État de l'art de l'ASP avec variables et de ses extensions au premier ordre : études théoriques et solveurs

Délivrable - Projet ANR ASPIQ
Rapport de recherche - juin 2013

Résumé

Ce rapport présente un état de l'art des travaux importants concernant le traitement de l'ASP avec variables ainsi qu'au premier ordre tant d'un point de vue théorique que d'un point de vue pratique. La question du traitement des variables en ASP se situe à deux niveaux. D'une part, les travaux sur les solveurs concernent la manière de prendre en compte les variables en traitant directement les règles en évitant la phase d'instanciation initiale du programme, courante dans les solveurs efficaces et qui ramène le programme à un programme propositionnel. D'autre part, certains travaux théoriques concernent le traitement d'un « vrai » premier ordre dans l'ASP, en particulier avec l'utilisation de quantificateurs existentiels.

1 Introduction

L'Answer Set Programming (ASP) est un formalisme approprié pour représenter de nombreux problèmes issus de l'Intelligence Artificielle dans lesquels l'information disponible est incomplète comme dans le raisonnement non-monotone, la planification, le diagnostic, etc. D'un point de vue général, l'ASP est un cadre couvrant plusieurs sémantiques déclaratives pour différentes sortes de programmes logiques. En ASP, l'information est codée par des règles logiques et les solutions sont obtenues par des ensembles d'information appelés modèles stables. Chaque modèle est un ensemble minimal d'atomes (ou littéraux) contenant des informations sûres (des faits) et les déductions sont obtenues en appliquant certaines des règles par défaut. Ainsi, les conclusions dépendant des informations présentes et absentes, elles forment un ensemble cohérent d'hypothèses et représentent un point de vue rationnel du monde décrit par les règles.

L'ASP offre à la fois un modèle formel valide et des systèmes opérationnels. C'est aussi un cadre commode pour coder et résoudre des problèmes combinatoires. Plusieurs systèmes opérationnels sont disponibles aujourd'hui pour traiter l'ASP. Smodels [39, 37], Clasp [15, 16] et DLV [35] sont les plus connus d'entre eux. Cependant, ces solveurs travaillent sur des règles propositionnelles après une étape préalable d'instanciation

des variables. D'un autre côté, certains solveurs traitent le programme avec variables en travaillant directement sur les règles qui ne sont pas propositionnelles et en limitant l'instanciation [8, 24, 25, 9].

D'autre part, l'étude de l'utilisation de l'ASP pour le web sémantique est une thématique en plein essor. Certains travaux étudient le lien entre ASP et les logiques de description (DL). Ces travaux proposent de combiner le cadre des DL avec l'ASP. On obtient alors des programmes ASP hybrides dans lesquels certains atomes spéciaux du programme ASP font le lien avec un système de DL ou des programmes ASP standard énormes dus à la multiplicité du nombre de règles ASP nécessaires pour coder les DL. Si on veut pouvoir coder toutes les informations en ASP standard, il faut pouvoir prendre en compte certains aspects des DL (comme les variables existentielles en tête de règle) c'est-à-dire avoir un ASP au premier ordre.

Ce rapport a pour objet de présenter un tour d'horizon concernant l'ASP avec variables ainsi qu'au premier ordre.

Tout d'abord, nous présentons quelques résultats de décidabilité sur l'ASP avec variables. Ensuite, nous étudions les solveurs ASP qui traitent les informations avec variables et non au niveau propositionnel comme la plupart d'entre eux le font. Ainsi la phase d'instanciation (en anglais, *grounding*), habituellement utilisée en pré-traitement du solveur propositionnel est effectuée avec parcimonie lors du calcul des modèles. Nous expliquons le principe de l'instanciation d'une règle dans les solveurs avec variables. Ensuite, nous nous focalisons sur la procédure complète du solveur ASPeRiX qui est l'un de ces solveurs avec variables. Enfin, nous présentons quelques travaux théoriques qui donnent des orientations importantes pour les recherches actuelles concernant l'ASP au premier ordre.

2 Décidabilité de l'ASP avec variables

Le caractère décidable de l'ASP avec variables sans symbole de fonction est un élément déterminant de son succès. L'ajout des symboles de fonction (non interprétés) introduit l'indécidabilité du raisonnement si des restrictions ne sont pas posées. Le raisonnement revêt ici plusieurs formes : le raisonnement *ground* qui consiste en la présence d'un atome sans variable dans les conséquences d'un programme et le raisonnement *non-ground* qui consiste en le calcul de l'ensemble des solutions pour une question avec variables. Un autre problème est aussi d'importance : le test de cohérence qui vérifie si au moins un modèle stable existe. Pour conserver l'attrait de l'ASP avec des symboles de fonction, il est crucial de mettre en exergue des classes de programmes dont les raisonnements *ground* ou *non-ground* et le test de cohérence restent décidables. La complexité du *test d'appartenance* à une de ces classes est aussi une question d'importance car elle en détermine le caractère effectif ou non dans les solveurs ASP.

Les différentes classes de programmes dont les raisonnements et le test de cohérence sont décidables forment trois groupes. Le premier groupe est celui des programmes pour lesquels l'instanciation du programme est finie. Ce groupe comprend la classe des programmes finiment *ground* [6],

la classe des programmes à arguments restreints [26], la classe des programmes à domaines finis [6], la classe des programmes ω-restreints [38] et enfin la classe des programmes λ-restreints [17]. Les tests d'appartenance aux classes des programmes à arguments restreints, à domaines finis, ω-restreints et λ-restreints sont décidables mais le test d'appartenance à la classe des programmes finiment *ground* est seulement semi-décidable. Ce groupe correspond à une approche *bottom-up* du calcul des modèles stables qui passe par une phase d'instanciation préalable du programme avec variables via le *grounder* pour ensuite calculer le modèle sur un programme propositionnel. Le fait que l'instanciation du programme amène à un programme propositionnel fini garantit la décidabilité des raisonnements et du test de cohérence.

Le deuxième groupe est celui des programmes pour lesquels un nombre fini d'atomes est nécessaire pour répondre à une question. Ce groupe comprend la classe des programmes FP2 [4], la classe des programmes finiment résursifs stratifiés/positifs [7, 1] et la classe des programmes finitaires [5]. Ce groupe correspond à une approche *top-down* de recherche d'un ensemble de substitutions pour une question. Pour la classe des programmes FP2, le raisonnement *ground* est décidable tandis que le raisonnement *non-ground* est non calculable, le test de cohérence est aussi décidable et le test d'appartenance à cette classe l'est aussi. Cette classe de programme a une expressivité particulièrement limitée. Pour la classe des programmes finiment résursifs stratifiés/positifs, le raisonnement *ground* est décidable tandis que le raisonnement *non-ground* est non calculable, le test de cohérence est décidable et le test d'appartenance à cette classe est semi-décidable. Pour la classe des programmes finitaires, le raisonnement *ground* est (indirectement) décidable, le raisonnement *non-ground* est non calculable, le test de cohérence est non décidable ainsi que le test d'appartenance à cette classe.

Le troisième et dernier groupe est celui des programmes dont la représentation des modèles stables est finie sous la forme de forêts d'arbres. Ce groupe comprend la classe des programmes FDNC [36] et la classe des programmes bidirectionnels [10]. Pour la classe des programmes FDNC, le raisonnement *ground* et le test de cohérence sont décidables, le raisonnement *non-ground* est calculable et le test d'appartenance à cette classe est aussi décidable. Les programmes de cette classe sont particulièrement contraints au niveau expressivité. Pour la classe des programmes bidirectionnels, le raisonnement *ground* et le test de cohérence sont décidables, le raisonnement *non-ground* est calculable et le test d'appartenance à cette classe est aussi décidable.

3 Solveurs

L'une des raisons du succès actuel de l'ASP est le développement de solveurs efficaces pour résoudre les problèmes. La plupart des solveurs, dont les plus connus sont `Clasp` [15], `DLV` [35] et, le fondateur, `Smodels` [39], procèdent en deux phases sur des programmes *sûrs* c'est-à-dire que chaque variable apparaissant dans une règle doit apparaître dans le corps positif. La première phase consiste en une instanciation des variables pour

ramener le problème à une version propositionnelle. Ensuite, c'est la version propositionnelle du programme qui est traitée soit par une procédure dédiée soit en traduisant le programme en base logique propositionnelle (comme `ASSAT` [27] par exemple).

La phase d'instanciation peut s'avérer problématique. D'une part, elle peut être trop longue et inutile. Par exemple, l'instanciation peut concerner une règle qui ne sera jamais déclenchée. D'autre part, lorsqu'on traite des quantificateurs existentiels, l'instanciation de certaines règles n'a pas de sens. C'est pourquoi certains solveurs s'appliquent à ne faire l'instanciation qu'à la volée c'est-à-dire au fur et à mesure des besoins durant la phase de résolution. Plusieurs solveurs ont été développé dans cet esprit.

Le premier d'entre eux, `GASP` [8], n'est qu'un simple prototype visant à valider la présentation du principe et n'est plus développé. Il est implémenté en Prolog. Un autre solveur, `ASPeRiX` [25, 24], est le solveur avec variables le plus référencé et est en cours de développement. C'est pourquoi il fait l'objet d'une section dédiée de ce rapport. Il est implémenté en C++. Récemment, un troisième solveur, `Omiga`[9], a vu le jour et est actuellement en développement. Il reprend les principes d'`ASPeRiX` auxquels il a rajouté une phase de propagation plus efficace. Il est implémenté en Java.

