Answer Set Programming et interrogation

Projet ASPIQ

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.*

1. Introduction

Les ontologies sont utilisées pour la représentation et l'interrogation d'informations d'un domaine précis sous forme de réseau sémantique. Une ontologie regroupe un ensemble de concepts, permettant de représenter un domaine, liés entre eux par des relations taxonomiques et sémantiques. Par exemple, si on s'intéresse à l'ontologie académique la plus courante nommée `university` (*Benchmark university*, s. d. ; Guo *et al.*, 2005 ; Kollia *et al.*, 2011 ; Pérez-Urbina *et al.*, 2009), 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 ».

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. Pour ce faire, une ontologie est interrogée à l'aide d'une requête per-

mettant d'extraire des informations qui répondent à certains critères. Nous pouvons, par exemple, dans l'ontologie `university`, poser les interrogations 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 d'un modèle pour pouvoir répondre à une requête. Cet aspect est un atout pour l'interrogation d'ontologies qui sont constituées d'un grand nombre de données mais ne possèdent que très peu de dépendances entre elles, contrairement aux problèmes combinatoires qui possèdent souvent peu de données mais beaucoup de dépendances entre elles. Par exemple, 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 du programme n'est pas nécessaire pour répondre correctement à la requête. Ainsi, la difficulté pratique de la mise en œuvre de l'interrogation est d'isoler le plus petit ensemble de connaissances permettant de répondre correctement à la requête. Le principe de l'interrogation d'ontologie est alors de se servir des informations présentes dans la requête pour calculer uniquement les informations nécessaires pour y répondre.

Du point de vue de la représentation d'ontologies issues du web, c'est le langage OWL, basé sur RDF, qui est recommandé par le W3C. Il a pour avantage d'être simple à interpréter pour une machine grâce à son format balisé, mais il est beaucoup plus difficile pour un humain d'élaborer une ontologie sous ce format à cause de sa syntaxe, et encore plus difficile de comprendre une ontologie en la lisant sous ce format. Néanmoins, cette représentation en OWL peut être ramenée 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). En pratique, il existe des raisonneurs efficaces (Sirin *et al.*, 2007 ; Glimm *et al.*, 2014) sur les ontologies qui permettent d'exécuter des requêtes quel que soit le format initial des données et des traductions entre les différents formats sont disponibles (*OWL2DLGP GRAAL*, s. d.).

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 « Un enseignant-chercheur n'est pas étudiant sauf s'il est doctorant ».

Il est donc intéressant d'utiliser un formalisme dans lequel il est possible d'exprimer des informations utilisant la négation par défaut. Le formalisme de l'Answer Set Programming (ASP) (Gelfond, Lifschitz, 1988) est adéquat pour la prise en compte de telles informations et permet ainsi d'augmenter l'expressivité des ontologies. D'autre part, des travaux ont déjà été menés pour traduire les règles existentielles (et donc certaines logiques de description ou formats de OWL) en ASP (Garreau *et al.*, 2015).

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 appellés answer sets. Notons que 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). Un point important est que, contrairement aux ontologies traditionnelles pour lesquelles il n'y a qu'une réponse possible, un même programme ASP peut avoir aucun, un ou plusieurs answer sets. Ainsi, lors de l'interrogation en ASP, nous devons définir les réponses à une requête en cas d'incohérence (aucun answer set) ou d'alternatives (plusieurs answer sets). D'autre part, en pratique, l'isolation des règles doit aussi prendre en compte ces particularités. En ASP, à notre connaissance, seuls les travaux concernant le solveur `DLV` traitent de l'interrogation. Cependant, ceux-ci ne permettent pas de traiter tous les programmes mais ne concernent que des catégories particulières : programmes avec variables existentielles sans négation par défaut (Alviano *et al.*, 2012) ou programmes ASP possédant au moins un modèle (Alviano, Faber, 2011), quels que soient les faits initiaux, appelés super-consistants (Alviano *et al.*, 2014). Par ailleurs, notons que le raisonneur `NoHR` (Costa *et al.*, 2015) est le seul qui accepte les ontologies avec défaut mais dans un cadre restreint.

Dans cet article, nous présentons une étude algorithmique et pratique de la mise en œuvre de l'interrogation en ASP. 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'utiliser 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 étudions 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 au traitement des ontologies traditionnelles et permet en outre une plus grande expressivité.

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 logique normal* (ou plus simplement un *programme*) est un couple $(\mathcal{F}, \mathcal{R})$ avec $\mathcal{R}$ un ensemble de règles r 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]

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".

Pour une règle r (ou par extension un ensemble de règles), on utilise les notations suivantes : $t\hat{e}te(r) = c$ 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 $r^+ = t\hat{e}te(r) \leftarrow corps^+(r)$.

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 n'apparaissant nulle part ailleurs. En outre, on dit qu'une règle r est *monotone* (respectivement *non monotone*) si son corps négatif est vide (respectivement non vide).

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).

Un programme logique *défini* $P = (\mathcal{F}, \mathcal{R})$ est un ensemble de faits $\mathcal{F}$ et de règles monotones $\mathcal{R}$. Un programme logique 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 constantes (respectivement de fonctions et de prédicats) 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 et aux atomes. 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) t, noté $\sigma(t)$ est le terme (respectivement l'atome) 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.

Le résultat de l'application d'une substitution $\sigma : \mathcal{V} \rightarrow \mathcal{U}$, constituée de couples $X_i \mapsto t_i$, à une règle r, noté $\sigma(r)$ est la règle r dont toutes les occurrences des X_i ont été remplacées simultanément par les t_i. On dit alors que $\sigma(r)$ est une *instance* de r.

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 $t\hat{e}te(\sigma(r)) \in X$. 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 logique 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 :

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

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 logique normal P par rapport à un ensemble d'atomes $X \subseteq \mathcal{A}$ est le programme logique 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 :

$$P^X = \left(\begin{array}{rl} \mathcal{F} \cup \{t\hat{e}te(\sigma(r)). \mid & r \in \mathcal{R},\ \sigma \text{ une substitution}, \\ & \sigma(corps^-(r)) \cap X = \emptyset, corps^+(r) = \emptyset\}, \\ \{\sigma(r^+) \mid & r \in \mathcal{R},\ \sigma \text{ une substitution}, \\ & \sigma(corps^-(r)) \cap X = \emptyset, corps^+(r) \neq \emptyset\} \end{array} \right)$$

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 set*. 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 set* 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 set* 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 set* du programme P' (et donc par conséquent, les *answer set* 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 efficace dans la plupart des cas, 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 set*. 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

Afin de représenter graphiquement les connaissances d'un programme à base de règles, il est commun d'utiliser une représentation sous forme de graphe. 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. Il existe pour cela principalement deux types de graphes : le *graphe de dépendance des règles* (Konczak *et al.*, 2006) et le *graphe de dépendance des prédicats* (Apt, Bol, 1994). 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 $(A_1, \ldots, A_n)$ avec $A_1 = (S_1, S'_1)$ et $A_n = (S_n, S'_n)$ tel que $S'_n = S_1$.

2.2.1. Graphe de dépendance des symboles de prédicat

Le *graphe de dépendance des symboles de prédicat* (Apt, Bol, 1994) d'un programme est un graphe orienté avec ses sommets étiquetés par les symboles de prédicat d'un programme et dont les arcs représentent les dépendances entre symboles de prédicat. Soit r une règle, un symbole de prédicat p dépend d'un symbole de prédicat q si 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. Ces nouvelles dépendances vont ajouter de l'expressivité au graphe précédent. Soit *r* une règle, un symbole de prédicat p *dépend positivement* (resp. *négativement*) d'un symbole de prédicat q si 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. La fonction $gdsp$ est telle que $gdsp(\mathcal{R})$ calcule a partir d'une ensemble de règles le graphe de dépendance des symboles de prédicat de l'ensemble de règles $\mathcal{R}$.

2.2.2. Graphe de dépendance des règles

Le *graphe de dépendance des règles* (Konczak *et al.*, 2006) d'un programme est un graphe orienté dont les sommets sont étiquetés par les règles du programme et dont les arcs représentent les dépendances entre les règles. Soit r_1 et r_2 deux règles d'un programme, r_2 dépend de r_1 si $tête(r_1)$ est unifiable avec un atome de $corps(r_2)$. Dans le graphe apparaît un arc allant de la règle r_1 vers la règle r_2. Nous définissons la *dépendance indirecte de règle* comme étant la clôture transitive de la dépendance (directe) entre règles définie ci-dessus.

