Traiter les exceptions en ASP à partir d'une représentation compacte des informations

Stéphane NGOMA, Laurent GARCIA et Pascal NICOLAS

LERIA, UFR Sciences, Université d'Angers
2 boulevard Lavoisier, 49045 ANGERS CEDEX 01
{ngoma,garcia,pn}@info.univ-angers.fr

Résumé : L'ASP, utilisé dans la forme des programmes logiques normaux, est un cadre adéquat pour le raisonnement non-monotone dans la mesure où il offre un modèle formel valide ainsi que des systèmes opérationnels. Le problème est que la représentation des informations dans ce cadre n'est pas une opération aisée. C'est pourquoi nous proposons de générer automatiquement ces programmes logiques normaux à partir d'une représentation compacte des informations à base de programmes logiques définis. Pour cela, nous appliquons une méthode définie dans le cadre de la logique des défauts dont nous proposons une adaptation.

Mots-clés : ASP, exceptions, spécificité

1 Introduction

Delgrande et Schaub (Delgrande & Schaub, 1997) ont présenté une approche générale et automatique pour introduire les informations de spécificité dans les théories non monotones. Cette approche est illustrée, entre autres, pour la logique des défauts (Reiter, 1980) ; pour autant, aucun système opérationnel n'a été développé.

Pour le travail présenté ici, nous avons choisi un cadre où le modèle formel est correct et pour lequel il existe des systèmes opérationnels performants : l'*Answer Set Programming* (ASP) (qui peut être vu comme un sous-domaine de la logique des défauts). Les informations sont codées sous forme de règles, un processus d'inférence permet de faire du raisonnement ; certaines règles peuvent être bloquées (d'où la propriété de non monotonie) et permettent d'exprimer les exceptions.

Certaines définitions ont été proprosées par Delgrande et Schaub de manière indépendante du formalisme. Néanmoins, pour être mené à terme, le processus est lié à une représentation particulière. C'est pourquoi nous redéfinissons l'intégralité des définitions dans le cadre de l'ASP, ce qui permet d'associer le travail théorique à une mise en œuvre pratique.

D'un point de vue formel, on distingue deux aspects dans l'ASP : les programmes logiques normaux, qui cadrent bien à la problématique mais sont difficiles à écrire, et les programmes logiques définis, plus simples mais moins intéressants (pas de gestion des exceptions). Il convient donc d'écrire les informations sous forme de programme

logique défini, de le transformer automatiquement en un programme logique normal adéquat puis d'utiliser le moteur d'inférence des programmes logiques normaux pour raisonner.

Dans la suite du document, nous présentons formellement l'ASP dans la section 2, puis nous exposons les travaux sur lesquels nous nous appuyons dans la section 3, à savoir l'approche de Delgrande et Schaub. Ensuite, dans la section 4, nous abordons l'aspect théorique du travail effectué.

2 Answer Set Programming (ASP)

Cette section présente le formalisme de programmation logique non monotone que nous utilisons dans ce travail. Il existe plusieurs variantes autour de ce formalisme que l'on regroupe sous le paradigme de l'*Answer Set Programming* (ASP) et nous présentons celles qui nous sont nécessaires.

Un *programme logique défini* est un ensemble de règles de la forme :

$$b \leftarrow a_1, \ \ldots \ , a_n. \qquad (n \geq 0) \qquad .$$

Pour une telle règle r, $tête(r) = b$ est un atome appelé *tête* et $corps(r) = \{a_1, \ldots, a_n\}$ est un ensemble d'atomes appelé *corps*. La règle $b \leftarrow a.$ peut se lire : « si l'on peut prouver a, alors on conclut b ». Étant donné une règle r et un ensemble d'atomes A, on dit que r est *applicable* (ou *déclenchable*) dans A si $corps(r) \subseteq A$. Un ensemble d'atomes A est dit *clos* par rapport à un programme P si et seulement si pour toute règle r de P, si $corps(r) \subseteq A$, alors $tête(r) \in A$. On appelle $Cn(P)$, ou *modèle de Herbrand*, le plus petit ensemble d'atomes clos par rapport au programme P. Pour un programme P et un ensemble d'atomes A, l'opérateur T_P défini par :

