

Swan ROCHER

# Outils d'Analyse d'une Base de Règles

# Remerciements

Nous tenons à remercier Marie-Laure Mugnier ainsi que Michel Leclere et Michaël Thomazo pour leur encadrement tout au long de ce projet. De plus, merci par avance aux examinateurs de la soutenance pour leur attention. Enfin, merci également à Mountaz Hascoët pour l'organisation de l'ensemble des TER.

# Table des matières

<b>1</b>	<b>Contexte</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.2	Problématique . . . . .	3
<b>2</b>	<b>Notions de base</b>	<b>4</b>
2.1	Prédicat . . . . .	4
2.2	Atome . . . . .	4
2.3	Conjonction d'atomes . . . . .	4
2.4	Représentation graphique d'une conjonction d'atomes . . . . .	5
2.5	Règle . . . . .	5
2.6	Représentation graphique d'une règle . . . . .	5
2.7	Règle à conclusion atomique . . . . .	6
2.8	Base de connaissance . . . . .	7
2.9	Chânage avant . . . . .	8
2.10	Chânage arrière . . . . .	8
<b>3</b>	<b>Graphe de dépendances des règles</b>	<b>9</b>
3.1	Définition . . . . .	9
3.2	Unification de règles . . . . .	9

<b>4</b>	<b>Classes de règles</b>	<b>15</b>
4.1	Classes abstraites . . . . .	15
4.2	Classes concrètes . . . . .	16
4.3	Schéma d’inclusion des classes de règles . . . . .	18
4.4	Combinaisons . . . . .	19
<b>5</b>	<b>Implémentation</b>	<b>21</b>
5.1	Outils utilisés . . . . .	21
5.2	Structures de donnée . . . . .	21
5.3	Analyseur . . . . .	22
5.4	Détermination d’une classe concrète . . . . .	22
5.5	Combinaison des classes abstraites . . . . .	22
5.6	Formats de fichiers . . . . .	23
<b>6</b>	<b>Perspectives</b>	<b>24</b>

# Chapitre 1

## Contexte

### 1.1 Introduction

A l'heure actuelle, les bases de connaissance sont de plus en plus répandues, et ce dans un grand nombre de domaines. En effet pouvoir représenter les informations et obtenir des réponses à des requêtes sur celles-ci est primordial. Si nous considérons une base de données classique, les seules réponses envisageables sont celle contenues directement dans ces données. Ainsi une requête consiste à vérifier la présence de certaines propriétés sur celles-ci.

Mais pouvoir déduire des informations supplémentaires à partir des initiales est très important, et donc pour cela les bases de connaissance définissent une *ontologie* autour des données. Cette dernière met en place un ensemble de règles qui peuvent être plus ou moins complexes.

Pendant longtemps, seules des règles *simples* étaient prises en compte. Mais le développement du Web-sémantique et le besoin d'effectuer des requêtes toujours plus complexes ont amené à ajouter aux règles la possibilité de créer de nouveaux individus. Ainsi, bien qu'il est soit possible d'obtenir une réponse d'une base de connaissance *simple* (par exemple au format Datalog) à coup sûr, si nous ajoutons ces règles particulières (par exemple au format Datalog+/-) alors il peut arriver qu'aucune réponse ne soit donnée en temps fini. Nous disons donc que le problème de réponse à une requête dans une base de connaissance est de manière général indécidable.

Prenons l'exemple d'une règle disant que "si un individu est un homme, alors il existe un autre individu qui est un homme et le père du premier", et une donnée "Tom est un homme". Il est évident que si nous souhaitons déduire toutes les informations possibles à partir de cette base, il se posera vite un problème. En effet, Tom est un homme, donc il existe un autre homme  $x_1$  qui est le père de Tom. Et donc il existe encore un autre homme  $x_2$  qui a également un père  $x_3$ , etc...

Il est donc nécessaire de pouvoir déterminer des classes de règles (les plus générales possibles), permettant de s'assurer que les réponses aux requêtes soient données en temps fini.

## 1.2 Problématique

Malgré le fait que de manière général, il n'existe aucun algorithme permettant de répondre à ce problème, certaines règles peuvent entrer dans des catégories (qui seront nommées *classes de règles*) qui en ajoutant des contraintes sur la forme des règles s'assurent que le problème soit décidable. Selon quelles contraintes sont satisfaites, il est nécessaire d'appliquer différentes méthodes de réponse sur différents sous-ensembles des règles.

L'objectif de ce TER est donc d'implémenter un outil permettant d'analyser une base de règles afin de construire son graphe de dépendances associés, de déterminer quelles contraintes sont satisfaites, sur quel sous-ensemble, et si la base est décidable d'en déduire quels algorithmes utiliser sur chacun d'eux. De plus, cet outil doit pouvoir charger des bases de règles à partir de fichiers, ainsi que les y écrire, et être suffisamment modulable pour permettre l'ajout de nouvelles vérifications de contrainte.

## Chapitre 2

# Notions de base

Avant toute chose il est nécessaire de définir un certain nombre de notions et de termes.

### 2.1 Prédicat

Un prédicat noté  $p \setminus n$  est un symbole relationnel d'arité  $n$ . Dans la suite, on supposera que tout nom de prédicat est unique et on notera  $p_i$  la  $i^{eme}$  position de  $p$ .

### 2.2 Atome

Un atome  $a = p(a_1, a_2, \dots, a_n)$  associe un terme à chaque position d'un prédicat  $p \setminus n$ . On note :

- $a_i$  le terme en position  $i$  dans  $a$ . Un terme peut être une *constante* ou une *variable*. Une variable peut être *libre* ou *quantifiée* universellement (notée  $\forall - var$ ) ou existentiellement (notée  $\exists - var$ ).
- $dom(a) = \{a_i : \forall i \in [1, n]\}$ , l'ensemble des termes de  $a$
- $var(a)$  l'ensemble des variables de  $a$
- $cst(a)$  l'ensemble des constantes de  $a$

### 2.3 Conjonction d'atomes

Une conjonction de  $n$  atomes  $A$  est définie telle que :  
 $A = \bigwedge_{i=1}^n k_i$  avec  $\forall i \in [1, n]$   $a_i = p_i(a_{i1}, a_{i2}, \dots, a_{in_i})$  un atome de prédicat  $p_i \setminus n_i$ .

## 2.4 Représentation graphique d'une conjonction d'atomes

Une conjonction d'atomes peut être représentée par le graphe non orienté  $G_A = (V_A, E_A, \omega)$  avec  $V_A$  son ensemble de sommets,  $E_A$ , son ensemble d'arêtes et  $\omega$  une fonction de poids sur les arêtes construits de la manière suivante :

- $V_A = P_A \cup T_A$  avec  $P_A = \{i : a_i \in A\}$  et  $T_A = \{t_j \in \text{dom}(A)\}$
- $E_A = \{(i, t_j) : \forall a_i \in A, \forall t_j \in \text{dom}(a_i)\}$
- $\omega : E_A \rightarrow \mathbb{N}$  telle que  $\omega(i, t_j) = j : \forall (i, t_j) \in E_A$

Cette représentation a de nombreux avantages, elle permet notamment de parcourir rapidement les atomes liés à un terme (et réciproquement), ainsi que de pouvoir être visualisée agréablement (voir figure 2.1).



FIGURE 2.1 – Représentation de  $\forall x, \forall y(salle(x) \wedge date(y) \wedge reservee(x, y))$

On remarque que  $G_A$  admet une bipartition de ses sommets, en effet toutes les arêtes ont une extrémité dans  $P_A$  et l'autre dans  $T_A$ , or par construction  $P_A \cap T_A = \emptyset$ .

## 2.5 Règle

Une règle  $R = (H, C)$  est constituée de deux conjonctions d'atomes  $H$  et  $C$  représentant respectivement l'hypothèse (le corps) et la conclusion (la tête) de  $R$ . Toutes les variables apparaissant dans  $H$  sont quantifiées universellement tandis que celles apparaissant uniquement dans  $C$  le sont existentiellement. Ainsi une règle est toujours sous la forme  $R : \forall x_i(H \rightarrow \exists z_j(C))$ .

On note :

- $\text{dom}(R) = \text{dom}(H) \cup \text{dom}(C)$ , le domaine de  $R$
- $\text{var}(R) = \text{var}(H) \cup \text{var}(C)$ , les variables de  $R$
- $\text{cst}(R) = \text{cst}(H) \cup \text{cst}(C)$ , les constantes de  $R$
- $\text{fr}(R) = \text{var}(H) \cap \text{var}(C)$ , l'ensemble des variables frontières de  $R$
- $\text{cutp}(R) = \text{fr}(R) \cup \text{cst}(R)$ , l'ensemble des points de coupure de  $R$

## 2.6 Représentation graphique d'une règle

Tout comme une simple conjonction d'atomes, une règle  $R = (H, C)$  peut être représentée par un graphe similaire, en ajoutant une coloration à deux couleurs : une pour les atomes de l'hypothèse, l'autre pour ceux de la conclusion.

Ainsi le graphe associé  $G_R = (V_R, E_R, \omega, \chi)$  est défini comme :

- $V_R = P_R \cup T_R$  avec  $P_R = \{i : r_i \in R\}$  et  $T_R = \{t_j \in \text{dom}(R)\}$



- $E_R = \{(i, t_j) : \forall r_i \in R, \forall t_j \in \text{dom}(R_i)\}$
- $\omega : E_R \rightarrow [1, p]$  telle que  $\omega(i, t_j) = j : \forall (i, t_j) \in E_R$
- $\chi : P_R \rightarrow \{1, 2\}$  telle que  $\chi(r_i) = 1$  si  $r_i \in H$  et  $\chi(r_i) = 2$  si  $r_i \in C$ .

Par exemple la règle  $R : \forall x \forall y (salle(x) \wedge date(y) \wedge reservee(x, y) \rightarrow \exists z (cours(z) \wedge aLieu(z, x, y)))$  peut être visualisée de la façon suivante :

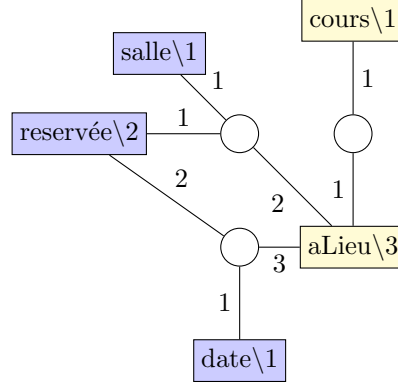


FIGURE 2.2 – Exemple de représentation d'une règle

Sur la figure suivante (2.3) représentant la même règle, on peut voir les différentes parties de celle-ci, de plus les variables ont été étiquetées pour mieux visualiser les différents termes.

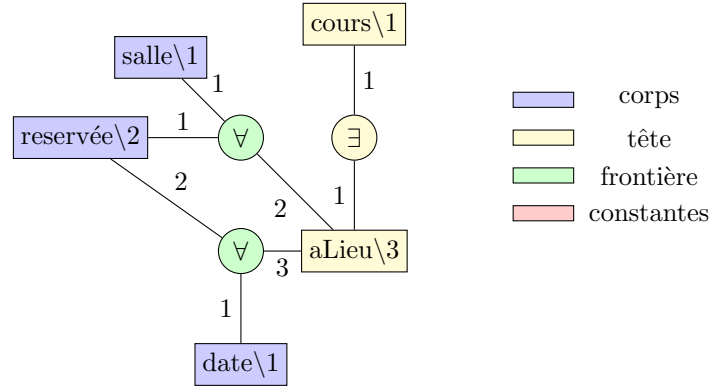


FIGURE 2.3 – Les différents éléments d'une règle

## 2.7 Règle à conclusion atomique

Une règle à conclusion atomique ajoute une contrainte sur la forme de sa conclusion qui ne doit contenir qu'un seul atome. Ces règles ont l'avantage d'être plus simples à unifier (voir section 3.2), et la plupart des algorithmes présentés dans ce rapport sont plus efficaces sur ce type de règle, tandis que d'autres ne fonctionnent uniquement sur celles-ci.

Mais ceci n'est pas un problème puisqu'il est possible de réécrire une règle à conclusion non atomique en un ensemble de règles à conclusions atomiques équivalentes.

En effet, quelle que soit une règle  $R = (H, C)$ , nous pouvons définir un nouveau prédicat  $p_R$  d'arité  $|var(R)|$  ainsi qu'un nouvel ensemble de règles à conclusions atomiques  $R^A$  dont le premier élément aura la même hypothèse que  $R$  et une conclusion de prédicat  $p_R$  contenant toutes ses variables, et dont les suivants auront pour hypothèse cette nouvelle conclusion, et comme conclusion atomique les atomes de  $C$ .

Cet ensemble est donc défini de la manière suivante :

$$R^A = \{R_i^A = (H_i^A, C_i^A) : \forall i \in [0, |C|]\}, \text{ tels que :}$$

$$R_0^A = \begin{cases} H_0^A = H, \\ C_0^A = p_R(\{x_j \in var(R)\}) \end{cases} \quad \forall i \in [1, |C|] \quad R_i^A = \begin{cases} H_i^A = C_0^A, \\ C_i^A = c_i \in C \end{cases}$$

Ainsi nous pouvons utiliser l'algorithme 1 afin d'effectuer cette conversion.

---

**Algorithm 1** Conversion d'une règle à conclusion non atomique

---

**Require:**  $R = (H, C)$  : une règle quelconque

**Ensure:**  $R^A$  : un ensemble de règles à conclusions atomiques équivalent à  $R$

```

1  $H_0^A \leftarrow H$ 
2  $C_0^A \leftarrow p_R(\{x_i \in var(R)\})$ 
3  $R^A \leftarrow \{(H_0^A, C_0^A)\}$ 
4 for all atome  $c_i \in C$  do
5    $R_i^A \leftarrow (C_0^A, c_i)$ 
6    $R^A \leftarrow R^A \cup \{R_i^A\}$ 
7 end for
8 return  $R^A$ 
```

---

Reprenons l'exemple de la section précédente

$R : \forall x \forall y (salle(x) \wedge date(y) \wedge reservee(x, y) \rightarrow \exists z (cours(z) \wedge aLieu(z, x, y)))$ .

Si on applique l'algorithme sus-mentionné nous obtenons l'ensemble de règles :  $R^A =$

$\{R_0^A : \forall x \forall y (salle(x) \wedge date(y) \wedge reservee(x, y) \rightarrow \exists z (p_R(x, y, z)))$ ;

$R_1^A : \forall x \forall y \forall z (p_R(x, y, z) \rightarrow cours(z))$ ;

$R_2^A : \forall x \forall y \forall z (p_R(x, y, z) \rightarrow aLieu(z, x, y))\}$

Toute règle pouvant donc se réécrire de manière équivalente en un ensemble de règles à conclusion atomique, dans la suite nous ne considérerons que des règles sous cette forme.

## 2.8 Base de connaissance

Une base de connaissance  $K = (F, R)$  est constituée d'un ensemble de *faits* représenté par une conjonction d'atomes  $F$ , ainsi que d'une *ontologie* représentée par un ensemble de règles  $R$ .

Les faits sont souvent considérés comme complètement instanciés, c'est à dire ne contenant que des constantes, mais ici, les règles contenant des variables existentielles peuvent générer de nouveaux individus. Donc nous définissons  $F$  comme une conjonction d'atomes existentiellement fermée.

## 2.9 Chaînage avant

Afin de déterminer si une requête booléenne  $Q$  peut être déduite d'une base de connaissance  $B = (R, F)$ , le chaînage avant *dérive* de manière itérative la base de faits  $F$  (qui peut être vue comme un fait unique) par l'ensemble de règles  $R$  de manière à générer (en cherchant de nouveaux homomorphismes) de nouveaux faits qui devront à leur tour être étudiés. Avant chaque dérivation, l'algorithme vérifie si  $F^i$  contient  $Q$  auquel cas la réponse est positive, et si  $F^i = F^{i-1}$  afin de savoir si la chaîne de dérivation est finie et dans ce cas donner une réponse négative.

## 2.10 Chaînage arrière

Le chaînage arrière quant à lui consiste à réécrire la requête  $Q$  en plusieurs nouvelles requêtes jusqu'à ce qu'une de celles-ci soit dans  $F$ . Pour cela les conclusions des règles de la base sont *unifiées* avec des sous-ensembles de  $Q$  (voir section 3.2 pour plus de détails).

## Chapitre 3

# Graphe de dépendances des règles

Le graphe de dépendances de règles est une représentation d'une base de règles très intéressante. En effet il permet de vérifier rapidement quelles règles pourront éventuellement être déclenchées après l'application d'une règle donnée.

De plus il permet de déterminer l'appartenance à certaines classes de règles, et le calcul de ses composantes fortement connexes permet de "découper" la base de manière à effectuer les requêtes de manières différentes selon celles-ci.

### 3.1 Définition

Le graphe de dépendances des règles associé à une base de règles  $B_R$  est défini comme le graphe orienté  $GRD = (V_{GRD}, E_{GRD})$  avec :

- $V_{GRD} = \{R_i \in B_R\}$ ,
- $E_{GRD} = \{(R_i, R_j) : \exists \text{ un bon unificateur } \mu : \mu(C_i) = \mu(H_j)\}$ .

Intuitivement, on crée un sommet par règle et on relie  $R_i$  à  $R_j$  si  $R_i$  "peut amener à déclencher"  $R_j$  ( $R_j$  dépend de  $R_i$ ). La notion d'unificateur est abordée dans la section suivante.

### 3.2 Unification de règles

Afin de pouvoir construire ce graphe, il faut donc pouvoir déterminer si une règle peut en déclencher une autre, c'est à dire s'il existe un unificateur entre la conclusion de la première et l'hypothèse de la seconde. Tout d'abord, l'unification est définie, s'en suit un algorithme permettant de vérifier si un tel unificateur existe, puis la correction de celui-ci ainsi que ses complexités.

### 3.2.1 Définitions

#### Substitution

Une substitution de taille  $n$  d'un ensemble de symboles  $X$  dans un ensemble de symboles  $Y$  est une fonction de  $X$  vers  $Y$  représentée par l'ensemble de couples suivants (avec  $n \leq |X|$ ) :

- $s = \{(x_i, y_i) : \forall i \in [1, n] \ x_i \in X, y_i \in Y, \forall j \neq i \ x_i \neq x_j\}$
- $s(x_i) = y_i \ \forall i \in [1, n]$
- $s(x_i) = x_i \ \forall i \in [n+1, |X|] \ x_i \in X$

#### Unificateur logique

Un unificateur logique entre deux atomes  $a_1$  et  $a_2$  est une substitution  $\mu$  telle que :

- $\mu : \text{var}(a_1) \cup \text{var}(a_2) \rightarrow \text{dom}(a_1) \cup \text{dom}(a_2)$
- $\mu(a_1) = \mu(a_2)$

Cette définition s'étend aux conjonctions d'atomes.

#### Unificateur de conclusion atomique

Un unificateur de conclusion atomique est un unificateur logique  $\mu = \{(x_i, t_i) : \forall i \in [1, n]\}$  entre l'hypothèse d'une règle  $R_1 = (H_1, C_1)$  et la conclusion atomique d'une règle  $R_2 = (H_2, C_2)$  et est défini de la manière suivante :

- $\mu : \text{fr}(R_2) \cup \text{var}(H_1) \rightarrow \text{dom}(C_2) \cup \text{cst}(H_1)$
- $\forall (x_i, t_i) \in \mu \text{ si } x_i \in \text{fr}(R_2) \text{ alors } t_i \in \text{cutp}(R_2) \cup \text{cst}(H_1)$

#### Bonne unification atomique

Un bon unificateur de conclusion atomique est un unificateur de conclusion atomique  $\mu = \{(x_i, t_i) : \forall i \in [1, n]\}$  entre un sous ensemble  $Q$  de l'hypothèse d'une règle  $R_1 = (H_1, C_1)$  et la conclusion d'une règle atomique  $R_2 = (H_2, C_2)$  tel que :

$\forall (x_i, t_i) \in \mu : \text{si } x_i \in H_1 \setminus Q \text{ alors } t_i \text{ n'est pas une } \exists - \text{var}$

Un tel ensemble  $Q$  est appelé un *bon ensemble d'unification atomique* de l'hypothèse de  $R_1$  par la conclusion de  $R_2$ . On note que  $Q$  est donc défini comme suit :

- $Q \subseteq H_1$
- $\forall \text{ position } i \text{ de } \exists - \text{var dans } C_2, \forall \text{ atome } a \in Q, \text{ si } a_i \in \text{var}(H_1) \text{ alors } \forall \text{ atome } b \in H_1 : \text{si } \exists b_j \in b : a_i = b_j, \text{ alors } b \in Q$

#### Bon ensemble d'unification atomique minimal

Un *bon ensemble d'unification atomique minimal*  $Q$  de  $H_1$  par  $C_2$  enraciné en  $a$  est défini tel que :

- $Q$  est un bon ensemble d'unification atomique de  $H_1$  par  $C_2$
- $a \in Q$
- $|Q| = \min(|Q_i| : Q_i \text{ est un bon ensemble d'unification atomique de } H_1 \text{ par } C_2 \text{ et } a \in Q_i)$

### 3.2.2 Algorithmes

Vérifier qu'une règle atomique  $R_i$  peut déclencher  $R_j$  consiste donc à trouver un bon unificateur atomique entre  $R_i$  et  $R_j$ . Dans cette section, un algorithme permettant de répondre à ce problème est détaillé.

Dans la suite, les règles sont supposées représentées par des graphes (tels que définis en 2.6) et à conclusion atomique.

Le premier algorithme fait appel aux deux suivants de manière à déterminer si il existe au moins un unificateur entre les deux conjonctions d'atomes. En première phase, il vérifie l'existence d'unificateurs avec chaque atome de manière indépendante. S'ensuit une extension à partir des atomes préselectionnés, et dès qu'un bon ensemble d'unification est entièrement unifié, l'algorithme s'arrête en répondant avec succès.