3. Interrogation en ASP

L'interrogation a pour but de questionner sur l'existence d'un élément, ou calculer l'ensemble des éléments, vérifiant une requête. La représentation utilisée pour les re-

quêtes ASP est la même que celle utilisée pour DATALOG (Gallaire *et al.*, 1984 ; Ceri *et al.*, 1989). Une requête est donc représentée sous forme de règle afin de l'intégrer à l'ensemble des règles du programme et permettre par la suite de faire abstraction des faits pour certains traitements.

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

$$\mathtt{ans}(\mathtt{X_1},\ \ldots,\ \mathtt{X_k}) \leftarrow \mathtt{a_1},\ \ldots,\ \mathtt{a_n},\ \mathtt{not}\ \mathtt{b_1},\ \ldots,\ \mathtt{b_m}.$$

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,* ans *un symbole de prédicat n'apparaissant pas dans le programme ni dans le corps de la requête, appelé* prédicat réponse*, et* $\mathtt{X_1},\ \ldots,\ \mathtt{X_k}$ *des variables apparaissant dans au moins un atome* $\mathtt{a_i}$*. 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 la requête Q à l'ensemble des règles $\mathcal{R}$ d'un programme P.

Soit $P = (\mathcal{F}, \mathcal{R})$ un programme ASP, nous notons $(P?Q) = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ le programme P avec $\mathcal{F}$ la base de faits, $\mathcal{R}$ l'ensemble des règles et Q la requête sur le programme P.

Nous obtenons aisément le résultat suivant qui met en correspondance les *answer set* d'un programme P et les *answer set* de P requêté par Q : si $AS_1,\ \ldots,\ AS_n$ sont les *answer set* de P alors les ensembles $AS_1^Q,\ \ldots,\ AS_n^Q$ tels que

$$AS_i^Q = AS_i \cup \{t\hat{e}te(\sigma(Q)) \mid corps^+(\sigma(Q)) \subseteq AS_i\}$$

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

Pour traiter le problème de l'interrogation nous allons utiliser dans les exemples un programme basé sur l'ontologie university mais simplifiée avec seulement 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 permettant pas de mettre en valeur les problèmes que nous souhaitons souligner, et ne comportant pas d'exception, nous avons 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 étant d'arité 1.*

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

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

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

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

Le programme P_2 *possède deux answer set*

$$\begin{array}{ll} AS_1 = \{ & \mathtt{ad(marie), dir(marie), pU(jean), dir(jean), et(jean), cA(marie),} \\ & \mathtt{ad(jean), cA(jean)}\} \\ AS_2 = \{ & \mathtt{ad(marie), dir(marie), pU(jean), dir(jean), et(jean), cA(marie),} \\ & \mathtt{eC(jean), etT(jean), chr(jean)}\} \end{array}$$

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

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 set*. 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 set*. 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 ($\mathcal{AS} = \{\emptyset\}$). Le premier correspond à un programme n'ayant pas d'*answer set* tandis que le second correspond à un programme dont l'unique *answer set* ne contient pas d'atome. Nous proposons ici de considérer la réponse à un programme inconsistant comme absurde. La crédibilité d'une réponse à un programme inconsistant étant discutable car l'inconsistance provient d'un conflit dans les faits ou dans les règles du programme. Une interrogation étant posée sur un modèle s'il n'existe pas de modèle alors une réponse est absurde car le programme n'admet pas d'*answer set*.

Nous distinguons deux types de requêtes, nous définissons dans un premier temps les requêtes conjonctives classiques (non-booléennes) dont la réponse est un ensemble d'atomes, puis 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 : $(\mathtt{ans(X_1,\ \ldots,\ X_k)} \leftarrow \mathtt{a_1,\ \ldots,\ a_n.})$ *avec* $k > 0$ *et* $n > 0$, *une requête conjonctive. Soit* $\mathcal{AS}$ *l'ensemble des answer set de* $(P?Q)$. *La* réponse *à*

Q dans P[2] est `absurde` *si $\mathcal{AS} = \emptyset$ sinon la réponse est* valide *et est l'ensemble des substitutions σ telles que :*

– $\forall AS \in \mathcal{AS},\ \sigma(\texttt{ans(X}_1,\ \ldots,\ \texttt{X}_\texttt{k}\texttt{))} \in AS$ *pour la* réponse sceptique*,*

– $\exists AS \in \mathcal{AS},\ \sigma(\texttt{ans(X}_1,\ \ldots,\ \texttt{X}_\texttt{k}\texttt{))} \in AS$ *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* ans(marie) *appartient aux deux answer set. 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 : $(\texttt{ans} \leftarrow a_1,\ \ldots,\ a_n.)$ avec $n > 0$, une requête booléenne conjonctive et $\mathcal{AS}$ l'ensemble des answer set de $(P?Q)$.*
Si $\mathcal{AS} = \emptyset$ alors la réponse à Q dans P[3] est `absurde`*,*
sinon la réponse est valide *et :*

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

– *pour la réponse crédule :*
  - *si $\exists AS \in \mathcal{AS}$,* 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 : $(\texttt{ans} \leftarrow \texttt{cA(jean)}.)$. Les deux answer set de $(P_2?Q_6)$ sont les suivants :*

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

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 - Manuel d'utilisation*, s. d.) 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 erronées. 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` car ans appartient bien à tous les modèles existants. A contra-

2. ou la réponse dans le programme $(P?Q)$
3. ou la réponse dans le programme $(P?Q)$

rio la réponse crédule est faux en l'absence de modèle car il n'existe pas d'*answer set*, et donc pas d'*answer set* contenant ans. En considérant la réponse absurde dans le cas où le programme est inconsistant il est facile de différencier la réponse vide (il existe un *answer set* mais aucune réponse n'est présente dans un/tous les *answer set*) 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 set* 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. Nous souhaitons interroger ces programmes et la méthode la plus simple pour répondre à une requête est de calculer l'ensemble des *answer set* puis de chercher les réponses à notre requête directement dans les *answer set* calculés. Il existe néanmoins un inconvénient à l'interrogation de programmes ASP, c'est que le temps de calcul sera au minimum aussi long que la recherche de l'ensemble des *answer set* et dans le cas d'*answer set* 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 set* et de réduire au maximum le nombre de règles et d'instances nécessaires pour obtenir une réponse. Mais avant cela il faudra 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. Le problème est que lors de l'interrogation d'un programme, il faut s'assurer que celui-ci possède au moins un *answer set*, dans le cas contraire la réponse serait absurde. Dans un premier temps nous souhaitons nous assurer qu'un programme n'est pas inconsistant en vérifiant qu'il existe au moins un *answer set* à notre programme 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 [4]. 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 set*. Cette

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

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 donc 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) avec l'aide des *magic sets* (Faber *et al.*, 2007) 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 vérifier si un programme est inconsistant en limitant le nombre de calcul 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 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.*

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

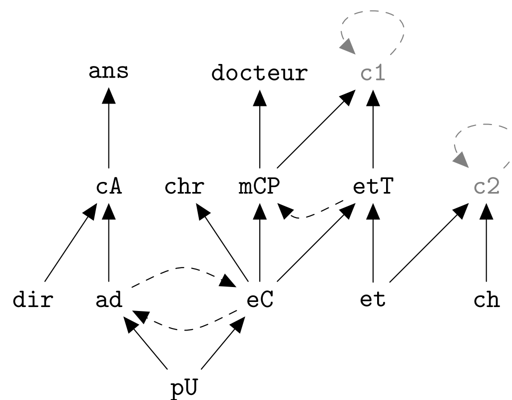

Figure 1. Graphe de dépendance des symboles de prédicat pour le programme de l'exemple 2

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. Dans cet exemple*

les 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 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* $\text{tête}(r)$.

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.

EXEMPLE 10. — *Reprenons le programme* $P_2 = (\mathcal{F}_2, \mathcal{R}_2)$ *de l'exemple 2 avec la requête* Q_{10} : $(\texttt{ans} \leftarrow \texttt{c1}.)$.

