Swan ROCHER

Outils d'Analyse d'une Base de Règles

2012 Université Montpellier 2

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

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 peut être vue comme un ensemble de règles qui peuvent être plus ou moins complexes.

Par exemple, si nous nous contentons des données suivantes :
– "Jean est un homme."
– "Jean est un des parents de Tom."
et que nous souhaitons obtenir une réponse à la requête "Jean est il le père de Tom ?", celle-ci sera négative. Par contre si nous ajoutons la règle : "**Si** un homme est le parent d'un individu **alors** il est son père.", la réponse sera positive.

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 est le père de x_1 qui a également un père x_3, etc...

Or savoir si un ensemble de règles n'amènera jamais ce genre de situation est un problème qui est également non décidable. 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 à quelles classes appartiennent les règles, 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 $p\backslash n$ 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 $a = p(a_1, a_2, ..., a_n)$ associe un terme à chaque position d'un prédicat $p\backslash n$. 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 $\forall\ i \in [1, n]$ $a_i = p_i(a_{i1}, a_{i2}, ..., a_{in_i})$ un atome de prédicat $p_i\backslash n_i$.

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 ω 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 $T_A = \{t_j \in dom(A)\}$
- $E_A = \{(i, t_j) : \forall a_i \in A, \forall t_j \in dom(a_i)\}$
- $\omega : E_A \to \mathbb{N}$ 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 $R : \forall x_i(H \to \exists z_j(C))$.

On note :
- $dom(R) = dom(H) \cup dom(C)$, le domaine de R
- $var(R) = var(H) \cup var(C)$, les variables de R
- $cst(R) = cst(H) \cup cst(C)$, les constantes de R
- $fr(R) = var(H) \cap var(C)$, l'ensemble des variables frontières de R
- $cutp(R) = fr(R) \cup cst(R)$, 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 $T_R = \{t_j \in dom(R)\}$

– $E_R = \{(i, t_j) : \forall r_i \in R, \forall t_j \in dom(R_i)\}$
– $\omega : E_R \to [1, p]$ telle que $\omega(i, t_j) = j : \forall (i, t_j) \in E_R$
– $\chi : P_R \to \{1, 2\}$ 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 $R : \forall x \forall y\ (salle(x) \wedge date(y) \wedge reservee(x, y) \to \exists z\ (cours(z) \wedge aLieu(z, x, y)))$ peut être visualisée de la façon suivante :

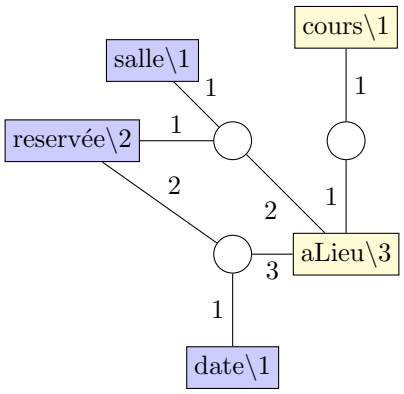

FIGURE 2.2 – Exemple de représentation d'une règle

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.

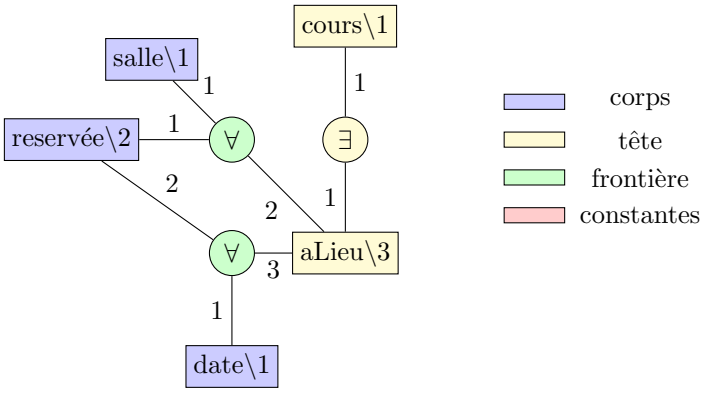

FIGURE 2.3 – Les différents éléments d'une règle

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 algorithme 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 équivalent.

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

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

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 conlusions atomiques équivalent à R

```
1 H_0^A ← H
2 C_0^A ← p_R({x_i ∈ var(R)})
3 R^A ← {(H_0^A, C_0^A)}
4 for all atome c_i ∈ C do
5       R_i^A ← (C_0^A, c_i)
6       R^A ← R^A ∪ {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 considèrerons 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 bouléenne 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 juqu'à ce que la réponse à 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 sur l'unification).

Que l'on considère le chaînage avant ou arrière, la notion de dépendance de règles est très importante. En effet lors de la génération de nouveaux faits seules certaines règles ont besoin d'être appliquées et il en est de même lors de la réécriture de la requête.

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 :
- $V_{GRD} = \{R_i \in B_R\}$,
- $E_{GRD} = \{(R_i, R_j) : \exists$ 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.

Il faut noter que l'unification de règles est de façon générale un problème NP-complet, mais ici nous décidons de ne traiter que les règles à conclusion atomique (voir section 2.7) et ainsi nous simplifions grandement le problème sans que notre analyse en soit perturbée.

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 μ telle que :
- $\mu : var(a_1) \cup var(a_2) \rightarrow dom(a_1) \cup dom(a_2)$
- $\mu(a_1) = \mu(a_2)$

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

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

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

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

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'exitence 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 ∈ H_1 do
3      if UnificationLocale(a, R) ≠ échec then
4          localementUnifiable[a] ← vrai
5      else
6          localementUnifiable[a] ← faux
7      end if
8  end for
9  ▷ Initialisation du tableau contenant les positions des variables existentielles de C_2
10 E ← {i : c_i est une ∃ − var de C_2}
11 ▷ Extension des ensembles
12 for all sommet atome a ∈ H_1 : localementUnifiable[a] = noir do
13     if Q ← Extension(H_1, a, localementUnifiable, E) ≠ échec then
14         if UnificationLocale(Q, R) ≠ échec then
15             return succès
16         end if
17     end if
18     localementUnifiable[a] ← faux
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, $localementUnifiable$: tableau de taille égal au nombre d'atomes dans H_1 tel que $localementUnifiable[a] = vrai$ 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 ∈ H1 do
3      if localementUnifiable[a] = vrai then
4          for all sommet terme t ∈ voisins(a) do
5              couleur[t] = blanc
6          end for
7          couleur[a] = blanc
8      end if
9  end for
10 couleur[a] ← noir
11 Q ← {a} ▷ conjonction d'atomes à traiter
12 attente ← {a} ▷ file d'attente du parcours
13 while attente ≠ ∅ do
14     u ← haut(attente)
15     if u est un atome then
16         for all i ∈ E do
17             v ← voisin(u, i)
18             if v est une constante then
19                 return échec
20             else if couleur[v] = blanc then
21                 ▷ v est une ∀ − var non marquée par le parcours
22                 couleur[v] ← noir
23                 attente ← attente ∪ {v}
24             end if
25         end for
26     else
27         ▷ u est un terme
28         for all v ∈ voisins(u) do
29             if localementUnifiable[v] = faux then
30                 return échec
31             else
32                 if couleur[v] = blanc then
33                     couleur[v] ← noir
34                     attente ← attente ∪ {v}
35                     Q ← Q ∪ {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 $\bigcirc(nombre\ d'atomes\ \times arite\ max\ de\ H_1)$. Or ici, la complexité ne dépend pas de cette arité max, mais uniquement de larité du prédicat de la conclusion C_2.

Le dernier algorithme est quant à lui 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 séparement, et permettre une préselection des atomes (qui vont servir de racine pour l'extension). 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 ∈ H1 do
3      if prédicat(a) ≠ prédicat(C2) then
4          return échec
5      end if
6  end for
7  u ← ∅   ▷ substitution
8  for all terme t_i ∈ C2 do
9      ▷ def : a_i = terme de a en position i
10     E ← {a_i : ∀ atome a ∈ H1}
11     if t_i est une constante then
12         if ∃ v ∈ E : v est une constante et v ≠ t_i, ou v est une ∃ − variable then
13             return échec
14         else
15             u ← {(v, t_i) : v ∈ E et v ≠ t_i}
16         end if
17     else if t_i est une ∃ − variable then
18         if ∃ v ∈ E : v est une ∃ − variable et v ≠ t_i, ou v est une constante then
19             return échec
20         else
21             u ← {(v, t_i) : v ∈ E et v ≠ t_i}
22         end if
23     else
24         if ∃ v1, v2 ∈ E : v1 ≠ v2 et v1, v2 ne sont pas des ∀ − variables then
25             return échec
26         else if ∃ c ∈ E : c est une constante then
27             u ← {(v, c) : v ∈ E ∪ {t_i} et v ≠ c}
28         else
29             u ← {(v, t_i) : v ∈ E et v ≠ t_i}
30         end if
31     end if
32     H1 ← u(H1)
33     C2 ← u(C2)
34 end for
35 return succès
```

3.2.3 Complexites

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

et :

– $n = k + t \leq (k + 1) \times p$: nombre de sommets dans G_R
– $m \leq k \times p$: nombre d'arêtes dans G_R

Avec notre représentation nous avons donc les complexités suivantes :
– Parcourir les sommets prédicats : $\bigcirc(k)$.
– Parcourir tous les sommets : $\bigcirc(n)$.
– Accéder au i^{eme} voisin d'un sommet : $\bigcirc(1)$.

Ainsi, les complexités en temps d'exécution des algorithmes nécessaires à l'unification sont les suivantes :
– UnificationLocale (pire des cas) = $\bigcirc(k \times p)$
– Extension (pire des cas) = $\bigcirc(k \times p)$
– Unification (pire des cas) = $\bigcirc(k^2 \times p)$

Chapitre 4

Classes de règles

Une fois le graphe de dépendances des règles construit, il nous faut déterminer les classes auxquelles appartiennent les règles. Nous allons vois que celles-ci sont divisées en deux types, les classes abstraites et concrètes.

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 [3]).

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 vérifiant des propriétés sur la forme des règles ou sur un ensemble des règles, permet de s'assurer que celui-ci appartient à certaines des classes abstraites.

De plus, la liste des classes concrètes citées ci-après n'est pas exhaustive, seules les classes détectées par notre outil d'analyse sont présentées.

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 apparaîssant en position q_k il existe un arc *spécial* de p_i vers q_k.

Un des sommets de ce graphe (et donc une position de prédicat) est dite *de rang fini* s'il n'existe aucun circuit contenant passant par un arc *spécial* et par ce sommet.

Si toutes les positions de prédicat sont de *rang fini* alors l'ensemble de règles est dit *faiblement acyclique* et appartient à FES ([2]).

