# Justifications dans les approches ASP basées sur les règles : application au backjumping dans le solveur ASPeRiX

Christopher Beatrix



## ▶ To cite this version:

Christopher Beatrix. Justifications dans les approches ASP basées sur les règles : application au backjumping dans le solveur ASPeRiX. Logique en informatique [cs.LO]. Université d'Angers, 2016. Français. NNT : 2016ANGE0026 . tel-01455527

## HAL Id: tel-01455527

## https://theses.hal.science/tel-01455527

Submitted on 3 Feb 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE BRETAGNE LOIRE

université angers

# Thèse de Doctorat


Christopher BÉATRIX


*Mémoire présenté en vue de l'obtention du*
***grade de Docteur de l'Université d'Angers***
*sous le sceau de l'Université Bretagne Loire*

**École doctorale : Sciences et technologies de l'information, et mathématiques**

**Discipline : Informatique et applications, section CNU 27**
**Unité de recherche : Laboratoire d'Etude et de Recherche en Informatique d'Angers (LERIA)**

**Soutenue le 3 novembre 2016**

## Justifications dans les approches ASP basées sur les règles

### Application au backjumping dans le solveur ASPeRiX

**JURY**

| | |
|---|---|
| Présidente : | **Mme Marie-Laure MUGNIER**, Professeur, Université de Montpellier |
| Rapporteurs : | **M. Philippe BESNARD**, Directeur de Recherche CNRS, Université de Toulouse |
| | **M. Daniel LE BERRE**, Professeur, Université d'Artois |
| Examinateurs : | **Mme Claire LEFÈVRE**, Maître de Conférences, Université d'Angers |
| | **M. Laurent GARCIA**, Maître de Conférences, Université d'Angers |
| Directeur de thèse : | **M. Igor STEPHAN**, Maître de Conférences, Université d'Angers |

# Remerciements

Dans un premier temps, je tiens à remercier Philippe Besnard et Daniel Le Berre pour avoir accepté d'être rapporteurs de cette thèse. Merci également à Marie-Laure Mugnier d'avoir accepté d'être examinatrice de cette thèse.

Je souhaite à présent remercier mon directeur de thèse, Igor Stéphan ainsi que mes co-encadrants, Claire Lefèvre et Laurent Garcia. Je leur suis particulièrement reconnaissant pour leurs précieux conseils, leur disponibilité et leur soutien sans faille dans les moments difficiles.

Je remercie également l'ensemble des membres permanents du LERIA et du département informatique de l'université d'Angers que j'ai pu côtoyer pour la plupart durant l'intégralité de mon cursus universitaire.

Je remercie aussi mes collègues doctorants que j'ai pu croiser durant ces quelques années pour les bons moments passés en leur compagnie. J'adresse d'ailleurs un remerciement spécial à Fabien Garreau qui m'a encouragé à poursuivre mes études par un doctorat durant notre parcours universitaire et avec qui j'ai pu échanger régulièrement sur l'avancée de nos travaux respectifs durant l'intégralité de cette thèse.

Je remercie mes amis, ma famille et ma belle famille pour leur soutien et leur intérêt pour mon travail.

Enfin, je remercie plus que tout ma compagne, Johanne, qui a toujours été là pour moi, qui a partagé mes moments de doutes et qui me comble de bonheur au quotidien.

1

# Introduction

Cette thèse se situe dans le cadre de la programmation par ensembles réponses, plus communément désignée par son nom anglais d'*Answer Set Programming*. L'Answer Set Programming (ASP en abrégé) [25, 26] est un formalisme déclaratif apparu dans les années 90 qui puise ses origines dans la programmation logique et le raisonnement non monotone en s'inspirant de la logique des défauts de Reiter [46]. Au cours de ces dernières années, l'ASP a prouvé son aptitude dans de nombreux domaines d'applications. En effet, l'ASP s'adapte à la représentation des connaissances en intelligence artificielle notamment pour le raisonnement de sens commun [5, 4], le diagnostic [3] ou encore le web sémantique [15]. De plus, il permet de résoudre de nombreux problèmes combinatoires tels que la planification [36], la configuration [28], la satisfaction de contraintes et les problèmes liés à la théorie des graphes [42]. Sa popularité grandissante est particulièrement liée à la mise à disposition de solveurs performants qui ont démontré tout l'intérêt de ce formalisme [49, 34, 22].

Un programme ASP est un ensemble de règles qui codent de manière déclarative le problème à résoudre. Les *answer sets* (ou *modèles stables*) du programme représentent les solutions du problème. Les règles sont généralement exprimées au premier ordre c'est-à-dire avec des variables. Elles sont vues comme une représentation compacte de l'ensemble de toutes leurs instanciations possibles dans l'univers de Herbrand.

Pour illustrer l'utilisation de l'ASP, nous donnons deux exemples, le premier encodant un problème en intelligence artificielle, le second encodant un problème combinatoire. Le premier exemple est l'exemple canonique pour le raisonnement par défaut en présence d'informations incomplètes, constitué de cinq règles.

$oiseau(titi).$
$autruche(lola).$
$oiseau(X) \leftarrow autruche(X).$
$vole(X) \leftarrow oiseau(X), not\ autruche(X).$
$non_vole(X) \leftarrow autruche(X).$

Les deux premières règles expriment que *titi* est un oiseau et *lola* une autruche. Les règles suivantes expriment qu'une autruche est un oiseau, qu'un oiseau vole sauf si c'est une autruche et qu'une autruche ne vole pas. Ces règles permettent de déduire que *titi* vole et que *lola* est un oiseau qui ne vole pas. Le fait que *titi* vole est inféré grâce à la quatrième règle qui permet

de déduire qu'un oiseau vole à moins qu'on dispose de l'information que c'est une autruche. L'unique answer set du programme est
$\{oiseau(titi), vole(titi), autruche(lola), oiseau(lola), non_vole(lola)\}$
et correspond à la solution du problème, c'est-à-dire ici aux informations que l'on peut déduire à propos de *titi* et *lola*.

Le second exemple encode un problème combinatoire : la 2-coloration d'un graphe. Il fonctionne selon le principe générer/tester : on génère les colorations possibles pour les sommets du graphe et on teste que deux sommets adjacents ont des couleurs différentes.

$sommet(1).$
$sommet(2).$
$arc(1, 2).$
$rouge(X) \leftarrow sommet(X), not\ bleu(X).$
$bleu(X) \leftarrow sommet(X), not\ rouge(X).$
$\leftarrow rouge(X), rouge(Y), arc(X, Y).$
$\leftarrow bleu(X), bleu(Y), arc(X, Y).$

Les trois premières règles encodent un graphe à deux sommets et un arc. Les deux règles suivantes permettent de générer toutes les colorations possibles pour les sommets : un sommet qui n'est pas bleu doit être rouge, et vice-versa. Enfin, les deux dernières règles sont des contraintes qui permettent d'interdire que deux sommets adjacents aient la même couleur. Le problème a donc deux solutions de coloration qui correspondent aux deux answer sets du programme : $\{sommet(1), sommet(2), arc(1, 2), bleu(1), rouge(2)\}$ et
$\{sommet(1), sommet(2), arc(1, 2), rouge(1), bleu(2)\}$.

Dans cette thèse, nous nous intéressons plus particulièrement à l'aspect résolution des problèmes c'est-à-dire, en ASP, au calcul des answer sets. En pratique, pour déterminer les answer sets d'un programme, la plupart des solveurs actuels opèrent en deux phases. D'une part, une phase d'instanciation que l'on nomme généralement *le grounding* qui consiste à transformer le programme ASP initial $P$ en un programme propositionnel $P'$ ne contenant plus de variable mais conservant les mêmes answer sets que le programme $P$. D'autre part, une phase de résolution calculant les answer sets du programme $P'$ (et donc par conséquent, les answer sets de $P$). Ces solveurs sont basés sur des algorithmes de recherche guidés par les atomes : les nœuds ou points de choix de l'arbre de recherche portent sur l'appartenance ou non d'un atome à l'answer set recherché.