$$T_P(A) = \{tête(r) \,|r \in P, corps(r) \subseteq A\}$$

calcule l'ensemble des atomes déductibles à partir de P et A. À partir de cet opérateur, on définit la suite : $T_P^0 = T_P(\emptyset)$ et $T_P^{k+1} = T_P(T_P^k), \forall k \geq 0$. $Cn(P)$ est le plus petit point fixe de T_P et $Cn(P) = \bigcup_{k \geq 0} T_P^k$. Ainsi, $Cn(P)$ contient tous les atomes que l'on peut déduire du programme P. On dit qu'un programme logique défini est *enraciné* s'il peut être ordonné selon la suite $\langle r_1, \ \ldots \ , r_n \rangle$ telle que $\forall i$, $1 \leq i \leq n$, r_i est applicable dans $\{tête(r_j) \mid 1 \leq j < i\}$.

Un *programme logique normal* est un ensemble de règles de la forme :

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

Comme précédemment, les a_i, b_j et c sont des atomes et pour une telle règle r on note $corps^+(r) = \{a_1, \ldots, a_n\}$ (respectivement $corps^-(r) = \{b_1, \ldots, b_m\}$) son corps *positif* (respectivement *négatif*). En outre, $r^+ = c \leftarrow a_1, \ \ldots \ , a_n.$ désigne la *projection positive* de la règle r. Intuitivement, la règle $c \leftarrow a, \ not \ b.$ peut se lire : « si l'on peut prouver a et si l'on ne peut pas prouver b, alors on peut conclure c ». Les « not » du corps négatif s'interprètent donc comme des *négations par défaut*. Soit une règle r et un ensemble d'atomes A. On dit que r est *applicable* (ou *déclenchable*) dans A si $corps^+(r) \subseteq A$ et $corps^-(r) \cap A = \emptyset$.

Définition 1
Le réduit d'un programme logique normal P par rapport à un ensemble d'atomes A est le programme logique défini : $P^A = \{r^+ \,|r \in P, corps^-(r) \cap A = \emptyset\}$.

La sémantique originelle (Gelfond & Lifschitz, 1988) des programmes logiques normaux est celle des modèles stables rappelée ci-après.

Définition 2
Soit P un programme logique normal et S un ensemble d'atomes. S est un modèle stable *de P si et seulement si $S = Cn(P^S)$.*

Pour des besoins de représentation des connaissances on peut être amené à utiliser conjointement la « vraie » négation (appelée aussi *négation forte*) et la négation par défaut au sein d'un même programme. Ceci est rendu possible par les *programmes logiques étendus* (Gelfond & Lifschitz, 1991), où l'on autorise les littéraux à la place des atomes dans l'écriture des règles. Mais si, comme c'est notre cas, on refuse les modèles inconsistants alors la sémantique associée à ces programmes est réductible à celle des modèles stables en prenant en compte les considérations suivantes :

- tout littéral $\neg x$ est codé par l'atome nx,
- on ajoute une règle $\bot \leftarrow x,\ nx.$ pour tout atome x,
- un modèle stable ne doit pas contenir le symbole $\bot$.

Les règles de type $\bot \leftarrow x,\ nx.$, parfois simplement notées $\leftarrow x,\ nx.$ (sans tête), sont appelées des *contraintes*. L'ensemble des contraintes d'un programme P est noté P_K. L'ajout des contraintes induit le comportement attendu, à savoir qu'il est impossible d'obtenir un modèle stable contenant à la fois x et nx. Ainsi, seuls les modèles stables consistants sont conservés. Cette manière de faire est celle implantée dans tous les solveurs disponibles qui sont donc capables de traiter les programmes logiques étendus tout en restant dans le cadre des programmes logiques normaux.

