Answer Set Programming et interrogation

Fabien Garreau, Laurent Garcia, Claire Lefèvre, Igor Stéphan

Université d'Angers, 2 boulevard Lavoisier, 49045 Angers, France
fabien.garreau\|laurent.garcia\|claire.lefevre\|igor.stephan@univ-angers.fr

RÉSUMÉ. *Les ontologies permettent de décrire des informations sur des concepts et les liens entre ceux-ci et de nombreux raisonneurs efficaces sont disponibles pour les interroger. Lorsque les informations à traiter sont de nature imparfaite ou sujettes à exception, les formalismes habituels ne sont plus adaptés et nous proposons ici d'utiliser l'Answer Set Programming (ASP) qui permet une meilleure expressivité des ontologies. Nous nous intéressons à la définition formelle de l'interrogation en ASP et nous montrons que les implémentations associées permettent d'obtenir des résultats intéressants à la fois sur les ontologies traditionnelles et sur celles admettant des exceptions.*

ABSTRACT. *Ontologies are used to describe information about concepts and links between them. Several efficient reasoners are available for query answering with ontologies. When information to be processed is imperfect or is subject to exception, common formalisms are not suitable anymore, that is why we propose the use of Answer Set Programming (ASP) that offers a better expressivity for ontologies. We take interest in the formal definition of query answering in ASP and we show that related implementations give interesting results both on traditional ontologies and with those containing exceptions.*

MOTS-CLÉS : *Answer Set Programming, interrogation, ontologie.*

KEYWORDS: *Answer Set Programming, query answering, ontology.*

DOI:10.3166/RIA.32.555-602

1. Introduction

Lorsqu'on souhaite interroger des connaissances issues du Web, il est courant d'utiliser des ontologies qui représentent les informations sous forme de réseaux sémantiques. Une ontologie regroupe un ensemble de concepts, permettant de représenter un domaine, liés entre eux par des relations taxonomiques et sémantiques. Les liens entre les concepts permettent de raisonner sur l'ontologie en inférant de nouvelles connaissances qui ne sont pas stockées de manière explicite dans la base de connaissances mais en intention. D'un point de vue pratique, c'est le langage OWL (*OWL 2 Web Ontology Language Document Overview (Second Edition)*, 2012), basé sur RDF (Prud'hommeaux, Seaborne, 2008), qui est recommandé par le W3C pour la représentation de ces ontologies. Ce langage a pour avantage d'être simple à interpréter

pour une machine grâce à son format balisé. Plusieurs versions d'OWL plus ou moins expressives et plus ou moins simples à mettre en œuvre sont définies. En pratique, c'est la version *OWL DL* qui est considérée comme la plus raisonnable et il existe des raisonneurs efficaces (Pellet (Sirin *et al.*, 2007), Hermit (Glimm *et al.*, 2014)) pour la traiter. Dans cet article, les exemples d'ontologies utilisés seront basés sur l'ontologie académique la plus courante nommée `university` (Guo *et al.*, 2005 ; Kollia *et al.*, 2011 ; Pérez-Urbina *et al.*, 2009). Dans celle-ci, on peut représenter des informations telles que « Un enseignant-chercheur est un personnel de l'université », « Si un personnel de l'université enseigne un cours alors il est enseignant » ou « Pierre est un enseignant-chercheur ».

Une ontologie est interrogée à l'aide d'une requête permettant d'extraire des informations qui répondent à certains critères. Nous pouvons, par exemple, dans l'ontologie `university`, poser les questions suivantes :

« Jean enseigne-t-il l'informatique ? »,
« Existe-t-il une matière enseignée par Jean ? » ou encore
« Quelles sont les matières enseignées par Jean ? ».

Pour les deux premières interrogations, nous attendons une réponse par oui ou non et pour la dernière l'ensemble des matières qui sont enseignées par Jean. Un des aspects importants de l'interrogation est qu'il n'est pas nécessaire de calculer l'ensemble de l'extension pour pouvoir répondre à une requête. Ceci est un atout pour l'interrogation d'ontologies car elles sont constituées d'un grand nombre de données qui ne possèdent que très peu de dépendances entre elles. Si nous interrogeons l'ontologie `university` en demandant si Jean enseigne un cours d'informatique, nous n'avons pas besoin de savoir que Pierre suit un cours de mathématique, et donc une partie de la connaissance n'est pas nécessaire pour répondre à la requête. Le principe de l'interrogation d'ontologie est alors d'utiliser un mécanisme de marche arrière partant du but en se servant des informations présentes dans la requête pour y répondre.

Dans le cadre d'un formalisme logique (on parle alors de programmes), les premiers travaux sur l'interrogation concernent `DATALOG` (Gallaire *et al.*, 1984 ; Ceri *et al.*, 1989 ; Abiteboul *et al.*, 1995). Néanmoins, ce formalisme ne permet pas de représenter les ontologies couramment admises et des extensions intégrant les variables quantifiées existentiellement ont été proposées comme `DATALOG+/-` ou les règles existentielles (Baget *et al.*, 2011).

Dans la mesure où les données représentées peuvent être issues de plusieurs sources dont les connaissances sont plus ou moins précises, certaines données peuvent être incomplètes ou bien l'ensemble des données peut être incohérent. Cet aspect des connaissances n'est pas représentable dans les ontologies traditionnelles pour lesquelles les connaissances sont sûres et cohérentes. La réponse à une requête est alors unique (un seul ensemble d'informations). Si on revient à notre exemple de l'ontologie `university`, on ne peut pas représenter d'informations admettant des exceptions telles que « Par défaut, un enseignant-chercheur n'est pas étudiant sauf s'il est doctorant ».

Pour répondre à cette problématique, il a fallu étendre les modèles de représentation utilisés pour le traitement des ontologies afin d'intégrer la prise en compte de la négation par défaut. L'ajout de cette dernière dans DATALOG amène à définir DATALOG$^{\neg}$ (Abiteboul *et al.*, 1995). Dans ce formalisme, sont considérés des programmes stratifiés et des programmes utilisant la sémantique bien fondée (Van Gelder *et al.*, 1991). Dans ces cas, pour chaque programme, il existe un modèle unique. Par ailleurs, notons, qu'en pratique, le raisonneur NoHR (Costa *et al.*, 2015), basé sur la sémantique bien fondée, est le seul qui accepte les ontologies avec défauts.

D'un autre côté, le formalisme de l'*Answer Set Programming* (ASP) (Gelfond, Lifschitz, 1988) est adéquat pour la prise en compte de la négation par défaut et permet ainsi d'augmenter l'expressivité des ontologies. En ASP, un problème est représenté sous forme d'un programme qui est un ensemble de règles logiques avec négation par défaut et les réponses au problème sont des ensembles d'atomes appelés *answer sets*. L'ASP est basé sur la sémantique des modèles stables qui diffère de la sémantique bien fondée. Un point important est que, contrairement à la sémantique bien fondée pour laquelle il n'y a qu'un seul modèle, un même programme ASP peut avoir aucun, un ou plusieurs *answer sets*. Notons que la résolution est basée sur une technique de marche avant. Par ailleurs, d'un point de vue pratique, de nombreux solveurs efficaces en ASP sont disponibles, en particulier DLV (Leone *et al.*, 2006), WASP (Alviano *et al.*, 2013) et Clasp (Gebser *et al.*, 2007).

En ASP, il y a assez peu de travaux sur l'interrogation car la sémantique des modèles stables se prête mal à une marche arrière partant du but. En effet, dans une résolution en marche avant, on utilise *a priori* toutes les règles du programme ; mais, pour être plus efficace, il serait intéressant d'isoler les seules règles utiles pour répondre à la requête. Le problème est qu'en ASP, si on se contente d'utiliser les règles en lien avec la requête alors la réponse risque d'être incorrecte. De fait, si un programme n'a aucun *answer set*, la réponse à une requête n'a pas de sens. L'isolation des règles doit donc prendre en compte ces cas d'incohérence. Cette spécificité est inhérente à la sémantique des modèles stables qui ne respecte pas les propriétés de pertinence (la dérivabilité d'un atome dépend seulement du sous-programme dont cet atome dépend dans le graphe de dépendance) et de modularité (la sémantique d'un programme peut être composée à partir de la sémantique de ses sous-programmes) (Dix, 1992).

À notre connaissance, les seuls travaux spécifiques sur l'interrogation en ASP concernent le solveur DLV. Cependant, ceux-ci ne permettent pas de traiter tous les programmes mais ne concernent que des catégories particulières : programmes avec variables existentiellement quantifiées sans négation par défaut (Alviano *et al.*, 2012) ou programmes ASP possédant au moins un modèle quels que soient les faits initiaux, appelés super-consistants (Alviano, Faber, 2011). D'autre travaux liés à l'interrogation en ASP sont des applications sur le solveur Clingo. Le système *quontroller* (Gebser *et al.*, 2013) est une sur-couche du solveur « réactif » *oclingo* qui utilise la capacité de ce solveur à intégrer de nouvelles informations durant le processus de résolution pour pouvoir ajouter des questions successivement en reprenant l'exécution du solveur. Les questions sont implémentées via des contraintes qui vont être ajou-

tées temporairement dans le processus de résolution. Les capacités incrémentales et réactives du solveur sont intégrées aujourd'hui dans le solveur *clingo 4* (Gebser *et al.*, 2014).

L'interrogation dans le cadre de règles avec négation par défaut a été beaucoup étudiée ces dernières années sous l'angle de la décidabilité et de la complexité. L'objectif de la plupart de ces travaux est de définir des fragments de règles existentielles ou DATALOG+/- avec négation qui soient décidables et de complexité raisonnable. Pour ce faire, la plupart des approches définissent des restrictions permettant d'assurer l'existence d'un seul modèle : DATALOG+/- gardé et stratifié (Cali *et al.*, 2010 ; Arenas *et al.*, 2014), DATALOG+/- gardé avec sémantique bien fondée (Hernich *et al.*, 2013 ; Gottlob *et al.*, 2012), règles existentielles gardées avec sémantique des modèles stables (Gottlob *et al.*, 2014), règles existentielles avec sémantique des modèles stables avec des conditions de stratification et d'acyclicité garantissant l'unicité et la finitude des modèles (Magka *et al.*, 2013). Les travaux de (Alviano *et al.*, 2017) concernent également les problèmes de complexité de l'interrogation, mais ils se situent dans le cadre d'une caractérisation des *answer sets* par la logique du second ordre (Ferraris *et al.*, 2011) qui fournit une approche alternative à la skolémisation pour la sémantique des variables existentiellement quantifiées. D'autre part, des approches hybrides qui combinent programmation logique avec sémantique des modèles stables et logiques de description sont proposées dans (Lukasiewicz, 2004 ; Motik, Rosati, 2008). Ces dernières approches sont les seules, à notre connaissance, à avoir fait l'objet d'une implémentation (Eiter *et al.*, 2008). Enfin, certains travaux ont traité de l'ajout de la non-monotonie dans le cadre des règles existentielles (Baget *et al.*, 2014) ou de la transformation des règles existentielles en programme ASP via la skolémisation et des liens entre les deux formalismes (Garreau *et al.*, 2015 ; Baget *et al.*, 2018). Suite à ces derniers travaux, nous nous focalisons dans cet article sur l'interrogation en ASP sans variables existentiellement quantifiées.

Nous nous plaçons dans le cadre de la programmation ASP basée sur la sémantique des modèles stables pour laquelle nous nous intéressons à la mise en œuvre pratique et effective de l'interrogation. Dans un premier temps, nous définissons le principe de l'interrogation en ASP, inspiré par l'interrogation dans les ontologies. Nous traitons les problèmes soulevés par ASP en ce qui concerne l'inconsistance et l'utilité d'un pré-traitement sur un programme pour tester la consistance de celui-ci. Nous donnons ensuite des méthodes permettant d'optimiser l'interrogation dans le cadre de l'ASP en isolant les règles nécessaires pour répondre à une requête spécifique. Nous présentons enfin une implémentation de l'interrogation définie dans l'article et utilisant des solveurs ASP. Nous réalisons des tests sur des ontologies traditionnelles (sans négation par défaut) ainsi que sur des ontologies autorisant les exceptions (avec négation par défaut). Les résultats obtenus montrent que le formalisme ASP est adapté à un traitement efficace des ontologies traditionnelles et permet en outre une plus grande expressivité. Les preuves des théorèmes sont données en annexe.

2. Préliminaires

2.1. Answer Set Programming

Nous présentons ici les bases de l'*Answer Set Programming*. Nous considérons des programmes logiques au premier ordre sans disjonction.

Soient $\mathcal{V}$ l'ensemble des *variables*, $\mathcal{FS}$ l'ensemble des *symboles de fonction*, $\mathcal{CS}$ l'ensemble des *symboles de constante* et $\mathcal{PS}$ l'ensemble des *symboles de prédicat*. On suppose que les ensembles $\mathcal{V}$, $\mathcal{CS}$, $\mathcal{FS}$ et $\mathcal{PS}$ sont disjoints et que l'ensemble $\mathcal{CS}$ est non vide. La fonction ar représente la fonction d'arité, de $\mathcal{FS}$ dans $\mathbb{N}^*$ et de $\mathcal{PS}$ dans $\mathbb{N}$, qui associe à chaque fonction ou symbole de prédicat son arité. Soit $\mathbf{T}$ l'ensemble des *termes* défini par induction par :

- si $v \in \mathcal{V}$ alors $v \in \mathbf{T}$,
- si $c \in \mathcal{CS}$ alors $c \in \mathbf{T}$,
- si $f \in \mathcal{FS}$ avec $ar(f) = n$ et $t_1, \ldots, t_n \in \mathbf{T}$ alors $f(t_1, \ldots, t_n) \in \mathbf{T}$.

L'*univers de Herbrand*, noté $\mathcal{U}$, est l'ensemble des termes construits seulement à partir des deux derniers points de la définition précédente. Autrement dit, l'univers de Herbrand est l'ensemble des termes sans variables.

Soit $\mathbf{A}$ l'ensemble des *atomes* défini de la manière suivante :

- si $a \in \mathcal{PS}$ avec $ar(a) = 0$ alors $a \in \mathbf{A}$,
- si $p \in \mathcal{PS}$ avec $ar(p) = n > 0$ et $t_1, \ldots, t_n \in \mathbf{T}$ alors $p(t_1, \ldots, t_n) \in \mathbf{A}$.

La *base de Herbrand*, notée $\mathcal{A}$, est l'ensemble des atomes construits seulement à partir des termes de l'univers de Herbrand.

Un *programme ASP* (ou plus simplement un *programme*) est un couple $(\mathcal{F}, \mathcal{R})$ avec $\mathcal{R}$ un ensemble de règles de la forme :

$$(c \leftarrow a_1,\ \ldots,\ a_n,\ not\ b_1,\ \ldots,\ not\ b_m.) \quad n \geq 0, m \geq 0, n + m > 0 \qquad (1)$$

où $c, a_1, \ldots, a_n, b_1, \ldots, b_m$ sont des atomes et $\mathcal{F}$ un ensemble de faits (ou la base de faits) de la forme $(c.)$ où c est un élément de la base de Herbrand [1]. Pour une règle r (ou par extension un ensemble de règles), on utilise les notations suivantes : $tête(r) = c$ sa *tête*, $corps^+(r) = \{a_1, \ldots, a_n\}$ son *corps positif*, $corps^-(r) = \{b_1, \ldots, b_m\}$ son *corps négatif*, $corps(r) = corps^+(r) \cup corps^-(r)$ les atomes de son *corps* et $r^+ = tête(r) \leftarrow corps^+(r)$.

Le symbole not est une négation par défaut (appelée également négation faible). Une règle de cette forme signifie intuitivement "si tous les a_i sont vrais et si l'on peut considérer tous les b_j comme faux, alors on peut conclure que c est vrai". De plus,

1. Dans notre présentation d'ASP, et pour faire apparaître la base de faits, nous séparons les faits (règles avec corps vide) des règles (règles avec corps non vide). Par abus de notation, on pourra noter la base de faits par un ensemble d'atomes.

chaque règle du programme est *safe* c'est-à-dire que toutes les variables de la règle apparaissent au moins une fois dans un atome du corps positif de la règle.

On peut également exprimer une contrainte à l'aide d'une règle sans tête considérée équivalente à $(bug \leftarrow \ldots,\ not\ bug.)$ avec bug un nouveau symbole de prédicat n'apparaissant nulle part ailleurs. En outre, on dit qu'une règle est *monotone* (respectivement *non monotone*) si son corps négatif est vide (respectivement non vide).

Un programme *défini* $P = (\mathcal{F}, \mathcal{R})$ est tel que $\mathcal{F}$ est un ensemble de faits et $\mathcal{R}$ est un ensemble de règles monotones. Un programme défini ne possède donc aucune négation par défaut.

Pour chaque programme P, on considère que l'ensemble $\mathcal{CS}$ (respectivement $\mathcal{FS}$ et $\mathcal{PS}$) représente tous les symboles de constante (respectivement de fonction et de prédicat) qui apparaissent dans P.

Une *substitution* est une fonction $\sigma : \mathcal{V} \rightarrow \mathcal{U}$ des variables dans les termes de l'univers de Herbrand notée $[X_1 \mapsto t_1, \ldots, X_n \mapsto t_n]$ avec :

- pour tout i, $1 \leq i \leq n$, $X_i \in \mathcal{V}$,
- pour tout i, $1 \leq i \leq n$, $t_i \in \mathcal{U}$,
- pour tout i, j, $1 \leq i, j \leq n$ avec $i \neq j$, $X_i \neq X_j$.

Cette fonction de substitution peut être étendue aux termes, aux atomes et aux règles. Le résultat de l'application d'une substitution $\sigma : \mathcal{V} \rightarrow \mathcal{U}$, constituée de couples $X_i \mapsto t_i$, à un terme (respectivement à un atome ou une règle) t, noté $\sigma(t)$ est le terme (respectivement l'atome ou la règle) t dont toutes les occurrences des X_i ont été remplacées simultanément par les t_i. On dit alors que $\sigma(t)$ est une *instance* de t.

Une *règle instanciée* r' issue d'une règle r d'un programme P est une règle dans laquelle chaque variable de r est remplacée par un terme de l'univers de Herbrand $\mathcal{U}$ de P. Pour une règle r d'un programme P, on définit $ground(r)$ l'ensemble des règles instanciées obtenu en substituant chaque variable de r par un terme de l'univers de Herbrand de P. On définit alors $ground(P) = \bigcup_{r \in P} ground(r)$ le programme sans variables obtenu à partir du programme P au premier ordre.

Un ensemble d'atomes X inclus dans $\mathcal{A}$ est clos par rapport à un programme défini $P = (\mathcal{F}, \mathcal{R})$ si $\mathcal{F} \subseteq X$ et si pour toute règle r de $\mathcal{R}$ et toute substitution σ, si $corps(\sigma(r)) \subseteq X$ alors $tête(\sigma(r)) \in X$. On note $Cn(P)$ le plus petit ensemble d'atomes clos par rapport à P. $Cn(P)$ est le modèle de Herbrand minimal de P.

Le modèle de Herbrand minimal d'un programme P est aussi le plus petit point fixe de l'opérateur de conséquence immédiate T_P défini ci-après.

Soit $P = (\mathcal{F}, \mathcal{R})$ un programme défini et X un ensemble d'atomes tel que $X \subseteq \mathcal{A}$. L'*opérateur de conséquence immédiate* $T_P : 2^{\mathcal{A}} \rightarrow 2^{\mathcal{A}}$ est défini de la manière suivante (Lloyd, 1987) :

$$T_P(X) = \\ \mathcal{F} \cup \{tête(\sigma(r)) \mid r \in \mathcal{R},\ \sigma \text{ est une substitution telle que } \sigma(corps^+(r)) \subseteq X\}$$

On définit la suite T_P^i par : $T_P^0 = \emptyset$, $T_P^1 = \mathcal{F}$ et, pour tout i > 1, $T_P^{i+1} = T_P(T_P^i)$. Cette suite admet un plus petit point fixe $\bigcup_{i \geq 0} T_P^i = Cn(P)$.

Le *réduit* de Gelfond et Lifschitz (Gelfond, Lifschitz, 1988) d'un programme ASP P par rapport à un ensemble d'atomes $X \subseteq \mathcal{A}$ est le programme défini P^X obtenu à partir de $ground(P)$ en supprimant :

– chaque instance de règle ayant un atome b dans son corps négatif tel que $b \in X$ et

– tous les corps négatifs des instances de règles restantes.