Bien que cette méthode soit efficace dans la plupart des cas, la phase d'instanciation complète préalable à la phase de résolution pose parfois des problèmes irrémédiables car le programme instancié peut être d'une taille considérable par rapport au programme initial. C'est pourquoi des nouveaux solveurs sont apparus ces dernières années avec pour objectif d'intégrer la phase d'instanciation à la recherche d'answer sets [7, 33, 10]. Ces solveurs travaillent directement à partir du programme au premier ordre, en instanciant les règles au fur et à mesure des besoins. Les points de choix de l'arbre de recherche portent sur le déclenchement ou non d'une règle instanciée à la volée. On parle alors d'approche guidée par les règles, par opposition aux approches guidées par les atomes.

Le solveur `ASPeRiX` [33, 32, 31] qui fait l'objet de cette thèse appartient à cette dernière catégorie de solveurs. `ASPeRiX` est développé en C++ au LERIA à Angers depuis une dizaine d'années. Il est resté à l'état de prototype dans le sens où la procédure de recherche n'a pas été optimisée, et le logiciel lui même n'était pas documenté. Un gros travail a donc été nécessaire pour lire le code d'`ASPeRiX` et comprendre son fonctionnement afin de décrire précisément les différents algorithmes utilisés. Ce travail préliminaire a également permis de mettre en lu-

mière quelques bugs ou insuffisances et de les corriger. Quelques extensions ont également été implémentées.

Parmi les améliorations à apporter à ASPeRiX, nous nous sommes particulièrement intéressés au parcours de l'arbre de recherche qui était implémenté par un backtracking systématique. Lorsqu'une branche gauche d'un nœud de l'arbre de recherche échoue, autrement dit elle n'aboutit pas à un answer set, on parcourt systématiquement sa branche droite en proposant de bloquer la règle du point de choix au lieu de la déclencher. De la même façon, lorsqu'une branche droite d'un nœud de l'arbre échoue, on remonte au nœud précédent (s'il existe) et l'on réitère ce processus jusqu'à ce que tous les nœuds de l'arbre soient parcourus. Cette technique de retour-arrière est à la base de l'implémentation des algorithmes de recherche. Cependant, dans certains cas il est inutile de parcourir certaines branches de l'arbre si l'on est certain qu'elles aboutiront à des échecs.

Le backtracking intelligent ou plus communément appelé *backjumping* est une technique issue de la résolution du problème SAT [41] qui consiste à éviter de parcourir des branches vouées à l'échec en remontant au nœud de l'arbre le plus récent qui participe à l'échec de la branche. Cette technique a déjà été adaptée avec succès dans des solveurs basés sur des atomes notamment dans DLV [47]. L'idée est donc d'adapter le backjumping aux spécificités du solveur ASPeRiX afin de parcourir plus efficacement l'arbre de recherche.

Ces améliorations du parcours de l'arbre de recherche nécessitent de comprendre les raisons de l'échec d'une branche (ou d'un sous-arbre) de manière à éviter de parcourir d'autres branches qui sont vouées au même échec. Nous nous sommes donc intéressés à la notion de *justification*.

En programmation logique, les *justifications* permettent d'expliquer pourquoi certaines propriétés sont vérifiées. On s'intéresse par exemple souvent aux raisons pour lesquelles un littéral appartient ou non à un answer set avec pour objectif d'aider l'utilisateur à comprendre le fonctionnement du programme [45, 48, 54] et, éventuellement, à le déboguer [23, 13, 9]. Dans les systèmes de diagnostic ou les systèmes de décision, il peut être essentiel de comprendre pourquoi le système prend une décision ou, à l'inverse, pourquoi une décision que l'utilisateur aurait pu attendre n'est pas prise. De la même manière, le débogage nécessite de comprendre pourquoi une solution non désirée est trouvée, ou pourquoi une solution attendue n'est pas un answer set.

Dans [45], une distinction entre justifications *on-line* et *off-line* est proposée. Une justification off-line est une raison de l'appartenance ou non d'un litteral à un answer set donné (une interprétation totale) ; elle est souvent calculée a posteriori. Une justification on-line est une raison de l'appartenance ou non d'un litteral à un answer set en cours de construction (une interprétation partielle) ; elle peut être calculée pendant le processus de recherche de solution.

Nous nous intéressons dans cette thèse plus particulièrement aux justifications on-line dans les approches basées sur les règles. De ce point de vue, une justification (ou raison) sera un ensemble de règles, chacune ayant un statut particulier lors du calcul en cours (déclenchée, bloquée, ...). Nous allons définir des raisons pour expliquer pourquoi un littéral est dans l'answer set en cours de construction, pourquoi un littéral est resté indéterminé, pourquoi le processus de recherche de solution a échoué, etc. Une étude formelle des propriétés de ces raisons est réalisée. Nous proposons divers théorèmes montrant que les raisons définies sont des conditions suffisantes pour justifier l'assertion en cause (le littéral appartient à l'answer set, le littéral est indéterminé, le calcul échoue...).

Dans une optique de backjumping et d'apprentissage de lemmes, nous nous focalisons en particulier sur les raisons d'échec : comprendre pourquoi un calcul échoue est la base nécessaire pour ne pas explorer les branches de l'arbre de recherche où le même échec va se reproduire. Nous démontrons ainsi que, une fois connues des raisons d'échec, on peut garantir que certaines branches de l'arbre de recherche ne contiennent pas de solution. Ces résultats sont exploités

pour la mise en œuvre du backjumping dans ASPeRiX. Les raisons définies auparavant sont constituées d'un ensemble de règles responsables de l'échec. D'un point de vue pratique, ce qui nous intéresse c'est l'endroit dans l'arbre où ces règles ont été utilisées. Autrement dit, ce qui nous importe c'est le nœud de l'arbre auquel est lié chaque règle responsable de l'échec. Nous redéfinissons alors une version simplifiée des raisons constituée d'un ensemble de nœuds représentant les points de choix de l'arbre de recherche.

Le manuscrit est organisé comme suit. Le chapitre 2 présente le formalisme de l'ASP. Le chapitre 3 présente les principales approches de recherche d'answer sets (solveurs). Le chapitre suivant est consacré à la description d'ASPeRiX : la notion de computation permet de décrire formellement son fonctionnement, puis les algorithmes qu'il utilise sont détaillés, et enfin le langage précis accepté par ASPeRiX est donné. Le chapitre 5 introduit la notion de justification dans les computations et expose divers théorèmes, en particulier sur les raisons d'échec des computations. Ces raisons sont ensuite adaptées pour implémenter du backjumping dans ASPeRiX. C'est l'objet du chapitre 6 qui présente les principes du backjumping que nous avons adopté ainsi que son implémentation dans ASPeRiX. Le manuscrit se termine par quelques perspectives sur la suite de ces travaux.

# Answer Set Programming

Nous présentons ici les bases de l'Answer Set Programming qui sont utilisées dans cette thèse. Nous considérons des programmes logiques au premier ordre sans disjonction.

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

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

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

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

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

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

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

$$c \leftarrow a_1,\ \ldots,\ a_n,\ not\ b_1,\ \ldots,\ not\ b_m. \quad n \geq 0, m \geq 0 \tag{2.1}$$

où $c, a_1, \ldots, a_n, b_1, \ldots, b_m$ sont des atomes. Le symbole $not$ est une négation par défaut (appelée également négation faible). Une règle de cette forme signifie intuitivement "si tous les $a_i$ sont vrais et si l'on peut considérer tous les $b_j$ comme faux, alors on peut conclure que $c$ est vrai".

Pour une règle $r$ (ou par extension un ensemble de règle), on utilise les notations suivantes :

- $t\hat{e}te(r) = c$ sa *tête*,
- $corps^+(r) = \{a_1, \ldots, a_n\}$ son *corps positif*,
- $corps^-(r) = \{b_1, \ldots, b_m\}$ son *corps négatif*,
- $corps(r) = corps^+(r) \cup corps^-(r)$ les atomes de son *corps*,
- $r^+ = t\hat{e}te \leftarrow corps^+(r)$.

Dans le cas où le corps de la règle est vide ($n = m = 0$), on dit que la règle est un *fait* et on le note simplement $c.$ (ou $c \leftarrow .$). On peut également exprimer une contrainte à l'aide d'une règle sans tête considérée équivalente à ($bug \leftarrow \ldots,\ not\ bug.$) avec $bug$ un nouveau symbole n'apparaissant nulle part ailleurs. En outre, on dit qu'une règle $r$ est *monotone* (respectivement *non monotone*) si son corps négatif est vide (respectivement non vide).