Exemple 2.1
L'exemple habituel des oiseaux (où O désigne « oiseau », A « avoir des ailes », M « manchot » et V « vole ») peut donc être codé de cette manière : $\{V \leftarrow O,\ not\ M.,\ A \leftarrow O.,\ O \leftarrow M.,\ nV \leftarrow M.,\ \bot \leftarrow V,\ nV.\}$. En présence d'un manchot (codé par l'ajout de $M \leftarrow .$), on obtient un seul modèle stable : $\{A,\ M,\ O,\ nV\}$, ce qui correspond à l'intuition.

Il est reconnu que les programmes logiques normaux sont adaptés à notre problématique de représentation de règles avec exceptions. Mais ils sont difficiles à écrire dans la mesure où il faut expliciter les exceptions ou tout au moins décrire les anormalités. Or on veut pouvoir donner une représentation simple des informations, ce que permettent les programmes logiques définis qui, par contre, peuvent mener à des incohérences car ils ne prennent pas en compte la notion d'exception. Notre travail consiste donc à résoudre ce problème en déterminant de manière automatique des classes et sous-classes d'information. La notion de *spécificité* (Pearl, 1990) prend alors tout son sens. Cet aspect ayant déjà été traité formellement pour la logique des défauts (Delgrande & Schaub, 1997), il nous reste à l'appliquer à l'ASP et à définir un processus automatique gérant correctement les exceptions. Ainsi, à partir d'informations présentées sous la forme simple suivante : $\{O \rightarrow V, O \rightarrow A, M \rightarrow O, M \rightarrow \neg V\}$, on veut obtenir automatiquement le programme logique normal adéquat : $\{V \leftarrow O,\ not\ nV,\ not\ M.,\ A \leftarrow O., O \leftarrow M., nV \leftarrow M,\ not\ V., \bot \leftarrow V,\ nV.\}$.

3 Spécificité en logique des défauts

Nous présentons l'approche de Delgrande et Schaub (Delgrande & Schaub, 1997) proposée dans le cadre de la logique des défauts (Reiter, 1980). Elle se compose de deux étapes majeures. Premièrement, à partir d'une théorie de défauts, il faut localiser les règles qui sont en conflit et qui n'ont pas la même spécificité ; ceci est accompli en utilisant une partie du mécanisme du Système Z (Pearl, 1990). Ainsi pour notre exemple, il est clair que les règles $O \rightarrow V$ et $M \rightarrow \neg V$ sont en conflit (une contradition est déduite, à savoir V et $\neg V$, à cause de $M \rightarrow O$) et que la seconde est plus spécifique que la première (M est une sous-classe de O). Deuxièmement, il faut modifier certaines règles de sorte que si les deux règles en contradiction sont potentiellement applicables alors seule la plus spécifique est appliquée.

Dans le Système Z, un ensemble de règles R est partitionné (stratifié) en sous-ensembles $R_0, \ldots, R_n$ où les règles d'une strate inférieure sont moins spécifiques que celles d'une strate supérieure. La partition résultante est appelée un *ordre Z* ; cet ordre fournit donc des informations de spécificité. Plutôt que calculer l'ordre Z complet, Delgrande et Schaub déterminent d'abord des ensembles minimaux de règles en conflit, qu'ils stratifient séparément ; ainsi ils classent les informations selon leur spécificité, relativement au(x) conflit(s) dans le(s)quel(s) elles interviennent. Dans notre exemple, $C = \{O \rightarrow V,\ M \rightarrow O,\ M \rightarrow \neg V\}$ est un conflit (en présence de M).

Delgrande et Schaub ont montré que l'ordre Z d'un tel ensemble C est une partition binaire (C_0, C_1) des règles ; les règles de C_0 sont moins spécifiques que celles de C_1. Par exemple, pour C nous obtenons : $C_0 = \{O \rightarrow V\}$ et $C_1 = \{M \rightarrow O,\ M \rightarrow \neg V\}$. Par conséquent, si les règles de C_1 sont applicables, on voudrait être sûr que certaines règles de C_0 sont bloquées.

Maintenant que nous avons localisé les règles en conflit, il nous faut déterminer lesquelles sont candidates pour être bloquées, ainsi que le moyen de bloquer leur application dans le formalisme utilisé.