Nous pouvons observer que le seul moyen d'obtenir `ans` *dans un answer set est d'appliquer* Q_{10} *qui a besoin d'une instance de* `c1`*. Nous pouvons voir aussi que* `ans` *ne peut déclencher aucune autre règle.* `c1` *ne peut pas appartenir à un answer set car le programme serait inconsistant. Donc si* P_2 *est consistant alors* $(P_2?Q_{10})$ *est aussi consistant.*

Si nous ajoutons `mCP(jean)` *et* `eC(jean)` *à notre base de faits pour obtenir le programme* $P_{10} = (\mathcal{F}_2 \cup \{\texttt{mCP(jean)}, \texttt{eC(jean)}\}, \mathcal{R}_2)$ *alors nous avons* P_{10} *et* $(P_{10}?Q_{10})$ *inconsistants car nous déduisons toujours* `etT(jean)` *ce qui déclenche la règle* r_7 [5] *et* `ans` *ne peut pas rétablir la consistance car il n'apparaît dans aucun corps de règle.*

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 en positives et négatives 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 11 (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 :*

5. r_7 = `c1 ← mCP(X), etT(X), not c1.`

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

– *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 ;*

– *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 pour déterminer si le programme est inconsistant. 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 7.2 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 12. — *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*, représentés par les règles dont le symbole de prédicat en tête fait partie des symboles de prédicat entourés par une forme rouge : 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*

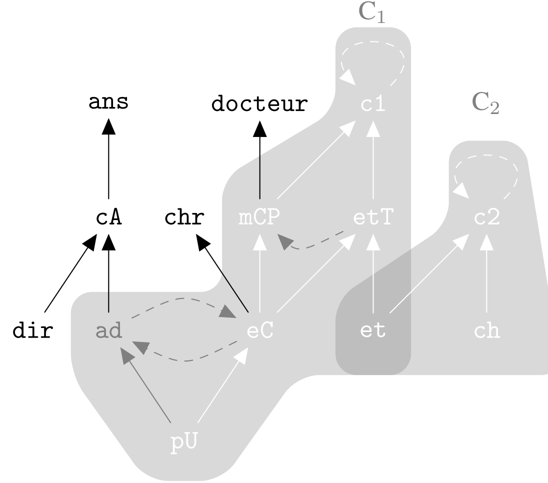

Figure 2. Graphe de dépendance des symboles de prédicat, cycles d'inconsistance et règles dangereuses pour le programme de l'exemple 2

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 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* r_3, r_6, r_9 *et* Q_2 *ne sont pas dangereuses.*

L'algorithme 1 réalise la fonction $ensembleR\grave{e}glesDangereuses$ qui est telle que $ensembleR\grave{e}glesDangereuses(\mathcal{R})$ calcule l'ensemble des ensembles des règles 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. Pour chaque cycle, nous détectons tout d'abord l'en-

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

semble 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 d'un graphe de dépendance des symboles de prédicat $\mathcal{G}$ 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. $règlesPositives(D)$, $règlesNegatives(D)$) 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 13 (Complétude et correction de l'algorithme 1). — *L'algorithme 1* $ensembleRèglesDangereuses$ *calcule l'ensemble des règles dangereuses, comme défini dans la définition 11.*

EXEMPLE 14. — *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 symbole de prédicat dangereux positifs ce qui ajoute* r_4 *et* r_5 *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* `et` *dépendent positivement ce qui fait que* r_2 *est dangereuse positive et* `pU` *est un fait donc n'engendre pas de règle dangereuse. 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*

Algorithme 1 : $ensembleR\grave{e}glesDangereuses$

Data : Un ensemble $\mathcal{R}$ de règles
Result : L'ensemble $\mathcal{D}$ des ensembles de règles dangereuses marquées

```
G = gdsp(R);
while D := CycleInconsistant(G, R) ≠ ∅ do
    foreach r ∈ règles(D) do
        foreach p ∈ pred(corps⁺(r)) do
            D := D ∪ dangereuses(p, R, +);
        end
    end
    foreach r ∈ règlesPositives(D) do
        foreach p ∈ pred(corps⁻(r)) do
            D := D ∪ dangereuses(p, R, −);
        end
    end
    foreach r ∈ règlesNegatives(D) do
        foreach p ∈ pred(corps(r)) do
            D := D ∪ dangereuses(p, R, −);
        end
    end
    𝒟 := 𝒟 ∪ {D};
end
return 𝒟
```

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'inconsistances ont été parcouru, nous avons alors l'ensemble des ensembles de règles dangereuses de* $\mathcal{R}_2, \mathcal{D}_2 = ensembleR\grave{e}glesDangereuses(\mathcal{R}_2) = \{D_1, D_2\}$.

Afin de différencier les règles dangereuses positives et les règles dangereuses négatives permettant ensemble la détection de l'inconsistance, nous montrons dans les exemples 15 et 16 l'influence de chacune d'entre elles sur le calcul d'un *answer set*.

EXEMPLE 15. — *Soit le programme* $P_{15} = (\mathcal{F}_{15}, \mathcal{R}_2)$*, l'ensemble de règles de l'exemple 2 avec la base de faits* $\mathcal{F}_{15} = \{\texttt{eC(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}\}$.

Nous observons que les règles r_5 [12] *et* r_7 [13] *seules permettent d'avoir un programme inconsistant et que c'est le déclenchement de* r_7 *qui est à l'origine de l'inconsistance. Si nous choisissons maintenant* $P'_{15} = (\mathcal{F}'_{15}, \mathcal{R}_2)$ *avec* $\mathcal{F}'_{15} = \{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{ad(jean)}\}$*, nous obtenons cette fois-ci un answer set* $\{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{ad(jean)}, \texttt{docteur(jean)}\}$*. Ici nous obtenons* `ad` *qui est un symbole de prédicat dangereux négatif qui permet d'empêcher l'inconsistance provoquée par le symbole de prédicat* `eC`*. De même si nous choisissons* $\{\texttt{eC(jean)}, \texttt{etT(jean)}\}$ *l'inconsistance est empêchée car la règle* r_4 [14] *ne peut pas être déclenchée et empêche donc le déclenchement de* r_7 *responsable de l'inconsistance.*

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 permettre d'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 16. — *Soit le programme* $P_{16} = (\mathcal{F}_{16}, \mathcal{R}_{16})$ *avec* $\mathcal{R}_{16} =$

$$\left\{ \begin{array}{ll} r_3 : \texttt{cA(X)} \leftarrow \texttt{ad(X), dir(X).}, & r_4 : \texttt{mCP(X)} \leftarrow \texttt{eC(X), not etT(X).}, \\ r_5 : \texttt{etT(X)} \leftarrow \texttt{eC(X), et(X).}, & r_7 : \texttt{c1} \leftarrow \texttt{mCP(X), etT(X), not c1.}, \\ r_8 : \texttt{c2} \leftarrow \texttt{et(X), ch(X), not c2.}, & r_9 : \texttt{docteur(X)} \leftarrow \texttt{mCP(X).}, \\ r_{10} : \texttt{ad(X)} \leftarrow \texttt{pU(X), biatss(X).}, & r_{11} : \texttt{eC(X)} \leftarrow \texttt{pU(X), not prag(X).}, \\ r_{12} : \texttt{prag(X)} \leftarrow \texttt{pU(X), noncher cheur(X), not ad(X).} & \end{array} \right\}$$

et $\mathcal{F}_{16} = \{\texttt{pU(jean)}, \texttt{nonchercheur(jean)}, \texttt{eC(jean)}, \texttt{et(jean)}\}$*. 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, seulement elle est bloquée par l'application de la règle dangereuse négative* r_{12} *donc le programme reste consistant et nous avons l'answer set* $\{\texttt{pU(jean)}, \texttt{nonchercheur(jean)}, \texttt{eC(jean)}, \texttt{mCP(jean)}, \texttt{docteur(jean)}\}$*. Si nous ajoutons maintenant* `biatss(jean)`*, le programme devient inconsistant car la règle dangereuse négative* r_{10} *empêche l'application de la règle dangereuse négative* r_{11} *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

12. $r_5 = \texttt{etT(X)} \leftarrow \texttt{eC(X), et(X).}$
13. $r_7 = \texttt{c1} \leftarrow \texttt{mCP(X), etT(X), not c1.}$
14. $r_4 = \texttt{mCP(X)} \leftarrow \texttt{eC(X), not etT(X).}$

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 17. — *Soit P_{17} un programme constitué de l'ensemble de règles*

$$\mathcal{R}_{17} = \left\{ \begin{array}{ll} \rho_1 : \texttt{t(X)} \leftarrow \texttt{b(X), not t(X).}, & \rho_2 : \texttt{s(X)} \leftarrow \texttt{d(X), not s(X).}, \\ \rho_3 : \texttt{b(X)} \leftarrow \texttt{a(X), not d(X).}, & \rho_4 : \texttt{d(X)} \leftarrow \texttt{c(X), not b(X).} \end{array} \right\}$$

et la base de faits $\mathcal{F}_{17} = \{\texttt{a(1)}, \texttt{c(1)}\}$. 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 set pour chacun des ensembles séparément nous obtenons un answer set $AS_1 = \{\texttt{a(1)}, \texttt{c(1)}, \texttt{d(1)}\}$ pour D_1 et un answer set $AS_2 = \{\texttt{a(1)}, \texttt{c(1)}, \texttt{b(1)}\}$ pour D_2. Le problème est que si nous calculons les answer set de P_{17}, 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 $unionEnsembleR\grave{e}glesDangereuses$ qui est telle que $unionEnsembleR\grave{e}glesDangereuses(\mathcal{R})$ 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.