Il est exprimé de la façon suivante : $P^X = (\mathcal{F}', \mathcal{R}')$ avec

$$\begin{aligned} \mathcal{F}' = \quad & \mathcal{F} \cup \{tête(\sigma(r)). \mid \quad r \in \mathcal{R},\ \sigma \text{ une substitution}, \\ & \qquad\qquad\qquad\quad \sigma(corps^-(r)) \cap X = \emptyset, corps^+(r) = \emptyset\} \\ \text{et} \quad \mathcal{R}' = \quad & \{\sigma(r^+) \mid \quad r \in \mathcal{R},\ \sigma \text{ une substitution}, \\ & \qquad\qquad\qquad\quad \sigma(corps^-(r)) \cap X = \emptyset, corps^+(r) \neq \emptyset\} \end{aligned}$$

Soit P un programme et X un ensemble d'atomes tel que $X \subseteq \mathcal{A}$. X est un *answer set* (ou *modèle stable*) de P si $X = Cn(P^X)$.

Un programme ASP P peut avoir zéro, un ou plusieurs *answer sets*. On dit que P est *inconsistant* s'il ne possède aucun *answer set* et *consistant* s'il en possède au moins un.

La manière la plus répandue dans les solveurs pour déterminer les *answer sets* d'un programme ASP est composée de deux phases distinctes. D'une part, une phase d'instanciation, que l'on nomme généralement le *grounding*, consiste à transformer le programme ASP initial P en un programme propositionnel P' ne contenant plus de variable mais conservant les mêmes *answer sets* que le programme P, comme le font le *grounder* interne de `DLV` (Faber *et al*., 2012) ainsi que `Gringo` (Gebser *et al*., 2011). D'autre part, une phase de résolution calcule les *answer sets* du programme P' (et par conséquent, les *answer sets* de P), comme dans les solveurs `DLV` et `Clasp`.

La phase d'instanciation ne cède sa place à la phase de résolution que lorsque le programme initial a été totalement instancié ce qui peut parfois être long et coûteux lorsque le *grounding* aboutit à un nombre exponentiel de règles par rapport au programme initial.

Bien que cette méthode soit souvent efficace, la phase d'instanciation complète préalable à la phase de résolution pose parfois des problèmes irrémédiables. C'est pourquoi des nouveaux solveurs, comme `ASPeRiX` (Lefèvre *et al*., 2017) et `Alpha` (Weinzierl, 2017), sont apparus ces dernières années avec pour objectif d'intégrer la phase d'instanciation à la recherche d'*answer sets*. Ces différents solveurs appliquent un chaînage avant sur les règles qui sont instanciées à la volée durant le processus de résolution.

2.2. Graphes de dépendance des symboles de prédicat

Cette section présente des graphes de dépendance entre symboles de prédicats utilisés, entre autres, pour l'instanciation intelligente des programmes.

Il existe différents graphes permettant de représenter les dépendances entre les éléments d'un programme et ainsi analyser le comportement d'un programme. Les graphes nous donnent la possibilité de détecter si une règle est susceptible d'en déclencher une autre et donc est susceptible de déduire une information à partir des faits initiaux. Nous précisons tout d'abord la notation utilisée pour les graphes : Un *graphe orienté* $\mathcal{G}$ est un couple (S,A) avec S un ensemble de sommets et A un ensemble d'arcs. Un *arc* est un couple de sommets (S_1, S_2) orienté de S_1 vers S_2. Un *chemin* dans un graphe orienté est une suite finie d'arcs $(A_1, \ldots, A_n)$ avec $A_i = (S_i, S_{i+1})$, pour tout i, $i \leq n$. Un *circuit* (ou *cycle*) est un chemin $(A_1, \ldots, A_n)$ tel que $S_{n+1} = S_1$.

Le *graphe de dépendance des symboles de prédicat* (Apt, Bol, 1994) d'un programme est un graphe orienté dont les sommets sont étiquetés par les symboles de prédicat d'un programme et dont les arcs représentent les dépendances entre symboles de prédicat. Un symbole de prédicat p dépend d'un symbole de prédicat q s'il existe une règle r telle que p est le symbole de prédicat de $tête(r)$ et q est le symbole de prédicat d'un atome apparaissant dans $corps(r)$. Dans le graphe apparaît un arc allant du symbole de prédicat q au symbole de prédicat p.

En plus des définitions précédentes nous devons prendre en compte des dépendances spéciales pour le cas de la négation par défaut en ASP. Un symbole de prédicat p *dépend positivement* (resp. *négativement*) d'un symbole de prédicat q s'il existe une règle r telle que p est le symbole de prédicat de $tête(r)$ et q est le symbole de prédicat d'un atome apparaissant dans $corps^+(r)$ (resp. $corps^-(r)$). Dans le graphe apparaît un arc positif (resp. négatif) allant du symbole de prédicat q vers le symbole de prédicat p. [2]

3. Interrogation en ASP

L'interrogation a pour but de tester l'existence d'un élément ou de calculer l'ensemble des éléments vérifiant une requête. La représentation utilisée pour les requêtes ASP est la même que celle utilisée pour DATALOG (Gallaire *et al.*, 1984 ; Ceri *et al.*, 1989 ; Abiteboul *et al.*, 1995). Une requête est représentée sous forme de règle afin de l'intégrer à l'ensemble des règles du programme.

DÉFINITION 1 (Requête conjonctive). — *Une* requête conjonctive *sur un programme est une règle* safe *de la forme :*

$$\texttt{ans}(\texttt{X}_1, \ldots, \texttt{X}_k) \leftarrow \texttt{a}_1, \ldots, \texttt{a}_n, \texttt{not b}_1, \ldots, \texttt{not b}_m.$$

2. Ces notions seront illustrées dans l'exemple 8.

avec $k \geq 0$, $n + m > 0$, $\{\mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{b}_1, \ldots, \mathtt{b}_m\}$ *un ensemble d'atomes et* ans *un symbole de prédicat n'apparaissant pas dans le programme ni dans le corps de la requête, appelé* prédicat réponse. *Une* requête conjonctive *est* booléenne *si* $k = 0$.

Nous simplifions par *requête* lorsque nous parlons des requêtes conjonctives et conjonctives booléennes indifféremment.

Nous pouvons ainsi intégrer une requête Q à l'ensemble des règles $\mathcal{R}$ d'un programme P. Soit $P = (\mathcal{F}, \mathcal{R})$ un programme ASP avec $\mathcal{F}$ la base de faits, $\mathcal{R}$ l'ensemble des règles et soit Q une requête sur le programme P, nous notons $(P?Q) = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ le programme P requêté par Q. Nous obtenons aisément le résultat suivant qui met en correspondance les *answer sets* d'un programme P et les *answer sets* de P requêté par Q : si $AS_1, \ldots, AS_n$ sont les *answer sets* de P alors les ensembles $AS_1^Q, \ldots, AS_n^Q$ tels que

$$AS_i^Q = AS_i \cup \{tête(\sigma(Q)) \mid corps^+(\sigma(Q)) \subseteq AS_i\}$$

pour tout i, $1 \leq i \leq n$, et pour toute substitution σ, sont les *answer sets* de $(P?Q)$.

Pour traiter le problème de l'interrogation nous allons utiliser dans les exemples un programme inspiré de l'ontologie `university` mais n'utilisant que des symboles de prédicat d'arité 1 afin de mettre en avant des problèmes spécifiques à l'utilisation de la négation par défaut. Les règles de l'ontologie originale ne comportant pas d'exception, elles ne permettent pas de mettre en valeur les problèmes que nous souhaitons souligner, nous avons donc créé des exemples spécifiques à ASP à partir de `university`.

EXEMPLE 2. — *Par la suite, nous utilisons les symboles de prédicat suivants :* `pU` *pour « personnel de l'université »,* `ad` *pour « personnel de l'administration »,* `dir` *pour « membre de la direction »,* `cA` *pour « membre du conseil d'administration »,* `eC` *pour « enseignant-chercheur »,* `mCP` *pour « professeur ou maître de conférences »,* `et` *pour « étudiant »,* `ch` *pour « chômeur »,* `chr` *pour « a une charge de recherche »,* `etT` *pour « étudiant en thèse »,* `docteur` *pour « dispose du grade de docteur ». Tous les symboles de prédicat sont d'arité 1.*

Soit $P_2 = (\mathcal{F}_2, \mathcal{R}_2)$ *le programme suivant avec* $\mathcal{R}_2 =$

$$\left\{\begin{array}{ll} r_1 : \mathtt{ad(X)} \leftarrow \mathtt{pU(X)}, \mathtt{not\ eC(X)}., & r_2 : \mathtt{eC(X)} \leftarrow \mathtt{pU(X)}, \mathtt{not\ ad(X)}., \\ r_3 : \mathtt{cA(X)} \leftarrow \mathtt{ad(X)}, \mathtt{dir(X)}., & r_4 : \mathtt{mCP(X)} \leftarrow \mathtt{eC(X)}, \mathtt{not\ etT(X)}., \\ r_5 : \mathtt{etT(X)} \leftarrow \mathtt{eC(X)}, \mathtt{et(X)}., & r_6 : \mathtt{chr(X)} \leftarrow \mathtt{eC(X)}., \\ r_7 : \mathtt{c1} \leftarrow \mathtt{mCP(X)}, \mathtt{etT(X)}, \mathtt{not\ c1}., & r_8 : \mathtt{c2} \leftarrow \mathtt{et(X)}, \mathtt{ch(X)}, \mathtt{not\ c2}., \\ r_9 : \mathtt{docteur(X)} \leftarrow \mathtt{mCP(X)}. & \end{array}\right\}$$

$$\mathcal{F}_2 = \{\mathtt{ad(marie)}, \mathtt{dir(marie)}, \mathtt{pU(jean)}, \mathtt{dir(jean)}, \mathtt{et(jean)}\}$$

et $Q_2 : (\mathtt{ans(X)} \leftarrow \mathtt{cA(X)}.)$.

Notons que les règles r_7 *et* r_8 *sont des contraintes. Les atomes* $c1$ *et* $c2$ *n'ont donc pas de sens particulier et ne servent qu'à représenter les contraintes sous forme de règles ordinaires (avec tête). La requête* Q_2 *exprime la question : « Quels sont les membres du conseil d'administration ? ».*

Le programme P_2 possède deux answer sets :

$$AS_1 = \{ \ \texttt{ad(marie)}, \texttt{dir(marie)}, \texttt{pU(jean)}, \texttt{dir(jean)}, \texttt{et(jean)}, \texttt{cA(marie)}, \texttt{ad(jean)}, \texttt{cA(jean)}\}$$
$$AS_2 = \{ \ \texttt{ad(marie)}, \texttt{dir(marie)}, \texttt{pU(jean)}, \texttt{dir(jean)}, \texttt{et(jean)}, \texttt{cA(marie)}, \texttt{eC(jean)}, \texttt{etT(jean)}, \texttt{chr(jean)}\}$$

Si nous ajoutons Q_2 à l'ensemble des règles $\mathcal{R}_2$ de P_2 alors la règle Q_2 est déclenchée et l'atome $\texttt{ans}(.)$ *est ajouté à l'answer set. Donc le programme $(P_2?Q_2)$ possède deux answer sets : $AS_1^Q = AS_1 \cup \{\texttt{ans(marie)}, \texttt{ans(jean)}\}$ et $AS_2^Q = AS_2 \cup \{\texttt{ans(marie)}\}$.*

L'interrogation, dans le cas d'un programme ASP, doit prendre en compte l'aspect non monotone qui induit la possibilité d'avoir aucun, un ou plusieurs *answer sets*. Ainsi nous définissons plusieurs réponses possibles pour une requête qui peuvent être soit une réponse crédule si la réponse est présente dans au moins un *answer set*, soit une réponse sceptique si la réponse est présente dans tous les *answer sets*. Le problème de l'interrogation en ASP réside dans le fait qu'un programme peut ne pas avoir d'*answer set* et être donc inconsistant. Notons que la non existence d'un *answer set* ($\mathcal{AS} = \emptyset$), est différente de l'existence d'un *answer set* vide ($\mathcal{AS} = \{\emptyset\}$). Nous proposons ici de considérer la réponse à un programme inconsistant comme absurde.

Nous distinguons deux types de requêtes : les requêtes conjonctives classiques (non-booléennes) dont la réponse est un ensemble d'atomes [3], et les requêtes conjonctives booléennes dont la réponse est `vrai` ou `faux`.

DÉFINITION 3 (Réponse à une requête conjonctive). — *Soit le programme $P = (\mathcal{F}, \mathcal{R})$ requêté par Q :* $(\texttt{ans}(\texttt{X}_1, \ldots, \texttt{X}_k) \leftarrow \texttt{a}_1, \ldots, \texttt{a}_n, \texttt{not b}_1, \ldots, \texttt{not b}_m.)$ *avec $k > 0$ et $n > 0$, une requête conjonctive. Soit $\mathcal{AS}$ l'ensemble des answer sets de $(P?Q)$. La* réponse *à Q dans P [4] est* `absurde` *si $\mathcal{AS} = \emptyset$ sinon la réponse est* valide *et est l'ensemble des substitutions σ telles que :*

- $\forall AS \in \mathcal{AS}, \ \sigma(\texttt{ans}(\texttt{X}_1, \ldots, \texttt{X}_k)) \in AS$ *pour la* réponse sceptique*,*
- $\exists AS \in \mathcal{AS}, \ \sigma(\texttt{ans}(\texttt{X}_1, \ldots, \texttt{X}_k)) \in AS$ *pour la* réponse crédule*.*

EXEMPLE 4. — *Soit le programme P_2 avec la requête conjonctive Q_2. La réponse sceptique à Q_2 est* $\{[\texttt{X} \mapsto \texttt{marie}]\}$ *car* $\texttt{ans}(\texttt{marie})$ *appartient aux deux answer sets. La réponse crédule, quant à elle, est* $\{[\texttt{X} \mapsto \texttt{marie}], [\texttt{X} \mapsto \texttt{jean}]\}$ *(nous avons donc deux substitutions possibles pour* `X`*).*

DÉFINITION 5 (Réponse à une requête booléenne conjonctive). — *Soit le programme $P = (\mathcal{F}, \mathcal{R})$ requêté par Q :* $(\texttt{ans} \leftarrow \texttt{a}_1, \ldots, \texttt{a}_n, \texttt{not b}_1, \ldots, \texttt{not b}_m.)$ *avec $n > 0$, une requête booléenne conjonctive et $\mathcal{AS}$ l'ensemble des answer sets de $(P?Q)$. Si $\mathcal{AS} = \emptyset$ alors la réponse à Q dans P [4] est* `absurde`*,*

3. La réponse est un ensemble de substitutions pour l'atome réponse $\texttt{ans}(.)$ que, par abus, nous identifions à l'ensemble d'atomes réponse instanciés correspondant.

4. la réponse dans le programme $(P?Q)$

sinon la réponse est valide *et :*

– *pour la réponse sceptique :*
  - *si* $\forall AS \in \mathcal{AS}$, $\texttt{ans} \in AS$ *alors la réponse est* `vrai`
  - *sinon la réponse est* `faux`
– *pour la réponse crédule :*
  - *si* $\exists AS \in \mathcal{AS}$, $\texttt{ans} \in AS$ *alors la réponse est* `vrai`
  - *sinon la réponse est* `faux`

EXEMPLE 6. — *Soit le programme* P_2 *avec la requête conjonctive* Q_6 : $(\texttt{ans} \leftarrow \texttt{cA}(\texttt{jean}).)$. *Les deux answer sets de* $(P_2?Q_6)$ *sont les suivants :*

$$AS_1 = \{\ \texttt{ad}(\texttt{marie}), \texttt{dir}(\texttt{marie}), \texttt{pU}(\texttt{jean}), \texttt{dir}(\texttt{jean}), \texttt{et}(\texttt{jean}), \texttt{cA}(\texttt{marie}), \texttt{ad}(\texttt{jean}), \texttt{cA}(\texttt{jean}), \texttt{ans}\}$$
$$AS_2 = \{\ \texttt{ad}(\texttt{marie}), \texttt{dir}(\texttt{marie}), \texttt{pU}(\texttt{jean}), \texttt{dir}(\texttt{jean}), \texttt{et}(\texttt{jean}), \texttt{cA}(\texttt{marie}), \texttt{eC}(\texttt{jean}), \texttt{chr}(\texttt{jean}), \texttt{etT}(\texttt{jean})\}$$

La réponse sceptique à Q_6 *est* `faux` *car* ans *n'appartient pas à* AS_2. *La réponse crédule est* `vrai` *car* ans *appartient à l'answer set* AS_1.

D'autres approches de l'interrogation en ASP ont été effectuées dans (Alviano, Faber, 2011). En comparaison, dans la documentation du solveur ASP `DLV` (*DLV - User Manual*, 2004) il est proposé de considérer une réponse à une requête comme vide ($\emptyset$) lorsqu'un programme est inconsistant. Ceci correspond dans notre cas à la réponse `absurde`, le programme contenant des faits ou des règles erronés. De même, il est proposé de considérer une réponse sceptique à une requête booléenne comme `vrai` lorsqu'un programme est inconsistant. En effet, nous cherchons l'appartenance de `ans` à tous les modèles, étant donné qu'il n'existe pas de modèle, la réponse est donc `vrai`. A contrario la réponse crédule est `faux` en l'absence de modèle. En considérant la réponse `absurde` dans le cas où le programme est inconsistant on peut différencier la réponse vide (il existe un *answer set* mais aucune réponse n'est présente dans un/tous les *answer sets*) de la réponse `absurde` (il n'existe pas d'*answer set*).

En ASP, nous distinguons donc différentes réponses en fonction de l'appartenance d'une réponse à aucun, un ou plusieurs *answer sets* à l'instar de l'appartenance d'une formule dans une ou toutes les extensions en logique des défauts (Schaub, Thielscher, 1996) et contrairement à d'autres langages, comme `DATALOG`, qui ne possèdent qu'un unique modèle. La méthode la plus simple pour répondre à une requête est de calculer l'ensemble des *answer sets* puis de chercher les réponses à notre requête directement dans les *answer sets* calculés. L'inconvénient est que le temps de calcul sera au minimum aussi long que la recherche de l'ensemble des *answer sets* et, dans le cas d'*answer sets* infinis, le calcul de la réponse ne se terminera jamais. Nous allons donc étudier différentes méthodes permettant de calculer les réponses à notre requête sans avoir à calculer entièrement les *answer sets* et de réduire au maximum le nombre de règles et d'instances nécessaires pour obtenir une réponse. Mais avant cela il faut traiter le problème de l'inconsistance en ASP.

4. Inconsistance et interrogation en ASP

Une des différences notables de l'ASP par rapport à DATALOG est la possibilité d'avoir des programmes inconsistants. Lors de l'interrogation d'un programme, il faut donc s'assurer que celui-ci possède au moins un *answer set* (dans le cas contraire la réponse serait absurde), et cela en effectuant le moins de calcul possible, pour ensuite pouvoir traiter la requête si nécessaire. Cette étape de test de la consistance d'un programme peut être vue comme optionnelle si l'on considère que les données du programme sont valides. Il est donc souvent proposé de considérer des programmes ayant au moins un *answer set* ou bien de ne tester la consistance que sur les données en lien avec la requête, étant donné que s'il y a une inconsistance sur des données déconnectées de la requête celle-ci n'a pas d'impact sur la réponse [5]. Dans notre cas, nous souhaitons nous assurer de la consistance des données pour fournir une réponse en concordance avec celles-ci, en considérant que si le programme est inconsistant alors il y a un problème soit sur les faits soit sur les règles du programme et que la réponse n'aurait pas de sens dans ce cas.

Nous proposons alors une méthode efficace afin de s'assurer qu'il existe une réponse valide à une requête sans avoir à calculer l'ensemble des *answer sets*. Cette méthode consiste à isoler les parties d'un programme pouvant le rendre inconsistant puis vérifier si ces parties rendent effectivement le programme inconsistant. Nous allons alors tester l'existence d'un *answer set* sur chacune de ces parties et montrer que cela assure l'existence d'un *answer set* sur le programme original.

Un des avantages est que notre méthode peut être utilisée en pré-traitement d'une requête et donc ne nécessite pas d'être exécutée une nouvelle fois pour une requête différente sur le même programme, il est même possible de passer cette étape si nous ne souhaitons pas nous assurer de la consistance du programme. Pour ce faire nous reprenons la notion de règles dangereuses de (Alviano *et al.*, 2012) où elle est utilisée dans le cadre de DATALOG¬ (Bancilhon *et al.*, 1986) pour répondre à une requête sur un programme consistant. Nous étendons ici l'utilisation des règles dangereuses pour requêter tout programme ASP.