---

**Algorithm 2** Unification

---

**Require:**  $H_1$  : conjonction d'atomes,  $R = (H_2, C_2)$  : règle à conclusion atomique

**Ensure:** succès si  $C_2$  peut s'unifier avec  $H_1$ , i.e. si  $\exists H \subseteq H_1, \mu$  une substitution :  $\mu(H_1) = \mu(C_2)$ , échec sinon

```

1  ▷ Précoloration
2  for all sommet atome  $a \in H_1$  do
3      if  $UnificationLocale(a, R) \neq \text{échec}$  then
4           $couleurLogique[a] \leftarrow \text{noir}$ 
5      else
6           $couleurLogique[a] \leftarrow \text{blanc}$ 
7      end if
8  end for
9  ▷ Initialisation du tableau contenant les positions des variables existentielles de  $C_2$ 
10  $E \leftarrow \{i : c_i \text{ est une } \exists\text{-var de } C_2\}$ 
11 ▷ Extension des ensembles
12 for all sommet atome  $a \in H_1 : couleurLogique[a] = \text{noir}$  do
13     if  $Q \leftarrow Extension(H_1, a, couleurLogique, E) \neq \text{échec}$  then
14         if  $UnificationLocale(Q, R) \neq \text{échec}$  then
15             return succès
16         end if
17     end if
18      $couleurLogique[a] \leftarrow \text{blanc}$ 
19 end for
20 return échec

```

---

Le deuxième algorithme est utilisé pour le calcul des bons ensembles d'unification à partir d'un atome racine. Tant qu'aucune erreur n'est détectée il *avale* les atomes voisins aux termes en positions existentielles. Les positions existentielles sont les indices des variables existentielles dans l'atome de conclusion.

---

**Algorithm 3** Extension

---

**Require:**  $H_1$  : conjonction d'atomes,  $a \in H_1$  : sommet atome racine,  $couleurLogique$  : tableau de taille égal au nombre d'atomes dans  $H_1$  tel que  $couleurLogique[a] = noir$  ssi  $UnificationLocale(a, R) = succès$ ,  $E$  : ensemble des positions des variables existentielles

**Ensure:**  $Q$  : bon ensemble d'unification minimal des atomes de  $H_1$  construit à partir de  $a$  s'il existe, échec sinon.

```
1  ▷ Initialisation du parcours
2  for all sommet atome  $a \in H_1$  do
3      if  $couleurLogique[a] = noir$  then
4          for all sommet terme  $t \in voisins(a)$  do
5               $couleur[t] = blanc$ 
6          end for
7           $couleur[a] = blanc$ 
8      end if
9  end for
10  $couleur[a] \leftarrow noir$ 
11  $Q \leftarrow \{a\}$  ▷ conjonction d'atomes à traiter
12  $attente \leftarrow \{a\}$  ▷ file d'attente du parcours
13 while  $attente \neq \emptyset$  do
14      $u \leftarrow haut(attente)$ 
15     if  $u$  est un atome then
16         for all  $i \in E$  do
17              $v \leftarrow voisin(u, i)$ 
18             if  $v$  est une constante then
19                 return échec
20             else if  $couleur[v] = blanc$  then
21                 ▷  $v$  est une  $\forall$  - var non marquée par le parcours
22                  $couleur[v] \leftarrow noir$ 
23                  $attente \leftarrow attente \cup \{v\}$ 
24             end if
25         end for
26     else
27         ▷  $u$  est un terme
28         for all  $v \in voisins(u)$  do
29             if  $couleurLogique[v] = blanc$  then
30                 return échec
31             else
32                 if  $couleur[v] = blanc$  then
33                      $couleur[v] \leftarrow noir$ 
34                      $attente \leftarrow attente \cup \{v\}$ 
35                      $Q \leftarrow Q \cup \{v\}$ 
36                 end if
37             end if
38         end for
39     end if
40 end while ▷ Fin du parcours
41 return  $Q$ 
```

---

Remarque :

La phase d'initialisation du parcours pourrait simplement parcourir tous les sommets de  $H_1$  et mettre leur couleur à blanc. En pratique, cette solution serait sans doute plus efficace, mais dépendrait donc du nombre de sommets total dans  $H_1$ . Ce qui en théorie amènerait la complexité de cette boucle en  $\mathcal{O}(\text{nombre d'atomes} \times \text{arite max de } H_1)$ . Or ici, la complexité ne dépend pas de cette arité max, mais uniquement de l'arité du prédicat de la conclusion  $C_2$ .

Le dernier algorithme est celui qui teste réellement si il existe un unificateur entre une conclusion atomique, et un bon ensemble d'unification atomique minimal. Il est appelé une première fois pour tester les atomes de la conjonction séparément, et permettre une préselection des atomes (qui vont servir de racine). Durant la dernière phase (lorsqu'il est appelé sur les ensembles étendus), s'il trouve un unificateur, celui-ci assure que la règle peut amener à déclencher la conjonction.

---

**Algorithm 4** UnificationLocale
 

---

**Require:**  $H_1$  : conjonction d'atomes,  $R = (H_2, C_2)$  : règle à conclusion atomique  
**Ensure:** succès si  $C_2$  peut s'unifier avec  $H_1$

```

1  ▷ Vérification des prédicats
2  for all atome  $a \in H_1$  do
3      if  $\text{prédicat}(a) \neq \text{prédicat}(C_2)$  then
4          return échec
5      end if
6  end for
7   $u \leftarrow \emptyset$  ▷ substitution
8  for all terme  $t_i \in C_2$  do
9      ▷ def :  $a_i$  = terme de  $a$  en position  $i$ 
10      $E \leftarrow \{a_i : \forall \text{ atome } a \in H_1\}$ 
11     if  $t_i$  est une constante then
12         if  $\exists v \in E : v$  est une constante et  $v \neq t_i$ , ou  $v$  est une  $\exists$  – variable then
13             return échec
14         else
15              $u \leftarrow \{(v, t_i) : v \in E \text{ et } v \neq t_i\}$ 
16         end if
17     else if  $t_i$  est une  $\exists$  – variable then
18         if  $\exists v \in E : v$  est une  $\exists$  – variable et  $v \neq t_i$ , ou  $v$  est une constante then
19             return échec
20         else
21              $u \leftarrow \{(v, t_i) : v \in E \text{ et } v \neq t_i\}$ 
22         end if
23     else
24         if  $\exists v_1, v_2 \in E : v_1 \neq v_2$  et  $v_1, v_2$  ne sont pas des  $\forall$  – variables then
25             return échec
26         else if  $\exists c \in E : c$  est une constante then
27              $u \leftarrow \{(v, c) : v \in E \cup \{t_i\} \text{ et } v \neq c\}$ 
28         else
29              $u \leftarrow \{(v, t_i) : v \in E \text{ et } v \neq t_i\}$ 
30         end if
31     end if
32      $H_1 \leftarrow u(H_1)$ 
33      $C_2 \leftarrow u(C_2)$ 
34 end for
35 return succès
  
```

---

### 3.2.3 Correction

### 3.2.4 Complexités

Soit  $R$  une règle, et  $G_R$  son graphe associé. On note :

- $k$  : nombre d'atomes dans  $R$
- $p$  : arité maximum des prédicats de  $R$



- $t \leq n \times p$  : nombre de termes de  $R$

et :

- $n = k + t$  : nombre de sommets dans  $G_R$
- $m \leq n \times p$  : nombre d'arêtes dans  $G_R$

Avec notre représentation nous avons donc les complexités suivantes :

- Parcourir les sommets prédicats :  $\mathcal{O}(k)$ .
- Parcourir tous les sommets :  $\mathcal{O}(n)$ .
- Accéder au  $i^{eme}$  voisin d'un sommet :  $\mathcal{O}(1)$ .

- $n$  = nombre d'atomes dans  $H_1$
- $p$  = arité de  $C_2$
- $t$  = nombre de termes "colorables" dans  $H_1$
- $m$  = nombre d'arêtes "suivables" dans  $H_1$

On remarque que dans le pire des cas on a :

- $t = n \times p$
- $m = n \times p$

## Temps

- UnificationLocale (pire des cas) =  $\mathcal{O}(np)$
- Extension (pire des cas) =  $\mathcal{O}(np)$
- Unification (pire des cas) =  $\mathcal{O}(n^2p)$

### Extension

On effectue simplement un parcours en largeur à partir d'un sommet donné qui peut éventuellement s'arrêter plus tôt qu'un parcours classique. La complexité en temps dans le pire des cas est donc au plus la même, c'est à dire linéaire au nombre de sommets plus le nombre d'arcs (voir [3] page 517-525). Le graphe représentant la conjonction d'atomes  $H_1$  possèdent  $n \times ariteMax(H_1)$  arêtes et  $n \times (ariteMax(H_1) + 1)$  sommets. On sait donc que  $C_{Extension}^{temps} = \mathcal{O}(n \times ariteMax(H_1))$ .

Deux cas :

i) Découverte d'un sommet atome :

parcours classique =  $\downarrow$  on récupère tous les voisins blancs

parcours extension =  $\downarrow$  on récupère tous les voisins blancs de couleur logique noire (en effet arrêt immédiat si un voisin de couleur logique noire est découvert).

ii) Découverte d'un sommet variable :

pas de différence

On a donc que l'algo ne suit que les sommets atomes pouvant être unifiés localement. C'est à dire (entre autres) que leur prédicat est égal à  $predicat(C_2)$ .

...

## Espace

# Chapitre 4

## Classes de règles

### 4.1 Classes abstraites

Trois classes abstraites ont été définies, chacune permettant l'usage de certains algorithmes sur l'ensemble de règles considéré. Elles sont dites *abstraites* puisque déterminer si un ensemble de règles appartient à l'une de ces classes est un problème non décidable. En effet ces classes n'imposent aucune contrainte vérifiable. De plus elles sont incomparables entre elles, et non exclusives.

#### 4.1.1 Finite Expansion Set

La première classe est définie comme assurant la finition des algorithmes de chaînage avant. Ainsi tout ensemble de règles appartenant à cette classe peut être utilisé pour les dérivations de ces algorithmes.

#### 4.1.2 Bounded Treewidth Set

La deuxième définit quant à elle les ensembles de règles où la production de nouveaux faits suit la forme d'un arbre de largeur bornée. Cette classe ne permet pas l'utilisation direct d'algorithmes, mais par contre la classe abstraite Greedy Bounded Treewidth qui est une spécialisation de celle-ci, s'assure que le chaînage avant s'exécute en temps fini, et ce via un algorithme glouton permettant de construire la décomposition de l'arbre et en utilisant une condition d'arrêt spécifique (l'algorithme est détaillé en [1]).

#### 4.1.3 Finite Unification Set

Enfin la dernière classe abstraite assure la finition des algorithmes de chaînage arrière utilisant les méthodes en *largeur* qui ne garderait que les faits *le plus généraux* lors de leur génération (voir [4]).

## 4.2 Classes concrètes

Comme expliqué dans la section précédente, il est impossible de déterminer à coup sûr si un ensemble de règles appartient à l'une de ces classes, par contre de nombreuses classes dites *concrètes* ont été définies. Chacune d'entre elle, en ajoutant des contraintes sur la forme des règles voir sur l'ensemble des règles, permet de s'assurer que celui-ci appartient à certaines des classes abstraites.

### 4.2.1 Acyclicité du graphe de dépendance des règles

Le seul fait que le graphe de dépendance soit sans circuit suffit à certifier que le chaînage avant et arrière s'exécutent en temps fini, impliquant que si cette contrainte est satisfaite, l'ensemble des règles appartient à *FES* et à *FUS*.

### 4.2.2 Faiblement acyclique

Cette classe est quant à elle un peu particulière puisqu'elle demande la génération d'une autre structure de graphes et qu'elle s'applique directement sur un ensemble de règles et pas uniquement sur chacune des règles indépendamment des autres.

On crée donc un graphe de dépendances des positions dont les sommets sont les positions des prédicats et dont la construction des arcs est la suivante : pour chaque variable  $x$  d'une règle  $R$  apparaissant dans l'hypothèse en position  $p_i$ , si  $x$  appartient à la frontière de  $R$  alors il existe un arc de  $p_i$  vers chacune des positions  $r_j$  de la conclusion de  $R$  dans laquelle apparaît  $x$ , de plus pour chacune des variables existentielles apparaissant en position  $q_k$  il existe un arc *spécial* de  $p_i$  vers  $q_k$ .

Si le graphe de dépendances des positions associés à un ensemble de règle ne contient aucun circuit passant par un arc *spécial* alors il est dit *faiblement acyclique* et appartient à *FES*.

Voir [2] pour les détails et les preuves.

### 4.2.3 Sticky

Un ensemble de règles satisfait cette contrainte si certaines de ses variables (marquées) n'apparaissent pas plusieurs fois dans l'hypothèse d'une de ses règles.

Plus précisément, on marque tout d'abord les variables en deux étapes :

- premièrement, pour chaque règle  $R$ , et pour chacune des variables  $x$  de l'hypothèse  $H$  de  $R$ , si  $x$  n'apparaît pas dans la conclusion de  $R$  alors on marque chaque occurrence de  $x$  dans  $H$ .

- Deuxièmement, pour chaque règle  $R$  si une variable marquée apparaît en position  $p_i$  alors pour chaque règle  $R'$  (incluant  $R = R'$ ), et pour chacune des variables  $x$  apparaissant en position  $p_i$  dans la conclusion de  $R'$ , on marque chaque occurrence de  $x$  dans l'hypothèse de  $R'$ .

Ensuite si aucune règle ne contient plusieurs occurrence d'une variable marquée dans son hypothèse, l'ensemble est dit *sticky*, et assure la finition des algorithmes basés sur le chaînage arrière, cette classe appartient donc à la classe abstraite *FUS* (voir [2]).

#### 4.2.4 Faiblement sticky

Cette classe est une généralisation des deux précédentes qui malheureusement n'appartient à aucune des classes abstraites citées dans la section précédente (cf [2]).

Tout comme la vérification de l'appartenance à la classe *faiblement acyclique* (4.2.2), il est nécessaire de construire le graphe de dépendances des positions. De plus, on dit qu'une position (un sommet de ce graphe) est *sûre* si elle n'apparaît dans aucun circuit contenant un arc *spécial*, et on marque chacune des variables en suivant le même algorithme que pour la classe *sticky* (4.2.3).

On peut maintenant définir un ensemble de règles *faiblement sticky* comme ne contenant que des variables sûres ou non marquées.

#### 4.2.5 Gardée

Une règle gardée est définie comme étant une règle dont un atome de son hypothèse (nommé *garde*) contient toutes les variables de celle-ci. Si toutes les règles de l'ensemble contiennent un garde, alors l'ensemble appartient à *GBTS*.

#### 4.2.6 Frontière gardée

On dit qu'une règle a une *frontière gardée* si un atome de son hypothèse possède toutes les variables de la frontière. On peut remarquer que cette classe est une généralisation de la précédente. Et dans le cas où toutes les règles de l'ensemble possède cette propriété, celui-ci appartient également à *GBTS*.

#### 4.2.7 Frontière-1

Cette classe contient les règles dont la frontière est de taille 1, elle est donc une spécialisation des règles à frontière gardée, Ainsi un ensemble de règles satisfaisant cette propriété appartient à *GBTS*.

### 4.2.8 Hypothèse atomique

Les règles ne contenant que des hypothèses atomiques s'assurent que la règle est *gardée* (4.2.5), et de plus assurent que les algorithmes basés sur le chaînage arrière se terminent en temps fini. Donc si toutes les règles d'un ensemble sont à hypothèse atomique alors celui-ci est *GBTS* et *FUS*.

### 4.2.9 Domaine restreint

Une règle satisfait cette contrainte si tous les atomes de sa conclusion contiennent soit toutes les variables de l'hypothèse, soit aucune. Dans le cas des règles à conclusion atomique, cela revient à s'assurer que la frontière de chaque règle est soit égale à 0 soit au nombre de variables universelles. Cette contrainte est suffisante pour que l'ensemble de règles appartienne à *FUS*.

### 4.2.10 Règle déconnectée

Une règle est dite déconnectée si sa frontière est vide, cette classe est donc une spécialisation des règles à *domaine restreint* (4.2.9), à *frontière gardée* (4.2.6) et *faiblement acyclique* (4.2.2). Ce type de règle n'est pas très utilisé étant donné que seules des constantes sont partagées entre l'hypothèse et la conclusion ce qui limite leur usage, mais elles ont l'avantage d'être à la fois *FES*, *GBTS* et *FUS* puisque qu'une règle déconnectée n'a besoin de s'appliquer qu'une seule fois.

### 4.2.11 Règles universelles

Les règles ne contenant aucune variable existentielle ( $vars(C) \subseteq vars(H)$ ) forment bien entendu un ensemble décidable. Celles-ci sont à la fois *FES* et *GBTS*. On peut également remarquer que toute ensemble de règles universelles est également *faiblement acyclique* (4.2.2).

## 4.3 Schéma d'inclusion des classes de règles

Sur la figure 4.1, les flèches représentent la spécialisation tandis que les couleurs l'appartenance aux classes abstraites. De plus, seules les classes citées ci-dessus apparaissent sur celle-ci.

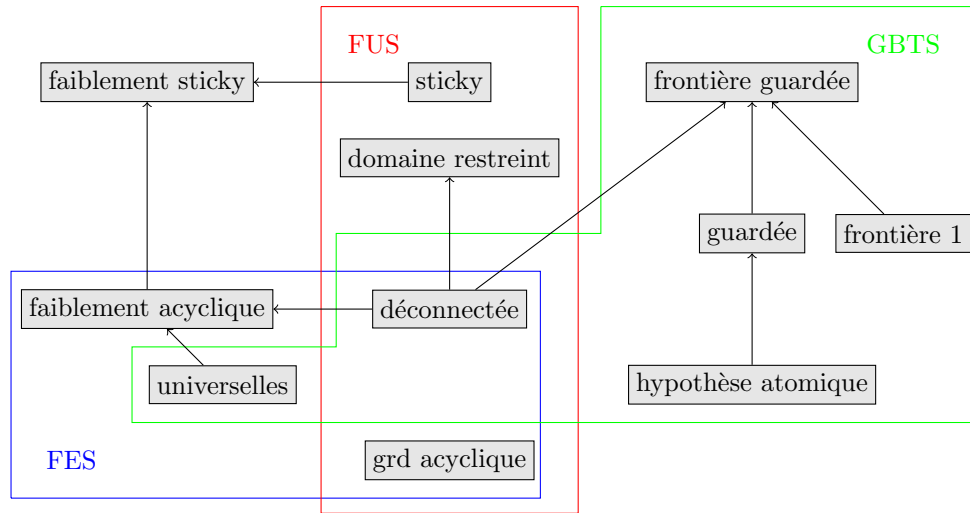


FIGURE 4.1 – Schéma récapitulatif des classes de règles

## 4.4 Combinaisons

Si l'ensemble des règles appartient à une des classes concrètes, alors il suffit d'appliquer les algorithmes correspondant pour répondre aux requêtes. Toutefois, il est possible que ce ne soit pas le cas, et il faut donc pouvoir *découper* l'ensemble des règles de manière à appliquer des méthodes de réponse différentes en fonction du sous-ensemble.

Dans cette optique nous utilisons les composantes fortement connexes du graphe de dépendances de façon à attribuer des étiquettes différentes à celles-ci en fonction des classes concrètes auxquelles leurs ensembles de règles appartiennent.

Puis une fois chaque sous ensemble étiqueté, il faut encore vérifier que celles-ci sont compatibles entre elles. Pour cela, nous définissons le graphe orienté des composantes fortement connexes associé  $G_C = (V_C, E_C)$  tel que son ensemble de sommets est l'ensemble des composantes du graphe, et qu'il existe un arc entre deux composantes  $C_i$  et  $C_j$  si et seulement s'il existe un arc d'un sommet de  $C_i$  vers un sommet de  $C_j$ . Par définition des composantes fortement connexes, ce graphe est évidemment sans circuit.

On dit que  $C_i$  *précède*  $C_j$  s'il n'existe aucun arc de  $C_j$  vers  $C_i$ , et on note cette relation  $C_i \triangleright C_j$ .

De plus, on définit une fonction *etiquette* :  $V_C \rightarrow \{FES, GBTS, FUS\}$  qui associe à chaque  $C_i$  une étiquette qui déterminera la classe abstraite considérée pour cette composante.

La propriété suivante s'assure que l'ensemble des règles est décidable (voir ??) :  $\{C_i : etiquette(C_i) = FES\} \triangleright \{C_i : etiquette(C_i) = GBTS\} \triangleright \{C_i : etiquette(C_i) = FUS\}$  C'est à dire qu'aucune règle *FES* ne doit dépendre d'une règle *FUS* ou *GBTS*, et qu'aucune règle *GBTS* ne doit dépendre d'une règle *FUS*.

En effet les algorithmes de chaînage arrière par exemple réécrivent la requête jusqu'à ce qu'elle corresponde à la base, tandis que ceux avant ajoutent des faits jusqu'à obtenir

la requête. Il est donc évident que si une composante n'accepte que le chaînage arrière, il ne doit exister aucune règle de celle-ci de laquelle dépende une règle de la composante acceptant uniquement le chaînage avant (si tel était le cas, cette règle ne serait jamais déclenchée).

## Chapitre 5

# Implémentation

### 5.1 Outils utilisés

Nous avons mis en place un dépôt *git* () comme gestionnaire de versions afin de pouvoir modifier et partager le code source sans problème, le langage *Java* a été choisi pour des raisons de compatibilité avec les futurs utilisateurs de l'analyseur.

### 5.2 Structures de donnée

Nous utilisons de nombreux graphes différents que ce soit pour stocker des informations ou même pour vérifier les contraintes de certaines classes de règles. Ainsi nous avons mis en place une structure générique, permettant de choisir le type de sommets et d'arcs, ainsi que de coder différemment les ensembles d'arêtes (ou d'arcs) et de sommets de manière différentes selon la situation, tout en fournissant les algorithmes de graphes classiques.