Algorithme 2 : $unionEnsembleR\grave{e}glesDangereuses$

```
Data : Un ensemble R de règles
Result : L'ensemble des ensembles unifiés de règles dangereuses
Δ = ensembleRèglesDangereuses(R);
foreach D_i ∈ Δ do
    foreach D_j ∈ Δ\D_i do
        if règles(D_i) ∩ règles(D_j) ≠ ∅ then
            D_i := (D_i unionRegle D_j);
            Δ := Δ\D_j;
        end
    end
end
return Δ;
```

THÉORÈME 18 (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.*

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 18, 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 nécessaires à l'obtention d'une réponse à notre 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 une 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.

EXEMPLE 19. — *Soit l'ensemble de règles de l'exemple 2 et sa requête* $Q_2 = (\mathtt{ans(X)} \leftarrow \mathtt{cA(X).})$*. Nous obtenons* $(\mathcal{R}_2 \downarrow Q_2) = \{Q_2, r_3, r_2, r_1\}$*,*

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 $(\mathcal{F}, (\mathcal{R} \downarrow Q))$. Nous montrons aussi la réciproque 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.

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 $\mathcal{Q}$;

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 20 (Réponse à un programme super-consistant). — *Soit $P = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ un programme ASP super-consistant requêté par Q. Si R est une réponse à Q dans P alors R est une réponse à Q dans $(\mathcal{F}, (\mathcal{R}\downarrow Q))$.*

Nous illustrons dans l'exemple 21 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 21. — *Soit le programme ASP super-consistant $P_{21} = (\emptyset, \mathcal{R}_{21})$ (i.e. $\mathcal{R}_2$ sans les contraintes) avec $\mathcal{R}_{21}$ =*

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

et la requête $Q_2 = (\texttt{ans(X)} \leftarrow \texttt{cA(X).})$. Nous 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, soit uniquement* `ad` *et* `dir`*, seul* `ad` *apparaît en tête d'une règle donc nous ajoutons la règle r_1 dans laquelle il apparaît en tête. Nous continuons avec les symboles de prédicat du corps de r_1 nous ajoutons ainsi r_2. Nous avons donc $(\mathcal{R}_{21}\downarrow Q_2) = \mathcal{Q} = \{Q_2, r_3, r_1, r_2\}$. Nous avons calculé ainsi toutes les 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 $(\mathcal{R}_{21}\downarrow Q_2)$. Les règles r_4, r_5, r_6 et r_9 ne sont donc pas nécessaires pour trouver une réponse à $(P_{21}?Q_2)$.*

Nous savons maintenant que lorsque notre programme est super-consistant seul 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 $\mathcal{R}$, 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.

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 $\mathcal{F}_{22} = \{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}\}$*, une base de faits. Nous posons la requête* Q_{22} : $(\texttt{ans} \leftarrow \texttt{travaildur(X)}.)$. *Nous obtenons* $(\mathcal{R}_{22} \downarrow Q_{22}) = \{Q_{22}, r_{13}, r_5, r_2, r_1\}$*,* $D\mathcal{Q}(\mathcal{R}_{22}, Q_{22}) = \{(r_7, +), (r_5, +), (r_4, +), (r_2, +), (r_1, -)\}$ *et* $D_2 = \{(r_8, +)\}$ *un ensemble de règles dangereuses qui n'intersectent pas de 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 un unique answer set pour* $(\mathcal{F}_{22}, règles(D\mathcal{Q}(\mathcal{R}_{22}, \mathcal{R}_{22})))$ *qui est :* $\{\texttt{pU(jean)}, \texttt{mCP(jean)}, \texttt{et(jean)}, \texttt{ad(jean)}\}$

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* `pU(jean)` *ne permettant par 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* `pU(jean)` *qui bloque ainsi la règle r_2. Si nous calculons les answer set 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 par contre `vrai` *car* `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, le modèle AS'_2 ne mène donc pas à un answer set dans le programme complet et ne devrait pas être pris en compte pour calculer la réponse au programme. 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.*

THÉORÈME 23 (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 $(\mathcal{F}, (\mathcal{R} \downarrow Q) \cup règles(D\mathcal{Q}(\mathcal{R}, Q)))$.*

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 set* complets sur cet ensemble. Nous présenterons plusieurs perspectives permettant de réduire le nombre d'instanciations nécessaire pour répondre à une requête : dans un premier temps en s'intéressant aux instances nécessaires pour l'ensemble des règles dont la requête dépend à l'aide des constantes présentes dans celle-ci, et dans un deuxième temps en s'intéressant aux ensembles de règles dangereuses dont l'instanciation est nécessaire pour valider une réponse. Avant cela, nous présentons dans la section suivante l'implémentation actuelle de l'interrogation en ASP que nous avons réalisée.

6. Résultats des implémentations

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

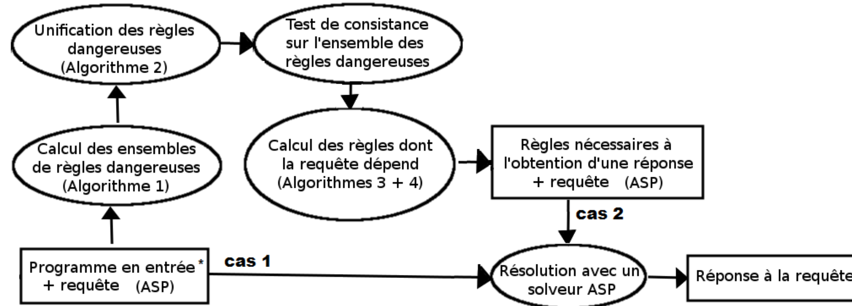

Figure 3. Processus d'interrogation

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 vue dans les sections précédentes (cas 2).

En pratique, l'isolation des règles est implémentée [15] dans le solveur ASPeRiX (Lefèvre *et al.*, 2017) et les solveurs utilisés sont ASPeRiX, DLV et Clasp. Nous avons comparé cette isolation avec les performances de NoHR (Costa *et al.*, 2015), un raisonneur pour ontologies avec défauts fonctionnant comme plug-in sur Protégé, et celles de l'interrogation proposée par le solveur DLV (Faber *et al.*, 2007). Enfin, dans le cas où les informations décrites sont des ontologies sans négation par défaut, nous avons comparé les performances des solveurs ASP avec le raisonneur Pellet. Nous appellerons ASP_Cas1 l'interrogation avec les solveurs ASP sur le programme initial, ASP_Cas2 l'interrogation avec les solveurs ASP avec isolation des règles, DLV_$Cas1$ l'interrogation avec DLV sans *magic sets*, DLV_$Cas2$ l'interrogation avec DLV avec *magic sets*.

ASPERIX. Le solveur ASPeRiX est utilisé dans sa version 0.2.6 avec module d'interrogation mais sans isolation des règles. Les *answer set* sont calculés entièrement et nous avons le choix entre une réponse crédule ou sceptique à l'aide des options −QS et −QC. Le temps calculé ne prend pas en compte le temps de *parsing* mais seulement le temps nécessaire au calcul des *answer set* et de la réponse à la requête.

ASPERIX Q. Le solveur ASPeRiX est aussi utilisé dans sa version 0.2.6 mais avec un module d'interrogation comprenant l'isolation des règles proposée dans cet article. Les *answer set* sont calculés seulement sur les règles isolées. Les options, la forme de la requête et le résultat obtenu sont les mêmes que pour ASPeRiX. Comme pour ASPeRiX, le temps de parsing n'est pas pris en compte, seuls le temps de calcul de l'isolation des règles et de l'interrogation du sous-programme sont comptabilisés.