Tous ces solveurs suivent la même procédure générale. Ils évitent de faire une instanciation préalable complète du programme. Ils utilisent une procédure par chaînage avant dans laquelle ils instancient les règles « à la volée » c'est-à-dire au fur et à mesure des possibilités.

La construction d'une interprétation partielle pour un programme P se fait grâce à un chaînage avant sur les règles qui sélectionne et instancie une règle à la fois jusqu'à ce qu'un modèle stable soit trouvé. Cette procédure ne nécessite pas de mener à leur terme toutes les instanciations pour avoir une réponse.

Une *interprétation partielle* pour un programme P est un couple $\langle IN, OUT\rangle$ d'ensembles disjoints de littéraux de la base de Herbrand de P. Intuitivement, IN représente les éléments appartenant au modèle en cours de construction et OUT ceux qui en sont exclus. Les littéraux n'apparaissant ni dans IN, ni dans OUT sont *indéterminés*.

Soit une règle instanciée r et une interprétation partielle $I = \langle IN, OUT\rangle$. On dit que :

- r est *supportée* si et seulement si $corps^+(r) \subseteq IN$;
- r est *bloquée* si et seulement si $corps^-(r) \cap IN \neq \emptyset$;
- r est *débloquée* si et seulement si $corps^-(r) \subseteq OUT$;
- r est *déclenchable* si et seulement si r est supportée et débloquée ;
- r est *applicable* si et seulement si r est supportée et non bloquée.

4 Instanciation d'une règle

Nous décrivons ici les principes guidant l'instanciation d'une règle, instanciation qui va être effectuée au fur et à mesure des besoins. Ne considérant que des règles sûres, instancier une règle revient à instancier

son corps positif. Et dans une approche en chaînage avant, les seules instances qui nous intéressent sont celles qui rendent la règle déclenchable ou applicable. L'instanciation des règles va donc être principalement guidée par les atomes instanciés déjà inférés, i.e., présents dans l'ensemble IN.

L'algorithme présenté ici, et utilisé dans le solveur `ASPeRiX`, s'inspire de travaux réalisés sur le solveur DLV [34] basés sur une technique d'évaluation semi-naïve [41]. L'objectif est de trouver une substitution pour l'ensemble des atomes du corps d'une règle r compte tenu des éléments présents dans IN et dans OUT. Pour cela, une substitution partielle θ est construite au fur et à mesure que l'on trouve des valeurs possibles pour les variables des atomes du corps de la règle r. On considère que les atomes a_1, a_2, ..., a_n du corps de la règle r sont ordonnés par une liste $[a_1, a_2, \ldots, a_n]$ où $premierAtome(r)$ (resp. $dernierAtome(r)$) correspond à a_1 (resp. a_n) et $precedentAtome(r)$ (resp. $prochainAtome(r)$) correspond à l'atome qui précède (resp. qui suit) celui qu'on est en train d'étudier dans la liste. On note $subst(r)$ l'ensemble de toutes les instances de la règle r déjà déclenchées au cours de la recherche de modèle stable. La recherche d'une substitution pour un atome a d'une règle r se fait à l'aide des fonctions $firstMatch$ et $nextMatch$ qui recherchent une substitution de l'atome qui ne mène pas à une substitution déjà produite (donc appartenant déjà à $subs(r)$). Si l'atome a appartient au corps positif de r, on recherche une substitution telle que l'atome substitué appartienne à l'ensemble IN. S'il s'agit d'un atome du corps négatif, on recherche une substitution telle que l'atome substitué appartienne à OUT si le but est d'obtenir une règle déclenchable, ou on vérifie simplement que l'atome substitué n'appartient pas à IN si le but est d'obtenir une règle applicable [1]. Dans les fonctions $firstMatch$ et $nextMatch$ qui suivent, le paramètre γ indique si on recherche une instance de règle déclenchable ou applicable.

- $firstMatch(a, \theta, \gamma, IN, OUT, subst)$ est une fonction qui cherche la première substitution possible pour un atome a compte tenu des éléments présents dans IN et dans OUT, du critère de sélection γ (règle déclenchable ou règle applicable) et de la substitution partielle actuelle θ. $firstMatch$ renvoie vrai et met à jour la substitution partielle θ en cas de succès. Dans le cas contraire, la fonction renvoie faux.
- $nextMatch(a, \theta, \gamma, IN, OUT, subst)$ est une fonction qui cherche la prochaine substitution possible pour un atome a compte tenu des substitutions déjà effectuées.

Pour une règle r, on appelle *variable libre* d'un atome a, une variable X qui apparaît pour la première fois dans le corps de r lorsque l'on parcourt a. Autrement dit, aucun atome qui précède a dans le corps de r ne contient la variable X. Lors de l'instanciation d'une règle, on recherche la prochaine substitution possible pour toutes les variables libres de chaque atome que l'on parcourt et on conserve les substitutions des variables précedemment calculées. Si un atome ne possède aucune variable libre, on vérifie simplement que sa substitution est valide par rapport au critère de sélection γ utilisé, c'est à dire que l'atome substitué $a\theta \in IN$ si

1. dans ce cas, le corps de la règle est ordonné de façon à ce tout atome du corps négatif apparaisse après ceux du corps positif qui contiennent ses variables

$a \in corps^+(r)$, et $a\theta \in OUT$ ou $a\theta \notin IN$ si $a \in corps^-(r)$.

Exemple 1. Soit la règle suivante :

$$a(X, Y, Z) \leftarrow b(X, Y), c(X, Y), d(X, Z).$$

La liste ordonnée du corps de la règle est $[a_1 = b(X, Y), a_2 = c(X, Y), a_3 = d(X, Z)]$ avec :

- $variablesLibres(a_1) = \{X, Y\}$
- $variablesLibres(a_2) = \emptyset$
- $variablesLibres(a_3) = \{Z\}$.

Algorithm 1: *instantiateRule*

```
Function instantiateRule(rule, γ, IN, OUT, Subst);
θ ← dernièreSubstitution(rule);
if θ ≠ ∅ then
    /* Recherche la prochaine substitution possible pour le
       dernier atome                                          */
    a ← dernierAtome(rule);
    θ ← θ\substitutionVariablesLibres(a);
    matchFound ← nextMatch(a, θ, γ, IN, OUT, subst(rule));
else
    /* Recherche une première substitution possible pour le
       premier atome                                          */
    a ← premierAtome(rule);
    matchFound ← firstMatch(a, θ, γ, IN, OUT, subst(rule));
while true do
    if matchFound then
        if a ≠ dernierAtome(rule) then
            a ← prochainAtome(rule);
            matchfound ← firstMatch(a, θ, γ, IN, OUT, subst(rule))
        else
            /* Une substitution complète est trouvée          */
            return θ;
    else
        /* Aucune substitution pour l'atome a, on revient sur
           l'atome précédent (si il existe) pour trouver sa
           prochaine substitution possible                    */
        if a ≠ premierAtome(rule) then
            a ← precedentAtome(rule);
            θ ← θ\substitutionVariablesLibres(a);
            matchFound ← nextMatch(a, θ, γ, IN, OUT, subst(rule));
        else
            return NULL;
```

La fonction $instantiateRule$ présente le principe d'instanciation d'une règle $rule$ pour des ensembles IN et OUT constants. Elle débute avec pour valeur de la substitution partielle θ la dernière substitution trouvée pour la règle $rule$, s'il y en a une. Si donc on avait déjà construit une substitution pour $rule$, la fonction recherche alors une nouvelle substitution possible pour la règle. Pour cela, elle recherche la prochaine instance possible du dernier atome de la règle $rule$ en supprimant de θ les substitutions des variables libres de cet atome (grâce à la fonction $substitutionVariablesLibres$) et en faisant appel à la fonction $nextMatch$. Dans le cas contraire (c'est la première tentative d'instanciation de cette règle), $instantiateRule$ recherche une première substitution pour le premier atome du corps de la règle $rule$ à l'aide de la fonction $firstMatch$. Lors de l'exécution de la boucle principale, elle vérifie tout d'abord qu'une substitution a bien été trouvée pour l'atome courant a. Si c'est le cas, elle recherche une première substitution pour l'atome suivant du corps de la règle respectant la substitution partielle θ. Lorsque tous les atomes ont été parcourus, une substitution complète est trouvée. Elle renvoie alors cette substitution. Lorsque l'instanciation d'un atome échoue (aucune substitution possible), elle revient sur l'atome précédent puis met à jour θ en supprimant les substitutions des variables libres de cet atome. Ensuite, elle fait appel à la fonction $nextMatch$ recherchant la prochaine instanciation possible pour cet atome. L'instanciation d'une règle $rule$ échoue lorsque plus aucune substitution n'est possible pour le premier atome.

En réalité, l'algorithme d'instanciation d'une règle r est légèrement plus compliqué que cela car il faut tenir compte des atomes ajoutés dynamiquement aux ensembles IN et OUT durant la recherche et faire en sorte d'essayer une et une seule fois chaque substitution possible. Ainsi, `ASPeRiX` utilise des files d'atomes, dits *atomes à propager*, qui contiennent tous les atomes ajoutés aux ensembles IN et OUT pendant l'instanciation de la règle. Appelons ces files $IN\_a\_propager$ et $OUT\_a\_propager$. Quand toutes les instances d'une règle r, pour des ensembles IN_0 et OUT_0 donnés, ont été générées, on extrait d'une file un atome à propager a_p dont le symbole de prédicat p apparaît dans le corps de la règle r. Notons $\langle IN_0', OUT_0' \rangle$ l'interprétation partielle obtenue en ajoutant a_p à $\langle IN_0, OUT_0 \rangle$, et $\langle IN, OUT \rangle$ représente l'interprétation partielle $\langle IN_0 \cup IN\_a\_propager, OUT_0 \cup OUT\_a\_propager \rangle$. Le corps de la règle est ordonné de façon à placer en premier les atomes dont le prédicat est celui de l'atome à propager a_p (ce sont les atomes susceptibles de s'unifier avec a_p). Puis on va marquer successivement chacun de ces atomes de prédicat p. L'atome marqué ne peut prendre comme valeur que celle de l'atome à propager a_p alors que les atomes qui suivent peuvent prendre toute valeur possible dans $\langle IN_0', OUT_0' \rangle$.