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 précisemment, 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'apparait pas dans la conclusion de R alors on marque chaque occurence 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 apparaîssant en position p_i dans la conclusion de R', on marque chaque occurence de x dans l'hypothèse de R'.

Ensuite si aucune règle ne contient plusieurs occurence 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 Guardée

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* ([1]).

4.2.5 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* ([1]).

4.2.6 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* .

4.2.7 Hypothèse atomique

Les règles ne contenant que des hypothèses atomiques s'assurent que la règle est *gardée (4.2.4)*, 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* ([**?**]).

4.2.8 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* ([**?**]).

4.2.9 Règle déconnectée

Une règle est dite déconnectée si sa frontière est vide, cette classe est donc une spécialisation des règles à *domaine restreint (4.2.8)*, à *frontière gardée (4.2.5)* 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 ([3]).

4.2.10 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)*.

4.2.11 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 *de rang fini* si elle n'apparait 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.

4.3 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.

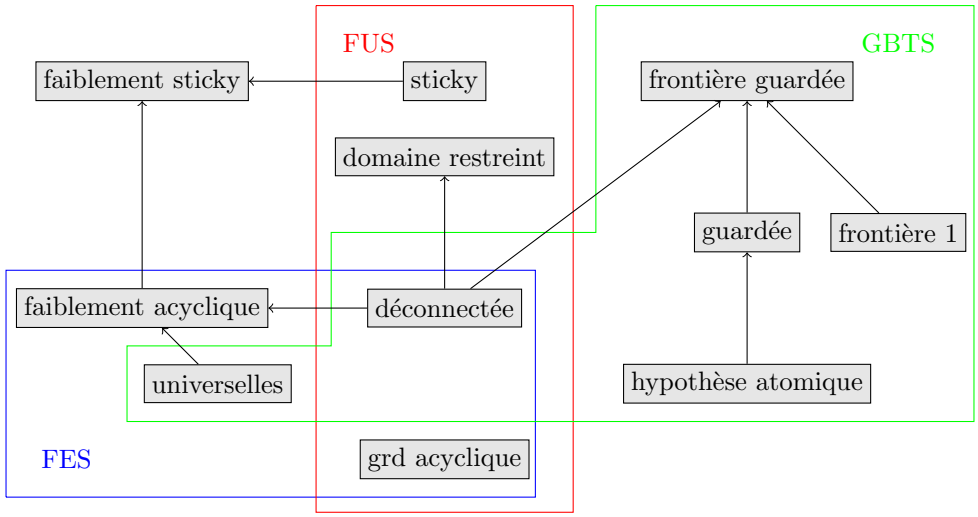

FIGURE 4.1 – Schéma récapitulatif des classes de règles

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. En effet par définition des composantes, il paraît naturel d'exécuter nos algorithmes presque indépendamment sur chacune d'elle. La seule condition est de ne traiter une composante qu'une fois que chacun de ses prédécesseurs a été traité. Ainsi nous imposons tout de même un ordre sur celles-ci et il faut donc que l'on puisse effectuer notre requête en le respectant.

Donc 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 $etiquette : V_C \to \{FES, GBTS, FUS\}$ 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).

Nous avons donc bien mis en place un système permettant de vérifier autant que possible s'il existe un algorithme permettant d'obtenir une réponse à une requête sur un ensemble de règles en temps fini.

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* était imposé comme contrainte 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 (4.2.11)) 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 :
$atome_1; atome_2; ...; atome_n - - > atome_c$
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 : $p_i(t_{i1}, t_{i2}, ..., t_{ik})$
Les termes sont interprêtés comme des constantes s'ils sont encadrés par des simple guillemets.
Par exempe 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 débuter par //. Ici les règles sont sous le format suivant :
[!]$atome_c$:- $atome_h$.
Le point d'explamation 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

6.0.1 Gestion du projet

Comme signalé dans les outils utilisés (section 5.1) git a été utilisé comme gestionnaire de version. La communication a quant à elle était principalement effectuée par courriels, et des réunions avec les encadrants ont été organisées presque hebdomadairement.

6.0.2 Problèmes rencontrés

Le principal problème a été l'abandon du projet par l'un des membres du groupe. De plus, il a été relativement difficile de prendre du recul rapidement. Ainsi certaines implémentations auraient sans doute été faites différemment avec une meilleure organisation du temps.

6.0.3 Contributions

Malgré tout, les contraintes du projet ont été respectées. Tout d'abord, afin de subvenir aux besoins de l'implémentation nous avons donc programmé une bibliothèque adaptée pour nos graphes, et nous avons conçu un algorithme permettant de vérifier la dépendance entre deux règles afin de pouvoir construire le graphe de dépendances associé à un ensemble. De plus une grande partie des classes concrètes exhibées à ce jour est reconnue 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. Comme prévu, une entrée à partir des fichiers respectant un format spécifique interne ou le format Datalog a été ajoutée. Et enfin nous avons également trouvé agréable de mettre en place une sortie PostScript minimale pour la visualisation des différents graphes, ainsi qu'une entrée

6.0.4 Perspectives

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. Calì, G. Gottlob, and A. Pieris. Query answering under non-guarded rules in datalog+/-. *Web Reasoning and Rule Systems*, pages 1–17, 2010.

[3] M.L. Mugnier. Ontological query answering with existential rules. *Web Reasoning and Rule Systems*, pages 2–23, 2011.

```
import moca.graphs.*;
import moca.graphs.vertices.*;
import moca.graphs.edges.*;
import java.lang.Exception;
import java.util.Iterator;
import java.util.ArrayList;
import obr.*;

public class Main {
    public static void main(String args[]) {
        try {
            System.out.println("Reading file : "+args[0]);
            GraphRuleDependencies grd = null;
            String[] filePathSubs = args[0].split("\\.");
            if (filePathSubs.length == 1)
                grd = new GraphRuleDependencies(args[0]);
            else if (filePathSubs[filePathSubs.length-1].equals("dtg")) {
                System.out.println("Invoking DTG parser...");
                grd = DTGParser.parseRules(args[0]);
            }

            System.out.println("\n[1] GRAPH OF RULE DEPENDENCIES\n\n"+grd);
            System.out.println("\n\n[2] STRONGLY CONNECTED COMPONENTS\n");
            System.out.println(grd.stronglyConnectedComponentsToString());

            GRDAnalyser analyser = new GRDAnalyser(grd);
            analyser.addDecidableClassCheck(new DisconnectedCheck());
            analyser.addDecidableClassCheck(new RangeRestrictedCheck());
            analyser.addDecidableClassCheck(new AtomicHypothesisCheck());
            analyser.addDecidableClassCheck(new WeaklyAcyclicCheck());
            analyser.addDecidableClassCheck(new FrontierOneCheck());
            analyser.addDecidableClassCheck(new GuardedCheck());
            analyser.addDecidableClassCheck(new FrontierGuardedCheck());
            analyser.addDecidableClassCheck(new DomainRestrictedCheck());
            analyser.addDecidableClassCheck(new StickyCheck());
            analyser.addDecidableClassCheck(new WeaklyStickyCheck());

            System.out.println("\n\n[3] DIAGNOSTIC\n");
            analyser.process();
            System.out.println(analyser.diagnostic());

            if (args.length >= 2)
                grd.toPostScript(args[1]);
            if (args.length >= 3)
                grd.sccToPostScript(args[2]);
        }
        catch (Exception e) {
            System.out.println(e);
            e.printStackTrace();
        }
    }
};
```

```java
package obr;

import java.lang.String;
import java.lang.Iterable;
import java.util.NoSuchElementException;
import java.util.ArrayList;

/**
 * Represents an atom.<br />
 * Note that instances of this class can be returned by the atom conjunction class, but this will imply
 * the atom creation. I.e., this class is not used to store atom information, but to get a convenient
 * instance.
 */
public class Atom implements Iterable<Term> {

    /** Represents the begin of a term list when the atom is under a String format. */
    public static final String BEGIN_TERM_LIST = "\\(";
    public static final String BEGIN_TERM_LIST_W = "(";
    /** Represents the end of a term list when the atom is under a String format. */
    public static final String END_TERM_LIST = "\\)";
    public static final String END_TERM_LIST_W = ")";
    /** Represents the term separator when the atom is under a String format. */
    public static final String TERM_SEPARATOR = ",";

    /**
     * Constructor. <br />
     * Note that there is no default constructor since this class should not be used directly but be get by an
     * atom conjunction method.
     * @param predicate The predicate of the atom to be instantiated.
     */
    public Atom(Predicate predicate) {
        _predicate = predicate;
        _terms = new ArrayList<Term>(_predicate.getArity());
        for (int i = 0 ; i < getArity() ; i++)
            _terms.add(null);
    }

    /**
     * Predicate getter.
     * @return The atom predicate.
     */
    public Predicate getPredicate() {
        return _predicate;
    }

    /**
     * Predicate setter.
     * @param value The new predicate to set.
     */
    public void setPredicate(Predicate value) {
        _predicate = value;
        _terms.clear();
        for (int i = 0 ; i < getArity() ; i++)
            _terms.add(null);
    }

    /**
     * Convenient method to get the predicate arity.
     * @return The atom arity.
     */
    public int getArity() {
        return _predicate.getArity();
    }

    /**
     * Term getter.
     * @param i The term index inside the atom.
     * @return The term corresponding to this index.
     * @throws NoSuchElementException If i is out of range.
     */
    public Term get(int i) throws NoSuchElementException {
        if ((i >= getArity()) || (i < 0))
            throw new NoSuchElementException();
        return _terms.get(i);
    }

    public boolean contains(Term term) {
        for (Term thisterm : this) {
            if (term.getLabel().compareTo(thisterm.getLabel()) == 0)
                return true;
        }
        return false;
    }
```

```
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     public int getNbVariables() {
101         int result = 0;
102         for (Term term : this)
103             if (term.isVariable())
104                 result++;
105         return result;
106     }
107
108     public ArrayList<Term> variables() {
109         ArrayList<Term> result = new ArrayList<Term>();
110         for (Term term : this)
111             if (term.isVariable())
112                 result.add(term);
113         return result;
114     }
115
116     /**
117      * Term setter.
118      * @param i The term index inside the atom.
119      * @param t The new term to be set instead of the old one.
120      * @throws NoSuchElementException If i is out of range.
121      */
122     public void set(int i, Term t) throws NoSuchElementException {
123         if ((i >= getArity()) || (i < 0))
124             throw new NoSuchElementException();
125         _terms.set(i,t);
126     }
127
128     /**
129      * Converts the atom into a String.
130      */
131     @Override
132     public String toString() {
133         StringBuilder string = new StringBuilder(getPredicate().getLabel());
134         string.append(BEGIN_TERM_LIST_W);
135         for (int i = 0 ; i < getArity()-1 ; i++) {
136             string.append(get(i));
137             string.append(TERM_SEPARATOR);
138         }
139         if (getArity() >= 1)
140             string.append(get(getArity()-1));
141         string.append(END_TERM_LIST_W);
142         return string.toString();
143     }
144
145     /**
146      * Access to the nested Iterator over terms instance.
147      * @return The iterator over terms corresponding to this atom instance.
148      */
149     public java.util.Iterator<Term> iterator() {
150         return new Iterator();
151     }
152
153     /** Predicate of the atom. */
154     private Predicate _predicate;
155     /** List of terms. */
156     private ArrayList<Term> _terms;
157
158     /**
159      * Nested class used to iterate over the terms.<br />
160      * Note that the remove() operation is not supported, and will always throw an ↩
          UnsupportedOperationException.
161      */
162     public class Iterator implements java.util.Iterator<Term> {
163         /**
164          * Allows to know if there is still element to be iterated.
165          * @return True if the next call to next() method will not throw a ↩
              NoSuchElementException, false otherwise.
166          */
167         @Override
168         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 ↩
        into this partition.
27  * But the vertex class provides a global ID which will can be used to identify an atom.<br ↩
        />
28  * The second partition represents the terms. A term may be either a variable or a constant.
29  * Since terms are also stored into vertices, they have a global ID.<br />
30  * The edges are used to connect atoms with terms. And their value is the position of the ↩
        term inside the atom.
31  * A well-built atom conjunction will ensure that there is one edge for each position of each↩
         atom predicate and that
32  * they are added in their position ordering.<br/>
33  * </p>
34  *
35  * <p>
36  * <b>About IDs :</b><br />
37  * Each vertex (thus, each atom and each term) has a different ID, which can be get by Vertex↩
        .getID() method.
38  * But since the internal structure provides a bipartition, atoms can be enumerated from 0 to↩
         getNbAtoms()-1, and terms
39  * from 0 to getNbTerms()-1.<br />
40  * To convert a global ID to a local one apply the following computation :
41  * <ul><li>if atom : local ID = global ID</li>
42  * <li>if term : local ID = global ID - getNbAtoms()</li></ul>
43  * </p>
44  *
45  */
46 public class AtomConjunction implements Iterable<Atom> {
47
48     /**
49      * Represents the separator between atoms when the conjunction is under a String format.
50      */
51     public static final String ATOM_SEPARATOR   =   new String(";");
52
53     /**
54      * Represents the separator between terms when an atom is under a String format.
55      * @see Atom#TERM_SEPARATOR
56      */
57     public static final String TERM_SEPARATOR   =   Atom.TERM_SEPARATOR;
58
59     /**
60      * Represents the begin of a term list when an atom is under a String format.
61      * @see Atom#BEGIN_TERM_LIST
62      */
63     public static final String BEGIN_TERM_LIST  =   Atom.BEGIN_TERM_LIST;
64
65     /**
66      * Represents the end of a term list when an atom is under a String format.
67      * @see Atom#END_TERM_LIST
68      */
69     public static final char END_TERM_LIST      =   Atom.END_TERM_LIST.charAt(Atom.↩
            END_TERM_LIST.length()-1);
70
71     /**
72      * Default constructor.<br />
73      * The atom conjunction will be empty.
74      */
75     public AtomConjunction() {
76         try {
77             _graph = new BipartedGraph<Object,Integer>(
78                 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     /**
114      * Number of atoms getter.
115      * @return The current number of atoms.
116      */
117     public int getNbAtoms() {
118         return _graph.getNbVerticesInFirstSet();
119     }
120
121     /**
122      * Number of terms getter.
123      * @return The current number of terms.
124      */
125     public int getNbTerms() {
126         return _graph.getNbVerticesInSecondSet();
127     }
128
129     /**
130      * Allows to know if a vertex is an atom.
131      * @param v The vertex to be checked.
132      * @return True if v is an atom, false otherwise.
133      */
134     public boolean isAtom(Vertex<Object> v) {
135         if (v == null)
136             return false;
137         if (v.getID() < getNbAtoms())
138             return true;
139         return false;
140     }
141
142     /**
143      * Allows to know if a vertex is a term.
144      * @param v The vertex to be checked.
145      * @return True if v is a term, false otherwise.
146      */
147     public boolean isTerm(Vertex<Object> v) {
148         if (v == null)
149             return false;
150         if (v.getID() < getNbAtoms())
151             return false;
152         return true;
153     }
154
155     /**
156      * Creates a new atom instance for the atom whose id matches i.<br />
157      * The structure does not contain atom instances, that is why only a new instance can be ↩
           generated.
158      * Furthermore, any modifications on this atom instance will <b>NOT</b> be applied on the↩
           atom representation in the
159      * conjunction.
160      * @param i The id of the atom.
161      * @return A new instance of atom class.
162      * @throws NoSuchElementException If the id does not match in the atom conjunction.
163      */
```