Intuitivement, O est une sur-classe de M. Si O et M sont vrais, alors dans la traduction en logique des défauts, on voudrait que le défaut correspondant à $M \rightarrow \neg V$ soit applicable de préférence à $O \rightarrow V$. Donc on veut que les règles les plus spécifiques soient applicables de préférence aux règles moins spécifiques en conflit, indépendamment des autres règles de l'ensemble. Ceci est fait en localisant les règles dont l'application conjointe crée une inconsistance. Dans notre exemple, il s'agit de $O \rightarrow V$ et $M \rightarrow \neg V$. $O \rightarrow V \in C_0$ et $M \rightarrow \neg V \in C_1$. Les règles sélectionnées de cette manière dans C_0 constituent les candidates pour être bloquées. Ce critère de sélection a la propriété importante d'être *indépendant du contexte*. Pour des théories de défauts R et R' telles que $R \subseteq R'$, si $r \in R$ est choisie, alors elle devrait également être choisie dans R'. De plus, si l'on souhaite bloquer une règle dans un théorie de défauts R, alors on voudra aussi la bloquer dans tout sur-ensemble R'.

Le deuxième problème (*« Comment bloquer l'application d'une règle ? »*) dépend du formalisme utilisé. Pour la logique des défauts par exemple, la théorie de défauts correspondant à notre ensemble de règles est composée de défauts normaux, excepté pour ceux représentant les règles sélectionnées de la manière décrite ci-dessus, qui deviennent semi-normaux ; pour ces derniers, la justification se compose du conséquent

avec une assertion assurant que les règles de C_1 sélectionnées comme précédemment ne sont pas applicables (on ajoute ces règles, transformées en implications matérielles, puis on simplifie si possible).

Considérons l'ensemble C. Les règles $M \rightarrow O$ et $M \rightarrow \neg V$ sont transformées respectivement en : $\frac{M:O}{O}$ et $\frac{M:\neg V}{\neg V}$. La règle $O \rightarrow V$ est quant à elle transformée en : $\frac{O:V \wedge (M \rightarrow \neg V)}{V}$, qui peut être simplifié en : $\frac{O:V \wedge \neg M}{V}$.

Pour résumer, on peut décomposer l'approche décrite ci-dessus de la manière suivante, pour un ensemble de règles R :

Algorithme 3.1 Procédure `transformer_ensemble`

Entrées : Un ensemble R de règles avec exceptions
Sorties : L'ensemble R modifié de manière à gérer la spécificité

```
1 conf ← conflits_minimaux(R)
2 pour chaque C ∈ conf faire
3       stratifier(C)
4       sélectionner_règles(C)
5 fin pour
6 transformer_règles(R, conf)
```

Nous allons procéder de la même manière dans le cadre de l'ASP : nous commencerons par chercher les conflits, nous les stratifierons puis les partitionnerons et, enfin, nous modifierons les règles candidates.

4 Spécificité en ASP

Pour notre travail, nous avons adapté pour l'ASP toutes les notions définies dans les travaux sur lesquels nous nous basons ainsi que les algorithmes liés.

Définition 3
Pour une règle r d'un programme logique défini P telle que $r = b \leftarrow a_1, \ldots, a_n.(n \geq 0)$, un ensemble d'atomes A :

- *satisfait r si $\{a_1, \ldots, a_n\} \not\subseteq A$ ou $b \in A$;*
- *vérifie r si $\{a_1, \ldots, a_n\} \subseteq A$ et $b \in A$;*
- *falsifie r si $\{a_1, \ldots, a_n\} \subseteq A$ et $b \notin A$.*

Définition 4
Une règle r d'un programme P est dite tolérée *par P si et seulement s'il existe un ensemble d'atomes A, clos par rapport à P et consistant, qui vérifie r.*

La tolérance d'une règle caractérise le fait que son application ne génère pas de contradiction. À partir de cette notion de tolérance on va pouvoir obtenir une stratification du programme appelée *ordre Z*.