Ensuite, si l'instanciation du premier atome échoue, on marque l'atome de prédicat p suivant et on recommence l'instanciation de la règle. Les atomes qui précèdent l'atome marqué peuvent alors prendre comme valeurs toutes les valeurs dans $\langle IN_0, OUT_0 \rangle$ (ce qui exclut la valeur de a_p qui a déjà été essayée) tandis que l'atome marqué ne peut prendre comme valeur que celle de l'atome à propager. Si l'instanciation du premier atome échoue et qu'il n'y a plus d'autre atome à marquer, l'instanciation de la

règle échoue.

Exemple 2. Soit la règle et les ensembles suivants :

$$r_0 = a(X+Y) \leftarrow a(X), b(X,Y), a(Y).$$
$$IN_0 = \{b(1,1), b(1,2)\}$$
$$IN\_a\_propager = \{a(1)\}$$
$$IN_0{}' = \{b(1,1), b(1,2), a(1)\}$$
$$IN = \{b(1,1), b(1,2), a(1)\}$$

On souhaite propager l'atome $a(1)$ en instanciant la règle r_0. Les atomes à marquer (dont le symbole de prédicat est a) du corps de la règle sont $a(X)$ et $a(Y)$. Ils sont placés en début du corps de r_0 qui est ordonné ainsi : $[a_1 = a(X), a_2 = a(Y), a_3 = b(X,Y)]$. L'atome $a_1 = a(X)$ est alors marqué et ne peut prendre comme unique valeur que celle de l'atome à propager $a(1)$. X est donc substitué par la valeur 1 dans θ. Ensuite, l'atome suivant du corps de la règle, $a_2 = a(Y)$, devient l'atome courant et prend comme valeur la première parmi celles présentes dans l'ensemble $IN_0{}'$ qui est également $a(1)$. Y est substitué par la valeur 1 dans θ. On passe alors au dernier atome, $a_3 = b(X,Y)$, qui ne possède aucune variable libre donc on vérifie simplement que l'atome $b(1,1)$ obtenu en substituant $b(X,Y)$ par les valeurs de X et de Y dans la substitution θ appartient bien à IN. Il ne reste plus d'atome à parcourir donc une substitution complète est trouvée. L'atome de tête $a(X+Y)$ prend alors les valeurs de la substitution θ. Ainsi, l'algorithme de recherche en chaînage avant pourra ajouter $a(2)$ à IN ainsi qu'à la file $IN\_a\_propager$.

Ensuite, lors d'une nouvelle tentative d'instanciation de la règle pour l'atome à propager $a(1)$, on repart de la dernière substitution de la règle avec $\theta = \{X/1, Y/1\}$ et on recherche une nouvelle substitution pour l'atome $a_3 = b(X,Y)$. Comme $b(X,Y)$ ne possède aucune variable libre, il ne peut avoir d'autres substitutions que l'actuelle. On revient alors sur l'atome $a_2 = a(Y)$ qui lui non plus ne possède aucune autre substitution dans $IN_0{}'$ ($a(2)$ a été inféré après $a(1)$ et n'est pas dans l'ensemble $IN_0{}'$ actuel). L'atome $a_1 = a(X)$ ne pouvant prendre que la valeur $a(1)$ échoue également. Comme le premier atome a échoué, on marque maintenant l'atome $a_2 = a(Y)$ (à la place de $a(X)$). L'atome $a_1 = a(X)$ ne pouvant prendre comme valeur que les atomes de IN_0, aucune substitution n'est possible. On échoue donc sur le premier atome. Étant donné qu'il ne reste plus d'atomes à marquer, l'instanciation de la règle se termine en échec pour l'atome à propager $a(1)$. Les ensembles deviennent les suivants :

$$IN_0 = \{b(1,1), b(1,2), a(1)\}$$
$$IN\_a\_propager = \{a(2)\}$$
$$IN_0{}' = \{b(1,1), b(1,2), a(1), a(2)\}$$
$$IN = \{b(1,1), b(1,2), a(1), a(2)\}$$

On extrait le prochain atome $a(2)$ de la file à propager. Les atomes $a(X)$ et $a(Y)$ se retrouvent à nouveau atomes à marquer. On recommence l'instanciation de la règle avec l'atome $a_1 = a(X)$ qui est l'atome marqué. X est substitué par la valeur 2 car seule la valeur de l'atome à propager $a(2)$ est autorisée. On passe alors à l'atome suivant $a_2 = a(Y)$ où Y peut être substitué par la valeur 1 car $a(1) \in IN_0{}'$. L'atome $b(X,Y)$ ne possède pas de variable libre et, comme l'atome $b(2,1)$ respectant la substitution

$a(X+Y)$	$\leftarrow$	$a(X)$,	$a(Y)$,	$b(X,Y)$	
		*** Premier parcours ***			
		$a(1)$	-	-	(a_1 marqué)
		$a(1)$	$a(1)$	-	
		$a(1)$	$a(1)$	$b(1,1)$	$\Rightarrow$ instanciation complète
		*** Deuxième parcours ***			
		$a(1)$	$a(1)$	NO	
		$a(1)$	NO	-	
		NO	-	-	$\Rightarrow$ échec
		NO	-	-	(a_2 marqué) $\Rightarrow$ échec

TABLE 1 – Décomposition des instanciations de la règle r_0 pour l'atome à propager $a(1)$ (exemple 1)

$\theta = \{X/2, Y/1\}$ n'appartient pas à l'ensemble IN, l'atome $b(X,Y)$ n'a aucune substitution possible. On tente alors d'instancier $a_2 = a(Y)$ avec sa prochaine valeur possible 2 (car $a(2) \in IN_0{}'$). De nouveau, l'atome $b(2,2)$ respectant la substitution $\theta = \{X/2, Y/2\}$ n'appartient pas à IN et l'atome $b(X,Y)$ n'a aucune substitution possible. On revient alors à $a_2 = a(Y)$ qui n'a plus aucune valeur possible. On revient à $a_1 = a(X)$ qui à son tour n'a plus de valeur possible étant donné que sa seule valeur autorisée était la valeur 2 de l'atome à propager $a(2)$.

Le premier atome marqué ayant échoué, on reprend l'instanciation en marquant le second atome $a(Y)$ et on redémarre l'instanciation de la règle. Le premier atome $a_1 = a(X)$ peut prendre les valeurs de IN_0. X est donc substitué par la valeur 1. L'atome $a_2 = a(Y)$ substitue Y par la valeur 2 de l'atome à propager $a(2)$. L'atome $a_3 = b(X,Y)$ n'a aucune variable libre et, comme $b(1,2)$ respectant la substitution $\theta = \{X/1, Y/2\}$ appartient à IN, une substitution complète est trouvée. L'atome de tête $a(X+Y)$ prend alors les valeurs de la substitution θ. Ainsi, l'algorithme de recherche en chaînage avant pourra ajouter $a(3)$ à IN et à la file $IN\_a\_propager$.

Ensuite, lors d'une nouvelle tentative d'instanciation de la règle r_0, l'atome à propager est toujours $a(2)$. On repart de la dernière substitution de la règle, $\theta = \{X/1, Y/2\}$, et on recherche une nouvelle substitution pour l'atome $a_3 = b(X,Y)$. Comme $b(X,Y)$ ne possède aucune variable libre, il ne peut avoir d'autres substitutions que la substitution actuelle. On revient alors sur l'atome $a_2 = a(Y)$ qui lui non plus ne possède aucune autre substitution car l'atome marqué n'accepte que la valeur 2 de l'atome à propager $a(2)$. L'atome $a_1 = a(X)$ échoue également car plus aucune valeur n'est disponible (il ne peut prendre pour valeurs celles de IN_0, donc ni l'atome à propager $a(2)$, ni $a(3)$ apparu après $a(2)$ dans IN, ne sont possibles). Étant donné qu'il ne reste plus d'atomes à marquer, l'instanciation de la règle en échec se termine pour cet atome à propager. On passe alors à la propagation de l'atome $a(3)$ en tentant à nouveau d'instancier la règle r_0. Cette tentative d'instanciation ne donnera rien.

$a(X+Y)$	$\leftarrow$	$a(X)$,	$a(Y)$,	$b(X,Y)$	
		*** Premier parcours ***			
		$a(2)$	-	-	(a_1 marqué)
		$a(2)$	$a(1)$	-	
		$a(2)$	$a(1)$	NO	
		$a(2)$	$a(2)$	-	
		$a(2)$	$a(2)$	NO	
		$a(2)$	NO	-	
		NO	-	-	$\Rightarrow$ échec
		$a(1)$	-	-	(a_2 marqué)
		$a(1)$	$a(2)$	-	
		$a(1)$	$a(2)$	$b(1,2)$	$\Rightarrow$ instanciation complète
		*** Deuxième parcours ***			
		$a(1)$	$a(2)$	NO	
		$a(1)$	NO	-	
		NO	-	-	$\Rightarrow$ échec

TABLE 2 – Décomposition des instanciations de la règle r_0 pour l'atome à propager $a(2)$ (suite exemple 1)

5 Le solveur ASPeRiX

ASPeRiX [25] est un solveur ASP développé en C++ qui a pour particularité d'intégrer la phase d'instanciation d'un programme ASP à la phase de recherche de modèle stable. En effet, les règles d'un programme ASP sont instanciées à la volée durant le calcul des modèles stables. Cela est rendu possible grâce à une recherche de modèles stables basée sur les règles [24] contrairement à la majorité des solveurs existants qui se base sur les atomes. En outre, ASPeRiX impose que les règles d'un programme ASP soient *sûres* c'est à dire que chaque variable apparaissant dans une règle doit apparaître dans le corps positif.