#### 5.2.1 Règle

Nous avons donc choisi d'utiliser la représentation graphique des règles telle que définie dans la section 2.6. Ainsi une structure de graphe biparti non orienté a été mise en place, l'ensemble des arêtes est codé par une liste de voisinage pour chaque sommet. En effet l'opération de parcours des voisins doit être la plus efficace possible, étant donné qu'elle nécessaire pour grand nombre d'algorithmes ainsi pour la création des atomes. Le graphe étant biparti, l'ensemble de sommets est divisé en deux parties permettant des accès rapides à l'une comme à l'autre, mais perdant de l'efficacité sur certaines opérations peu fréquentes telles que la suppression d'un sommet. Les arêtes sont quant à elle typées par un entier, permettant de connaître la position d'un terme dans un atome, les termes devraient être de préférence ajoutés dans l'ordre (et après les prédicats) de manière à faciliter leur parcours (et à optimiser le temps nécessaire à la création de la règle).

Notons tout de même qu'une règle est une spécialisation d'une conjonction d'atomes, et que c'est cette-dernière qui est en charge de la gestion du graphe hormis la séparation entre l'hypothèse et la conclusion.



### 5.2.2 Graphe de dépendances des règles

Ensuite, le graphe de dépendances des règles est quant à lui un graphe orienté dont les arcs sont également codés par listes de voisinage et ne possèdent pas de poids. Il est évident que les arcs d'un graphe de dépendances ont plus intérêt à être implémentés de cette manière puisque l'objectif de construire un tel graphe est de connaître rapidement de quel sommet dépend quel autre.

### 5.2.3 Graphe de dépendances des positions

Enfin le graphe de dépendances des positions utilisés pour vérifier l'appartenance à certaines classes de règles (faiblement acyclique (4.2.2) et faiblement sticky (4.2.4)) est aussi un graphe orienté, mais ici la fonction de poids sur les arcs permet de savoir si un arc est *spécial* ou non. Les sommets représentent les positions des prédicats et sont stockées à la suite. De plus, une table de hachage gère l'accès à la première position d'un prédicat en fonction de son nom, ainsi les opérations d'accès sont aussi légères que possible, et ce type de graphe n'étant conservé en mémoire que le temps de la vérification des contraintes, l'espace supplémentaire utilisé par la table est négligeable.

## 5.3 Analyseur

Les règles (les données) sont donc stockées directement dans le graphe de dépendances des règles dont une instance est encapsulée dans la classe *GRDAnalyser* qui représente notre analyseur. De plus celui-ci est en fait constitué de deux parties distinctes supplémentaires : la première est en charge de la détermination des classes concrètes, tandis que la seconde vérifie si la base est décidable et combine les classes abstraites de manière à savoir quels algorithmes utiliser.

## 5.4 Détermination d'une classe concrète

En section 4.2 nous avons défini de nombreuses classes de règles qu'il faut donc pouvoir déterminer. Pour cela, nous avons déclaré une interface de fonction *DecidableClassCheck* fournissant une méthode renvoyant une étiquette à partir d'un ensemble de règles. L'analyseur de classes concrètes contient une liste des contraintes à tester, et lors de son exécution, il vérifie tout d'abord l'ensemble complet des règles sur chacune de celles-ci, puis ensuite sur chaque composante fortement connexe du graphe de dépendances.

## 5.5 Combinaison des classes abstraites

Comme expliqué plus en détails dans la section 4.4 il est ensuite nécessaire de vérifier si l'ensemble est bel et bien décidable. L'analyseur de classes abstraites regarde donc tout

d'abord si l'ensemble des règles est étiqueté par une classe concrète, si tel est le cas, une des approches pour répondre à une requête est donc d'exécuter l'algorithme correspondant. Si ce n'est pas le cas, il associe la valeur 1 à l'étiquette *FES*, 2 à *GBTS* et 3 à *FUS* de manière à avoir un ordre sur celles-ci, puis il effectue un parcours en largeur du graphe des composantes fortement connexes à partir de l'ensemble des sources de celui-ci, attribuant à chaque sommet découvert (qu'il soit déjà traité ou non) la plus petite étiquette fournie par ses classes concrètes et supérieure à celle de son prédécesseur ou 0 si ce n'est pas possible. Une fois cette opération effectuée, si tous les sommets sont étiquetés par des valeurs strictement positives, l'ensemble de règles est décidable.

## 5.6 Formats de fichiers

Le graphe de dépendances des règles est capable de charger une base à partir d'un fichier, celui-ci devant être écrit dans un format spécifique : chaque ligne doit être une règle de la forme suivante :

$atome_1; atome_2; \dots; atome_n - - > atome_c$

avec  $n$  le nombre d'atomes dans l'hypothèse et  $atome_c$  l'unique atome de la conclusion et où chaque atome  $i$  est écrit :  $p_i(t_{i1}, t_{i2}, \dots, t_{ik})$

Les termes sont interprétés comme des constantes s'ils sont encadrés par des simple guillemets.

Par exemple la règle  $\forall x \forall y (p(x, a) \wedge q(y) \rightarrow \exists z (r(x, y, z)))$  doit être écrite :

$p(x, 'a'); q(y) - - > r(x, y, z)$

En plus du format interne ci-dessus, il est également possible de fournir un fichier Datalog (.dtg) qui ne contient que des règles à hypothèse atomique. Chaque ligne est soit une règle, soit un commentaire auquel cas elle doit débiter par //. Ici les règles sont sous le format suivant :

$[!]atome_c :- atome_h.$

Le point d'exclamation est utilisé pour signaler la négation d'une conclusion, celle-ci sera convertie en une règle contenant ses deux atomes actuels dans son hypothèse et ayant une conclusion au prédicat spécial *ABSURD*. De plus, les termes sont maintenant considérés comme des variables s'ils commencent par un point d'interrogation et comme des constantes sinon.

Ainsi la ligne du fichier correspondant à la règle  $\forall x (p(x, a) \rightarrow \exists z (r(x, a, z)))$  doit être :

$r(?x, a, ?z) :- p(?x, a).$

## Chapitre 6

# Perspectives

En conclusion, nous pouvons dire qu’une grande partie des classes concrètes exhibées à ce jour sont reconnues par notre analyseur, et ainsi celui-ci est la plupart du temps capable de déterminer si le problème d’effectuer une requête sur un ensemble de règles est décidable ou non.

Mais à l’heure actuelle, d’autres classes sont définies régulièrement et il serait intéressant de les implémenter afin de fournir un outil complet, voir d’en rechercher de nouvelles.

Quant à elle, la recherche sur la combinaison des classes abstraites en est à ses débuts. Ainsi il est éventuellement possible de mettre en évidence d’autres situations où le problème reste décidable sans pour autant satisfaire la propriété de précédance utilisée.

De plus, pour le moment notre outil ne possède pas d’interface graphique, or il pourrait être agréable de visualiser le graphe de dépendances des règles et ses composantes fortement connexes directement. En effet, on pourrait ainsi ajouter, ou retirer des règles pour étudier les changements de décidabilité lors de ces modifications.

Enfin il existe de nombreux formats différents utilisés pour représenter des ontologies, et il serait intéressant de pouvoir transformer un maximum de bases afin de les analyser, et dans cette optique il faudrait donc mettre en place d’autres convertisseurs de fichiers.

# Bibliographie

- [1] J.F. Baget, M.L. Mugnier, S. Rudolph, M. Thomazo, et al. Walking the complexity lines for generalized guarded existential rules. In *Proc. 22nd Int. Conf. on Artificial Intelligence (IJCAI'11)*. *IJCAI*, 2011.
- [2] A. Cali, G. Gottlob, and A. Pieris. Query answering under non-guarded rules in datalog+/- . *Web Reasoning and Rule Systems*, pages 1–17, 2010.
- [3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, T.H. Cormen, and T.H. Cormen. *Introduction à l'algorithmique*. Dunod, 1994.
- [4] M.L. Mugnier. Ontological query answering with existential rules. *Web Reasoning and Rule Systems*, pages 2–23, 2011.

```

1  import moca.graphs.*;
2  import moca.graphs.vertices.*;
3  import moca.graphs.edges.*;
4  import java.lang.Exception;
5  import java.util.Iterator;
6  import java.util.ArrayList;
7  import obr.*;
8
9  public class Main {
10     public static void main(String args[]) {
11         try {
12             System.out.println("Reading file : "+args[0]);
13             GraphRuleDependencies grd = null;
14             String[] filePathSubs = args[0].split("\\.");
15             if (filePathSubs.length == 1)
16                 grd = new GraphRuleDependencies(args[0]);
17             else if (filePathSubs[filePathSubs.length-1].equals("dtg")) {
18                 System.out.println("Invoking DTG parser ...");
19                 grd = DTGParser.parseRules(args[0]);
20             }
21
22             System.out.println("\n[1] GRAPH OF RULE DEPENDENCIES\n"+grd);
23             System.out.println("\n[2] STRONGLY CONNECTED COMPONENTS\n");
24             System.out.println(grd.stronglyConnectedComponentsToString());
25
26             GRDAnalyser analyser = new GRDAnalyser(grd);
27             analyser.addDecidableClassCheck(new DisconnectedCheck());
28             analyser.addDecidableClassCheck(new RangeRestrictedCheck());
29             analyser.addDecidableClassCheck(new AtomicHypothesisCheck());
30             analyser.addDecidableClassCheck(new WeaklyAcyclicCheck());
31             analyser.addDecidableClassCheck(new FrontierOneCheck());
32             analyser.addDecidableClassCheck(new GuardedCheck());
33             analyser.addDecidableClassCheck(new FrontierGuardedCheck());
34             analyser.addDecidableClassCheck(new DomainRestrictedCheck());
35             analyser.addDecidableClassCheck(new StickyCheck());
36             analyser.addDecidableClassCheck(new WeaklyStickyCheck());
37
38             System.out.println("\n[3] DIAGNOSTIC\n");
39             analyser.process();
40             System.out.println(analyser.diagnostic());
41
42             if (args.length >= 2)
43                 grd.toPostScript(args[1]);
44             if (args.length >= 3)
45                 grd.sccToPostScript(args[2]);
46         }
47         catch (Exception e) {
48             System.out.println(e);
49             e.printStackTrace();
50         }
51     }
52 };

```

```

1 package obr;
2
3 import java.lang.String;
4 import java.lang.Iterable;
5 import java.util.NoSuchElementException;
6 import java.util.ArrayList;
7
8 /**
9  * Represents an atom.<br />
10  * Note that instances of this class can be returned by the atom conjunction class, but this ←
11  * will imply
12  * the atom creation. I.e., this class is not used to store atom information, but to get a ←
13  * convenient
14  * instance.
15  */
16 public class Atom implements Iterable<Term> {
17
18     /** Represents the begin of a term list when the atom is under a String format. */
19     public static final String BEGIN_TERM_LIST = "\\(";
20     public static final String BEGIN_TERM_LIST_W = "(";
21     /** Represents the end of a term list when the atom is under a String format. */
22     public static final String END_TERM_LIST = "\\)";
23     public static final String END_TERM_LIST_W = ")";
24     /** Represents the term separator when the atom is under a String format. */
25     public static final String TERM_SEPARATOR = ",";
26
27     /**
28      * Constructor. <br />
29      * Note that there is no default constructor since this class should not be used directly ←
30      * but be get by an
31      * atom conjunction method.
32      * @param predicate The predicate of the atom to be instantiated.
33      */
34     public Atom(Predicate predicate) {
35         _predicate = predicate;
36         _terms = new ArrayList<Term>(_predicate.getArity());
37         for (int i = 0 ; i < getArity() ; i++)
38             _terms.add(null);
39     }
40
41     /**
42      * Predicate getter.
43      * @return The atom predicate.
44      */
45     public Predicate getPredicate() {
46         return _predicate;
47     }
48
49     /**
50      * Predicate setter.
51      * @param value The new predicate to set.
52      */
53     public void setPredicate(Predicate value) {
54         _predicate = value;
55         _terms.clear();
56         for (int i = 0 ; i < getArity() ; i++)
57             _terms.add(null);
58     }
59
60     /**
61      * Convenient method to get the predicate arity.
62      * @return The atom arity.
63      */
64     public int getArity() {
65         return _predicate.getArity();
66     }
67
68     /**
69      * Term getter.
70      * @param i The term index inside the atom.
71      * @return The term corresponding to this index.
72      * @throws NoSuchElementException If i is out of range.
73      */
74     public Term get(int i) throws NoSuchElementException {
75         if ((i >= getArity()) || (i < 0))
76             throw new NoSuchElementException();
77         return _terms.get(i);
78     }
79
80     public boolean contains(Term term) {
81         for (Term thisterm : this) {
82             if (term.getLabel().compareTo(thisterm.getLabel()) == 0)
83                 return true;
84         }
85         return false;
86     }
87 }

```

```

84
85     public int getNbConstants() {
86         int result = 0;
87         for (Term term : this)
88             if (term.isConstant())
89                 result++;
90         return result;
91     }
92
93     public ArrayList<Term> constants() {
94         ArrayList<Term> result = new ArrayList<Term>();
95         for (Term term : this)
96             if (term.isConstant())
97                 result.add(term);
98         return result;
99     }
100
101     public int getNbVariables() {
102         int result = 0;
103         for (Term term : this)
104             if (term.isVariable())
105                 result++;
106         return result;
107     }
108
109     public ArrayList<Term> variables() {
110         ArrayList<Term> result = new ArrayList<Term>();
111         for (Term term : this)
112             if (term.isVariable())
113                 result.add(term);
114         return result;
115     }
116
117     /**
118      * Term setter.
119      * @param i The term index inside the atom.
120      * @param t The new term to be set instead of the old one.
121      * @throws NoSuchElementException If i is out of range.
122      */
123     public void set(int i, Term t) throws NoSuchElementException {
124         if ((i >= getArity()) || (i < 0))
125             throw new NoSuchElementException();
126         _terms.set(i, t);
127     }
128
129     /**
130      * Converts the atom into a String.
131      */
132     @Override
133     public String toString() {
134         StringBuilder string = new StringBuilder(getPredicate().getLabel());
135         string.append(BEGIN_TERM_LIST_W);
136         for (int i = 0 ; i < getArity()-1 ; i++) {
137             string.append(get(i));
138             string.append(TERM_SEPARATOR);
139         }
140         if (getArity() >= 1)
141             string.append(get(getArity()-1));
142         string.append(END_TERM_LIST_W);
143         return string.toString();
144     }
145
146     /**
147      * Access to the nested Iterator over terms instance.
148      * @return The iterator over terms corresponding to this atom instance.
149      */
150     public java.util.Iterator<Term> iterator() {
151         return new Iterator();
152     }
153
154     /** Predicate of the atom. */
155     private Predicate _predicate;
156     /** List of terms. */
157     private ArrayList<Term> _terms;
158
159     /**
160      * Nested class used to iterate over the terms.<br />
161      * Note that the remove() operation is not supported, and will always throw an  $\leftrightarrow$ 
162      * UnsupportedOperationException.
163      */
164     public class Iterator implements java.util.Iterator<Term> {
165         /**
166          * Allows to know if there is still element to be iterated.
167          * @return True if the next call to next() method will not throw a  $\leftrightarrow$ 
168          *         NoSuchElementException, false otherwise.
169          */
170         @Override
171         public boolean hasNext() {

```

```

169         return (_current < getArity());
170     }
171     /**
172     * Access to the next element.
173     * @return The next term of the atom.
174     * @throws NoSuchElementException If there is no more term to be iterated.
175     */
176     @Override
177     public Term next() throws NoSuchElementException {
178         if (!hasNext())
179             throw new NoSuchElementException();
180         Term t = get(_current);
181         _current++;
182         return t;
183     }
184     /**
185     * Unsupported operation.
186     * @throws UnsupportedOperationException Whenever this method is called.
187     */
188     public void remove() throws UnsupportedOperationException {
189         throw new UnsupportedOperationException();
190     }
191     /** Current term index. */
192     private int _current = 0;
193 };
194
195 };

```



```

1 package obr;
2
3 import moca.graphs.BipartedGraph;
4 import moca.graphs.vertices.VertexCollection;
5 import moca.graphs.vertices.Vertex;
6 import moca.graphs.vertices.VertexArrayList;
7 import moca.graphs.edges.Edge;
8 import moca.graphs.edges.NeighbourEdge;
9 import moca.graphs.edges.UndirectedNeighboursLists;
10 import moca.graphs.edges.IllegalEdgeException;
11
12 import java.lang.Iterable;
13 import java.lang.String;
14 import java.lang.Integer;
15
16 import java.util.Iterator;
17 import java.util.NoSuchElementException;
18
19
20 /**
21  * Represents a conjunction of atoms.<br />
22  *
23  * <p>
24  * <b>Internal structure :</b><br />
25  * The internal structure is a biparted graph where edges are implemented by neighbours lists<br />
26  * The first partition of vertices represents the atoms. Only their predicates are stored<br />
27  * into this partition.
28  * But the vertex class provides a global ID which will can be used to identify an atom.<br />
29  * The second partition represents the terms. A term may be either a variable or a constant.
30  * Since terms are also stored into vertices, they have a global ID.<br />
31  * The edges are used to connect atoms with terms. And their value is the position of the<br />
32  * term inside the atom.
33  * A well-built atom conjunction will ensure that there is one edge for each position of each<br />
34  * atom predicate and that
35  * they are added in their position ordering.<br />
36  * </p>
37  *
38  * <p>
39  * <b>About IDs :</b><br />
40  * Each vertex (thus, each atom and each term) has a different ID, which can be get by Vertex<br />
41  * .getID() method.
42  * But since the internal structure provides a bipartition, atoms can be enumerated from 0 to<br />
43  * getNbAtoms()-1, and terms
44  * from 0 to getNbTerms()-1.<br />
45  * To convert a global ID to a local one apply the following computation :
46  * <ul><li>if atom : local ID = global ID</li>
47  * <li>if term : local ID = global ID - getNbAtoms()</li></ul>
48  * </p>
49  *
50  */
51 public class AtomConjunction implements Iterable<Atom> {
52
53     /**
54      * Represents the separator between atoms when the conjunction is under a String format.
55      */
56     public static final String ATOM_SEPARATOR = new String(";");
57
58     /**
59      * Represents the separator between terms when an atom is under a String format.
60      * @see Atom#TERM_SEPARATOR
61      */
62     public static final String TERM_SEPARATOR = Atom.TERM_SEPARATOR;
63
64     /**
65      * Represents the begin of a term list when an atom is under a String format.
66      * @see Atom#BEGIN_TERM_LIST
67      */
68     public static final String BEGIN_TERM_LIST = Atom.BEGIN_TERM_LIST;
69
70     /**
71      * Represents the end of a term list when an atom is under a String format.
72      * @see Atom#END_TERM_LIST
73      */
74     public static final char END_TERM_LIST = Atom.END_TERM_LIST.charAt(Atom.
75     END_TERM_LIST.length()-1);
76
77     /**
78      * Default constructor.<br />
79      * The atom conjunction will be empty.
80      */
81     public AtomConjunction() {
82         try {
83             _graph = new BipartedGraph<Object,Integer>(
84                 new VertexArrayList<Object>(),

```

```

79         new VertexArrayList<Object>(),
80         new UndirectedNeighboursLists<Integer>());
81     }
82     catch (Exception e) { }
83 }
84
85 /**
86  * Constructor from a string representation of an atom conjunction.
87  * @param stringRepresentation The atom conjunction under a String format.
88  * @see #fromString(String)
89  */
90 public AtomConjunction(String stringRepresentation) {
91     try {
92         _graph = new BipartedGraph<Object, Integer>(
93             new VertexArrayList<Object>(),
94             new VertexArrayList<Object>(),
95             new UndirectedNeighboursLists<Integer>());
96         fromString(stringRepresentation);
97     }
98     catch (Exception e) { }
99 }
100
101 /**
102  * Copy constructor.
103  * @param toCopy The atom conjunction to be copied.
104  */
105 public AtomConjunction(AtomConjunction toCopy) {
106     try {
107         _graph = new BipartedGraph<Object, Integer>(toCopy._graph);
108     }
109     catch (Exception e) { }
110 }
111
112 /**
113  * Number of atoms getter.
114  * @return The current number of atoms.
115  */
116 public int getNbAtoms() {
117     return _graph.getNbVerticesInFirstSet();
118 }
119
120 /**
121  * Number of terms getter.
122  * @return The current number of terms.
123  */
124 public int getNbTerms() {
125     return _graph.getNbVerticesInSecondSet();
126 }
127
128 /**
129  * Allows to know if a vertex is an atom.
130  * @param v The vertex to be checked.
131  * @return True if v is an atom, false otherwise.
132  */
133 public boolean isAtom(Vertex<Object> v) {
134     if (v == null)
135         return false;
136     if (v.getID() < getNbAtoms())
137         return true;
138     return false;
139 }
140
141 /**
142  * Allows to know if a vertex is a term.
143  * @param v The vertex to be checked.
144  * @return True if v is a term, false otherwise.
145  */
146 public boolean isTerm(Vertex<Object> v) {
147     if (v == null)
148         return false;
149     if (v.getID() < getNbAtoms())
150         return false;
151     return true;
152 }
153
154 /**
155  * Creates a new atom instance for the atom whose id matches i.<br />
156  * The structure does not contain atom instances, that is why only a new instance can be ←
157  * generated.
158  * Furthermore, any modifications on this atom instance will <b>NOT</b> be applied on the ←
159  * atom representation in the
160  * conjunction.
161  * @param i The id of the atom.
162  * @return A new instance of atom class.
163  * @throws NoSuchElementException If the id does not match in the atom conjunction.
164  */

