integerVal;
    std::string* stringVal;
    Node* node;
    Condition* cond;
    NumericalVar* numvar;
}

%token END_OF_FILE "end_of_file"
%token EOL "end_of_line"
%token AND "and"
%token OR "or"
%token EQUAL "equal"
%token EQUAL2 "equal_equal"
%token LEFT_PAR "left_parenthesis"
%token RIGHT_PAR "right_parenthesis"
%token POWER "^ie_power"
%token LEFT_BRA "{ie_left_brace"
%token RIGHT_BRA "}ie_right_braces"
%token HOLE "@ie_hole"
%token COMMA ",ie_comma"
%token <stringVal> MAJ "strings_beginning_by_a_high_case_char"
%token <stringVal> MIN "strings_beginning_by_a_low_case_char"
%token <integerVal> INT "integer"
%token PLUS "+"
%token TIMES "*"
%token RUN "run"
%token DOT
%token SET
%token HELP
%token STEP
%token INFO
%token ZERO
%token BEEP

%type <node> T Tand Tor Tf A Input Ligne Tp Tpf Teq
%type <numvar> NumV
%type <cond> Cond
%type <stringVal> FUNC

%start Input

%%

Input:
    Ligne { if($1) { $$ = new And(); ((And*)$$)->addBranch($1); *result = $$;} else {$$ = NULL; *
        result = $$; } }
```

```

    | Input Ligne { if($1) $$ = $1; else { $$ = new And(); *result = $$; } if($2) ((And*)$$)->addBranch
    | ($2); }
;

Ligne:
EOL { $$ = NULL; }
| RUN EOL { $$ = NULL; return 0; }
| HELP EOL { Configuration::printConsol(); $$ = NULL; }
| SET OPTION EOL { $$ = NULL; }
| T EOL { $$ = $1; }
;

OPTION:
STEP { extParser->config->pauseEveryStep = true; }
| INFO { extParser->config->printRule = true; }
| ZERO { /* extParser->config->NcanbeZero = true; */ }
| BEEP { extParser->config->beep = 1; }
;

T:
Tf
| Tand AND Tf { $$ = $1; ((And*)$$)->addBranch($3); }
| Tor OR Tf { $$ = $1; ((Or*)$$)->addBranch($3); }
/* | Tf EQUAL Tf { $$ = new Equal($1, $3); } */
;

Tand:
Tf { $$ = new And(); ((And*)$$)->addBranch($1); }
| Cond { $$ = new And(); ((And*)$$)->addCondition($1); }
| Tand AND Cond { $$ = $1; ((And*)$$)->addCondition($3); }
| Tand AND Tf { $$ = $1; ((And*)$$)->addBranch($3); }
;

Tor:
Tf { $$ = new Or(); ((Or*)$$)->addBranch($1); }
| Tor OR Tf { $$ = $1; ((Or*)$$)->addBranch($3); }
;

Tf:
LEFT_PAR T RIGHT_PAR { $$ = $2; }
| Teq { $$ = $1; }
;

Teq:
Tp EQUAL Tp { $$ = new Equal($1, $3); }
;

Tp:
HOLE { $$ = new Hole(); }
| FUNC LEFT_PAR A RIGHT_PAR { $$ = $3; ((Function*)$$)->setName($1->c_str(), extParser); delete
$1; }
| MIN { $$ = new Function($1->c_str(), extParser ); delete $1; }
| MAJ { $$ = new Variable($1->c_str(), extParser ); delete $1; }
| Tp POWER LEFT_BRA NumV RIGHT_BRA DOT Tpf { $$ = new Expo($1,$4,$7); }
| Tp POWER LEFT_BRA INT RIGHT_BRA DOT Tpf { $$ = new Expo($1,$4,$7); }
;

/* experience pourrie de Gaby pour desambiguifier la grammaire */
Tpf:
LEFT_PAR Tp RIGHT_PAR { $$ = $2; }
| FUNC LEFT_PAR A RIGHT_PAR { $$ = $3; ((Function*)$$)->setName($1->c_str(), extParser); delete
$1; }
| MIN { $$ = new Function($1->c_str(), extParser ); delete $1; }
| MAJ { $$ = new Variable($1->c_str(), extParser ); delete $1; }
;

FUNC:
MAJ { $$ = $1; }
| MIN { $$ = $1; }
;

A:
Tp { $$ = new Function(); ((Function*)$$)->addArgument($1); }
| A COMMA Tp { $$ = $1; ((Function*)$$)->addArgument($3); }
;

Cond:
NumV EQUAL2 INT TIMES NumV PLUS INT { $$ = new CondANplusB($1, $3, $5, $7); }
| NumV EQUAL2 INT TIMES NumV { $$ = new CondANplusB($1, $3, $5, 0); }
| NumV EQUAL2 NumV PLUS NumV { $$ = new CondNplusN($1, $3, $5); }
| NumV EQUAL2 NumV { $$ = new CondN($1, $3); }
;

NumV:
MAJ { $$ = new NumericalVar($1->c_str(), extParser ,false); delete $1; }
| MIN { $$ = new NumericalVar($1->c_str(), extParser ,false); delete $1; }
;

%%

```

```

Parser* extParser;
Node **result;
yyFlexLexer* lexer;
int numLine;

void set_result(Node **r)
{
    result = r;
}

void set_extParser(Parser* extP)
{
    extParser = extP;
}

int yyerror(char *s)
{
    printf("\nErreur : %s\n", s, numLine);
    return 1;
}

int yylex(void)
{
    return lexer->yylex();
}

```

## 6.6.9 synnaeve.guillame.bert.h

```

#include "synnaeve.guillame.bert.h"
#include "unification.h"
#include <assert.h>
#include <iostream>
#include <string>
#include <climits>

namespace Solver
{
    bool Vertex::setBounds(int min, int max)
    {
        bool ch = false;
        if(min > this->min)
            { this->min = min; ch = true; }
        if(max < this->max)
            { this->max = max; ch = true; }
        return ch;
    }

    bool Vertex::setBoundMin(int min)
    {
        if(min > this->min)
            { this->min = min; return true; }
        return false;
    }

    bool Vertex::setBoundMax(int max)
    {
        if(max < this->max)
            { this->max = max; return true; }
        return false;
    }

    Graph::Graph(Node* node, Unification *unification)
    {
        this->unification = unification;
        solus = new SolverSolution();
        solus->unification = unification;

        // ===== DEBUG =====
        Vertex *v0 = addVertex(0);
        Vertex *v1 = addVertex(1);
        Vertex *v2 = addVertex(2);
        Vertex *v3 = addVertex(3);
        Vertex *v4 = addVertex(4);
        Vertex *v5 = addVertex(5);
        Vertex *v6 = addVertex(-1);
        Vertex *v7 = addVertex(-2);

        addEdge(v0, v1, 2, 0);

        addEdge(v1, v7, 1, 0);
    }

```

```

        addEdge(v2,v7,1,0);
        addEdge(v3,v7,-1,0);

        addEdge(v0,v6,1,0);
        addEdge(v5,v6,1,0);
        addEdge(v4,v6,-1,0);

        addEdge(v4,v3,1,2);

        success = true;
        solus->ndvar = 0;
        return;

        solus->ndvar = unification->parser->variabletab.size();
        solus->variableSolution = new VariableSolution[solus->ndvar];
        success = explo(node);
    }

Vertex* Graph::addVertex(int id)
{
    if(id>=0)
    {
        list<Vertex*>::iterator it = vertexs.begin();
        while(it!=vertexs.end())
        {
            if((*it)->variable==id)
                return *it;
            it++;
        }
    }

    Vertex *v = new Vertex();
    v->count = 0;
    v->variable = id;
    v->min = (unification->configuration->NcanbeZero)?0:1;
    v->max = INT_MAX;
    v->father = NULL;
    v->visited = false;
    this->vertexs.push_back(v);
    return v;
}

Edge* Graph::addEdge(Vertex *v1,Vertex *v2,int a,int b)
{
    Edge *e = new Edge();
    e->a = a;
    e->b = b;
    e->n1 = v1;
    e->n2 = v2;
    v1->out.push_back(e);
    v2->in.push_back(e);
    this->edges.push_back(e);
    return e;
}

bool Graph::explo(Node* n)
{
    bool val = true;
    int first = (unification->configuration->NcanbeZero)?0:1;

    if(typeid(*n)==typeid(Clash))
        val = false;
    else if(typeid(*n)==typeid(Trivial))
        val = true;
    else if(typeid(*n)==typeid(Equal))
    {
        Equal *e = (Equal*)n;
        val = true;
        if(typeid(*e->n1)==typeid(Variable))
            solus->variableSolution[(((Variable*)(e->n1))->id).definition = e->n2;
        else
            return false;

        if(!explo(e->n2))
            val = false;
    }
    else if(typeid(*n)==typeid(And))
    {
        And *a = (And*)n;
        val = true;
        list<Node*>::iterator it = a->branchs.begin();
        while(it!=a->branchs.end())
        {
            if(!explo(*it))
            {
                val = false;
                break;
            }
        }
    }
}

```