Il est à noter qu'ici la tête d'une règle est constituée d'un seul atome mais qu'une extension des programmes logiques normaux dans [26] autorise une disjonction d'atomes en tête de règle.

Un programme logique est dit *défini* s'il est exclusivement composé de règles monotones. Autrement dit, un programme défini ne possède aucune négation par défaut.

Pour chaque programme $P$, on considère que l'ensemble $\mathcal{SC}$ (respectivement $\mathcal{SF}$ et $\mathcal{SP}$) représente tous les symboles de constantes (respectivement de fonctions et de predicats) qui apparaissent dans $P$.

**Définition 1** (Substitution)**.** Une *substitution* est une fonction $\sigma : \mathcal{V} \rightarrow \mathcal{U}$ des variables dans les termes de l'univers de Herbrand notée $[X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n]$ avec :

- $\forall X \in [1, \ldots, n], X_i \in \mathcal{V}$,
- $\forall t \in [1, \ldots, n], t_i \in \mathcal{U}$,
- $\forall i, j \in [1, \ldots, n]$ avec $i \neq j$, $X_i \neq X_j$.

Cette fonction de substitution peut être étendue aux termes et aux atomes.

**Définition 2** (Substitution d'un terme ou d'un atome)**.** Le résultat de l'application d'une substitution $\sigma : \mathcal{V} \rightarrow \mathcal{U}$, constituée de couples $X_i \leftarrow t_i$, à un terme (respectivement à un atome) $t$, noté $\sigma(t)$ est le terme (respectivement l'atome) $t$ dont toutes les occurences des $X_i$ ont été remplacées simultanément par les $t_i$. On dit alors que $\sigma(t)$ est une *instance* de $t$.

**Exemple 1.** Soit $a(X, f(Y)) \in \mathbf{A}$ et $\sigma = [X \leftarrow 1, Y \leftarrow 2]$ une fonction de substitution. Alors $\sigma(a(X, f(Y))) = a(1, f(2))$ est une instance de $a(X, f(Y))$.

Une *règle instanciée* $r'$ issue d'une règle $r$ d'un programme $P$ est une règle dans laquelle chaque variable de $r$ est remplacée par un terme de l'univers de Herbrand $\mathcal{U}$ de $P$.

**Définition 3** (Substitution d'une règle)**.** Le résultat de l'application d'une substitution $\sigma : \mathcal{V} \rightarrow \mathcal{U}$, constituée de couples $X_i \leftarrow t_i$, à une règle $r$, noté $\sigma(r)$ est la règle $r$ dont toutes les occurences des $X_i$ ont été remplacées simultanément par les $t_i$. On dit alors que $\sigma(r)$ est une *instance* de $r$.

**Exemple 2.** Soit $P_2$ le programme suivant : $\left\{ \begin{array}{l} r_1 :\ a(1). \\ r_2 :\ b(X) \leftarrow a(X),\ not\ c(X). \end{array} \right\}$.
Soit la fonction de substitution $\sigma = [X \leftarrow 1]$. Alors $\sigma(r_2) = b(1) \leftarrow a(1),\ not\ c(1).$ est une instance de $r_2$.

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

**Exemple 3.** Le programme

$$P_3 = \left\{ \begin{array}{l} n(1).,\ n(2)., \\ a(X) \leftarrow n(X),\ not\ b(X)., \\ b(X) \leftarrow n(X),\ not\ a(X). \end{array} \right\}$$

admet pour programme sans variables

$$ground(P_3) = \left\{ \begin{array}{l} n(1).,\ n(2)., \\ a(1) \leftarrow n(1),\ not\ b(1)., \\ b(1) \leftarrow n(1),\ not\ a(1)., \\ a(2) \leftarrow n(2),\ not\ b(2)., \\ b(2) \leftarrow n(2),\ not\ a(2). \end{array} \right\}$$

**Définition 4** (Ensemble clos et modèle de Herbrand)**.** Un ensemble d'atomes $X$ inclus dans $\mathcal{A}$ est clos par rapport à un programme défini $P$ si pour toute règle $r$ de $P$ et toute substitution $\sigma$, si $corps(\sigma(r)) \subseteq X$ alors $t\hat{e}te(\sigma(r)) \in X$. On note $Cn(P)$ le plus petit ensemble d'atomes clos par rapport à $P$. $Cn(P)$ est le modèle de Herbrand minimal de $P$.

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

**Définition 5** (Opérateur de conséquence immédiate)**.** Soit $P$ un programme logique défini et $X$ un ensemble d'atomes tel que $X \subseteq \mathcal{A}$. L'*opérateur de conséquence immédiate* $T_P : 2^{\mathcal{A}} \rightarrow 2^{\mathcal{A}}$ est défini de la manière suivante :

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

On définit la suite $T_P^i$ par :

$$\begin{array}{rcl} T_P^0 & = & \emptyset \\ T_P^{i+1} & = & T_P(T_P^i),\ \forall i > 0 \end{array}$$

Cette suite admet un plus petit point fixe $\bigcup_{i \geq 0} T_P^i = Cn(P)$.

**Exemple 4.** Soit le programme logique défini $P_4 = \left\{ \begin{array}{l} a(X) \leftarrow b(X),\ c(X). \\ b(1). \\ c(X) \leftarrow b(X). \\ d(X) \leftarrow c(X). \\ e(X) \leftarrow a(X),\ f(X). \\ f(X) \leftarrow g(X). \end{array} \right\}$.

L'ensemble d'atomes $\{a(1), b(1), c(1), d(1)\}$ est un modèle de Herbrand minimal de $P_4$ car :

- $T_{P_4}^0 = \emptyset$
- $T_{P_4}^1 = T_{P_4}(T_{P_4}^0) = T_{P_4}(\emptyset) = \{b(1)\}$
- $T_{P_4}^2 = T_{P_4}(T_{P_4}^1) = T_{P_4}(\{b(1)\}) = \{b(1), c(1)\}$
- $T_{P_4}^3 = T_{P_4}(T_{P_4}^2) = T_{P_4}(\{b(1), c(1)\}) = \{a(1), b(1), c(1), d(1)\}$

- $T^4_{P_4} = T_{P_4}(T^3_{P_4}) = T_{P_4}(\{a(1), b(1), c(1), d(1)\}) = \{a(1), b(1), c(1), d(1)\}$ qui est le point fixe.

On a donc $Cn(P_4) = \bigcup_{i \geq 0} T^i_{P_4} = \{a(1), b(1), c(1), d(1)\}$.

**Définition 6** (Réduit)**.** Le *réduit* de Gelfond et Lifschitz [25] d'un programme logique normal $P$ par rapport à un ensemble d'atomes $X \subseteq \mathcal{A}$ est le programme logique défini $P^X$ obtenu à partir de $ground(P)$ en supprimant :

- chaque instance de règle ayant un atome $b$ dans son corps négatif tel que $b \in X$ et
- tous les corps négatifs des instances de règles restantes.

Il est exprimé de la façon suivante :

$$P^X = \{\sigma(r^+) \mid r \in P,\ \sigma \text{ est une substitution telle que } \sigma(corps^-(r)) \cap X = \emptyset\}$$

**Exemple 5.** Le réduit du programme $P_5 = \left\{ \begin{array}{l} a(1). \\ b(X) \leftarrow d(X),\ not\ c(X). \\ c(X) \leftarrow not\ d(X). \\ d(X) \leftarrow a(X),\ not\ c(X). \end{array} \right\}$ par l'ensemble d'atomes $S = \{a(1), b(1), d(1)\}$ est le suivant :

$$P_5^S = \left\{ \begin{array}{l} a(1). \\ b(1) \leftarrow d(1),\ \cancel{not\ c(1)}. \\ \cancel{c(1) \leftarrow not\ d(1)}. \\ d(1) \leftarrow a(1),\ \cancel{not\ c(1)}. \end{array} \right\} = \left\{ \begin{array}{l} a(1). \\ b(1) \leftarrow d(1). \\ d(1) \leftarrow a(1). \end{array} \right\}$$

**Définition 7** (Answer set)**.** [25] Soit $P$ un programme logique normal et $X$ un ensemble d'atomes tel que $X \subseteq \mathcal{A}$. $X$ est un *answer set* de $P$ si et seulement si $X = Cn(P^X)$.