La première étape pour tester si un programme est inconsistant en limitant le nombre de calculs est d'isoler les parties du programme pouvant être à l'origine d'une inconsistance. Il existe dans la littérature (Dung, 1992) une méthode utilisant le graphe de dépendance des symboles de prédicat (voir section 2.2) pour isoler ces parties dans un programme. Pour cela nous cherchons à identifier les *cycles d'inconsistance* qui sont les cycles du graphe de dépendance des symboles de prédicat comprenant un nombre impair d'arcs négatifs.

THÉORÈME 7 (Inconsistance). — *(Dung, 1992) Un programme peut être inconsistant seulement s'il possède un cycle d'inconsistance.*

5. i.e., il peut être acceptable, en pratique, de répondre à la requête en ignorant l'inconsistance même si, en théorie, la réponse est incorrecte.

EXEMPLE 8. — *Nous établissons en figure 1 le graphe de dépendance des symboles de prédicat du programme* $(P_2?Q_2)$.

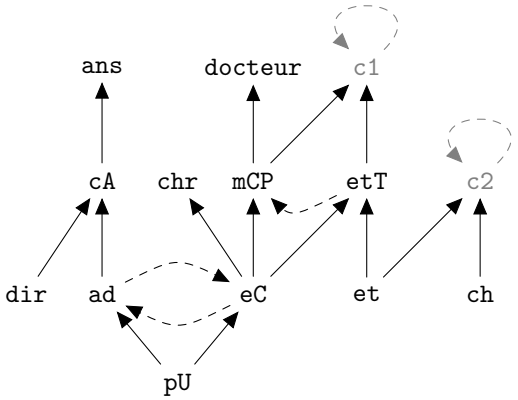

Figure 1. Graphe de dépendance des symboles de prédicat pour le programme de l'exemple 2. Les dépendances positives (resp. négatives) sont représentées par des arcs pleins (resp. pointillés)

Cet exemple possède deux cycles impairs d'arcs négatifs sur les symboles de prédicat c1 *et* c2 *représentés en pointillés gris sur le graphe associé. Ces deux cycles d'inconsistance de l'ensemble de règles* $\mathcal{R}_2 \cup \{Q_2\}$ *sont issus des règles* r_7 *et* r_8 *qui sont les deux contraintes pouvant rendre le programme inconsistant.*

DÉFINITION 9 (Dépendance d'un cycle). — *Un cycle d'inconsistance* C *dépend d'un symbole de prédicat* p *s'il existe un chemin dans le graphe des symboles de prédicat allant de* p *à un symbole de prédicat de* C. *Un cycle* C *dépend d'une règle* r *si* C *dépend du symbole de prédicat de* $\hat{t}\hat{e}te(r)$.

Nous obtenons aisément la propriété suivante : si un programme $P = (\mathcal{F}, \mathcal{R})$ est consistant (resp. inconsistant) alors le programme $(P?Q)$ est aussi consistant (resp. inconsistant) pour toute requête Q. En effet, le symbole de prédicat ans de l'atome de tête de Q n'apparaissant nulle part ailleurs dans le programme, il ne participe à aucun cycle et n'a donc aucun impact sur la consistance du programme.

Nous savons que l'inconsistance d'un programme est liée aux cycles d'inconsistance mais ce que nous souhaitons c'est isoler les règles d'un programme pouvant être responsables de l'inconsistance. Pour cela nous allons nous intéresser aux *règles dangereuses* de (Alviano *et al.*, 2012). Les règles identifiées comme dangereuses sont toutes les règles d'un programme dont l'application est susceptible de le rendre inconsistant. Ce sont toutes les règles dont un cycle d'inconsistance dépend. Nous associons alors un ensemble de règles dangereuses à chaque cycle d'inconsistance. Une fois les règles dangereuses identifiées sous forme d'ensembles nous verrons ensuite qu'il suffit de tester leur consistance pour déterminer si un programme est consistant ou non. Une différence par rapport à (Alviano *et al.*, 2012) est que les règles dangereuses sont signées afin de différencier les règles dangereuses positives, qui peuvent rendre un

programme inconsistant, des règles dangereuses négatives, qui (à l'inverse) si elles sont appliquées pourront bloquer l'application d'une règle dangereuse positive qui aurait rendu le programme inconsistant ou bien l'application d'une règle dangereuse négative pouvant déjà bloquer une autre règle.

DÉFINITION 10 (Règle dangereuse). — *Soit $\mathcal{G}$ le graphe de dépendance des symboles de prédicat d'un programme ASP. Un symbole de* prédicat *est dit* dangereux positif *si :*

– *celui-ci apparaît dans un* cycle d'inconsistance *de $\mathcal{G}$ ou*

– *celui-ci apparaît dans le corps positif d'une règle avec un symbole de prédicat dangereux positif en tête.*

Un symbole de prédicat *est dit* dangereux négatif *si celui-ci n'est pas dangereux positif et si :*

– *celui-ci apparaît dans le corps négatif d'une règle avec un symbole de prédicat dangereux positif en tête ou*

– *celui-ci apparaît dans le corps (positif ou négatif) d'une règle avec un symbole de prédicat dangereux négatif en tête.*

Une règle *est dite* dangereuse positive *(resp.* négative*) si celle-ci possède un symbole de prédicat dangereux positif (resp. négatif) en tête.*
Pour un cycle d'inconsistance de $\mathcal{G}$, on appelle ensemble de règles dangereuses l'ensemble des règles dangereuses positives et négatives dont ce cycle dépend.

Il existe autant d'ensembles de règles dangereuses associées à un graphe qu'il existe de cycles d'inconsistance. Pour déterminer si un programme est inconsistant nous avons besoin des deux types de règles dangereuses, chaque type de règle ayant son rôle. Une instance d'une règle dangereuse positive peut être directement à l'origine d'une inconsistance et nécessite que nous calculions toutes les instances de la règle pouvant provoquer l'inconsistance. Une règle dangereuse négative peut bloquer une autre règle dangereuse pouvant empêcher une inconsistance, si la règle bloquée est dangereuse positive, ou la rétablir, si celle-ci est dangereuse négative, et n'a besoin que des instances pouvant bloquer les instances d'une règle dangereuse. Les règles dangereuses dans leur ensemble servent à traiter l'inconsistance mais nous verrons dans la partie 7.3 que les règles dangereuses négatives peuvent servir à optimiser l'instanciation. Une règle dangereuse ne peut pas être positive et négative à la fois, une règle dangereuse positive nécessitant toutes ses instances prévaudra sur une règle dangereuse négative ne nécessitant que les instances pouvant bloquer une autre règle dangereuse.

EXEMPLE 11. — *Nous considérons en figure 2 à nouveau le programme de l'exemple 2 et son graphe de dépendance des symboles de prédicat associé dans lequel sont identifiés les symboles de prédicat dangereux associés aux deux cycles d'inconsistance de P_2 (entourés en gris clair) notés C_1 et C_2, les symboles de prédicat dangereux positifs (resp. négatifs) sont notés en blanc (resp. gris). Nous pouvons identifier deux ensembles de règles dangereuses dans P_2 (les règles dont le symbole de*

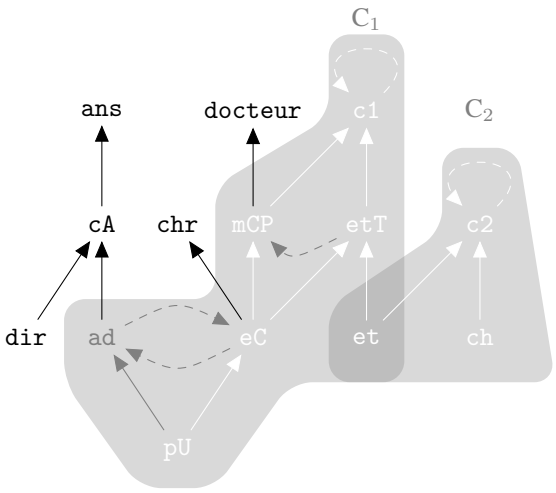

Figure 2. Graphe de dépendance des symboles de prédicat, cycles d'inconsistance et symboles de prédicat dangereux pour le programme de l'exemple 2

prédicat en tête est entouré par une forme grise) représentant les règles dangereuses dont dépend le cycle C_1 et celles dont dépend le cycle C_2.

Pour C_1, nous avons un cycle d'inconsistance sur `c1` *qui dépend de lui-même négativement donc* `c1` *est dangereux positif et* `mCP` *et* `etT` *dont dépend* `c1` *sont aussi dangereux positif de même pour* `et`, `eC` *et* `pU`. *Les règles r_2 [6], r_4 [7], r_5 [8] et r_7 [9] ayant un de ces symboles de prédicat en tête sont donc des règles dangereuses positives. Le symbole de prédicat* `ad` *dont dépend négativement* `eC` *est dangereux négatif. La règle r_1 [10] est donc dangereuse négative.* `mCP` *dépend aussi négativement du symbole de prédicat* `etT` *mais étant donné que celui-ci est déjà dangereux positif il reste dangereux positif.*

Pour le cycle d'inconsistance C_2 nous avons un cycle impair sur `c2` *avec* `et` *et* `ch` *qui sont dangereux positifs donc r_8 [11] est dangereuse positive.*

Pour finir aucun cycle ne dépend de `chr`, `cA`, `dir`, `docteur` *et* `ans` *donc les règles ayant ces symboles de prédicat en tête (r_3, r_6, r_9 et Q_2) ne sont pas dangereuses.*

L'algorithme 1 réalise la fonction $ensembleRèglesDangereuses$ qui est telle que $ensembleRèglesDangereuses(\mathcal{R})$ calcule l'ensemble des ensembles des règles

6. r_2 : `eC(X)` ← `pU(X)`, `not ad(X)`.
7. r_4 : `mCP(X)` ← `eC(X)`, `not etT(X)`.
8. r_5 : `etT(X)` ← `eC(X)`, `et(X)`.
9. r_7 : `c1` ← `mCP(X)`, `etT(X)`, `not c1`.
10. r_1 : `ad(X)` ← `pU(X)`, `not eC(X)`.
11. r_8 : `c2` ← `et(X)`, `ch(X)`, `not c2`.

dangereuses (marquées positivement ou négativement) d'un ensemble de règles $\mathcal{R}$. Nous traitons chaque cycle d'inconsistance afin de calculer toutes les règles dangereuses qui lui sont associées. Nous utilisons pour cela un ensemble de couples règle/marquage pour stocker les règles dangereuses en différenciant les règles dangereuses positives des négatives. Nous appelons d'abord la fonction $gdsp$ qui est telle que $gdsp(\mathcal{R})$ calcule à partir d'une ensemble de règles le graphe de dépendance $\mathcal{G}$ des symboles de prédicat de l'ensemble de règles $\mathcal{R}$. Pour chaque cycle, nous détectons tout d'abord l'ensemble des règles dangereuses positives associées aux symboles de prédicat du cycle. Nous appelons dans un premier temps la fonction $CycleInconsistant(\mathcal{G}, \mathcal{R})$ qui à partir du graphe $\mathcal{G}$ renvoie à chaque appel l'ensemble des règles de $\mathcal{R}$ avec un symbole de prédicat en tête appartenant à un cycle d'inconsistance non parcouru et les marque positivement. Si tous les cycles ont été parcourus la fonction renvoie un ensemble vide. Soit D un ensemble de couples règle/marquage, $règles(D)$ (resp. $règlesPositives(D)$, $règlesNégatives(D)$) renvoie les règles dangereuses (resp. positives, négatives) de D sans le marquage. Une fois les règles d'un cycle calculées nous ajoutons à cet ensemble toutes les règles du programme dont le cycle dépend. Dans un premier temps nous marquons les règles dangereuses positives puis les négatives. Soit $pred(A)$ l'ensemble des symboles de prédicat de l'ensemble d'atomes A. Nous parcourons donc toutes les règles dangereuses positives dont un cycle dépend et nous utilisons la fonction $dangereuses(p, \mathcal{R}, +)$, sur les symboles de prédicat du corps positif, qui, à partir d'un symbole de prédicat p et de l'ensemble de règles du programme $\mathcal{R}$, renvoie l'ensemble des règles qui possèdent ce symbole de prédicat en tête et les marque positivement. Une fois l'ensemble des règles dangereuses positives dépendant du cycle calculées nous appliquons la fonction $dangereuses(p, \mathcal{R}, -)$, sur les corps négatifs de cet ensemble, qui, à partir d'un symbole de prédicat p et de $\mathcal{R}$, renvoie l'ensemble des règles qui possèdent ce symbole de prédicat en tête et les marque négativement à moins qu'elles ne soient déjà marquées positivement. Nous terminons en utilisant $dangereuses(p, \mathcal{R}, -)$ sur le corps (positif et négatif) des règles dangereuses négatives. L'ensemble $\mathcal{D}$ contient tous les ensembles de règles dangereuses marquées détectés.

THÉORÈME 12 (Complétude et correction de l'algorithme 1). — *L'algorithme 1 $ensembleRèglesDangereuses$ calcule l'ensemble des ensembles des règles dangereuses, comme défini dans la définition 10.*

EXEMPLE 13. — *Reprenons l'ensemble de règles de l'exemple 2. Dans cet exemple nous appliquons l'algorithme 1 pour construire l'ensemble d'ensembles de règles dangereuses $\mathcal{D}_2$ de $\mathcal{R}_2$. Soit $\mathcal{G}_2 = gdsp(\mathcal{R}_2)$ le graphe des symboles de prédicat de P_2, il existe deux cycles d'inconsistance dans $\mathcal{G}_2$ le premier sur le symbole de prédicat $c1$ et le second sur le symbole de prédicat* `c2`*. Le premier ensemble de règles dangereuses D_1 sera donc constitué d'abord de la règle r_7 marquée positivement car* `c1` *est le seul symbole de prédicat composant le cycle d'inconsistance C_1 et r_7 la seule règle ayant* `c1` *en tête, donc $D_1 = \{(r_7, +)\}$. Nous continuons en regardant l'ensemble des symboles de prédicat dont dépend* `c1`*, nous avons* `mCP` *et* `etT` *qui sont des symboles de prédicat dangereux positifs ce qui ajoute r_4 et r_5*

Algorithme 1 : $ensembleRèglesDangereuses$

Data : Un ensemble $\mathcal{R}$ de règles
$\mathcal{G} = gdsp(\mathcal{R})$;
$\mathcal{D} = \emptyset$;
while $D := CycleInconsistant(\mathcal{G}, \mathcal{R}) \neq \emptyset$ **do**
  foreach $r \in règles(D)$ **do**
    foreach $p \in pred(corps^{+}(r))$ **do**
      $D := D \cup dangereuses(p, \mathcal{R}, +)$;
    end
  end
  foreach $r \in règlesPositives(D)$ **do**
    foreach $p \in pred(corps^{-}(r))$ **do**
      $D := D \cup dangereuses(p, \mathcal{R}, -)$;
    end
  end
  foreach $r \in règlesNégatives(D)$ **do**
    foreach $p \in pred(corps(r))$ **do**
      $D := D \cup dangereuses(p, \mathcal{R}, -)$;
    end
  end
  $\mathcal{D} := \mathcal{D} \cup \{D\}$;
end
return $\mathcal{D}$

à D_1, étant donné que ces symboles de prédicat apparaissent en tête de ces règles ce qui donne $D_1 = \{(r_7, +), (r_4, +), (r_5, +)\}$. De la même manière nous avons `eC` *dont* `mCP` *et* `etT` *dépendent positivement ce qui fait que r_2 est dangereuse positive et* `pU` *et* `et` *sont des faits donc ils n'engendrent pas de règles dangereuses. Nous avons alors $D_1 = \{(r_7, +), (r_4, +), (r_5, +), (r_2, +)\}$, si nous regardons les dépendances de* `eC` *nous remarquons que celui-ci dépend négativement de* `ad` *ce qui induit que r_1 est une règle dangereuse négative, de même avec la dépendance de* `mCP` *avec* `etT` *et* `ad` *qui dépend négativement de* `eC` *mais cela n'ajoute pas de règle dangereuse négative car celles-ci sont déjà dangereuses positives. Nous avons alors $D_1 = \{(r_7, +), (r_4, +), (r_5, +), (r_2, +), (r_1, -)\}$, et il n'existe pas d'autres dépendances au cycle d'inconsistance C_1.*
Le second ensemble de règles dangereuses D_2 est issu du cycle d'inconsistance C_2 et est composé uniquement du symbole de prédicat `c2`*. Nous avons alors r_8 règle dangereuse positive, donc $D_2 = \{(r_8, +)\}$ car les seuls symboles de prédicat dont dépend le symbole de prédicat* `c2` *sont* `et` *et* `ch` *qui n'apparaissent dans aucune tête de règle. L'algorithme s'arrête ainsi car toutes les dépendances de tous les cycles d'inconsistance ont été parcourues, nous avons alors l'ensemble des ensembles de règles dangereuses de $\mathcal{R}_2$, $\mathcal{D}_2 = ensembleRèglesDangereuses(\mathcal{R}_2) = \{D_1, D_2\}$.*

Afin de différencier les règles dangereuses positives et les règles dangereuses négatives permettant ensemble la détection de l'inconsistance, nous montrons dans les exemples 14 et 15 l'influence de chacune d'entre elles sur le calcul d'un *answer set*.

EXEMPLE 14. — *Soit le programme* $P_{14} = (\mathcal{F}_{14}, \mathcal{R}_2)$*, l'ensemble de règles de l'exemple 2 avec la base de faits* $\mathcal{F}_{14} = \{\texttt{eC(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}\}$. *Nous observons que les règles* r_5 [12] *et* r_7 [13] *avec* $X = \texttt{jean}$ *rendent le programme inconsistant et que c'est le déclenchement de* r_7 *qui provoque l'inconsistance. Ici, c'est donc le symbole de prédicat dangereux positif* etT *qui déclenche* c1*. Si nous choisissons maintenant* $P'_{14} = (\mathcal{F}'_{14}, \mathcal{R}_2)$ *avec* $\mathcal{F}'_{14} = \{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{ad(jean)}\}$*, nous obtenons cette fois-ci un answer set* $\{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{ad(jean)}, \texttt{docteur(jean)}\}$*. L'atome* ad(jean) *bloque la règle* r_2 [14] *et empêche de déduire* eC(jean) *(et, par enchaînement, empêche également d'obtenir* etT(jean) *par la règle* r_5*). Ici, c'est le symbole de prédicat dangereux négatif* ad *qui bloque* eC *et* etT *et donc,* in fine*,* c1*. De même si nous choisissons* $\{\texttt{eC(jean)}, \texttt{etT(jean)}\}$ *l'inconsistance est empêchée car la règle* r_4 [15] *ne peut pas être déclenchée et empêche donc le déclenchement de* r_7 *responsable de l'inconsistance. Dans ce dernier cas, c'est le symbole de prédicat* etT *qui bloque* mCP *et donc* c1*.*

Contrairement aux symboles de prédicat et règles dangereuses positives qui sont la cause de l'inconsistance, les symboles de prédicat et les règles dangereuses négatives peuvent empêcher une inconsistance. Mais une règle dangereuse négative peut aussi rendre son inconsistance à un programme si elle empêche une autre règle dangereuse négative de s'appliquer.