```

```

164 public Atom getAtom(int i) throws NoSuchElementException {
165     if ((i >= getNbAtoms()) || (i < 0))
166         throw new NoSuchElementException();
167     Atom atom = new Atom((Predicate)(_graph.getVertex(i).getValue()));
168     for (Iterator<NeighbourEdge<Integer>> iterator = _graph.neighbourIterator(i) ; ↵
169         iterator.hasNext() ; ) {
170         NeighbourEdge<Integer> edge = iterator.next();
171         atom.set(edge.getValue(), (Term)(_graph.getVertex(edge.getIDV()).getValue()));
172     }
173     return atom;
174 }
175 /**
176  * Fullfills the atom conjunction from its string representation.<br />
177  * The string must be under the following format (otherwise an exception will be thrown) ↵
178  * :
179  * <code>&lt;atom 0>&gt;ATOM_SEPARATOR&lt;atom 1>&gt;ATOM_SEPARATOR...ATOM_SEPARATOR&lt;↵
180  * atom n>&gt;</code>
181  * Where atoms must be under the following format :
182  * <code>&lt;predicate label>&gt;BEGIN_TERM_LIST&lt;term 0>&gt;TERM_SEPARATOR&lt;term 1>&gt;↵
183  * TERM_SEPARATOR...TERM_SEPARATOR
184  * &lt;term n>&gt;END_TERM_LIST</code><br />
185  * The predicate labels and arity are used to check their equality, and only labels for ↵
186  * terms equality.
187  * @param str The atom conjunction under a string format.
188  * @throws UnrecognizedStringFormatException Whenever the string passed as parameter does↵
189  * not follow a good format.
190 */
191 protected void fromString(String str) throws UnrecognizedStringFormatException {
192     String[] sub1 = null;
193     String[] sub2 = null;
194     String[] sub3 = null;
195     sub1 = str.split(ATOM_SEPARATOR);
196     Predicate predicate = null;
197     int arity = 0;
198     int atomID = -1;
199     int termID = -1;
200     for (int i = 0 ; i < sub1.length ; i++) {
201         sub2 = sub1[i].split(BEGIN_TERM_LIST);
202         switch (sub2.length) {
203             case 1 : // if there is no begin term list char we suppose the predicate ↵
204                 arity to be null
205                 arity = 0;
206                 break;
207             case 2 :
208                 if (sub2[1].charAt(sub2[1].length()-1) != END_TERM_LIST)
209                     throw new UnrecognizedStringFormatException();
210                 sub2[1] = sub2[1].substring(0, sub2[1].length()-1); // the last char is ↵
211                 to be removed
212                 sub3 = sub2[1].split(TERM_SEPARATOR);
213                 arity = sub3.length;
214                 break;
215             default :
216                 throw new UnrecognizedStringFormatException();
217         }
218         atomID = addAtom(new Predicate(sub2[0], arity));
219         for (int j = 0 ; j < arity ; j++) {
220             termID = addTerm(sub3[j]);
221             try {
222                 _graph.addEdge(atomID, termID, new Integer(j));
223             }
224             catch (Exception e) {
225                 throw new UnrecognizedStringFormatException();
226             }
227         }
228     }
229 }
230 /**
231  * Converts the atom conjunction into a String format.<br />
232  * Note that this is just a convenient method for toString(getNbAtoms()).
233  * @return A well-formatted string representation.
234  * @see #toString(int)
235 */
236 public String toString() {
237     return toString(getNbAtoms());
238 }
239 /**
240  * Converts a subset of the atom conjunction into a String format.<br />
241  * Only atoms whose id is between 0 and the parameter will be converted.
242  * This method is usefull for inheritance which specializes some of atoms.
243  * @param nbAtoms The number of atoms to be converted.
244 */
245 public String toString(int nbAtoms) {
246     StringBuilder string = new StringBuilder();
247     if (nbAtoms > getNbAtoms())

```

```

243     nbAtoms = getNbAtoms();
244     for (int i = 0 ; i < nbAtoms - 1; i++) {
245         string.append(getAtom(i));
246         string.append(ATOM_SEPARATOR);
247     }
248     string.append(getAtom(nbAtoms-1));
249     return string.toString();
250 }
251 public String toStringExcluding(int exclude) {
252     StringBuilder string = new StringBuilder();
253     final int nbAtoms = getNbAtoms();
254     for (int i = 0 ; i < nbAtoms; i++) {
255         if (i != exclude) {
256             string.append(getAtom(i));
257             string.append(ATOM_SEPARATOR);
258         }
259     }
260     string.deleteCharAt(string.length()-1);
261     return string.toString();
262 }
263
264
265
266 /**
267  * Adds an atom into the atom conjunction.<br />
268  * The internal representation of an atom is a graph vertex with predicate value, that is ←
269  * why only a predicate is
270  * needed.
271  * @param predicate The predicate of the new atom.
272  * @return The global index of the new atom in the conjunction.
273  */
274 public int addAtom(Predicate predicate) {
275     _graph.addInFirstSet(predicate);
276     return _graph.getNbVerticesInFirstSet() - 1;
277 }
278
279 public void removeCleanlyAtom(int atomID) throws NoSuchElementException {
280     final Vertex<Object> atom = getVertexAtom(atomID);
281     int neighbourAtomID;
282     int currentTerm;
283     final int nbTerms = _graph.getNbNeighbours(atomID);
284     for (currentTerm = 0 ; currentTerm < nbTerms ; currentTerm++) {
285         neighbourAtomID = getVertexTermFromAtom(atomID, currentTerm).getID();
286         if (_graph.getNbNeighbours(neighbourAtomID) == 1) {
287             _graph.removeVertex(neighbourAtomID);
288             currentTerm--;
289         }
290     }
291     removeAtom(atomID);
292 }
293
294 protected void removeAtom(int atomID) throws NoSuchElementException {
295     _graph.removeVertex(atomID);
296 }
297
298 public int getNbConstants() {
299     int result = 0;
300     final int nbTerms = getNbTerms();
301     for (int i = 0 ; i < nbTerms ; i++)
302         if (getTerm(i).isConstant())
303             result++;
304     return result;
305 }
306
307 public int getNbVariables() {
308     int result = 0;
309     final int nbTerms = getNbTerms();
310     for (int i = 0 ; i < nbTerms ; i++)
311         if (getTerm(i).isVariable())
312             result++;
313     return result;
314 }
315
316 public VertexCollection<Object> constants() {
317     VertexArrayList<Object> result = new VertexArrayList<Object>();
318     final int nbTerms = getNbTerms();
319     for (int i = 0 ; i < nbTerms ; i++)
320         if (getTerm(i).isConstant())
321             result.add(getVertexTerm(i));
322     return result;
323 }
324
325 public VertexCollection<Object> variables() {
326     VertexArrayList<Object> result = new VertexArrayList<Object>();
327     final int nbTerms = getNbTerms();
328     for (int i = 0 ; i < nbTerms ; i++)
329         if (getTerm(i).isVariable())

```

```

329         result.add(getVertexTerm(i));
330     return result;
331 }
332
333 public VertexCollection<Object> domain(){
334     return _graph.getSecondSet();
335 }
336
337 public VertexCollection<Object> constants(int atomID) {
338     if ((atomID > getNbAtoms()) || (atomID < 0))
339         return null;
340     VertexArrayList<Object> result = new VertexArrayList<Object>();
341     Vertex<Object> term;
342     for (NeighbourIterator iterator = vertexTermIteratorFromAtom(atomID) ; iterator.hasNext() ; ) {
343         term = iterator.next();
344         if (((Term)(term.getValue())).isConstant())
345             && (!result.contains(term))
346             result.add(term);
347     }
348     return result;
349 }
350
351 public VertexCollection<Object> variables(int atomID) {
352     if ((atomID > getNbAtoms()) || (atomID < 0))
353         return null;
354     VertexArrayList<Object> result = new VertexArrayList<Object>();
355     Vertex<Object> term;
356     for (NeighbourIterator iterator = vertexTermIteratorFromAtom(atomID) ; iterator.hasNext() ; ) {
357         term = iterator.next();
358         if (((Term)(term.getValue())).isVariable())
359             && (!result.contains(term))
360             result.add(term);
361     }
362     return result;
363 }
364
365 /**
366  * Adds a term into the atom conjunction.<br />
367  * Note that the term will not be connected to any atom. Use addEdge method to make these
368  * connections.<br />
369  * Furthermore, if the term is already in the conjunction, no new term will be added.
370  * @param term The new term to add.
371  * @return The term index.
372  */
373 public int addTerm(Term term) {
374     for (int i = getNbAtoms() ; i < _graph.getNbVertices() ; i++) {
375         if (((Term)(getVertex(i).getValue())).getLabel().compareTo(term.getLabel()) == 0) {
376             return i;
377         }
378     }
379     _graph.addInSecondSet(term);
380     return _graph.getNbVertices() - 1;
381 }
382
383 /**
384  * Adds a term into the atom conjunction from its label.
385  * @param termLabel The new term label to add.
386  * @return The term index.
387  * @see #addTerm(Term)
388  */
389 public int addTerm(String termLabel) {
390     for (int i = getNbAtoms() ; i < _graph.getNbVertices() ; i++) {
391         if (((Term)(getVertex(i).getValue())).getLabel().compareTo(termLabel) == 0) {
392             return i;
393         }
394     }
395     _graph.addInSecondSet(new Term(termLabel));
396     return _graph.getNbVertices() - 1;
397 }
398
399 /**
400  * Adds an edge between an atom and a term.<br />
401  * If the arity of the atom id is n, then the position must be in [0;n-1].
402  * @param atomID The id of the atom to connect.
403  * @param termID The id of the term to connect.
404  * @param position The position of the term in the atom.
405  * @return True if success, false otherwise.
406  */
407 public boolean addEdge(int atomID, int termID, Integer position) {
408     try {
409         _graph.addEdge(atomID, termID, position);
410         return true;
411     }

```

```

412         catch (Exception e) {
413             return false;
414         }
415     }
416
417     /**
418     * Add an atom from another atom conjunction.<br />
419     * If the first parameter is not really an atom vertex, this method will silently fail.
420     * @param atom The atom vertex to add.
421     * @param source The atom conjunction where to find atom vertex connections.
422     */
423     public void addAtom(Vertex<Object> atom, AtomConjunction source) {
424         if (source.isAtom(atom)) {
425             Iterator<NeighbourEdge<Integer>> termIterator = source.neighbourIterator(atom.getID());
426             NeighbourEdge<Integer> edge = null;
427             int termID = -1;
428             addAtom((Predicate)(atom.getValue()));
429             while (termIterator.hasNext()) {
430                 edge = termIterator.next();
431                 termID = addTerm(source.getTerm(edge.getIDV() - source.getNbAtoms()));
432                 addEdge(getNbAtoms() - 1, termID, new Integer(edge.getValue()));
433             }
434         }
435     }
436
437     /**
438     * Access to the value of a vertex from its global ID.
439     * @param i The vertex global id.
440     * @return The value of the vertex.
441     * @throws NoSuchElementException If i does not match in the atom conjunction.
442     */
443     public Object get(int i) throws NoSuchElementException {
444         return _graph.get(i);
445     }
446
447     /**
448     * Access to the predicate at the specified index.<br />
449     * Convenient method for getVertexAtom(int).
450     * @param i The predicate vertex local id.
451     * @return The predicate.
452     * @throws NoSuchElementException If i does not match in the atom conjunction.
453     * @see #getVertexAtom(int)
454     */
455     public Predicate getPredicate(int i) throws NoSuchElementException {
456         return (Predicate)(getVertexAtom(i).getValue());
457     }
458
459     /**
460     * Access to a vertex from its global ID.
461     * @param i The vertex global id.
462     * @return The vertex whose id matches i.
463     * @throws NoSuchElementException If i does not match in the atom conjunction.
464     */
465     public Vertex<Object> getVertex(int i) throws NoSuchElementException {
466         return _graph.getVertex(i);
467     }
468
469     /**
470     * Access to a vertex term from its local ID.
471     * @param i The vertex term local id.
472     * @return The vertex term whose local id matches i.
473     * @throws NoSuchElementException If i does not match any vertex term local id.
474     */
475     public Vertex<Object> getVertexTerm(int i) throws NoSuchElementException {
476         return _graph.getInSecondSet(i);
477     }
478
479     /**
480     * Access to a vertex atom from its local ID.
481     * @param i The vertex atom local id.
482     * @return The vertex atom whose local id matches i.
483     * @throws NoSuchElementException If i does not match any vertex atom local id.
484     */
485     public Vertex<Object> getVertexAtom(int i) throws NoSuchElementException {
486         return _graph.getInFirstSet(i);
487     }
488
489     /**
490     * Access to a term from its local ID. <br />
491     * Convenient method for getVertexTerm(int).
492     * @param i The vertex term local id.
493     * @return The term whose vertex local id matches i.
494     * @throws NoSuchElementException If i does not match any vertex term local id.
495     */
496     public Term getTerm(int i) throws NoSuchElementException {
497         return (Term)getVertexTerm(i).getValue();

```

```

498     }
499
500     /**
501     * Access to a term from an atom and its position into this atom.<br />
502     * @param atomID The id of the atom where to get the term.
503     * @param termIndex The position of the term inside the atom.
504     * @return The term corresponding.
505     * @throws NoSuchElementException If atomID does not match, or if termIndex is greater ←
506     *         than or equals to the
507     *         corresponding predicate arity.
508     */
509     public Term getTermFromAtom(int atomID, int termIndex) throws NoSuchElementException {
510         return (Term)_graph.getNeighbourValue(atomID,termIndex);
511     }
512
513     /**
514     * Access to a vertex term from an atom and its position into this atom.<br />
515     * @param atomID The id of the atom where to get the term.
516     * @param termIndex The position of the term inside the atom.
517     * @return The vertex term corresponding.
518     * @throws NoSuchElementException If atomID does not match, or if termIndex is greater ←
519     *         than or equals to the
520     *         corresponding predicate arity.
521     */
522     public Vertex<Object> getVertexTermFromAtom(int atomID, int termIndex) throws ←
523         NoSuchElementException {
524         return _graph.getNeighbour(atomID,termIndex);
525     }
526
527     /**
528     * Returns a new atom conjunction which is a clone of a subset of the current atom ←
529     * conjunction.<br />
530     * The two parameters will be used as the range of atoms to be used.
531     * All terms connected to one of these atoms will also be added into the clone.
532     * Thus, all edges which were connected to the atoms in the current atom conjunction will ←
533     * also be present in the
534     * clone.<br />
535     * Note that the first atom will be included but not the last. I.e., if this method is ←
536     * used on 0,getNbAtom(), all
537     * the atom conjunction will be cloned.
538     * @param beginAtomID The ID of the first atom (included).
539     * @param endAtomID The ID of the last atom (not included).
540     * @return The new atom conjunction if the parameters are not absurds, false otherwise.
541     */
542     public AtomConjunction subAtomConjunction(int beginAtomID, int endAtomID) {
543         if ((beginAtomID > endAtomID) || (endAtomID > getNbAtoms()) || (beginAtomID < 0))
544             return null;
545         AtomConjunction result = new AtomConjunction();
546         NeighbourEdge<Integer> edge = null;
547         int termID = -1;
548         for (int i = beginAtomID ; i < endAtomID ; i++) {
549             result._graph.addInFirstSet(getPredicate(i).clone());
550         }
551         for (int i = beginAtomID ; i < endAtomID ; i++) {
552             for (Iterator<NeighbourEdge<Integer> > iterator = _graph.neighbourIterator(i) ; ←
553                 iterator.hasNext() ; ) {
554                 edge = iterator.next();
555                 termID = result.addTerm(getTerm(edge.getIDV())-getNbAtoms()).clone();
556                 try { result._graph.addEdge(i-beginAtomID,termID,new Integer(edge.getValue()))←
557                     }; }
558                 catch (IllegalEdgeException e) { }
559             }
560         }
561         return result;
562     }
563
564     /**
565     * Clones the atom conjunction into a copy instance.
566     * @param copy The instance where to clone.
567     * @return The clone.
568     */
569     public AtomConjunction cloneIn(AtomConjunction copy) {
570         for (Iterator<Object> iterator = _graph.firstIterator() ; iterator.hasNext() ; ) {
571             copy.addAtom(((Predicate)(iterator.next())).clone());
572         }
573         for (Iterator<Object> iterator = _graph.secondIterator() ; iterator.hasNext() ; ) {
574             copy.addTerm(((Term)(iterator.next())).clone());
575         }
576         Edge<Integer> edge = null;
577         for (Iterator<Edge<Integer> > iterator = _graph.edgeIterator() ; iterator.hasNext() ;←
578             ) {
579             edge = iterator.next();
580             copy.addEdge(edge.getIDU(),edge.getIDV(),new Integer(edge.getValue()));
581         }
582         return copy;
583     }
584
585
586

```

```

576  /**
577   * Clones the atom conjunction.<br />
578   * Convenient method for cloneIn(new AtomConjunction()).
579   * @return A clone of the atom conjunction.
580   */
581  @Override
582  public AtomConjunction clone() {
583      return cloneIn(new AtomConjunction());
584  }
585
586  /**
587   * Access to the iterator over the atoms.
588   * @return An iterator over the atoms.
589   * @see .AtomIterator
590   */
591  public AtomIterator iterator() {
592      return new AtomIterator();
593  }
594
595  /**
596   * Access to the iterator over the vertex atoms.
597   * @return An iterator over the vertices representing atoms.
598   */
599  public Iterator<Vertex<Object>> vertexAtomIterator() {
600      return _graph.firstVertexIterator();
601  }
602
603  /**
604   * Access to the iterator over the vertex terms.
605   * @return An iterator over the vertices representing terms.
606   */
607  public Iterator<Vertex<Object>> vertexTermIterator() {
608      return _graph.secondVertexIterator();
609  }
610
611  /**
612   * Access to an iterator over the vertex terms which are connected to the specified atom<←
613   * .<br />
614   * If the atom conjunction is well built, the terms will be iterated in their positions <←
615   * ordering.
616   * @param atomID The global id of the atom.
617   * @return An iterator over the vertex terms connected to the atom.
618   * @see .NeighbourIterator
619   */
620  public NeighbourIterator vertexTermIteratorFromAtom(int atomID) {
621      return new NeighbourIterator(_graph.neighbourIterator(atomID));
622  }
623
624  /**
625   * Access to an iterator over the vertex atoms which are connected to the specified term.
626   * @param termID The global id of the term.
627   * @return An iterator over the vertex atoms connected to the term.
628   * @see .NeighbourIterator
629   */
630  public NeighbourIterator vertexAtomIteratorFromTerm(int termID) {
631      return new NeighbourIterator(_graph.neighbourIterator(termID));
632  }
633
634  /**
635   * Access to an iterator over the edges from a specific vertex.
636   * @param vertexID The global id of the vertex which is the source of edges.
637   * @return An iterator over the edges connected to the vertex.
638   */
639  public Iterator<NeighbourEdge<Integer>> neighbourIterator(int vertexID) {
640      return _graph.neighbourIterator(vertexID);
641  }
642
643  /**
644   * The structure of the atom conjunction.<br />
645   * The first vertices partition is used to store atoms (the vertices contain only <←
646   * predicates).
647   * And the second one to store terms.
648   * An edge represents a connection between an atom and a term, and its value the position<←
649   * of the term in the atom.
650   */
651  protected BipartiteGraph<Object, Integer> _graph = null;
652
653  /**
654   * Iterator over the atoms of the conjunction.<br />
655   * Since atoms are not stored into the internal structure, the remove operation is not <←
656   * supported.
657   */
658  public class AtomIterator implements Iterator<Atom> {
659
660      /**
661       * Allows to know if the next call to next() will fail.