La recherche de modèle stable d'ASPeRiX pour un programme avec variables P repose sur la construction d'instances de règles à partir des atomes présents dans les ensembles IN et OUT. Ici, on considère que les contraintes sont représentées avec le symbole $\bot$ en tête de règle. Initialement, $IN = \emptyset$ et $OUT = \{\bot\}$ [2]. Ensuite, deux types d'inférences se succèdent tour à tour (voir figure 1) :

- une phase de propagation déterministe qui consiste à déclencher toute instance de règle supportée et débloquée en ajoutant la tête de règle dans l'ensemble IN,
- une phase de point de choix non déterministe qui choisit une instance de règle non monotone supportée et non bloquée r_0 et décide soit de forcer son déclenchement durant la prochaine phase de propagation en ajoutant le corps négatif de la règle dans l'ensemble OUT,

2. Comme les ensembles d'atomes IN et OUT sont disjoints, le symbole $\bot$ ne peut appartenir à IN lors de la construction de l'interprétation partielle $\langle IN, OUT\rangle$ d'un programme P sous peine de provoquer un échec dans la construction du modèle stable en cours

ou alors d'interdire son déclenchement en ajoutant une contrainte $\bot \leftarrow corps^-(r_0)$ au programme ASP permettant d'empêcher que la totalité des atomes du corps négatif de la règle appartienne à OUT.

Tout au long de la recherche, la condition $IN \cap OUT = \emptyset$ est vérifiée. Lorsque plus aucune propagation ni point de choix ne peuvent être effectués, l'ensemble IN représente alors un modèle stable.

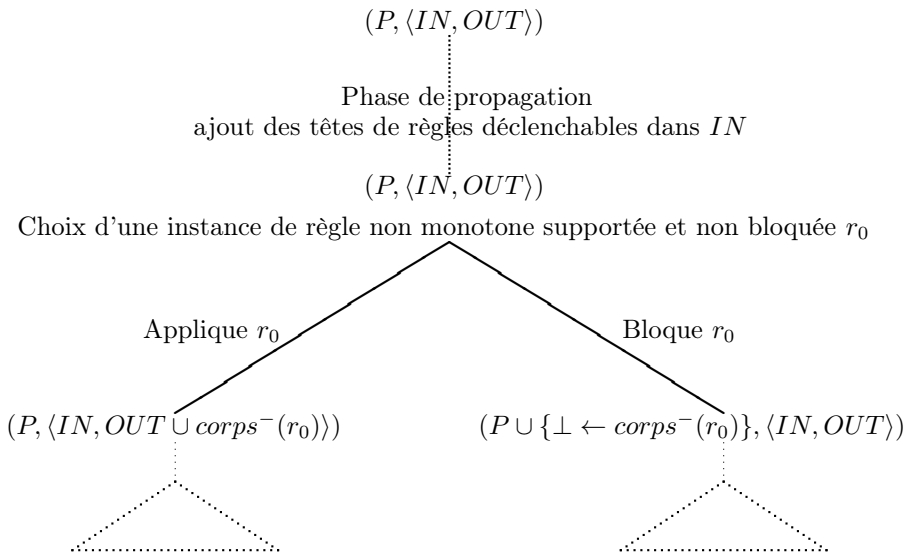

FIGURE 1 – Procédure de recherche d'`ASPeRiX`

5.1 Déroulement de l'algorithme principal

Comme évoqué précedemment, l'algorithme principal d'`ASPeRiX` est basé sur la construction de deux ensembles disjoints d'atomes, IN et OUT, au fur et à mesure de la recherche de modèle stable grâce à une alternance de deux phases principales. D'une part, une phase de propagation qui instancie les règles déclenchables qui peuvent être déduites à partir des éléments dans IN et OUT et ajoute les têtes de règles dans l'ensemble IN. D'autre part, une phase de point de choix qui force ou interdit le déclenchement d'une règle instanciée non monotone. Par ailleurs, `ASPeRiX` utilise également un ensemble MBT (must be true) d'atomes qui nécessitent d'appartenir au modèle stable pour éviter une contradiction. Cet ensemble contient des éléments qui devront appartenir à l'ensemble IN mais n'ont pas encore prouvé leur appartenance. Par exemple, si on a une contrainte $\bot \leftarrow a, not\ b.$ avec $a \in IN$ et b indéterminé, on peut en déduire que b doit nécessairement appartenir au modèle stable pour que la contrainte ne s'applique pas. b sera alors ajouté à l'ensemble MBT. Cet ensemble est utilisé durant la phase de propagation afin de réduire l'espace de recherche. Ainsi, si durant la phase de propagation on a une règle $c \leftarrow b, not\ d$ avec $b \in MBT$ ($b \notin IN$) et $d \in OUT$, on ajoutera alors

la tête de règle c à l'ensemble MBT.

Les règles d'un programme P sont rangées selon les composantes fortement connexes (CFC) du graphe des dépendances de P. On note $pred(l)$ le prédicat d'un atome a [3]. Les noeuds du graphe des dépendances d'un programme P sont ses prédicats et les arcs sont définis par $\{(p,q)|\exists r \in P,\ p = pred(tête(r)),\ q \in pred(corps(r))\}$. Les composantes fortement connexes $\{C_1, ..., C_n\}$ sont ordonnées de sorte que si $i < j$ aucun prédicat de C_i ne dépend de prédicats de C_j, autrement dit, il n'existe aucun chemin dans le graphe qui a pour origine un prédicat de C_i et pour extrémité un prédicat de C_j. On dit qu'une règle r appartient à une CFC C si son prédicat de tête est inclus dans la composante C.

Exemple 3. Soit le programme P_3 suivant :

$$\left\{\begin{array}{l} n(1). \\ n(X+1) \leftarrow n(X), (X+1) <= 2. \\ a(X) \leftarrow n(X), not\ b(X), not\ b(X+1). \\ b(X) \leftarrow n(X), not\ a(X). \\ c(X) \leftarrow n(X), not\ b(X+1). \end{array}\right\}$$

Les composantes fortement connexes (CFC) du graphe de P_3 sont $C_1 = \{n\}$, $C_2 = \{a, b\}$ et $C_3 = \{c\}$ (voir figure 2).

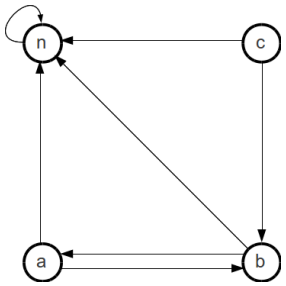

FIGURE 2 – Graphe des dépendances de P_3

L'algorithme de recherche de modèle stable consiste à résoudre une à une les CFC $\{C_1, ..., C_n\}$ d'un programme P en commençant par C_1. Lorsqu'on ne peut plus faire de point de choix sur la CFC courante, on dit que les prédicats de la CFC sont *résolus*. Cela signifie que l'on ne peut plus rien déduire à partir de ces prédicats. Les instances des prédicats de la CFC courante qui ne sont pas dans IN sont alors implicitement ajoutées à OUT de la manière suivante : pour chaque prédicat p appartenant à la CFC courante et pour chaque atome a tel que $pred(a) = p, a \notin IN \Rightarrow a \in OUT$.

Comme évoqué précédemment, les règles d'un programme P sont instanciées à la volée pendant les phases de propagation et de points de choix. Ainsi, le programme propositionnel $ground(P)$ contenant toutes les instances des règles d'un programme P n'est jamais réellement calculé. Les phases de propagation et de points de choix sont exécutées dans l'algorithme de résolution de modèle stable d'`ASPeRiX` par les fonctions γ_{pro} et

3. cette notation est étendue à des ensembles d'atomes

γ_{cho}. La fonction γ_{pro} recherche une instance de règle déclenchable parmi les CFC non résolues (la CFC courante et les suivantes), c'est à dire une règle dont la tête peut être déduite à partir des éléments dans IN et OUT. On note $subst(r)$ l'ensemble de toutes les instanciations qui ont été *propagées* pour la règle r et $Subst = \bigcup_{r \in P} subst(r)$. La fonction γ_{cho} quant à elle choisit une règle à appliquer dans la CFC courante lorsque plus rien ne peut être propagé. Pour qu'une règle soit applicable, il faut que son corps positif soit inclus dans l'ensemble IN et qu'aucun des éléments du corps négatif n'appartienne à l'ensemble IN. Les fonctions γ_{pro} et γ_{cho} sont définies de la façon suivante :

- $\gamma_{pro}(P, IN, OUT, CFC, Subst)$: fonction non déterministe qui sélectionne une règle (ou une contrainte) instanciée r appartenant à une CFC supérieure ou égale à la CFC courante dans le graphe des dépendances de P telle que $corps^+(r) \subseteq IN$, $corps^-(r) \subseteq OUT$ et $r \in ground(P) \backslash Subst$ ou retourne $NULL$ si une telle règle n'existe pas.
- $\gamma_{cho}(P, IN, OUT, CFC, Subst)$: fonction non déterministe qui sélectionne une règle instanciée r appartenant à la CFC courante dans le graphe des dépendances de P telle que $corps^+(r) \subseteq IN$, $corps^-(r) \cap IN = \emptyset$ et $r \in ground(P) \backslash Subst$ ou retourne $NULL$ si une telle règle n'existe pas.