**Exemple 6** (answer sets)**.** Le programme $P_5 = \left\{ \begin{array}{l} a(1). \\ b(X) \leftarrow d(X),\ not\ c(X). \\ c(X) \leftarrow not\ d(X). \\ d(X) \leftarrow a(X),\ not\ c(X). \end{array} \right\}$ de l'exemple 5 admet deux answer sets :

- $S_1 = \{a(1), b(1), d(1)\}$ car $P_5^{S_1} = \left\{ \begin{array}{l} a(1). \\ b(1) \leftarrow d(1). \\ d(1) \leftarrow a(1). \end{array} \right\}$ et $Cn(P_5^{S_1}) = \{a(1), b(1), d(1)\} = S_1$.
- $S_2 = \{a(1), c(1)\}$ car $P_5^{S_2} = \left\{ \begin{array}{l} a(1). \\ c(1). \end{array} \right\}$ et $Cn(P_5^{S_2}) = \{a(1), c(1)\} = S_2$.

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

Une autre définition d'un answer set est basée sur la notion de *règles génératrices*. Ces dernières correspondent aux règles qui participent à la construction d'un answer set.

Dans le cas du solveur `ASPeRiX`, ce sont ces règles qui vont guider la recherche des answer sets d'un programme.

**Définition 8** (Règles génératrices)**.** [30] Soit $P$ un programme logique normal et $X$ un ensemble d'atomes tel que $X \subseteq \mathcal{A}$. L'ensemble des *règles génératrices* de $P$, noté $GR_P(X)$, est défini de la manière suivante :
$GR_P(X) = \{\sigma(r) \mid r \in P, \sigma$ est une substitution telle que $\sigma(corps^+(r)) \subseteq X$ et $\sigma(corps^-(r)) \cap X = \emptyset\}$.

**Définition 9** (Règles enracinées)**.** Soit $R$ un ensemble de règles instanciées. $R$ est enraciné s'il existe une énumération $\langle r_1, \ldots, r_n \rangle$ des règles de $R$ tel que $\forall i \in [1..n], corps^+(r_i) \subseteq tête(\{r_j \mid j < i\})$.

Le théorème suivant est inspiré de [30]. Dans [30], $X$ est un answer set d'un programme $P$ si et seulement si $X = Cn(GR_P(X)^\emptyset)$. On peut le reformuler de la manière suivante :

**Théorème 1.** Soit $P$ un programme logique normal et $X$ un ensemble d'atomes tel que $X \subseteq \mathcal{A}$. Alors, $X$ est un answer set de $P$ si et seulement si $X = tête(GR_P(X))$ et $GR_P(X)$ est enraciné.

*Démonstration.* Soit $P$ un programme logique normal et $X \subseteq \mathcal{A}$. Remarquons d'abord que si $GR_P(X)$ est enraciné alors $Cn(GR_P(X)^\emptyset) = tête(GR_P(X))$.

Si $X$ est un answer set de $P$ alors, d'après un théorème de [30], $GR_P(X)$ est enraciné et, toujours selon [30], $X = Cn(GR_P(X)^\emptyset)$. Or, $Cn(GR_P(X)^\emptyset) = tête(GR_P(X))$. Donc $X = tête(GR_P(X))$.

Supposons maintenant que $X = tête(GR_P(X))$ et $GR_P(X)$ est enraciné. On a $Cn(GR_P(X)^\emptyset) = tête(GR_P(X))$, donc $X = Cn(GR_P(X)^\emptyset)$ et, d'après [30], $X$ est un answer set de $P$. □

**Exemple 7.** En reprenant le programme $P_5 = \left\{ \begin{array}{l} a(1). \\ b(X) \leftarrow d(X), not\ c(X). \\ c(X) \leftarrow not\ d(X). \\ d(X) \leftarrow a(X), not\ c(X). \end{array} \right\}$ de l'exemple 5 et l'ensemble d'atomes $S_1 = \{a(1), b(1), d(1)\}$, on a :

$$GR_{P_5}(S_1) = \left\{ \begin{array}{l} a(1). \\ b(1) \leftarrow d(1), not\ c(1). \\ d(1) \leftarrow a(1), not\ c(1). \end{array} \right\}.$$

$tête(GR_{P_5}(S_1)) = \{a(1), b(1), d(1)\} = S_1$
et il existe une énumération $\langle a(1)., d(1) \leftarrow a(1), not\ c(1)., b(1) \leftarrow d(1), not\ c(1).\rangle$ qui vérifie la condition de l'enracinement de $GR_{P_5}(S_1)$.
$S_1$ est donc un answer set.

En revanche, si l'on choisit un ensemble d'atomes $S_2 = \{a(1), b(1)\}$, on a :
$GR_{P_5}(S_2) = \left\{ \begin{array}{l} a(1). \\ c(1) \leftarrow not\ d(1). \\ d(1) \leftarrow a(1), not\ c(1). \end{array} \right\}$ et $tête(GR_{P_5}(S_2)) = \{a(1), c(1), d(1)\} \neq S_2$ donc $S_2$ n'est pas un answer set.

Si on considère maintenant programme $P_7 = \left\{ \begin{array}{l} a(X) \leftarrow b(X). \\ b(X) \leftarrow a(X). \\ c(1). \end{array} \right\}$ et l'ensemble d'atomes $S = \{a(1), b(1), c(1)\}$, on a :

$$GR_{P_7}(S) = \left\{ \begin{array}{l} a(1) \leftarrow b(1). \\ b(1) \leftarrow a(1). \\ c(1). \end{array} \right\}.$$

$tête(GR_{P_7}(S)) = \{a(1), b(1), c(1)\} = S$
mais il n'existe pas d'énumération qui vérifie la condition de l'enracinement de $GR_{P_7}(S)$.
$S$ n'est donc pas un answer set.

En ASP, il est également possible d'utiliser la négation classique $\neg a$ d'un atome $a$ qui signifie que la valeur de vérité de $a$ est faux. La négation classique est nommée *négation forte* et peut être utilisée conjointement avec la négation par défaut dans les programmes ASP. Pour un atome $a$, la négation par défaut $not\ a$ indique l'absence d'une information sur $a$ tandis que la négation forte $\neg a$ assure que l'atome $a$ n'est pas vérifié.

Un *littéral* est défini soit par un atome $a$, soit par la négation forte de cet atome $\neg a$. Un programme logique est alors dit *étendu* s'il est constitué de règles de la forme :

$$c \leftarrow a_1,\ \ldots,\ a_n,\ not\ b_1,\ \ldots,\ not\ b_m. \quad n \geq 0, m \geq 0 \tag{2.2}$$

où $c, a_1, \ldots, a_n, b_1, \ldots, b_m$ sont des littéraux. Une sémantique pour les programmes logiques étendus a été définie dans [26].

**Exemple 8.** Soit le programme logique étendu $P_8$ suivant qui distingue les oiseaux qui volent ou non :

$$\left\{\begin{array}{l} perroquet(coco). \\ manchot(tux). \\ oiseau(X) \leftarrow perroquet(X). \\ oiseau(X) \leftarrow manchot(X). \\ vole(X) \leftarrow oiseau(X), not\ \neg vole(X). \\ \neg vole(X) \leftarrow manchot(X). \end{array}\right\}$$

La règle $(vole(X) \leftarrow oiseau(X), not\ \neg vole(X).)$ exprime qu'un oiseau vole par défaut et la règle $(\neg vole(X) \leftarrow manchot(X).)$ indique que les manchots ne volent pas. Ce programme admet pour unique answer set $\{perroquet(coco)$, $oiseau(coco)$, $vole(coco)$, $manchot(tux)$, $oiseau(tux)$, $\neg vole(tux)\}$.

Il est également possible d'exprimer la négation forte dans les programmes logiques normaux par le biais d'une réécriture de la négation classique. Pour chaque atome $a$ d'un programme, on remplace $\neg a$ par $na$ et on ajoute la contrainte $\leftarrow a, na.$ qui empêche les atomes $a$ et $na$ d'appartenir à un même answer set. Ainsi, les définitions et théorèmes précédents restent valides avec les programmes logiques étendus.