```



```

658     * @return True if the next call to next() will not fail, false otherwise.
659     */
660     public boolean hasNext() {
661         return _current < getNbAtoms();
662     }
663
664     /**
665     * Access to the next atom.
666     * @return The next atom.
667     * @throws NoSuchElementException If there is no more atom to be iterated over.
668     */
669     public Atom next() throws NoSuchElementException {
670         if (!hasNext())
671             throw new NoSuchElementException();
672         Atom result = getAtom(_current);
673         _current++;
674         return result;
675     }
676
677     /**
678     * This operation is not supported.
679     * @throws UnsupportedOperationException Always.
680     */
681     public void remove() throws UnsupportedOperationException { throw new ←
        UnsupportedOperationException(); }
682
683     /** Current atom index. */
684     protected int _current = 0;
685
686 }
687
688
689 /**
690 * Iterator over the vertices connected to a specific one.
691 */
692 protected class NeighbourIterator implements Iterator<Vertex<Object>> {
693
694     /**
695     * Constructor.
696     * @param iterator The iterator over the neighbour edges.
697     */
698     public NeighbourIterator(Iterator<NeighbourEdge<Integer>> iterator) {
699         _iterator = iterator;
700     }
701
702     /**
703     * Allows to know if the next call to next() method will not fail.
704     * @return True if the next call to next() will not fail, false otherwise.
705     */
706     @Override
707     public boolean hasNext() {
708         return _iterator.hasNext();
709     }
710
711     /**
712     * Iterates and returns the next vertex.
713     * @return The next vertex.
714     * @throws NoSuchElementException If there is no more vertex to be iterated.
715     */
716     @Override
717     public Vertex<Object> next() throws NoSuchElementException {
718         return getVertex(_iterator.next().getIDV());
719     }
720
721     /**
722     * This operation is not supported.
723     * @throws UnsupportedOperationException Always.
724     */
725     @Override
726     public void remove() throws UnsupportedOperationException { throw new ←
        UnsupportedOperationException(); }
727
728     /** An iterator over the neighbour edges. */
729     private Iterator<NeighbourEdge<Integer>> _iterator;
730
731 };
732
733
734 };

```

```

1 package obr;
2
3 import moca.graphs.IllegalConstructionException;
4 import moca.graphs.vertices.Vertex;
5 import moca.graphs.edges.NeighbourEdge;
6 import moca.graphs.edges.IllegalEdgeException;
7
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.NoSuchElementException;
11
12 /**
13  * Checks if all rules in the GRD or in one of its strongly connected components satisfy the  $\leftrightarrow$ 
14  * atomic-hypothesis property.
15  * I.e., if all rules contain only one atom in their body.
16  */
17 public class AtomicHypothesisCheck implements DecidableClassCheck {
18     /**
19      * Atomic-hypothesis property label.
20      */
21     public static final DecidableClassLabel LABEL = new DecidableClassLabel("atomic- $\leftrightarrow$ 
22         hypothesis", true, false, true);
23
24     /**
25      * Checks a graph of rule dependencies.
26      * @return the decidable class label if the grd belongs to this decidable class, null  $\leftrightarrow$ 
27      * otherwise.
28      */
29     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
30         if (check(grd.getVertexCollection()))
31             return LABEL;
32         return null;
33     }
34
35     /**
36      * Checks only a strongly connected component of the graph of rule dependencies.
37      * @param grd The graph of rule dependencies.
38      * @param sccID The id of the strongly connected component to be checked.
39      * @return The decidable class label if the strongly connected component belongs to this  $\leftrightarrow$ 
40      * decidable class, null otherwise.
41      */
42     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
43         if (check(grd.getComponent(sccID)))
44             return LABEL;
45         return null;
46     }
47
48     /**
49      * Internal check.
50      * @return True if the set of rules satisfies the atomic-hypothesis property, false  $\leftrightarrow$ 
51      * otherwise.
52      */
53     protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
54         for (Vertex<AtomicRule> vrule : rules) {
55             if (vrule.getValue().getNbAtoms() > 2) // one for the head, the other for the  $\leftrightarrow$ 
56                 body
57                 return false;
58         }
59         return true;
60     }
61 }

```

```

1 package obr;
2
3 import moca.graphs.BipartiteGraph;
4 import moca.graphs.vertices.VertexCollection;
5 import moca.graphs.vertices.Vertex;
6 import moca.graphs.vertices.VertexArrayList;
7 import moca.graphs.edges.NeighbourEdge;
8 import moca.graphs.edges.UndirectedNeighboursLists;
9 import moca.graphs.edges.IllegalEdgeException;
10 import moca.lists.Fifo;
11
12 import java.lang.String;
13 import java.lang.Integer;
14
15 import java.util.Iterator;
16 import java.util.NoSuchElementException;
17 import java.util.ArrayList;
18 import java.util.ConcurrentModificationException;
19
20 /**
21  * Represents an atomic rule.<br />
22  * A rule is an atom conjunction which adds the notions of "body" and "head".<br />
23  * An atomic rule contains only one atom in its head.
24  * The structure of this class is the same as a classical atom conjunction but its head  $\leftrightarrow$ 
25  * pointer.
26  * @see AtomConjunction
27  */
28 public class AtomicRule extends AtomConjunction {
29
30     /** Represents the separator between the body and the head when the rule is under a  $\leftrightarrow$ 
31         String format. */
32     public static final String HEAD_SEPARATOR = new String("→");
33
34     /**
35      * Default constructor.
36      * It creates an empty rule.
37      */
38     public AtomicRule() {
39         super();
40     }
41
42     /**
43      * Constructor from a string representation.
44      * @param stringRepresentation The rule under a string format.
45      * @see #fromString(String)
46      */
47     public AtomicRule(String stringRepresentation) {
48         super();
49         try {
50             fromString(stringRepresentation);
51         }
52         catch (UnrecognizedStringFormatException e) { }
53     }
54
55     /**
56      * Head vertex getter.
57      * @return The head vertex.
58      */
59     public Vertex<Object> getHead() {
60         return _head;
61     }
62
63     public void setHead(int atomID) {
64         _head = getVertex(atomID);
65     }
66
67     /**
68      * Converts the head into an atom instance.
69      * @see AtomConjunction#getAtom(int)
70      */
71     public Atom getHeadAtom() {
72         return getAtom(_head.getID());
73     }
74
75     /**
76      * Allows to know if a vertex is in the body.
77      * @param v The vertex to be checked.
78      * @return True if v belongs to the body, false otherwise.
79      * @throws NoSuchElementException If v is null or if its id does not match in the rule.
80      */
81     public boolean isBody(Vertex<Object> v) throws NoSuchElementException {
82         if (v == null)
83             throw new NoSuchElementException();
84         if (v == _head)
85             return false;
86         if (v.getID() < getNbAtoms())

```

```

85         return true;
86         for (Iterator<NeighbourEdge<Integer> > iterator = _graph.neighbourIterator(v.getID())↵
            ; iterator.hasNext() ; )
87             if (_graph.getVertex(iterator.next().getIDV()) != _head)
88                 return true;
89         return false;
90     }
91
92     /**
93     * Convenient method for isBody(Vertex<Object>).
94     * @param vertexID The id of the vertex to be checked.
95     * @return True if the vertex belongs to the body, false otherwise.
96     * @throws NoSuchElementException If the id does not match in the rule.
97     * @see #isBody(Vertex)
98     */
99     public boolean isBody(int vertexID) throws NoSuchElementException {
100         return isBody(getVertex(vertexID));
101     }
102
103     /**
104     * Allows to know if a vertex is in the head.
105     * @param v The vertex to be checked.
106     * @return True if v belongs to the head, false otherwise.
107     * @throws NoSuchElementException If v is null or if its id does not match in the rule.
108     */
109     public boolean isHead(Vertex<Object> v) throws NoSuchElementException {
110         if (v == null)
111             throw new NoSuchElementException();
112         if (v == _head)
113             return true;
114         for (Iterator<NeighbourEdge<Integer> > iterator = _graph.neighbourIterator(_head.↵
            getID()) ; iterator.hasNext() ; )
115             if (_graph.getVertex(iterator.next().getIDV()) == v)
116                 return true;
117         return false;
118     }
119
120     /**
121     * Convenient method for isHead(Vertex<Object>).
122     * @param vertexID the id of the vertex to be checked.
123     * @return True if the vertex belongs to the head, false otherwise.
124     * @throws NoSuchElementException If the id does not match in the rule.
125     * @see #isHead(Vertex)
126     */
127     public boolean isHead(int vertexID) throws NoSuchElementException {
128         return isHead(getVertex(vertexID));
129     }
130
131     /**
132     * Creates a new array list containing all vertices which belong to the frontier of the
133     * rule.
134     * A variable is said to be in the frontier iff it belongs to the body <b>and</b> to the
135     * head of the rule.
136     * @return An array list of Vertex<Object> which contains all variables in the
137     * frontier.
138     */
139     public ArrayList<Vertex<Object> > frontier() {
140         NeighbourEdge<Integer> edge = null;
141         ArrayList<Vertex<Object> > result = new ArrayList<Vertex<Object> >();
142         for (Iterator<NeighbourEdge<Integer> > iterator = neighbourIterator(_head.getID()) ; ↵
            iterator.hasNext() ; ) {
143             edge = iterator.next();
144             if ((getTerm(edge.getIDV())-getNbAtoms()).isVariable())
145                 && (isBody(edge.getIDV()))
146                 && (!result.contains(getVertex(edge.getIDV())))
147             )
148                 result.add(getVertex(edge.getIDV()));
149         }
150         return result;
151     }
152
153     /**
154     * Creates a new array list containing all vertices which belong to the frontier of the ↵
155     * rule.
156     * A term is said to be in the frontier iff it belongs to the body <b>and</b> to the head↵
157     * of the rule.
158     * @return An array list of Vertex<Object> which contains all terms in the frontier↵
159     */
160     public ArrayList<Vertex<Object> > frontierTerms() {
161         NeighbourEdge<Integer> edge = null;
162         ArrayList<Vertex<Object> > result = new ArrayList<Vertex<Object> >();
163         for (Iterator<NeighbourEdge<Integer> > iterator = neighbourIterator(_head.getID()) ; ↵
            iterator.hasNext() ; ) {
164             edge = iterator.next();
165             if ((isBody(edge.getIDV())) && (!result.contains(getVertex(edge.getIDV()))))
166                 result.add(getVertex(edge.getIDV()));

```

```

165     }
166     return result;
167 }
168
169 public int getNbUniversalVariables() {
170     int result = 0;
171     final int nbTerms = getNbTerms();
172     for (int i = 0 ; i < nbTerms ; i++)
173         if (isUniversal(i))
174             result++;
175     return result;
176 }
177
178 public int getNbExistentialVariables() {
179     int result = 0;
180     final int nbTerms = getNbTerms();
181     for (int i = 0 ; i < nbTerms ; i++)
182         if (isExistential(i))
183             result++;
184     return result;
185 }
186
187 public VertexCollection<Object> universalVariables() {
188     VertexArrayList<Object> result = new VertexArrayList<Object>();
189     final int nbTerms = getNbTerms();
190     for (int i = 0 ; i < nbTerms ; i++)
191         if (isUniversal(i))
192             result.add(getVertexTerm(i));
193     return result;
194 }
195
196 public VertexCollection<Object> existentialVariables() {
197     VertexArrayList<Object> result = new VertexArrayList<Object>();
198     final int nbTerms = getNbTerms();
199     for (int i = 0 ; i < nbTerms ; i++)
200         if (isExistential(i))
201             result.add(getVertexTerm(i));
202     return result;
203 }
204
205 /**
206  * Allows to know if a vertex belongs to the frontier of a rule.
207  * A term is said to be in the frontier iff it belongs to the body <b>and</b> to the head←
208  * of the rule.
209  * @return True if the vertex is in the frontier, false otherwise.
210  * @throws NoSuchElementException If the vertex is null, or its id does not match to the ←
211  * rule, or it is an atom vertex.
212  */
213 public boolean isFrontier(Vertex<Object> v) throws NoSuchElementException {
214     if ((v == null) || /*(v.getID() < getNbAtoms())*/ || (v.getID() > getNbVertices())←
215         ) ) {
216         throw new NoSuchElementException();
217     }
218     return isBody(v) && isHead(v);
219 }
220
221 /**
222  * Checks if a vertex is an existential variable.
223  * An existential variable is a variable which appears only in the head of the rule.
224  * @param v The vertex to be checked.
225  * @return True if the vertex is an existential variable, false otherwise.
226  */
227 public boolean isExistential(Vertex<Object> v) {
228     if ((v == null) || (v.getID() < getNbAtoms()))
229         return false;
230     return !(((Term)(v.getValue())).isConstant()) || isBody(v);
231 }
232
233 /**
234  * Checks if a vertex is a universal variable.
235  * A universal variable is a variable which appears at least in the body of the rule.
236  * @param v The vertex to be checked.
237  * @return True if the vertex is a universal variable, false otherwise.
238  */
239 public boolean isUniversal(Vertex<Object> v) {
240     if ((v == null) || (v.getID() < getNbAtoms()))
241         return false;
242     return !(((Term)(v.getValue())).isConstant()) && isBody(v);
243 }
244
245 public boolean isExistential(int i) {
246     try {
247         return isExistential(getVertexTerm(i));
248     }
249     catch (NoSuchElementException e) {
250         return false;
251     }
252 }

```

```

249     }
250
251     public boolean isUniversal(int i) {
252         try {
253             return isUniversal(getVertexTerm(i));
254         }
255         catch (NoSuchElementException e) {
256             return false;
257         }
258     }
259
260     /**
261     * Creates an array containing all positions of existential variables.<br />
262     * The length of the array matches the number of existential variables.
263     * This method is used by the unification process.
264     * @return An array of int containing the positions of all existential variables in the ←
265     *         rule.
266     */
267     public int[] existentialIndex() {
268         int cpt = 0;
269         int arity = ((Predicate)(_head.getValue())).getArity();
270         for (int i = 0 ; i < arity ; i++) {
271             if (isExistential(getVertexTermFromAtom(_head.getID(), i)))
272                 cpt++;
273         }
274         int result[] = new int[cpt];
275         cpt = 0;
276         for (int i = 0 ; i < arity ; i++) {
277             if (isExistential(getVertexTermFromAtom(_head.getID(), i))) {
278                 result[cpt] = i;
279                 cpt++;
280             }
281         }
282         return result;
283     }
284
285     /**
286     * Fullfills the rule from its string representation.
287     * @param str The string representation of the rule.
288     * @throws UnrecognizedStringFormatException If the string cannot be converted.
289     */
290     @Override
291     protected void fromString(String str) throws UnrecognizedStringFormatException {
292         String[] sub1 = str.split(HEAD_SEPARATOR);
293         if (sub1.length != 2)
294             throw new UnrecognizedStringFormatException();
295         sub1[0] += ATOM_SEPARATOR;
296         sub1[0] += sub1[1];
297         super.fromString(sub1[0]);
298         _head = getVertex(getNbAtoms() - 1);
299     }
300
301     /**
302     * Converts the rule into a String.
303     * @return The string representation of the rule.
304     */
305     @Override
306     public String toString() {
307         return super.toStringExcluding(_head.getID()) + HEAD_SEPARATOR + getHeadAtom();
308     }
309
310     /**
311     * Creates a new atom conjunction corresponding to the body of the rule.
312     * @return The atom conjunction of the body.
313     */
314     public AtomConjunction getBody() {
315         AtomConjunction body = new AtomConjunction();
316         super.cloneIn(body);
317         body.removeCleanlyAtom(_head.getID());
318         return body;
319     }
320
321     /**
322     * Clone the rule into a copy.
323     * @param copy The instance where to copied the rule.
324     * @return The copy of the rule.
325     */
326     public AtomicRule cloneIn(AtomicRule copy) {
327         super.cloneIn(copy);
328         copy._head = copy.getVertex(_head.getID());
329         return copy;
330     }
331
332     /**
333     * Clone the rule.
334     * @return A clone of the rule.
335     */

```

```

335 @Override
336 public AtomicRule clone() {
337     AtomicRule copy = new AtomicRule();
338     return cloneIn(copy);
339 }
340
341 /**
342  * Allows to know if the rule may imply the other one passed as parameter.
343  * I.e., if there exist a unificator between this rule head a subset of the other rule  $\leftrightarrow$ 
344     body.
345  * @param R The rule to be checked against.
346  * @return True if this rule may imply R, false otherwise.
347  * @see #existUnification(AtomConjunction, AtomicRule)
348  */
349 public boolean mayImply(AtomicRule R) {
350     return existUnification(R.getBody(), this);
351 }
352
353 /** Pointer to the head of the rule. */
354 private Vertex<Object> _head = null;
355
356 /* ALGORITHMS */
357
358 /*
359  * Unification
360  */
361
362 /**
363  * Static method used to know if there exists a unification between the atom conjunction  $\leftrightarrow$ 
364     and the rule
365     passed as parameters.
366  * @param H1 The atom conjunction to be unified.
367  * @param R The rule whose head will be unified.
368  * @return True if success, false otherwise.
369  */
370 public static boolean existUnification(AtomConjunction H1, AtomicRule R) {
371     /** init */
372     boolean isLocallyUnifiable[] = new boolean[H1.getNbAtoms()]; // valuing true while  $\leftrightarrow$ 
373     atom is supposed locally unifiable
374     int E[] = R.existentialIndex(); // positions of  $\leftrightarrow$ 
375     existential variables in the rule
376     Vertex<Object> head = null;
377     Vertex<Object> current = null;
378     AtomConjunction Q = null;
379
380     /** preprocessing */
381     for (int i = 0 ; i < H1.getNbAtoms() ; i++) {
382         Q = H1.subAtomConjunction(i,i+1);
383         if (localUnification(Q,R,E) == true)
384             isLocallyUnifiable[i] = true;
385         else
386             isLocallyUnifiable[i] = false;
387     }
388
389     /** extension */
390     for (int i = 0 ; i < H1.getNbAtoms() ; i++) {
391         current = H1.getVertex(i);
392         if (isLocallyUnifiable[i] == true) {
393             try {
394                 Q = extension(H1, i, isLocallyUnifiable, E);
395                 if (localUnification(Q,R,E) == true)
396                     return true;
397             }
398             catch (ExtensionFailureException e) {
399                 // the extended set cannot be unified with R
400             }
401             isLocallyUnifiable[current.getID()] = false;
402         }
403     }
404
405     // no set have been found => failure
406     return false;
407 }
408
409 /**
410  * Extends an atom conjunction for the unification algorithm. <br />
411  * This methods returns the atom conjunction corresponding to a minimal good unification  $\leftrightarrow$ 
412     set rooted in the atom ID from
413     the complete atom conjunction H1.
414  * @param H1 The complete atom conjunction.
415  * @param atomRootID The id of the atom which will be the root of the returned atom  $\leftrightarrow$ 
416     conjunction.
417  * @param isLocallyUnifiable An array of booleans such as isLocallyUnifiable[i] = true  $\leftrightarrow$ 
418     iff atom i is still known as a
419     locally unifiable one.
420  * @param E An array containing all existential positions of the rule whose head will be  $\leftrightarrow$ 

```

```

415     checked against
416     * the returned atom conjunction.
417     * @return A minimal good unification set rooted in a specific atom.
418     * @throws ExtensionFailureException If there exists no good unification set rooted in ←
419     atomID from H1.
420     */
421     protected static AtomConjunction extension(AtomConjunction H1, int atomRootID, boolean ←
422     isLocallyUnifiable[], int E[]) throws ExtensionFailureException {
423         AtomConjunction result = new AtomConjunction();
424         boolean isColored[] = new boolean[H1.getNbAtoms()+H1.getNbTerms()];
425         Fifo<Vertex<Object>> waiting = new Fifo<Vertex<Object>>();
426         Vertex<Object> current = H1.getVertex(atomRootID);
427         Vertex<Object> neighbour = null;
428         for (int i = 0 ; i < H1.getNbAtoms() ; i++) {
429             if (isLocallyUnifiable[i]) {
430                 isColored[i] = false;
431                 for (NeighbourIterator neighbourIterator = H1.vertexTermIteratorFromAtom(i) ; ←
432                 neighbourIterator.hasNext() ; )
433                     isColored[neighbourIterator.next().getID()] = false;
434             }
435         }
436         isColored[atomRootID] = true;
437         result.addAtom(current, H1);
438         waiting.put(current);
439         while (!waiting.isEmpty()) {
440             current = waiting.pop();
441             if (H1.isAtom(current)) {
442                 for (int i = 0 ; i < E.length ; i++) {
443                     neighbour = H1.getVertexTermFromAtom(current.getID(), E[i]);
444                     if (((Term)(neighbour.getValue())).isConstant())
445                         throw new ExtensionFailureException();
446                     if (!isColored[neighbour.getID()]) {
447                         isColored[neighbour.getID()] = true;
448                         waiting.put(neighbour);
449                     }
450                 }
451             } else { // current is a term
452                 for (NeighbourIterator neighbourIterator = H1.vertexAtomIteratorFromTerm(←
453                 current.getID() ; neighbourIterator.hasNext() ; ) {
454                     neighbour = neighbourIterator.next();
455                     if (!isLocallyUnifiable[neighbour.getID()])
456                         throw new ExtensionFailureException();
457                     if (!isColored[neighbour.getID()]) {
458                         isColored[neighbour.getID()] = true;
459                         waiting.put(neighbour);
460                         result.addAtom(neighbour, H1);
461                     }
462                 }
463             }
464         }
465         return result;
466     }
467
468     /**
469     * Checks if there is a unification between the head of the rule and the full atom ←
470     conjunction passed as parameters.
471     * @param H1 The atom conjunction to be unified.
472     * @param R The rule whose head is to be unified.
473     * @param existentialIndex The position of all existential variables of R. This parameter ←
474     is here for optimization only.
475     * @return True if there exists a unification, false otherwise.
476     */
477     protected static boolean localUnification(AtomConjunction H1, AtomicRule R, int [] ←
478     existentialIndex) {
479         /** predicate check */
480         for (Iterator<Vertex<Object>> iterator = H1._graph.firstVertexIterator() ; iterator.←
481         hasNext() ;)
482             if (((((Predicate)(iterator.next().getValue())).compareTo((Predicate)(R.getHead().←
483             getValue())))) != 0)
484                 return false;
485
486         /** graph generation */
487         AtomConjunction unification = H1;
488         unification._graph.addInFirstSet(((Predicate)(R.getHead().getValue())).clone());
489         int termID = -1;
490         int firstHeadTermID = -1;
491         for (Iterator<NeighbourEdge<Integer>> iterator = R._graph.neighbourIterator(R.←
492         getHead().getID() ; iterator.hasNext() ;) {
493             NeighbourEdge<Integer> edge = iterator.next();
494             Term t = (Term)(R.get(edge.getIDV()));
495             if (t.isConstant())
496                 termID = unification.addTerm(t);
497         }

```



```

491     else {
492         termID = -1;
493         if (firstHeadTermID > 0) {
494             for (int i = firstHeadTermID ; i < unification._graph.getNbVertices() ; i++) {
495                 if (unification.getTerm(i-unification.getNbAtoms()).getLabel().compareTo(t.getLabel()) == 0)
496                     termID = i;
497             }
498         }
499         if (termID < 0) {
500             unification._graph.addInSecondSet(t);
501             termID = unification._graph.getNbVertices()-1;
502         }
503         if (firstHeadTermID < 0)
504             firstHeadTermID = termID;
505     }
506     try {
507         unification._graph.addEdge(unification.getNbAtoms()-1, termID, new Integer(edge.getValue()));
508     }
509     catch (IllegalEdgeException e) { }
510 }
511
512 /** index */
513 int headIndex = 0;
514
515 Vertex<Object> headVertex = null;
516 Vertex<Object> bodyVertex = null;
517 Term headTerm = null;
518 Term bodyTerm = null;
519 int arity = ((Predicate)(R.getHead().getValue())).getArity();
520
521 boolean[] isexistential = new boolean[unification.getNbTerms()];
522 for (int i = 0 ; i < unification.getNbTerms() ; i++)
523     isexistential[i] = false;
524 for (int i = 0 ; i < existentialIndex.length ; i++)
525     isexistential[unification.getVertexTermFromAtom(unification.getNbAtoms()-1, existentialIndex[i]).getID()-unification.getNbAtoms()] = true;
526
527
528 /** algorithm */
529 while (headIndex < arity) {
530     headVertex = unification.getVertexTermFromAtom(unification.getNbAtoms()-1, headIndex);
531     headTerm = (Term)(headVertex.getValue());
532     for (int i = 0 ; i < unification.getNbAtoms() - 1 ; i++) {
533         bodyVertex = unification.getVertexTermFromAtom(i, headIndex);
534         bodyTerm = (Term)(bodyVertex.getValue());
535         if (headVertex.getID() != bodyVertex.getID()) {
536             if ((headTerm.isConstant() || isexistential[headVertex.getID()-unification.getNbAtoms()])
537                 && (bodyTerm.isConstant() || isexistential[bodyVertex.getID()-unification.getNbAtoms()]))
538                 return false;
539             else if ((bodyTerm.isVariable()) && (!isexistential[bodyVertex.getID()-unification.getNbAtoms()])) {
540                 try {
541                     unification._graph.contract(bodyVertex.getID(), headVertex.getID());
542                 }
543                 catch (Exception e) {
544                     //e.printStackTrace();
545                     // TODO ConcurrentModificationException ???
546                 }
547             }
548             else {
549                 try {
550                     unification._graph.contract(headVertex.getID(), bodyVertex.getID());
551                 }
552                 catch (Exception e) {
553                     //e.printStackTrace();
554                     // TODO ConcurrentModificationException ???
555                 }
556             }
557         }
558     }
559     headIndex++;
560 }
561 return true;
562 }
563
564 }
565
566 }
567