Nous développons par la suite les quatre étapes de l'algorithme concernant les conflits (à savoir les lignes 1, 3, 4 et 6 de la procédure $transformer\_ensemble$). Un conflit étant un ensemble minimal, les règles qui le composent ne peuvent introduire qu'une et une seule inconsistance ; ainsi, un conflit comporte une et une seule contrainte (voir la section 2 pour la définition d'une contrainte). Pour un conflit C, on définit :

- $contrainte(C)$ la contrainte de C ;
- $règles(C) = C \setminus \{contrainte(C)\}$;
- $contr(C) = corps(contrainte(C))$ (atomes en conflit).

4.1 Calcul des conflits *(ligne 1 de l'algorithme)*

Si déterminer les conflits pour un nombre restreint de règles peut sembler simple, ce n'est pas forcément vrai dans le cas général. La définition d'un conflit (adaptée dans notre cadre) donnée dans (Delgrande & Schaub, 1997) est la suivante, où un ordre Z trivial possède une seule strate :

Définition 5
Soit P un programme logique défini. Un ensemble de règles $C \subseteq P$ est un conflit *dans P si et seulement si C a un ordre Z non trivial et si chaque $C' \subset C$ a un ordre Z trivial.*

Dans le pire des cas, pour un programme P de n règles, il faudrait donc isoler et stratifier les 2^n sous-ensembles de P. Heureusement, des travaux similaires ont été menés. En logique classique, et en particulier dans le cadre du problème SAT, un MUS (*Minimally Unsatisfiable Subformula*) est un ensemble insatisfiable de clauses tel que tous ses sous-ensembles propres sont satisfiables. Un tel ensemble fournit donc une « explication » de l'incohérence qui ne peut être plus petite en termes de nombre de clauses impliquées, ce qui correspond à notre idée de conflits. Des travaux ont montré que le calcul de MUS n'est pas réalisable en pratique dans le pire dans cas ; en effet, décider si un ensemble de clauses est un MUS est DP-complet (Papadimitriou & Wolfe, 1988), et tester si une formule appartient à l'ensemble des MUS est Σ_2^p-difficile (Eiter & Gottlob, 1992). Mais Bruni (Bruni, 2005) a montré que pour certaines classes de clauses (dont les clauses de Horn), l'extraction d'un MUS pouvait être réalisée en un temps polynomial.

Nous avons donc décidé d'utiliser l'algorithme HYCAM[1] (Grégoire *et al.*, 2007) qui permet de déterminer tous les MUS d'une instance SAT en un temps raisonnable. La fonction $conflits\_minimaux$ détermine les conflits grâce à des appels à HYCAM.

On veut connaître les règles qui posent *potentiellement* problème (lorsqu'elles sont applicables conjointement), il faut donc ajouter des faits au programme avant de le passer à HYCAM (ligne 7). Choisir successivement les atomes du corps de chaque règle (lignes 3 à 7) plutôt que chaque atome (présent dans le programme) séparément assure que chaque règle soit déclenchée au moins une fois ; l'algorithme trouve ainsi tous les MUS dans lesquels elle intervient. Nous cherchons des ensembles minimaux ; nous devons donc nous assurer qu'aucun conflit généré n'est un sur-ensemble d'un autre conflit (lignes 8 à 10).

Exemple 4.1
Soit le programme : P = {$Co \leftarrow Mo.(r_1)$, $Mo \leftarrow Ce.(r_2)$, $nCo \leftarrow Ce.(r_3)$, $Ce \leftarrow Na.(r_4)$, $Co \leftarrow Na.(r_5)$, $\bot \leftarrow Co,\ nCo.(c_1)$} où généralement, les mollusques (Mo)

[1] Il s'agit en réalité d'une amélioration d'un algorithme existant, CAMUS (*Computing All MUS*) (Liffiton & Sakallah, 2008).