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

```
            nbAtoms = getNbAtoms();
        for (int i = 0 ; i < nbAtoms -1; i++) {
            string.append(getAtom(i));
            string.append(ATOM_SEPARATOR);
        }
        string.append(getAtom(nbAtoms-1));
        return string.toString();
    }
    public String toStringExcluding(int exclude) {
        StringBuilder string = new StringBuilder();
        final int nbAtoms = getNbAtoms();
        for (int i = 0 ; i < nbAtoms; i++) {
            if (i != exclude) {
                string.append(getAtom(i));
                string.append(ATOM_SEPARATOR);
            }
        }
        string.deleteCharAt(string.length()-1);
        return string.toString();
    }



    /**
     * Adds an atom into the atom conjunction.<br />
     * The internal representation of an atom is a graph vertex with predicate value, that is↩
         why only a predicate is
     * needed.
     * @param predicate The predicate of the new atom.
     * @return The global index of the new atom in the conjunction.
     */
    public int addAtom(Predicate predicate) {
        _graph.addInFirstSet(predicate);
        return _graph.getNbVerticesInFirstSet() - 1;
    }

    public void removeCleanlyAtom(int atomID) throws NoSuchElementException {
        final Vertex<Object> atom = getVertexAtom(atomID);
        int neighbourAtomID;
        int currentTerm;
        final int nbTerms = _graph.getNbNeighbours(atomID);
        for (currentTerm = 0 ; currentTerm < nbTerms ; currentTerm++) {
            neighbourAtomID = getVertexTermFromAtom(atomID,currentTerm).getID();
            if (_graph.getNbNeighbours(neighbourAtomID) == 1) {
                _graph.removeVertex(neighbourAtomID);
                currentTerm--;
            }
        }
        removeAtom(atomID);
    }

    protected void removeAtom(int atomID) throws NoSuchElementException {
        _graph.removeVertex(atomID);
    }

    public int getNbConstants() {
        int result = 0;
        final int nbTerms = getNbTerms();
        for (int i = 0 ; i < nbTerms ; i++)
            if (getTerm(i).isConstant())
                result++;
        return result;
    }

    public int getNbVariables() {
        int result = 0;
        final int nbTerms = getNbTerms();
        for (int i = 0 ; i < nbTerms ; i++)
            if (getTerm(i).isVariable())
                result++;
        return result;
    }

    public VertexCollection<Object> constants() {
        VertexArrayList<Object> result = new VertexArrayList<Object>();
        final int nbTerms = getNbTerms();
        for (int i = 0 ; i < nbTerms ; i++)
            if (getTerm(i).isConstant())
                result.add(getVertexTerm(i));
        return result;
    }

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

```
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 itertor 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.<br />
613      * If the atom conjunction is well built, the terms will be iterated in their positions ordering.
614      * @param atomID The global id of the atom.
615      * @return An iterator over the vertex terms connected to the atom.
616      * @see .NeighbourIterator
617      */
618     public NeighbourIterator vertexTermIteratorFromAtom(int atomID) {
619         return new NeighbourIterator(_graph.neighbourIterator(atomID));
620     }
621 
622     /**
623      * Access to an iterator over the vertex atoms which are connected to the specified term.
624      * @param termID The global id of the term.
625      * @return An iterator over the vertex atoms connected to the term.
626      * @see .NeighbourIterator
627      */
628     public NeighbourIterator vertexAtomIteratorFromTerm(int termID) {
629         return new NeighbourIterator(_graph.neighbourIterator(termID));
630     }
631 
632     /**
633      * Access to an iterator over the edges from a specific vertex.
634      * @param vertexID The global id of the vertex which is the source of edges.
635      * @return An iterator over the edges connected to the vertex.
636      */
637     public Iterator<NeighbourEdge<Integer> > neighbourIterator(int vertexID) {
638         return _graph.neighbourIterator(vertexID);
639     }
640 
641     /**
642      * The structure of the atom conjunction.<br />
643      * The first vertices partition is used to store atoms (the vertices contain only predicates).
644      * And the second one to store terms.
645      * An edge represents a connection between an atom and a term, and its value the position of the term in the atom.
646      */
647     protected BipartedGraph<Object,Integer> _graph = null;
648 
649 
650     /**
651      * Iterator over the atoms of the conjunction.<br />
652      * Since atoms are not stored into the internal structure, the remove operation is not supported.
653      */
654     public class AtomIterator implements Iterator<Atom> {
655 
656         /**
657          * 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    };
```

```java
package obr;

import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.IllegalEdgeException;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Checks if all rules in the GRD or in one of its strongly connected components satisfy the atomic-hypothesis property.
 * I.e., if all rules contain only one atom in their body.
 */
public class AtomicHypothesisCheck implements DecidableClassCheck {

    /**
     * Atomic-hypothesis property label.
     */
    public static final DecidableClassLabel LABEL = new DecidableClassLabel("atomic-hypothesis",true,false,true);

    /**
     * Checks a graph of rule dependencies.
     * @return the decidable class label if the grd belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return The decidable class label if the strongly connected component belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    /**
     * Internal check.
     * @return True if the set of rules satisfies the atomic-hypothesis property, false otherwise.
     */
    protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
        for (Vertex<AtomicRule> vrule : rules) {
            if (vrule.getValue().getNbAtoms() > 2)  // one for the head, the other for the body
                return false;
        }
        return true;
    }

};
```

```java
package obr;

import moca.graphs.BipartedGraph;
import moca.graphs.vertices.VertexCollection;
import moca.graphs.vertices.Vertex;
import moca.graphs.vertices.VertexArrayList;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.UndirectedNeighboursLists;
import moca.graphs.edges.IllegalEdgeException;
import moca.lists.Fifo;

import java.lang.String;
import java.lang.Integer;

import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.ArrayList;
import java.util.ConcurrentModificationException;