**Exemple 9.** Le programme précédent $P_8$ sur les oiseaux peut être représenté par le programme logique normal suivant :

$$\left\{\begin{array}{l} perroquet(coco). \\ manchot(tux). \\ oiseau(X) \leftarrow perroquet(X). \\ oiseau(X) \leftarrow manchot(X). \\ vole(X) \leftarrow oiseau(X), not\ nvole(X). \\ nvole(X) \leftarrow manchot(X). \\ \leftarrow vole(X), nvole(X). \end{array}\right\}$$

Dans la suite, les programmes logiques évoqués utiliseront la sémantique des programmes logiques normaux tout en autorisant la négation forte.

# 3

# Solveurs ASP

Depuis plusieurs années, plusieurs solveurs dédiés à l'ASP sont apparus pour permettre d'utiliser ce formalisme de représentation logique. Ainsi, plusieurs méthodes de recherche d'answer sets ont été proposées afin de déterminer le plus efficacement possible les answer sets d'un problème donné.

Dans ce chapitre, une présentation des principaux groupes de solveurs est exposée.

## 1 Approche traditionnelle

La manière la plus répandue pour déterminer les modèles stables d'un programme ASP est de guider la recherche à partir des atomes de ce dernier. Pour cela, les solveurs utilisant cette approche calculent les modèles stables d'un programme ASP $P$ en suivant une architecture composée de deux phases distinctes. D'une part, une phase d'instanciation que l'on nomme généralement *le grounding* qui consiste à transformer le programme ASP initial $P$ en un programme propositionnel $P'$ ne contenant plus de variable mais conservant les mêmes modèles stables que le programme $P$. D'autre part, une phase de résolution calculant les modèles stables du programme $P'$ (et donc par conséquent, les modèles stables de $P$).

### 1.1 Phase d'instanciation (grounding)

La phase d'instanciation est réalisée par un programme dédié, *le grounder* (voir figure 3.1). Les principaux sont `Lparse` [50], `Gringo` [24] et le grounder interne du solveur DLV [16, 17]. Leur objectif est de proposer un grounding le plus minimaliste possible tout en conservant les modèles stables du programme initial. Ainsi, ils conservent, dans la mesure du possible, uniquement les règles propositionnelles qui pourront être utiles au solveur durant la prochaine phase en écartant celles qui ne font qu'alourdir davantage le programme instancié.

**Exemple 10.** Soit le programme $P_{gr} = \left\{ \begin{array}{ll} a(1) \leftarrow not\ a(2). & b(2) \leftarrow not\ b(3). \\ a(2) \leftarrow not\ a(1). & b(3) \leftarrow not\ b(2). \\ p(X) \leftarrow a(X), b(X). & \end{array} \right\}$

L'instanciation complète du programme contient les 3 règles :

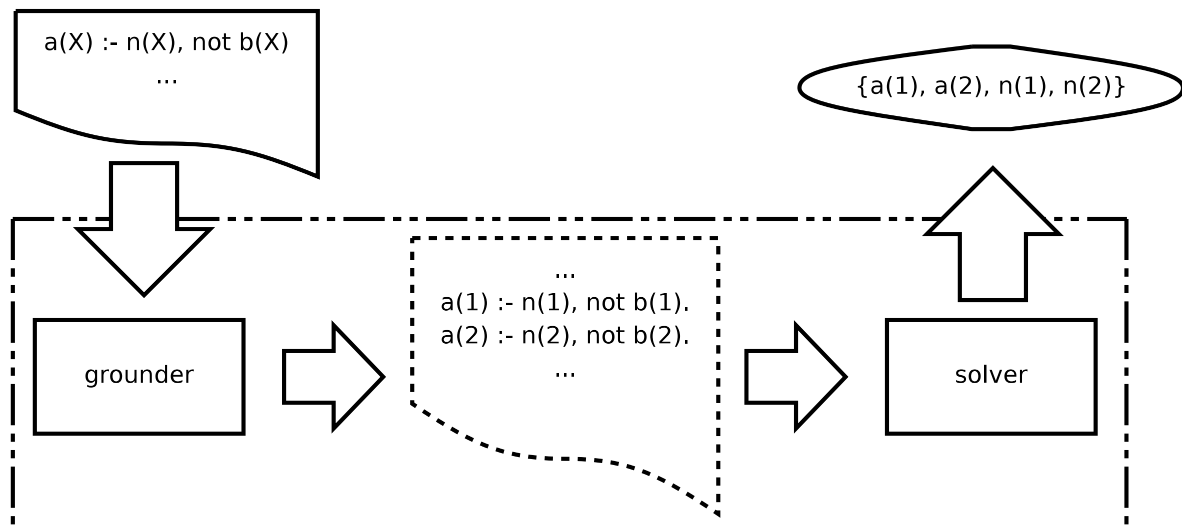



FIGURE 3.1 – Architecture traditionnelle des solveurs ASP

- $p(1) \leftarrow a(1), b(1).$
- $p(2) \leftarrow a(2), b(2).$
- $p(3) \leftarrow a(3), b(3).$

Or la première et la troisième règle sont inutiles car elles contiennent des atomes qui ne sont pas dérivables à partir de $P_{gr}$ ($b(1)$ et $a(3)$ n'apparaissent dans aucune tête de règle). Le grounder va d'abord calculer les ensembles d'atomes pour $a$ et pour $b$ qui sont dérivables à partir de $P_{gr}$ et ainsi ne garder que la deuxième règle dans son programme propositionnel.

De nombreuses techniques ont donc été mises en place pour un grounding efficace. De plus, des restrictions sont imposées pour garantir la décidabilité. Tout d'abord, l'usage de symboles de fonctions est limité car cela peut engendrer un grounding infini. Ensuite, les grounders imposent que les règles du programme ASP initial soient *sûres*, c'est à dire, que chaque variable d'une règle apparaisse dans le corps positif de la règle. C'est le cas notamment du grounder interne de DLV ainsi que `Gringo`. Le grounder `Lparse` impose quant à lui une *restriction de domaine*, ce qui signifie que chaque variable d'une règle doit apparaître dans un domaine de prédicats positif.

La phase d'instanciation ne cède sa place à la phase de résolution que lorsque le programme initial a été totalement instancié ce qui peut parfois être long et coûteux lorsque le grounding aboutit à un nombre exponentiel de règles par rapport au programme initial (voir section 1.3).

## 1.2 Phase de résolution

La phase de résolution s'effectue à partir du programme propositionnel obtenu lors de la phase d'instanciation. Il existe deux grandes approches de résolution [14], l'une qui s'appuie sur les propriétés de la sémantique des answer sets, et l'autre qui transforme les programmes ASP en des problèmes de satisfaisabilité booléenne afin d'effectuer la résolution avec un solveur SAT.

### 1.2.1 Systèmes utilisant les propriétés de la sémantique des answer sets

Les principaux solveurs appartenant à cette catégorie sont `Smodels` [51, 49], `Clasp` [21, 22], DLV [18, 34, 40] et plus récemment `WASP` [1] qui utilise le grounder interne de DLV. L'idée

générale de la résolution consiste à générer dans un premier temps un modèle candidat, puis, dans un second temps, à vérifier que ce modèle candidat est bien un answer set. Pour réaliser cela, les solveurs ASP traditionnels ont une procédure de recherche d'answer sets similaire à la procédure DPLL pour SAT [11]. La recherche d'answer sets alterne des phases de propagation unitaire déterministes et des phases de points de choix non déterministes sur les atomes. Des valeurs de vérité sont affectées aux atomes durant ces deux phases. Ainsi, si un atome se retrouve en conflit (à la fois *vrai* et *faux*) au cours de la recherche, un retour en arrière est effectué. Un modèle candidat est trouvé lorsque la totalité des atomes possède une valeur de vérité. Le solveur vérifie alors que le modèle candidat est bien un answer set et, en cas de succès, retourne tous les atomes ayant la valeur de vérité *vrai*.