```

        it++;
    }
    list<Condition*>::iterator it2 = a->conditions.begin();
    while(it2!=a->conditions.end())
    {
        Vertex *v = addVertex((*it2)->n1->id);

        if(typeid(**it2)==typeid(CondANplusB))
        {
            Edge *e = addEdge(addVertex(((CondANplusB*)(*it2))->n2->id),v,((CondANplusB*)(*it2))->
                a,((CondANplusB*)(*it2))->b);
            if((e->a<0)&&(e->b<=0))
            {
                e->n1->setBoundMax(-(e->b-first)/e->a);
                e->n2->setBoundMax(e->b + e->a*first);
            }
            else
            {
                if((e->a<0)&&(e->b<0))
                {
                    if(first==0)
                    {
                        e->n1->setBounds(0,0);
                        e->n2->setBounds(0,0);
                    }
                    else
                        return false;
                }
            }
            else if(typeid(**it2)==typeid(CondNplusN))
            {
                Vertex *onode = addVertex(-1);
                addEdge(v,onode,-1,0);
                addEdge(addVertex(((CondNplusN*)(*it2))->n2->id),onode,1,0);
                addEdge(addVertex(((CondNplusN*)(*it2))->n3->id),onode,1,0);
            }
            else if(typeid(**it2)==typeid(CondN))
            {
                addEdge(addVertex(((CondANplusB*)(*it2))->n2->id),v,1,0);
            }
            else if(typeid(**it2)==typeid(CondA))
            {
                v->setBounds(((CondA*)(*it2))->a,((CondA*)(*it2))->a);
            }
            it2++;
        }
    }
    else if(typeid(*n)==typeid(Function))
    {
        Function *f = (Function*)n;
        vector<Node*>::iterator it = f->arguments.begin();
        val = true;
        while(it!=f->arguments.end())
        {
            if(!explo(*it))
            {
                val = false;
                break;
            }
            it++;
        }
    }
    else if(typeid(*n)==typeid(Or))
    {
        assert(false); // Ceci n'est pas une forme disjonctive (les Ors on déjà été traités)
    }
    else if(typeid(*n)==typeid(Expo))
    {
        Expo *e = (Expo*)n;
        val = explo(e->n1);
        if(!explo(e->n2))
            val = false;
        if(e->free)
            addVertex(e->var->id);
        //solution->numericalvariableSolution[e->var->id].seeoutside = true;
    }
    else if(typeid(*n)==typeid(Variable))
    {
        Variable *v = (Variable*)n;
        solus->variableSolution[v->id].seeoutside = true;
        val = true;
    }
    return val;
}

SolverSolution *Graph::run()
{
    if(!success)
    {
        solus->success = false;
        return solus;
    }

    cout << "NDNoeuds:" << this->vertexs.size() << endl;
    cout << "NDarcs:" << this->edges.size() << endl;

```

```

cout << endl;

print();

list<Vertex*> vertexs;
list<Edge*> edges;
while(seekCycle(vertexs, edges))
{
    cout << "Cycle: " << vertexs.size() << " " << edges.size() << endl;
    cout << "uuu";
    list<Vertex*>::iterator it;
    it = vertexs.begin();
    while(it!=vertexs.end())
    {
        cout << (*it)->variable;
        it++;
    }
    cout << endl;

    list<Contribution> contribution;
    list<Contribution>::iterator it3;

    list<Edge*>::iterator it2 = edges.begin();
    float a = 1;
    float b = 0;
    Vertex *prec;

    it = vertexs.end();
    it--;
    it--;
    Vertex *cur = *it;

    it = vertexs.begin();
    while(it2!=edges.end())
    {
        prec = cur;
        cur = *it2;
        it++;

        if(cur->variable < 0)
        {
            list<Edge*>::iterator it4;
            it4 = cur->in.begin();
            while(it4!=cur->in.end())
            {
                if((((*it4)->n1)!=prec)&&((( (*it4)->n1)!=(*it))))
                {
                    Contribution tmp;
                    tmp.a = (float)(*it4)->a;
                    b += (float)(*it4)->b;
                    tmp.v = cur;
                    contribution.push_back(tmp);
                }
                it4++;
            }

            it4 = cur->out.begin();
            while(it4!=cur->out.end())
            {
                if((((*it4)->n2)!=prec)&&((( (*it4)->n2)!=(*it))))
                {
                    Contribution tmp;
                    tmp.a = (float)(*it4)->a;
                    b += (float)(*it4)->b;
                    tmp.v = cur;
                    contribution.push_back(tmp);
                }
                it4++;
            }
        }
        it2++;
    }

    float way = (cur==(*it2)->n1)?1.f:-1.f;

    if(way==1)
    {
        a *= (*it2)->a;
        b *= (*it2)->a;
        b += (*it2)->b;
    }
    else
    {
        a /= (*it2)->a;
        b /= (*it2)->a;
        b -= (*it2)->b/(*it2)->a;
    }

    it3 = contribution.begin();

```

```

        while (it3 != contribution.end())
        {
            if (way == 1)
                (*it3).a *= (*it2) -> a;
            else
                (*it3).a /= (*it2) -> a;
            it3++;
        }

        it2++;
    }
    a--;

    if (contribution.empty())
    {
        if (b != 0)
        {
            solus -> success = false;
            return solus;
        }
        if (a != 1)
        {
            solus -> success = false;
            return solus;
        }
        else
        {
            if (!unification -> configuration -> NcanbeZero)
            {
                solus -> success = false;
                return solus;
            }
            else
            {
                vertices.front() -> setBounds(0, 0);
                // removeEdge(edges.front());
            }
        }
    }
    else
    {
        // Vertex *o = addVertex(-1);
        // TODO
    }

    contribution.clear();
    vertices.clear();
    edges.clear();
}

cout << "Plus de cycle" << endl;
print();

return solus;
}

bool SolverSolution::check()
{
    return success;
}

void SolverSolution::display(ostream &o)
{
    unsigned int i;
    for (i = 0; i < ndvar; i++)
    {
        o << "u" << unification -> parser -> variabletab[i].name << "u=u";
        if (!variableSolution[i].definition)
            o << "**free**";
        else
            variableSolution[i].definition -> print(o);
        o << endl;
    }
}

Graph::~Graph()
{
}

bool Graph::seekCycle(list<Vertex*> &ver, list<Edge*> &ed)
{
    if (vertices.empty())
        return false;

    resetVisited();

    list<Vertex*> tovisit;
    vertices.front() -> visited = true;

```

```

    tovisit.push_back(vertices.front());
    Vertex *v;
    Vertex *v2;
    Edge *e;

    while(!tovisit.empty())
    {
        v = tovisit.back();
        tovisit.pop_back();

        list<Edge*>::iterator it2;

        it2 = v->in.begin();
        while(it2!=v->in.end())
        {
            if(*it2!=v->father)
            {
                tovisit.push_back((*it2)->n1);
                if((*it2)->n1->visited)
                {
                    e = *it2;
                    v2 = (*it2)->n1;
                    goto buildpath;
                }
                (*it2)->n1->father = *it2;
                (*it2)->n1->visited = true;
            }
            it2++;
        }

        it2 = v->out.begin();
        while(it2!=v->out.end())
        {
            if(*it2!=v->father)
            {
                tovisit.push_back((*it2)->n2);
                if((*it2)->n2->visited)
                {
                    e = *it2;
                    v2 = (*it2)->n2;
                    goto buildpath;
                }
                (*it2)->n2->father = *it2;
                (*it2)->n2->visited = true;
            }
            it2++;
        }
    }
    return false;
buildpath::

    ed.push_back(e);
    resetVisited();

    Vertex *cur = v2;
    while(cur)
    {
        cur->visited = true;
        if(cur->father)
            cur = (cur==cur->father->n1)?cur->father->n2:cur->father->n1;
        else
            cur = NULL;
    }

    cur = v;
    while(!cur->visited)
        cur = (cur==cur->father->n1)?cur->father->n2:cur->father->n1;

    while(v2!=cur)
    {
        ver.push_front(v2);
        ed.push_front(v2->father);
        v2 = (v2==v2->father->n1)?v2->father->n2:v2->father->n1;
    }

    while(v!=cur)
    {
        ver.push_back(v);
        ed.push_back(v->father);
        v = (v==v->father->n1)?v->father->n2:v->father->n1;
    }

    ver.push_front(cur);
    ver.push_back(cur);
    return true;
}

void Graph::resetVisited()

```



```

    {
        list<Vertex*>::iterator it = vertexs.begin();
        while(it!=vertexs.end())
            (*(it++))>visited = false;
    }

void Graph::print()
{
    cout << "└─Vertex:" << endl;
    list<Vertex*>::iterator it = vertexs.begin();
    while(it!=vertexs.end())
    {
        cout << "┌┌┌" << (*(it))>variable << "[" << (*(it))>min << "," << (*(it))>max << "]" << endl;
        it++;
    }

    cout << "└─Edges:" << endl;
    list<Edge*>::iterator it2 = edges.begin();
    while(it2!=edges.end())
    {
        cout << "┌┌┌" << (*(it2))>n1->variable << "\t-[" << (*(it2))>a << ";" << (*(it2))>b << "]->\t" <<
            (*(it2))>n2->variable << endl;
        it2++;
    }
}

};

```

### 6.6.10 unification.h

```

#include "unification.h"
#include "exception.h"
#include "display.h"
#include <iostream>
#include <typeinfo>
#include <time.h>
#include "synnaeve.guillame.bert.h"
#include "automata.h"
#include <map>
#include <string.h>

#ifdef WIN32
#include <windows.h>
#else

#endif

Unification::Unification(Configuration *configuration, Parser *parser)
{
    this->configuration = configuration;
    this->parser = parser;
}

/**
 * display current state of the memory
 */
void Unification::debugDisplay(ostream &s)
{
    if(globconfig->writeMode==Configuration::LATEX)
        s << "$";

    if(typeid(*parser->result)==typeid(And))
        dynamic_cast<And*>(parser->result)->printNoBracket(s);
    else
        parser->result->print(s);

    if(globconfig->writeMode==Configuration::LATEX)
        s << "$\\\\\\\\";

    s << endl;

    /**
     * Exemple of iteration
     */
    NodeIterator it(parser->result,&parser->result,NodeIterator::DEPTH_FIRST);
    Node *cur;
    while(cur=it.next())
    {
        s << typeid(*cur).name() << endl;
    }
}