EXEMPLE 15. — *Soit le programme* $P_{15} = (\mathcal{F}_{15}, \mathcal{R}_{15})$ *avec* $\mathcal{R}_{15} =$

$$\left\{\begin{array}{ll} r_3 : \texttt{cA(X)} \leftarrow \texttt{ad(X)}, \texttt{dir(X)}. & r_4 : \texttt{mCP(X)} \leftarrow \texttt{eC(X)}, \texttt{not etT(X)}. \\ r_5 : \texttt{etT(X)} \leftarrow \texttt{eC(X)}, \texttt{et(X)}. & r_7 : \texttt{c1} \leftarrow \texttt{mCP(X)}, \texttt{etT(X)}, \texttt{not c1}. \\ r_8 : \texttt{c2} \leftarrow \texttt{et(X)}, \texttt{ch(X)}, \texttt{not c2}. & r_{10} : \texttt{ad(X)} \leftarrow \texttt{pU(X)}, \texttt{ingenieur(X)}. \\ r_{11} : \texttt{eC(X)} \leftarrow \texttt{pU(X)}, \texttt{not prag(X)}. & \\ r_{12} : \texttt{prag(X)} \leftarrow \texttt{pU(X)}, \texttt{nonchercheur(X)}, \texttt{not ad(X)}. & \end{array}\right\}$$

et $\mathcal{F}_{15} = \{\texttt{pU(jean)}, \texttt{nonchercheur(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}\}$*. Sur ce programme nous savons que* jean *est un personnel universitaire et non chercheur. Si la règle dangereuse positive* r_{11} *est appliquée alors le programme est inconsistant car, par* r_5*, on pourrait déduire* etT(jean) *et la contrainte* r_7 *serait alors violée. Mais* r_{11} *est bloquée par l'application de la règle dangereuse négative* r_{12} *donc le programme reste consistant et nous avons l'answer set* $\{\texttt{pU(jean)}, \texttt{nonchercheur(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{prag(jean)}\}$*. Si nous ajoutons maintenant* ingenieur(jean)*, le programme devient inconsistant car la*

12. $r_5 : \texttt{etT(X)} \leftarrow \texttt{eC(X)}, \texttt{et(X)}.$
13. $r_7 : \texttt{c1} \leftarrow \texttt{mCP(X)}, \texttt{etT(X)}, \texttt{not c1}.$
14. $r_2 : \texttt{eC(X)} \leftarrow \texttt{pU(X)}, \texttt{not ad(X)}.$
15. $r_4 : \texttt{mCP(X)} \leftarrow \texttt{eC(X)}, \texttt{not etT(X)}.$

règle dangereuse négative r_{10} empêche l'application de la règle dangereuse négative r_{12} qui empêchait l'inconsistance. Ainsi nous avons une règle dangereuse négative qui rétablit l'inconsistance de notre programme. Donc les règles dangereuses positives peuvent créer une inconsistance tandis que les règles dangereuses négatives peuvent soit empêcher une inconsistance ou alors rétablir une inconsistance empêchée par une autre règle dangereuse négative.

Les ensembles de règles dangereuses maintenant isolées dans l'ensemble $\mathcal{D}$ nous souhaitons calculer uniquement à partir de l'ensemble $\mathcal{D}$ si un programme est consistant. Jusqu'ici il est possible d'avoir une redondance de règles dangereuses entre deux ensembles de $\mathcal{D}$. Nous souhaitons supprimer ces redondances pour que par la suite un traitement ne soit pas effectué deux fois sur une même règle. De plus, deux ensembles de règles dangereuses peuvent être consistants séparément vis-à-vis d'une base de faits alors que l'union de ces deux ensembles est inconsistant.

EXEMPLE 16. — *Soit P_{16} un programme constitué de l'ensemble de règles*

$$\mathcal{R}_{16} = \left\{ \begin{array}{ll} \rho_1 : \texttt{c1} \leftarrow \texttt{b(X)}, \texttt{not c1}., & \rho_2 : \texttt{c2} \leftarrow \texttt{d(X)}, \texttt{not c2}., \\ \rho_3 : \texttt{b(X)} \leftarrow \texttt{a(X)}, \texttt{not d(X)}., & \rho_4 : \texttt{d(X)} \leftarrow \texttt{e(X)}, \texttt{not b(X)}. \end{array} \right\}$$

et la base de faits $\mathcal{F}_{16} = \{\texttt{a(1)}, \texttt{e(1)}\}$. Nous avons ici deux ensembles de règles dangereuses $D_1 = \{(\rho_1, +), (\rho_3, +), (\rho_4, -)\}$ et $D_2 = \{(\rho_2, +), (\rho_4, +), (\rho_3, -)\}$. Si nous calculons les answer sets pour chacun des ensembles séparément nous obtenons un answer set $AS_1 = \{\texttt{a(1)}, \texttt{e(1)}, \texttt{d(1)}\}$ pour D_1 et un answer set $AS_2 = \{\texttt{a(1)}, \texttt{e(1)}, \texttt{b(1)}\}$ pour D_2. Par contre, si nous calculons les answer sets de P_{16}, qui correspond à l'union des règles dangereuses (sans marques) des ensembles D_1 et D_2, le programme ne possède pas d'answer set. En effet, les règles ρ_3 et ρ_4 appartiennent toutes deux à la fois à D_1 et D_2, l'application de ρ_3 permet d'obtenir AS_2 mais empêche AS_2 d'être un answer set de D_1 tandis que l'application de ρ_4 permet d'obtenir AS_1 mais empêche AS_1 d'être un answer set de D_2.

L'algorithme 2 réalise la fonction $unionEnsembleRèglesDangereuses$ qui est telle que $unionEnsembleRèglesDangereuses(\mathcal{R})$ calcule l'ensemble des ensembles unifiés de règles dangereuses qui correspond aux unions des ensembles de règles dangereuses d'un ensemble de règles $\mathcal{R}$ qui possèdent au moins une règle en commun. Nous parcourons tous les ensembles de règles dangereuses en cherchant les ensembles ayant une règle en commun, nous faisons l'union de ces ensembles et nous retirons de $\mathcal{D}$ un des deux ensembles. Pour cela nous utilisons l'opérateur $unionRegle$ qui calcule l'union de deux ensembles de règles dangereuses, avec la particularité de ne garder que les règles dangereuses marquées positivement lorsqu'une règle apparaît positive dans un ensemble et négative dans le second.

THÉORÈME 17 (Inconsistance et règles dangereuses). — *Soit $P = (\mathcal{F}, \mathcal{R})$ un programme et Δ l'ensemble des ensembles unifiés de règles dangereuses de $\mathcal{R}$. Le programme P est inconsistant si et seulement si le programme $(\mathcal{F}, \bigcup_{D \in \Delta} D)$ est inconsistant.*

Algorithme 2 : $unionEnsembleRèglesDangereuses$

Data : Un ensemble $\mathcal{R}$ de règles
Result : L'ensemble des ensembles unifiés de règles dangereuses
$\Delta = ensembleRèglesDangereuses(\mathcal{R})$;
foreach $D_i \in \Delta$ **do**
  foreach $D_j \in \Delta \backslash D_i$ **do**
    if $règles(D_i) \cap règles(D_j) \neq \emptyset$ **then**
      $D_i := (D_i \; unionRegle \; D_j)$;
      $\Delta := \Delta \backslash D_j$;
    end
  end
end
return Δ;

Afin d'assurer la consistance nous réutilisons l'ensemble calculé par l'algorithme 2 et nous calculons pour l'union des ensembles de règles dangereuses un premier *answer set*. D'après le théorème 17, si l'algorithme calcule un premier *answer set* pour cet ensemble alors le programme est consistant. Si la consistance est vérifiée alors nous pouvons nous intéresser aux règles dont la requête dépend, sinon la réponse est `absurde`.

5. Dépendance de la requête

Le but étant de limiter le nombre de règles utilisées pour répondre à une requête, nous cherchons à isoler l'ensemble des règles suffisantes pour répondre à celle-ci. Nous cherchons dans un premier temps à isoler l'ensemble des règles dont dépend directement la requête et permettant de générer toute instance de l'atome réponse lorsqu'elles sont appliquées. Mais nous verrons que ces règles ne sont pas suffisantes pour obtenir une réponse sur le programme complet, et que nous devons étendre cet ensemble aux ensembles de règles dangereuses possédant une règle en commun avec les règles dont dépend la requête, car elles peuvent empêcher l'appartenance d'une instance de l'atome réponse à un *answer set* du programme complet. Une fois ces ensembles de règles isolés, nous avons réduit considérablement le nombre de règles de notre programme nécessaires pour répondre à notre requête. Nous rappelons qu'avec les ontologies il est fréquent d'avoir des données très peu connectées entre elles, il est donc particulièrement intéressant d'écarter les règles qui n'ont pas d'impact sur la réponse à notre requête.

Soit $(P?Q)$ le programme ASP $P = (\mathcal{F}, \mathcal{R})$ requêté par Q, nous notons $(\mathcal{R} \downarrow Q)$ l'ensemble des règles de $(P?Q)$ dont dépend Q auquel nous ajoutons Q. L'algorithme 3 construit l'ensemble de règles $(\mathcal{R} \downarrow Q)$ à partir de $\mathcal{R}$ et Q. L'algorithme initialise la variable $\mathcal{Q}$ stockant les règles dont dépend la requête à $\{Q\}$ puis parcourt cet ensemble en y ajoutant toutes les règles qui possèdent un symbole de prédicat

en tête appartenant au corps d'une règle de $\mathcal{Q}$. Nous utilisons pour cela la fonction $définition$ qui est telle que $définition(\mathtt{p}, \mathcal{R})$ calcule l'ensemble des règles de $\mathcal{R}$ ayant le symbole de prédicat p en tête.

Algorithme 3 : $(. \downarrow .)$

Data : Un ensemble de règles $\mathcal{R}$, une requête Q
Result : L'ensemble des règles $(\mathcal{R} \downarrow Q)$ de $\mathcal{R}$ dont dépend la requête Q
$\mathcal{Q}$:= $\{Q\}$;
foreach $r \in \mathcal{Q}$ **do**
  foreach $p \in corps(r)$ **do**
    $\mathcal{Q} := \mathcal{Q} \cup définition(p, \mathcal{R})$;
  end
end
return $\mathcal{Q}$;

EXEMPLE 18. — *Soit l'ensemble de règles de l'exemple 2 et sa requête* $Q_2 = (\mathtt{ans(X)} \leftarrow \mathtt{cA(X)}.)$. *Nous commençons par initialiser* $\mathcal{Q} = \{Q_2\}$, *nous calculons ensuite toutes les règles avec* cA *en tête, donc* $\mathcal{Q} = \{Q_2, r_3\}$. *Nous prenons ensuite les symboles de prédicat apparaissant dans le corps de* r_3, *c'est-à-dire* ad *et* dir. *Seul* ad *apparaît en tête d'une règle donc nous ajoutons la règle* r_1 *qui est concernée. Nous continuons avec les symboles de prédicat du corps de* r_1 *nous ajoutons ainsi* r_2. *Nous avons donc* $(\mathcal{R}_2 \downarrow Q_2) = \mathcal{Q} = \{Q_2, r_3, r_1, r_2\}$,

Maintenant que nous avons isolé l'ensemble des règles dont Q dépend, nous montrons que la réponse à une requête dans $(P?Q) = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ est aussi une réponse dans $(\mathcal{F}, (\mathcal{R} \downarrow Q))$ et réciproquement, lorsque nous considérons des programmes *super-consistants* (Alviano *et al.*, 2014), i.e. des programmes $P = (\mathcal{F}, \mathcal{R})$ tels que pour toute base de faits $\mathcal{F}'$, $(\mathcal{F}', \mathcal{R})$ est consistant.

Lorsque nous cherchons une réponse sur un programme super-consistant (Alviano *et al.*, 2014), il n'est pas nécessaire de vérifier l'existence d'un *answer set* sur les ensembles de règles dangereuses car le programme est défini pour avoir au moins un *answer set* quels que soient les faits initiaux.

THÉORÈME 19 (Réponse à un programme super-consistant). — *Soit* $P = (\mathcal{F}, \mathcal{R})$ *un programme ASP super-consistant et* Q *une requête sur* P. R *est une réponse à* Q *dans* P *si et seulement si* R *est une réponse à* Q *dans* $(\mathcal{F}, (\mathcal{R} \downarrow Q))$.

Le théorème 19 est une conséquence directe d'un théorème de (Alviano, Faber, 2011).

Nous illustrons dans l'exemple 20 que seules les règles de $(\mathcal{R} \downarrow Q)$ sont nécessaires pour répondre à une requête Q sur un programme super-consistant $P = (\mathcal{F}, \mathcal{R})$.

EXEMPLE 20. — *Soit le programme ASP super-consistant* $P_{20} = (\emptyset, \mathcal{R}_{20})$ *(i.e.* $\mathcal{R}_2$ *sans les contraintes) avec* $\mathcal{R}_{20} =$

$$\left\{\begin{array}{ll} r_1 : \texttt{ad(X)} \leftarrow \texttt{pU(X), not eC(X).}, & r_2 : \texttt{eC(X)} \leftarrow \texttt{pU(X), not ad(X).}, \\ r_3 : \texttt{cA(X)} \leftarrow \texttt{ad(X), dir(X).}, & r_4 : \texttt{mCP(X)} \leftarrow \texttt{eC(X), not etT(X).}, \\ r_5 : \texttt{etT(X)} \leftarrow \texttt{eC(X), et(X).}, & r_6 : \texttt{chr(X)} \leftarrow \texttt{eC(X).}, \\ r_9 : \texttt{docteur(X)} \leftarrow \texttt{mCP(X).} & \end{array}\right\}$$

et la requête $Q_2 = (\texttt{ans(X)} \leftarrow \texttt{cA(X).})$*. Nous avons, comme dans l'exemple 18,* $(\mathcal{R}_{20} \downarrow Q_2) = \{Q_2, r_1, r_2, r_3\}$*, l'ensemble des règles dont la requête dépend. Nous pouvons constater qu'il n'est pas possible de créer une instance de* ans *avec les règles* r_4*,* r_5*,* r_6 *et* r_9*, les symboles de prédicat en tête de ces règles n'apparaissant pas dans le corps des règles de l'ensemble* $(\mathcal{R}_{20} \downarrow Q_2)$*. Les règles* r_4*,* r_5*,* r_6 *et* r_9 *ne sont donc pas nécessaires pour trouver une réponse à* $(P_{20}?Q_2)$*.*

Nous savons maintenant que lorsque notre programme est super-consistant l'ensemble de règles $(\mathcal{R} \downarrow Q)$ dont la requête dépend est suffisant pour répondre à une requête sur $(P?Q)$. Nous élargissons maintenant cette propriété à tous les programmes consistants en prenant en compte les ensembles de règles dangereuses du programme.

Nous cherchons maintenant à déterminer quelles sont les règles dangereuses dont l'application a un impact sur la réponse à une requête. Pour cela nous nous intéressons aux règles dangereuses appartenant à l'ensemble $(\mathcal{R} \downarrow Q)$.

L'algorithme 4 réalise la fonction $D\mathcal{Q}$ qui est telle que $D\mathcal{Q}(\mathcal{R}, Q)$ calcule l'ensemble des règles dangereuses qui intersectent l'ensemble des règles de $\mathcal{R}$ dont la requête Q dépend. Pour chaque ensemble de Δ, l'ensemble des ensembles unifiés de règles dangereuses de $\mathcal{R}$ (voir l'algorithme 2), nous cherchons s'il existe une règle appartenant à $(\mathcal{R} \downarrow Q)$. Nous ajoutons ainsi tous les ensembles de règles de Δ qui intersectent $(\mathcal{R} \downarrow Q)$ à l'ensemble $D\mathcal{Q}(\mathcal{R}, Q)$.

Algorithme 4 : $D\mathcal{Q}$

Data : L'ensemble $\mathcal{R}$ des règles et la requête Q
Result : L'ensemble des règles dangereuses $D\mathcal{Q}(\mathcal{R}, Q)$ qui intersectent $(\mathcal{R} \downarrow Q)$
Δ=$unionEnsembleRèglesDangereuses(\mathcal{R})$;
$D\mathcal{Q} := \emptyset$;
foreach $D \in \Delta$ **do**
  if $règles(D) \cap (\mathcal{R} \downarrow Q) \neq \emptyset$ **then**
    $D\mathcal{Q} := (D\mathcal{Q}\ unionRegle\ D)$
  end
end
return $D\mathcal{Q}$;

Nous isolons pour le moment deux ensembles de règles d'un programme $(P?Q) = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ dont Q dépend, $(\mathcal{R} \downarrow Q)$ et $D\mathcal{Q}(\mathcal{R}, Q)$. Nous montrons maintenant qu'une réponse à Q dans $(P?Q)$ est aussi une réponse dans $(\mathcal{F}, (\mathcal{R} \downarrow Q) \cup règles(D\mathcal{Q}(\mathcal{R}, Q)))$ et réciproquement. Autrement dit, une réponse sur le programme original est aussi une réponse sur le programme composé seulement des règles dont la requête dépend et des règles dangereuses qui intersectent celles-ci et réciproquement.

THÉORÈME 21 (Réponse à un programme consistant). — *Soit P un programme ASP consistant. R est une réponse à Q dans P si et seulement si R est une réponse dans $(\mathcal{F}, (\mathcal{R} \downarrow Q) \cup règles(D\mathcal{Q}(\mathcal{R}, Q)))$.*

EXEMPLE 22. — *Soit le programme $P_{22} = (\mathcal{F}_{22}, \mathcal{R}_{22})$ composé de $\mathcal{R}_{22} = \mathcal{R}_2 \cup \{r_{13} : \texttt{travaildur(X)} \leftarrow \texttt{etT(X).}\}$ et $\mathcal{F}_{22} = \{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}\}$. Nous posons la requête Q_{22} : $(\texttt{ans} \leftarrow \texttt{travaildur(X).})$. Nous obtenons $(\mathcal{R}_{22} \downarrow Q_{22}) = \{Q_{22}, r_{13}, r_5, r_2, r_1\}$. Les ensembles de règles dangereuses sont $D_1 = \{(r_7, +), (r_5, +), (r_4, +), (r_2, +), (r_1, -)\}$ et $D_2 = \{(r_8, +)\}$ (voir exemple 13) et $D\mathcal{Q}(\mathcal{R}_{22}, Q_{22}) = D_1$ (D_2 n'intersecte pas avec les règles dont dépend la requête). $(P_{22}?Q_{22})$ est consistant car tous les ensembles de règles dangereuses possèdent un answer set : $\mathcal{F}_{22}$ pour $(\mathcal{F}_{22}, règles(D_2))$ et $\{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{ad(jean)}\}$ pour $(\mathcal{F}_{22}, règles(D\mathcal{Q}(\mathcal{R}_{22}, \mathcal{R}_{22}))) = (\mathcal{F}_{22}, règles(D_1))$.*

La réponse (sceptique et crédule) à Q_{22} est `faux` *pour le programme $(\mathcal{F}_{22}, (\mathcal{R}_{22} \downarrow Q_{22}) \cup règles(D\mathcal{Q}(\mathcal{R}_{22}, Q_{22})))$. L'application de la règle r_2 avec* $\texttt{pU(jean)}$ *ne permet pas d'obtenir un answer set car cela déclenche la règle r_5 puis la contrainte r_7, le seul answer set vient de l'application de r_1 avec* $\texttt{pU(jean)}$ *qui bloque ainsi la règle r_2.*
Si nous calculons les answer sets de $(\mathcal{F}_{22}, (\mathcal{R}_{22} \downarrow Q_{22}))$ nous avons

$AS'_1 = \{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{ad(jean)}\}$
$AS'_2 = \{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{eC(jean)}, \texttt{etT(jean)},$
$\texttt{travaildur(jean)}, \texttt{ans}\}$

La réponse crédule à la requête Q_{22} pour le programme $(\mathcal{R}_{22} \downarrow Q_{22})$ est donc `vrai` *puisque* ans *appartient à AS'_2. Nous avons donc deux réponses différentes pour les ensembles $(\mathcal{R}_{22} \downarrow Q_{22})$ et $(\mathcal{R}_{22} \downarrow Q_{22}) \cup D\mathcal{Q}(\mathcal{R}_{22}, Q_{22})$ sachant que la réponse sur le programme correspond à la réponse de $(\mathcal{R}_{22} \downarrow Q_{22}) \cup règles(D\mathcal{Q}(\mathcal{R}_{22}, Q_{22}))$. L'ensemble $(\mathcal{R}_{22} \downarrow Q_{22})$ possède un answer set de trop qui provient du fait que les règles contenant le cycle d'inconsistance ne font pas partie de cet ensemble. La contrainte r_7 n'est donc pas appliquée et AS'_2 ne mène pas un answer set du le programme complet. L'ensemble de règles $(\mathcal{R}_{22} \downarrow Q_{22})$ n'est donc pas suffisant pour trouver une réponse à Q_{22} dans P_{22} il faut l'étendre à $règles(D\mathcal{Q}(\mathcal{R}_{22}, Q_{22}))$ pour obtenir les mêmes réponses.*

Les règles nécessaires pour répondre à une requête sur un programme sont désormais isolées. Nous avons mis en avant que pour répondre à une requête sur un programme les règles dont la requête dépend ne sont pas suffisantes et qu'il faut ajouter à celles-ci les ensembles de règles dangereuses ayant une règle en commun. Notons qu'il n'est par contre souvent pas nécessaire de calculer des *answer sets* complets sur cet ensemble. Nous présenterons des perspectives permettant de réduire le nombre d'instanciations nécessaire pour répondre à une requête. Avant cela, nous présentons dans la section suivante l'implémentation actuelle de l'interrogation en ASP que nous avons réalisée.