**Algorithme 1** : algorithme de recherche d'un answer set

```
1  Fonction résolution(Interprétation I);
2  I ← propage(I);
3  si "conflit présent dans I" alors
4  |  retourner faux;
5  si "aucun atome indéfini dans I" alors
6  |  si "I est un answer set" alors
7  |  |  afficherModèle(I);
8  |  |  retourner vrai;
9  |  sinon
10 |  |  retourner faux;
11 "Choix d'un atome indéfini a selon une heuristique";
12 si résolution(I ∪ {a}) alors
13 |  retourner vrai;
14 sinon
15 |  retourner résolution(I ∪ {not a});
```

L'algorithme 1 présente de manière simplifiée le processus de résolution afin d'obtenir un answer set. La fonction $résolution$ débute avec une interprétation $I$ où tous les atomes sont indéterminés. Une première phase de propagation permet tout d'abord d'affecter des valeurs de vérité aux atomes pouvant être déterminés à partir des règles du programme instancié. Ensuite, plusieurs cas sont possibles.

Tout d'abord, un retour arrière est effectué si la propagation mène à une inconsistance. Dans le cas contraire, lorsque tous ses atomes sont définis, l'interprétation $I$ est bien un answer set si $I$ est un modèle de Herbrand minimal du réduit du programme par rapport à $I$. Sinon, un retour arrière sur le dernier choix d'un atome est réalisé. Enfin, un atome $a$ est choisi selon une heuristique propre à chaque solveur lorsque la propagation n'a pas détecté d'inconsistance et qu'il reste des atomes indéfinis. La fonction $résolution$ est alors rappelée avec $I \cup \{a\}$ ou avec $I \cup \{not\ a\}$ selon l'heuristique. Ainsi, l'atome $a$ permet de réaliser des embranchements ayant un rôle similaire à ceux des solveurs SAT.

Les solveurs `Clasp` et `WASP` présentent de nombreuses similitudes avec les solveurs SAT. L'algorithme de recherche d'answer sets de `Clasp` est basée sur le *Conflict-Driven Constraint Learning* des solveurs SAT. Les différentes règles d'un programme sont exprimées sous forme de contraintes nommées *nogoods* qui vont permettre de déduire les atomes par propagation unitaire. Ainsi, `Clasp` utilise les mêmes stratégies de résolution que les solveurs SAT sans

passer par une conversion du programme au contraire des solveurs qui sont abordés dans la section suivante.

### 1.2.2 Systèmes utilisant une résolution via SAT

Certains systèmes ASP reposent sur une résolution basée sur un solveur SAT. On peut citer notamment `Assat` [37], `Cmodels` [27], `Pbmodels` [38] ou bien encore `LP2ATOMIC` / `LP2SAT` [29]. Leur objectif principal est de profiter de l'efficacité des solveurs SAT pour calculer les answer sets d'un programme ASP. Pour cela, ces différents systèmes transforment le programme logique propositionnel obtenu lors de la phase d'instanciation en formule propositionnelle grâce à la complétion de Clark [6] puis en un ensemble de clauses. Afin de traiter les éventuelles boucles dans le programme ASP, des *formules de boucles* sont ajoutées à la complétion. Par exemple, le programme $\{(a \leftarrow b.), (b \leftarrow a.)\}$ qui possède un cycle $a \rightarrow b \rightarrow a$ a deux modèles dans la logique classique ($\emptyset$ et $\{a, b\}$) mais seulement un seul answer set ($\emptyset$). Ainsi, une formule de boucle ($a \vee b \supset false$) permet de définir qu'aucun de ces atomes ne peut appartenir à un answer set et donc d'exclure le modèle logique qui n'est pas un answer set.

Lorsque le programme ASP est converti en un ensemble de clauses, ces systèmes font alors appel à un solveur SAT existant qui se charge de trouver les answer sets du programme. L'algorithme 2 présente brièvement le processus de résolution d'`Assat` tel qu'il est décrit sur la page web dédiée au solveur[1].

**Algorithme 2** : algorithme de recherche d'un answer set du solveur ASSAT

1 Calculer la completion du programme ASP $P$;
2 Convertir la complétion en un ensemble $C$ de clauses;
3 **répéter**
4     Appeler le solveur SAT sur l'ensemble de clauses $C$ pour obtenir un modèle $M$ (échec si aucun modèle n'existe);
5     **si** *$M$ est un answer set de $P$* **alors**
6         **retourner** $M$;
7     **sinon**
8         Trouver des boucles dans $P$ telle que les formules de boucles ne sont pas satisfaite par $M$;
9         Ajouter leur clauses à l'ensemble de clauses $C$;
10 **jusqu'à** *échec* ;

L'algorithme d'`Assat` commence par calculer la complétion de Clark d'un programme logique propositionnel (obtenu à partir du grounder `Lparse`). Ensuite, il convertit le résultat obtenu en un ensemble de clauses. Puis, il utilise un solveur SAT pour trouver un modèle logique. Il vérifie alors si le modèle est un answer set ou non à partir de la sémantique d'ASP. Dans le premier cas, l'algorithme renvoie le modèle calculé. Dans le second cas, il recherche des formules de boucles insatisfaites par le modèle calculé et les ajoute à l'ensemble de clauses. Il entreprend alors la recherche d'un nouveau modèle à partir d'un solveur SAT. L'algorithme s'arrête si aucun modèle n'est trouvé. Cela signifie qu'aucun answer set n'existe pour le programme.

[1] http ://assat.cs.ust.hk/

## 1.3 Limites de cette approche

L'approche traditionnelle composée d'une phase d'instanciation réalisée par un grounder suivi d'une phase de résolution effectuée par un solveur montre son efficacité pour de nombreux problèmes. Néanmoins, elle n'est pas exempte de tout reproche dans certains cas. En effet, la séparation grounder/solveur provoque parfois une explosion du nombre de règles pour le calcul d'answer sets. Parmi les règles générées par le grounder, il est possible d'avoir des règles totalement inutiles pour la phase de résolution. Ceci est dû au fait que le grounder n'utilise pas la sémantique des answer sets. C'est le cas notamment dans les exemples suivants :

**Exemple 11.** Soit $P_{11}$ le programme ASP suivant :

$$P_{11} = \left\{ \begin{array}{l} a \leftarrow not\ b., \\ b \leftarrow not\ a., \\ \leftarrow a., \\ p(0)., \\ p(X+1) \leftarrow a,\ p(X). \end{array} \right\}$$

Le grounding de $P_{11}$ est infini dans le cas où une borne supérieure pour les entiers n'est pas fixée. Pourtant, $P_{11}$ admet un unique answer set fini $\{b,\ p(0)\}$.

**Exemple 12.** Soit $P_{12}$ le programme ASP suivant :

$$P_{12} = \left\{ \begin{array}{ll} p(1)., p(2)., \ldots,\ p(N)., & \\ a \leftarrow not\ b., & aa(X,Y) \leftarrow pa(X),\ pa(Y),\ not\ bb(X,Y)., \\ b \leftarrow not\ a., & bb(X,Y) \leftarrow pb(X),\ pb(Y),\ not\ cc(X,Y)., \\ pa(X) \leftarrow a,\ p(X)., & cc(X,Y) \leftarrow aa(X,Y),\ X < Y., \\ pb(X) \leftarrow b,\ p(X)., & \leftarrow a. \end{array} \right\}$$

Ici, les $N$ règles contenant $a$ dans leur corps positif comme $(pa(1) \leftarrow a, p(1).)$ et les $N^2$ règles contenant $pa(X)$ dans leur corps positif comme $(aa(1,1) \leftarrow pa(1),\ pa(1),\ not\ bb(1,1).)$ sont inutiles.

Dans ces exemples, la contrainte $(\leftarrow a.)$ empêche chaque ensemble d'atomes contenant $a$ d'être un answer set. C'est pourquoi les règles $(p(X+1) \leftarrow a,\ p(X).)$ de $P_{11}$ et $(pa(X) \leftarrow a,\ p(X).)$ de $P_{12}$ sont totalement inutiles car elles ne pourront jamais servir à générer un answer set.

Par ailleurs, l'exemple $P_{11}$ peut être vu comme un cas de planification classique qui pose énormément de problèmes aux grounders actuels. En planification, l'étape $i+1$ doit être générée seulement si le but n'est pas atteint à l'étape $i$ qui précède. Cependant, le grounder est incapable de déterminer à quel moment arrêter l'instanciation. Pour pallier cette difficulté, le nombre d'étapes maximum autorisées pour atteindre le but est souvent donné en entrée dans les problèmes de planification mais la borne maximum n'est pas toujours estimable lorsqu'on ne connaît pas par avance la solution au problème posé.