/**
 * Represents an atomic rule.<br />
 * A rule is an atom conjunction which adds the notions of "body" and "head".<br />
 * An atomic rule contains only one atom in its head.
 * The structure of this class is the same as a classical atom conjunction but its head pointer.
 * @see AtomConjunction
 */
public class AtomicRule extends AtomConjunction {

    /** Represents the separator between the body and the head when the rule is under a String format. */
    public static final String HEAD_SEPARATOR   =   new String("-->");

    /**
     * Default constructor.
     * It creates an empty rule.
     */
    public AtomicRule() {
        super();
    }

    /**
     * Constructor from a string representation.
     * @param stringRepresentation The rule under a string format.
     * @see #fromString(String)
     */
    public AtomicRule(String stringRepresentation) {
        super();
        try {
            fromString(stringRepresentation);
        }
        catch (UnrecognizedStringFormatException e) { }
    }

    /**
     * Head vertex getter.
     * @return The head vertex.
     */
    public Vertex<Object> getHead() {
        return _head;
    }

    public void setHead(int atomID) {
        _head = getVertex(atomID);
    }

    /**
     * Converts the head into an atom instance.
     * @see AtomConjunction#getAtom(int)
     */
    public Atom getHeadAtom() {
        return getAtom(_head.getID());
    }

    /**
     * Allows to know if a vertex is in the body.
     * @param v The vertex to be checked.
     * @return True if v belongs to the body, false otherwise.
     * @throws NoSuchElementException If v is null or if its id does not match in the rule.
     */
    public boolean isBody(Vertex<Object> v) throws NoSuchElementException {
        if (v == null)
            throw new NoSuchElementException();
        if (v == _head)
            return false;
        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&lt;Object&gt;).
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&lt;Object&gt;).
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&lt;Object&gt; 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 ↩
              rule.
155      * A term is said to be in the frontier iff it belongs to the body <b>and</b> to the head↩
              of the rule.
156      * @return An array list of Vertex&lt;Object&gt; which contains all terms in the frontier↩
              .
157      */
158     public ArrayList<Vertex<Object> > frontierTerms() {
159         NeighbourEdge<Integer> edge = null;
160         ArrayList<Vertex<Object> > result = new ArrayList<Vertex<Object> >();
161         for (Iterator<NeighbourEdge<Integer> > iterator = neighbourIterator(_head.getID()) ; ↩
                iterator.hasNext() ; ) {
162             edge = iterator.next();
163             if ((isBody(edge.getIDV())) && (!result.contains(getVertex(edge.getIDV()))))
164                 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     /**
207      * Allows to know if a vertex belongs to the frontier of a rule.
208      * A term is said to be in the frontier iff it belongs to the body <b>and</b> to the head↩
           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 ↩
           rule, or it is an atom vertex.
211      */
212     public boolean isFrontier(Vertex<Object> v) throws NoSuchElementException {
213         if ((v == null) || /*(*/(v.getID() < getNbAtoms()))/* || (v.getID() > getNbVertices()↩
               )) */
214             throw new NoSuchElementException();
215         return isBody(v) && isHead(v);
216     }
217 
218     /**
219      * Checks if a vertex is an existential variable.
220      * An existential variable is a variable which appears only in the head of the rule.
221      * @param v The vertex to be checked.
222      * @return True if the vertex is an existential variable, false otherwise.
223      */
224     public boolean isExistential(Vertex<Object> v) {
225         if ((v == null) || (v.getID() < getNbAtoms()))
226             return false;
227         return !((((Term)(v.getValue())).isConstant()) || isBody(v));
228     }
229 
230     /**
231      * Checks if a vertex is a universal variable.
232      * A universal variable is a variable which appears at least in the body of the rule.
233      * @param v The vertex to be checked.
234      * @return True if the vertex is a universal variable, false otherwise.
235      */
236     public boolean isUniversal(Vertex<Object> v) {
237         if ((v == null) || (v.getID() < getNbAtoms()))
238             return false;
239         return (!((((Term)(v.getValue())).isConstant()) && isBody(v));
240     }
241 
242     public boolean isExistential(int i) {
243         try {
244             return isExistential(getVertexTerm(i));
245         }
246         catch (NoSuchElementException e) {
247             return false;
248         }
```

```
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 ↩
           rule.
265      */
266     public int[] existentialIndex() {
267         int cpt = 0;
268         int arity = ((Predicate)(_head.getValue())).getArity();
269         for (int i = 0 ; i < arity ; i++) {
270             if (isExistential(getVertexTermFromAtom(_head.getID(),i)))
271                 cpt++;
272         }
273         int result[] = new int[cpt];
274         cpt = 0;
275         for (int i = 0 ; i < arity ; i++) {
276             if (isExistential(getVertexTermFromAtom(_head.getID(),i))) {
277                 result[cpt] = i;
278                 cpt++;
279             }
280         }
281         return result;
282     }
283
284     /**
285      * Fullfills the rule from its string representation.
286      * @param str The string representation of the rule.
287      * @throws UnrecognizedStringFormatException If the string cannot be converted.
288      */
289     @Override
290     protected void fromString(String str) throws UnrecognizedStringFormatException {
291         String[] sub1 = str.split(HEAD_SEPARATOR);
292         if (sub1.length != 2)
293             throw new UnrecognizedStringFormatException();
294         sub1[0]+=ATOM_SEPARATOR;
295         sub1[0]+=sub1[1];
296         super.fromString(sub1[0]);
297         _head = getVertex(getNbAtoms()-1);
298     }
299
300     /**
301      * Converts the rule into a String.
302      * @return The string representation of the rule.
303      */
304     @Override
305     public String toString() {
306         return super.toStringExcluding(_head.getID())+HEAD_SEPARATOR+getHeadAtom();
307     }
308
309     /**
310      * Creates a new atom conjunction corresponding to the body of the rule.
311      * @return The atom conjunction of the body.
312      */
313     public AtomConjunction getBody() {
314         AtomConjunction body = new AtomConjunction();
315         super.cloneIn(body);
316         body.removeCleanlyAtom(_head.getID());
317         return body;
318     }
319
320     /**
321      * Clone the rule into a copy.
322      * @param copy The instance where to copied the rule.
323      * @return The copy of the rule.
324      */
325     public AtomicRule cloneIn(AtomicRule copy) {
326         super.cloneIn(copy);
327         copy._head = copy.getVertex(_head.getID());
328         return copy;
329     }
330
331     /**
332      * Clone the rule.
333      * @return A clone of the rule.
334      */
```

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

```
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 atomID from H1.
419      */
420     protected static AtomConjunction extension(AtomConjunction H1, int atomRootID, boolean isLocallyUnifiable[], int E[]) throws ExtensionFailureException {
421         AtomConjunction result = new AtomConjunction();
422         boolean isColored[] = new boolean[H1.getNbAtoms()+H1.getNbTerms()];
423         Fifo<Vertex<Object> > waiting = new Fifo<Vertex<Object> >();
424         Vertex<Object> current = H1.getVertex(atomRootID);
425         Vertex<Object> neighbour = null;
426         for (int i = 0 ; i < H1.getNbAtoms() ; i++) {
427             if (isLocallyUnifiable[i]) {
428                 isColored[i] = false;
429                 for (NeighbourIterator neighbourIterator = H1.vertexTermIteratorFromAtom(i) ; neighbourIterator.hasNext() ; )
430                     isColored[neighbourIterator.next().getID()] = false;
431             }
432         }
433         isColored[atomRootID] = true;
434 
435         result.addAtom(current,H1);
436         waiting.put(current);
437 
438         while (!waiting.isEmpty()) {
439             current = waiting.pop();
440             if (H1.isAtom(current)) {
441                 for (int i = 0 ; i < E.length ; i++) {
442                     neighbour = H1.getVertexTermFromAtom(current.getID(),E[i]);
443                     if (((Term)(neighbour.getValue())).isConstant())
444                         throw new ExtensionFailureException();
445                     if (!isColored[neighbour.getID()]) {
446                         isColored[neighbour.getID()] = true;
447                         waiting.put(neighbour);
448                     }
449                 }
450             }
451             else {      // current is a term
452                 for (NeighbourIterator neighbourIterator = H1.vertexAtomIteratorFromTerm(current.getID()) ; neighbourIterator.hasNext() ;) {
453                     neighbour = neighbourIterator.next();
454                     if (!isLocallyUnifiable[neighbour.getID()])
455                         throw new ExtensionFailureException();
456                     if (!isColored[neighbour.getID()]) {
457                         isColored[neighbour.getID()] = true;
458                         waiting.put(neighbour);
459                         result.addAtom(neighbour,H1);
460                     }
461                 }
462             }
463         }
464         return result;
465     }
466 
467     /**
468      * Checks if there is a unification between the head of the rule and the full atom conjunction passed as parameters.
469      * @param H1 The atom conjunction to be unified.
470      * @param R The rule whose head is to be unified.
471      * @param existentialIndex The position of all existential variables of R. This parameter is here for optimization only.
472      * @return True if there exists a unification, false otherwise.
473      */
474     protected static boolean localUnification(AtomConjunction H1, AtomicRule R, int[] existentialIndex) {
475 
476             /** predicate check */
477         for (Iterator<Vertex<Object> > iterator = H1._graph.firstVertexIterator() ; iterator.hasNext() ;)
478             if ((((Predicate)(iterator.next().getValue())).compareTo((Predicate)(R.getHead().getValue()))) != 0)
479                 return false;
480 
481 
482             /** graph generation */
483         AtomConjunction unification = H1;
484         unification._graph.addInFirstSet(((Predicate)(R.getHead().getValue())).clone());
485         int termID = -1;
486         int firstHeadTermID = -1;
487         for (Iterator<NeighbourEdge<Integer> > iterator = R._graph.neighbourIterator(R.getHead().getID()) ; iterator.hasNext() ;) {
488         NeighbourEdge<Integer> edge = iterator.next();
489         Term t = (Term)(R.get(edge.getIDV()));
490         if (t.isConstant())
491             termID = unification.addTerm(t);
```