6. Implémentation et expérimentations

Pour mettre en avant l'intérêt d'utiliser l'interrogation présentée dans cet article, nous avons mis en place des protocoles pour comparer les différentes approches sur des benchmarks avec ou sans négation par défaut.

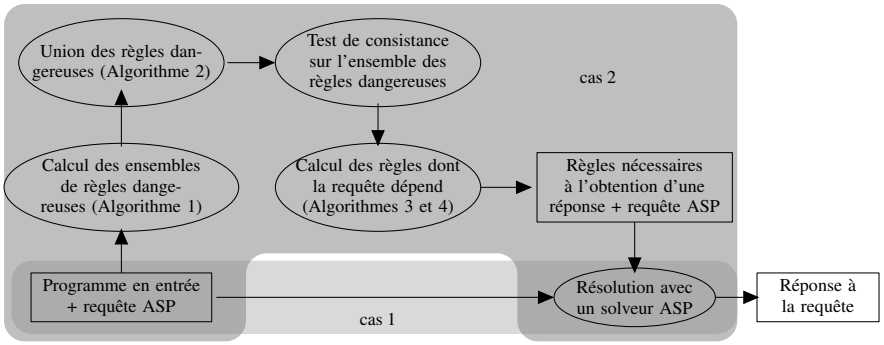

Figure 3. Processus d'interrogation

La figure 3 présente le processus mis en place pour l'interrogation en ASP. Il y a deux déroulements possibles : une interrogation basique (cas 1) qui utilise le solveur directement sur le programme initial et une interrogation intégrant l'isolation des règles vues dans les sections précédentes (cas 2).

L'implémentation de l'isolation des règles [16] permet de l'utiliser en prétraitement de tout solveur ASP. Nous avons mis en parallèle différentes approches pour l'interrogation : l'utilisation du solveur ASP `Clingo` sans isolation des règles que nous appellerons `Clingo`; l'approche d'isolation des règles présentée dans cet article avec le solveur `Clingo` appelée `Clingo`+Q; l'interrogation proposée par le solveur `DLV` (Faber *et al.*, 2007) appelée `DLV`+M; l'interrogation à l'aide d'un raisonneur pour ontologies classiques (sans négation par défaut) avec `Pellet`; et l'utilisation d'un raisonneur pour ontologies avec défauts `NoHR` (Costa *et al.*, 2015), fonctionnant comme plug-in sur `protégé` (Musen, 2015).

`NoHR` utilise la sémantique bien fondée (Van Gelder *et al.*, 1991). Avec cette sémantique, les programmes admettent un et un seul modèle.

`DLV`+M utilise les `magic sets` afin d'optimiser l'interrogation (Alviano, Faber, 2011 ; Faber *et al.*, 2007). Le principe des `magic sets` est de simuler la marche arrière dans un système fonctionnant en marche avant afin de limiter les instanciations et donc le temps de calcul. Cela nécessite une ré-écriture du programme en partant de la

16. L'implémentation est disponible sur http://forge.info.univ-angers.fr/~fgarreau/ASPeRiX-Q.php

requête. Ensuite, le calcul des réponses est effectué en marche avant sur le programme ré-écrit.
La méthode proposée dans (Faber *et al.*, 2007) ne fonctionne que si le programme initial est consistant. En théorie, les règles utilisées pour la ré-écriture semblent être les mêmes que celles considérées dans le présent article : les règles dont dépend la requête ainsi que les règles dangereuses qui intersectent celles-ci. La ré-écriture est assez lourde mais permet de rendre le calcul plus efficace si la requête contient des constantes.
Toutefois, seule une version simplifiée de la méthode a été implémentée (Alviano, Faber, 2011). `DLV+M` n'implémente les `magic sets` que sur les règles dont dépend la requête. Le système n'est donc juste que si le programme initial est super-consistant.

Ces différentes approches ont été comparées sur plusieurs versions de l'ontologie `university` présentée plus en détail en annexe B. Les ontologies sont souvent représentées en OWL mais ces représentations peuvent être ramenées à une représentation logique (logique de description (Calvanese *et al.*, 2007) ou règles existentielles (Baget *et al.*, 2011)) dans laquelle les informations sont représentées par deux ensembles : un ensemble de faits décrivant les objets à représenter (ABox en logique de description) et un ensemble de règles représentant les liens entre ces objets (TBox en logique de description). Nous utilisons par la suite les termes ensemble de faits et ensemble de règles car ces termes sont plus généraux et peuvent être utilisés à la fois pour les ontologies et les programmes ASP. L'ontologie `university` est composée d'un ensemble de règles qui décrit les différents concepts au sein des universités, d'un générateur de faits qui génère les faits du nombre d'universités voulu et d'un jeu de quatorze requêtes conjonctives non-booléennes. Elle ne possède pas de négation par défaut. La version classique de cette ontologie existe sous deux formats, la version originale `LUBM` au format OWL DL et la version plus légère mais moins complète correspondant aux formats OWL2 $\mathcal{E}$L et $\mathcal{Q}$L, appelée ici `GRAAL`. Nous traitons la version `GRAAL` couramment utilisée pour comparer les raisonneurs et solveurs (Costa *et al.*, 2015 ; Leone *et al.*, 2012) car moins complexe. Notons que `NoHR` ne traite que la version `GRAAL` de l'ontologie. Elle peut être obtenue directement au format OWL sur le site du projet `GRAAL` (*Benchmark university GRAAL*, 2014) ou encore au format ASP sur le site de `DLV` (*Benchmark university DLV*, 2012). La version au format ASP peut aussi être obtenue à partir de deux outils de traduction, le premier permettant de traduire une ontologie du format OWL vers les règles existentielles (*OWL2DLGP GRAAL*, 2015), et le second de traduire des règles existentielles en ASP (Garreau *et al.*, 2015).

Nous proposons ensuite de tester les différentes approches sur des versions de l'ontologie avec négation par défaut que nous décrirons dans la section 6.2. Le raisonneur `Pellet` ne sera pas testé sur ces versions car celui-ci ne traite que les ontologies classiques sans négation par défaut.

Les tests ont été effectués sur plusieurs ensembles de faits correspondant à une, deux et dix universités mais nous ne présentons en détail dans l'article que les tests sur dix universités, les tests sur une et deux universités ne mettant pas assez en valeur les

différences entre les approches. Les tests sur plus d'universités nécessitent l'utilisation de trop de mémoire sur la machine test par les solveurs et les raisonneurs. L'ensemble des résultats est disponible en annexe C.

L'ensemble des tests ont été réalisés sur une machine avec la configuration suivante : Xubuntu 16.04 64 bits, Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz, 2x8GiB à 1600MHz de mémoire vive, disque dur 7200tr/min. Les résultats correspondent à la moyenne de vingt itérations, la variance étant quasiment nulle.

6.1. Benchmark `university` classique

Pour comparer les approches, différents temps sont à prendre en compte. Dans le cas des raisonneurs sur les ontologies il faut tout d'abord charger l'ontologie, puis démarrer le raisonneur sur cette ontologie et enfin effectuer l'interrogation, il est ainsi possible d'effectuer plusieurs requêtes rapidement une fois l'ontologie préchargée. Le problème est qu'il est nécessaire de précharger l'ontologie à chaque changement dans celle-ci, nous prenons donc en compte le temps de préchargement dans le calcul de la réponse pour chaque requête individuelle puis pour l'ensemble des requêtes (les requêtes $q1$ à $q14$ sont présentées en annexe B). Dans le cas des solveurs ASP (`Clingo` et `DLV`) les temps de chargement sont intégrés directement à l'interrogation, il n'est pas possible de précharger l'ontologie et d'interroger le programme préchargé sauf en calculant entièrement les *answer sets* de l'ontologie.

Le tableau 1 présente les temps de réponse pour dix universités sur la version `GRAAL` sans négation par défaut. Les temps correspondent au temps total pour répondre à une requête. Les temps entre parenthèses correspondent à la préparation pour l'interrogation, ce qui représente pour les raisonneurs `Pellet` et `NoHR` le temps nécessaire pour charger l'ontologie et pour `Clingo`+Q le temps nécessaire pour isoler les règles comme proposé dans l'article.

Tableau 1. Temps de réponse en secondes pour dix universités sur la version `GRAAL`

	q1	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**
`Clingo`	12,0	12,0	12,0	12,0	12,0	12,0	12,0
`Clingo`+Q	5,6 (1,3)	4,9 (1,2)	3,2 (0,9)	7,2 (1,5)	7,4 (1,5)	2,1 (0,9)	5,4 (1,4)
`DLV`+M	5,6	6,2	5,5	5,8	6,0	6,0	6,3
`Pellet`	-	39,1 (36,8)	37,2 (36,8)	37,8 (36,8)	37,9 (36,8)	36,9 (36,8)	37,6 (36,8)
`NoHR`	93 (93)	94,2 (93)	93 (93)	93 (93)	96,4 (93)	103,3 (93)	94,2 (93)
	q8	**q9**	**q10**	**q11**	**q12**	**q13**	**q14**
`Clingo`	12,0	12,0	12,0	12,0	12,0	12,0	12,0
`Clingo`+Q	5,3 (0,9)	5,6 (1,5)	5,5 (1,3)	1,1 (0,9)	1,8 (0,9)	7,3 (1,5)	1,9 (1,3)
`DLV`+M	5,7	6,1	5,6	5,4	7,7	5,7	5,4
`Pellet`	37,8 (36,8)	37,8 (36,8)	37,1 (36,8)	681,8(36,8)	37 (36,8)	37 (36,8)	36,9 (36,8)
`NoHR`	103,5 (93)	101,2 (93)	93 (93)	93 (93)	93 (93)	93 (93)	109,8 (93)

Le tableau 2 présente les temps nécessaires pour répondre à l'ensemble des requêtes, qui sont données en annexe B, en chargeant l'ontologie un minimum de fois pour chacune des approches. Ainsi le temps total nécessaire pour répondre à l'ensemble des requêtes dans le cas de `Pellet` et `NoHR` ne comptabilise qu'une seule fois le temps de chargement de l'ontologie. Pour `Clingo` et `Clingo`+Q, il est pos-

sible de calculer l'ensemble des réponses en une seule fois en incluant l'ensemble des requêtes dans le programme. DLV+M ne permet pas quant à lui de procéder à plusieurs requêtes dans un seul calcul. Il est donc nécessaire d'ajouter tous les temps de calcul pour le comparer.

Tableau 2. Temps de réponse en secondes pour dix universités sur l'ensemble des requêtes avec la version GRAAL

	Clingo	Clingo+Q	DLV+M	Pellet	NoHR
q_{tot}	12	13,7 (3,5)	83	690,3 (36,8)	144,6 (93)

Le tableau 3 présente la taille du programme instancié après isolation des règles. Les valeurs pour $q1$ à $q14$ correspondent à la taille de l'instanciation pour chaque requête respectivement, $qtot$ correspond à celle des requêtes traitées simultanément. La valeur tot est la taille du programme instancié sans isolation. L'instanciation est réalisée par Gringo. Notons que pour un programme sans négation, l'instanciation intelligente opère toutes les déductions, le résultat est donc un ensemble de faits.

Tableau 3. Taille après instanciation du programme isolé pour dix universités sur la version GRAAL

q1	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**	**q8**
570 258	488 158	213 400	516 273	892 247	165 014	757 235	600 978
q9	**q10**	**q11**	**q12**	**q13**	**q14**	**qtot**	**tot**
796 490	570 258	5 843	103 051	892 247	75 547	1 939 965	2 086 004

Nous constatons, dans le tableau 1, des temps raisonnables avec les solveurs ASP pour l'obtention d'une réponse sur l'ensemble des requêtes du benchmark avec dix universités. Les deux premières lignes du tableau permettent de comparer l'approche sans isolation avec l'approche proposée dans cet article en utilisant le solveur ASP Clingo. L'ajout de l'isolation des règles améliore le temps de calcul en limitant les règles nécessaires pour répondre à une requête. Sur l'ensemble des requêtes individuelles l'amélioration est d'au moins 40 % et jusqu'à 90 %. L'isolation des règles étant encore à l'état de prototype, ces résultats peuvent être optimisés. Les requêtes bénéficiant le plus de l'isolation des règles correspondent aux requêtes contenant peu d'atomes, comme la requête $q14$ avec 12 secondes nécessaires pour le solveur Clingo seul et seulement 0,6 secondes pour Clingo après isolation (l'isolation nécessitant ici 1,3 secondes). Le temps de calcul est corrélé à la taille du programme (isolé) instancié. Les requêtes ayant le temps de calcul le plus rapide ($q6$, $q11$, $q12$ et $q14$) sont celles correspondant aux programmes instanciés les plus réduits. Le tableau 3 montre l'intérêt de l'isolation qui réduit de manière importante la taille du programme traité.

Si nous comparons maintenant Clingo+Q avec les autres approches, nous constatons des temps raisonnables pour DLV+M avec en moyenne 6 secondes nécessaires pour chaque requête ce qui reste dans le même ordre de grandeur que Clingo+Q mais avec des temps variant moins d'une requête à l'autre. Pour Pellet et NoHR nous avons respectivement des temps autour de 37 et 93 secondes. L'écart de temps peut paraître grand lorsque l'on intègre le temps de chargement à chaque requête, mais

il se resserre si l'on prend en compte seulement le temps de réponse à une requête qui sera de l'ordre de la seconde pour la plupart des requêtes. Néanmoins nous constatons pour `Pellet` que la réponse à la requête $q1$ n'est pas correcte (la réponse contient 3 instances au lieu de 4), ainsi qu'un temps très long pour la requête $q11$ qui demande 645 secondes. Pour `NoHR`, nous constatons des temps qui dépassent les 8 secondes pour les requêtes $q6$, $q8$, $q9$ et $q14$. Les requêtes demandant plus de temps correspondent principalement aux requêtes ne contenant pas de constantes, les raisonneurs étant plus efficaces lorsque la requête contient des constantes car leur résolution est basée sur un chaînage arrière ce qui réduit grandement l'espace de recherche. La comparaison uniquement du temps de réponse à une requête sans chargement correspond plus à l'utilisation de l'interrogation sur les ontologies où une ontologie n'est chargée qu'une seule fois puis toutes les requêtes sont testées.

Dans le tableau 2, nous avons 144,6 secondes pour `NoHR` et 690,3 secondes pour `Pellet` dont 36,8 secondes de chargement de l'ontologie, ce temps est principalement impacté par la requête $q11$ qui à elle seule nécessite 645 secondes. Le fait que `DLV` ne permette pas de lancer plusieurs requêtes en une seule itération donne un désavantage au solveur `DLV`. Pour `Clingo` le temps de calcul est le même que pour chaque requête prise isolément et correspond au temps de calcul de l'*answer set*. Pour `Clingo`+Q l'isolation des règles ne permet pas de gagner du temps car le programme isolé est quasiment identique au programme original donc l'amélioration pour répondre à la requête est quasi-nulle et ne compense pas le temps nécessaire pour isoler les règles du programme.

Pour conclure sur ces tableaux, ces premiers résultats nous indiquent que le traitement de l'interrogation d'une ontologie aux formats OWL2 $\mathcal{EL}$ et $\mathcal{QL}$ avec un solveur ASP est comparable aux approches classiques. Les résultats obtenus avec `Clingo` après isolation des règles et ceux obtenus avec `DLV`+M ne permettent pas de déterminer si une méthode est plus efficace que l'autre, la distance entre les résultats aux requêtes n'étant pas significative sauf lorsque l'on souhaite exécuter plusieurs requêtes à la fois, `DLV`+M ne permettant pas de le faire. Nous pouvons constater que les raisonneurs sont efficaces après avoir chargé l'ontologie et seulement sur des requêtes spécifiques pour `NoHR`. Ils sont plus efficaces lorsque la requête contient des constantes car leur résolution est basée sur un chaînage arrière alors que notre version actuelle de l'isolation est effectuée sur les règles non instanciées et ne tire pas profit des constantes. Cette particularité est l'objet de perspectives d'optimisation pour notre isolation (section 7). Nous souhaitons surtout mettre en avant la possibilité des solveurs ASP de traiter des ontologies plus expressives que les raisonneurs classiques, notamment avec l'ajout d'exceptions, ce qui est l'objet de la section suivante.

6.2. Benchmark `university` avec règles par défaut

Nous proposons maintenant l'ajout de règles par défaut au benchmark `university` afin de tester la démarche présentée dans cet article. En effet, il n'existe pas de benchmark avec l'expressivité souhaitée pour tester notre isolation, il est donc

nécessaire de modifier une ontologie existante. Les tests provenant directement de l'ontologie originale et d'autres articles sont présentés en anglais pour ne pas dénaturer l'ontologie tandis que les exemples provenant de cet article sont présentés en français.

L'ajout de règles par défaut ne permet plus l'utilisation de raisonneurs classiques ce qui rend impossible l'interrogation avec `Pellet`. Il existe différentes classes de programmes avec négation par défaut. La première concerne les programmes stratifiés qui sont ceux ne comprenant pas de cycle impliquant une dépendance négative. Ces programmes ne comportent qu'un seul *answer set* et ne peuvent pas être inconsistants. Le raisonneur `NoHR` a été développé dans le contexte de la sémantique bien fondée qui ne coïncide avec la sémantique des modèles stables que dans le cas de programmes stratifiés. La deuxième classe concerne les programmes super-consistants, qui ont au moins un *answer set* quelque soit la base de faits. Enfin les programmes possiblement inconsistants peuvent avoir aucun, un ou plusieurs *answer sets* et nécessitent un test de consistance pour s'assurer que le résultat d'une requête est correct. Il n'existe cependant pas de benchmark super-consistant ni possiblement consistant pour tester notre approche. Les benchmarks avec défauts présentés dans cet article concernent l'expressivité plus que la performance pour laquelle il serait nécessaire d'élaborer des benchmarks plus complexes dans la représentation des défauts. Cela représente une perspective de recherche nécessaire car peu explorée, les études récentes se portant principalement sur les programmes stratifiés. Seuls les solveurs ASP permettent de traiter les autres programmes. Pour ces raisons, les résultats des tests suivants n'ont pas d'intérêt à être présentés dans un tableau pour être comparés en terme de performance et sont donc présentés directement dans le texte.

Considérons le programme stratifié proposé dans (Costa *et al.*, 2015) avec l'ensemble de règles suivant ajouté au benchmark `university` :

```
replacement(X,Y) :-
   professor(X), worksFor(X,Y), lowTeachingLoad(X),
   not onSabbatical(X), not ill(X).
ans(X,Y) :- replacement(X,Y).
```

Les prédicats `onSabbatical` et `ill` n'apparaissent pas dans l'ontologie et le nombre de règles dont la requête dépend est très réduit. Le temps de réponse pour la requête est donc instantané avec `Clingo`+Q, de même pour `NoHR` et `DLV`+M. Notons que l'interrogation dans un programme stratifié se traite de la même manière que dans un programme sans négation par défaut.

Nous testons maintenant des programmes non-stratifiés (traités uniquement par les solveurs ASP).

Nous proposons d'abord un programme super-consistant obtenu en ajoutant des règles par défaut au benchmark `university` avec une seule université :

```
takesCourse(X,Y) :- student(X).
```

```
graduateCourse(Y) :-
  graduateStudent(X), takesCourse(X,Y),
   not undergraduateCourse(Y).
undergraduateCourse(Y) :-
   undergraduateStudent(X), takesCourse(X,Y),
   not graduateCourse(Y).
graduateStudent(X) :- student(X), degreeFrom(X,Y).
```

avec la requête $(\texttt{ans}(\texttt{X}) \leftarrow \texttt{graduateCourse}(\texttt{X}).)$ pour calculer l'ensemble des `graduateCourse`.

Les règles ajoutées permettent l'obtention de plusieurs *answer sets*, lorsque l'information qu'un cours est `graduateCourse` ou `undergraduateCourse` est inconnue et qu'il existe un étudiant dont on sait qu'il est à la fois `graduateStudent` et `undergraduateStudent`. Dans ce cas pour chaque étudiant étant à la fois `graduateStudent` et `undergraduateStudent` le nombre d'*answer sets* est doublé. Cet exemple est non-stratifié à cause du cycle sur les prédicats `graduateCourse` et `undergraduateCourse`. Les résultats pour `Clingo`+Q et `DLV`+M sont comparables. Le temps de réponse à une requête dépend principalement du nombre d'*answer sets* qui peut croître exponentiellement.