```

568  
569  
570

} ;

```

1 package obr;
2
3 import java.util.Scanner;
4 import java.util.regex.Pattern;
5 import java.io.File;
6
7 /**
8  * This class provides static functions to parse a single rule from its string
9  * representation, or a full set of rules from a file.
10 */
11 public class DTGParser {
12
13     public static final String HEAD_SEPARATOR = " :- ";
14     public static final String BEGIN_TERM_LIST = "\\("; // TODO
15     public static final String END_TERM_LIST = "\\)";
16     public static final String TERM_SEPARATOR = ", ";
17     /** Not marker constant. */
18     public static final char NOT_MARK = '!';
19     /** Variable marker. */
20     public static final char VARIABLE_MARK = '?';
21     /** End of line marker. */
22     public static final char END_LINE = '.';
23     /**
24      * Absurd predicate used to convert <code>body —&gt; !head</code> to
25      * <code>body;head—&gt;ABSURD</code>.
26      */
27     public static final Predicate ABSURD = new Predicate("ABSURD",0);
28
29     /**
30      * Parses a rule under a DTG format.
31      * @param str The rule string representation.
32      * @throws UnrecognizedStringFormatException If the string format is not ok.
33      */
34     public static AtomicRule parseRule(String str) throws UnrecognizedStringFormatException {
35         String[] subs = str.split(HEAD_SEPARATOR);
36         if (subs.length != 2)
37             throw new UnrecognizedStringFormatException();
38         if (subs[1].charAt(subs[1].length()-1) != END_LINE)
39             throw new UnrecognizedStringFormatException();
40         AtomicRule rule = new AtomicRule();
41
42         String[] terms;
43         String[] atomSubs;
44         String term;
45         int termID, atomID;
46         boolean not = false;
47
48         if (subs[0].charAt(0) == NOT_MARK) {
49             subs[0] = subs[0].substring(1,subs[0].length());
50             not = true;
51         }
52
53         /* BODY */
54         subs[1] = subs[1].substring(0,subs[1].length()-1); // last char is a dot
55         atomSubs = subs[1].split(BEGIN_TERM_LIST);
56         if (atomSubs.length != 2)
57             throw new UnrecognizedStringFormatException();
58         atomSubs[1] = atomSubs[1].substring(0,atomSubs[1].length()-1);
59         terms = atomSubs[1].split(TERM_SEPARATOR);
60         atomID = rule.addAtom(new Predicate(atomSubs[0],terms.length));
61         for (int i = 0 ; i < terms.length ; i++) {
62             if (terms[i].charAt(0) == VARIABLE_MARK)
63                 term = terms[i].substring(1,terms[i].length());
64             else
65                 term = "!" + terms[i] + "!";
66             termID = rule.addTerm(term);
67             rule.addEdge(atomID,termID,i);
68         }
69
70         /* HEAD */
71         atomSubs = subs[0].split(BEGIN_TERM_LIST);
72         if (atomSubs.length != 2)
73             throw new UnrecognizedStringFormatException();
74         atomSubs[1] = atomSubs[1].substring(0,atomSubs[1].length()-1);
75         terms = atomSubs[1].split(TERM_SEPARATOR);
76         atomID = rule.addAtom(new Predicate(atomSubs[0],terms.length));
77         for (int i = 0 ; i < terms.length ; i++) {
78             if (terms[i].charAt(0) == VARIABLE_MARK)
79                 term = terms[i].substring(1,terms[i].length());
80             else
81                 term = "!" + terms[i] + "!";
82             termID = rule.addTerm(term);
83             rule.addEdge(atomID,termID,i);
84         }
85         rule.setHead(atomID);
86     }

```

```

87         if (not) {
88             atomID = rule.addAtom(ABSURD);
89             rule.setHead(atomID);
90         }
91
92         return rule;
93
94     }
95
96     /**
97     * Parses a file and returns the graphe of rule dependencies generated from the rule
98     * set.
99     * @param filePath The path to the file to read.
100    * @return The graph of rule dependencies corresponding, or null if an error occurred.
101    */
102    public static GraphRuleDependencies parseRules(String filePath) {
103        try {
104            GraphRuleDependencies grd = new GraphRuleDependencies();
105            String line = null;
106            Scanner scan = new Scanner(new File(filePath));
107            scan.useDelimiter(Pattern.compile("\n"));
108            line = scan.next();
109            while (scan.hasNext()) {
110                if ((line.length() >= 2) && ((line.charAt(0) != '/') || (line.charAt(1) != '/'↵
111                    '))) {
112                    try {
113                        grd.addVertex(parseRule(line));
114                    }
115                    catch (UnrecognizedStringFormatException exception) { }
116                }
117                line = scan.next();
118            }
119            scan.close();
120            return grd;
121        }
122        catch (Exception e) {
123            System.out.println(e);
124            e.printStackTrace();
125            return null;
126        }
127    }
128
129    /** This class cannot be instantiated. */
130    protected DTGParser() { }
131
132 };

```

```

1 package obr;
2
3 /**
4  * Interface for functions checking if a graph of rule dependencies or a subset of this graph↵
5  * belongs to a concrete
6  * decidable class. <br />
7  * It provides two methods, the first one is used to check the full set of rules, and the ↵
8  * second one to check only
9  * a strongly connected component of this set.
10 * The classes which implement this interface may be used as a check function in the GRD ↵
11 * Analyser.
12 * @see GRDAnalyser
13 */
14 public interface DecidableClassCheck {
15
16     /**
17     * Checks a graph of rule dependencies.
18     * @return The associated decidable class label if the grd belongs to this decidable ↵
19     * class,
20     * null otherwise.
21     */
22     DecidableClassLabel grdCheck(GraphRuleDependencies grd);
23
24     /**
25     * Checks only a strongly connected component of the graph of rule dependencies.
26     * @param grd The graph of rule dependencies.
27     * @param sccID The id of the strongly connected component to be checked.
28     * @return The associated decidable class label if the strongly connected component ↵
29     * belongs to this decidable class,
30     * null otherwise.
31     */
32     DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID);
33 };

```

```

1 package obr;
2
3 /**
4  * Label for a decidable class check.
5  * @see GRDAnalyser
6  */
7 public class DecidableClassLabel {
8
9     static final int FES = 1;
10    static final int GBTS = 2;
11    static final int FUS = 3;
12    static final String FES_STR = "fes";
13    static final String GBTS_STR = "gbts";
14    static final String FUS_STR = "fus";
15    static final String FES_STRING = "Finite Expansion Set";
16    static final String GBTS_STRING = "Greedy Bounded Treewidth Set";
17    static final String FUS_STRING = "Finite Unification Set";
18
19    static final String shortName(int abstractClassID) {
20        switch (abstractClassID) {
21            case FUS :
22                return FUS_STR;
23            case FES :
24                return FES_STR;
25            case GBTS :
26                return GBTS_STR;
27            default :
28                return "";
29        }
30    }
31
32    static final String longName(int abstractClassID) {
33        switch (abstractClassID) {
34            case FUS :
35                return FUS_STRING;
36            case FES :
37                return FES_STRING;
38            case GBTS :
39                return GBTS_STRING;
40            default :
41                return "";
42        }
43    }
44
45    /**
46     * Constructor which let all boolean attributes to false.
47     * @param label The label of the decidable class.
48     */
49    public DecidableClassLabel(String label) {
50        _label = label;
51    }
52
53    /**
54     * Complete constructor.
55     * @param label The label of the decidable class.
56     * @param fus True if the concrete class belongs to fus abstract class.
57     * @param fes True if the concrete class belongs to fes abstract class.
58     * @param gbts True if the concrete class belongs to gbts abstract class.
59     */
60    public DecidableClassLabel(String label, boolean fus, boolean fes, boolean gbts) {
61        _label = label;
62        _fus = fus;
63        _fes = fes;
64        _gbts = gbts;
65    }
66
67    /**
68     * Label getter.
69     * @return The label of the concrete class.
70     */
71    public String getLabel() {
72        return _label;
73    }
74
75    public boolean isFUS() {
76        return _fus;
77    }
78
79    public boolean isFES() {
80        return _fes;
81    }
82
83    public boolean isGBTS() {
84        return _gbts;
85    }
86

```

```

87  /**
88   * Converts the label into a string.
89   * @return A string representation of the concrete class label.
90   */
91  @Override
92  public String toString() {
93      String result = _label;
94      result += "(" + " ";
95      if (_fus)
96          result += FUS_STR + " ";
97      if (_fes)
98          result += FES_STR + " ";
99      if (_gbts)
100         result += GBTS_STR + " ";
101         result += ")";
102         return result;
103     }
104
105     /** The real label of the concrete class. */
106     private String _label;
107
108     /** True if the concrete class belongs to fus abstract class. */
109     private boolean _fus = false;
110
111     /** True if the concrete class belongs to fes abstract class. */
112     private boolean _fes = false;
113
114     /** True if the concrete class belongs to gbts abstract class. */
115     private boolean _gbts = false;
116
117 };

```

```

1 package obr;
2
3 import moca.graphs.IllegalConstructionException;
4 import moca.graphs.vertices.Vertex;
5 import moca.graphs.edges.NeighbourEdge;
6 import moca.graphs.edges.IllegalEdgeException;
7
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.NoSuchElementException;
11
12 /**
13  * Checks if a set of rules satisfies the disconnected property.<br />
14  * I.e., if all rules have a frontier of size 0.
15  */
16 public class DisconnectedCheck implements DecidableClassCheck {
17
18     public static final DecidableClassLabel LABEL = new DecidableClassLabel("disconnected", ←
19         true, true, true);
20
21     /**
22      * Checks a graph of rule dependencies.
23      * @return The decidable class label if the grd belongs to this decidable class, null ←
24      * otherwise.
25      */
26     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
27         if (check(grd.getVertexCollection()))
28             return LABEL;
29         return null;
30     }
31
32     /**
33      * Checks only a strongly connected component of the graph of rule dependencies.
34      * @param grd The graph of rule dependencies.
35      * @param sccID The id of the strongly connected component to be checked.
36      * @return The decidable class label if the strongly connected component belongs to this ←
37      * decidable class, null otherwise.
38      */
39     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
40         if (check(grd.getComponent(sccID)))
41             return LABEL;
42         return null;
43     }
44
45     /**
46      * Internal check.
47      * @param rules The set of rules to be checked.
48      */
49     protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
50         for (Vertex<AtomicRule> vrule : rules) {
51             if (vrule.getValue().frontier().size() != 0)
52                 return false;
53         }
54         return true;
55     }
56 }

```



```

1 package obr;
2
3 import moca.graphs.IllegalConstructionException;
4 import moca.graphs.vertices.Vertex;
5 import moca.graphs.edges.NeighbourEdge;
6 import moca.graphs.edges.IllegalEdgeException;
7
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.NoSuchElementException;
11
12 /**
13  * Checks if a set of rules satisfies the domain-restricted property.
14  */
15 public class DomainRestrictedCheck implements DecidableClassCheck {
16
17     /** Associated label. */
18     public static final DecidableClassLabel LABEL = new DecidableClassLabel("domain-↔
19         restricted", true, false, false);
20
21     /**
22      * Checks a graph of rule dependencies.
23      * @return the decidable class label if the grd belongs to this decidable class, null ↔
24      * otherwise.
25      */
26     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
27         if (check(grd.getVertexCollection()))
28             return LABEL;
29         return null;
30     }
31
32     /**
33      * Checks only a strongly connected component of the graph of rule dependencies.
34      * @param grd The graph of rule dependencies.
35      * @param sccID The id of the strongly connected component to be checked.
36      * @return the decidable class label if the strongly connected component belongs to this ↔
37      * decidable class, null otherwise.
38      */
39     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
40         if (check(grd.getComponent(sccID)))
41             return LABEL;
42         return null;
43     }
44
45     protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
46         AtomicRule rule = null;
47         int frontierSize = 0;
48         for (Vertex<AtomicRule> vrule : rules) {
49             rule = vrule.getValue();
50             frontierSize = rule.frontier().size();
51             if ((frontierSize > 0) && (frontierSize < rule.getNbUniversalVariables()))
52                 return false;
53         }
54         return true;
55     }
56 }

```

```
1 package obr;  
2  
3 /**  
4  * Exception thrown by the extension algorithm if failure.  
5  * @see AtomicRule#extension  
6  */  
7 public class ExtensionFailureException extends Exception { };
```

```

1 package obr;
2
3 import java.util.ArrayList;
4 import java.util.Iterator;
5
6 import moca.graphs.vertices.Vertex;
7
8 public class FrontierGuardedCheck implements DecidableClassCheck{
9
10     public static final DecidableClassLabel LABEL = new DecidableClassLabel("frontier-guarded↔
11         ", false, false, true);
12
13     /**
14      * Checks a graph of rule dependencies.
15      * @return the decidable class label if the grd belongs to this decidable class, null ↵
16      * otherwise.
17      */
18     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
19         if (check(grd.getVertexCollection()))
20             return LABEL;
21         return null;
22     }
23
24     /**
25      * Checks only a strongly connected component of the graph of rule dependencies.
26      * @param grd The graph of rule dependencies.
27      * @param scc The strongly connected component to be checked.
28      * @return the decidable class label if the strongly connected component belongs to this ↵
29      * decidable class, null otherwise.
30      */
31     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
32         if (check(grd.getComponent(sccID)))
33             return LABEL;
34         return null;
35     }
36
37     protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
38         for (Vertex<AtomicRule> vrule : rules) {
39             if (!ruleCheck(vrule.getValue()))
40                 return false;
41         }
42         return true;
43     }
44
45     protected boolean ruleCheck(AtomicRule rule) {
46         ArrayList<Vertex<Object>> frontier = rule.frontier();
47         boolean guarded;
48         for (Atom atom : rule) {
49             guarded = true;
50             for (Vertex<Object> vertexTerm : frontier) {
51                 Term t = (Term)(vertexTerm.getValue());
52                 if (!atom.contains(t)) {
53                     guarded = false;
54                     break;
55                 }
56             }
57             if (guarded)
58                 return true;
59         }
60         return false;
61     }
62 }

```

```

1 package obr;
2
3 import moca.graphs.IllegalConstructionException;
4 import moca.graphs.vertices.Vertex;
5 import moca.graphs.edges.NeighbourEdge;
6 import moca.graphs.edges.IllegalEdgeException;
7
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.NoSuchElementException;
11
12 /**
13  * Checks if a set of rules satisfies the frontier-1 property.<br />
14  * I.e., if all rules have a frontier of size 1.
15  */
16 public class FrontierOneCheck implements DecidableClassCheck {
17
18     /** Associated label. */
19     public static final DecidableClassLabel LABEL = new DecidableClassLabel("frontier-1", ←
20         false, false, true);
21
22     /**
23      * Checks a graph of rule dependencies.
24      * @return the decidable class label if the grd belongs to this decidable class, null ←
25      * otherwise.
26      */
27     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
28         if (check(grd.getVertexCollection()))
29             return LABEL;
30         return null;
31     }
32
33     /**
34      * Checks only a strongly connected component of the graph of rule dependencies.
35      * @param grd The graph of rule dependencies.
36      * @param sccID The id of the strongly connected component to be checked.
37      * @return the decidable class label if the strongly connected component belongs to this ←
38      * decidable class, null otherwise.
39      */
40     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
41         if (check(grd.getComponent(sccID)))
42             return LABEL;
43         return null;
44     }
45
46     protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
47         for (Vertex<AtomicRule> vrule : rules) {
48             if (vrule.getValue().frontier().size() != 1){
49                 return false;
50             }
51         }
52         return true;
53     }
54 }

```

```

1 package obr;
2
3 import moca.graphs.vertices.Vertex;
4 import moca.graphs.vertices.VertexBinaryFunction;
5 import moca.graphs.Graph;
6 import java.util.ArrayList;
7
8 /**
9  * Analyse a graph of rule dependencies to determine if it belongs to decidable class.<br />
10  * Furthermore, the GRDAnalyser will check the strongly connected components of the GRD.
11  * Then, it will check if a query will be answered in a finite time.
12  */
13 public class GRDAnalyser {
14
15     public abstract class ClassChecker {
16         public void unprocess() { _processed = false; }
17         public void process() { _processed = true; }
18         public abstract String diagnostic();
19         public boolean isProcessed() { return _processed; }
20         protected boolean _processed = false;
21     }
22
23     protected class DecidableClassChecker extends ClassChecker {
24
25         public static final String UNPROCESSED_MSG = "Decidable class checks unprocessed!";
26
27         /**
28          * Adds a new decidable class check function into the list.
29          * If the function already exists, it will not be added.
30          * @param checkFunction The new function to add.
31          */
32         public void addDecidableClassCheck(DecidableClassCheck checkFunction) {
33             if (!_checkFunctions.contains(checkFunction))
34                 _checkFunctions.add(checkFunction);
35         }
36
37         public boolean isFES(int sccID) {
38             if (!_processed)
39                 return false;
40             ArrayList<DecidableClassLabel> labels = _sccLabels.get(sccID);
41             for (DecidableClassLabel label : labels)
42                 if (label.isFES())
43                     return true;
44             return false;
45         }
46
47         public boolean isGBTS(int sccID) {
48             if (!_processed)
49                 return false;
50             ArrayList<DecidableClassLabel> labels = _sccLabels.get(sccID);
51             for (DecidableClassLabel label : labels)
52                 if (label.isGBTS())
53                     return true;
54             return false;
55         }
56
57         public boolean isFUS(int sccID) {
58             if (!_processed)
59                 return false;
60             ArrayList<DecidableClassLabel> labels = _sccLabels.get(sccID);
61             for (DecidableClassLabel label : labels)
62                 if (label.isFUS())
63                     return true;
64             return false;
65         }
66
67         public boolean isFES() {
68             if (_grdLabels.size() > 0) {
69                 for (int i = 0 ; (i < _grdLabels.size()) ; i++) {
70                     if (_grdLabels.get(i).isFES())
71                         return true;
72                 }
73             }
74             return false;
75         }
76
77         public boolean isGBTS() {
78             if (_grdLabels.size() > 0) {
79                 for (int i = 0 ; (i < _grdLabels.size()) ; i++) {
80                     if (_grdLabels.get(i).isGBTS())
81                         return true;
82                 }
83             }
84             return false;
85         }
86
87         public boolean isFUS() {
88             if (_grdLabels.size() > 0) {
89                 for (int i = 0 ; (i < _grdLabels.size()) ; i++) {
90                     if (_grdLabels.get(i).isFUS())
91                         return true;
92                 }
93             }
94             return false;
95         }
96     }
97 }

```

```

87     }
88     return false;
89 }
90
91 /**
92  * <p>Process all checks in a specific order :
93  * <ul>
94  * <li>checks if the graph is cyclic ;</li>
95  * <li>if it is, checks all functions onto the complete graph of rule dependencies ←
96  * <li>and checks all functions onto each strongly connected component of the GRD.</li>
97  * </ul>
98  */
99 @Override
100 public void process() {
101     // reset old data
102     if (_processed == true)
103         unprocess();
104     // init
105     _grdLabels = new ArrayList<DecidableClassLabel>();
106     _sccLabels = new ArrayList<ArrayList<DecidableClassLabel>>>(_grd.getNbComponents() ←
107         ());
108     // starts checking
109     if (_grd.isCyclic()) {
110         DecidableClassLabel l = null;
111         for (DecidableClassCheck function : _checkFunctions) {
112             try {
113                 l = function.grdCheck(_grd);
114                 if (l != null)
115                     _grdLabels.add(l);
116             } catch (UnsupportedOperationException e) { /* this check function cannot ←
117                 be used against the full grd */ }
118         }
119         for (int i = 0 ; i < _grd.getNbComponents() ; i++) {
120             _sccLabels.add(new ArrayList<DecidableClassLabel>());
121             for (DecidableClassCheck function : _checkFunctions) {
122                 try {
123                     l = function.sccCheck(_grd, i);
124                     if (l != null)
125                         _sccLabels.get(i).add(l);
126                 } catch (UnsupportedOperationException e) { /* ... against a single scc ←
127                     */ }
128             }
129         }
130     } else
131         _grdLabels.add(AGRD_LABEL);
132     _processed = true;
133 }
134
135 @Override
136 public void unprocess() {
137     _grdLabels = null;
138     _sccLabels = null;
139     _processed = false;
140 }
141
142 @Override
143 public String diagnostic() {
144     if (!_processed)
145         return UNPROCESSED_MSG;
146
147     StringBuilder result = new StringBuilder();
148
149     // complete grd
150     result.append("Graph of Rule Dependencies labels :\n");
151     for (DecidableClassLabel label : _grdLabels) {
152         result.append("\t");
153         result.append(label);
154         result.append('\n');
155     }
156     result.append('\n');
157
158     // scc
159     if (_sccLabels.size() > 0)
160         result.append("Strongly Connected Components labels :\n");
161     for (int i = 0 ; i < _sccLabels.size() ; i++) {
162         result.append("C ");
163         result.append(i);
164         result.append('\t');
165         if (_sccLabels.get(i).size() > 0) {
166             for (int j = 0 ; j < _sccLabels.get(i).size() ; j++) {
167                 result.append(_sccLabels.get(i).get(j));
168                 result.append(" ");