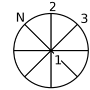
Des travaux ont permis à certains solveurs utilisant la séparation grounder/solveur de résoudre ce genre de problème. C'est le cas notamment d'`iClingo` [20] qui combine le grounder `Gringo` avec le solveur `Clasp` grâce à une interface interne. `iClingo` utilise un procédé d'incrémentation qui instancie les variables étapes par étapes et s'arrête lorsque un answer set est détecté par `Clasp` à une certaine étape.

Un autre souci causé par le fait de séparer l'instanciation de la résolution est que l'instanciation génère tout l'espace de recherche potentiel sans se préoccuper du nombre d'answer sets

que l'on souhaite obtenir. En effet, l'instanciation d'un programme ASP $P$ sera identique que l'on recherche un seul ou bien tous les answer sets de $P$. Il est donc évident que beaucoup d'informations seront inutilisées lorsque l'on recherche un seul answer set d'un programme qui en contient énormément. Les exemples suivants mettent en avant cette particularité.

**Exemple 13.** Soit $P_{13}$ le programme ASP tel qu'il est décrit dans [43] représentant un problème de 3-coloration dans un graphe de $N$ sommets reliés entre eux comme une roue de vélo. $s$ correspond aux *sommets*, $a$ aux *arcs*, $c$ aux *couleurs*, $col(X,Y)$ signifie qu'un sommet $X$ est coloré d'une certaine couleur $Y$ et $ncol(X,Y)$ signifie qu'un sommet $X$ n'est pas coloré d'une certaine couleur $Y$.

$$P_{13} = \left\{ \begin{array}{l} s(1).,\ldots,s(N).,\ \ c(rouge).,\ c(bleu).,\ c(vert).,\\ a(1,2).,\ldots,a(1,N).,\\ a(2,3).,a(3,4).,\ldots,a(N,2).,\\ col(S,C) \leftarrow s(S),\ c(C),\ not\ ncol(S,C).,\\ ncol(S,C) \leftarrow col(S,D),\ c(C),\ C \neq D.,\\ \leftarrow a(S,T),\ col(S,C),\ col(T,C). \end{array} \right\}$$




Si $N$ est pair alors $P_{13}$ n'admet aucun answer set, sinon il en admet 6. Sur cet exemple, les grounders actuels génèrent environ $18N$ règles.

Si $N$ est impair et que l'on suppose qu'il y a un premier answer set dans lequel on a $col(1,rouge)$ alors certaines contraintes deviennent totalement inutiles. Les $N-1$ contraintes qui vérifient que $(\leftarrow a(1,T),\ col(1,rouge),\ col(T,rouge).)$ pour $T \in \{2,\ldots,N\}$ sont nécessaires car elles permettent de vérifier qu'aucun sommet qui est relié à un arc avec le sommet $1$ n'est de la couleur rouge. En revanche, les contraintes $(\leftarrow a(1,T),\ col(1,bleu),\ col(T,bleu).)$, et $(\leftarrow a(1,T),\ col(1,vert),\ col(T,vert).)$ pour chaque $T \in \{2,\ldots,N\}$ sont totalement inutiles étant donné que le sommet $1$ n'est pas coloré en bleu ni en vert. Ainsi, du temps et de la mémoire sont dédiés à ces $2N-2$ contraintes, en vain dans le cas où l'on ne rechercherait qu'un seul answer set.

**Exemple 14.** Soit $P_{14}$ le programme représentant le problème de cycle hamiltonien d'un graphe complet orienté de $N$ sommets, inspiré par [43]. $s$ correspond aux *sommets*, $a$ aux *arcs*, $ch$ signifie qu'un arc appartient au cycle hamiltonien, $nch$ qu'un arc n'appartient pas au cycle hamiltonien, $debut$ correspond au sommet qui *débute* le cycle et $atteint$ signifie qu'un sommet est *atteint*.

$$P_{14} = \left\{ \begin{array}{l} debut(1).,\quad s(1).,\ldots,s(N).,\qquad a(X,Y) \leftarrow s(X),\ s(Y).,\\ ch(X,Y) \leftarrow debut(X),\ a(X,Y),\ not\ nch(X,Y).,\\ ch(X,Y) \leftarrow atteint(X),\ a(X,Y),\ not\ nch(X,Y).,\\ nch(X,Y) \leftarrow ch(X,Z),\ a(X,Y),\ Y \neq Z.,\\ nch(X,Y) \leftarrow ch(Z,Y),\ a(X,Y),\ X \neq Z.,\\ atteint(Y) \leftarrow ch(X,Y).,\\ \leftarrow s(X),\ not\ atteint(X). \end{array} \right\}$$

Ce programme admet $(N-1)!$ answer sets. Les groundeurs actuels génèrent environ $2N^2$ règles avec $ch$ en tête et $2N^3$ avec $nch$ en tête quel que soit le nombre d'answer sets désiré. Par conséquent, le grounding consomme énormément de temps lorsque le graphe possède quelques centaines de sommets même si l'on ne désire qu'un seul answer set.

Dans tous ces exemples, le problème principal est lié au nombre de règles et d'atomes générés lors de la phase d'instanciation qui retarde et complexifie la phase de résolution. Si l'on reprend l'exemple 13, une approche plus naturelle ne construirait pas toutes les instanciations

possibles des règles du programme directement. À la place, elle instancierait certaines variables au fur et à mesure et vérifierait les contraintes. Ainsi, l'espace de recherche serait parcouru complètement par retour-arrière successif. C'est dans cette optique de résolution que d'autres solveurs qui ne séparent pas l'instanciation de la résolution sont apparus.

## 2 Approche guidée par les règles

On a vu précédemment une approche qui consistait à guider la recherche à partir des atomes du programme ASP. Bien que cette méthode soit efficace dans la plupart des cas, la phase d'instanciation complète préalable à la phase de résolution pose parfois des problèmes irrémédiables (voir 1.3). C'est pourquoi des nouveaux solveurs sont apparus ces dernières années avec pour objectif d'intégrer la phase d'instanciation à la recherche d'answer sets.

Ces différents solveurs appliquent un chaînage avant sur les règles qui sont instanciées à la volée durant le processus de résolution. Cela est rendu possible grâce à une recherche d'answer sets basée sur les règles. Dans cette catégorie de solveurs, on peut citer `GASP` [8], `OMiGA`[10] et `ASPeRiX` [33, 32, 31] qui fait l'objet de cette thèse. Ces derniers sont tous basés sur la notion de computation introduite dans [39].

La recherche d'answer sets de ces différents solveurs s'appuie sur la construction d'une interprétation partielle pour un programme $P$ grâce à un chaînage avant sur les règles qui sélectionne et instancie une règle à la fois jusqu'à ce qu'un answer set soit trouvé. Une *interprétation partielle* pour un programme $P$ est un couple $\langle IN, OUT \rangle$ d'ensembles disjoints d'atomes de $P$ où $IN$ représente les éléments appartenant à l'answer set en cours de construction, et $OUT$ ceux qui en sont exclus.

Le principe de résolution général repose sur deux phases distinctes. D'une part, une phase de propagation déterministe ajoute dans l'ensemble $IN$ la tête de toute instance de règle $r$ telle que son corps positif ($corps^+(r)$) est inclus dans l'ensemble $IN$ et son corps négatif ($corps^-(r)$) est inclus dans l'ensemble $OUT$. D'autre part, une phase de point de choix non déterministe choisit, dans un premier temps, une instance de règle non monotone *applicable* $r_0$ telle que son corps positif est inclus dans l'ensemble $IN$ ($corps^+(r_0) \in IN$) et qu'aucun des atomes de son corps négatif n'est inclus dans l'ensemble $IN$ ($corps^-(r_0) \cap IN = \emptyset$), puis, dans un second temps, décide soit de forcer son déclenchement durant la prochaine phase de propagation, soit d'interdire son déclenchement. Un answer set est trouvé lorsque plus aucun choix ne peut être fait (plus aucune règle non monotone n'est applicable).