/**
 * transformation in disjonctive normal form

```

```

*/
Node* Unification::SetDisjunctiveSystem(Node *start)
{
    if(!start)
        start = parser->result;

    Node *ret = start;

    bool action = false;
    bool nothingtodo;
    do
    {
        nothingtodo = true;
        NodeIterator it(ret,&ret,NodeIterator::BREADTH_FIRST);
        Node *cur;

        while((cur=it.next()))
        {
            if(typeid(*cur)==typeid(And))
            {
                list<Node*>::iterator it2 = ((And*)cur)->branches.begin();
                while(it2!=((And*)cur)->branches.end())
                {
                    if(typeid(*(it2))==typeid(Or))
                    {
                        Or *newOr = new Or();
                        list<Node*>::iterator it3 = ((Or*)(it2))->branches.begin();
                        while(it3!=((Or*)(it2))->branches.end())
                        {
                            And *newAnd = new And();
                            bool last = ((it3)==((Or*)(it2))->branches.back());
                            newAnd->addBranch(*it3);

                            //parcour des conditions et ajout
                            list<Condition*>::iterator it5 = ((And*)cur)->conditions.begin();
                            while(it5!=((And*)cur)->conditions.end())
                            {
                                newAnd->addCondition((*it5)->duplicate());
                                //if(last)
                                //    delete (*it5);
                                it5++;
                            }

                            //parcour du And et ajout (sauf it2)
                            list<Node*>::iterator it4 = ((And*)cur)->branches.begin();
                            while(it4!=((And*)cur)->branches.end())
                            {
                                if(it4!=it2)
                                    newAnd->addBranch((*it4)->duplicate());
                                //if(last)
                                //    delete (*it4);
                                it4++;
                            }

                            newOr->addBranch(newAnd);

                            if(last)
                                break;

                            it3++;
                        }

                        it.replaceBy(newOr);
                        action = true;
                        nothingtodo = false;
                        goto endthisloop;
                    }
                    it2++;
                }
            }
        }
    }
    endthisloop::
    {
        ret = ApplyR0(ret);
    }
    while(!nothingtodo);
    return ret;
}

/**
 * Application of rules R0 (simplification)
 */
Node* Unification::ApplyR0(Node *start)
{
    if(!start)
        start = parser->result;

```

```

Node *ret = start;

bool nothingtodo;
do
{
    nothingtodo = true;
    NodeIterator it(ret,&ret,NodeIterator::DEPTH_FIRST);
    it.setUnderEqual(true);
    Node *cur;

    while((cur=it.next()))
    {
        if(typeid(*cur)==typeid(And))
        {
            if(((And*)cur)->branches.size()==1&&(((And*)cur)->conditions.empty()))
            {
                it.replaceBy(((And*)cur)->branches.front());
                delete cur;
                nothingtodo = false;
                goto endthisloop;
            }
            list<Node*>::iterator it2 = ((And*)cur)->branches.begin();
            while(it2!=((And*)cur)->branches.end())
            {
                if(typeid(*(it2))==typeid(And))
                {
                    And *subAnd = (And*)(it2);
                    ((And*)cur)->branches.erase(it2);

                    list<Node*>::iterator it3 = subAnd->branches.begin();
                    while(it3!=subAnd->branches.end())
                    {
                        ((And*)cur)->branches.push_back(*it3);
                        it3++;
                    }

                    list<Condition*>::iterator it4 = subAnd->conditions.begin();
                    while(it4!=subAnd->conditions.end())
                    {
                        ((And*)cur)->conditions.push_back(*it4);
                        it4++;
                    }

                    nothingtodo = false;
                    goto endthisloop;
                }
                if(typeid(*(it2))==typeid(Clash))
                {
                    it.replaceBy(new Clash());
                    nothingtodo = false;
                    goto endthisloop;
                }
                if(typeid(*(it2))==typeid(Trivial))
                {
                    ((And*)cur)->branches.erase(it2);
                    nothingtodo = false;
                    goto endthisloop;
                }
                it2++;
            }
        }

        if(typeid(*cur)==typeid(Expo))
        if(!((Expo*)cur)->free)
        {
            if(((Expo*)cur)->expNum==0)
            {
                {
                    // TODO -> delete ((Expo*)cur)->n1
                    it.replaceBy(((Expo*)cur)->n2);
                    nothingtodo = false;
                    goto endthisloop;
                }
            }
            else
            {
                Node *n = ((Expo*)cur)->build(((Expo*)cur)->expNum, NULL);
                // TODO -> delete ((Expo*)cur)
                it.replaceBy(n);
                nothingtodo = false;
                goto endthisloop;
            }
        }

        if(typeid(*cur)==typeid(Or))
        {
            if(((Or*)cur)->branches.size()==1)
            {
                it.replaceBy(((Or*)cur)->branches.front());
                delete cur;
            }
        }
    }
}

```

```

        nothingtodo = false;
        goto endthisloop;
    }
    list<Node*>::iterator it2 = ((Or*)cur)->branches.begin();
    while(it2!=((Or*)cur)->branches.end())
    {
        if(typeid(*(*it2))==typeid(Or))
        {
            Or *subOr = (Or*)(*it2);
            ((Or*)cur)->branches.erase(it2);

            list<Node*>::iterator it3 = subOr->branches.begin();
            while(it3!=subOr->branches.end())
            {
                ((Or*)cur)->branches.push_back(*it3);
                it3++;
            }

            nothingtodo = false;
            goto endthisloop;
        }
        if(typeid(*(*it2))==typeid(Trivial))
        {
            it.replaceBy(new Clash());
            nothingtodo = false;
            goto endthisloop;
        }
        if(typeid(*(*it2))==typeid(Clash))
        {
            ((And*)cur)->branches.erase(it2);
            nothingtodo = false;
            goto endthisloop;
        }
        it2++;
    }
}
}
endthisloop::
}
while(!nothingtodo);
return ret;
}

/**
 * Application of rules R1 (basic unification rules)
 */
bool Unification::tryApplyR1()
{
    NodeIterator it(parser->result,&parser->result,NodeIterator::DEPTH_FIRST);
    Node *cur;

    // x=x (TODO : pour s=s) -> <triv>
    while((cur=it.next()))
        if(typeid(*cur)==typeid(Equal))
        {
            Equal *e = (Equal*)cur;

            /* if ((typeid(*e->n1)==typeid(Variable))&&
               (typeid(*e->n2)==typeid(Variable))&&
               (((Variable*)(e->n1))->id == ((Variable*)(e->n2))->id))
            */
            if(e->n1->isEqual(e->n2))
            {
                if(configuration->printRule)
                {
                    {
                        Display::texHspace(*mycout);
                        (*mycout) << "R1:  $x = x$ " << ((configuration->writeMode==Configuration::LATEX) ? "\\rightarrow" : ">") << endl;
                    }
                    /*
                     (*mycout) << "\t";
                     Display::texEndl(*mycout);
                     Display::texHspace(*mycout);
                     e->n1->print(*mycout);
                     Display::texEndl(*mycout);
                     (*mycout) << endl;
                     */
                }
                it.replaceBy(new Trivial());
                return true;
            }
        }
}

// s = x et s non var -> x = s
it.reset();
while((cur=it.next()))
    if(typeid(*cur)==typeid(Equal))
    {
        Equal *e = (Equal*)cur;

```

```

        if ((typeid(*e->n1) != typeid(Variable)) &&
            (typeid(*e->n2) == typeid(Variable)))
        {
            if (configuration->printRule)
            {
                Display::texHspace(*mycout);
                (*mycout) << "R1 $\cup$ :  $\cup s \cup = \cup x \cup$ " << ((configuration->writeMode == Configuration::LATEX)
                    ? "\\rightarrow" : "->") << " $\cup x \cup = \cup s$ " << endl;

                /*
                (*mycout) << "\t";
                Display::texEndl(*mycout);
                Display::texHspace(*mycout);
                e->n1->print((*mycout));
                (*mycout) << endl << "\t";
                Display::texEndl(*mycout);
                Display::texHspace(*mycout);
                e->n2->print((*mycout));
                (*mycout) << endl;
                Display::texEndl(*mycout); */
            }
            Node *swap = e->n1;
            e->n1 = e->n2;
            e->n2 = swap;
            return true;
        }
    }

// f(x)=f(y) -> x=y
it.reset();
while((cur=it.next()))
    if (typeid(*cur) == typeid(Equal))
    {
        Equal *e = (Equal*)cur;
        if ((typeid(*e->n1) == typeid(Function)) &&
            (typeid(*e->n2) == typeid(Function)) &&
            (((Function*)(e->n1))->id == ((Function*)(e->n2))->id))
        {
            if (configuration->printRule)
            {
                Display::texHspace(*mycout);
                (*mycout) << "R1 $\cup$ :  $\cup f(x) = \cup f(y) \cup$ " << ((configuration->writeMode == Configuration::
                    LATEX) ? "\\rightarrow" : "->") << " $\cup x = \cup y$ " << endl;

                /*
                (*mycout) << "\t";
                Display::texEndl(*mycout);
                Display::texHspace(*mycout);
                e->n1->print((*mycout));
                (*mycout) << endl << "\t";
                Display::texEndl(*mycout);
                Display::texHspace(*mycout);
                e->n2->print((*mycout));
                (*mycout) << endl;
                Display::texEndl(*mycout); */
            }
            And *add = new And();

            vector<Node*>::iterator it1 = ((Function*)(e->n1))->arguments.begin();
            vector<Node*>::iterator it2 = ((Function*)(e->n2))->arguments.begin();
            while(it1 != ((Function*)(e->n1))->arguments.end())
            {
                add->addBranch(new Equal(*it1, *it2));
                it1++;
                it2++;
            }

            it.replaceBy(add);
            return true;
        }
    }

// f(x)=g(y) -> <clash>
it.reset();
while((cur=it.next()))
    if (typeid(*cur) == typeid(Equal))
    {
        Equal *e = (Equal*)cur;
        if ((typeid(*e->n1) == typeid(Function)) &&
            (typeid(*e->n2) == typeid(Function)) &&
            (((Function*)(e->n1))->id != ((Function*)(e->n2))->id))
        {
            if (configuration->printRule)
            {
                Display::texHspace(*mycout);
                (*mycout) << "R1 $\cup$ :  $\cup f(x) = \cup g(y) \cup$ " << ((configuration->writeMode == Configuration::
                    LATEX) ? "\\rightarrow \cup \bot" : "-><clash>") << endl;