La fonction $solve(P_R,\ P_K,\ IN,\ OUT,\ MBT,\ CFC,\ Subst)$ (voir algorithme 2) décrit le déroulement de l'algorithme de recherche d'un modèle stable pour un programme P. P_K représente l'ensemble des contraintes de P et P_R les autres règles. À noter que l'algorithme de la fonction *solve* présenté ici décrit le calcul d'un modèle stable (ou aucun si le programme est inconsistant) grâce au booléen *stop* qui arrête la recherche dès lors qu'un modèle stable a été trouvé. Cet algorithme peut être étendu aisément pour le calcul d'un nombre arbitraire de modèles stables. Initialement, $IN = \emptyset$, $OUT = \{\bot\}$, $MBT = \emptyset$ et CFC prend pour valeur l'indice de la première composante fortement connexe.

L'étape de propagation consiste à déclencher successivement toute instance de règle supportée et débloquée r_0 à l'aide de la fonction $\gamma_{pro}(P_R \cup P_K, IN \cup MBT,\ OUT,\ CFC,\ Subst)$ qui sélectionne et instancie une unique règle du programme pouvant être déclenchée. Si une telle règle r_0 existe, son atome de tête doit appartenir au modèle stable que l'on recherche. Il est alors ajouté à l'ensemble IN si le corps positif de la règle est inclus dans l'ensemble IN ou bien ajouté à l'ensemble MBT lorsqu'au moins un atome a du corps positif de la règle n'a pas encore prouvé son appartenance à IN ($a \in MBT$ mais $a \notin IN$). Par ailleurs, un atome de tête qui est ajouté à IN doit être retiré de l'ensemble MBT si il y apparaît puisque une preuve de son appartenance au modèle stable a été trouvée. Lorsque plus aucune instance de règle supportée et débloquée r_0 ne peut être déclenchée, on vérifie que les ensembles $IN \cup MBT$ et OUT sont disjoints et on passe à la phase de choix si aucune contradiction n'est détectée.

L'étape de choix cherche à appliquer une instance de règle supportée et non bloquée r_0 grâce à la fonction $\gamma_{cho}(P_R,\ IN,\ OUT,\ CFC,\ Subst)$ qui sélectionne et instancie une unique règle applicable de P_R dont la tête de la règle appartient à la CFC courante. Si r_0 existe, son corps négatif

est ajouté à OUT pour forcer son déclenchement lors de la prochaine phase de propagation et on rappelle la fonction *solve* avec ses nouveaux paramètres. Lorsqu'un rappel récursif à la fonction *solve* mène à un échec, on revient en arrière et on bloque la dernière instance de règle applicable r_0. Pour cela, on fait en sorte que la règle choisie ne puisse plus être utilisée en interdisant le fait que $corps^-(r_0) \subseteq OUT$. On distingue deux cas de figure. Si le corps négatif de r_0 ne contient qu'un seul atome alors celui-ci est ajouté à l'ensemble MBT. Sinon, on ajoute une contrainte qui empêche l'ensemble des atomes du corps négatif d'appartenir à OUT [4].

Lorsque plus aucun point de choix n'est possible, la CFC courante peut être résolue. Avant de passer à la résolution des CFC suivantes, on s'assure qu'aucun élément de MBT n'est une instance d'un prédicat de la CFC courante. Si un tel élément existe dans MBT, les ensembles MBT et OUT ne seraient plus disjoints. En effet, lorsqu'une CFC est résolue, toute instance de prédicat qui n'apparaît pas dans IN est alors implicitement ajoutée à OUT. Ainsi, si un élément de MBT est une instance d'un prédicat de la CFC courante, un échec est constaté. On revient alors au dernier point de choix et on bloque la règle précédemment choisie. Lorsque la dernière CFC est résolue, l'ensemble IN représente un modèle stable de P si aucune contrainte ne peut être appliquée lors de l'appel à la fonction γ_{check} définie de la manière suivante :

- $\gamma_{check}(P, IN, OUT, CFC)$: fonction qui vérifie s'il existe une contrainte c tel que $corps^+(c) \subseteq IN$ et $corps^-(c) \cap IN = \emptyset$ ou retourne faux dans le cas contraire.

4. Ici on considère seulement les atomes du corps négatif de r_0 appartenant à la CFC courante car ceux des CFC inférieures à la CFC courante sont déjà résolus et par conséquent appartiennent nécessairement à OUT

Algorithm 2: *solve*

```
Function solve(P_R, P_K, IN, OUT, MBT, CFC, Subst);
repeat // Phase de propagation
    r_0 ← γ_pro(P_R ∪ P_K, IN ∪ MBT, OUT, CFC, Subst);
    if r_0 ≠ NULL then
        if (corps^+(r_0) ∩ MBT) ≠ ∅ then
            MBT ← MBT ∪ {tête(r_0)};
        else
            IN ← IN ∪ {tête(r_0)};
            if (tête(r_0) ∈ MBT) then
                MBT ← MBT\{tête(r_0)};
until r_0 = NULL;
if ((IN ∪ MBT) ∩ OUT) ≠ ∅ then // Contradiction détectée
    return false;
else
    r_0 ← γ_cho(P_R, IN, OUT, CFC, Subst);
    if r_0 = NULL then // La CFC courante est résolue
        if (pred(MBT) ∩ pred(CFC)) ≠ ∅ then
            if ¬derniere(CFC) then
                solve(P_R, P_K, IN, OUT, MBT, CFC + 1, Subst);
            else
                if γ_check(P_K, IN, OUT, CFC) then // Il existe une
                contrainte non satisfaite
                    return false;
                else// Un modèle stable est trouvé
                    return IN;
        else// Un atome appartenant à MBT est non prouvable
            return false;
    else//Point de choix
        stop ← solve(P_R, P_K, IN, OUT ∪ corps^−(r_0), MBT, CFC, Subst);
        if ¬stop then
            atomes ← {a ∈ corps^−(r_0)|pred(a) ∈ pred(CFC)};
            if (|atomes| = 1) then
                MBT ← MBT ∪ atomes;
            else
                P_K ← P_K ∪ {⊥ ← ∪_{a_i∈atomes} not a_i};
            stop ← solve(P_R, P_K, IN, OUT, MBT, CFC, Subst);
        return stop ;
```

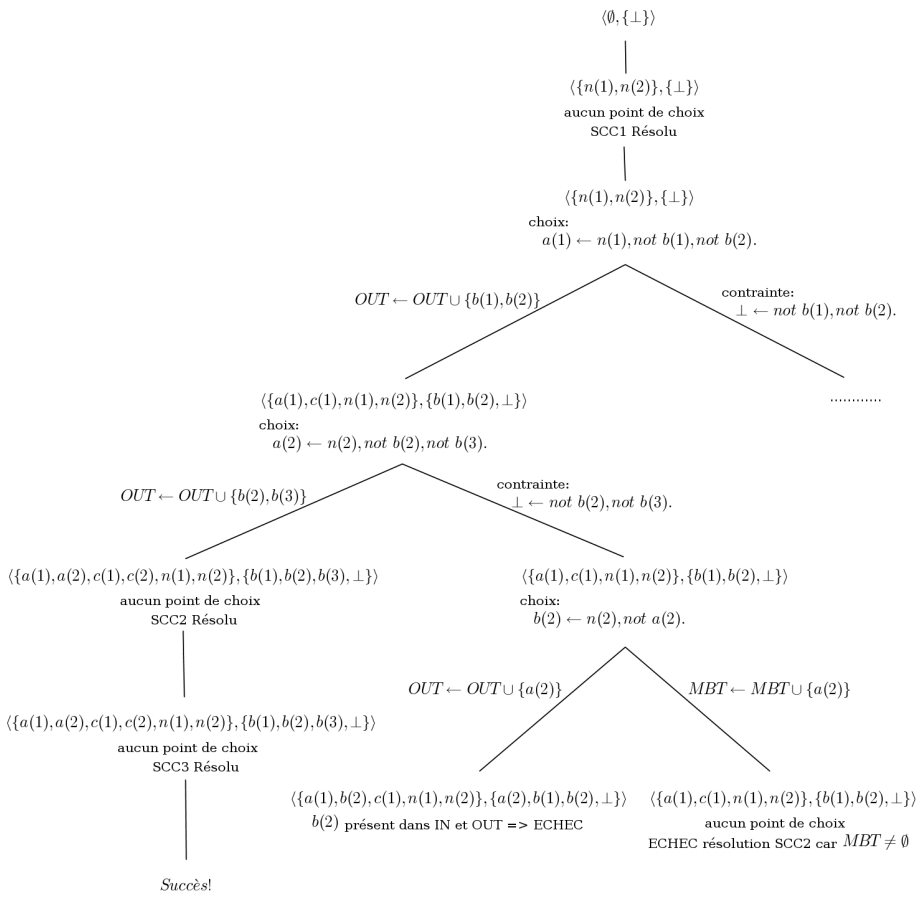

FIGURE 3 – Arbre de résolution des modèles stables de P_3

Exemple 4. Le déroulement de l'algorithme d'`ASPeRiX` pour le programme P_3 (voir exemple 3) est représenté par l'arbre de la figure 3. Au départ, $IN = \emptyset$, $OUT = \{\bot\}$, $MBT = \emptyset$ et la CFC courante est la composante $C_1 = \{n\}$. Après une première propagation, $n(1)$ et $n(2)$ se retrouvent dans IN grâce aux règles déclenchables $n(1).$ et $n(2) \leftarrow n(1), (1+1) <= 2$. Aucun point de choix ne peut être effectué. La première CFC est alors résolue et comme MBT est vide, la composante $C_2 = \{a, b\}$ devient alors la CFC courante.