Nous avons ensuite utilisé le programme possiblement inconsistant suivant, en ajoutant un autre ensemble de règles par défaut au benchmark `university` avec dix universités. Le programme possède au plus un *answer set* :

```
nonTeacher(X) :- student(X), not hasMaster(X).
:- teacher(X), nonTeacher(X).
teacher(X) :- teacherOf(X,Y).
hasMaster(X) :- student(X), teacher(X).
hasMaster(X) :- mastersdegreeFrom(X,Y).
```

avec la requête $(\texttt{ans}(\texttt{X}) \leftarrow \texttt{nonTeacher}(\texttt{X}).)$ permettant de calculer l'ensemble des individus n'étant pas enseignants.

L'intérêt est de pouvoir déduire du manque d'information sur les diplômes d'un étudiant s'il est possible qu'il soit enseignant ou non. Ici, c'est la contrainte disant qu'une personne ne peut pas être à la fois enseignant et non enseignant qui rend ce programme non-stratifié. Dans cet exemple, selon la base de faits, soit on obtient un seul *answer set*, soit le programme est inconsistant. C'est pourquoi il est nécessaire de tester la consistance du programme. Les temps nécessaires pour l'obtention d'une réponse pour cet exemple de programme non-stratifié restent du même ordre de grandeur que pour les interrogations sur les programmes sans négation par défaut.

`DLV`+M ne mettant pas en œuvre de tests de consistance complet, la justesse de l'interrogation n'est garantie que pour des programmes super-consistants. Le solveur propose d'utiliser les `magic sets` lorsqu'il détecte que le programme est super-consistant et une interrogation sans optimisation lorsque le pro-

gramme ne l'est pas pour assurer la correction de la réponse. Il semble toutefois que le test de super-consistance ne consiste qu'à vérifier l'absence de contraintes (cycles impairs de taille un). Par exemple, lorsque nous considérons l'exemple 22, le solveur DLV+M obtient les bonnes réponses car il n'utilise pas les magic sets à cause de la contrainte. Cependant, si nous remplaçons la contrainte (c1 ← mCP(X), etT(X), not c1.) par les trois règles dangereuses équivalentes, (d1 ← mCP(X), etT(X), not d3.), (d2 ← mCP(X), etT(X), not d1.) et (d3 ← mCP(X), etT(X), not d2.), DLV+M obtient la réponse vrai alors que celle-ci devrait être faux. Dans ce cas, DLV+M utilise les magic sets alors que le programme n'est pas super-consistant. Un avertissement prévenant que les réponses peuvent ne pas être correctes si le programme n'est pas super-consistant est tout de même affiché avant l'affichage des réponses.

En comparaison avec DLV+M, la méthode proposée dans la présent article est juste et complète. Elle inclut un test de consistance initial. L'isolation des règles est certainement plus rapide que la ré-écriture du programme. Par contre, l'instanciation du programme résultant est plus lourde et le calcul des réponses risque donc d'être moins efficace si la requête contient des constantes.

Le tableau 4 récapitule les classes de programmes pouvant être interrogés par les solveurs et raisonneurs testés.

Tableau 4. Expressivités tolérées par les outils d'interrogation testés

	Pellet	NoHR	DLV+M	Clingo+Q
programmes sans défaut	✓	✓	✓	✓
programmes stratifiés	x	✓	✓	✓
programmes super-consistants	x	x	✓	✓
programmes sans restriction	x	x	x	✓

D'après ce tableau et les résultats des tests réalisés sur university, nous pouvons conclure que Clingo+Q offre la plus aboutie des implémentations actuelles du point de vue de l'expressivité des ontologies traitées, tout en offrant des résultats corrects et efficaces sur les ontologies classiques.

7. Perspectives d'optimisation

Pour améliorer l'efficacité de l'interrogation nous proposons des pistes pour minimiser l'instanciation des règles nécessaires pour répondre à une requête. Nous avons précédemment isolé les parties du programme suffisantes pour répondre à une requête. Maintenant, nous souhaitons instancier le moins d'atomes possible parmi les règles isolées avant de calculer les réponses à la requête. Pour cela nous présentons trois axes d'amélioration. Le premier axe porte sur l'ensemble $(\mathcal{R} \downarrow Q)$: pour les requêtes dont le corps contient au moins une constante, nous allons prendre en compte cette information en répercutant celle-ci sur les règles du programme afin de réduire le nombre d'instanciations de certains atomes. Dans le deuxième axe nous souhaitons réduire les instanciations sur l'ensemble $règles(DQ(\mathcal{R}, Q))$. Les constantes présentes dans la

requête ne sont pas suffisantes pour instancier les règles qui composent cet ensemble ce qui implique d'utiliser d'autres éléments. Enfin, la troisième perspective concerne le test de consistance initial. Nous utiliserons les particularités des règles dangereuses négatives pour réduire les instances nécessaires à ce calcul. Les deux premiers axes d'amélioration apparaissent comme une étape intermédiaire dans l'interrogation après l'isolation des règles nécessaires et avant d'effectuer le calcul des *answer sets* pour obtenir les réponses à la requête.

7.1. Instanciation des règles dont la requête dépend

Lors d'une interrogation, les arguments des atomes de la requête peuvent être des variables ou des constantes. Lorsqu'il y a des variables dans une requête, les réponses fournies seront des ensembles de substitutions possibles pour ces variables tandis que les constantes agiront comme une restriction en imposant une substitution à l'argument. Une première amélioration possible consiste à utiliser les constantes présentes dans la requête pour limiter le nombre d'instanciations effectuées. Il est possible d'utiliser un algorithme de marche arrière pour calculer les instances nécessaires pour les règles dont la requête dépend. Nous ne présentons ici qu'une intuition de l'algorithme, celui-ci n'étant pas encore implémenté. Dans un premier temps, on utilise les constantes présentes dans le corps de la requête et on instancie les règles avec les constantes selon leur position dans les atomes en tête. Prenons par exemple la requête $(\texttt{ans(X)} \leftarrow \texttt{p}(1, \texttt{X}).)$ et la règle $(\texttt{p(X, Y)} \leftarrow \texttt{a(X, Y)}, \texttt{ not q(X, Y)}.)$ on peut instancier partiellement la règle en remplaçant X par 1 pour obtenir $(\texttt{p}(1, \texttt{X}) \leftarrow \texttt{a}(1, \texttt{X}), \texttt{ not q}(1, \texttt{X}).)$. Dans un second temps, on instancie récursivement l'ensemble des règles dont la requête dépend en utilisant les constantes présentes dans le corps des règles instanciées. Une marche arrière suivie de l'interrogation présentée dans la partie 3 suffisent à garantir une réponse correcte si le programme ne contient pas de règle dangereuse.

Cette étape correspond à l'utilisation des `magic sets`, faite dans DLV+M (Alviano, Faber, 2011). Notre objectif est de l'adapter à l'isolation que nous proposons sans les inconvénients d'une réécriture complète des règles comme réalisée avec les `magic sets`.

EXEMPLE 23. — *Soit le programme* $P_{23} = (\mathcal{F}_{23}, \mathcal{R}_{23})$ *composé de :* $\mathcal{R}_{23}$ =

$$\left\{ \begin{array}{ll} \rho_1 : \texttt{p(X, Y)} \leftarrow \texttt{a(X, Y)}, \texttt{ not q(X, Y)}., & \rho_2 : \texttt{q(X, Y)} \leftarrow \texttt{a(X, Y)}, \texttt{ not p(X, Y)}., \\ \rho_3 : \texttt{e(X)} \leftarrow \texttt{d(X)}., & \rho_4 : \texttt{c1} \leftarrow \texttt{s(X, Y)}, \texttt{ e(Y)}, \texttt{ not c1}., \\ \rho_5 : \texttt{s(X, Y)} \leftarrow \texttt{p(X, Y)}. & \end{array} \right\}$$

et de la base de faits $\mathcal{F}_{23} = \{\texttt{a}(1, 1), \texttt{a}(3, 3), \texttt{a}(1, 2), \texttt{d}(2), \texttt{d}(4)\}$ *et la requête* $Q_{23} = (\texttt{ans(X)} \leftarrow \texttt{p}(1, \texttt{X}).)$. *Nous avons* $(\mathcal{R}_{23} \downarrow Q_{23}) = \{Q_{23}, \rho_1, \rho_2\}$. *Si nous instancions partiellement les règles de* $(\mathcal{R}_{23} \downarrow Q_{23})$ *avec une technique de marche arrière et en utilisant les constantes de la requête, on obtient :* $\mathcal{R}'_{23}$ =

$$\left\{ \begin{array}{ll} Q_{23} : \texttt{ans(X)} \leftarrow \texttt{p}(1, \texttt{X}). & \\ \rho'_1 : \texttt{p}(1, \texttt{X}) \leftarrow \texttt{a}(1, \texttt{X}), \texttt{ not q}(1, \texttt{X}). & \rho'_2 : \texttt{q}(1, \texttt{X}) \leftarrow \texttt{a}(1, \texttt{X}), \texttt{ not p}(1, \texttt{X}). \end{array} \right\}$$

Une fois les règles instanciées par la marche arrière, l'application de ρ'_1 avec $(\{[\mathtt{X} \mapsto \mathtt{1}], [\mathtt{X} \mapsto \mathtt{2}]\})$ suffit alors pour répondre à la requête. La marche arrière permet d'éviter les instanciations non nécessaires. Cette étape n'est cependant pas suffisante dès lors que des règles dangereuses sont contenues dans l'ensemble $(\mathcal{R} \downarrow Q)$. Dans cet exemple, en considérant $\mathcal{R}'_{23}$, nous avons quatre answer sets :
$AS_1 : \mathcal{F}_{23} \cup \{\mathtt{p}(\mathtt{1},\mathtt{1}), \mathtt{p}(\mathtt{1},\mathtt{2}), \mathtt{ans}(\mathtt{1}), \mathtt{ans}(\mathtt{2})\}$
$AS_2 : \mathcal{F}_{23} \cup \{\mathtt{p}(\mathtt{1},\mathtt{1}), \mathtt{q}(\mathtt{1},\mathtt{2}), \mathtt{ans}(\mathtt{1})\}$
$AS_3 : \mathcal{F}_{23} \cup \{\mathtt{q}(\mathtt{1},\mathtt{1}), \mathtt{p}(\mathtt{1},\mathtt{2}), \mathtt{ans}(\mathtt{2})\}$
$AS_4 : \mathcal{F}_{23} \cup \{\mathtt{q}(\mathtt{1},\mathtt{1}), \mathtt{q}(\mathtt{1},\mathtt{2})\}$

La réponse crédule obtenue est donc $(\{[\mathtt{X} \mapsto \mathtt{1}], [\mathtt{X} \mapsto \mathtt{2}]\})$ alors que la réponse à la même interrogation sur le programme complet est $(\{[\mathtt{X} \mapsto \mathtt{1}]\})$. En effet, $p(1,2)$ ne peut pas être solution car, conjugué au fait $\mathtt{d}(\mathtt{2})$, les règles ρ_5 et ρ_3 peuvent être appliquées, ce qui viole la contrainte ρ_4. AS_1 et AS_3 ne peuvent donc pas être étendus à des answer sets du programme complet. La réponse n'est donc pas correcte, il reste à traiter les règles dangereuses à l'intersection $règles(DQ(\mathcal{R}_{23}, Q_{23})) = \{\rho_1, \rho_2, \rho_3, \rho_4\}$, ce qui fait l'objet de la section suivante.

7.2. Instanciation des règles dangereuses

Le second point concerne l'instanciation des règles dangereuses qui intersectent $(\mathcal{R} \downarrow Q)$. Nous souhaitons identifier les instances nécessaires pour s'assurer qu'une réponse est correcte et ne conserver que celles-ci. Nous savons que notre programme est consistant car le test de consistance a été effectué au préalable. L'idée est d'utiliser les atomes des *answer sets* obtenus lors de l'instanciation des règles dont la requête dépend faite précédemment afin d'instancier partiellement, en marche avant, les règles des cycles d'inconsistance (dans le cas de l'exemple précédent nous cherchons à instancier la contrainte ρ_4 à l'aide des atomes des *answer sets* trouvés à partir de $\mathcal{R}'_{23}$). Une fois l'instanciation effectuée il faut tester si les règles instanciées vont être appliquées, pour cela il faut de nouveau effectuer une marche arrière pour déterminer quelles sont les instances nécessaires pour les règles dangereuses encore non instanciées.

EXEMPLE 24. — *Reprenons le programme $P_{23} = (\mathcal{F}_{23}, \mathcal{R}_{23})$ avec la requête $Q_{23} = (\mathtt{ans}(\mathtt{X}) \leftarrow \mathtt{p}(\mathtt{1},\mathtt{X}).)$. Nous avons $(\mathcal{R}_{23} \downarrow Q_{23}) = \{Q_{23}, \rho_1, \rho_2\}$, $DQ(\mathcal{R}_{23}, Q_{23}) = \{(\rho_1, +), (\rho_2, -), (\rho_3, +), (\rho_4, +), (\rho_5, +)\}$. Si nous considérons l'ensemble de règles $\mathcal{R}'_{23}$, la réponse à la requête Q_{23} n'est pas correcte car nous obtenons $(\{[\mathtt{X} \mapsto \mathtt{1}], [\mathtt{X} \mapsto \mathtt{2}]\})$ alors que la réponse attendue sur le programme complet est $\{[\mathtt{X} \mapsto \mathtt{1}]\}$. Nous souhaitons donc vérifier que les instances trouvées lors de l'instanciation des règles dont la requête dépend font bien partie d'un answer set. Nous allons dans un premier temps instancier en marche avant les règles permettant possiblement l'application de ρ_4 à l'aide des atomes faisant partie des answer sets AS_1, AS_2, AS_3 et AS_4. Nous pouvons donc instancier, selon les answer sets, ρ_5 puis ρ_4 ainsi :*

$$\left\{ \begin{array}{ll} \rho'_5 : \mathtt{s}(1,1) \leftarrow \mathtt{p}(1,1). & \rho''_5 : \mathtt{s}(1,2) \leftarrow \mathtt{p}(1,2). \\ \rho'_4 : \mathtt{c1} \leftarrow \mathtt{s}(1,1), \mathtt{e}(1), \mathtt{not\ c1}. & \rho''_4 : \mathtt{c1} \leftarrow \mathtt{s}(1,2), \mathtt{e}(2), \mathtt{not\ c1}. \end{array} \right\}$$

Nous pouvons maintenant vérifier si l'une des contraintes va être appliquée en utilisant une marche arrière, ce qui revient à ajouter les règles instanciées :

$$\left\{ \rho'_3 : \mathtt{e}(1) \leftarrow \mathtt{d}(1). \quad \rho''_3 : \mathtt{e}(2) \leftarrow \mathtt{d}(2). \right\}$$

La règle ρ'_3 ne sera pas appliquée car $\mathtt{d}(1) \notin \mathcal{F}_{23}$. Mais la règle ρ''_3 est appliquée ce qui déclenche la contrainte ρ''_4 lorsque $\mathtt{p}(1,2)$ est déduit. Il n'y a au final que deux answer sets corrects qui sont AS_2 et AS_4 et donc la réponse crédule à la requête est $\{[\mathtt{X} \mapsto 1]\}$. Dans cet exemple nous n'avons pas eu besoin d'instancier la règle ρ_3 à partir de $\mathtt{d}(4)$ qui n'est pas en relation avec la requête.

7.3. Instanciation des règles dangereuses négatives

Un autre axe d'optimisation du nombre d'instanciations concerne les règles dangereuses négatives. Contrairement aux règles dangereuses positives qui peuvent être responsables de l'application d'une règle d'un cycle dangereux, les règles dangereuses négatives ont pour particularité de pouvoir bloquer l'application d'autres règles dangereuses et ne peuvent pas être responsables de l'application d'une règle d'un cycle dangereux. Il est donc possible de réduire le nombre d'instanciations de ces règles à la fois pour le test de consistance et lors de la recherche de réponses à une requête. Dans le cadre du test d'inconsistance, nous pouvons conjecturer que l'instanciation d'une règle dangereuse négative n'est pas nécessaire tant que l'instanciation d'une règle dangereuse positive ne provoque pas d'inconsistance. Dans le cas où une instance est responsable d'une inconsistance il est possible d'utiliser une marche arrière pour vérifier s'il existe une instance d'une règle dangereuse négative qui permet de bloquer la règle instanciée responsable de l'inconsistance.

EXEMPLE 25. — *Prenons l'exemple suivant : Soit le programme $P_{25} = (\mathcal{F}_{25}, \mathcal{R}_{25})$ composé de : $\mathcal{R}_{25}$ =*

$$\left\{ \begin{array}{ll} \rho_1 : \mathtt{p}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{a}(\mathtt{X},\mathtt{Y}), \mathtt{not\ q}(\mathtt{X},\mathtt{Y}). & \rho_2 : \mathtt{q}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{a}(\mathtt{X},\mathtt{Y}), \mathtt{not\ p}(\mathtt{X},\mathtt{Y}). \\ \rho_3 : \mathtt{e}(\mathtt{X}) \leftarrow \mathtt{d}(\mathtt{X}). & \rho_6 : \mathtt{s}(\mathtt{Y}) \leftarrow \mathtt{p}(\mathtt{X},\mathtt{Y}). \\ \rho_7 : \mathtt{c1} \leftarrow \mathtt{e}(\mathtt{X}), \mathtt{not\ s}(\mathtt{X}), \mathtt{not\ c1}. & \end{array} \right\}$$

et de la base de faits $\mathcal{F}_{25} = \{\mathtt{a}(1,1), \mathtt{a}(3,3), \mathtt{a}(1,2), \mathtt{a}(1,4), \mathtt{d}(2), \mathtt{d}(4)\}$. Ce programme correspond au programme P_{23} dont les règles ρ_4 et ρ_5 ont été modifiées et auquel le fait $\mathtt{a}(1,4)$ a été ajouté. Les règles dangereuses positives sont ρ_7 et ρ_3 et les règles dangereuses négatives sont ρ_6, ρ_1 et ρ_2. L'instanciation des règles dangereuses négatives n'est nécessaire que si les règles dangereuses positives permettent de déclencher la contrainte : seules les instances permettant de vérifier si la règle est bloquée sont nécessaires. Dans le programme P_{25}, nous allons d'abord tester si les instances de ρ_3 permettent d'appliquer la contrainte ρ_7 avant d'instancier ρ_6, car une application de ρ_6 ne peut pas permettre l'application de ρ_7 donc si une instance de ρ_3 ne permet pas d'appliquer ρ_7 alors il n'est pas nécessaire de tester les instances

correspondantes de ρ_6. Pour tester la consistance il est juste nécessaire d'instancier ρ_3 avec $\mathtt{d}(2)$ *et* $\mathtt{d}(4)$ *puis d'effectuer une marche arrière pour vérifier si l'on peut obtenir* $\mathtt{s}(2)$ *et* $\mathtt{s}(4)$*. Il n'est donc pas nécessaire d'instancier ρ_1 et ρ_2 avec* $\mathtt{a}(1,1)$ *et* $\mathtt{a}(3,3)$ *pour montrer que P_{25} est consistant.*

Pour conclure sur l'instanciation, nous constatons qu'il est possible de réduire le nombre d'instanciations en fonction de la requête même en prenant en considération les programmes possiblement inconsistants. Cette démarche diffère de ce qui est proposé avec DLV+M où seule la réponse pour les programmes super-consistant est assurée correcte. Elle est plus proche de l'utilisation des `magic sets` de (Faber *et al.*, 2007) qui traite des règles dangereuses mais pas des programmes inconsistants.

8. Conclusion

Dans cet article, nous nous sommes intéressés à l'interrogation en ASP qui permet d'utiliser des ontologies plus expressives que les ontologies classiques. Nous avons proposé une définition formelle de l'interrogation dont une implémentation a été réalisée. Nous avons étudié le problème de l'inconsistance lors de l'interrogation et nous avons montré comment isoler des règles, en utilisant les dépendances entre les prédicats, pour rendre plus efficace le calcul de la réponse à une requête tout en conservant la correction de la réponse. D'un point de vue pratique, les différents tests effectués ont montré que les solveurs ASP offraient des résultats pouvant rivaliser avec les raisonneurs sur les ontologies tout en étant capables de traiter des informations plus riches. L'interrogation pourrait néanmoins être encore améliorée dans le cadre de l'ASP en essayant de limiter les instanciations.

Bibliographie

Abiteboul S., Hull R., Vianu V. (1995). *Foundations of Databases*. Addison-Wesley.