                /*

```

```

        (*mycout) << "\t";
        Display::texEndl(*mycout);
        Display::texHspace(*mycout);
        e->n1->print((*mycout));
        (*mycout) << endl<< "\t";
        Display::texEndl(*mycout);
        Display::texHspace(*mycout);
        e->n2->print((*mycout));
        Display::texEndl(*mycout);
        (*mycout) << endl;*/
    }
    it.replaceBy(new Clash());
    return true;
}

}

// x=s^P et x !E s et x E P et (si s est une var => s E P) -> x=s^P{x->s}
it.reset();
while((cur=it.next()))
{
    if(typeid(*cur)==typeid(And))
    {
        And *andd = (And*)cur;

        list<Node*>::iterator it1 = andd->branches.begin();
        while(it1!=andd->branches.end())
        {
            list<Node*>::iterator it2 = andd->branches.begin();
            while(it2!=andd->branches.end())
            {
                if((*it1)!=(*it2))
                {
                    if((typeid(*it1)==typeid(Equal))&&
                       (typeid(*((Equal*)(*it1))->n1)==typeid(Variable)))
                    {
                        Variable *x = (Variable*)(*(Equal*)(*it1))->n1;
                        Node *s = ((Equal*)(*it1))->n2;
                        Node *p = *it2;
                        if(!s->containVar(x->id)&&
                           (p->containVar(x->id)&&
                            (typeid(*s)!=typeid(Variable)) || (p->containVar(((Variable*)(s))->id))))
                        {
                            if(configuration->printRule)
                            {
                                Display::texHspace(*mycout);
                                (*mycout) << "R1:  $x=s^P$ " << ((configuration->writeMode==
                                    Configuration::LATEX)? "\\rightarrow" : "->") << " $x=s^P\{x->s\}$ "
                                << endl;
                                Display::texEndl(*mycout);
                            }
                            p->replace(x->id,s);
                            return true;
                        }
                    }
                }
                it2++;
            }
            it1++;
        }
    }
}

// x=s et x E s -> <clash>
it.reset();
while((cur=it.next()))
{
    if(typeid(*cur)==typeid(Equal))
    {
        Equal *e = (Equal*)cur;
        if((typeid(*e->n1)==typeid(Variable))&&
            (e->n2->containVar(((Variable*)(e->n1))->id)))
        {
            if(configuration->printRule)
            {
                Display::texHspace(*mycout);
                (*mycout) << "R1:  $x=s \text{ et } x \text{ } \bot$ " << ((configuration->writeMode==Configuration::
                    LATEX)? "\\in_{s \text{ } \bot} \rightarrow_{\text{bot}}" : " $E_{s \text{ } \rightarrow \text{clash}}$ ") << endl;
                /*
                (*mycout) << "\t";
                Display::texEndl(*mycout);
                Display::texHspace(*mycout);
                e->n1->print((*mycout));
                (*mycout) << endl<< "\t";
                Display::texEndl(*mycout);
                Display::texHspace(*mycout);
                e->n2->print((*mycout));
                (*mycout) << endl;
                Display::texEndl(*mycout);*/
            }
            it.replaceBy(new Clash());
        }
    }
}

```

```

        return true;
    }
    return false;
}

/**
 * Application of rules R2 (Unfold 1)
 */
bool Unification::tryApplyR2()
{
    NodeIterator it(parser->result,&parser->result,NodeIterator::DEPTH_FIRST);
    Node *cur;
    // s = t[@]^N_p.u -> ...
    while((cur=it.next()))
        if(typeid(*cur)==typeid(Equal))
        {
            Equal *e = (Equal*)cur;
            Node *s;
            Node *other;
            int way = 0;
            if((typeid(*(other=e->n2))==typeid(Expo))&&(typeid(*e->n1)!=typeid(Variable)))
                s=e->n1;
            else if((typeid(*(other=e->n1))==typeid(Expo))&&(typeid(*e->n2)!=typeid(Variable)))
            {
                s=e->n2;
                way = 1;
            }
            else continue;

rerun::
            Node *curSearchExpo = s;
            list<unsigned int>::iterator it2 = other->holePosition->begin();
            bool fail = false;

            if(typeid(*s)==typeid(Expo))
                fail = true;
            else
                while(it2!=other->holePosition->end())
                {
                    if(typeid(*curSearchExpo)!=typeid(Function))
                        break;
                    if(((Function*)curSearchExpo)->arguments.size()<=(it2))
                        break;
                    curSearchExpo = ((Function*)curSearchExpo)->arguments[*it2];
                    if(typeid(*curSearchExpo)==typeid(Expo))
                    {
                        fail = true;
                        break;
                    }
                    it2++;
                }

            if(fail)
            {
                if(way==0)
                    if((typeid(*(other=e->n1))==typeid(Expo))&&(typeid(*e->n2)!=typeid(Variable)))
                    {
                        s=e->n2;
                        way = 1;
                        goto rerun;
                    }
                continue;
            }

            if(configuration->printRule)
            {
                Display::texHspace(*mycout);
                (*mycout) << "R21(Unfold1)1:1s1=1t1[@]^Np.u1" << ((configuration->writeMode==
                    Configuration::LATEX)? "\\rightarrow" : "->") << "..." << endl;
                /*
                (*mycout) << "\t";
                Display::texEndl(*mycout);
                Display::texHspace(*mycout);
                cur->print((*mycout);
                Display::texEndl(*mycout);
                (*mycout) << endl;
                */
            }

            And *add2 = NULL;
            And *add3 = NULL;
            int num = (configuration->NcanbeZero)?0:1;
            Expo *expoterm = (Expo*)other;

            if((expoterm->free)|| (expoterm->expNum==num))
            {
                add2 = new And();
                if(expoterm->free)

```

```

        add2->addCondition(new CondA(expoterm->var->duplicate(), num));

        if(configuration->NcanbeZero)
            add2->addBranch(new Equal(s->duplicate(), expoterm->n1->duplicate()));
        else
            add2->addBranch(new Equal(s->duplicate(), expoterm->buildOne()));
    }

    if((expoterm->free) || (expoterm->expNum > 1))
    {
        add3 = new And();
        Expo *newExpo = (Expo *)expoterm->duplicate();

        if(newExpo->free)
        {
            NumericalVar *var = new NumericalVar(parser->genNumericalValName().c_str(), parser)
            ;
            add3->addCondition(new CondANplusB(expoterm->var->duplicate(), 1, var, 1));
            delete newExpo->var;
            newExpo->var = var->duplicate();
        }
        else
            newExpo->expNum--;

        add3->addBranch(new Equal(s->duplicate(), expoterm->buildOne(newExpo)));
    }

    if((add2)&&(!add3))
        it.replaceBy(add2);
    else if((add3)&&(!add2))
        it.replaceBy(add3);
    else
    {
        Or *add = new Or();
        add->addBranch(add2);
        add->addBranch(add3);
        it.replaceBy(add);
    }
    return true;
}

return false;
}

int Unification::gcd(unsigned int a, unsigned int b)
{
    return ( b != 0 ? gcd(b, a % b) : a );
}

/**
 * is l1 a prefix of l2 ?
 */
bool Unification::prefix(const list<unsigned int> &l1, const list<unsigned int> &l2)
{
    if(l1.size() > l2.size())
        return false;

    list<unsigned int>::const_iterator itp = l1.begin();
    list<unsigned int>::const_iterator itq2 = l2.begin();

    while(itp != l1.end())
    {
        if((*itp) != (*itq2))
            return false;
        itq2++;
        itp++;
    }
    return true;
}

/**
 * merging of l1 and l2
 */
list<unsigned int> Unification::append(const list<unsigned int> &l1, const list<unsigned int> &l2)
{
    list<unsigned int> ret = l1;
    list<unsigned int>::const_iterator it = l2.begin();
    while(it != l2.end())
    {
        ret.push_back(*it);
        it++;
    }
    return ret;
}

/**
 * Application of rules R3 (Unfold 2)

```



```

*/
bool Unification::tryApplyR3()
{
    NodeIterator it(parser->result,&parser->result,NodeIterator::DEPTH_FIRST);
    Node *cur;
    // s = t[@]^N.p.u -> ...
    while((cur=it.next()))
        if(typeid(*cur)==typeid(Equal))
        {
            int way = 0;
            while(way<2)
            {
                way++;

                Equal *e = (Equal*)cur;
                Node *u;
                Node *v;

                if((typeid(*(v=e->n2))==typeid(Expo))&&((u=e->n1)->rhoterm));
                else if((typeid(*(v=e->n1))==typeid(Expo))&&((u=e->n2)->rhoterm));
                else continue;

                NodeIterator it2(u,NULL,NodeIterator::DEPTH_FIRST);
                Node *cur2;
                while((cur2=it2.next()))
                    if(typeid(*cur2)==typeid(Expo))
                    {
                        const list<unsigned int> p = it2.getPosition();

                        assert(((Expo*)cur2)->holePosition);
                        const list<unsigned int> &q1 = *(((Expo*)cur2)->holePosition);
                        const list<unsigned int> &q2 = *v->holePosition;