Ensuite, un premier point de choix $a(1) \leftarrow n(1), not\ b(1), not\ b(2).$ est effectué sur la CFC courante. On force alors la règle à être déclenchée en ajoutant $b(1)$ et $b(2)$ à l'ensemble OUT. Une nouvelle phase de propagation permet de déterminer que $a(1)$ et $c(1)$ appartiennent à IN car $a(1) \leftarrow n(1), not\ b(1), not\ b(2).$ et $c(1) \leftarrow n(1), not\ b(2).$ sont déclenchables. Après cela, un nouveau point de choix est effectué et force le déclenchement de $a(2) \leftarrow n(2), not\ b(2), not\ b(3).$ en ajoutant l'atome $b(3)$ à l'ensemble OUT. Une nouvelle phase de propagation détermine que

$a(2)$ et $c(2)$ appartiennent à IN car $a(2) \leftarrow n(2), not\ b(2), not\ b(3).$ et $c(2) \leftarrow n(2), not\ b(3).$ sont déclenchables. La deuxième CFC est alors résolue car plus aucune règle n'est applicable. Comme MBT est toujours vide, la composante $C_3 = \{c\}$ devient la CFC courante. De la même façon, rien ne peut être propagé ni appliqué dans la CFC C_3. Comme aucune contrainte n'est applicable, on obtient donc un premier modèle stable $\{a(1), a(2), c(1), c(2), n(1), n(2)\}$.

Si l'on souhaite trouver de nouveaux modèles stables, on revient au dernier point de choix $a(2) \leftarrow n(2), not\ b(2), not\ b(3).$ de la composante C_2 et on bloque la règle en ajoutant une contrainte $\perp \leftarrow not\ b(2), not\ b(3)$. Un nouveau point de choix est effectué et force le déclenchement de $b(2) \leftarrow n(2), not\ a(2).$ en ajoutant l'atome $a(2)$ à l'ensemble OUT. Durant la phase de propagation, $b(2)$ est ajouté à l'ensemble IN car $b(2) \leftarrow n(2), not\ a(2).$ est déclenchable. $b(2)$ se retrouve alors dans IN et OUT ce qui mène à un échec.

On revient alors au point de choix $b(2) \leftarrow n(2), not\ a(2).$ de la composante C_2 et on bloque la règle en ajoutant $a(2)$ à l'ensemble MBT. Plus aucun point de choix ne peut être effectué mais l'ensemble MBT contient un atome de la CFC courante ce qui mène à un nouvel échec. L'algorithme revient alors au premier point de choix $a(1) \leftarrow n(1), not\ b(1), not\ b(2).$ de la composante C_2, bloque la règle et recherche à nouveau d'éventuels modèles stables.

5.2 Les fonctions γ

Les fonctions γ ont un rôle majeur dans les deux étapes importantes de la recherche de modèle stable. La fonction γ_{pro} intervient durant la phase de propagation et sélectionne des règles instanciées déclenchables, permettant ainsi d'ajouter les têtes de règles dans l'ensemble IN (ou MBT). La fonction γ_{cho} quant à elle intervient durant la phase de point de choix. Elle sélectionne une règle instanciée applicable dont le déclenchement sera forcé ou interdit durant la prochaine phase de propagation. Enfin, la fonction γ_{check} intervient seulement pour vérifier si une contrainte est applicable lorsque plus aucune autre règle ne peut être déclenchée ou appliquée. Comme le principe du solveur `ASPeRiX` est d'instancier des règles d'un programme ASP à la volée durant la recherche de modèle stable, les fonctions γ requièrent des appels à la fonction *instantiateRule* (voir section 4) permettant de réaliser des instanciations des règles avec variables susceptibles d'être déclenchées ou appliquées.

5.2.1 γ_{pro}

La fonction γ_{pro} recherche une règle à déclencher à partir des éléments déjà présents dans les ensembles IN et OUT. Pour cela, elle recherche une instanciation complète telle que le corps positif soit inclus dans IN et le corps négatif soit inclus dans OUT parmi un ensemble R de règles contenant les prédicats des *atomes à propager* c'est à dire les atomes récemment ajoutés aux ensembles IN et OUT qui n'ont pas encore été utilisés pour la phase de propagation. Ainsi, lorsque un atome a est ajouté à l'ensemble IN (resp. OUT), toutes les règles contenant $pred(a)$ dans leur

corps positif (resp. corps négatif) seront comprises dans l'ensemble R lors du prochain appel à γ_{pro} afin de propager cet atome et obtenir de nouvelles règles déclenchables (si il en existe). Lors du premier appel à la fonction *solve*, l'ensemble R contient toutes les règles qui possèdent un prédicat d'un atome apparaissant dans un fait. Lors d'un appel après un point de choix, l'ensemble R contient toutes les règles qui possèdent dans leur corps négatif les prédicats des atomes ajoutés à OUT durant ce point de choix. Enfin, lors d'un appel après le passage à la CFC suivante, l'ensemble R contient toutes les règles qui possèdent dans leur corps négatif un prédicat qui vient d'être résolu afin de traiter les instances ajoutées implicitement à OUT car n'apparaissant pas dans IN.

L'algorithme de la fonction γ_{pro} choisit une règle *rule* parmi l'ensemble R (la première règle de cet ensemble) et tente de trouver une instanciation qui peut être déclenchée. Pour cela, il appelle la fonction *instantiateRule* qui renvoie la prochaine instanciation pouvant être déclenchée pour la règle *rule* si elle existe. Dans le cas où la règle *rule* ne contient plus aucune instanciation déclenchable, γ_{pro} supprime la règle de l'ensemble R et passe à la prochaine règle. Ce processus est réitéré jusqu'à ce qu'une règle instanciée déclenchable soit trouvée ou bien qu'il ne reste plus aucune règle dans l'ensemble R.

Algorithm 3: γ_{pro}

```
Function γ_pro(P, IN, OUT, CFC, Subst);
R ← Ensemble de règles (contraintes inclus) contenant les prédicats des
atomes à propager;
if R ≠ ∅ then
    repeat
        rule ← premier(R);
        /* Recherche une instanciation de la règle avec
           corps⁺ ⊆ IN et corps⁻ ⊆ OUT                          */
        θ ← instantiateRule(rule, γ_pro, IN, OUT, Subst);
        if θ = NULL then
            R ← R\{rule};
    until θ ≠ NULL or R = ∅;
    if θ ≠ NULL then
        /* Une règle instanciée déclenchable est trouvée       */
        subst(rule) ← subst(rule) ∪ {θ};
        return θ(rule);
    else
        return NULL;
else
    return NULL;
```

Exemple 5. Reprenons le déroulement de l'algorithme de recherche de modèle stable pour le programme P_3 (voir exemple 3) représenté par

l'arbre de la figure 3. Après le premier point de choix, c'est-à-dire $a(1) \leftarrow n(1), not\ b(1), not\ b(2).$, les atomes $b(1)$ et $b(2)$ sont ajoutés à l'ensemble OUT pour forcer le déclenchement de la règle. Lors de la phase de propagation qui suit ce point de choix, plusieurs appels à la fonction γ_{pro} sont exécutés. Lors du premier appel, les atomes à propager sont $b(1)$ et $b(2)$ qui viennent d'être ajoutés à OUT et l'ensemble R est constitué de toutes les règles contenant le prédicat b dans leur corps négatif. Cet ensemble R contient alors les règles $a(X) \leftarrow n(X), not\ b(X), not\ b(X+1).$ et $c(X) \leftarrow n(X), not\ b(X+1)$. On choisit la première règle de l'ensemble R et on trouve une instanciation déclenchable $a(1) \leftarrow n(1), not\ b(1), not\ b(2)$. γ_{pro} renvoit alors l'instanciation de la règle et l'algorithme de recherche de modèle stable se charge d'ajouter $a(1)$ dans IN. Lors du prochain appel à γ_{pro}, l'ensemble R doit contenir, en plus de ses règles précédentes, toutes les règles contenant le prédicat a de l'atome à propager $a(1)$ dans leur corps positif (car $a(1)$ a été ajouté à IN). Comme aucune règle ne respecte cette condition, l'ensemble R contient toujours uniquement ses deux règles précédemment ajoutées. γ_{pro} recherche alors une nouvelle instanciation déclenchable de la règle $a(X) \leftarrow n(X), not\ b(X), not\ b(X+1)$. Aucune instanciation ne peut être trouvée et la règle est retirée de l'ensemble R. γ_{pro} recherche ensuite une instanciation déclenchable de la règle $c(X) \leftarrow n(X), not\ b(X+1)$. L'instanciation $c(1) \leftarrow n(1), not\ b(2).$ est alors renvoyée à l'algorithme de recherche de modèle stable qui ajoute $c(1)$ dans IN. Lors de l'appel suivant à la fonction γ_{pro}, les règles contenant dans leur corps positif le prédicat c de l'atome à propager $c(1)$ sont ajoutées à l'ensemble R. Comme précédemment, aucune règle ne respecte cette condition et l'ensemble R contient toujours uniquement la règle $c(X) \leftarrow n(X), not\ b(X+1)$. Une nouvelle instanciation déclenchable est recherchée pour cette règle mais mène à un échec. La règle $c(X) \leftarrow n(X), not\ b(X+1).$ est alors retirée de l'ensemble R qui devient vide. γ_{pro} renvoie alors la valeur $NULL$ et la phase de propagation de l'algorithme de recherche de modèle stable se termine.