Alviano M., Dodaro C., Faber W., Leone N., Ricca F. (2013). WASP: A Native ASP Solver Based on Constraint Learning. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, vol. 8148, p. 55-67.

Alviano M., Faber W. (2011). Dynamic Magic Sets and Super-Coherent Answer Set Programs. *AI Communication*, vol. 24, n° 2, p. 125–145.

Alviano M., Faber W., Woltran S. (2014). Complexity of Super-Coherence Problems in ASP. *Theory and Practice of Logic Programming*, vol. 14, p. 339–361.

Alviano M., Leone N., Manna M., Terracina G., Veltri P. (2012). Magic Sets for Datalog with Existential Quantifiers. In *Proceedings of the 2nd International Workshop on Datalog in Academia and Industry, Datalog 2.0*, p. 31–43.

Alviano M., Morak M., Pieris A. (2017). Stable Model Semantics for Tuple-Generating Dependencies Revisited. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, p. 377–388.

Apt K. R., Bol R. (1994). Logic Programming and Negation: A Survey. *Journal of Logic Programming*, vol. 19, p. 9–71.

Arenas M., Gottlob G., Pieris A. (2014). Expressive Languages for Querying the Semantic Web. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, p. 14–26.

Baget J.-F., Garcia L., Garreau F., Lefèvre C., Rocher S., Stéphan I. (2018). Bringing Existential Variables in Answer Set Programming and Bringing Non-Monotony in Existential Rules: Two Sides of the Same Coin. *Annals of Mathematics and Artificial Intelligence*, vol. 82, n° 1-3, p. 3–41.

Baget J.-F., Garreau F., Mugnier M.-L., Rocher S. (2014). Revisiting Chase Termination for Existential Rules and their Extension to Nonmonotonic Negation. In *Proceedings of the 15th International Workshop on Non-Monotonic Reasoning (NMR'14).*

Baget J.-F., Leclère M., Mugnier M.-L., Salvat E. (2011). On Rules with Existential Variables: Walking the Decidability Line. *Artificial Intelligence*, vol. 175, n° 9-10, p. 1620-1654.

Bancilhon F., Maier D., Sagiv Y., Ullman J. D. (1986). Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract). In *Proceedings of the* 5^{th} *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, p. 1–15.

Benchmark university dlv. (2012). http://www.mat.unical.it/kr2012/.

Benchmark university graal. (2014). http://graphik-team.github.io/graal/experiments1.

Cali A., Gottlob G., Lukasiewicz T., Marnette B., Pieris A. (2010). Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science*, p. 228–242.

Calvanese D., Giacomo G. D., Lembo D., Lenzerini M., Rosati R. (2007). Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated Reasoning*, vol. 39, n° 3, p. 385-429.

Ceri S., Gottlob G., Tanca L. (1989). What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, n° 1, p. 146–166.

Costa N., Knorr M., Leite J. (2015). Querying LUBM with Non-monotonic Features in Protege using NoHR. In *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the* 14^{th} *International Semantic Web Conference (ISWC'15).*

Dix J. (1992). A Framework for Representing and Characterizing Semantics of Logic Programs. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, p. 591–602.

Dung P. (1992). On the Relations between Stable and Well-Founded Semantics of Logic Programs. *Theoretical Computer Science*, vol. 105, n° 1, p. 7 - 25.

Eiter T., Ianni G., Lukasiewicz T., Schindlauer R., Tompits H. (2008). Combining Answer Set Programming with Description Logics for the Semantic Web. *Artificial Intelligence*, vol. 172, n° 12-13, p. 1495–1539.

Faber W., Greco G., Leone N. (2007). Magic Sets and their Application to Data Integration. *Journal of Computer and System Sciences*, vol. 73, n° 4, p. 584 - 609. (Special Issue: Database Theory 2005)

Faber W., Leone N., Perri S. (2012). The Intelligent Grounder of DLV. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz, LNCS*, vol. 7265, p. 247-264.

Ferraris P., Lee J., Lifschitz V. (2011). Stable Models and Circumscription. *Artificial Intelligence*, vol. 175, n° 1, p. 236 - 263.

Gallaire H., Minker J., Nicolas J. (1984). Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, vol. 16, n° 2, p. 153–185.

Garreau F., Garcia L., Lefèvre C., Stéphan I. (2015). ∃-ASP. In *Proceedings of ONTOLP Workshop (co-located with IJCAI'15).*

Gebser M., Kaminski R., Kaufmann B., Schaub T. (2014). Clingo = ASP + Control: Preliminary Report. In *Proceedings of Technical Communications of the 29th International Conference on Logic Programming (ICLP'14).*

Gebser M., Kaminski R., König A., Schaub T. (2011). Advances in *gringo* Series 3. In *Proceedings of* 11th *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11), LNCS*, vol. 6645, p. 345-351.

Gebser M., Kaufmann B., Neumann A., Schaub T. (2007). Conflict-Driven Answer Set Solving. In *Proceedings of the* 20th *International Joint Conference on Artificial Intelligence (IJCAI'07)*, p. 386-392.

Gebser M., Obermeier P., Schaub T. (2013). A System for Interactive Query Answering with Answer Set Programming. In *Proceedings of the* 6th *Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'13).*

Gelfond M., Lifschitz V. (1988). The Stable Model Semantics for Logic Programming. In *Proceedings of the* 5th *International Conference and Symposium on Logic Programming (iclp'88)*, p. 1070-1080.

Glimm B., Horrocks I., Motik B., Stoilos G., Wang Z. (2014). Hermit: An OWL 2 Reasoner. *Journal of Automated Reasoning*, vol. 53, n° 3, p. 245–269.

Gottlob G., Hernich A., Kupke C., Lukasiewicz T. (2012). Equality-friendly Well-founded Semantics and Applications to Description Logics. In *Proceedings of the* 26th *National Conference on Artificial Intelligence (AAAI'12)*, p. 757–764.

Gottlob G., Hernich A., Kupke C., Lukasiewicz T. (2014). Stable Model Semantics for Guarded Existential Rules and Description Logics. In *Proceedings of the 14th International Conference on the Principles of Knowledge Representation and Reasoning (KR'14)*, p. 258–267.

Guo Y., Pan Z., Heflin J. (2005). LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, vol. 3, n° 2-3, p. 158–182.

Hernich A., Kupke C., Lukasiewicz T., Gottlob G. (2013). Well-founded Semantics for Extended Datalog and Ontological Reasoning. In *Proceedings of the 32rd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, p. 225–236.

Kollia I., Glimm B., Horrocks I. (2011). SPARQL Query Answering over OWL Ontologies. In *The Semantic Web: Research and Applications -* 8th *Extended Semantic Web Conference (ESWC'11)*, p. 382–396.

Lefèvre C., Béatrix C., Stéphan I., Garcia L. (2017). *ASPeRiX*, a First Order Forward Chaining Approach for Answer Set Computing. *Theory and Pratice of Logic Programming*, vol. 17, n° 3, p. 266-310.

Leone N., Manna M., Terracina G., Veltri P. (2012). Efficiently Computable Datalog∃ Programs. In *Proceedings of the* 13th *International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, p. 13–23.

Leone N., Pfeifer G., Faber W., Eiter T., Gottlob G., Perri S. *et al*. (2006). The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, vol. 7, n° 3, p. 499-562.

Lloyd J. (1987). *Foundations of Logic Programming*. Springer-Verlag New York, Inc.

Lukasiewicz T. (2004). A Novel Combination of Answer Set Programming with Description Logics for the Semantic Web. In *Proceedings of the 14^{th} International Conference on Principles of Knowledge Representation and Reasoning (KR'04)*, p. 141–151.

Magka D., Krötzsch M., Horrocks I. (2013). Computing Stable Models for Nonmonotonic Existential Rules. In *Proceedings of the 23^{rd} International Joint Conference on Artificial Intelligence (IJCAI'13)*, p. 1031–1038.

Motik B., Rosati R. (2008). Reconciling Description Logics and Rules. *Journal of ACM*, vol. 57, n° 5, p. 30:1–30:62.

Musen M. A. (2015). The protégé Project: a Look Back and a Look Forward. *AI Matters*, vol. 1, n° 4, p. 4–12.

Owl2dlgp graal. (2015). http://graphik-team.github.io/graal/downloads/owl2dlgp.

OWL 2 Web Ontology Language Document Overview (Second Edition). (2012). https://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

Pérez-Urbina H., Horrocks I., Motik B. (2009). Efficient Query Answering for OWL 2. In *Proceedings of the 8^{th} International Semantic Web Conference, (ISWC'09)*, p. 489–504.

Prud'hommeaux E., Seaborne A. (2008). SPARQL Query Language for RDF. In *W3c: Available at http://www.w3.org/tr/rdf-sparql-query/.*

Schaub T., Thielscher M. (1996). Skeptical Query-Answering in Constrained Default Logic. In *Proceedings of the International Conference on Formal and Applied Practical Reasoning (FAPR'96)*, p. 567–581.

Sirin E., Parsia B., Grau B. C., Kalyanpur A., Katz Y. (2007). Pellet: A Practical OWL-DL Reasoner. *Journal of Web Semantics*, vol. 5, n° 2, p. 51–53.

DLV - User Manual. (2004). http://www.dlvsystem.com/html/DLV_User_Manual.html.

Van Gelder A., Ross K. A., Schlipf J. S. (1991). The Well-founded Semantics for General Logic Programs. *Journal of the ACM (JACM)*, vol. 38, n° 3, p. 619–649.

Weinzierl A. (2017). Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In *Proceedings of the 14^{th} International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17), LNCS*, p. 191–204.

Annexes

A. Présentation des tests

Pour la réalisation des tests de cet article, différents outils, comme des solveurs ASP et les raisonneurs `NoHR` et `Pellet`, ont été utilisés avec différents paramètres. Nous présentons ici les détails d'utilisation de ces outils.

`CLINGO` Le solveur `Clingo` est utilisé dans sa version 4.5.4. `Clingo` ne propose pas d'interrogation, nous proposons alors de calculer entièrement les *answer sets* à l'aide des options `-brave` et `-cautious` pour obtenir les *answer sets* crédules et sceptiques puis d'afficher uniquement les atomes réponses en tête des règles désignées comme des requêtes. La requête est une règle ASP avec l'atome réponse en tête en y ajoutant la directive `#show ans/n.`, celle-ci permettant de désigner dans les *answer sets* les atomes à afficher uniquement [17]. Le résultat obtenu est l'ensemble des instances de l'atome réponse. Plusieurs requêtes sont autorisées pour un programme. Les temps de `Clingo` sont obtenus avec la commande `time` UNIX et comprend donc le temps de parsing et le calcul de l'ensemble des *answer sets*.

`CLINGO`+Q Le solveur `Clingo` est utilisé dans sa version 4.5.4 en utilisant en pré-traitement l'isolation des règles proposée dans cet article. Ainsi seul le programme d'entrée fourni à `Clingo` est différent, ne comprenant que les règles nécessaires à la requête après le pré-traitement effectué par l'isolation des règles. Le temps correspond au temps nécessaire à l'isolation des règles, au parsing et au calcul de tous les *answer sets* du sous-programme.

`DLV`+M. Le solveur `DLV` est utilisé dans sa version du 17 décembre 2012 avec module d'interrogation utilisant les `magic sets` avec l'option `-OMS`. Les *answer sets* sont calculés partiellement à l'aide des `magic sets` et nous avons le choix entre une réponse crédule ou sceptique à l'aide des options `-brave` et `-cautious`. Une seule requête est autorisée pour un programme. Les temps affichés sont les temps obtenus avec la commande `time` UNIX et comprend donc le temps de parsing et de calcul de la réponse à la requête.

`PROTÉGÉ` Pour charger les ontologies avant l'utilisation des raisonneurs nous utilisons `protégé` en version 5.2.0.

`PELLET` Le raisonneur Pellet est utilisé dans sa version 2.3.0. Le raisonneur ne permet pas de traiter des programmes avec négation par défaut. Le temps correspond au chargement de l'ontologie et à l'interrogation.

`NoHR` Le raisonneur `NoHR` est utilisé dans sa version 3.0. Le raisonneur ne permet pas de choisir une réponse crédule ou sceptique car il utilise la sémantique bien fondée avec laquelle les programmes admettent un unique *answer set*. Le temps correspond au chargement de l'ontologie, au chargement dans le module XSB [18] et à l'interrogation.

---

17. `/n` correspondant à l'arité du prédicat de l'atome réponse
18. Permettant de simplifier la marche arrière utilisée pour l'interrogation

B. Présentation détaillée de `university`

`university` est un benchmark souvent utilisé pour l'interrogation sur les ontologies (Guo *et al*., 2005 ; Kollia *et al*., 2011 ; Pérez-Urbina *et al*., 2009). C'est une ontologie académique servant de jeu de test pour comparer les différents outils traitant les ontologies et l'interrogation de données.

STRUCTURE DE LA BASE DE RÈGLES (TBOX). La TBOX du benchmark `university` contient l'ensemble des règles définissant l'ontologie `university`. Elle peut être récupérée à l'adresse suivante :

http://swat.cse.lehigh.edu/onto/univ-bench.owl

Cette ontologie définit 43 classes et 32 propriétés (dont 25 *object properties* et 7 *datatype properties*). Elle utilise les fonctions du langage OWL-Lite suivantes

- `inverseOf`
- `TransitiveProperty`
- `someValuesFrom restrictions`
- `intersectionOf`

Cette ontologie comporte un petit nombre de classes mais beaucoup de restrictions et de propriétés par classe.

GÉNÉRATION DES INSTANCES AVEC UBA. L'outil UBA est un outil permettant de générer des données, définissant la base de faits (ABOX), pour le benchmark `university`. Il est disponible à l'adresse suivante :

http://swat.cse.lehigh.edu/projects/lubm/uba1.7.zip

Les paramètres du générateur sont les suivants :

- `-univ <univ num>` : pour définir le nombre d'universités à générer (par défaut 1)
- `-onto <ontology url>`: pour définir l'url de l'ontologie contenant les règles (dans notre cas la TBOX)

Les données générées dans l'article sont générées avec UBA pour une, deux et dix universités. Nous générons alors les ABOX avec la commande :

```
java edu.lehigh.swat.bench.uba.Generator -univ i -onto
http://swat.cse.lehigh.edu/onto/univ-bench.owl
```

avec `i` le nombre d'universités souhaité. UBA génère alors aléatoirement (selon une graine) entre quinze et vingt-cinq ABOX par université correspondant aux départements de l'université avec pour nom `Univesityi_j.owl` avec `i` le numéro de l'université et `j` le numéro de la ABOX correspondant au numéro du département allant de 0 à 25 en fonction du nombre de ABOX générées. Pour une université générée

avec les paramètres par défaut, il y a quinze départements donc quinze ABOX générées allant de `University0_0.owl` à `University0_14.owl`. La taille des données correspondante en ASP est de :

- 82 règles pour la version GRAAL,
- 114 règles pour la version LUBM,
- 102.108 faits pour 1 université et 15 départements,
- 236.269 faits pour 2 universités et 34 départements,
- 1.311.410 faits pour 10 universités et 189 départements.

Après instanciation des règles avec `Gringo` nous obtenons :

- 165.186 règles instanciées pour 1 université avec la version GRAAL,
- 177.737 règles instanciées pour 1 université avec la version LUBM,
- 377.132 règles instanciées pour 2 universités avec la version GRAAL,
- 405.885 règles instanciées pour 2 universités avec la version LUBM,
- 2.086.004 règles instanciées 10 universités avec la version GRAAL,
- 2.246.699 règles instanciées pour 10 universités avec la version LUBM.

Le nombre de règles obtenu avec `Gringo` correspond à la taille de l'unique *answer set* dans le cas d'un programme défini.

LES REQUÊTES. Le benchmark `university` est proposé avec quatorze requêtes conjonctives non-booléennes. Ces requêtes sont au format SPARQL mais peuvent être rapidement traduite en ASP. Les requêtes au format ASP sont les suivantes :

```
Q1:  ans(X) :-
        graduateStudent(X),
        takesCourse(X,"GraduateCourse0").
Q2:  ans(X) :-
        graduateStudent(X), university(Y), department(Z),
        memberOf(X,Z), subOrganizationOf(Z,Y),
        undergraduateDegreeFrom(X,Y).
Q3:  ans(X) :-
        publication(X),
        publicationAuthor(X,"AssistantProfessor0").
Q4:  ans(X,Y1,Y2,Y3) :-
        professor(X), worksFor(X,"Department0"),
        name(X,Y1), emailAddress(X,Y2), telephone(X,Y3).
Q5:  ans(X) :-
        person(X), memberOf(X,"Department0").
Q6:  ans(X) :- student(X).
Q7:  ans(X) :-
        student(X), course(Y),
        teacherOf("AssociateProfessor0",Y),
        takesCourse(X,Y).
```

```
Q8:  ans(X,Y,Z) :-
        student(X), department(Y), memberOf(X,Y),
        subOrganizationOf(Y,"University0"),
        emailAddress(X,Z).
Q9:  ans(X,Y,Z) :-
        student(X), faculty(Y), course(Z),
        advisor(X,Y), takesCourse(X,Z), teacherOf(Y,Z).
Q10: ans(X) :-
        student(X), takesCourse(X,"GraduateCourse0").
Q11: ans(X) :-
        researchGroup(X),
        subOrganizationOf(X,"University0").
Q12: ans(X,Y) :-
        chair(X), department(Y), worksFor(X,Y),
        subOrganizationOf(Y,"University0").
Q13: ans(X) :-
        person(X), hasAlumnus("University0",X).
Q14: ans(X) :-
        undergraduateStudent(X).
```

C. Tableaux de résultats complets

Nous proposons maintenant la comparaison des solveurs sur la version GRAAL et LUBM du benchmark university, sans négation par défaut, avec les différents protocoles. Les tableaux de 6 à 14 présentent les temps de réponse : pour une université sur la version GRAAL (tableaux 6 et 7) et sur la version LUBM (tableaux 8 et 9), pour deux universités sur la version GRAAL (tableaux 10 et 11) et sur la version LUBM (tableaux 12 et 13) et pour dix universités sur la version LUBM (tableaux 14 et 15).

Pour rappel, les raisonneurs ont besoin de charger l'ontologie avant d'effectuer l'interrogation. Le tableau 5 montre les temps de chargement nécessaires pour les raisonneurs Pellet et NoHR avec une, deux et dix universités, en prenant en compte le temps nécessaire pour charger l'ontologie dans protégé (temps entre parenthèses dans le tableau) avant l'utilisation des raisonneurs. Le temps de chargement pour protégé est identique pour les deux raisonneurs, le test étant réalisé indépendamment du raisonneur. Chaque mesure correspond à une moyenne sur vingt chargements.

Tableau 5. Temps de chargement en secondes sur les différentes versions d'university pour les raisonneurs

	GRAAL 1	GRAAL 2	GRAAL 10	LUBM 1	LUBM 2	LUBM 10
Pellet	(4,6)+1,1	(7,3)+2,5	(21,6)+15,2	(11,7)+2,3	(13,3)+4,9	(23,2)+26
NoHR	(4,6)+6	(7,3)+15,4	(21,6)+71,4	-	-	-

Dans le tableau 6 (GRAAL pour une université), on remarque que les temps des solveurs sont quasiment instantanés mais nous pouvons tout de même constater un gain allant de 10 à 90 % lors de l'utilisation de l'algorithme d'isolation des règles (cf.

Tableau 6. Temps de réponse en secondes pour une université sur la version GRAAL

	q1	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**
Clingo	0,8	0,8	0,8	0,8	0,8	0,8	0,8
Clingo+Q	0,3(0,1)	0,3(0,1)	0,22(0,1)	0,4(0,1)	0,4(0,1)	0,1(0,1)	0,4(0,1)
DLV+M	0,4	0,4	0,4	0,4	0,4	0,4	0,4
Pellet	-	5,7(5,7)	5,7(5,7)	5,7(5,7)	5,7(5,7)	5,7(5,7)	5,7(5,7)
NoHR	10,6(10,6)	10,6(10,6)	10,6(10,6)	10,6(10,6)	10,8(10,6)	11,1(10,6)	10,6(10,6)
	q8	**q9**	**q10**	**q11**	**q12**	**q13**	**q14**
Clingo	0,8	0,8	0,8	0,8	0,8	0,8	0,8
Clingo+Q	0,4(0,1)	0,4(0,1)	0,3(0,1)	0,2(0,1)	0,1(0,1)	0,4(0,1)	0,1(0,1)
DLV+M	0,5	0,4	0,4	0,4	0,5	0,4	0,4
Pellet	5,7(5,7)	5,7(5,7)	5,7(5,7)	5,7(5,7)	5,7(5,7)	5,7(5,7)	5,7(5,7)
NoHR	11,1(10,6)	10,9(10,6)	10,6(10,6)	10,6(10,6)	10,6(10,6)	10,6(10,6)	11,2(10,6)

requête q14 avec Clingo+Q). Les requêtes bénéficiant des meilleures améliorations de temps correspondent aux requêtes contenant peu d'atomes dans le corps de celles-ci et pour lesquelles le programme obtenu après isolation contient un nombre de règles très réduit.