Algorithme 4.1 Fonction `conflits_minimaux`

Entrées : Un programme logique P
Sorties : L'ensemble des ensembles minimaux de conflit de P

```
1  conflits ← ∅
2  traités ← ∅
3  pour chaque r ∈ P \ P_K faire
4      A ← corps(r)
5      si A ∉ traités alors
6          traités ← traités ∪ {A}
7          conf ← HYCAM(P ∪ A)
8          pour chaque C ∈ conf faire
9              si ∀C' ∈ conflits, C' ⊈ C alors
10                 conflits ← (conflits ∪ {C}) \ {C' ∈ conflits | C ⊂ C'}
11             fin si
12         fin pour
13     fin si
14 fin pour
15 retourner conflits
```

sont des coquillages (Co), les céphalopodes (Ce) sont des mollusques, les céphalopodes ne sont pas des coquillages, les nautiles (Na) sont des cépalopodes et les nautiles sont des coquillages. Les ensembles de faits qu'il faut ajouter sont $\{Mo\}$, $\{Ce\}$ et $\{Na\}$. Pour $\{Mo\}$, aucun MUS (conflit) n'est détecté. Pour $\{Ce\}$, on obtient le conflit $C = \{r_1,\ r_2,\ r_3,\ c_1\}$. Pour $\{Na\}$, on obtient $C' = \{r_1,\ r_2,\ r_3,\ r_4,\ c_1\}$ et $C'' = \{r_3,\ r_4,\ r_5,\ c_1\}$. $C \subset C'$, donc C' n'est pas minimal. Les deux seuls conflits de P sont donc : C = {$Co \leftarrow Mo.$, $Mo \leftarrow Ce.$, $nCo \leftarrow Ce.$, $\bot \leftarrow Co,\ nCo.$} et C'' = {$nCo \leftarrow Ce.$, $Ce \leftarrow Na.$, $Co \leftarrow Na.$, $\bot \leftarrow Co,\ nCo.$}

4.2 Stratification des conflits *(ligne 3 de l'algorithme)*

Delgrande et Schaub ont montré qu'un conflit possède un ordre Z avec exactement deux strates.

Algorithme 4.2 Fonction `stratifier`

Entrées : Un ensemble minimal de conflit C
Sorties : La stratification (C_0, C_1) de C

```
1  C_0 ← ∅
2  C_1 ← ∅
3  pour chaque r ∈ règles(C) faire
4      si r est tolérée par (règles(C) \ {r}) ∪ {contrainte(C)} alors
5          C_0 ← C_0 ∪ {r}
6      sinon
7          C_1 ← C_1 ∪ {r}
8      fin si
9  fin pour
10 retourner (C_0, C_1)
```

On sait qu'il n'y aura que deux strates, donc soit une règle est tolérée dans le conflit (ligne 4) et elle appartient à la strate la plus générale (ligne 5), soit elle ne l'est pas (ligne 6) et elle appartient à la strate la plus spécifique (ligne 7).

Exemple 4.2
Considérons un conflit de l'exemple précédent : C = {$Co \leftarrow Mo.(r_1)$, $Mo \leftarrow Ce.(r_2)$, $nCo \leftarrow Ce.(r_3)$, $\perp \leftarrow Co,\ nCo.(c_1)$}. $\{Mo,\ Co\}$ *vérifie* r_1, *est clos par rapport à* C *et est consistant. Donc* r_1 *est tolérée dans* C, *et* $r_1 \in C_0$. *Le seul ensemble clos qui vérifie* r_2 *et ne falsifie ni* r_1 *ni* r_3 *est* $A = \{Ce,\ Mo, Co,\ nCo\}$; *mais* A *est inconsistant. Alors* $r_2 \in C_1$. *De même, on trouve que* $r_3 \in C_1$. *Finalement, l'ordre Z associé à* C *est le suivant :* C_0 = {$Co \leftarrow Mo.$} *et* C_1 = {$Mo \leftarrow Ce.$, $nCo \leftarrow Ce.$}.

4.3 Sélection des règles candidates *(ligne 4 de l'algorithme)*

Une fois un conflit C stratifié, on peut faire ressortir trois sous-ensembles : les règles candidates pour être modifiées (règles générales ayant des exceptions), $min(C)$; les règles indiquant comment modifier les règles précédentes (les exceptions), $max(C)$; et les règles restantes faisant le lien entre ces deux ensembles, $inf(C)$. Pour cela, il faut d'abord déterminer le plus petit sous-ensemble de règles dont l'application conjointe provoque une inconsistance.