                        if((q1.size()!=q2.size())&&(prefix(p,q2)))
                        {
                            int i;
                            int d = gcd((unsigned int)q1.size(),(unsigned int)q2.size());
                            int alpha1 = (unsigned int)q1.size() / d;
                            int alpha2 = (unsigned int)q2.size() / d;
                            //int m = alpha1 * alpha2 * d;

                            if(configuration->printRule)
                            {
                                {
                                    Display::texHspace(*mycout);
                                    (*mycout) << "R3U(UnfoldU2)U:Us[t1[@]^N1_q1.u1]_pU=Ut2[@]^N2_q2.u2U" << ((
                                        configuration->writeMode==Configuration::LATEX)?"\\rightarrow":"->")
                                        << "u..." << endl;
                                }
                                /*
                                (*mycout) << "\t";
                                Display::texEndl(*mycout);
                                Display::texHspace(*mycout);
                                cur->print((*mycout));
                                Display::texEndl(*mycout);
                                (*mycout) << endl;
                                */
                                // Debug
                                /*
                                (*mycout) << "p:" << p << endl;
                                (*mycout) << "q1:" << q1 << endl;
                                (*mycout) << "q2:" << q2 << endl;
                                (*mycout) << "d:" << d << endl;
                                (*mycout) << endl;
                                */
                            }

                            Or *addOr = new Or();

                            int first = (this->configuration->NcanbeZero)?0:1;

                            int r1,r2;
                            for(r1=first;r1<alpha2;r1++)
                            {
                                if(((Expo*)cur2)->free)
                                {
                                    And *addAnd = new And();
                                    addAnd->addCondition(new CondA(((Expo*)cur2)->var->duplicate(),r1));
                                    Node *newTerm = u->duplicate();

                                    Node *curModificationExpNum = newTerm;
                                    list<unsigned int>::const_iterator it3 = p.begin();
                                    while(it3!=p.end())
                                    {
                                        assert(typeid(*curModificationExpNum)==typeid(Function));
                                        curModificationExpNum = ((Function*)curModificationExpNum)->
                                            arguments[*it3];
                                        it3++;
                                    }
                                }
                            }
                        }
                    }
            }
        }
    Or *addOr = new Or();

    int first = (this->configuration->NcanbeZero)?0:1;

    int r1,r2;
    for(r1=first;r1<alpha2;r1++)
    {
        if(((Expo*)cur2)->free)
        {
            And *addAnd = new And();
            addAnd->addCondition(new CondA(((Expo*)cur2)->var->duplicate(),r1));
            Node *newTerm = u->duplicate();

            Node *curModificationExpNum = newTerm;
            list<unsigned int>::const_iterator it3 = p.begin();
            while(it3!=p.end())
            {
                assert(typeid(*curModificationExpNum)==typeid(Function));
                curModificationExpNum = ((Function*)curModificationExpNum)->
                    arguments[*it3];
                it3++;
            }
        }
    }
}

```

```

        assert (typeid(*curModificationExpNum)==typeid(Expo));
        if(((Expo*)curModificationExpNum)->free)
        {
            delete ((Expo*)curModificationExpNum)->var;
            ((Expo*)curModificationExpNum)->free = false;
        }
        ((Expo*)curModificationExpNum)->expNum = r1;

        addAnd->addBranch(new Equal(newTerm,v->duplicate()));
        addOr->addBranch(addAnd);
    }
    else if (((Expo*)cur2)->expNum==r1)
    {
        And *addAnd = new And();
        addAnd->addBranch(new Equal(u->duplicate(),v->duplicate()));
        addOr->addBranch(addAnd);
    }
}

for(r2=first;r2<alpha1;r2++)
{
    if(((Expo*)v)->free)
    {
        And *addAnd = new And();
        addAnd->addCondition(new CondA(((Expo*)v)->var->duplicate(),r2));

        Node *newTerm = v->duplicate();
        if(((Expo*)newTerm)->free)
        {
            delete ((Expo*)newTerm)->var;
            ((Expo*)newTerm)->free = false;
        }
        ((Expo*)newTerm)->expNum = r2;

        addAnd->addBranch(new Equal(u->duplicate(),newTerm));
        addOr->addBranch(addAnd);
    }
    else if (((Expo*)v)->expNum==r2)
    {
        And *addAnd = new And();
        addAnd->addBranch(new Equal(u->duplicate(),v->duplicate()));
        addOr->addBranch(addAnd);
    }
}

for(r2=0;r2<alpha1;r2++)
for(r1=0;r1<alpha2;r1++)
{
    And *addAnd = new And();
    bool f1 = ((Expo*)cur2)->free;
    bool f2 = ((Expo*)v)->free;

    NumericalVar *m1;
    NumericalVar *m2;

    int mip;
    int m2p;

    if(f1)
    {
        m1 = new NumericalVar(parser->genNumericalValName().c_str(),parser);
        addAnd->addCondition(new CondANplusB(((Expo*)cur2)->var->duplicate(),
            alpha2,m1,r1));
    }
    else
    {
        mip = (((Expo*)cur2)->expNum - r1);
        if((mip%alpha2) != 0)
        {
            delete addAnd;
            continue;
        }
        mip /= alpha2;
        if(mip<first)
        {
            delete addAnd;
            continue;
        }
    }
}

if(f2)
{
    m2 = new NumericalVar(parser->genNumericalValName().c_str(),parser);
    addAnd->addCondition(new CondANplusB(((Expo*)v)->var->duplicate(),
        alpha1,m2,r2));
}

```

```

    }
    else
    {
        m2p = (((Expo*)v)->expNum - r2) / alpha1;
        if((m2p%alpha1) != 0)
        {
            if(f1)
            {
                delete m1;
                delete addAnd;
                continue;
            }
            m2p /= alpha1;
            if(m2p<first)
            {
                if(f1)
                {
                    delete m1;
                    delete addAnd;
                    continue;
                }
            }
        }
    }

    Node *t1 = ((Expo*)cur2)->n2->duplicate();
    for(i=0;i<r1;i++)
        t1 = ((Expo*)cur2)->buildOne(t1);

    Node *t2 = ((Expo*)v)->n2->duplicate();
    for(i=0;i<r2;i++)
        t2 = ((Expo*)v)->buildOne(t2);

    Node *h1 = ((Expo*)cur2)->n1->duplicate();
    for(i=0;i<alpha2-1;i++)
        h1 = ((Expo*)cur2)->buildOne(h1);

    Node *h2 = ((Expo*)v)->n1->duplicate();
    for(i=0;i<alpha1-1;i++)
        h2 = ((Expo*)v)->buildOne(h2);

    list<unsigned int> tmp = p;

    Node *k;
    if(f1)
        k = ((Function*)u)->buildOneAndReplace(tmp, new Expo(h1,m1,t1));
    else
        k = ((Function*)u)->buildOneAndReplace(tmp, new Expo(h1,m1p,t1));

    if(f2)
        addAnd->addBranch(new Equal(k,new Expo(h2,m2,t2)));
    else
        addAnd->addBranch(new Equal(k,new Expo(h2,m2p,t2)));

    addOr->addBranch(addAnd);
}
it.replaceBy(addOr);
return true;
}
}
}
}
return false;
}

/**
 * Application of rules R4 (Unfold 3)
 */
bool Unification::tryApplyR4()
{
    NodeIterator it(parser->result,&parser->result,NodeIterator::DEPTH_FIRST);
    Node *cur;

    while((cur=it.next()))
        if(typeid(*cur)==typeid(Equal))
        {
            int way = 0;
            while(way<2)
            {
                way++;

                Equal *e = (Equal*)cur;
                Node *u;
                Node *v;

                if((typeid(*(v=e->n2))==typeid(Expo))&&((u=e->n1)->rhoterm));
                else if((typeid(*(v=e->n1))==typeid(Expo))&&((u=e->n2)->rhoterm));
                else continue;

                NodeIterator it2(u,NULL,NodeIterator::DEPTH_FIRST);
                Node *cur2;
                while((cur2=it2.next()))

```

```

if (typeid(*cur2) == typeid(Expo))
{
    const list<unsigned int> p = it2.getPosition();
    const list<unsigned int> &q1 = *((Expo*)cur2)->holePosition;
    const list<unsigned int> &q2 = *v->holePosition;

    if ((prefix(p, q2)) && (!prefix(q2, append(p, q1))) || (!prefix(append(p, q1), append(q2, q2))))
    {
        if (configuration->printRule)
        {
            Display::texHspace(*mycout);
            (*mycout) << "R4U(UnfoldU3)U:Us[t1[0]N1q1.u1]UpU=Ut2[0]N2q2.u2U" << ((
                configuration->writeMode == Configuration::LATEX) ? "\\rightarrow" : "->")
                << "U..." << endl;

            /*
            (*mycout) << "\t";
            Display::texEndl(*mycout);
            Display::texHspace(*mycout);
            cur->print(*mycout);
            Display::texEndl(*mycout);
            (*mycout) << endl;
            */

            // Debug
            /*
            (*mycout) << "p:" << p << endl;
            (*mycout) << "q1:" << q1 << endl;
            (*mycout) << "q2:" << q2 << endl;
            (*mycout) << endl;
            */
        }

        int first = (configuration->NcanbeZero) ? 0 : 1;
        assert(!configuration->NcanbeZero); // pas encore fait