Tableau 7. Temps de réponse en secondes pour une université sur l'ensemble des requêtes avec la version GRAAL

	Clingo	Clingo+Q	DLV+M	Pellet	NoHR
q_{tot}	0,8	0,8(0,2)	5,8	5,7(5,7)	12,7(10,6)

Tableau 8. Temps de réponse en secondes pour une université sur la version LUBM

	q1	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**
Clingo	0,9	0,9	0,9	0,9	0,9	0,9	0,9
Clingo+Q	0,8(0,1)	0,8(0,1)	0,22(0,1)	1(0,1)	0,8(0,1)	0,8(0,1)	0,8(0,1)
DLV+M	0,4	0,7	0,4	1,0	1,3	0,7	1,2
Pellet	-	14(14)	14(14)	14(14)	14(14)	14(14)	14(14)
	q8	**q9**	**q10**	**q11**	**q12**	**q13**	**q14**
Clingo	0,9	0,9	0,9	0,9	0,9	0,9	0,9
Clingo+Q	0,8(0,1)	0,8(0,1)	1(0,1)	0,1(0,1)	0,8(0,1)	0,8(0,1)	0,1(0,1)
DLV+M	2,0	0,7	1,0	0,4	1,0	0,9	0,4
Pellet	14(14)	14(14)	14(14)	14(14)	14(14)	14(14)	14(14)

Dans le tableau 8 (LUBM pour une université), nous pouvons faire le même constat que pour la version GRAAL avec des temps légèrement plus longs et l'absence de NoHR qui ne permet pas de traiter l'ontologie sous le format LUBM. Les résultats pour deux universités sont similaires (tableaux 10 et 12).

Le tableau 14 (LUBM pour dix universités) permet de constater que le benchmark LUBM est plus difficile à traiter que la version GRAAL. Nous remarquons dans un premier temps que DLV+M et Clingo+Q prennent environ deux fois plus de temps pour traiter la majorité des requêtes. Ces résultats peuvent s'expliquer par le nombre d'instances nécessaires plus élevé avec LUBM qui reste proche du programme sans isolation des règles (cf. tableau 16). Il existe plus de dépendances entre les symboles de prédicats dans la version LUBM ce qui explique que le nombre d'instance est souvent proche de la version non isolée. Dans ce cas l'isolation proposée n'est pas très efficace sauf pour certaines requêtes précises comme les requêtes q3, q11 ou q14.

Tableau 9. Temps de réponse en secondes pour une université sur l'ensemble des requêtes avec la version LUBM

	Clingo	Clingo+Q	DLV+M	Pellet
q_{tot}	0,9	0,9(0,2)	83	14(14)

Tableau 10. Temps de réponse en secondes pour deux universités sur la version GRAAL

	q1	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**
Clingo	2,0	2,0	2,0	2,0	2,0	2,0	2,0
Clingo+Q	0,6(0,1)	0,5(0,1)	0,4(0,1)	0,8(0,1)	0,8(0,1)	0,22(0,1)	0,7(0,1)
DLV+M	0,9	1,1	1,0	1,0	1,0	0,9	1,1
Pellet	-	9,8(9,8)	9,8(9,8)	9,8(9,8)	9,8(9,8)	9,8(9,8)	9,8(9,8)
NoHR	22,7(22,7)	22,9(22,7)	22,7(22,7)	22,7(22,7)	23,1(22,7)	24(22,7)	22,9(22,7)
	q8	**q9**	**q10**	**q11**	**q12**	**q13**	**q14**
Clingo	2,0	2,0	2,0	2,0	2,0	2,0	2,0
Clingo+Q	0,7(0,1)	0,8(0,1)	0,6(0,1)	0,1(0,1)	0,2(0,1)	0,8(0,1)	0,1(0,1)
DLV+M	1,1	1,0	0,9	0,9	1,2	1,0	0,9
Pellet	9,8(9,8)	9,8(9,8)	9,8(9,8)	9,8(9,8)	9,8(9,8)	9,8(9,8)	9,8(9,8)
NoHR	25(22,7)	24,5(22,7)	22,7(22,7)	22,7(22,7)	22,7(22,7)	22,7(22,7)	26,9(22,7)

Tableau 11. Temps de réponse en secondes pour deux universités sur l'ensemble des requêtes avec la version GRAAL

	Clingo	Clingo+Q	DLV+M	Pellet	NoHR
q_{tot}	2	1,8(0,2)	14	9,8(9,8)	33(22,7)

Tableau 12. Temps de réponse en secondes pour deux universités sur la version LUBM

	q1	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**
Clingo	2,2	2,2	2,2	2,2	2,2	2,2	2,2
Clingo+Q	1,8(0,1)	1,8(0,1)	0,4(0,1)	2,1(0,1)	1,8(0,1)	1,8(0,1)	1,8(0,1)
DLV+M	0,9	1,8	0,9	2,5	3,0	1,7	2,9
Pellet	-	18,2(18,2)	18,2(18,2)	18,2(18,2)	18,2(18,2)	18,2(18,2)	18,2(18,2)
	q8	**q9**	**q10**	**q11**	**q12**	**q13**	**q14**
Clingo	2,2	2,2	2,2	2,2	2,2	2,2	2,2
Clingo+Q	1,8(0,1)	1,8(0,1)	1,8(0,1)	0,1(0,1)	1,8(0,1)	1,8(0,1)	0,1(0,1)
DLV+M	4,2	1,9	2,5	0,9	2,5	2,5	0,9
Pellet	18,2(18,2)	18,2(18,2)	18,2(18,2)	18,2(18,2)	18,2(18,2)	18,2(18,2)	18,2(18,2)

Tableau 13. Temps de réponse en secondes pour deux universités sur l'ensemble des requêtes avec la version LUBM

	Clingo	Clingo+Q	DLV+M	Pellet
q_{tot}	2,2	2,3(0,4)	29,1	18,2(18,2)

Tableau 14. Temps de réponse en secondes pour dix universités sur la version LUBM

	q1	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**
Clingo	13,8	13,8	13,8	13,8	13,8	13,8	13,8
Clingo+Q	12,9(2,6)	13,2(2,8)	3,1(1,6)	15,5(3,5)	12,8(2,6)	13,2(2,6)	12,8(2,6)
DLV+M	5,6	10,5	6,0	15,2	17,4	10,7	17,3
Pellet	-	52,4(49,2)	49,6(49,2)	50,2(49,2)	50,3(49,2)	49,3(49,2)	50(49,2)
	q8	**q9**	**q10**	**q11**	**q12**	**q13**	**q14**
Clingo	13,8	13,8	13,8	13,8	13,8	13,8	13,8
Clingo+Q	12,9(2,6)	12,9(2,6)	12,8(2,7)	0,9(0,9)	12,7(2,6)	12,7+(2,6)	1,5(0,9)
DLV+M	18,9	11,2	14,8	5,7	15,1	14,1	6,0
Pellet	50,2(49,2)	50,2(49,2)	49,5(49,2)	744,2(49,2)	49,4(49,2)	49,4(49,2)	49,3(49,2)

Par contre, le temps de Clingo n'augmente que de 15 %. Pour l'interrogation avec Pellet, les temps sont les mêmes que pour la version GRAAL sauf pour la requête q2 mais le temps de préchargement a augmenté d'environ 33 % passant de 36,8 secondes à 49,2 secondes.

Tableau 15. Temps de réponse en secondes pour dix universités sur l'ensemble des requêtes avec la version LUBM

	Clingo	Clingo+Q	DLV+M	Pellet
q_{tot}	13,8	15,4(4,1)	168,5	752,8(49,2)

Tableau 16. Nombres de règles après instanciation du programme isolé pour dix universités sur la version LUBM

q1	**q2**	**q3**	**q4**	**q5**	**q6**	**q7**	**q8**
1 811 495	1 811 495	213 400	2 015 104	1 811 495	1 811 495	1 811 495	1 811 495
q9	**q10**	**q11**	**q12**	**q13**	**q14**	**qtot**	**tot**
1 811 495	1 811 495	22 590	1 811 495	1 811 495	75 547	2 088 677	2 246 699

Les tableaux 7, 9, 11, 13 et 15 présentent les temps de réponse pour l'ensemble des requêtes. Les commentaires faits à propos de la version GRAAL pour dix universités sont également pertinents dans tous ces autres cas.

Preuves

PREUVE. — du théorème 12 (Correction et complétude de l'algorithme 1). Soit l'ensemble $\mathcal{R}$ des règles d'un programme P et le graphe $\mathcal{G}$ de dépendance des symboles de prédicat de P. Nous avons $\mathcal{D}$ l'ensemble des ensembles de règles dangereuses marquées de P à l'issue de l'algorithme 1. Supposons d_r^+ une règle dangereuse positive telle que $d_r^+ \notin D$ avec $D \in \mathcal{D}$. D'après la définition 10 une règle est dangereuse positive si son symbole de prédicat en tête appartient à un cycle d'inconsistance ou au corps positif d'une règle dangereuse positive. Si d_r^+ appartient à un cycle d'inconsistance alors $(d_r^+, +) \in D$ avec $D \in \mathcal{D}$ suite à l'application de la fonction CycleInconsistant$(\mathcal{G}, \mathcal{R})$ qui calcule toutes les règles ayant un symbole de prédicat du cycle d'inconsistance en tête. d_r^+ ne peut donc pas appartenir à un cycle auquel cas il appartiendrait à un ensemble de $\mathcal{D}$. Dans ce cas, $\mathtt{p} = pred(tête(d_r^+))$ tel que $\mathtt{p} \in pred(corps^+(r^+))$ avec $(r^+, +) \in D$ une règle dangereuse positive. L'algorithme 1 calcule, pour chaque cycle d'inconsistance C, toutes les règles $r \in \mathcal{R}$ avec un symbole de prédicat $\mathtt{p} = pred(tête(r))$ tel que $\mathtt{p} \in pred(corps^+(r^+))$ avec $(r^+, +) \in D$ et D l'ensemble des règles dépendantes du cycle d'inconsistance C, ce qui revient à parcourir toutes les règles dont le cycle C dépend. Si d_r^+ n'est pas détectée alors $tête(d_r^+) \notin corps^+(r^+)$, et d_r^+ ne peut donc pas être dangereuse positive sans appartenir à un ensemble de $\mathcal{D}$.
Supposons maintenant d_r^- une règle dangereuse négative non détectée par l'algorithme 1. D'après la définition 10, une règle dangereuse est négative si elle n'est pas positive et que son symbole de prédicat en tête apparaît dans le corps négatif d'une règle dangereuse positive ou dans le corps d'une règle dangereuse négative. Si $(d_r^-, +) \in D$ avec $D \in \mathcal{D}$ alors $(d_r^-, -) \notin D$ d'après la définition de la

fonction $dangereuses$ qui marque négativement une règle seulement si elle n'est pas déjà marquée positivement, donc $(d_r^-, +) \notin D$. Soit le symbole de prédicat $\mathtt{p} = pred(tête(d_r^-))$ tel que $\mathtt{p} \in pred(corps^-(r^+))$ avec $(r^+, +) \in D$ alors $d_r^- \in D$ car toutes les règles dangereuses positives sont déjà détectées par l'algorithme et la seconde boucle ajoute toutes les règles dangereuses négatives respectant cette condition. Si $\mathtt{p} = pred(tête(d_r^-))$ tel que $\mathtt{p} \in pred(corps(r^-))$ avec $(r^-, -) \in D$ alors $d_r^- \in D$ car l'ensemble des règles dangereuses négatives pouvant provenir d'une règle dangereuse positive sont détectées lors de la deuxième boucle de l'algorithme, la troisième boucle détecte ainsi toutes les règles dangereuses négatives issues du corps des règles dangereuses négatives détectées précédemment. d_r^- ne peut donc pas être dangereuse négative sans appartenir à un ensemble de $\mathcal{D}$. Cet algorithme est donc complet car toutes les règles dangereuses sont stockées.
Supposons maintenant qu'il existe une règle r non dangereuse appartenant à un ensemble de $\mathcal{D}$, alors le symbole de prédicat $pred(tête(r))$ n'appartient ni à un cycle d'inconsistance de $\mathcal{G}$ ni au corps d'une règle dangereuse. L'algorithme stocke dans un premier temps les règles dont le symbole de prédicat en tête appartient à un cycle d'inconsistance, r n'est donc pas stockée à ce moment là. Par la suite nous parcourons les règles dangereuses de $\mathcal{D}$ afin de stocker les règles ayant un symbole de prédicat en tête appartenant au corps (positif ou négatif) des règles parcourues. Les règles stockées dans $\mathcal{D}$ à ce moment sont des règles dangereuses (positives ou négatives) car elles appartiennent au corps d'une règle dangereuse. Nous avons donc parcouru toutes les règles stockées dans $\mathcal{D}$ sans pouvoir stocker r. L'algorithme est donc correct car il est impossible de stocker une règle non dangereuse dans $\mathcal{D}$. ■

PREUVE. — du théorème 17 (Inconsistance et règles dangereuses). Soit $P = (\mathcal{F}, \mathcal{R})$ un programme et Δ l'ensemble des ensembles de règles dangereuses unifiées de P. Supposons que P soit inconsistant, avec $\mathcal{G}$ son graphe des symboles de prédicat, et que $P' = (\mathcal{F}, \bigcup_{D \in \Delta} D)$ possède un *answer set*. Si P est inconsistant alors d'après le théorème 7, il existe un cycle d'inconsistance dans $\mathcal{G}$ le graphe des symboles de prédicat de P. D'après la définition 10, il existe un ensemble de règles dangereuses pour chaque cycle d'inconsistance de $\mathcal{G}$, les règles dangereuses d'un ensemble sont toutes les règles dont dépend un cycle d'inconsistance. Soit $D \in \Delta$ un ensemble de règles dangereuses responsable de l'inconsistance dans P, d'après l'algorithme 1, D contient alors toutes les règles dépendantes d'au moins un cycle d'inconsistance de $\mathcal{G}$. L'algorithme 2 calcule l'union des règles dangereuses de chaque ensemble de $\mathcal{D}$ contenant l'ensemble des règles rendant P inconsistant. Les seules règles écartées sont les règles n'ayant aucune dépendance avec les cycles d'inconsistance et ne pouvant pas rendre le programme inconsistant. L'ensemble D appartient donc à Δ. S'il existe un *answer set* pour P', P ne peut pas être inconsistant étant donné que les cycles d'inconsistance de P ne dépendent que des règles de D. Nous avons donc, si P est inconsistant alors P' est aussi inconsistant.

Supposons maintenant que P' soit inconsistant et que P possède un *answer set*. Si P' est inconsistant alors d'après le théorème 7 il existe un cycle d'inconsistance dans $\mathcal{G}'$, le graphe des symboles de prédicat de P', et donc $\Delta' \neq \emptyset$ avec Δ' l'ensemble des ensembles de règles dangereuses unifiées du programme P'. D'après les algorithmes

1 et 2, $\bigcup_{D\in\Delta} D$ contient toutes les règles dangereuses de l'ensemble de règles $\mathcal{R}$. Le programme P' contient donc toutes les règles dangereuses de P, l'ensemble Δ étant construit à partir des dépendances des règles dangereuses dont le symbole de prédicat en tête appartient à un cycle d'inconsistance, ces règles dangereuses appartiennent à Δ et donc appartiennent aussi à P, ainsi que toutes les règles dont elles dépendent (c'est le résultat de l'algorithme 1). Étant donné que Δ contient toutes les règles pouvant rendre le programme P inconsistant, nous pouvons en déduire qu'en calculant Δ' nous calculons les cycles d'inconsistance de l'ensemble de règles $\mathcal{R}$. Nous avons donc $\Delta' = \Delta$. De plus nous avons $\mathcal{R}\backslash(\bigcup_{D\in\Delta} D)$ un ensemble de règles n'ayant aucune dépendance avec les règles appartenant aux ensembles de Δ. Donc si P' est inconsistant il n'existe aucune règle appartenant à $\mathcal{R}$ pouvant modifier la consistance du programme P. Donc si P' est inconsistant alors P ne peut pas être consistant. ■

PREUVE. — du théorème 21 (Réponse à un programme consistant). Soit $(P?Q) = (\mathcal{F}, \mathcal{R}\cup\{Q\})$ un programme consistant. Supposons qu'il existe une réponse dans ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) n'étant pas une réponse dans $(P?Q)$. D'après l'algorithme 4, $D\mathcal{Q}(\mathcal{R},Q)$ contient l'ensemble des règles dangereuses à l'intersection de la requête ainsi que les règles dépendantes de celles-ci. Étant donné que $(P?Q)$ est consistant et que les autres ensembles de règles dangereuses n'ont pas de dépendance avec $règles(D\mathcal{Q}(\mathcal{R},Q))$ alors $(\mathcal{F}, règles(D\mathcal{Q}(\mathcal{R},Q)))$ est consistant. Étant donné que $D\mathcal{Q}(\mathcal{R},Q)$ contient toutes les règles dangereuses à l'intersection de $(\mathcal{R}\downarrow Q)$, il n'existe pas d'autres règles dangereuses dont l'ensemble $(\mathcal{R}\downarrow Q)$ est dépendant. Comme $règles(D\mathcal{Q}(\mathcal{R},Q))$ est consistant, nous pouvons en déduire que ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) est consistant. Il existe donc au moins une réponse dans $(P?Q)$ et dans ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) non `absurde`. D'après le théorème 19, nous savons que si le programme est consistant quelque soit $\mathcal{F}$ alors une réponse dans ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)$) est une réponse dans $(P?Q)$. D'après l'algorithme 3, $(\mathcal{R}\downarrow Q)$ contient toutes les règles dont la requête dépend et $règles(D\mathcal{Q}(\mathcal{R},Q))$ contient toutes les règles pouvant empêcher un ensemble d'atomes sur $(\mathcal{R}\downarrow Q)$ d'être un *answer set*. Étant donné que ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) contient l'ensemble des règles dont Q dépend ainsi que les règles dangereuses à l'intersection de ($\mathcal{F}$,$(\mathcal{R}\downarrow Q)$), toute application de règles dont Q dépend va déclencher les règles dangereuses à l'intersection. Donc s'il existe un *answer set* AS de ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) alors nous pouvons étendre celui-ci avec les applications des règles de $(P?Q)$ n'apparaissant pas dans ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) car $(P?Q)$ est consistant et il n'existe pas de règle dans $(P?Q)$ pouvant modifier les atomes issus de l'application des règles de ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) (il n'existe pas de dépendance). Nous en déduisons que toute réponse dans ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) est aussi une réponse dans $(P?Q)$.

Supposons maintenant qu'une réponse dans $(P?Q)$ n'est pas une réponse dans ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$). Alors il existe un *answer set* AS de $(P?Q)$ qui donne une instance de `ans`(.) n'apparaissant pas dans ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$). Nous savons que ($\mathcal{F}$, $(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$) possède toutes les règles dont Q dépend ($(\mathcal{R}\downarrow Q)\cup règles(D\mathcal{Q}(\mathcal{R},Q))$ contient $(\mathcal{R}\downarrow Q)$) et que toutes les instances de `ans` possibles seront calculées. Nous déduisons qu'il existe un ensemble d'atomes

équivalent à AS sur ($\mathcal{F}$, $(\mathcal{R} \downarrow Q) \cup règles(D\mathcal{Q}(\mathcal{R}, Q))$), qui est un *answer set* à l'ajout des instances provenant des règles appartenant seulement à $(P?Q)$ près. Étant donné que ($\mathcal{F}$, $(\mathcal{R} \downarrow Q) \cup règles(D\mathcal{Q}(\mathcal{R}, Q))$) possède toutes les règles dangereuses pouvant empêcher un ensemble d'atomes d'être un *answer set* les réponses obtenues sur ($\mathcal{F}$, $(\mathcal{R} \downarrow Q) \cup règles(D\mathcal{Q}(\mathcal{R}, Q))$) sont équivalentes à celles obtenues sur P. ■