15. L'implémentation et les tests sont disponibles sur http://forge.info.univ-angers.fr/~fgarreau/RIA17.php

DLV. Le solveur DLV est utilisé dans sa version du 17 décembre 2012 avec module d'interrogation sans optimisation. Les *answer set* sont calculés entièrement 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 pour DLV 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.

DLV MS. Utilisation du solveur DLV en version du 17 décembre 2012 avec module d'interrogation utilisant les *magic set* avec l'option -OMS. Les *answer set* sont calculés partiellement à l'aide des *magic set*. Les options, la forme de la requête et le résultat obtenu sont les mêmes que pour DLV. Les temps affichés pour DLV MS sont calculés de la même manière que DLV simple.

CLASP. Le solveur Clasp est utilisé dans la version 4.5.4. de clingo. Clasp ne propose pas d'interrogation, nous proposons alors de calculer entièrement les *answer set* à l'aide des options -brave et -cautious pour obtenir les *answer set* crédules et sceptiques puis d'afficher uniquement les atomes réponses en tête des règles désignées comme des requêtes. La requête est de la même forme que pour ASPeRiX en y ajoutant la directive

```
#show ans/1.
```

, celle-ci permettant de désigner dans les *answer set* les atomes à afficher uniquement [16]. Le résultat obtenu est l'ensemble des instances de l'atome réponse. Plusieurs requêtes sont autorisées pour un programme. Les temps de Clasp sont obtenus avec la commande time UNIX et comprend donc le temps de parsing et le calcul de l'ensemble des *answer set*.

CLASP + Q. Le solveur Clasp est utilisé dans la version 4.5.4. de clingo. Clasp en utilisant en pré-traitement l'isolation des règles avec ASPeRiX proposée dans cet article. Ainsi seul le programme d'entrée fourni à Clasp est différent, ne comprenant que les règles nécessaires à la requête après le pré-traitement effectué par ASPeRiX. Comme pour la version utilisant le programme complet le temps correspond au temps nécessaire pour le parsing et le calcul de tous les *answer set* du sous-programme. Pour obtenir le temps complet il faut ajouter le temps nécessaire à l'isolation des règles.

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 obtenu par le raisonneur correspond uniquement à l'interrogation après avoir charger l'ontologie en mémoire. Pour obtenir le temps complet il faut y ajouter le temps de pré-traitement de l'ontologie.

NOHR Le raisonneur NoHR est utilisé dans sa version 3.0. Le raisonneur ne permet pas de choisir une réponse crédule ou sceptique mais propose un résultat proche du crédule lorsqu'il existe plusieurs *answer set*, cependant le bon fonctionnement n'est

16. /1 correspondant à l'arité du prédicat de l'atome réponse

assuré que pour les programmes stratifiés. Le temps obtenu par le raisonneur correspond uniquement à l'interrogation après avoir charger l'ontologie en mémoire. Pour obtenir le temps complet il faut y ajouter le temps de pré-traitement de l'ontologie et du chargement dans le module XSB [17].

6.1. Benchmark `university` classique

`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). L'ontologie est définie en anglais, ainsi tous les tests effectués à partir de `university` seront présentés en anglais et ceux ne dépendant pas de l'ontologie en français. 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 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, 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>`: définit l'url de l'ontologie contenant les règles (dans notre cas la TBOX)

17. Permettant de simplifier la marche arrière utilisée pour l'interrogation

Les données générées dans le rapport sont générées avec UBA pour 1, 2, 5, 10 et 20 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 15 et 25 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 'a 25 en fonction du nombre de ABOX générées. Pour une université générée avec les paramètres par défaut, il y a 15 départements donc 15 ABOX générées allant de `University0_0.owl` à `University0_14.owl`.

LES REQUÊTES. Le benchmark `university` est proposé avec 14 requêtes conjonctives non-booléennes. Ces requêtes sont au format SPARQL mais peuvent être rapidement traduite en ASP.

Certaines d'entre elles ont été réordonnées pour permettre d'obtenir une réponse dans un temps raisonnable avec le solveur `ASPeRiX` qui ne propose pas de réordonner automatiquement les règles contrairement aux autres solveurs. Les prédicats binaires des requêtes sont placés en premier pour limiter le nombre d'instanciations testées.

Une seconde version de `university` simplifiée est proposée au format OWL2 $\mathcal{E}$L. Ces versions sont couramment utilisées pour comparer les raisonneurs et solveurs (Costa *et al.*, 2015 ; Leone *et al.*, 2012). L'intérêt est de simplifier le benchmark pour gagner du temps sur l'interrogation tout en gardant une cohérence dans les règles. Le formalisme OWL2 $\mathcal{E}$L étant plus simple à traiter et possédant tout de même un fort potentiel de représentation de l'information. Seule la TBOX est modifiée comprenant moins de règles ne conservant que celles qui correspondent au format OWL2 $\mathcal{E}$L. Cette version de `university` peut être trouvée directement au format OWL sur le site du projet `GRAAL` (*Benchmark university GRAAL*, s. d.) ou encore au format ASP sur le site de `DLV` (*Benchmark university DLV*, s. d.).

Nous proposons maintenant la comparaison des solveurs sur la version `GRAAL` du benchmark `university`, sans négation par défaut, avec les différents protocoles.

Les tableaux de 1 à 6 présentent les temps de réponse pour 1 université sur la version `GRAAL` (tableau 1) et sur la version `LUBM` (tableau 2), pour 2 universités sur la version `GRAAL` (tableau 3) et sur la version `LUBM` (tableau 4), pour 10 universités sur la version `GRAAL` (tableau 5) et sur la version `LUBM` (tableau 6)

On peut constater sur le benchmark `GRAAL` contenant une seule université 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. requête q14 avec `Clasp` + Q et `ASPeRiX` Q). Les requêtes bénéficiant des meilleures améliorations de temps correspondent aux requêtes contenant peu d'atomes

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

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14
`ASPeRiX`	1.2	0.1	1.2	0.1	1.3	0.7	0.7	1.6	0.6	4	0.6	0.6	0.2	0.7
`ASPeRiX` Q	1.2	0.1	0.6	0.1	1.1	<0.1	0.1	1.4	0.1	3.3	<0.1	<0.1	0.1	0.1
`DLV`	0.4	0.5	0.5	0.4	0.4	0.5	0.4	0.5	0.5	0.4	0.5	0.5	0.4	0.5
`DLV` MS	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.5	0.4	0.4	0.4	0.5	0.4	0.44
`Clasp`	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8
`Clasp` + Q	0.2	0.2	0.12	0.3	0.3	<0.1	0.3	0.3	0.3	0.2	<0.1	<0.1	0.3	<0.1
`NoHR`	-	<0.1	<0.1	<0.1	0.2	0.5	<0.1	0.5	0.3	<0.1	-	-	<0.1	0.6

dans le corps de celle-ci et souvent un nombre de règles très réduit lorsqu'on isole les règles dont la requête dépend.

Tableau 2. Temps de réponse en secondes pour 1 université sur la version `LUBM`.

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14
`DLV`	0.4	0.7	0.4	1.0	1.3	0.7	1.2	2.0	0.7	1.0	0.4	1.0	0.9	0.7
`DLV` MS	0.4	0.7	0.4	1.0	1.3	0.7	1.2	2.0	0.7	1.0	0.4	1.0	0.9	0.4
`Clasp`	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
`Clasp` + Q	0.7	0.7	0.12	0.9	0.7	0.7	0.7	0.7	0.7	0.9	<0.1	0.7	0.7	<0.1

Sur le benchmark `LUBM`, `ASPeRiX` ne parvient pas à obtenir une réponse en un temps convenable à cause de certaines règles pour lesquelles il n'est pas optimisé. Pour les autres solveurs nous pouvons faire le même constat que pour la version `GRAAL` avec des temps légèrement plus longs.