```
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 
513             /** index */
514         int headIndex = 0;
515 
516         Vertex<Object> headVertex = null;
517         Vertex<Object> bodyVertex = null;
518         Term headTerm = null;
519         Term bodyTerm = null;
520         int arity = ((Predicate)(R.getHead().getValue())).getArity();
521 
522         boolean[] isexistential = new boolean[unification.getNbTerms()];
523         for (int i = 0 ; i < unification.getNbTerms() ; i++)
524             isexistential[i] = false;
525         for (int i = 0 ; i < existentialIndex.length ; i++)
526             isexistential[unification.getVertexTermFromAtom(unification.getNbAtoms()-1,↩
                    existentialIndex[i]).getID()-unification.getNbAtoms()] = true;
527 
528 
529             /** algorithm */
530         while (headIndex < arity) {
531             headVertex = unification.getVertexTermFromAtom(unification.getNbAtoms()-1,↩
                    headIndex);
532             headTerm = (Term)(headVertex.getValue());
533             for (int i = 0 ; i < unification.getNbAtoms() - 1 ; i++) {
534                 bodyVertex = unification.getVertexTermFromAtom(i,headIndex);
535                 bodyTerm = (Term)(bodyVertex.getValue());
536                 if (headVertex.getID() != bodyVertex.getID()) {
537                     if ((headTerm.isConstant() || isexistential[headVertex.getID()-↩
                            unification.getNbAtoms()])
538                         &&  (bodyTerm.isConstant() || isexistential[bodyVertex.getID()-↩
                            unification.getNbAtoms()]))
539                         return false;
540                     else if ((bodyTerm.isVariable()) && (!isexistential[bodyVertex.getID()-↩
                            unification.getNbAtoms()])) {
541                         try {
542                             unification._graph.contract(bodyVertex.getID(),headVertex.getID()↩
                                    );
543                         }
544                         catch (Exception e) {
545                             //e.printStackTrace();
546                             // TODO ConcurrentModificationException ???
547                         }
548                     }
549                     else {
550                         try {
551                             unification._graph.contract(headVertex.getID(),bodyVertex.getID()↩
                                    );
552                             headVertex = bodyVertex;
553                             headTerm = bodyTerm;
554                         }
555                         catch (Exception e) {
556                             //e.printStackTrace();
557                             // TODO ConcurrentModificationException ???
558                         }
559                     }
560                 }
561             }
562             headIndex++;
563         }
564         return true;
565 
566     }
567 
```

```
568
569
570 };
```

```java
package obr;

import java.util.Scanner;
import java.util.regex.Pattern;
import java.io.File;

/**
 * This class provides static functions to parse a single rule from its string
 * representation, or a full set of rules from a file.
 */
public class DTGParser {

    public static final String HEAD_SEPARATOR = " :- ";
    public static final String BEGIN_TERM_LIST = "\\(";     // TODO
    public static final String END_TERM_LIST = "\\)";
    public static final String TERM_SEPARATOR = ", ";
    /** Not marker constant. */
    public static final char NOT_MARK = '!';
    /** Variable marker. */
    public static final char VARIABLE_MARK = '?';
    /** End of line marker. */
    public static final char END_LINE = '.';
    /**
     * Absurd predicate used to convert <code>body —&gt; !head</code> to
     * <code>body;head—&gt;ABSURD</code>.
     */
    public static final Predicate ABSURD = new Predicate("ABSURD",0);

    /**
     * Parses a rule under a DTG format.
     * @param str The rule string representation.
     * @throws UnrecognizedStringFormatException If the string format is not ok.
     */
    public static AtomicRule parseRule(String str) throws UnrecognizedStringFormatException {
        String[] subs = str.split(HEAD_SEPARATOR);
        if (subs.length != 2)
            throw new UnrecognizedStringFormatException();
        if (subs[1].charAt(subs[1].length()-1) != END_LINE)
            throw new UnrecognizedStringFormatException();
        AtomicRule rule = new AtomicRule();

        String[] terms;
        String[] atomSubs;
        String term;
        int termID, atomID;
        boolean not = false;

        if (subs[0].charAt(0) == NOT_MARK) {
            subs[0] = subs[0].substring(1,subs[0].length());
            not = true;
        }

        /* BODY */
        subs[1] = subs[1].substring(0,subs[1].length()-1);  // last char is a dot
        atomSubs = subs[1].split(BEGIN_TERM_LIST);
        if (atomSubs.length != 2)
            throw new UnrecognizedStringFormatException();
        atomSubs[1] = atomSubs[1].substring(0,atomSubs[1].length()-1);
        terms = atomSubs[1].split(TERM_SEPARATOR);
        atomID = rule.addAtom(new Predicate(atomSubs[0],terms.length));
        for (int i = 0 ; i < terms.length ; i++) {
            if (terms[i].charAt(0) == VARIABLE_MARK)
                term = terms[i].substring(1,terms[i].length());
            else
                term = "'" + terms[i] + "'";
            termID = rule.addTerm(term);
            rule.addEdge(atomID,termID,i);
        }

        /* HEAD */
        atomSubs = subs[0].split(BEGIN_TERM_LIST);
        if (atomSubs.length != 2)
            throw new UnrecognizedStringFormatException();
        atomSubs[1] = atomSubs[1].substring(0,atomSubs[1].length()-1);
        terms = atomSubs[1].split(TERM_SEPARATOR);
        atomID = rule.addAtom(new Predicate(atomSubs[0],terms.length));
        for (int i = 0 ; i < terms.length ; i++) {
            if (terms[i].charAt(0) == VARIABLE_MARK)
                term = terms[i].substring(1,terms[i].length());
            else
                term = "'" + terms[i] + "'";
            termID = rule.addTerm(term);
            rule.addEdge(atomID,termID,i);
        }
        rule.setHead(atomID);

```

```
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 occured.
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                     try {
112                         grd.addVertex(parseRule(line));
113                     }
114                     catch (UnrecognizedStringFormatException exception) { }
115                 }
116                 line = scan.next();
117             }
118             scan.close();
119             return grd;
120         }
121         catch (Exception e) {
122             System.out.println(e);
123             e.printStackTrace();
124             return null;
125         }
126     }
127
128     /** This class cannot be instantiated. */
129     protected DTGParser() { }
130
131 };
```

```java
package obr;

/**
 * Interface for functions checking if a graph of rule dependencies or a subset of this graph belongs to a concrete
 * decidable class. <br />
 * It provides two methods, the first one is used to check the full set of rules, and the second one to check only
 * a strongly connected component of this set.
 * The classes which implement this interface may be used as a check function in the GRD Analyser.
 * @see GRDAnalyser
 */
public interface DecidableClassCheck {

    /**
     * Checks a graph of rule dependencies.
     * @return The associated decidable class label if the grd belongs to this decidable class,
     * null otherwise.
     */
    DecidableClassLabel grdCheck(GraphRuleDependencies grd);

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return The associated decidable class label if the strongly connected component belongs to this decidable class,
     * null otherwise.
     */
    DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID);

};
```

```java
package obr;

/**
 * Label for a decidable class check.
 * @see GRDAnalyser
 */
public class DecidableClassLabel {

    static final int FES = 1;
    static final int GBTS = 2;
    static final int FUS = 3;
    static final String FES_STR = "fes";
    static final String GBTS_STR = "gbts";
    static final String FUS_STR = "fus";
    static final String FES_STRING = "Finite Expansion Set";
    static final String GBTS_STRING = "Greedy Bounded Treewidth Set";
    static final String FUS_STRING = "Finite Unification Set";

    static final String shortName(int abstractClassID) {
        switch (abstractClassID) {
            case FUS :
                return FUS_STR;
            case FES :
                return FES_STR;
            case GBTS :
                return GBTS_STR;
            default :
                return "";
        }
    }

    static final String longName(int abstractClassID) {
        switch (abstractClassID) {
            case FUS :
                return FUS_STRING;
            case FES :
                return FES_STRING;
            case GBTS :
                return GBTS_STRING;
            default :
                return "";
        }
    }

    /**
     * Constructor which let all boolean attributes to false.
     * @param label The label of the decidable class.
     */
    public DecidableClassLabel(String label) {
        _label = label;
    }

    /**
     * Complete constructor.
     * @param label The label of the decidable class.
     * @param fus True if the concrete class belongs to fus abstract class.
     * @param fes True if the concrete class belongs to fes abstract class.
     * @param gbts True if the concrete class belongs to gbts abstract class.
     */
    public DecidableClassLabel(String label, boolean fus, boolean fes, boolean gbts) {
        _label = label;
        _fus = fus;
        _fes = fes;
        _gbts = gbts;
    }

    /**
     * Label getter.
     * @return The label of the concrete class.
     */
    public String getLabel() {
        return _label;
    }

    public boolean isFUS() {
        return _fus;
    }

    public boolean isFES() {
        return _fes;
    }

    public boolean isGBTS() {
        return _gbts;
    }

```

```
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 };
```

```java
package obr;

import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.IllegalEdgeException;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Checks if a set of rules satisfies the disconnected property.<br />
 * I.e., if all rules have a frontier of size 0.
 */
public class DisconnectedCheck implements DecidableClassCheck {

    public static final DecidableClassLabel LABEL = new DecidableClassLabel("disconnected",↩
        true,true,true);

    /**
     * Checks a graph of rule dependencies.
     * @return The decidable class label if the grd belongs to this decidable class, null ↩
       otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return The decidable class label if the strongly connected component belongs to this ↩
       decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    /**
     * Internal check.
     * @param rules The set of rules to be checked.
     */
    protected boolean check(Iterable<Vertex<AtomicRule> > rules) {
        for (Vertex<AtomicRule> vrule : rules) {
            if (vrule.getValue().frontier().size() != 0)
                return false;
        }
        return true;
    }

};
```

```java
package obr;

import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.IllegalEdgeException;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Checks if a set of rules satisfies the domain-restricted property.
 */
public class DomainRestrictedCheck implements DecidableClassCheck {

    /** Associated label. */
    public static final DecidableClassLabel LABEL = new DecidableClassLabel("domain-restricted",true,false,false);

    /**
     * Checks a graph of rule dependencies.
     * @return the decidable class label if the grd belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return the decidable class label if the strongly connected component belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    protected boolean check(Iterable<Vertex<AtomicRule> > rules) {
        AtomicRule rule = null;
        int frontierSize = 0;
        for (Vertex<AtomicRule> vrule : rules) {
            rule = vrule.getValue();
            frontierSize = rule.frontier().size();
            if ((frontierSize > 0) && (frontierSize < rule.getNbUniversalVariables()))
                return false;
        }
        return true;
    }

};
```

```
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 { };
```

```java
package obr;

import java.util.ArrayList;
import java.util.Iterator;

import moca.graphs.vertices.Vertex;

public class FrontierGuardedCheck implements DecidableClassCheck{

    public static final DecidableClassLabel LABEL = new DecidableClassLabel("frontier-guarded",false,false,true);