Définition 6
Soit (C_0, C_1) *la stratification des règles du conflit* C. *Un* noyau *de* C *est une paire d'ensembles minimaux* $\big(min(C), max(C)\big)$ *tels que* $min(C) \subseteq C_0$, $max(C) \subseteq C_1$ *et* {$a \leftarrow .|a \in corps(r) \cup \{tête(r)\}$ *tq* $r \in min(C) \cup max(C)$} $\cup$ {$contrainte(C)$} *possède un modèle inconsistant.*

Algorithme 4.3 Procédure `sélectionner_règles`

Entrées : Un ensemble minimal de conflit stratifié C
Sorties : Les règles candidates ($min(C)$), les exceptions ($max(C)$) et les autres ($inf(C)$)
1 $noyau \leftarrow \{r \in$ règles(C) | tête(r) $\in$ contr(C)$\}$
2 min(C) $\leftarrow noyau \cap C_0$
3 max(C) $\leftarrow noyau \cap C_1$
4 inf(C) $\leftarrow$ règles(C) $\setminus noyau$

Le noyau est constitué des règles qui concluent sur des atomes en conflit (ligne 1) ; ainsi, nous sommes assurés qu'un tel noyau est unique puisque les têtes des règles ne contiennent qu'un atome. Il est alors facile de partitionner le conflit (lignes 2 à 4).

Exemple 4.3
Reprenons le conflit stratifié précédemment : C_0 = {$Co \leftarrow Mo.$}, C_1 = {$Mo \leftarrow Ce.$, $nCo \leftarrow Ce.$}, $contrainte(C) = \perp \leftarrow Co,\ nCo.$, $contr(C)$ = {$Co,\ nCo$}, *alors le noyau est :* $\big($ {$Co \leftarrow Mo.$}, {$nCo \leftarrow Ce.$}$\big)$ *et donc :* $min(C)$ = {$Co \leftarrow Mo.$}, $max(C)$ = {$nCo \leftarrow Ce.$} *et* $inf(C)$ = {$Mo \leftarrow Ce.$}.

4.4 Modification des règles *(ligne 6 de l'algorithme)*

Une fois les noyaux des conflits déterminés, les règles sont transformées de telle sorte que les plus générales sont bloquées si au moins une de leurs exceptions est applicable,

mais restent déclenchables sinon. Pour cela, nous transformons les règles les plus générales en explicitant les exceptions dans leur corps négatif. Le corps négatif des autres règles non sujettes à exceptions est laissé vide. L'algorithme suivant réalise cette tâche en transformant un programme logique défini P en le programme logique normal correspondant à partir de l'ensemble de ses conflits $conflits$:

Algorithme 4.4 Procédure `transformer_règles`

Entrées : Un programme logique défini P, un ensemble $conflits$ d'ensembles minimaux de conflit
Sorties : P transformé en un programme logique normal tenant compte de la spécificité

```
1  pour chaque r ∈ P faire
2      si r est une contrainte alors
3          pour chaque r' ∈ P tel que tête(r') ∈ corps+(r) faire
4              corps−(r') ← corps−(r') ∪ (corps+(r) \ tête(r'))
5          fin pour
6      sinon
7          conf ← {C ∈ conflits | r ∈ min(C)}
8          si conf ≠ ∅ alors
9              pour chaque C ∈ conf faire
10                 pour chaque r' ∈ max(C) faire
11                     si |corps+(r')| = 1 alors
12                         corps−(r) ← corps−(r) ∪ corps+(r')
13                     sinon
14                         P ← P ∪ {σ_r'}
15                         corps−(r) ← corps−(r) ∪ {tête(σ_r')}
16                     fin si
17                 fin pour
18             fin pour
19         fin si
20     fin si
21 fin pour
```