```

```

169         }
170     }
171     else
172         result.append("none");
173     result.append('\n');
174 }
175
176     return result.toString();
177 }
178
179 /** Contains all determined decidable class labels for the GRD. */
180 private ArrayList<DecidableClassLabel> _grdLabels;
181 /** Contains all determined decidable class labels for each strongly connected ←
182     component of the GRD. */
183 private ArrayList<ArrayList<DecidableClassLabel>> _sccLabels;
184
185 /** The check functions to be used. */
186 private ArrayList<DecidableClassCheck> _checkFunctions = new ArrayList<←
187     DecidableClassCheck>();
188
189 };
190
191 protected class AbstractClassChecker extends ClassChecker
192     implements VertexBinaryFunction<ArrayList<Vertex<AtomicRule>>> {
193
194     public static final String UNPROCESSED_MSG = "Abstract class checks unprocessed!";
195
196     @Override
197     public void unprocess() {
198         _grdAbstractClass = 0;
199         _sccAbstractClass = null;
200         _processed = false;
201     }
202
203     @Override
204     public void process() {
205         if (_processed)
206             unprocess();
207         _decidable = true;
208         if (isFES())
209             _grdAbstractClass = DecidableClassLabel.FES;
210         else if (isGBTS())
211             _grdAbstractClass = DecidableClassLabel.GBTS;
212         else if (isFUS())
213             _grdAbstractClass = DecidableClassLabel.FUS;
214         else
215             _decidable = false;
216         //if (!_decidable) {
217             _sccAbstractClass = new int[_grd.getNbComponents()];
218             processCombine();
219         //}
220         _processed = true;
221     }
222
223     protected void processCombine() {
224         ArrayList<Integer> sourceID = new ArrayList<Integer>();
225         final int nbComponents = _grd.getNbComponents();
226         int i, j;
227         boolean ended;
228         for (i = 0 ; i < nbComponents ; i++) {
229             ended = false;
230             for (j = 0 ; (j < nbComponents) && !ended ; j++) {
231                 if ((i != j) && (_grd.getStronglyConnectedComponentsGraph().isEdge(j,i)))
232                     ended = true;
233             }
234             if (j == nbComponents)
235                 sourceID.add(new Integer(i));
236         }
237
238         _decidable = true;
239         Graph.WalkIterator iterator;
240         for (i = 0 ; (i < sourceID.size()) && _decidable ; i++)
241             setMin(sourceID.get(i), 0);
242         iterator = _grd.getStronglyConnectedComponentsGraph().BFSIterator(sourceID, this, ←
243             null);
244         while ((iterator.hasNext()) && _decidable)
245             iterator.next();
246     }
247
248     @Override
249     public String diagnostic() {
250         if (!_processed)
251             return UNPROCESSED_MSG;
252
253         if (!_decidable)
254             return "This set of rule cannot be used to make a query.";
255     }

```

```

253     StringBuilder result = new StringBuilder();
254     if (_grdAbstractClass > 0) {
255         result.append("All the GRD rules belongs to the ");
256         result.append(DecidableClassLabel.longName(_grdAbstractClass));
257         result.append(" abstract class.\n");
258         result.append("But you can also use specific algorithms :\n\n");
259         result.append("GRD \t");
260         result.append(DecidableClassLabel.longName(_grdAbstractClass));
261         result.append('\n');
262     }
263     else
264         result.append("You must use different algorithms depending on the strongly ↔
//         connected component.\n\n");
265     //
266     //
267     for (int i = 0 ; i < _sccAbstractClass.length ; i++) {
268         result.append("SCC[");
269         result.append(i);
270         result.append("]\t");
271         result.append(DecidableClassLabel.longName(_sccAbstractClass[i]));
272         result.append('\n');
273     }
274     //
275     return result.toString();
276 }
277
278 /**
279  * Abstract class to "use" on the full GRD.
280  * 0 means there is not only one abstract class which contains all rules.
281  */
282 private int _grdAbstractClass = 0;
283 /**
284  * Contains the abstract class label to use on the specific strongly connected
285  * component.
286  */
287 private int _sccAbstractClass[] = null;
288 private boolean _decidable = false;
289
290 @Override
291 public void exec(Vertex<ArrayList<Vertex<AtomicRule>>> pred,
292                 Vertex<ArrayList<Vertex<AtomicRule>>> next) {
293     final int sccPredID = pred.getID();
294     final int sccNextID = next.getID();
295     if (_sccAbstractClass[sccPredID] > _sccAbstractClass[sccNextID])
296         setMin(sccNextID, _sccAbstractClass[sccPredID]);
297 }
298 private void setMin(int sccNextID, int label) {
299     if (label <= DecidableClassLabel.FES) {
300         if (isFES(sccNextID)) {
301             _sccAbstractClass[sccNextID] = DecidableClassLabel.FES;
302             return;
303         }
304     }
305     if (label <= DecidableClassLabel.GBTS) {
306         if (isGBTS(sccNextID)) {
307             _sccAbstractClass[sccNextID] = DecidableClassLabel.GBTS;
308             return;
309         }
310     }
311     if (label <= DecidableClassLabel.FUS) {
312         if (isFUS(sccNextID)) {
313             _sccAbstractClass[sccNextID] = DecidableClassLabel.FUS;
314             return;
315         }
316     }
317     _decidable = false;
318     _sccAbstractClass[sccNextID] = 0;
319 }
320
321 };
322
323
324
325 public static final DecidableClassLabel AGRD_LABEL = new DecidableClassLabel("acyclic-grd↔
326     ", true, true, false);
327
328 /**
329  * Constructor.
330  * @param grd The graph of rule dependencies to be analysed.
331  */
332 public GRDAnalyser(GraphRuleDependencies grd) {
333     _grd = grd;
334 }
335
336 public void setGRD(GraphRuleDependencies grd) {
337     _grd = grd;
338     _decidableChecker.unprocess();

```



```

338     _abstractChecker.unprocess();
339 }
340
341 public String diagnostic() {
342     StringBuilder result = new StringBuilder();
343     result.append("[a] DECIDABLE CLASS CHECKS\n\n");
344     result.append(_decidableChecker.diagnostic());
345     result.append("\n\n[b] ABSTRACT CLASS TO USE\n\n");
346     result.append(_abstractChecker.diagnostic());
347     return result.toString();
348 }
349
350 public void addDecidableClassCheck(DecidableClassCheck checkFunction) {
351     _decidableChecker.addDecidableClassCheck(checkFunction);
352 }
353
354 public void process() {
355     processDecidableClass();
356     processAbstractClass();
357 }
358
359 public void processDecidableClass() {
360     _decidableChecker.process();
361 }
362
363 public void processAbstractClass() {
364     _abstractChecker.process();
365 }
366
367 public boolean isFUS() {
368     return _decidableChecker.isFUS();
369 }
370
371 public boolean isFES() {
372     return _decidableChecker.isFES();
373 }
374
375 public boolean isGBTS() {
376     return _decidableChecker.isGBTS();
377 }
378
379 public boolean isFUS(int sccID) {
380     return _decidableChecker.isFUS(sccID);
381 }
382
383 public boolean isFES(int sccID) {
384     return _decidableChecker.isFES(sccID);
385 }
386
387 public boolean isGBTS(int sccID) {
388     return _decidableChecker.isGBTS(sccID);
389 }
390
391 /** The graph of rule dependencies to be analysed. */
392 private GraphRuleDependencies _grd;
393
394 private DecidableClassChecker _decidableChecker = new DecidableClassChecker();
395 private AbstractClassChecker _abstractChecker = new AbstractClassChecker();
396
397 };

```

```

1 package obr;
2
3 import moca.graphs.DirectedSimpleGraph;
4 import moca.graphs.IllegalConstructionException;
5 import moca.graphs.vertices.Vertex;
6 import moca.graphs.edges.IllegalEdgeException;
7 import moca.graphs.edges.NeighbourEdge;
8
9 import java.util.ArrayList;
10 import java.util.Iterator;
11 import java.util.HashMap;
12 import java.util.NoSuchElementException;
13
14
15 /**
16  * Represents the graph of position dependencies corresponding to a GRD or to one of its  $\leftrightarrow$ 
17  * strongly connected components.<br />
18  * The graph maintains a hash map to find quickly the vertex corresponding to a predicate and  $\leftrightarrow$ 
19  * its position.
20  * It is used to check the weakly-acyclic and the weakly-sticky properties.
21  */
22 public class GraphPositionDependencies extends DirectedSimpleGraph<Predicate, Boolean> {
23
24     /**
25      * Default constructor.
26      */
27     public GraphPositionDependencies() throws IllegalConstructionException {
28         super();
29     }
30
31     /**
32      * Constructor from a set of rules.
33      * @param rules The set of rule to convert.
34      */
35     public GraphPositionDependencies(Iterable<Vertex<AtomicRule>> rules) throws  $\leftrightarrow$ 
36         IllegalConstructionException {
37         super();
38         init(rules);
39     }
40
41     /**
42      * Adds a predicate into the graph if not already in.
43      * Note that the number of created vertices is equals to the predicate arity.
44      * @param p The predicate to be added.
45      */
46     public void addPredicate(Predicate p) {
47         if (!_predicateIndex.containsKey(p)) {
48             _predicateIndex.put(p, getNbVertices());
49             for (int i = 0 ; i < p.getArity() ; i++)
50                 addVertex(p);
51         }
52     }
53
54     /**
55      * Convenient method to add a non special edge between two predicate positions.
56      * (from the first to the second)
57      * @param p The first predicate.
58      * @param positionP The first position (it must be lesser than p arity).
59      * @param q The second predicate.
60      * @param positionQ The second position (it mus be lesser than q arity).
61      */
62     public void addEdge(Predicate p, int positionP, Predicate q, int positionQ) {
63         addEdge(p, positionP, q, positionQ, false);
64     }
65
66     /**
67      * Convenient method to add a non special edge between two predicate positions.
68      * (from the first to the second)
69      * @param p The first predicate.
70      * @param positionP The first position (it must be lesser than p arity).
71      * @param q The second predicate.
72      * @param positionQ The second position (it mus be lesser than q arity).
73      * @param special The edge value, if true it will be a special edge.
74      */
75     public void addEdge(Predicate p, int positionP, Predicate q, int positionQ, Boolean  $\leftrightarrow$ 
76         special) {
77         try {
78             addEdge(getVertex(p, positionP), getVertex(q, positionQ), special);
79         }
80         catch (IllegalEdgeException e) {
81             // if an edge already exists its value should be checked against the special  $\leftrightarrow$ 
82             parameter.
83         }
84     }
85 }
86
87 /**

```

```

82 * Returns the vertex associated to a predicate position.<br />
83 * Uses an internal hash map to provide fast access.
84 * @param p The predicate to be found.
85 * @param position The position inside the predicate.
86 * @throws NoSuchElementException Whenever a predicate does not belong to the graph, or ↵
      if the position is greater than predicate arity.
87 */
88 public Vertex<Predicate> getVertex(Predicate p, int position) throws ↵
      NoSuchElementException {
89     if (position >= p.getArity())
90         throw new NoSuchElementException();
91     Integer pIndex = _predicateIndex.get(p);
92     if (pIndex == null)
93         throw new NoSuchElementException();
94     return getVertex(pIndex+position);
95 }
96
97 /**
98 * Used to check the weakly acyclic property.
99 * @return True if there is no position with infinite rank, i.e. if it is weakly acyclic.
100 */
101 public boolean finiteRank() {
102     if (_finiteRanks != null) {
103         for (int i = 0 ; i < getNbComponents() ; i++)
104             if (!_finiteRanks[i])
105                 return false;
106     }
107     if (isAcyclic())
108         return true;
109     boolean success = true;
110     ArrayList<Vertex<Predicate>> scc = null;
111     for (int i = 0 ; (i < getNbComponents()) && success ; i++) {
112         scc = getComponent(i);
113         // NOT OPTIMIZED TODO
114         for (int j = 0 ; (j < scc.size()) && success ; j++) {
115             for (int k = 0 ; (k < scc.size()) && success ; k++) {
116                 try {
117                     if (getEdgeValue(scc.get(j).getID(), scc.get(k).getID()) == true)
118                         success = false;
119                 }
120                 catch (NoSuchElementException e) {
121                     // the edge does not exist,
122                     // in particular there is no special edge
123                 }
124             }
125         }
126     }
127     return success;
128 }
129
130 /**
131 * Used to check the weakly sticky property.
132 * @param vertexID The vertex to be checked.
133 * @return True if the position corresponding to the vertex has a finite rank.
134 */
135 public boolean finiteRank(int vertexID) {
136     if (_finiteRanks == null) {
137         if (isAcyclic())
138             return true;
139         processFiniteRanks();
140     }
141     return _finiteRanks[vertexID];
142 }
143
144 /**
145 * Convenient method for finiteRank(vertexID).
146 * @see #finiteRank(int)
147 */
148 public boolean finiteRank(Predicate p, int position) {
149     return finiteRank(getComponentID(getVertex(p, position).getID()));
150 }
151
152 /**
153 * Internal method which process the predicate positions ranks.
154 * Will only be called by the finiteRank(int vertexID) method.
155 */
156 protected void processFiniteRanks() {
157     _finiteRanks = new boolean[getNbComponents()];
158     ArrayList<Vertex<Predicate>> scc = null;
159     for (int i = 0 ; i < getNbComponents() ; i++) {
160         scc = getComponent(i);
161         _finiteRanks[i] = true;
162         // NOT OPTIMIZED TODO
163         for (int j = 0 ; (j < scc.size()) && _finiteRanks[i] ; j++) {
164             for (int k = 0 ; (k < scc.size()) && _finiteRanks[i] ; k++) {
165                 try {
166                     if (getEdgeValue(scc.get(j).getID(), scc.get(k).getID()) == true)

```

```

167         _finiteRanks[i] = false;
168     }
169     } catch (NoSuchElementException e) { }
170 }
171 }
172 }
173 }
174 }
175 /**
176  * Converts a vertex to the predicate position string.
177  * @param vertexID The id of the vertex to be converted.
178  * @return A string of form "p[i]"
179  */
180 public String predicatePositionToString(int vertexID) {
181     Predicate p = get(vertexID);
182     int position = 1;
183     boolean end = false;
184     while ((vertexID - position >= 0) && (!end)) {
185         if (p.compareTo(get(vertexID-position)) == 0)
186             position++;
187         else
188             end = true;
189     }
190     position--;
191     return p.getLabel() + "[" + position + "]";
192 }
193
194 /**
195  * Converts the full graph into a string.<br />
196  * It will first contain the vertices values, and then the edges.
197  */
198 @Override
199 public String toString() {
200     StringBuilder result = new StringBuilder();
201     NeighbourEdge<Boolean> edge = null;
202
203     // vertices
204     for (int i = 0 ; i < getNbVertices() ; i++) {
205         result.append(i);
206         result.append(" : ");
207         result.append(predicatePositionToString(i));
208         result.append('\n');
209     }
210     result.append('\n');
211
212     /*while (predicateIndex < getNbVertices()) {
213         positionIndex = 0;
214         while (positionIndex < get(predicateIndex).getArity()) {
215             result.append(predicateIndex + positionIndex);
216             result.append(" : ");
217             result.append(get(predicateIndex).getLabel());
218             result.append("[");
219             result.append(positionIndex);
220             result.append("]");
221             result.append('\n');
222             positionIndex++;
223         }
224         predicateIndex += positionIndex;
225     }*/
226
227     // edges
228     for (int i = 0 ; i < getNbVertices() ; i++) {
229         for (Iterator<NeighbourEdge<Boolean>> iterator = neighbourIterator(i) ; iterator.hasNext() ; ) {
230             edge = iterator.next();
231             result.append(predicatePositionToString(i));
232             if (edge.getValue() == true)
233                 result.append("\tx->\t");
234             else
235                 result.append("\t-->\t");
236             result.append(predicatePositionToString(edge.getIDV()));
237             result.append('\n');
238         }
239     }
240     return result.toString();
241 }
242
243 /**
244  * Fullfills the graph from the set of rules.
245  * @param rules The set of rule.
246  */
247 protected void init(Iterable<Vertex<AtomicRule>> rules) {
248
249     // vertices
250     for (Vertex<AtomicRule> rule : rules) {
251         for (int i = 0 ; i < rule.getValue().getNbAtoms() ; i++)
252             addPredicate(rule.getValue().getPredicate(i));

```

```

253     }
254
255     // edges
256     Vertex<Object> headVertex = null;
257     AtomicRule rule = null;
258     Term neighbour = null;
259     Vertex<Object> vertex = null;
260     NeighbourEdge<Integer> edge = null;
261     NeighbourEdge<Integer> edge2 = null;
262     NeighbourEdge<Integer> edge3 = null;
263     Predicate p,q;
264     Vertex<Predicate> ri = null;
265     Vertex<Predicate> sj = null;
266     int positionP, positionQ;
267     for (Vertex<AtomicRule> vertexRule : rules) {
268         rule = vertexRule.getValue();
269         headVertex = rule.getHead();
270         for (Iterator<NeighbourEdge<Integer> > neighbourIterator = rule.neighbourIterator↵
271             (headVertex.getID());
272             neighbourIterator.hasNext(); ) {
273             edge = neighbourIterator.next();
274             neighbour = rule.getTerm(edge.getIDV()-rule.getNbAtoms());
275             if (rule.isUniversal(rule.getVertex(edge.getIDV()))) {
276                 for (Iterator<NeighbourEdge<Integer> > neighbourIterator2 = rule.↵
277                     neighbourIterator(edge.getIDV());
278                     neighbourIterator2.hasNext(); ) {
279                     edge2 = neighbourIterator2.next();
280                     vertex = rule.getVertex(edge2.getIDV());
281                     if (rule.isBody(vertex)) {
282                         p = (Predicate)(vertex.getValue());
283                         positionP = edge2.getValue();
284                         ri = getVertex(p, positionP);
285                         for (Iterator<NeighbourEdge<Integer> > neighbourIterator3 = rule.↵
286                             neighbourIterator(edge.getIDV()); neighbourIterator3.hasNext↵
287                             (); ) {
288                             edge3 = neighbourIterator3.next();
289                             if (rule.isHead(rule.getVertex(edge3.getIDV()))) {
290                                 q = rule.getPredicate(edge3.getIDV());
291                                 positionQ = edge3.getValue();
292                                 addEdge(p, positionP, q, positionQ);
293                             }
294                         }
295                     }
296                 }
297                 for (Iterator<NeighbourEdge<Integer> > neighbourIterator3 = rule.↵
298                     neighbourIterator(headVertex.getID()); neighbourIterator3.↵
299                     hasNext(); ) {
300                     edge3 = neighbourIterator3.next();
301                     if (rule.isExistential(rule.getVertex(edge3.getIDV()))) {
302                         q = (Predicate)(headVertex.getValue());
303                         positionQ = edge3.getValue();
304                         sj = getVertex(q, positionQ);
305                         if (isEdge(ri.getID(), sj.getID()))
306                             removeEdge(ri.getID(), sj.getID());
307                         addEdge(p, positionP, q, positionQ, true);
308                     }
309                 }
310             }
311         }
312     }
313 }
314
315 /** Internal hash map used to access to the first vertex corresponding to the predicate. ↵
316 */
317 private HashMap<Predicate, Integer> _predicateIndex = new HashMap<Predicate, Integer>();
318 /** Array of same length as the number of components, and valuing true if the scc has a ↵
319     finite rank. */
320 private boolean _finiteRanks[] = null;
321
322 };

```

```

1 package obr;
2
3 import moca.graphs.DirectedSimpleGraph;
4 import moca.graphs.IllegalConstructionException;
5 import moca.graphs.vertices.Vertex;
6 import moca.graphs.edges.IllegalEdgeException;
7 import moca.graphs.edges.Edge;
8 import moca.graphs.edges.NeighbourEdge;
9 import moca.graphs.visu.AcyclicGraphLocalizer;
10 import moca.graphs.visu.PolygonLocalizer;
11 import moca.graphs.visu.PostScriptConverter;
12 import moca.graphs.visu.SCCIdentityFunction;
13
14 import java.util.ArrayList;
15 import java.util.Scanner;
16 import java.util.Iterator;
17 import java.util.regex.Pattern;
18 import java.io.File;
19 //import java.io.OutputStreamWriter;
20 //import java.io.BufferedOutputStream;
21 //import java.io.FileOutputStream;
22
23
24 /**
25  * Unsecure :
26  * if a vertex already in the graph is added once again, it may produce strange behaviour
27  * if an edge is added manually it may produce errors
28  */
29 public class GraphRuleDependencies extends DirectedSimpleGraph<AtomicRule, Boolean> {
30
31     public GraphRuleDependencies() throws IllegalConstructionException {
32         super();
33     }
34
35     public GraphRuleDependencies(String filePath) throws IllegalConstructionException {
36         super();
37         fromFile(filePath);
38     }
39
40     public GraphRuleDependencies(GraphRuleDependencies g) throws IllegalConstructionException ←
41     {
42         super(g);
43     }
44
45     /**
46      * Whenever a vertex is added, the rule is checked for unification against all others.
47      * This provides an always updated graph.
48      */
49     public void addVertex(AtomicRule rule) {
50         super.addVertex(rule);
51         Vertex<AtomicRule> r = getVertex(getNbVertices()-1);
52         Vertex<AtomicRule> r2 = null;
53         for (Iterator<Vertex<AtomicRule>> ruleIterator = vertexIterator(); ruleIterator.↵
54             hasNext(); ) {
55             r2 = ruleIterator.next();
56             try {
57                 if (r.getValue().mayImply(r2.getValue()))
58                     addEdge(r.getID(), r2.getID(), true);
59             }
60             catch (IllegalEdgeException e) {
61                 // the edge already exists
62             }
63             try {
64                 if ((r2 != r) && (r2.getValue().mayImply(r.getValue())))
65                     addEdge(r2.getID(), r.getID(), true);
66             }
67             catch (IllegalEdgeException e) {
68                 // the edge already exists
69             }
70         }
71     }
72
73     public void fromFile(String filePath) {
74         try {
75             String line = null;
76             Scanner scan = new Scanner(new File(filePath));
77             scan.useDelimiter(Pattern.compile("\n"));
78             while (scan.hasNext())
79                 addVertex(new AtomicRule(scan.next()));
80             scan.close();
81         }
82         catch (Exception e) {
83             System.out.println(e);
84             e.printStackTrace();
85         }
86     }
87 }