    /**
     * Checks a graph of rule dependencies.
     * @return the decidable class label if the grd belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param scc The strongly connected component to be checked.
     * @return the decidable class label if the strongly connected component belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
        for (Vertex<AtomicRule> vrule : rules) {
            if (!ruleCheck(vrule.getValue()))
                return false;
        }
        return true;
    }

    protected boolean ruleCheck(AtomicRule rule) {
        ArrayList<Vertex<Object>> frontier = rule.frontier();
        boolean guarded;
        int i = 0;
        for (Atom atom : rule) {
            if (!rule.isHead(i)) {
            guarded = true;
            for (Vertex<Object> vertexTerm : frontier) {
                Term t = (Term)(vertexTerm.getValue());
                if (!atom.contains(t)) {
                    guarded = false;
                    break;
                }
            }
            if (guarded)
                return true;
            }
            i++;
        }
        return false;
    }

};
```

```java
package obr;

import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.IllegalEdgeException;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Checks if a set of rules satisfies the frontier-1 property.<br />
 * I.e., if all rules have a frontier of size 1.
 */
public class FrontierOneCheck implements DecidableClassCheck {

    /** Associated label. */
    public static final DecidableClassLabel LABEL = new DecidableClassLabel("frontier-1",
        false,false,true);

    /**
     * Checks a graph of rule dependencies.
     * @return the decidable class label if the grd belongs to this decidable class, null
        otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return the decidable class label if the strongly connected component belongs to this
        decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
        for (Vertex<AtomicRule> vrule : rules) {
            if (vrule.getValue().frontier().size() != 1){
                return false;
            }
        }
        return true;
    }

};
```

```
package obr;

import moca.graphs.vertices.Vertex;
import moca.graphs.vertices.VertexBinaryFunction;
import moca.graphs.Graph;
import java.util.ArrayList;

/**
 * Analyse a graph of rule dependencies to determine if it belongs to decidable class.<br />
 * Furthermore, the GRDAnalyser will check the strongly connected components of the GRD.
 * Then, it will check if a query will be answered in a finite time.
 */
public class GRDAnalyser {

    public abstract class ClassChecker {
        public void unprocess() { _processed = false; }
        public void process() { _processed = true; }
        public abstract String diagnostic();
        public boolean isProcessed() { return _processed; }
        protected boolean _processed = false;
    }

    protected class DecidableClassChecker extends ClassChecker {

        public static final String UNPROCESSED_MSG = "Decidable class checks unprocessed!";

        /**
         * Adds a new decidable class check function into the list.
         * If the function already exists, it will not be added.
         * @param checkFunction The new function to add.
         */
        public void addDecidableClassCheck(DecidableClassCheck checkFunction) {
            if (!_checkFunctions.contains(checkFunction))
                _checkFunctions.add(checkFunction);
        }
        public boolean isFES(int sccID) {
            if (!_processed)
                return false;
            ArrayList<DecidableClassLabel> labels = _sccLabels.get(sccID);
            for (DecidableClassLabel label : labels)
                if (label.isFES())
                    return true;
            return false;
        }
        public boolean isGBTS(int sccID) {
            if (!_processed)
                return false;
            ArrayList<DecidableClassLabel> labels = _sccLabels.get(sccID);
            for (DecidableClassLabel label : labels)
                if (label.isGBTS())
                    return true;
            return false;
        }
        public boolean isFUS(int sccID) {
            if (!_processed)
                return false;
            ArrayList<DecidableClassLabel> labels = _sccLabels.get(sccID);
            for (DecidableClassLabel label : labels)
                if (label.isFUS())
                    return true;
            return false;
        }
        public boolean isFES() {
            if (_grdLabels.size() > 0) {
                for (int i = 0 ; (i < _grdLabels.size()) ; i++) {
                    if (_grdLabels.get(i).isFES())
                        return true;
                }
            }
            return false;
        }
        public boolean isGBTS() {
            if (_grdLabels.size() > 0) {
                for (int i = 0 ; (i < _grdLabels.size()) ; i++) {
                    if (_grdLabels.get(i).isGBTS())
                        return true;
                }
            }
            return false;
        }
        public boolean isFUS() {
            if (_grdLabels.size() > 0) {
                for (int i = 0 ; (i < _grdLabels.size()) ; i++) {
                    if (_grdLabels.get(i).isFUS())
                        return true;
                }
```

```
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 ;</li>
96           * <li>and checks all functions onto each strongly connected component of the GRD.</li>
97           * </p>
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                 // starts checking
108             if (_grd.isCyclic()) {
109                 DecidableClassLabel l = null;
110                 for (DecidableClassCheck function : _checkFunctions) {
111                     try {
112                         l = function.grdCheck(_grd);
113                         if (l != null)
114                             _grdLabels.add(l);
115                     }
116                     catch (UnsupportedOperationException e) { /* this check function cannot be used against the full grd */ }
117                 }
118                 for (int i = 0 ; i < _grd.getNbComponents() ; i++) {
119                     _sccLabels.add(new ArrayList<DecidableClassLabel>());
120                     for (DecidableClassCheck function : _checkFunctions) {
121                         try {
122                             l = function.sccCheck(_grd,i);
123                             if (l != null)
124                                 _sccLabels.get(i).add(l);
125                         }
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 ↩
                component of the GRD. */
182         private ArrayList<ArrayList<DecidableClassLabel> > _sccLabels;
183 
184         /** The check functions to be used. */
185         private ArrayList<DecidableClassCheck> _checkFunctions = new ArrayList<↩
                DecidableClassCheck>();
186 
187     };
188 
189     protected class AbstractClassChecker extends ClassChecker
190                 implements VertexBinaryFunction<ArrayList<Vertex<AtomicRule> > > {
191 
192         public static final String UNPROCESSED_MSG = "Abstract class checks unprocessed!";
193 
194         @Override
195         public void unprocess() {
196             _grdAbstractClass = 0;
197             _sccAbstractClass = null;
198             _processed = false;
199         }
200 
201         @Override
202         public void process() {
203             if (_processed)
204                 unprocess();
205             _decidable = true;
206             if (isFES())
207                 _grdAbstractClass = DecidableClassLabel.FES;
208             else if (isGBTS())
209                 _grdAbstractClass = DecidableClassLabel.GBTS;
210             else if (isFUS())
211                 _grdAbstractClass = DecidableClassLabel.FUS;
212             else
213                 _decidable = false;
214             if (!_decidable) {
215                 _sccAbstractClass = new int[_grd.getNbComponents()];
216                 processCombine();
217             }
218             _processed = true;
219         }
220 
221         protected void processCombine() {
222             ArrayList<Integer> sourceID = new ArrayList<Integer>();
223             final int nbComponents = _grd.getNbComponents();
224             int i,j;
225             boolean ended;
226             for (i = 0 ; i < nbComponents ; i++) {
227                 ended = false;
228                 for (j = 0 ; (j < nbComponents) && !ended ; j++) {
229                     if ((i != j) && (_grd.getStronglyConnectedComponentsGraph().isEdge(j,i)))
230                         ended = true;
231                 }
232                 if (j == nbComponents)
233                     sourceID.add(new Integer(i));
234             }
235 
236             _decidable = true;
237             Graph.WalkIterator iterator;
238             for (i = 0 ; (i < sourceID.size()) && _decidable ; i++)
239                 setMin(sourceID.get(i),0);
240             iterator = _grd.getStronglyConnectedComponentsGraph().BFSIterator(sourceID,this,↩
                null);
241             while ((iterator.hasNext()) && _decidable)
242                 iterator.next();
243         }
244 
245         @Override
246         public String diagnostic() {
247             if (!_processed)
248                 return UNPROCESSED_MSG;
249 
250             if (!_decidable)
251                 return "This set of rule cannot be used to make a query.";
252 
```

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

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

```java
package obr;

import moca.graphs.DirectedSimpleGraph;
import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.IllegalEdgeException;
import moca.graphs.edges.NeighbourEdge;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.HashMap;
import java.util.NoSuchElementException;


/**
 * Represents the graph of position dependencies corresponding to a GRD or to one of its strongly connected components.<br />
 * The graph maintains a hash map to find quickly the vertex corresponding to a predicate and its position.
 * It is used to check the weakly-acyclic and the weakly-sticky properties.
 */
public class GraphPositionDependencies extends DirectedSimpleGraph<Predicate,Boolean> {

    /**
     * Default constructor.
     */
    public GraphPositionDependencies() throws IllegalConstructionException {
        super();
    }

    /**
     * Constructor from a set of rules.
     * @param rules The set of rule to convert.
     */
    public GraphPositionDependencies(Iterable<Vertex<AtomicRule>> rules) throws IllegalConstructionException {
        super();
        init(rules);
    }

    /**
     * Adds a predicate into the graph if not already in.
     * Note that the number of created vertices is equals to the predicate arity.
     * @param p The predicate to be added.
     */
    public void addPredicate(Predicate p) {
        if (!_predicateIndex.containsKey(p)) {
            _predicateIndex.put(p,getNbVertices());
            for (int i = 0 ; i < p.getArity() ; i++)
                addVertex(p);
        }
    }

    /**
     * Convenient method to add a non special edge between two predicate positions.
     * (from the first to the second)
     * @param p The first predicate.
     * @param positionP The first position (it must be lesser than p arity).
     * @param q The second predicate.
     * @param positionQ The second position (it mus be lesser than q arity).
     */
public void addEdge(Predicate p, int positionP, Predicate q, int positionQ) {
        addEdge(p,positionP,q,positionQ,false);
    }

    /**
     * Convenient method to add a non special edge between two predicate positions.
     * (from the first to the second)
     * @param p The first predicate.
     * @param positionP The first position (it must be lesser than p arity).
     * @param q The second predicate.
     * @param positionQ The second position (it mus be lesser than q arity).
     * @param special The edge value, if true it will be a special edge.
     */
    public void addEdge(Predicate p, int positionP, Predicate q, int positionQ, Boolean special) {
        try {
            addEdge(getVertex(p,positionP),getVertex(q,positionQ),special);
        }
        catch (IllegalEdgeException e) {
            // if an edge already exists its value should be checked against the special parameter.
        }
    }

    /**
```

```
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)
```

```
                            _finiteRanks[i] = false;
                        }
                        catch (NoSuchElementException e) { }
                    }
                }
            }
        }

        /**
         * Converts a vertex to the predicate position string.
         * @param vertexID The id of the vertex to be converted.
         * @return A string of form "p[i]"
         */
        public String predicatePositionToString(int vertexID) {
            Predicate p = get(vertexID);
            int position = 1;
            boolean end = false;
            while ((vertexID - position >= 0) && (!end)) {
                if (p.compareTo(get(vertexID-position)) == 0)
                    position++;
                else
                    end = true;
            }
            position--;
            return p.getLabel() + "[" + position + "]";
        }

        /**
         * Converts the full graph into a string.<br />
         * It will first contain the vertices values, and then the edges.
         */
        @Override
        public String toString() {
            StringBuilder result = new StringBuilder();
            NeighbourEdge<Boolean> edge = null;