5.2.2 γ_{cho}

La fonction γ_{cho} est exécutée lorsque plus aucune règle ne peut être déclenchée alors qu'il reste des CFC à résoudre. Elle cherche une règle instanciée applicable appartenant à la CFC courante (une règle telle que l'atome de tête appartient à la CFC courante). Les règles applicables sont telles que leur corps positif est inclus dans l'ensemble IN et qu'aucun élément de leur corps négatif n'appartient à ce même ensemble IN.

L'algorithme de la fonction γ_{cho} présente des similitudes avec celui de γ_{pro}. γ_{cho} recherche une règle applicable parmi un ensemble R de règles de la CFC courante ayant dans leur corps négatif un ou plusieurs prédicats de la CFC courante (les prédicats non résolus). Pour cela, γ_{cho} choisit la première règle appartenant à cet ensemble R avant de faire appel à la fonction *instantiateRule* recherchant la prochaine instanciation applicable pour la règle étudiée. De manière analogue à γ_{pro}, le processus est réitéré jusqu'à ce qu'une instanciation applicable soit trouvée pour une règle de l'ensemble R ou bien qu'il ne reste plus aucune règle susceptible d'être appliquée dans R.

Algorithm 4: γ_{cho}

```
Function γ_cho(P, IN, OUT, CFC, Subst);
R ← Ensemble de règles de la CFC courante telles que le corps négatif
contient au moins un prédicat non résolu;
if R ≠ ∅ then
    repeat
        rule ← premier(R);
        /* Recherche une instanciation de la règle avec
           corps⁺ ⊆ IN et corps⁻ ∩ IN = ∅                          */
        θ ← instantiateRule(rule, γ_cho, IN, OUT, Subst);
        if θ = NULL then
            R ← R\{rule};
    until θ ≠ NULL or R = ∅;
    if θ ≠ NULL then
        /* Une règle instanciée applicable est trouvée            */
        return θ(rule);
    else
        return NULL;
else
    return NULL;
```

Exemple 6. Reprenons le déroulement de l'algorithme de recherche de modèle stable pour le programme P_3 (voir exemple 3) représenté par l'arbre de la figure 3. Après que la première CFC soit résolue, un premier point de choix est effectué sur la composante courante $C_2 = \{a, b\}$ à l'aide de la fonction γ_{cho}. Les règles de la composante C_2 qui contiennent dans leur corps négatif au moins un des prédicats a ou b de C_2 sont ajoutés à l'ensemble R de règles susceptibles d'être appliquées. Les règles $a(X) \leftarrow n(X), not\ b(X), not\ b(X+1)$. et $b(X) \leftarrow n(X), not\ a(X)$. appartiennent alors à cet ensemble R. γ_{cho} recherche ensuite une instanciation applicable pour la première règle de cet ensemble et le point de choix $a(1) \leftarrow n(1), not\ b(1), not\ b(2)$. est retourné à l'algorithme de recherche de modèle stable. Après une phase de propagation, γ_{cho} recherche une nouvelle instanciation applicable de la règle $a(X) \leftarrow n(X), not\ b(X), not\ b(X+1)$. et le point de choix $a(2) \leftarrow n(2), not\ b(2), not\ b(3)$. est retourné à l'algorithme de recherche de modèle stable. Après une nouvelle phase de propagation, γ_{cho} recherche en vain une nouvelle instanciation applicable de la règle $a(X) \leftarrow n(X), not\ b(X), not\ b(X+1)$. Cette dernière est alors retirée de l'ensemble R et γ_{cho} recherche alors une instanciation applicable de la règle $b(X) \leftarrow n(X), not\ a(X)$. qui mène également à un échec. L'ensemble R se retrouve vide et la fonction γ_{cho} renvoie $NULL$ à l'algorithme de recherche de modèle stable pour signifier que plus aucun point de choix ne peut être effectué sur la CFC courante.

5.2.3 γ_{check}

La fonction γ_{check} s'exécute dès lors que plus aucun point de choix n'est possible pour la dernière CFC. Elle vérifie qu'aucune contrainte contenant dans son corps négatif un ou plusieurs prédicats de la dernière CFC ne peut être appliquée afin de déterminer si l'ensemble IN obtenu est un modèle stable.

L'algorithme de γ_{check} est similaire à celui de γ_{cho}. Il recherche une contrainte instanciée applicable parmi un ensemble C de contraintes dont le corps négatif contient un ou plusieurs des prédicats de la dernière CFC. γ_{check} choisit une contrainte appartenant à l'ensemble C et fait appel à la fonction *instantiateRule* qui recherche une instanciation applicable pour la contrainte. Si aucune contrainte instanciée n'est applicable pour l'ensemble de contraintes C, l'algorithme renvoie faux et l'ensemble IN représente alors un modèle stable du programme.

Algorithm 5: γ_{check}

```
Function γ_check(P, IN, OUT, CFC);
C ← Ensemble de contraintes tel que le corps négatif contient au moins
un prédicat non résolu;
if C ≠ ∅ then
    repeat
        constraint ← premier(C);
        /* Recherche une instanciation de la contrainte avec
           corps⁺ ⊆ IN et corps⁻ ∩ IN = ∅                          */
        θ ← instantiateRule(constraint, γ_check, IN, OUT, ∅);
        if θ = NULL then
            C ← C\{constraint};
    until θ ≠ NULL or C = ∅;
    if θ ≠ NULL then
        /* Une contrainte instanciée applicable est trouvée       */
        return true;
    else
        return false;
else
    return false;
```

6 Études théoriques d'ASP au premier ordre

Nous nous intéressons dans cette section aux études théoriques existantes s'intéressant à l'extension de l'ASP au premier ordre.

Tout d'abord, certains travaux s'intéressent particulièrement aux applications des logiques de description et correspondent dans l'objectif à l'orientation que nous cherchons à mettre en place pour les extensions d'ASP. Ces travaux sont fortement guidés par les applications. Ainsi, dans [28], les auteurs s'attachent à représenter une base de données pratique décrivant des informations en chimie appelée ChEBI (pour database and ontology of Chemical Entities of Biological Interest). Ces informations sont représentées grâce à des ontologies codées en DL. Pour réaliser ce codage, et dans la mesure où les quantificateurs utilisés dans l'ASP standard sont implicitement des quantificateurs universels, les auteurs proposent une extension de l'ASP ciblée sur l'introduction des quantificateurs existentiels dans un cadre restreint à leur application.

Dans ces travaux, les auteurs s'inspirent des règles existentielles pour introduire des quantificateurs existentiels dans les règles non-monotones. L'objectif du travail est d'introduire des quantificateurs existentiels dans les règles non-monotones. Il s'appuie sur les réflexions menées dans le cadre des règles existentielles. Une séparation entre les faits et les règles est effectuée (alors qu'en ASP, toutes les informations sont codées sous forme de règles). Par rapport aux règles ASP traditionnelles, les extensions concernent l'introduction d'un quantificateur existentiel en tête de règle et la représentation des informations par des ensembles d'atomes. D'un autre côté, certaines limitations sont imposées. La principale concerne la construction de la partie négative de la règle qui ne permet l'utilisation que d'un seul *not* (et non de plusieurs comme dans les règles traditionnelles). Le traitement des quantificateurs existentiels se fait par le biais de la skolémisation. Une autre restriction de ce travail concerne le fait qu'un modèle stable unique est calculé (et non pas plusieurs modèles concurrents). Pour pouvoir déterminer ce modèle stable unique, on calcule une stratification basée sur les liens entre les règles (influences positive et négative). Les influences positives et négatives consistent à exprimer les liens entre les atomes présents en commun dans les parties positives et négatives de deux règles. C'est l'existence de cette stratification qui assure l'existence d'un modèle stable unique.

Ce travail, guidé par une application concrète, s'attache à chercher comment trouver une réponse au programme dans le cas d'une extension d'ASP relativement limitée. D'un autre côté, un nombre en forte croissance de travaux commence à s'intéresser à un traitement d'un réel premier ordre en ASP et proposent des sémantiques nouvelles pour traiter le premier ordre. Ces travaux sont pour l'instant purement théoriques et ne s'attachent pas à la mise en œuvre pratique des solutions.

Les discussions concernant les quantificateurs existentiels et la programmation logique sont assez anciennes (par exemple, voir le fil de discussion « Do We Need Existential Quantifiers in Logic Programming »

daté de 2007 [5]).

L'essentiel des travaux théoriques à propos de l'ASP au premier ordre tiennent leur base de [12, 13]. Par exemple, il existe des travaux dans lesquels ont été ajoutées des préférences entre les règles [3, 2]. Dans les travaux initiaux, l'idée de base est d'essayer de donner un formalisme général permettant de définir la sémantique de toutes les constructions existant dans les différents solveurs comme par exemple les règles disjonctives, les règles de choix, les agrégats. Ce formalisme fournit également une base théorique pour étudier les propriétés de programmes. Pour ce faire, les auteurs abandonnent les définitions habituelles de l'ASP à savoir le *grounding*, le réduit et le calcul des modèles à base de point fixe. L'idée est de voir un programme logique comme une formule de premier ordre F à laquelle on applique un opérateur SM proche de la circonscription [29, 30]. $SM[F]$ est une formule du second ordre obtenue par une transformation syntaxique de F. Un modèle de F va alors être dit *stable* s'il satisfait $SM[F]$. Cette sémantique a été étendue dans [21, 14] pour traiter les agrégats, puis dans [22] pour introduire les quantificateurs généralisés.