        Or *addOr = new Or();

        if (((Expo*)v)->free)
        {
            And *addAnd = new And();
            addAnd->addCondition(new CondA(((Expo*)v)->var->duplicate(), 1));

            addAnd->addBranch(new Equal(((Expo*)v)->buildOne(), ((Expo*)u)->duplicate()));
            addOr->addBranch(addAnd);
        }
        else
        {
            if (((Expo*)v)->expNum == 1)
            {
                And *addAnd = new And();
                addAnd->addBranch(new Equal(((Expo*)v)->buildOne(), ((Expo*)u)->duplicate()));
                addOr->addBranch(addAnd);
            }
        }

        if (((Expo*)cur2)->free)
        {
            And *addAnd = new And();
            addAnd->addCondition(new CondA(((Expo*)cur2)->var->duplicate(), 1));

            assert(typeid(*u) == typeid(Function));

            list<unsigned int> tmp = p;
            addAnd->addBranch(new Equal(((Function*)u)->buildOneAndReplace(tmp, ((Expo*)cur2)->buildOne()), ((Expo*)v)->duplicate()));
            addOr->addBranch(addAnd);
        }
        else
        {
            if (((Expo*)cur2)->expNum == 1)
            {
                And *addAnd = new And();
                list<unsigned int> tmp = p;
                addAnd->addBranch(new Equal(((Function*)u)->buildOneAndReplace(tmp, ((Expo*)cur2)->buildOne()), ((Expo*)v)->duplicate()));
                addOr->addBranch(addAnd);
            }
        }

        if (((Expo*)v)->free)
        {
            And *addAnd = new And();
            addAnd->addCondition(new CondA(((Expo*)v)->var->duplicate(), 2));
        }
    }
}

```

```

        addAnd->addBranch(new Equal(((Expo*)v)->buildTwo(),((Expo*)u)->duplicate()
));
        addOr->addBranch(addAnd);
    }
    else
    {
        if(((Expo*)v)->expNum==2)
        {
            And *addAnd = new And();
            addAnd->addBranch(new Equal(((Expo*)v)->buildOne(),((Expo*)u)->
                duplicate()));
            addOr->addBranch(addAnd);
        }
    }

    bool f1 = ((Expo*)cur2)->free;
    bool f2 = ((Expo*)v)->free;

    And *addAnd = new And();

    NumericalVar *m1 = NULL;
    int m1i;
    if(f1)
    {
        m1 = new NumericalVar(parser->genNumericalValName().c_str(),parser);
        addAnd->addCondition(new CondANplusB(((Expo*)cur2)->var->duplicate(),1,m1
            ,1));
    }
    else
    {
        m1i = ((Expo*)cur2)->expNum-1;
        if(m1i<first)
        {
            delete addAnd;
            continue;
        }
    }

    NumericalVar *m2;
    int m2i;
    if(f2)
    {
        m2 = new NumericalVar(parser->genNumericalValName().c_str(),parser);
        addAnd->addCondition(new CondANplusB(((Expo*)v)->var->duplicate(),1,m2,2))
            ;
    }
    else
    {
        m2i = ((Expo*)v)->expNum-2;
        if(m2i<first)
        {
            if(f1)
            {
                delete addAnd->conditions.front();
                assert(m1);
                delete m1;
            }
            delete addAnd;
            continue;
        }
    }

    Expo *h1 = (Expo*)v->duplicate();
    if(f2)
    {
        delete h1->var;
        h1->var = m2;
    }
    else
    {
        h1->expNum = m2i;
    }

    Expo *h2 = (Expo*)cur2->duplicate();
    if(f1)
    {
        delete h2->var;
        h2->var = m1;
    }
    else
    {
        h2->expNum = m1i;
    }

    list<unsigned int> tmp = p;
    addAnd->addBranch(new Equal(((Function*)u)->buildOneAndReplace(tmp,((Expo*)
        cur2)->buildOne(h2)),((Expo*)v)->buildTwo(h1)));
    addOr->addBranch(addAnd);

    if(addOr->branches.empty())
    {

```

```

        delete addOr;
        continue;
    }

    it.replaceBy(addOr);
    return true;
}

    }
}

return false;
}

/**
 * Application of rules R5 (Decompose 2)
 */
bool Unification::tryApplyR5()
{
    NodeIterator it(parser->result,&parser->result,NodeIterator::DEPTH_FIRST);
    Node *cur;

    while((cur=it.next()))
        if(typeid(*cur)==typeid(Equal))
        {
            Equal *e = (Equal*)cur;
            Node *u;
            Node *v;

            int way = 0;
            if((typeid(*(v=e->n2))==typeid(Expo))&&((u=e->n1)->rhoterm))
                ;
            else if((typeid(*(v=e->n1))==typeid(Expo))&&((u=e->n2)->rhoterm))
                way = 1;
            else continue;

rerun::

            NodeIterator it2(u,NULL,NodeIterator::DEPTH_FIRST);
            Node *cur2;
            while((cur2=it2.next()))
                if(typeid(*cur2)==typeid(Expo))
                {
                    const list<unsigned int> p = it2.getPosition();
                    const list<unsigned int> &q1 = *((Expo*)cur2)->holePosition;
                    //const list<unsigned int> &q2 = *v->holePosition;

                    if(configuration->printRule)
                    {
                        Display::texHspace(*mycout);
                        (*mycout) << "R5u(Decomposeu2)u:us[v1[w1[0]p]N1q.u1pu(v2[w2[0]q]p)N2.u2u
                        " << ((configuration->writeMode==Configuration::LATEX)?"\\rightarrow":"->") <<
                        "u..." << endl;
                        /*(*mycout) << "\t";
                        Display::texEndl(*mycout);
                        Display::texHspace(*mycout);
                        cur->print((*mycout);
                        Display::texEndl(*mycout);
                        (*mycout) << endl;
                        */
                        // Debug
                        /*
                        (*mycout) << "p:" << p << endl;
                        (*mycout) << "q1:" << q1 << endl;
                        (*mycout) << "q2:" << q2 << endl;
                        (*mycout) << endl;
                        */
                    }

                    list<unsigned int> qprim;
                    list<unsigned int>::const_iterator it3 = q1.begin();
                    int num = q1.size() - p.size();
                    for(;num>0;num--)
                    {
                        qprim.push_back(*it3);
                        it3++;
                    }

                    And *addAnd = new And();

                    Function *addA = new Function(parser->genContstantValName().c_str(),parser);

                    list<unsigned int>tmp1;
                    list<unsigned int>tmp2;
                    list<unsigned int>tmp3;

                    Node *w1 = (((Expo*)cur2)->n1)->find(tmp1=qprim);
                    Node *w2 = (((Expo*)v)->n1)->find(tmp1=p);

                    addAnd->addBranch(new Equal(

```

```

        u->buildOneAndReplace(tmp1=p, addA->duplicate()),
        w1->buildOneAndReplace(tmp3=p, addA->duplicate())
    ));

addAnd->addBranch(new Equal(
    u->buildOneAndReplace(tmp1=p, addA->duplicate()),
    ((Expo*)v)->n1->buildOneAndReplace(tmp2=p, addA->duplicate())
));

addAnd->addBranch(new Equal(
    ((Expo*)cur2)->n1->buildOneAndReplace(tmp1=qprim, addA->duplicate()),
    w2->buildOneAndReplace(tmp2=qprim, addA->duplicate())
));

Or *addOr = new Or();

bool buildit = true;

And *addAnd2 = new And();
if (((((Expo*)cur2)->free) && !((Expo*)v)->free))
    {
        if (((Expo*)cur2)->expNum == ((Expo*)v)->expNum)
            ;
        else
            buildit = false;
    }
else
{
    if (((((Expo*)cur2)->free) && !((Expo*)v)->free))
        addAnd2->addCondition(new CondN(((Expo*)cur2)->var->duplicate(), ((Expo*)v)->
            var->duplicate()));
    else if (!((Expo*)cur2)->free)
        addAnd2->addCondition(new CondA(((Expo*)v)->var->duplicate(), ((Expo*)cur2)->
            expNum));
    else
        addAnd2->addCondition(new CondA(((Expo*)cur2)->var->duplicate(), ((Expo*)v)->
            expNum));
}

if (buildit)
{
    addAnd2->addBranch(new Equal(u->buildOneAndReplace(tmp1=p, ((Expo*)cur2)->n2->
        duplicate()), ((Expo*)v)->n2->duplicate()));
    addOr->addBranch(addAnd2);
}
else
    delete addAnd2;

int first = (this->configuration->NcanbeZero)?0:1;

addAnd2 = new And();
int m1i;
NumericalVar *m1 = NULL;
if (((((Expo*)cur2)->free) && !((Expo*)v)->free))
    {
        m1i = ((Expo*)cur2)->expNum - ((Expo*)v)->expNum;
        if (m1i >= first)
            ;
        else
            buildit = false;
    }
else
{
    m1 = new NumericalVar(parser->genNumericalValName().c_str(), parser);

    if (((((Expo*)cur2)->free) && !((Expo*)v)->free))
        addAnd2->addCondition(new CondNplusN(((Expo*)cur2)->var->duplicate(), ((Expo*)v)
            ->var->duplicate(), m1->duplicate()));
    else if (!((Expo*)cur2)->free)
        addAnd2->addCondition(new CondANplusB(((Expo*)v)->var->duplicate(), -1, m1->
            duplicate(), ((Expo*)cur2)->expNum));
    else
        addAnd2->addCondition(new CondANplusB(((Expo*)cur2)->var->duplicate(), 1, m1->
            duplicate(), ((Expo*)v)->expNum));
}

if (buildit)
{
    Node *h1 = u->duplicate();

    Expo *coup = (Expo*)h1->find(tmp1=p);
    assert(typeid(*coup) == typeid(Expo));

    coup->free = (m1 != NULL);
    if (m1)
        coup->var = m1;
    else
        coup->expNum = m1i;
}

```