En nous intéressant à la comparaison des solveurs sur la version `GRAAL` du benchmark `university`, sans négation par défaut, avec les différents protocoles, nous avons constaté des temps raisonnables avec les solveurs ASP pour l'obtention d'une réponse sur l'ensemble des requêtes du benchmark avec 10 universités, sauf pour `ASPeRiX` sur les requêtes q1, q3, q5 et q9 (mais ceci est dû au solveur lui-même qui

Tableau 3. Temps de réponse en secondes pour 2 universités sur la version `GRAAL`*.*

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14
`ASPeRiX`	11.7	0.3	7.6	0.3	8	0.1	0.3	6.3	0.2	45	0.1	0.1	1.5	0.1
`ASPeRiX` Q	9.7	0.2	4.8	0.2	6.5	0.1	0.1	4.8	0.1	35	0.1	0.1	0.8	0.1
`DLV`	0.9	1.3	1.3	1.0	1.0	1.3	1.1	1.1	1.2	1.0	1.3	1.3	1.0	1.3
`DLV` MS	0.9	1.1	1.0	1.0	1.0	0.9	1.1	1.1	1.0	0.9	0.9	1.2	1.0	0.9
`Clasp`	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
`Clasp` + Q	0.5	0.4	0.3	0.7	0.7	0.12	0.6	0.6	0.7	0.5	<0.1	0.1	0.7	<0.1
`NoHR`	-	0.2	<0.1	<0.1	0.4	1.3	0.2	2.3	1.8	<0.1	-	-	<0.1	4.2

Tableau 4. Temps de réponse en secondes pour 2 universités sur la version `LUBM`*.*

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14
`DLV`	0.9	1.8	0.9	2.5	3.0	1.9	2.9	4.2	1.9	2.5	0.9	2.5	2.5	1.9
`DLV` MS	0.9	1.8	0.9	2.5	3.0	1.7	2.9	4.2	1.9	2.5	0.9	2.5	2.5	0.9
`Clasp`	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.2
`Clasp` + Q	1.7	1.7	0.3	2.0	1.7	1.7	1.7	1.7	1.7	1.7	<0.1	1.7	1.7	<0.1

souffre d'un manque d'optimisation). Les requêtes bénéficiant des meilleures améliorations de temps avec l'isolation des règles correspondent aux requêtes contenant peu d'atomes, comme la requête q14 avec 12 secondes nécessaires pour le solveur `Clasp` seul et seulement 0,6 secondes pour `Clasp` après isolation (l'isolation nécessitant ici 0,3 secondes). Tandis que `DLV` classique demande 8,2 secondes et avec *magic set* 5,4 secondes. `NoHR` et `Pellet` répondent respectivement en 12 secondes et en 20,5 secondes. Ce résultat est lié au nombre de règles très réduit lorsqu'on isole les règles dont la requête dépend. Sur l'ensemble des requêtes l'amélioration est d'au moins 60% et jusqu'à 95% lors de l'utilisation de l'isolation des règles pour `Clasp`.

Si on compare ASP_Cas1 et ASP_Cas2, l'intérêt de l'isolation des règles est très claire car l'amélioration du temps de calcul est d'au moins 60% et jusqu'à 95%. L'isolation est d'autant plus intéressante que la requête contient moins d'atomes.

Tableau 5. Temps de réponse en secondes pour 10 universités sur la version GRAAL*.*

	q1	q2	q3	q4	q5	q6	q7
ASPeRiX	418	10.1	616	3.1	662	1.2	1.2
ASPeRiX Q	355	8.6	495	2.1	589	0.6	0.6
DLV	5.8	8.8	8.3	5.9	6.0	8.6	6.3
DLV MS	5.6	6.2	5.5	5.8	6.0	6.0	6.3
Clasp	12.0	12.0	12.0	12.0	12.0	12.0	12.0
Clasp + Q	4.3	3.7	2.3	5.7	5.9	1.2	4.0
NoHR	-	1.2	<0.1	<0.1	3.4	10.3	1.2

	q8	q9	q10	q11	q12	q13	q14
ASPeRiX	106	0.9	1479	1.1	0.9	31	1.1
ASPeRiX Q	91	0.3	1239	0.3	0.3	29.5	0.2
DLV	6.1	8.1	5.9	8.2	8.0	5.8	8.2
DLV MS	5.7	6.1	5.6	5.4	7.7	5.7	5.4
Clasp	12.0	12.0	12.0	12.0	12.0	12.0	12.0
Clasp + Q	4.4	4.1	4.2	0.2	0.9	5.8	0.6
NoHR	10.5	8.2	<0.1	-	-	<0.1	16.8

Tableau 6. Temps de réponse en secondes pour 10 universités sur la version LUBM*.*

	q1	q2	q3	q4	q5	q6	q7
DLV	5.6	11.0	6.1	15.5	17.6	11.2	18.0
DLV MS	5.6	10.5	6.0	15.2	17.4	10.7	17.3
Clasp	13.8	13.8	13.8	13.8	13.8	13.8	13.8
Clasp + Q	10.3	10.4	1.5	12.0	10.2	10.6	10.2

	q8	q9	q10	q11	q12	q13	q14
DLV	19.4	12.0	15.4	6.0	16.0	14.5	11.4
DLV MS	18.9	11.2	14.8	5.7	15.1	14.1	6.0
Clasp	13.8	13.8	13.8	13.8	13.8	13.8	13.8
Clasp + Q	10.3	10.3	10.1	<0.1	10.1	10.1	0.6

Les résultats obtenus avec ASP sont globalement comparables à ceux des raisonneurs et meilleurs que ceux obtenus avec DLV. Les raisonneurs 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).

Notons que, dans les versions ASP, il est possible de lancer un ensemble de requêtes en une seule exécution, ceci permettant de réduire le temps de calcul total.

Ces premiers résultats nous indiquent que le traitement de l'interrogation d'une ontologie au format OWL2 $\mathcal{E}$L avec un solveur ASP est comparable aux approches classiques.

Nous proposons maintenant l'ajout de règles par défaut au benchmark `university` afin de tester la démarche présentée dans cet article.

6.2. Benchmark **`university`** *avec règles par défaut*

L'ajout de règles par défaut ne permet plus l'utilisation de raisonneurs classiques pour raisonner sur l'ontologie. 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 causer d'inconsistance au sein du programme. Le raisonneur `NoHR` a été développé dans ce contexte. Dans ce cas, lorsqu'une réponse à la requête est trouvée, au cours de l'algorithme d'interrogation, celle-ci appartiendra forcément à la réponse finale.

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).
```

Afin d'obtenir une réponse à cette requête nous ajoutons aussi les faits suivants :

```
"lowTeachingLoad"(FullProfessor0).
"lowTeachingLoad"(AssistantProfessor0).
"lowTeachingLoad"(AssociateProfessor0).
"lowTeachingLoad"(FullProfessor1).
"onSabbatical"(FullProfessor0).
"ill"(AssistantProfessor0).
"onSabbatical"(FullProfessor1).
"ill"(FullProfessor1).
```

Nous obtenons alors les réponses sceptique et crédule suivantes :

```
[X<-"AssociateProfessor0", Y<-"Department0"]
```

Car seul l'`AssociateProfessor0` n'est ni malade, ni en congé. Les prédicats `onSabbatical` et `ill` n'apparaissent pas dans l'ontologie et le nombre de règles qui dépendent de la requête est très réduit. Le temps de réponse pour la requête est donc instantané avec ASP_Cas2, de même pour `NoHR` et `DLV`. Pour obtenir des

temps comparable il faudrait construire un ensemble de faits cohérent avec le programme original. De plus, même avec une base de faits adaptée, l'ajout de ces règles change peu le traitement effectué sur les ontologies étant donné que le programme reste stratifié et donc la négation par défaut n'ajoute pas vraiment de complexité, pas besoin de tester la consistance du programme ou bien de vérifier qu'une réponse appartient bien à notre *answer set*. Notons que l'interrogation dans un programme stratifié se traite de la même manière qu'un programme sans négation par défaut.

Nous testons maintenant des programmes non-stratifiés. Notons que ces derniers ne sont pas traités par `NoHR`.

Tout d'abord, nous avons utilisé le programme non-stratifié suivant, en ajoutant un premier ensemble de règles par défaut au benchmark `university` avec 20 universités pour obtenir un programme non-stratifié avec 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 $(\mathtt{ans(X)} \leftarrow \mathtt{nonTeacher(X)}.)$ 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 ce premier exemple de programme non-stratifié restent du même ordre de grandeur que pour les interrogations sur les programmes sans négation par défaut. Pour le solveur `Clasp` et 20 universités, il faut 38 secondes sans isolation des règles et 3 secondes une fois les règles isolées. Pour `ASPeRiX` il faut compter 27 secondes sans isolation et 15 secondes avec. Enfin pour `DLV` avec *magic set* une réponse est obtenue en 17 secondes pour le benchmark complet et 1,6 secondes pour les règles isolées.

Une seconde possibilité est d'ajouter des règles par défaut au benchmark `university` avec une seule université mais pour obtenir plusieurs *answer set* sur un programme non-stratifié :