            // vertices
            for (int i = 0 ; i < getNbVertices() ; i++) {
                result.append(i);
                result.append(" : ");
                result.append(predicatePositionToString(i));
                result.append('\n');
            }
            result.append('\n');

            /*while (predicateIndex < getNbVertices()) {
                positionIndex = 0;
                while (positionIndex < get(predicateIndex).getArity()) {
                    result.append(predicateIndex + positionIndex);
                    result.append(" : ");
                    result.append(get(predicateIndex).getLabel());
                    result.append("[");
                    result.append(positionIndex);
                    result.append("]");
                    result.append('\n');
                    positionIndex++;
                }
                predicateIndex += positionIndex;
            }*/

            // edges
            for (int i = 0 ; i < getNbVertices() ; i++) {
                for (Iterator<NeighbourEdge<Boolean> > iterator = neighbourIterator(i) ; iterator.hasNext() ;) {
                    edge = iterator.next();
                    result.append(predicatePositionToString(i));
                    if (edge.getValue() == true)
                        result.append("\tx->\t");
                    else
                        result.append("\t-->\t");
                    result.append(predicatePositionToString(edge.getIDV()));
                    result.append('\n');
                }
            }
            return result.toString();
        }

        /**
         * Fullfills the graph from the set of rules.
         * @param rules The set of rule.
         */
        protected void init(Iterable<Vertex<AtomicRule> > rules) {

            // vertices
            for (Vertex<AtomicRule> rule : rules) {
                for (int i = 0 ; i < rule.getValue().getNbAtoms() ; i++)
                    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↩
                    (headVertex.getID()) ;
271                 neighbourIterator.hasNext() ;) {
272                 edge = neighbourIterator.next();
273                 neighbour = rule.getTerm(edge.getIDV()-rule.getNbAtoms());
274                 if (rule.isUniversal(rule.getVertex(edge.getIDV()))) {
275                     for (Iterator<NeighbourEdge<Integer> > neighbourIterator2 = rule.↩
                            neighbourIterator(edge.getIDV()) ;
276                         neighbourIterator2.hasNext() ;) {
277                         edge2 = neighbourIterator2.next();
278                         vertex = rule.getVertex(edge2.getIDV());
279                         if (rule.isBody(vertex)) {
280                             p = (Predicate)(vertex.getValue());
281                             positionP = edge2.getValue();
282                             ri = getVertex(p,positionP);
283                             for (Iterator<NeighbourEdge<Integer> > neighbourIterator3 = rule.↩
                                    neighbourIterator(edge.getIDV()) ; neighbourIterator3.hasNext↩
                                    () ; ) {
284                                 edge3 = neighbourIterator3.next();
285                                 if (rule.isHead(rule.getVertex(edge3.getIDV()))) {
286                                     q = rule.getPredicate(edge3.getIDV());
287                                     positionQ = edge3.getValue();
288                                     addEdge(p,positionP,q,positionQ);
289                                 }
290                             }
291                             for (Iterator<NeighbourEdge<Integer> > neighbourIterator3 = rule.↩
                                    neighbourIterator(headVertex.getID()) ; neighbourIterator3.↩
                                    hasNext() ;) {
292                                 edge3 = neighbourIterator3.next();
293                                 if (rule.isExistential(rule.getVertex(edge3.getIDV()))) {
294                                     q = (Predicate)(headVertex.getValue());
295                                     positionQ = edge3.getValue();
296                                     sj = getVertex(q,positionQ);
297                                     if (isEdge(ri.getID(),sj.getID()))
298                                         removeEdge(ri.getID(),sj.getID());
299                                     addEdge(p,positionP,q,positionQ,true);
300                                 }
301                             }
302                         }
303                     }
304                 }
305             }
306         }
307     }
308
309
310     /** Internal hash map used to access to the first vertex corresponding to the predicate. ↩
            */
311     private HashMap<Predicate,Integer> _predicateIndex = new HashMap<Predicate,Integer>();
312     /** Array of same length as the number of components, and valuing true if the scc has a ↩
            finite rank. */
313     private boolean _finiteRanks[] = null;
314
315 };
```

```java
package obr;

import moca.graphs.DirectedSimpleGraph;
import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.IllegalEdgeException;
import moca.graphs.edges.Edge;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.visu.AcyclicGraphLocalizer;
import moca.graphs.visu.PolygonLocalizer;
import moca.graphs.visu.PostScriptConverter;
import moca.graphs.visu.SCCIdentityFunction;

import java.util.ArrayList;
import java.util.Scanner;
import java.util.Iterator;
import java.util.regex.Pattern;
import java.io.File;
//import java.io.OutputStreamWriter;
//import java.io.BufferedOutputStream;
//import java.io.FileOutputStream;


/**
 * Unsecure :
 *  if a vertex already in the graph is added once again, it may produce strange behaviour
 *  if an edge is added manually it may produce errors
 */
public class GraphRuleDependencies extends DirectedSimpleGraph<AtomicRule,Boolean> {

    public GraphRuleDependencies() throws IllegalConstructionException {
        super();
    }

    public GraphRuleDependencies(String filePath) throws IllegalConstructionException {
        super();
        fromFile(filePath);
    }

    public GraphRuleDependencies(GraphRuleDependencies g) throws IllegalConstructionException {
        super(g);
    }

    /**
     * Whenever a vertex is added, the rule is checked for unification against all others.
     * This provides an always updated graph.
     */
    public void addVertex(AtomicRule rule) {
        super.addVertex(rule);
        Vertex<AtomicRule> r = getVertex(getNbVertices()-1);
        Vertex<AtomicRule> r2 = null;
        for (Iterator<Vertex<AtomicRule>> ruleIterator = vertexIterator() ; ruleIterator.hasNext() ; ) {
            r2 = ruleIterator.next();
            try {
                if (r.getValue().mayImply(r2.getValue()))
                    addEdge(r.getID(),r2.getID(),true);
            }
            catch (IllegalEdgeException e) {
                // the edge already exists
            }
            try {
                if ((r2 != r) && (r2.getValue().mayImply(r.getValue())))
                    addEdge(r2.getID(),r.getID(),true);
            }
            catch (IllegalEdgeException e) {
                // the edge already exists
            }
        }
    }

    public void fromFile(String filePath) {
        try {
            String line = null;
            Scanner scan = new Scanner(new File(filePath));
            scan.useDelimiter(Pattern.compile("\n"));
            while (scan.hasNext())
                addVertex(new AtomicRule(scan.next()));
            scan.close();
        }
        catch (Exception e) {
            System.out.println(e);
            e.printStackTrace();
        }
    }
```

```
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 = ↩
                getStronglyConnectedComponentsGraph();
122         StringBuilder stringBuilder = new StringBuilder();
123         for (int i = 0 ; i < sccg.getNbVertices() ; i++) {
124             stringBuilder.append('C');
125             stringBuilder.append(i);
126             stringBuilder.append(":\t");
127             for (int j = 0 ; j < sccg.getVertex(i).getValue().size() ; j++) {
128                 stringBuilder.append(sccg.getVertex(i).getValue().get(j).getID());
129                 if (j != sccg.getVertex(i).getValue().size()-1)
130                     stringBuilder.append(',');
131             }
132             stringBuilder.append('\n');
133         }
134         boolean newelement = false;
135         for (int i = 0 ; i < sccg.getNbVertices() ; i++) {
136             newelement = false;
137             for (Iterator<NeighbourEdge<Boolean>> neighbour = sccg.neighbourIterator(i) ; ↩
                    neighbour.hasNext() ; ) {
138                 if (!newelement) {
139                     stringBuilder.append("C");
140                     stringBuilder.append(i);
141                     stringBuilder.append("\t-->\t");
142                 }
143                 stringBuilder.append("C");
144                 stringBuilder.append(neighbour.next().getIDV());
145                 stringBuilder.append(',');
146                 newelement = true;
147             }
148             if (newelement)
149                 stringBuilder.append('\n');
150         }
151         return stringBuilder.toString();
152     }
153
154     public void sccToPostScript(String filePath) {
155         PostScriptConverter psOutput = new PostScriptConverter(
156             getStronglyConnectedComponentsGraph(),
157             480,640,
158             new AcyclicGraphLocalizer(480,640),
159             SCCIdentityFunction.instance(),
160             "Strongly Connected Components");
161         psOutput.writeToFile(filePath);
162     }
163
164     public void toPostScript(String filePath) {
165         PostScriptConverter psOutput = new PostScriptConverter(
166             this,
167             480,640,
```

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

```java
package obr;

import java.util.ArrayList;
import java.util.Iterator;

import moca.graphs.vertices.Vertex;

public class GuardedCheck implements DecidableClassCheck{

    public static final DecidableClassLabel LABEL = new DecidableClassLabel("guarded",false,false,true);

    /**
     * Checks a graph of rule dependencies.
     * @return the decidable class label if the grd belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param scc The strongly connected component to be checked.
     * @return the decidable class label if the strongly connected component belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    protected boolean check(Iterable<Vertex<AtomicRule>> rules) {
        //System.out.println("--------------GUARDED?----------------");
        AtomicRule rule;
        int i,nbAtoms,nbVars;
        boolean guarded;
        for (Vertex<AtomicRule> vrule : rules) {
            rule = vrule.getValue();
            nbAtoms = rule.getNbAtoms();
            nbVars = rule.getNbUniversalVariables();
            guarded = false;
            //System.out.println("rule = "+rule+"\nnbAtoms = "+nbAtoms+"\nnbVars = "+nbVars);
            for (i = 0 ; (i < nbAtoms) && !guarded ; i++) {
                //System.out.println("i = "+i+"\nisHead()? "+rule.isHead(i)+"\nvars = "+rule.variables(i).size());
                if ((!rule.isHead(i))
                &&  (nbVars == rule.variables(i).size()))
                    guarded = true;
            }
            if (!guarded)
                return false;
        }
        return true;
    }

}
```

```
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 { };
```

```java
package obr;

import java.lang.Comparable;

/**
 * Represents a predicate with its label and its arity.<br />
 */
public class Predicate implements Comparable<Predicate> {

    /**
     * Constructor.
     * @param label The predicate label.
     * @param arity The predicate arity.
     */
    public Predicate(String label, int arity) {
        _label = label;
        _arity = arity;
    }

    /**
     * Copy constructor.
     * @param p The predicate to be copied.
     */
    public Predicate(Predicate p) {
        _label = new String(p.getLabel());
        _arity = p.getArity();
    }

    /**
     * Constructor from a string representation.
     * @param str The string representation.
     * @throws UnrecognizedStringFormatException If the string format is not correct.
     */
    public Predicate(String str) throws UnrecognizedStringFormatException {
        fromString(str);
    }