```

        addAnd2->addBranch(new Equal(h1,((Expo*)v)->n2));
        addOr->addBranch(addAnd2);
    }
    else
        delete addAnd2;

    addAnd2 = new And();
    m1 = NULL;
    if(!((Expo*)cur2)->free)&&!((Expo*)v)->free))
    {
        m1i = ((Expo*)v)->expNum - ((Expo*)cur2)->expNum;
        if(m1i>=first)
        ;
        else
            buildit= false;
    }
    else
    {
        m1 = new NumericalVar(parser->genNumericalValName().c_str(),parser);

        if((((Expo*)v)->free)&&(((Expo*)cur2)->free))
            addAnd2->addCondition(new CondNplusN(((Expo*)v)->var->duplicate(),((Expo*)cur2)->var->duplicate()),m1->duplicate());
        else if(!((Expo*)cur2)->free)
            addAnd2->addCondition(new CondANplusB(((Expo*)cur2)->var->duplicate(),-1,m1->duplicate()),((Expo*)v)->expNum));
        else
            addAnd2->addCondition(new CondANplusB(((Expo*)v)->var->duplicate(),1,m1->duplicate()),((Expo*)cur2)->expNum));
    }

    if(buildit)
    {
        Expo *h1 = (Expo*)v->duplicate();

        h1->free = (m1!=NULL);
        if(m1)
            h1->var = m1;
        else
            h1->expNum = m1i;

        addAnd2->addBranch(new Equal(w1->buildOneAndReplace(tmp1=p,((Expo*)cur2)->n2->duplicate()),h1));
        addOr->addBranch(addAnd2);
    }
    else
        delete addAnd2;

    if(addOr->branchs.empty())
        delete addOr;
    else
        addAnd->addBranch(addOr);

    //it.replaceBy(SetDisjunctiveSystem((Node*)addAnd));
    it.replaceBy((Node*)addAnd);
    return true;
}

if(way==0)
if((typeid(*(v=e->n1))==typeid(Expo))&&((u=e->n2)->rhoterm))
{
    way = 1;
    goto rerun;
}

}

return false;
}

/**
 * Explore and check definition (is the current system well define variables)
 */
bool Unification::exploDefinition(Node *n,SimpleSolution *solution)
{
    bool val = true;

    if(typeid(*n)==typeid(Clash))
        val = false;
    else if(typeid(*n)==typeid(Trivial))
        val = true;
    else if(typeid(*n)==typeid(Equal))
    {
        Equal *e = (Equal*)n;
        val = true;
        if(typeid(*e->n1)==typeid(Variable))
            solution->variableSolution[(((Variable*)(e->n1))->id).definition = e->n2;
        else

```



```

        return false;
    if(!exploDefinition(e->n2,solution))
        val = false;
    }
    else if(typeid(*n)==typeid(And))
    {
        And *a = (And*)n;
        val = true;
        list<Node*>::iterator it = a->branches.begin();
        while(it!=a->branches.end())
        {
            if(!exploDefinition(*it,solution))
            {
                val = false;
                break;
            }
            it++;
        }
        list<Condition*>::iterator it2 = a->conditions.begin();
        while(it2!=a->conditions.end())
        {
            /*
            // multi definition
            if(solution->numericalvariableSolution[( *it2)->n1->id].definitionc)
            {
                val = false;
                break;
            }
            else
            */

            solution->numericalvariableSolution[( *it2)->n1->id].definitionc.push_back(*it2);

            if(typeid(**it2)==typeid(CondANplusB))
                solution->numericalvariableSolution[(( CondANplusB*)(*it2))->n2->id].seeoutside = true;
            else if(typeid(**it2)==typeid(CondNplusN))
            {
                solution->numericalvariableSolution[(( CondNplusN*)(*it2))->n2->id].seeoutside = true;
                solution->numericalvariableSolution[(( CondNplusN*)(*it2))->n3->id].seeoutside = true;
            }
            else if(typeid(**it2)==typeid(CondN))
                solution->numericalvariableSolution[(( CondN*)(*it2))->n2->id].seeoutside = true;

            it2++;
        }
    }
    else if(typeid(*n)==typeid(Function))
    {
        Function *f = (Function*)n;
        vector<Node*>::iterator it = f->arguments.begin();
        val = true;
        while(it!=f->arguments.end())
        {
            if(!exploDefinition(*it,solution))
            {
                val = false;
                break;
            }
            it++;
        }
    }
    else if(typeid(*n)==typeid(Or))
    {
        assert(false); // Ceci n'est pas une forme disjonctive (les Ors ont dû être traités)
    }
    else if(typeid(*n)==typeid(Expo))
    {
        Expo *e = (Expo*)n;
        val = exploDefinition(e->n1,solution);
        if(!exploDefinition(e->n2,solution))
            val = false;
        if(e->free)
            solution->numericalvariableSolution[e->var->id].seeoutside = true;
    }
    else if(typeid(*n)==typeid(Variable))
    {
        Variable *v = (Variable*)n;
        solution->variableSolution[v->id].seeoutside = true;
        val = true;
    }
    return val;
}

/**
 * A function that verifies if a given solution is a real solution,
 * ie. can be satisfied. We then list the solutions.

```

```

* If not, THIS is not a solution but there may be another one that is valid.
*/
bool Unification::verifValidPresburger(Node* n)
{
#ifdef DEBUG
    (*mycout) << "-----" << endl;
#endif
    if (typeid(*n) == typeid(And)) { // test if there are (is a) Numerical Value Condition
        list<Automata*> automL;
        std::map<int,int> numvars; // trick to count without duplicates
        // OU UN TABLEAU
        list<Condition*>::iterator it = ((And*)n)->conditions.begin();
        while (it != ((And*)n)->conditions.end()) {
            Condition* tmpc = *it;
            if (numvars.find(tmpc->n1->id) == numvars.end())
                numvars[tmpc->n1->id] = numvars.size();
            if (typeid(*tmpc) == typeid(CondN)) {
                if (numvars.find(((CondN*)tmpc)->n2->id) == numvars.end())
                    numvars[(((CondN*)tmpc)->n2->id)] = numvars.size();
            } else if (typeid(*tmpc) == typeid(CondANplusB)) {
                if (numvars.find(((CondANplusB*)tmpc)->n2->id) == numvars.end())
                    numvars[(((CondANplusB*)tmpc)->n2->id)] = numvars.size();
            } else if (typeid(*tmpc) == typeid(CondNplusN)) {
                if (numvars.find(((CondNplusN*)tmpc)->n2->id) == numvars.end())
                    numvars[(((CondNplusN*)tmpc)->n2->id)] = numvars.size();
                if (numvars.find(((CondNplusN*)tmpc)->n3->id) == numvars.end())
                    numvars[(((CondNplusN*)tmpc)->n3->id)] = numvars.size();
            } // only CondN left and it is already taken into account
            it++;
        }
        /* vector<int> numvars; // trick to count without duplicates
        // OU UN TABLEAU
        list<Condition*>::iterator it = ((And*)n)->conditions.begin();
        while (it != ((And*)n)->conditions.end()) {
            Condition* tmpc = *it;
            if (numvars[tmpc->n1->id] == numvars.end())
                numvars[tmpc->n1->id] = numvars.size();
            if (typeid(*tmpc) == typeid(CondN)) {
                if (numvars.find(((CondN*)tmpc)->n2->id) == numvars.end())
                    numvars[(((CondN*)tmpc)->n2->id)] = numvars.size();
            } else if (typeid(*tmpc) == typeid(CondANplusB)) {
                if (numvars.find(((CondANplusB*)tmpc)->n2->id) == numvars.end())
                    numvars[(((CondANplusB*)tmpc)->n2->id)] = numvars.size();
            } else if (typeid(*tmpc) == typeid(CondNplusN)) {
                if (numvars.find(((CondNplusN*)tmpc)->n2->id) == numvars.end())
                    numvars[(((CondNplusN*)tmpc)->n2->id)] = numvars.size();
                if (numvars.find(((CondNplusN*)tmpc)->n3->id) == numvars.end())
                    numvars[(((CondNplusN*)tmpc)->n3->id)] = numvars.size();
            } // only CondN left and it is already taken into account
            it++;
        }
        */

#ifdef DEBUG
        (*mycout) << "===>BITS_SIZE_<numvars.size()>_<numvars.size()>" << endl;
#endif

        it = ((And*)n)->conditions.begin();
        // build the automatatas and push them into a list
        while (it != ((And*)n)->conditions.end()) {
            Automata* tmp = new Automata(*it, &numvars);
#ifdef DEBUG
            tmp->display(*mycout);
#endif
            automL.push_back(tmp);
            it++;
        }

        // now merge the automatatas with the help of the bits map ;

        // builds automatates groupes

        list<Groupe> groupes;
        list<Automata*>::iterator it7 = automL.begin();
        while (it7 != automL.end())
        {
            Groupe tmp;
            tmp.autos.push_back(*it7);
            groupes.push_back(tmp);
            it7++;
        }

        list<Groupe>::iterator itg1, itg2;
        list<Automata*>::iterator ita1, ita2;

        bool todo;
        do
    
```

```

{
    todo = false;
    itg1 = groupes.begin();
    while(itg1!=groupes.end())
    {
        itg2 = itg1;
        if(itg2!=groupes.end())
            itg2++;
        while(itg2!=groupes.end())
        {
            ita1 = itg1->autos.begin();
            while(ita1!=itg1->autos.end())
            {
                ita2 = itg2->autos.begin();
                while(ita2!=itg2->autos.end())
                {
                    if((*ita1)->isIntersect(*ita2,numvars.size()))
                    {
                        todo = true;
                        (*itg1).autos.push_back(*ita2);
                        (*itg2).autos.erase(ita2);
                        if((*itg2).autos.empty())
                            groupes.erase(itg2);
                        goto groupeend;
                    }
                    ita2++;
                }
                ita1++;
            }
            itg2++;
        }
        itg1++;
    }
groupeend;
}
while(todo);
#ifdef DEBUG
    (*mycout) << "NDGroupes: " << groupes.size() << endl;
#endif