```
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).
```

Les règles ajoutées permettent l'obtention de plusieurs *answer set*, lorsque l'information qu'un cours est `graduate` ou `undergraduate` est inconnue et qu'il existe un étudiant dont on sait qu'il est à la fois `graduate` et `undergraduate`. Dans ce cas pour chaque étudiant étant à la fois `graduate` et `undergraduate` le nombre d'*answer set* est doublé. Cet exemple est non-stratifié à cause du cycle sur les prédicats `graduate` et `undergraduate`. Les résultats pour ASP_Cas2 et DLV_$Cas2$ sont comparables. Le temps de réponse à une requête dépend principalement du nombre d'answer sets qui peut croître exponentiellement.

DLV_$Cas2$ ne mettant pas en œuvre de tests de consistance complet, la justesse de l'interrogation n'est garantie que pour des programmes super-consistants. Son comportement est imprévisible sur des programmes qui ne sont pas super-consistants. Par exemple, lorsque nous considérons l'exemple 22, le solveur DLV_$Cas2$ obtient les bonnes réponses. Cependant, si nous remplaçons la contrainte $(\mathtt{c1} \leftarrow \mathtt{mCP(X)},\ \mathtt{etT(X)},\ \mathtt{not\ c1.})$ par les trois règles dangereuses équivalentes, $(\mathtt{d1} \leftarrow \mathtt{mCP(X)},\ \mathtt{etT(X)},\ \mathtt{not\ d3.})$, $(\mathtt{d2} \leftarrow \mathtt{mCP(X)},\ \mathtt{etT(X)},\ \mathtt{not\ d1.})$ et $(\mathtt{d3} \leftarrow \mathtt{mCP(X)},\ \mathtt{etT(X)},\ \mathtt{not\ d2.})$, DLV_$Cas2$ obtient la réponse `vrai` alors qu'elle devrait être `faux`.

L'ensemble des tests réalisés nous permettent de conclure que ASP_Cas2 offre la plus aboutie des implémentations actuelles tant du point de vue de l'expressivité des ontologies traitées que de l'étendue des programmes traités.

7. Perspectives d'optimisation

Pour améliorer l'efficacité de l'interrogation nous proposons des méthodes encore à l'étude afin de minimiser l'instanciation des règles nécessaires pour répondre à une requête. Nous avons dans un premier temps isolé les parties du programme suffisantes pour répondre à une requête. Maintenant, nous souhaitons instancier le moins d'atomes possible parmi les règles isolées. Pour cela il y a deux grands axes. Le premier axe se porte sur l'ensemble $(\mathcal{R} \downarrow Q)$: pour les requêtes contenant 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 $règles(D\mathcal{Q}(\mathcal{R}, Q))$. Cet ensemble ne peut pas utiliser les constantes présentes dans la requête pour instancier les règles qui le compose ce qui implique d'utiliser d'autres éléments comme la négation par défaut et les règles dangereuses négatives pour réduire le nombre d'instanciations.

7.1. *Instanciation de la requête*

Lors d'une interrogation, les arguments des atomes peuvent être des variables ou des constantes. Lorsqu'il y a une variable 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. Nous allons chercher dans un premier temps à utiliser les constantes présentes dans la requête pour limiter le nombre d'instanciations effectuées.

EXEMPLE 24. — *Soit le programme $P_{24} = (\mathcal{F}_{24}, \mathcal{R}_{24})$ composé de :* $\mathcal{R}_{24}$ =

$$\left\{ \begin{array}{ll} \rho_1 : \mathtt{p(X,Y)} \leftarrow \mathtt{a(X,Y),\ not\ q(X,Y).}, & \rho_2 : \mathtt{q(X,Y)} \leftarrow \mathtt{b(X,Y),\ not\ p(X,Y).}, \\ \rho_3 : \mathtt{s(X,Y)} \leftarrow \mathtt{p(X,Y).}, & \rho_4 : \mathtt{t(X,Y)} \leftarrow \mathtt{q(X,Y),\ not\ t(X,Y).} \end{array} \right\}$$

et de la base de faits $\mathcal{F}_{24} = \{\mathtt{a(1,2)}, \mathtt{a(3,3)}, \mathtt{a(2,1)}, \mathtt{b(2,1)}\}$ *et la requête* $Q_{24} = (\mathtt{ans} \leftarrow \mathtt{s(1,2).})$. *Nous avons* $(\mathcal{R}_{24} \downarrow Q_{24}) = \{\rho_1, \rho_2, \rho_3\}$, $règles(D\mathcal{Q}(\mathcal{R}_{24}, Q_{24})) = \{\rho_1, \rho_2, \rho_4\}$. *Nous réécrivons les règles de* $(\mathcal{R}_{24} \downarrow Q_{24})$ *en instanciant partiellement celles-ci en utilisant les constantes de la requête, ce qui donne :*

$$\left\{ \begin{array}{ll} \rho'_1 : \mathtt{p(1,2)} \leftarrow \mathtt{a(1,2),\ not\ q(1,2).}, & \rho'_2 : \mathtt{q(1,2)} \leftarrow \mathtt{b(1,2),\ not\ p(1,2).}, \\ \rho'_3 : \mathtt{s(1,2)} \leftarrow \mathtt{p(1,2).} & \end{array} \right\}$$

Nous appliquons donc simplement ρ'_1 puis ρ'_3 pour avoir la réponse à notre requête puis nous étendons à $règles(D\mathcal{Q}(\mathcal{R}_{24}, Q_{24}))$ pour vérifier que la réponse n'est pas absurde. On remarque que le nombre d'instanciations nécessaires est diminué car les variables dans les règles ρ_1, ρ_2 et ρ_3 ont été remplacées par des constantes, il n'y a donc aucune autre instanciation à faire que celles déjà effectuées. Le problème est de s'assurer que toutes les instanciations nécessaires sont bien effectuées avec la marche arrière pour pouvoir obtenir les bonnes réponses avec la marche avant.

7.2. *Instanciation des règles dangereuses négatives*

Un inconvénient majeur avec les règles dangereuses reste le fait de devoir instancier l'ensemble des variables contrairement à l'ensemble des règles de la requête où l'on peut réduire le nombre d'instanciation grâce aux constantes présentes dans la requête. Nous nous sommes intéressés aux dépendances négatives dans le graphe des symboles de prédicat afin de réduire les instanciations des règles dangereuses au minimum. Nous remarquons alors que certaines instanciations ne sont pas nécessaires pour prouver l'existence d'un réponse non absurde tant que la consistance du programme est vérifiée.

EXEMPLE 25. — *Nous reprenons le programme $P_{24} = (\mathcal{F}_{24}, \mathcal{R}_{24})$ de l'exemple précédent avec la requête $Q_{24} = (\mathtt{ans} \leftarrow \mathtt{s(1,2).})$. Nous avons $(\mathcal{R}_{24} \downarrow Q_{24}) = \{Q_{24}, \rho_1, \rho_2, \rho_3\}$, $D\mathcal{Q}(\mathcal{R}_{24}, Q_{24}) = \{(Q_{24}, +), (\rho_1, -), (\rho_2, +), (\rho_4, +)\}$. En réécrivant les règles de $(\mathcal{R}_{24} \downarrow Q_{24})$ nous obtenons la réponse* `vrai` *à notre requête sur $(\mathcal{R}_{24} \downarrow Q_{24})$ en ayant déduit* $\mathtt{p(1,2)}$ *et* $\mathtt{s(1,2)}$. *Nous souhaitons vérifier que la réponse n'est pas absurde en étendant celle-ci à $règles(D\mathcal{Q}(\mathcal{R}_{24}, Q_{24}))$. Notre base*

de faits est donc $\{\mathtt{a}(1,2), \mathtt{a}(3,3), \mathtt{a}(2,1), \mathtt{b}(2,1), \mathtt{p}(1,2), \mathtt{s}(1,2)\}$. *En essayant d'appliquer les règles de* $règles(D\mathcal{Q}(\mathcal{R}_{24}, Q_{24}))$ *nous remarquons que les instances des règles dangereuses négatives (ici* ρ_1*) ne peuvent pas rendre absurde la réponse calculée. En effet, en appliquant* ρ_1 *avec* $\mathtt{a}(3,3)$ *il est impossible d'obtenir une instance de* $\mathtt{t}(\mathtt{X},\mathtt{Y})$ *pouvant par la suite rendre la réponse absurde. Par contre si nous déclenchons la règle* ρ_2 *avec* $\mathtt{b}(2,1)$ *nous créons une instance de* $\mathtt{t}(\mathtt{X},\mathtt{Y})$ *si la règle* ρ_1 *n'a pas déjà été déclenchée avec* $\mathtt{a}(2,1)$ *qui bloque l'application de* ρ_2*. L'application de* ρ_1 *ne peut donc pas rendre la réponse absurde mais peut rendre une réponse non absurde si elle bloque l'application d'une règle dangereuse positive. Dans notre cas nous économisons une instanciation de règle avec* $\mathtt{a}(3,3)$ *car il n'existe pas de règle qui sera bloquée par celle-ci et les instances de* $\mathtt{a}(\mathtt{X},\mathtt{Y})$ *ne peuvent pas rendre une réponse absurde.*

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