Un autre sémantique pour l'ASP de premier ordre est celle de la logique de l'équilibre quantifiée (QEL pour *Quantified Equilibrium Logic*) [33], une extension au premier ordre de la logique de l'équilibre [31]. Ces logiques sont basées sur les logiques du *here-and-there*. Un modèle de l'équilibre étant un modèle HT qui est total et minimal. Il est montré dans [13] que, pour une classe de théories correspondant aux programmes, les modèles stables et les modèles de l'équilibre coïncident.

Truszczynski [40] propose une autre caractérisation de l'ASP de premier ordre dans une logique propositionnelle infinitaire (*infinitary propositional logic*). Ces travaux s'inspirent des notions habituelles en ASP de grounding et de réduit. Le grounding d'une formule existentielle (resp. universelle) va se présenter sous forme d'une disjonction (resp. conjonction) éventuellement infinie de formules propositionnelles. Le concept de réduit d'un ensemble de formules propositionnelles est étendu au cas infinitaire. Et les modèles stables des formules de premier ordre sont alors définis comme les modèles stables de leur grounding, ou encore E est un modèle stable de F ssi E est un modèle minimal du réduit du grounding de F par E ($gr(F)^E$). Cette troisième notion de modèle stable est équivalente aux deux précédentes : celle avec l'opérateur SM et celle de la logique de l'équilibre quantifiée. Cette approche est étendue dans [23] pour traiter les quantificateurs généralisés.

Ces approches, dites de premier ordre, s'attaquent à deux limites de l'ASP classique : une limite syntaxique aux règles propositionnelles et une limite sémantique aux interprétations de Herbrand. D'autres travaux restreignent leur angle d'attaque.

Par exemple, l'ASP *ouvert* (*Open Answer Set Programming*) [19, 20], considère des interprétations de Herbrand étendues où chaque constante est interprétée par elle-même (l'hypothèse du nom unique est donc adop-

5. http ://www.cs.utexas.edu/∼vl/tag/existential

tée), mais où on autorise l'existence d'individus du domaine non explicitement nommés dans le programme. En pratique, on ajoute au vocabulaire du programme un nombre éventuellement infini de nouvelles constantes qui vont dénoter ces individus. Les *answer sets* ouverts peuvent être définis en termes de grounding et de point fixe. Il est montré dans [32] l'équivalence entre cette définition et la logique du *here-and-there* quantifiée si on ne considère que les modèles totaux et qu'on adopte l'hypothèse du nom unique.

Cette idée d'ajouter au vocabulaire un nombre éventuellement infini de nouvelles constantes a été utilisée auparavant dans [18]. Pour cela, les auteurs redéfinissent la sémantique des ensembles réponses par rapport à un programme logique normal. Ces travaux permettent de ne pas connaître obligatoirement toutes les valeurs des domaines de manière explicite et on peut accepter des instanciations des variables qui ne sont pas dans la base initiale pour certains prédicats. En particulier, ces travaux permettent de traiter les exceptions anonymes sur les défauts et amènent à une certaine prudence sur l'existence de certains individus là où l'ASP traditionnel répondra $faux$ si aucun individu n'a été inféré.

7 Conclusion

Dans ce rapport, nous avons étudié l'ASP avec variables ainsi que ses extensions au premier ordre. D'une part, nous avons discuté de l'ASP avec variables. Nous avons d'abord présenté des résultats de décidabilité. Ensuite, nous avons étudié les solveurs ASP avec variables traitant les programmes sans phase préalable d'instanciation. Ces solveurs sont peu nombreux. Nous avons présenté de manière détaillé le processus d'instanciation d'une règle ainsi que le solveur `ASPeRiX` qui est le solveur avec variables le mieux référencé actuellement. D'autre part, nous avons présenté les travaux importants concernant les développements théoriques de l'ASP au premier ordre.

Les recherches sur l'ASP au premier ordre tant d'un point de vue théorique que pratique commencent à être au cœur des préoccupations des recherches en ASP et vont connaître un développement important dans les années à venir.

Références

[1] Mario Alviano, Wolfgang Faber, and Nicola Leone. Disjunctive asp with functions : Decidable queries and effective computation. *TPLP*, 10(4-6) :497–512, 2010.

[2] Vernon Asuncion, Fangzhen Lin, Yan Zhang, and Yi Zhou. Ordered completion for first-order logic programs on finite structures. *Artif. Intell.*, 177-179 :1–24, 2012.

[3] Vernon Asuncion, Yan Zhang, and Yi Zhou. Preferred first-order answer set programs. *à paraître*, 2013.

[4] Sabrina Baselice and Piero A. Bonatti. A decidable subclass of finitary programs. *TPLP*, 10(4-6) :481–496, 2010.

[5] Piero A. Bonatti. Reasoning with infinite stable models. *Artif. Intell.*, 156(1) :75–111, 2004.

[6] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in asp : Theory and implementation. In *ICLP*, pages 407–424, 2008.

[7] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Magic sets for the bottom-up evaluation of finitely recursive programs. In *LPNMR*, pages 71–86, 2009.

[8] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Gasp : Answer set programming with lazy grounding. *Fundam. Inf.*, 96(3) :297–322, August 2009.

[9] Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. Omiga : An open minded grounding on-the-fly answer set solver. In Luis Fariñas del Cerro, Andreas Herzig, and Jérôme Mengin, editors, *JELIA*, volume 7519 of *Lecture Notes in Computer Science*, pages 480–483. Springer, 2012.

[10] Thomas Eiter and Mantas Simkus. Bidirectional answer set programs with function symbols. In *IJCAI*, pages 765–771, 2009.

[11] Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*. Springer, 2009.

[12] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In Manuela M. Veloso, editor, *IJCAI*, pages 372–379, 2007.

[13] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artif. Intell.*, 175(1) :236–263, 2011.

[14] Paolo Ferraris and Vladimir Lifschitz. On the stable model semantics of first-order formulas with aggregates. In *Proceedings of the 2010 Workshop on Nonmonotonic Reasoning*, 2010.

[15] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007. Available at http ://www.ijcai.org/papers07/contents.php.

[16] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. The conflict-driven answer set solver clasp : Progress report. In Erdem et al. [11], pages 509–514.

[17] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In *LPNMR*, pages 266–271, 2007.

[18] Michael Gelfond and Halina Przymusinska. Reasoning on open domains. In *LPNMR*, pages 397–413, 1993.

[19] Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Open answer set programming for the semantic web. *J. Applied Logic*, 5(1) :144–169, 2007.

[20] Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Open answer set programming with guarded programs. *ACM Trans. Comput. Log.*, 9(4), 2008.

[21] Joohyung Lee and Yunsong Meng. On reductive semantics of aggregates in answer set programming. In Erdem et al. [11], pages 182–195.

[22] Joohyung Lee and Yunsong Meng. Stable models of formulas with generalized quantifiers (preliminary report). In Agostino Dovier and Vítor Santos Costa, editors, *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, pages 61–71. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[23] Joohyung Lee and Yunsong Meng. Two new definitions of stable models of logic programs with generalized quantifiers. *CoRR*, abs/1301.1393, 2013.

[24] C. Lefèvre and P. Nicolas. A First Order Forward Chaining Approach for Answer Set Computing. In Erdem et al. [11], pages 196–208.

[25] C. Lefèvre and P. Nicolas. The First Version of a New ASP Solver : ASPeRiX. In Erdem et al. [11], pages 522–527.

[26] Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 489–493, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] Fangzhen Lin and Yuting Zhao. Assat : computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2) :115–137, 2004.

[28] Despoina Magka, Markus Krötzsch, and Ian Horrocks. Computing stable models for nonmonotonic existential rules. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*. AAAI Press/IJCAI, 2013. to appear.

[29] John McCarthy. Circumscription - a form of non-monotonic reasoning. *Artif. Intell.*, 13(1-2) :27–39, 1980.

[30] John McCarthy. Applications of circumscription to formalizing common-sense knowledge. *Artif. Intell.*, 28(1) :89–116, 1986.

[31] David Pearce. Equilibrium logic. *Ann. Math. Artif. Intell.*, 47(1-2) :3–41, 2006.

[32] David Pearce and Agustín Valverde. A first order nonmonotonic extension of constructive logic. *Studia Logica*, 80(2-3) :321–346, 2005.

[33] David Pearce and Agustín Valverde. Quantified equilibrium logic and foundations for answer set programs. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 546–560. Springer, 2008.

[34] Simona Perri, Francesco Scarcello, Gelsomina Catalano, and Nicola Leone. Enhancing dlv instantiator by backjumping techniques. *Ann. Math. Artif. Intell.*, 51(2-4) :195–228, 2007.

[35] Francesco Ricca and Nicola Leone. Disjunctive logic programming with types and objects : The dlv$^+$ system. *J. Applied Logic*, 5(3) :545–573, 2007.

[36] Mantas Simkus and Thomas Eiter. FDNC : Decidable non-monotonic disjunctive logic programs with function symbols. In *LPAR*, pages 514–530, 2007.

[37] Patrik Simons, Ilkka Niemelá, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2) :181–234, June 2002.

[38] Tommi Syrjänen. Omega-restricted logic programs. In *LPNMR*, pages 267–279, 2001.

[39] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 434–438. Springer, 2001.

[40] Miroslaw Truszczynski. Connecting first-order asp and the logic fo(id) through reducts. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, pages 543–559. Springer, 2012.

[41] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.