Si r est une contrainte, on transforme toutes les règles pouvant déclencher la contrainte c'est-à-dire pouvant être impliquées dans une contradiction (lignes 3 à 5). Si ce n'est pas une contrainte, on cherche les conflits pour lesquels $r \in min(C)$ (candidate pour être modifiée) (ligne 7) puis on modifie la règle en explicitant ses exceptions (lignes 9 à 18), avec un traitement différent selon le nombre d'atomes dans le corps positif de la règle r' définissant l'exception (ligne 12 ou lignes 14 et 15). Pour une telle règle r' (lignes 14 et 15), on définit $\sigma_{r'}$, une nouvelle règle telle que :

- $tête(\sigma_{r'})$ est un nouvel atome n'apparaissant pas déjà dans le programme ;
- $corps^+(\sigma_{r'}) = corps^+(r')$;
- $corps^-(\sigma_{r'}) = \emptyset$.

Par construction, $\sigma_{r'}$ est une règle qui sera déclenchée chaque fois que r sera applicable.

Ainsi, pour chacune de ses exceptions r', r est bloquée si tous les atomes du corps positif de r' sont prouvés (i.e. r' est applicable) ou si la tête de r' a été obtenue (ou si on a déjà conclu l'« opposé » de la tête de r).

Exemple 4.4
Pour notre exemple, le programme logique normal correspondant est : $P = \{Co \leftarrow Mo,\ not\ nCo,\ not\ Ce.,\ Mo \leftarrow Ce.,\ nCo \leftarrow Ce,\ not\ Co,\ not\ Na.,\ Ce \leftarrow Na.,\ Co \leftarrow Na,\ not\ nCo.,\ \bot \leftarrow Co,\ nCo.\}$.

5 Conclusion et perspectives

A partir d'une représentation compacte des informations, nous pouvons maintenant utiliser les ASP pour représenter automatiquement les informations tolérant les exceptions et pouvoir raisonner avec via la sémantique des modèles stables.

Ce travail, développé lors d'un stage de Master Recherche, comporte aussi le développement du système pratique opérationnel implantant les algorithmes décrits ici. Toutes les informations (rapport de stage et programmes) sont disponibles à l'adresse `http://www.info.univ-angers.fr/pub/pn/Softwares/aspSpecif.html`.

Nous nous intéressons maintenant à resituer notre travail par rapport aux autres travaux existants en particulier aux propriétés définies pour les systèmes de raisonnement non-monotone telles que celles du système P et la monotonie rationnelle.

Références

BRUNI R. (2005). On Exact Selection of Minimally Unsatisfiable Subformulae. *Annals of Mathematics and Artificial Intelligence*, **43**(1-4), 35–50.

DELGRANDE J. P. & SCHAUB T. (1997). Compiling specificity into approaches to nonmonotonic reasoning. *Artificial Intelligence*, **90**(1-2), 301–348.

EITER T. & GOTTLOB G. (1992). On the Complexity of Propositional Knowledge Base Revision, Updates and Counterfactuals. *Artificial Intelligence*, **57**(2-3), 227–270.

GELFOND M. & LIFSCHITZ V. (1988). The stable model semantics for logic programming. In R. A. KOWALSKI & K. BOWEN, Eds., *International Conference on Logic Programming*, p. 1070–1080, Cambridge, Massachusetts : The MIT Press.

GELFOND M. & LIFSCHITZ V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, **9(3-4)**, 363–385.

GRÉGOIRE E., MAZURE B. & PIETTE C. (2007). Boosting a Complete Technique to Find MSS and MUS thanks to a Local Search Oracle. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, volume 2, p. 2300–2305, Hyderabad (Inde).

LIFFITON M. H. & SAKALLAH K. A. (2008). Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *J. Autom. Reasoning*, **40**(1), 1–33.

PAPADIMITRIOU C. H. & WOLFE D. (1988). The Complexity of Facets Resolved. *Journal of Computer and System Sciences*, **37**(1), 2–13.

PEARL J. (1990). System Z : A natural ordering of defaults with tractable applications to nonmonotonic reasoning. In R. PARIKH, Ed., *Proceedings of Theoritical Aspects of Reasonning about Knowledge*, p. 121–135, San Mateo : Morgan Kaufmann Publishers.

REITER R. (1980). A logic for default reasoning. *Artificial Intelligence*, **13**, 81–132.