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. Springer.

Alviano M., Faber W. (2011). Dynamic magic sets and super-coherent answer set programs. *AI Commun.*, vol. 24, n° 2, p. 125–145.

Alviano M., Faber W., Woltran S. (2014, 5). 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 P. Barceló, R. Pichler (Eds.), *Proceedings of the second international workshop on datalog in academia and industry, datalog 2.0, vienna, austria, september 11-13, 2012.*

Apt K. R., Bol R. (1994). Logic programming and negation: A survey. *Journal Of Logic Programming*, vol. 19, p. 9–71.

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 fifth acm sigact-sigmod symposium on principles of database systems*, p. 1–15.

Benchmark university. (s. d.). http://swat.cse.lehigh.edu/projects/lubm. (Dernière visite: 2016-04-04)

Benchmark university dlv. (s. d.). https://www.mat.unical.it/kr2012/. (Dernière visite: 2017-09-27)

Benchmark university graal. (s. d.). http://graphik-team.github.io/graal/experiments1. (Dernière visite: 2017-09-27)

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. *J. Autom. Reasoning*, vol. 39, nº 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 Trans. Knowl. Data Eng.*, vol. 1, nº 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 iswc-2015), bethlehem, pa, usa, october 11, 2015.*

Dung P. (1992). On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, vol. 105, nº 1, p. 7 - 25.

Faber W., Greco G., Leone N. (2007). Magic sets and their application to data integration. *Journal of Computer and System Sciences*, vol. 73, nº 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*, vol. 7265, p. 247-264. Springer.

Gallaire H., Minker J., Nicolas J. (1984). Logic and databases: A deductive approach. *ACM Comput. Surv.*, vol. 16, nº 2, p. 153–185.

Garreau F., Garcia L., Lefèvre C., Stéphan I. (2015). ∃-asp. In *Proceedings of ontolp workshop (co-located with ijcai 2015), buenos aires, argentina, july 25-27, 2015.*

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)*, vol. 6645, p. 345-351. Springer.

Gebser M., Kaufmann B., Neumann A., Schaub T. (2007). Conflict-driven answer set solving. In *IJCAI 2007*, p. 386-392.

Gelfond M., Lifschitz V. (1988). The stable model semantics for logic programming. In *Logic programming, proceedings of the fifth international conference and symposium*, p. 1070-1080.

Glimm B., Horrocks I., Motik B., Stoilos G., Wang Z. (2014). Hermit: An OWL 2 reasoner. *J. Autom. Reasoning*, vol. 53, nº 3, p. 245–269.

Guo Y., Pan Z., Heflin J. (2005). LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, vol. 3, nº 2-3, p. 158–182.

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 2011, heraklion, crete, greece, may 29-june 2, 2011, proceedings, part I*, p. 382–396.

Konczak K., Linke T., Schaub T. (2006). Graphs and colorings for answer set programming. *TPLP*, vol. 6, n° 1-2, p. 61–106.

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 *Principles of knowledge representation and reasoning: Proceedings of the thirteenth international conference, KR 2012, rome, italy, june 10-14, 2012.*

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.

Owl2dlgp graal. (s. d.). http://graphik-team.github.io/graal/downloads/owl2dlgp. (Dernière visite: 2017-09-27)

Pérez-Urbina H., Horrocks I., Motik B. (2009). Efficient query answering for OWL 2. In *The semantic web - ISWC 2009, 8th international semantic web conference, ISWC 2009, chantilly, va, usa, october 25-29, 2009. proceedings*, p. 489–504.

Schaub T., Thielscher M. (1996). Skeptical query-answering in constrained default logic. In D. M. Gabbay, H. J. Ohlbach (Eds.), *Proceedings of the international conference on formal and applied practical reasoning (fapr'96), bonn, germany, june 3–7, 1996*, p. 567–581. Berlin, Heidelberg, Springer Berlin Heidelberg.

Sirin E., Parsia B., Grau B. C., Kalyanpur A., Katz Y. (2007). Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, vol. 5, n° 2, p. 51–53.

DLV - manuel d'utilisation. (s. d.). http://www.dlvsystem.com/html/DLV_User_Manual.html.

Weinzierl A. (2017). Blending lazy-grounding and CDNL search for answer-set solving. In M. Balduccini, T. Janhunen (Eds.), *Logic programming and nonmonotonic reasoning - 14th international conference, LPNMR 2017, espoo, finland, july 3-6, 2017, proceedings*, vol. 10377, p. 191–204. Springer.

Preuves

PREUVE. — du théorème 13 (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 11 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, $\mathtt{p} = pred(t\hat{e}te(d_r^+))$ 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 $\mathtt{p} = pred(t\hat{e}te(r))$ 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 $t\hat{e}te(d_r^+) \notin corps^+(r^+)$, 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 11, 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 $\mathtt{p} = pred(t\hat{e}te(d_r^-))$ 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 $\mathtt{p} = pred(t\hat{e}te(d_r^-))$ 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 $pred(t\hat{e}te(r))$ 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 18 (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, tel que $P' = (\mathcal{F}, \bigcup_{D\in\Delta} D)$ possède un *answer set*. Si P est inconsistant alors d'après le théorème 7 *inconsistance* (sans numérotation) il existe un cycle d'inconsistance dans $\mathcal{G}$ le graphe des symboles de prédicat de P'. D'après la définition 11, 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 $\mathcal{G}'$ 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 Δ' = Δ. 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 20 (Réponse à un programme super-consistant). Soit $(P?Q) = (\mathcal{F}, \mathcal{R} \cup \{Q\})$ un programme ASP super-consistant avec Q une requête sur $(P?Q)$, nous avons $(\mathcal{F}, (P?Q)) = (\mathcal{F}, (\mathcal{R} \downarrow Q))$ avec $(\mathcal{R} \downarrow Q)$ l'ensemble des règles dépendantes de la requête Q. D'après la définition du programme super-consistant, $(P?Q)$ possède au moins un *answer set* quelle que soit sa base de faits. $(P?Q)$ et $(\mathcal{F}, (P?Q))$ sont donc consistants si nous ne leur ajoutons pas de règles. Supposons qu'il existe une réponse dans $(\mathcal{F}, (P?Q))$ n'étant pas une réponse dans $(P?Q)$. Alors il existe une instance de `ans` dans un *answer set* de $(\mathcal{F}, (P?Q))$ n'appartenant pas à

un *answer set* de $(P?Q)$. $(\mathcal{R} \downarrow Q)$ est l'ensemble des règles de $\mathcal{R}$ dont la requête dépend contenant toutes les règles permettant d'obtenir des instances de `ans` donc s'il existe une réponse à Q dans $(\mathcal{F}, (P?Q))$ elle sera aussi une réponse dans $(P?Q)$.
Supposons maintenant qu'il existe une réponse dans $(P?Q)$ n'apparaissant pas dans $(\mathcal{F}, (P?Q))$. Étant donné que $(\mathcal{R} \downarrow Q)$ contient toutes les règles permettant d'obtenir des instances de `ans`, il ne peut pas exister de réponse sur $(P?Q)$ n'appartenant pas à $(\mathcal{F}, (P?Q))$. ■

PREUVE. — du théorème 23 (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 20, nous savons que si le programme est consistant quelque soit $\mathcal{F}$ alors une réponse dans $(\mathcal{F}, (P?Q))$ 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 $(\mathcal{R} \downarrow Q)$, 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 règles dont Q dépend ($(\mathcal{F}, (\mathcal{R} \downarrow Q) \cup règles(D\mathcal{Q}(\mathcal{R}, Q)))$ contient $(\mathcal{F}, (P?Q))$) 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. ■