```

```

85
86 public String toString(){
87     StringBuilder stringBuilder = new StringBuilder();
88     int i = 0;
89     for (AtomicRule rule : this) {
90         stringBuilder.append("R");
91         stringBuilder.append(i);
92         stringBuilder.append(":\t");
93         stringBuilder.append(rule);
94         stringBuilder.append('\n');
95         i++;
96     }
97     stringBuilder.append('\n');
98     Vertex<AtomicRule> rule = null;
99     boolean newelement = false; // TODO
100    for (Iterator<Vertex<AtomicRule>> ruleIterator = vertexIterator() ; ruleIterator.hasNext() ; ) {
101        newelement = false;
102        rule = ruleIterator.next();
103        for (Iterator<NeighbourEdge<Boolean>> neighbour = neighbourIterator(rule.getId() ; neighbour.hasNext() ; ) {
104            if (!newelement) {
105                stringBuilder.append("R");
106                stringBuilder.append(rule.getId());
107                stringBuilder.append("\t-->\t");
108            }
109            stringBuilder.append("R");
110            stringBuilder.append(neighbour.next().getIDV());
111            stringBuilder.append(',');
112            newelement = true;
113        }
114        if (newelement == true)
115            stringBuilder.append('\n');
116    }
117    return stringBuilder.toString();
118 }
119
120 public String stronglyConnectedComponentsToString() {
121     DirectedSimpleGraph<ArrayList<Vertex<AtomicRule>>, Boolean> sccg = ←
122         getStronglyConnectedComponentsGraph();
123     StringBuilder stringBuilder = new StringBuilder();
124     for (int i = 0 ; i < sccg.getNbVertices() ; i++) {
125         stringBuilder.append('C');
126         stringBuilder.append(i);
127         stringBuilder.append(":\t");
128         for (int j = 0 ; j < sccg.getVertex(i).getValue().size() ; j++) {
129             stringBuilder.append(sccg.getVertex(i).getValue().get(j).getID());
130             if (j != sccg.getVertex(i).getValue().size()-1)
131                 stringBuilder.append(',');
132         }
133         stringBuilder.append('\n');
134     }
135     boolean newelement = false;
136     for (int i = 0 ; i < sccg.getNbVertices() ; i++) {
137         newelement = false;
138         for (Iterator<NeighbourEdge<Boolean>> neighbour = sccg.neighbourIterator(i) ; ←
139             neighbour.hasNext() ; ) {
140             if (!newelement) {
141                 stringBuilder.append("C");
142                 stringBuilder.append(i);
143                 stringBuilder.append("\t-->\t");
144             }
145             stringBuilder.append("C");
146             stringBuilder.append(neighbour.next().getIDV());
147             stringBuilder.append(',');
148             newelement = true;
149         }
150         if (newelement)
151             stringBuilder.append('\n');
152     }
153     return stringBuilder.toString();
154 }
155
156 public void sccToPostScript(String filePath) {
157     PostScriptConverter psOutput = new PostScriptConverter(
158         getStronglyConnectedComponentsGraph(),
159         480,640,
160         new AcyclicGraphLocalizer(480,640),
161         SCCIdentityFunction.instance(),
162         "Strongly Connected Components");
163     psOutput.writeToFile(filePath);
164 }
165
166 public void toPostScript(String filePath) {
167     PostScriptConverter psOutput = new PostScriptConverter(
168         this,
169         480,640,

```

```
168         new PolygonLocalizer(480,640),
169         RuleIdentityFunction.instance(),
170         "Graph of Rule Dependencies");
171     psOutput.writeFile(filePath);
172     }
173 };
```



```

1 package obr;
2
3 import java.util.ArrayList;
4 import java.util.Iterator;
5
6 import moca.graphs.vertices.Vertex;
7
8 public class GuardedCheck implements DecidableClassCheck{
9
10     public static final DecidableClassLabel LABEL = new DecidableClassLabel("guarded",false,↵
11         false,true);
12
13     /**
14      * Checks a graph of rule dependencies.
15      * @return the decidable class label if the grd belongs to this decidable class, null ↵
16      * otherwise.
17      */
18     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
19         if (check(grd.getVertexCollection()))
20             return LABEL;
21         return null;
22     }
23
24     /**
25      * Checks only a strongly connected component of the graph of rule dependencies.
26      * @param grd The graph of rule dependencies.
27      * @param scc The strongly connected component to be checked.
28      * @return the decidable class label if the strongly connected component belongs to this ↵
29      * decidable class, null otherwise.
30      */
31     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
32         if (check(grd.getComponent(sccID)))
33             return LABEL;
34         return null;
35     }
36
37     protected boolean check(Iterable<Vertex<AtomicRule> > rules) {
38         //System.out.println("-----GUARDED?-----");
39         AtomicRule rule;
40         int i,nbAtoms,nbVars;
41         boolean guarded;
42         for (Vertex<AtomicRule> vrule : rules) {
43             rule = vrule.getValue();
44             nbAtoms = rule.getNbAtoms();
45             nbVars = rule.getNbUniversalVariables();
46             guarded = false;
47             //System.out.println("rule = "+rule+"\nnbAtoms = "+nbAtoms+"\nnbVars = "+nbVars);
48             for (i = 0 ; (i < nbAtoms) && !guarded ; i++) {
49                 //System.out.println("i = "+i+"\nisHead()? "+rule.isHead(i)+"\nvars = "+rule.↵
50                 variables(i).size());
51                 if ((!rule.isHead(i))
52                     && (nbVars == rule.variables(i).size()))
53                     guarded = true;
54             }
55             if (!guarded)
56                 return false;
57         }
58         return true;
59     }
60 }

```

```
1 package obr;
2
3 /**
4  * Exception thrown by the marking process of some concrete class checks when it fails.
5  * @see StickyCheck#mark
6  * @see WeaklyStickyCheck#mark
7  */
8 public class MarkFailureException extends Exception { };
```

```

1 package obr;
2
3 import java.lang.Comparable;
4
5 /**
6  * Represents a predicate with its label and its arity.<br />
7  */
8 public class Predicate implements Comparable<Predicate> {
9
10     /**
11      * Constructor.
12      * @param label The predicate label.
13      * @param arity The predicate arity.
14      */
15     public Predicate(String label, int arity) {
16         _label = label;
17         _arity = arity;
18     }
19
20     /**
21      * Copy constructor.
22      * @param p The predicate to be copied.
23      */
24     public Predicate(Predicate p) {
25         _label = new String(p.getLabel());
26         _arity = p.getArity();
27     }
28
29     /**
30      * Constructor from a string representation.
31      * @param str The string representation.
32      * @throws UnrecognizedStringFormatException If the string format is not correct.
33      */
34     public Predicate(String str) throws UnrecognizedStringFormatException {
35         fromString(str);
36     }
37
38     /**
39      * Label getter.
40      * @return The predicate label.
41      */
42     public String getLabel() {
43         return _label;
44     }
45
46     /**
47      * Arity getter.
48      * @return The predicate arity.
49      */
50     public int getArity() {
51         return _arity;
52     }
53
54     /**
55      * Label setter.
56      * @param value The new label to set.
57      */
58     public void setLabel(String value) {
59         _label = value;
60     }
61
62     /**
63      * Arity setter.
64      * @param value The new arity to set.
65      */
66     public void setArity(int value) {
67         _arity = value;
68     }
69
70     /**
71      * Converts the predicate into a string.
72      * @return A string representation of the predicate.
73      */
74     @Override
75     public String toString() {
76         return _label + "/" + _arity;
77     }
78
79     /**
80      * Converts from a string.<br />
81      * A well-formatted string is under the form of : <predicate label>/<arity>.
82      * @param str The string representation of a predicate.
83      * @throws UnrecognizedStringFormatException If the string is not well-formatted.
84      */
85     protected void fromString(String str) throws UnrecognizedStringFormatException {
86         String[] substr = str.split("/");

```

```

87         if (substr.length != 2)
88             throw new UnrecognizedStringFormatException();
89         _label = substr[0];
90         _arity = new Integer(substr[1]).intValue();
91     }
92
93     /**
94     * Clones the predicate.
95     * @return A clone.
96     */
97     public Predicate clone() {
98         return new Predicate(new String(_label), _arity);
99     }
100
101     /**
102     * Compares two predicates.<br />
103     * Both label and arity are used.
104     * @return <ul><li>-1 if this < p</li>
105     * <li>0 if this = p</li>
106     * <li>1 if this > p</li></ul>
107     */
108     @Override
109     public int compareTo(Predicate p) {
110         if (getArity() < p.getArity())
111             return -1;
112         else if (getArity() > p.getArity())
113             return 1;
114         else
115             return _label.compareTo(p.getLabel());
116     }
117
118     /**
119     * Overriden to provide an equals-consistent class.
120     * @param o The object to check against.
121     * @return True if this equals to o, false otherwise.
122     */
123     @Override
124     public boolean equals(Object o) {
125         if (!(o instanceof Predicate))
126             return false;
127         return compareTo((Predicate)o) == 0;
128     }
129
130     /**
131     * Hashcode of the predicate.<br />
132     * Hash the label and adds the arity.
133     * @return The hash code of the predicate.
134     */
135     @Override
136     public int hashCode() {
137         return _label.hashCode() + _arity;
138     }
139
140     /** The label of the predicate. */
141     private String _label;
142
143     /** The arity of the predicate. */
144     private int _arity = 1;
145
146 };

```

```

1 package obr;
2
3 import moca.graphs.IllegalConstructionException;
4 import moca.graphs.vertices.Vertex;
5 import moca.graphs.edges.NeighbourEdge;
6 import moca.graphs.edges.IllegalEdgeException;
7
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.NoSuchElementException;
11
12 /**
13  * Checks if a set of rules satisfies the range-restricted property.<br />
14  * I.e., if there is no existential variables.
15  */
16 public class RangeRestrictedCheck implements DecidableClassCheck {
17
18     /** Associated label. */
19     public static final DecidableClassLabel LABEL = new DecidableClassLabel("range-restricted ↔", false, true, true);
20
21     /**
22      * Checks a graph of rule dependencies.
23      * @return The decidable class label if the grd belongs to this decidable class, null otherwise.
24      */
25     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
26         if (check(grd.getVertexCollection()))
27             return LABEL;
28         return null;
29     }
30
31     /**
32      * Checks only a strongly connected component of the graph of rule dependencies.
33      * @param grd The graph of rule dependencies.
34      * @param sccID The id of the strongly connected component to be checked.
35      * @return The decidable class label if the strongly connected component belongs to this ↔ decidable class, null otherwise.
36      */
37     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
38         if (check(grd.getComponent(sccID)))
39             return LABEL;
40         return null;
41     }
42
43     /**
44      * Internal check.
45      * @param rules The set of rule to check.
46      * @return True if there is no existential variables, false otherwise.
47      */
48     protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
49         for (Vertex<AtomicRule> vrule : rules) {
50             for (Iterator<Vertex<Object>> termIterator = vrule.getValue().vertexTermIterator(); termIterator.hasNext(); termIterator.next()) {
51                 if (vrule.getValue().isExistential(termIterator.next()))
52                     return false;
53             }
54         }
55         return true;
56     }
57 }
58 };

```

```
1 package obr;
2
3 import moca.graphs.vertices.Vertex;
4 import moca.graphs.vertices.VertexIdentityFunction;
5
6 public class RuleIdentityFunction extends VertexIdentityFunction<AtomicRule> {
7     public String exec(Vertex<AtomicRule> v) {
8         return "R"+v.getID();
9     }
10    public static RuleIdentityFunction instance() {
11        if (_instance == null)
12            _instance = new RuleIdentityFunction();
13        return _instance;
14    }
15    private static RuleIdentityFunction _instance = null;
16    protected RuleIdentityFunction() { }
17 };
```

```

1 package obr;
2
3 import moca.graphs.IllegalConstructionException;
4 import moca.graphs.vertices.Vertex;
5 import moca.graphs.edges.NeighbourEdge;
6 import moca.graphs.edges.IllegalEdgeException;
7
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.NoSuchElementException;
11
12 /**
13  * Checks if a set of rules satisfies the stickiness property.
14  */
15 public class StickyCheck implements DecidableClassCheck {
16
17     /** Associated label. */
18     public static final DecidableClassLabel LABEL = new DecidableClassLabel("sticky", true, ←
19         false, false);
20
21     /**
22      * Checks a graph of rule dependencies.
23      * @param grd The graph of rule dependencies to check.
24      * @return The decidable class label if the grd belongs to this decidable class, null ←
25      * otherwise.
26      */
27     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
28         if (check(grd.getVertexCollection()))
29             return LABEL;
30         return null;
31     }
32
33     /**
34      * Checks only a strongly connected component of the graph of rule dependencies.
35      * @param grd The graph of rule dependencies.
36      * @param sccID The id of the strongly connected component to be checked.
37      * @return The decidable class label if the strongly connected component belongs to this ←
38      * decidable class, null otherwise.
39      */
40     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
41         if (check(grd.getComponent(sccID)))
42             return LABEL;
43         return null;
44     }
45
46     /**
47      * Internal check.
48      * @param rules The set of rules to check.
49      * @return True if the set of rules satisfies the stickiness property, false otherwise.
50      */
51     protected boolean check(Iterable<Vertex<AtomicRule> > rules) {
52         _positions = new ArrayList<Integer>();
53         Vertex<Object> vertexTerm = null;
54         NeighbourEdge<Integer> edge = null;
55         AtomicRule rule = null;
56
57         // first walk
58         for (Vertex<AtomicRule> vrule : rules) {
59             rule = vrule.getValue();
60             for (Iterator<Vertex<Object> > termIterator = rule.vertexTermIterator(); ←
61                 termIterator.hasNext(); ) {
62                 vertexTerm = termIterator.next();
63                 if (((Term)vertexTerm.getValue()).isVariable()) && (!rule.isHead(vertexTerm)←
64                     )) {
65                     try { mark(vertexTerm, rule); }
66                     catch (MarkFailureException e) { return false; }
67                 }
68             }
69         }
70
71         // second walk
72         for (int i = 0 ; i < _positions.size() ; i++) {
73             for (Vertex<AtomicRule> vrule : rules) {
74                 rule = vrule.getValue();
75                 try {
76                     vertexTerm = rule.getVertexTermFromAtom(rule.getHead().getID(), _positions←
77                         .get(i));
78                     try { mark(vertexTerm, rule); }
79                     catch (MarkFailureException e) { return false; }
80                 }
81                 catch (NoSuchElementException e) { /* nothing to be done */ }
82             }
83         }
84
85         return true;
86     }
87 }

```

```

81
82  /**
83   * Marking processing.
84   * @param term The vertex term to be marked.
85   * @param rule The rule where to mark.
86   * @throws MarkFailureException If the marked variable occurred more than once in the body↵
87   *   of rule.
88   */
89   protected void mark(Vertex<Object> term, AtomicRule rule) throws MarkFailureException {
90       NeighbourEdge<Integer> edge = null;
91       int cpt = 0;
92       for (Iterator<NeighbourEdge<Integer>> iterator = rule.neighbourIterator(term.getID())↵
93           ) ; iterator.hasNext() ; ) {
94           edge = iterator.next();
95           if (rule.isBody(edge.getIDV())) {
96               cpt++;
97               if (cpt >= 2)
98                   throw new MarkFailureException();
99               if (!_positions.contains(edge.getValue()))
100                   _positions.add(edge.getValue());
101           }
102       }
103   }
104
105   /**
106   * Internal list of positions to be checked.
107   */
108   protected ArrayList<Integer> _positions;
109 };

```



```

1 package obr;
2
3 import java.lang.String;
4
5 /**
6  * A term may be either a variable or a constant.<br />
7  * A constant is surround by simple quotes.
8  * Note that two different types of constants exist : the first one is a classical constant ↔
9  *   added from a fact base.
10 * The second one is a constant which have been generated by an existential rule application.
11 * The "existential constants" start by a simple quote immediatly followed by an underscore.
12 */
13 public class Term {
14     /**
15      * Constructor.
16      * @param label The term label.
17      */
18     public Term(String label) {
19         _label = label;
20     }
21
22     /**
23      * Label getter.
24      * @return The term label.
25      */
26     public String getLabel() {
27         return _label;
28     }
29
30     /**
31      * Label setter.
32      * @param value The new label to set.
33      */
34     public void setLabel(String value) {
35         _label = value;
36     }
37
38     /**
39      * Allows to know if a term is a constant.
40      * @return True if the term is a constant, false otherwise.
41      */
42     boolean isConstant() {
43         return (_label.charAt(0) == '\');
44     }
45
46     /**
47      * Allows to know if a term is a variable.
48      * @return True if the term is a variable, false otherwise.
49      */
50     boolean isVariable() {
51         return !isConstant();
52     }
53
54     /**
55      * Allows to know if a term is a constant which has been generated by an existential rule↔
56      *   application.
57      * @return True if the term is a generated constant, false otherwise.
58      */
59     boolean isExistential() {
60         return (isConstant() && (_label.charAt(1)=='_'));
61     }
62
63     /**
64      * Converts the term into a String.
65      * @return The string representation of the term.
66      */
67     @Override
68     public String toString() {
69         return _label;
70     }
71
72     /**
73      * Clones the term.
74      * @return A clone.
75      */
76     @Override
77     public Term clone() {
78         return new Term(new String(_label));
79     }
80
81     /** Term label. */
82     private String _label;
83 }

```

```
1 package obr;  
2  
3 import java.lang.Exception;  
4  
5 /**  
6  * Exception thrown whenever a string is not recognized.  
7  */  
8 public class UnrecognizedStringFormatException extends Exception { };
```

```

1 package obr;
2
3 import moca.graphs.IllegalConstructionException;
4 import moca.graphs.vertices.Vertex;
5 import moca.graphs.edges.NeighbourEdge;
6 import moca.graphs.edges.IllegalEdgeException;
7
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.NoSuchElementException;
11
12 /**
13  * Checks if a set of rules satisfies the weakly-acyclic property.
14  * @see GraphPositionDependencies
15  */
16 public class WeaklyAcyclicCheck implements DecidableClassCheck {
17
18     /** Associated label. */
19     public static final DecidableClassLabel LABEL = new DecidableClassLabel("weakly-acyclic", ←
        false, true, false);
20
21     /**
22      * Checks a graph of rule dependencies.
23      * @param grd The graph of rule dependencies.
24      * @return The decidable class label if the grd belongs to this decidable class, null ←
        otherwise.
25      */
26     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
27         try { _graphPosDep = new GraphPositionDependencies(grd.getVertexCollection()); }
28         catch (IllegalConstructionException e) { return null; }
29         if (_graphPosDep.finiteRank())
30             return LABEL;
31         return null;
32     }
33
34     /**
35      * Checks only a strongly connected component of the graph of rule dependencies.
36      * @param grd The graph of rule dependencies.
37      * @param sccID The id of the strongly connected component to be checked.
38      * @return the decidable class label if the strongly connected component belongs to this ←
        decidable class, null otherwise.
39      */
40     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
41         try { _graphPosDep = new GraphPositionDependencies(grd.getComponent(sccID)); }
42         catch (IllegalConstructionException e) { return null; }
43         if (_graphPosDep.finiteRank())
44             return LABEL;
45         return null;
46     }
47
48     /** The graph of position dependencies generated from the set of rules. */
49     private GraphPositionDependencies _graphPosDep;
50
51 };

```

```

1 package obr;
2
3 import moca.graphs.IllegalConstructionException;
4 import moca.graphs.vertices.Vertex;
5 import moca.graphs.edges.NeighbourEdge;
6 import moca.graphs.edges.IllegalEdgeException;
7
8 import java.util.ArrayList;
9 import java.util.Iterator;
10 import java.util.NoSuchElementException;
11
12 /**
13  * Checks if a set of rules satisfies the weakly-stickiness property.
14  * @see WeaklyAcyclicCheck
15  * @see StickyCheck
16  * @see GraphPositionDependencies
17  */
18 public class WeaklyStickyCheck extends StickyCheck implements DecidableClassCheck {
19
20     /** Associated label. */
21     public static final DecidableClassLabel LABEL = new DecidableClassLabel("weakly-sticky");
22
23     /**
24      * Checks a graph of rule dependencies.
25      * @param grd The graph of rule dependencies.
26      * @return The decidable class label if the grd belongs to this decidable class, null ←
27      *         otherwise.
28      */
29     public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
30         try { _graphPosDep = new GraphPositionDependencies(grd.getVertexCollection()); }
31         catch (IllegalConstructionException e) { }
32         if (check(grd.getVertexCollection()))
33             return LABEL;
34         return null;
35     }
36
37     /**
38      * Checks only a strongly connected component of the graph of rule dependencies.
39      * @param grd The graph of rule dependencies.
40      * @param sccID The id of the strongly connected component to be checked.
41      * @return The decidable class label if the strongly connected component belongs to this ←
42      *         decidable class, null otherwise.
43      */
44     public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
45         try { _graphPosDep = new GraphPositionDependencies(grd.getComponent(sccID)); }
46         catch (IllegalConstructionException e) { }
47         if (check(grd.getComponent(sccID)))
48             return LABEL;
49         return null;
50     }
51
52     /**
53      * Marks a term.
54      * @param term The vertex to mark.
55      * @param rule The rule where to mark.
56      * @throws MarkFailureException If a marked variable occurs more than once in the body at ←
57      *         a position with infinite rank.
58      * @see StickyCheck#mark(Vertex, AtomicRule)
59      * @see GraphPositionDependencies#finiteRank(Predicate, int)
60      */
61     protected void mark(Vertex<Object> term, AtomicRule rule) throws MarkFailureException {
62         NeighbourEdge<Integer> edge = null;
63         int cpt = 0;
64         for (Iterator<NeighbourEdge<Integer>> iterator = rule.neighbourIterator(term.getID() ←
65             ); iterator.hasNext(); ) {
66             edge = iterator.next();
67             if (rule.isBody(edge.getIDV())) {
68                 cpt++;
69                 if ((cpt >= 2) && (!_graphPosDep.finiteRank(rule.getPredicate(edge.getIDV()), ←
70                     edge.getValue()))
71                     throw new MarkFailureException();
72                 if (!_positions.contains(edge.getValue()))
73                     _positions.add(edge.getValue());
74             }
75         }
76     }
77
78     /** Graph of position dependencies created from the set of rules. */
79     private GraphPositionDependencies _graphPosDep;
80
81 };

```