// tests automates groupes
itg1 = groupes.begin();
while(itg1!=groupes.end())
{
    ExploPara exploPara;
    StatePara para;
    bool isgooda = false;

    // Initial state
    list<Automata*>::iterator it2 = itg1->autos.begin();
    while(it2!=itg1->autos.end())
    {
        para.actives.push_back((*it2)->si);
        it2++;
    }
    exploPara.statePara.push_back(para);

    // While empty
    while(!exploPara.statePara.empty())
    {
        para = exploPara.statePara.front();
        exploPara.statePara.pop_front();

        bool allfinal = true;
        list<State*>::iterator it3 = para.actives.begin();
        while(it3!=para.actives.end())
        {
            if((*it3)->value != (*it3)->automata->cvalue)
            {
                allfinal = false;
                break;
            }
            it3++;
        }

        if(allfinal)
        {
            isgooda = true;
            goto gooda;
        }

        State* first = para.actives.front();

        if(para.actives.empty())
            continue;
        if(first->step.empty())
            continue;
    }
}

```

```

        list<Transition*>::iterator it4 = first->step.begin();
        while(it4!=first->step.end())
        {
            StatePara newPara;
            newPara.actives.push_back((*it4)->st);

            bool good2 = true;
            it3 = para.actives.begin();
            it3++;
            while(it3!=para.actives.end())
            {
                bool good = false;
                list<Transition*>::iterator it5 = (*it3)->step.begin();
                while(it5!=(*it3)->step.end())
                {
                    // chercher *it5 "=" *it4
                    // if(!memcmp((*it5)->bits,(*it4)->bits,sizeof(int)*numvars.size()))

                    unsigned int i;
                    bool err = false;
                    for(i=0;i<numvars.size();i++)
                    if((( (*it5)->bits[i]==-1) || (( (*it4)->bits[i]==-1) || (( (*it4)->bits[i]==(*it5)
                        ->bits[i]))
                        ;
                    else
                        err = true;

                    if(!err)
                    {
                        newPara.actives.push_back((*it5)->st);
                        good = true;
                        break;
                    }

                    it5++;
                }

                if(!good)
                {
                    good2 = false;
                    break;
                }
                it3++;
            }

            if(!good2)
                break;

            exploPara.statePara.push_back(newPara);
            it4++;
        }
    }

gooda::
    if(!isgooda)
        return false;
    itg1++;
}

    return true;
}
// trivial case : no Numerical Value Condition
return true;
}

/**
 * check all solutions (and delete the bad ones).
 */
bool Unification::checkDefinition()
{
    if(typeid(*parser->result)==typeid(Or)) // many different solutions that we split
    {
        list<Node*>::iterator it = ((Or*)(parser->result))->branches.begin();
        while(it!=((Or*)(parser->result))->branches.end())
        {
            SimpleSolution *sol = new SimpleSolution((unsigned int)parser->variabletab.size(),(
                unsigned int)parser->numericalvariabletab.size(),this);
            solutions.push_back(sol);
            sol->fail = !exploDefinition(*it,sol);
            if(configuration->presBurgerVerif)
            if(!sol->fail)
                sol->fail = !verifValidPresburger(*it);

            /*
            Solver::Graph *gr = new Solver::Graph(*it,this);
            solutions.push_back(gr->run());
            */
            it++;
        }
    }
}

```

```

    }
    else // only one solution so there is no need to split
    {
        SimpleSolution *sol = new SimpleSolution((unsigned int)parser->variabletab.size(),(unsigned
            int)parser->numericalvariabletab.size(),this);
        solutions.push_back(sol);
        sol->fail = !exploDefinition(parser->result,sol);
        if(configuration->presBurgerVerif)
        if(!sol->fail)
            sol->fail = !verifValidPresburger(parser->result);
        /*
        Solver::Graph *gr = new Solver::Graph(parser->result,this);
        solutions.push_back(gr->run());
        */
    }

    list<Solution*>::iterator it = solutions.begin();
    while(it!=solutions.end())
    {
        if((*it)->check())
            it++;
        else
            it = solutions.erase(it);
    }
    return !solutions.empty();
}

/**
 * display a solution
 */
void SimpleSolution::display(ostream &o)
{
    unsigned int i;
    for(i=0;i<ndnumvar;i++)
    {
        if(numericalvariableSolution[i].definitionc.empty())
        {
            if((numericalvariableSolution[i].seeoutside)||(!unification->parser->numericalvariabletab[
                i].automatic))
            {
                o << "␣" <<unification->parser->numericalvariabletab[i].name << "␣=␣*free*";
                o << endl;
                if(globconfig->writeMode==Configuration::LATEX)
                    o << endl;
            }
        }
        else
        {
            list<Condition*>::iterator it = numericalvariableSolution[i].definitionc.begin();
            while(it!=numericalvariableSolution[i].definitionc.end())
            {
                o << "␣";
                (*it++)->print(o);
                o << endl;
                if(globconfig->writeMode==Configuration::LATEX)
                    o << endl;
            }
        }
    }
    for(i=0;i<ndvar;i++)
    {
        o << "␣" <<unification->parser->variabletab[i].name << "␣=␣";
        if(!variableSolution[i].definition)
            o << "*free*";
        else
            variableSolution[i].definition->print(o);
        o << endl;
        if(globconfig->writeMode==Configuration::LATEX)
            o << endl;
    }
}

VariableSolution::VariableSolution()
{
    seeoutside = false;
    definition = NULL;
}

SimpleSolution::SimpleSolution(unsigned int ndvar,unsigned int ndnumvar,Unification *unification)
{
    this->unification = unification;
    fail = false;
    this->ndvar = ndvar;
    this->ndnumvar = ndnumvar;
    variableSolution = new VariableSolution[ndvar];
    numericalvariableSolution = new VariableSolution[ndnumvar];
}

```

```

bool SimpleSolution::check()
{
    if(fail)
        return false;
    return true;
}

SimpleSolution::~SimpleSolution()
{
    delete [] variableSolution;
    delete [] numericalvariableSolution;
}

/**
 * Run unification
 */
void Unification::run()
{
    result = true;
    int step = 0;
    long startTime = clock();

    Display::texDi(*mycout);
    (*mycout) << "Initial set of equation";
    Display::texDi(*mycout);
    Display::texEndl(*mycout);
    (*mycout) << endl;
    debugDisplay((*mycout));
    Display::texEndl(*mycout);
    (*mycout) << endl;

    while(true)
    {
        step++;
        if((configuration->pauseEveryStep) || (configuration->printRule))
        {
            Display::texEndl(*mycout);
            (*mycout) << endl;
            debugDisplay((*mycout));
            //Display::texEndl(*mycout);
            (*mycout) << endl;
        }

        if(configuration->pauseEveryStep)
        {
            (*mycout) << "=====PAUSE=( "<< step <<")===== " << endl;
            if(configuration->writeMode==Configuration::LATEX)
                (*mycout) << "\\\\";
            while(true)
            {
                (*mycout) << "command:"<<endl;
                (*mycout) << "\c-continue(default)" << endl;
                (*mycout) << "\f-finish" << endl;
                (*mycout) << "\q-quit" << endl;
                (*mycout) << "\i-info" << endl;

                char c = getc(stdin);
                fflush(stdin);

                if((c=='c') || (c==10))
                    break;
                else if(c=='f')
                {
                    configuration->pauseEveryStep = false;
                    break;
                }
                else if(c=='q')
                    exit(0);
                else if(c=='i')
                {
                    (*mycout) << endl;
                    (*mycout) << "\Number of steps: " << step << endl;
                    (*mycout) << "\Number of terms: " << Node::ndterms << endl;
                    (*mycout) << "\Number of variables: " << parser->variabletab.size() << endl;
                    (*mycout) << "\Number of functions: " << parser->fonctiontab.size() << endl;
                    (*mycout) << "\Number of numericals: " << parser->numericalvariabletab.size() << endl;
                    (*mycout) << endl;
                }
                else
                    printf("<Unknown command>\n");
            }
        }

        if(configuration->beep!=0)
            Display::beep(Node::ndterms*10,20);

        parser->checkSoundness();
    }
}

```

```

        if(!tryApplyR1())
        if(!tryApplyR2())
        if(!tryApplyR3())
        if(!tryApplyR4())
        if(!tryApplyR5())
            break;

        parser->checkSoundness();

        this->parser->result = ApplyR0();

        // plus des calculs
        //parser->result = SetDisjunctiveSystem();
    }

    this->parser->result = ApplyR0();

    if(configuration->disjonction)
        parser->result = SetDisjunctiveSystem();

    Display::texDi(*mycout);
    (*mycout) << "End of computation";
    Display::texDi(*mycout);
    Display::texEndl(*mycout);
    (*mycout) << endl << endl;

    (*mycout) << "Number of steps" << step << endl;

    if(globconfig->writeMode==Configuration::LATEX)
        (*mycout) << endl;

    (*mycout) << "Computation time:";
    long computationtime = (clock() - startTime) *1000 / CLOCKS_PER_SEC; // ms
    Display::displayTime(computationtime);
    Display::texEndl(*mycout);
    (*mycout) << endl;

    if(configuration->printRule)
    {
        (*mycout) << endl;
        debugDisplay((*mycout));
    }

    if(configuration->disjonction)
    if(result)
        result = checkDefinition();
    }

    Unification::~Unification()
    {
    }

```

## 6.7 En chiffres

- 8114 lignes de code dont 5934 de C++ pur
- 20 tests de non-régression automatisés
- 14.4 mo de Doxygen
- plus de 140 révisions