    /**
     * Label getter.
     * @return The predicate label.
     */
    public String getLabel() {
        return _label;
    }

    /**
     * Arity getter.
     * @return The predicate arity.
     */
    public int getArity() {
        return _arity;
    }

    /**
     * Label setter.
     * @param value The new label to set.
     */
    public void setLabel(String value) {
        _label = value;
    }

    /**
     * Arity setter.
     * @param value The new arity to set.
     */
    public void setArity(int value) {
        _arity = value;
    }

    /**
     * Converts the predicate into a string.
     * @return A string representation of the predicate.
     */
    @Override
    public String toString() {
        return _label + '/' + _arity;
    }

    /**
     * Converts from a string.<br />
     * A well-formated string is under the form of : &lt;predicate label&gt;/&lt;arity&gt;.
     * @param str The string representation of a predicate.
     * @throws UnrecognizedStringFormatException If the string is not well-formated.
     */
    protected void fromString(String str) throws UnrecognizedStringFormatException {
        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 &lt; 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 };
```

```java
package obr;

import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.IllegalEdgeException;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Checks if a set of rules satisfies the range-restricted property.<br />
 * I.e., if there is no existential variables.
 */
public class RangeRestrictedCheck implements DecidableClassCheck {

    /** Associated label. */
    public static final DecidableClassLabel LABEL = new DecidableClassLabel("range-restricted",false,true,true);

    /**
     * Checks a graph of rule dependencies.
     * @return The decidable class label if the grd belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return The decidable class label if the strongly connected component belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    /**
     * Internal check.
     * @param rules The set of rule to check.
     * @return True if there is no existential variables, false otherwise.
     */
    protected boolean check(Iterable<Vertex<AtomicRule> > rules) {
        for (Vertex<AtomicRule> vrule : rules) {
            for (Iterator<Vertex<Object> > termIterator = vrule.getValue().vertexTermIterator() ; termIterator.hasNext() ;) {
                if (vrule.getValue().isExistential(termIterator.next()))
                    return false;
            }
        }
        return true;
    }

};
```

```
package obr;

import moca.graphs.vertices.Vertex;
import moca.graphs.vertices.VertexIdentityFunction;

public class RuleIdentityFunction extends VertexIdentityFunction<AtomicRule> {
    public String exec(Vertex<AtomicRule> v) {
        return "R"+v.getID();
    }
    public static RuleIdentityFunction instance() {
        if (_instance == null)
            _instance = new RuleIdentityFunction();
        return _instance;
    }
    private static RuleIdentityFunction _instance = null;
    protected RuleIdentityFunction() { }
};
```

```java
package obr;

import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.IllegalEdgeException;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Checks if a set of rules satisfies the stickiness property.
 */
public class StickyCheck implements DecidableClassCheck {

    /** Associated label. */
    public static final DecidableClassLabel LABEL = new DecidableClassLabel("sticky",true,false,false);

    /**
     * Checks a graph of rule dependencies.
     * @param grd The graph of rule dependencies to check.
     * @return The decidable class label if the grd belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return The decidable class label if the strongly connected component belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    /**
     * Internal check.
     * @param rules The set of rules to check.
     * @return True if the set of rules satisfies the stickiness property, false otherwise.
     */
    protected boolean check(Iterable<Vertex<AtomicRule> > rules) {
        _positions = new ArrayList<Integer>();
        Vertex<Object> vertexTerm = null;
        NeighbourEdge<Integer> edge = null;
        AtomicRule rule = null;

        // first walk
        for (Vertex<AtomicRule> vrule : rules) {
            rule = vrule.getValue();
            for (Iterator<Vertex<Object> > termIterator = rule.vertexTermIterator() ; termIterator.hasNext() ; ) {
                vertexTerm = termIterator.next();
                if ((((Term)vertexTerm.getValue()).isVariable()) && (!rule.isHead(vertexTerm))) {
                    try { mark(vertexTerm,rule); }
                    catch (MarkFailureException e) { return false; }
                }
            }
        }

        // second walk
        for (int i = 0 ; i < _positions.size() ; i++) {
            for (Vertex<AtomicRule> vrule : rules) {
                rule = vrule.getValue();
                try {
                    vertexTerm = rule.getVertexTermFromAtom(rule.getHead().getID(),_positions.get(i));
                    try { mark(vertexTerm,rule); }
                    catch (MarkFailureException e) { return false; }
                }
                catch (NoSuchElementException e) { /* nothing to be done */ }
            }
        }

        return true;
    }
```

```
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 occured more than once in the body↩
             of rule.
87       */
88      protected void mark(Vertex<Object> term, AtomicRule rule) throws MarkFailureException {
89          NeighbourEdge<Integer> edge = null;
90          int cpt = 0;
91          for (Iterator<NeighbourEdge<Integer> > iterator = rule.neighbourIterator(term.getID()↩
                ) ; iterator.hasNext() ; ) {
92              edge = iterator.next();
93              if (rule.isBody(edge.getIDV())) {
94                  cpt++;
95                  if (cpt >= 2)
96                      throw new MarkFailureException();
97                  if (!_positions.contains(edge.getValue()))
98                      _positions.add(edge.getValue());
99              }
100         }
101     }
102
103     /**
104      * Internal list of positions to be checked.
105      */
106     protected ArrayList<Integer> _positions;
107
108 };
```

```java
package obr;

import java.lang.String;

/**
 * A term may be either a variable or a constant.<br />
 * A constant is surround by simple quotes.
 * Note that two different types of constants exist : the first one is a classical constant added from a fact base.
 * The second one is a constant which have been generated by an existential rule application.
 * The "existential constants" start by a simple quote immediatly followed by an underscore.
 */
public class Term {

    /**
     * Constructor.
     * @param label The term label.
     */
    public Term(String label) {
        _label = label;
    }

    /**
     * Label getter.
     * @return The term label.
     */
    public String getLabel() {
        return _label;
    }

    /**
     * Label setter.
     * @param value The new label to set.
     */
    public void setLabel(String value) {
        _label = value;
    }

    /**
     * Allows to know if a term is a constant.
     * @return True if the term is a constant, false otherwise.
     */
    boolean isConstant() {
        return (_label.charAt(0) == '\'');
    }

    /**
     * Allows to know if a term is a variable.
     * @return True if the term is a variable, false otherwise.
     */
    boolean isVariable() {
        return !isConstant();
    }

    /**
     * Allows to know if a term is a constant which has been generated by an existential rule application.
     * @return True if the term is a generated constant, false otherwise.
     */
    boolean isExistential() {
        return (isConstant() && (_label.charAt(1)=='_'));
    }

    /**
     * Converts the term into a String.
     * @return The string representation of the term.
     */
    @Override
    public String toString() {
        return _label;
    }

    /**
     * Clones the term.
     * @return A clone.
     */
    @Override
    public Term clone() {
        return new Term(new String(_label));
    }

    /** Term label. */
    private String _label;

}
```

```
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 { };
```

```java
package obr;

import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.IllegalEdgeException;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Checks if a set of rules satisfies the weakly-acyclic property.
 * @see GraphPositionDependencies
 */
public class WeaklyAcyclicCheck implements DecidableClassCheck {

    /** Associated label. */
    public static final DecidableClassLabel LABEL = new DecidableClassLabel("weakly-acyclic",
        false,true,false);

    /**
     * Checks a graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @return The decidable class label if the grd belongs to this decidable class, null
       otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        try { _graphPosDep = new GraphPositionDependencies(grd.getVertexCollection()); }
        catch (IllegalConstructionException e) { return null; }
        if (_graphPosDep.finiteRank())
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return the decidable class label if the strongly connected component belongs to this
       decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        try { _graphPosDep = new GraphPositionDependencies(grd.getComponent(sccID)); }
        catch (IllegalConstructionException e) { return null; }
        if (_graphPosDep.finiteRank())
            return LABEL;
        return null;
    }

    /** The graph of position dependencies generated from the set of rules. */
    private GraphPositionDependencies _graphPosDep;

};
```

```java
package obr;

import moca.graphs.IllegalConstructionException;
import moca.graphs.vertices.Vertex;
import moca.graphs.edges.NeighbourEdge;
import moca.graphs.edges.IllegalEdgeException;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Checks if a set of rules satisfies the weakly-stickiness property.
 * @see WeaklyAcyclicCheck
 * @see StickyCheck
 * @see GraphPositionDependencies
 */
public class WeaklyStickyCheck extends StickyCheck implements DecidableClassCheck {

    /** Associated label. */
    public static final DecidableClassLabel LABEL = new DecidableClassLabel("weakly-sticky");

    /**
     * Checks a graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @return The decidable class label if the grd belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel grdCheck(GraphRuleDependencies grd) {
        try { _graphPosDep = new GraphPositionDependencies(grd.getVertexCollection()); }
        catch (IllegalConstructionException e) { }
        if (check(grd.getVertexCollection()))
            return LABEL;
        return null;
    }

    /**
     * Checks only a strongly connected component of the graph of rule dependencies.
     * @param grd The graph of rule dependencies.
     * @param sccID The id of the strongly connected component to be checked.
     * @return The decidable class label if the strongly connected component belongs to this decidable class, null otherwise.
     */
    public DecidableClassLabel sccCheck(GraphRuleDependencies grd, int sccID) {
        try { _graphPosDep = new GraphPositionDependencies(grd.getComponent(sccID)); }
        catch (IllegalConstructionException e) { }
        if (check(grd.getComponent(sccID)))
            return LABEL;
        return null;
    }

    /**
     * Marks a term.
     * @param term The vertex to mark.
     * @param rule The rule where to mark.
     * @throws MarkFailureException If a marked variable occurs more than once in the body at a position with infinite rank.
     * @see StickyCheck#mark(Vertex,AtomicRule)
     * @see GraphPositionDependencies#finiteRank(Predicate,int)
     */
    protected void mark(Vertex<Object> term, AtomicRule rule) throws MarkFailureException {
        NeighbourEdge<Integer> edge = null;
        int cpt = 0;
        for (Iterator<NeighbourEdge<Integer>> iterator = rule.neighbourIterator(term.getID()) ; iterator.hasNext() ; ) {
            edge = iterator.next();
            if (rule.isBody(edge.getIDV())) {
                cpt++;
                if ((cpt >= 2) && (!_graphPosDep.finiteRank(rule.getPredicate(edge.getIDV()),edge.getValue())))
                    throw new MarkFailureException();
                if (!_positions.contains(edge.getValue()))
                    _positions.add(edge.getValue());
            }
        }
    }

    /** Graph of position dependencies created from the set of rules. */
    private GraphPositionDependencies _graphPosDep;

};
```