Le solveur `GASP` est apparu en 2008. Il est implémenté en Prolog et utilise la *programmation logique par contraintes sur des domaines finis* [12]. Chaque instanciation de règle est réalisée en construisant et en résolvant un *problème de satisfaction de contraintes*. Sa résolution est basée sur le calcul de *modèle bien fondé* [53] du programme qui alterne avec des points de choix sur les différentes règles applicables. Ainsi, l'algorithme de `GASP` est basé sur la construction d'un arbre n-aire sur les différentes règles applicables à partir d'une interprétation. Des stratégies d'ordonnancement des règles applicables ont été mises en place pour empêcher le solveur de calculer plusieurs fois un même answer set.

`OMiGA` est apparu en 2012 et est implémenté dans le langage JAVA. Il utilise un réseau Rete[19] pour les instanciations de règles lors de la propagation et des points de choix. Chaque nœud du réseau Rete représente une partie du programme ASP d'entrée et possède son propre espace mémoire. Par ailleurs, `OMiGA` a commencé à introduire quelques techniques de clause learning propres à des solveurs comme `Clasp` et DLV et les a adaptées aux spécificités des solveurs guidés par les règles [55].

ASPeRiX est né à la même période que le solveur GASP. Il est implémenté en C++. Son processus de résolution est détaillé de manière approfondie dans le chapitre suivant où ses différentes particularités sont évoquées.

# 4

# ASPeRiX

ASPeRiX 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 d'answer sets. En effet, les règles d'un programme ASP sont instanciées à la volée durant le calcul des answer sets. Cela est rendu possible grâce à une recherche d'answer set basée sur les règles contrairement à la majorité des solveurs existants qui se base sur les atomes ou qui utilise un solveur SAT après avoir réalisé une transformation du programme ASP.

La recherche d'answer set du solveur ASPeRiX s'appuie sur la construction d'une interprétation partielle pour un programme $P$ grâce à un chaînage avant sur les règles qui sélectionne et instancie une règle à la fois jusqu'à ce qu'un answer set soit trouvé.

## 1 Aspect théorique

On a dit précedemment que les answer sets d'un programme logique au premier ordre peuvent être construits en utilisant un chaînage avant sur les règles. Ici, la notion de *computation* va permettre de construire un answer set d'un programme en instanciant des règles au premier ordre successivement. Cette notion a précédemment été introduite dans [39] dans le cas des programmes propositionnels.

On impose ici que chaque règle d'un programme soit *sûre* ce qui signifie que chaque variable apparaissant dans une règle doit apparaître dans le corps positif. En outre, on autorise ici la présence des littéraux tout en conservant la sémantique des programmes logiques normaux[1]. Par ailleurs, les contraintes seront représentées avec la tête de règle particulière $\perp$.

Une *interprétation partielle* va permettre de distinguer les littéraux qui appartiennent à l'answer set en cours de calcul lors d'une computation et ceux qui ne lui appartiennent pas. Elle est définie de la manière suivante :

**Définition 10** (Interprétation partielle)**.** Une *interprétation partielle* pour un programme $P$ est un couple $\langle IN, OUT\rangle$ d'ensembles disjoints de littéraux inclus dans la base de Herbrand de $P$.

Intuitivement, $IN$ représente les éléments appartenant à l'answer set en cours de construction et $OUT$ ceux qui en sont exclus. Les littéraux n'apparaissant ni dans $IN$, ni dans $OUT$

[1] Le solveur se charge en interne de la réécriture de la négation classique (voir exemple 9 du chapitre 2)

sont *indéterminés*.

**Définition 11** (Type de règle)**.** Soit $r$ une règle, $\sigma$ une fonction de substitution et $I = \langle IN, OUT \rangle$ une interprétation partielle. On dit que :

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

Une `ASPeRiX` computation utilise un chaînage avant qui instancie et déclenche une unique règle à chaque itération selon deux sortes d'inférences : une étape de *propagation* et une étape de *choix*. Le déclenchement d'une règle signifie ajouter sa tête à l'ensemble $IN$.

**Définition 12** ($\Delta_{pro}$ et $\Delta_{cho}$)**.** Soit $P$ un ensemble de règles, $I$ une interprétation partielle et $R$ un ensemble de règles instanciées.

- $\Delta_{pro}(P, I, R) = \{\sigma(r) \mid r \in P, \sigma$ est une fonction de substitution telle que $\sigma(r)$ est déclenchable par rapport à $I$ et $\sigma(r) \notin R\}$.
- $\Delta_{cho}(P, I, R) = \{\sigma(r) \mid r \in P, \sigma$ est une fonction de substitution telle que $\sigma(r)$ est applicable par rapport à $I$ et $\sigma(r) \notin R\}$.

Comme l'ensemble $ground(P)$, les deux ensembles définis précédemment n'ont pas besoin d'être explicitement calculés. Lorsque cela est nécessaire, une règle au premier ordre de $P$ est sélectionnée et instanciée avec des littéraux propositionnels apparaissant dans les ensembles $IN$ et $OUT$ afin de définir une nouvelle règle instanciée de $\Delta_{pro}$ ou $\Delta_{cho}$. L'instanciation complète est toujours possible grâce à l'obligation d'avoir uniquement des règles sûres dans le programme.

Les ensembles $\Delta_{pro}$ et $\Delta_{cho}$ sont utilisés dans la définition suivante d'une `ASPeRiX` computation. L'ajout de $\bot$ dans l'ensemble $OUT$ dès le début de la computation permet de traiter le déclenchement d'une contrainte. Ainsi, si une contrainte est déclenchée (on parle alors de contrainte *violée*), le littéral $\bot$ devrait alors être ajouté à l'ensemble $IN$ et rendre les ensembles $IN$ et $OUT$ non disjoints.

**Définition 13** (`ASPeRiX` computation)**.** Soit $P$ un programme logique. Une *`ASPeRiX` computation* pour $P$ est une suite $\langle R_i, I_i \rangle_{i=0}^{\infty}$ avec $R_i$ un ensemble de règles instanciées et $I_i = \langle IN_i, OUT_i \rangle$ une interprétation partielle qui satisfont les conditions suivantes :

- $R_0 = \emptyset$ et $I_0 = \langle \emptyset, \{\bot\} \rangle$,
- (Révision) $\forall i \geq 1$,

  (Propagation) $R_i = R_{i-1} \cup \{r_i\}$ avec $r_i \in \Delta_{pro}(P, I_{i-1}, R_{i-1})$
  et $I_i = \langle IN_{i-1} \cup \{tête(r_i)\}, OUT_{i-1} \rangle$

  ou (Choix de Règle) $\Delta_{pro}(P, I_{i-1}, R_{i-1}) = \emptyset$,
  $R_i = R_{i-1} \cup \{r_i\}$ avec $r_i \in \Delta_{cho}(P, I_{i-1}, R_{i-1})$
  et $I_i = \langle IN_{i-1} \cup \{tête(r_i)\}, OUT_{i-1} \cup corps^-(r_i) \rangle$

ou (Stabilité) $R_i = R_{i-1}$ et $I_i = I_{i-1}$,

- (Convergence) $\exists i \geq 0,\ \Delta_{cho}(P, I_i, R_i) = \emptyset$.

La computation converge en $IN_\infty = \bigcup_{i=0}^{\infty} IN_i$.

**Exemple 15.** Soit $P_{15}$ le programme suivant :

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

La suite suivante est une `ASPeRiX` computation pour $P_{15}$ :

$$\begin{array}{ll}
I_0 & = \langle \emptyset, \{\bot\} \rangle \\
\\
r_1 & = n(1). \in \Delta_{pro}(P_{15}, I_0, \emptyset) \\
I_1 & = \langle \{\mathbf{n(1)}\}, \{\bot\} \rangle \\
\\
r_2 & = n(2) \leftarrow n(1). \in \Delta_{pro}(P_{15}, I_1, \{r_1\}) \\
I_2 & = \langle \{n(1), \mathbf{n(2)}\}, \{\bot\} \rangle \\
\\
 & \Delta_{pro}(P_{15}, I_2, \{r_1, r_2\}) = \emptyset \\
r_3 & = a(1) \leftarrow n(1), not\ b(1), not\ b(2). \in \Delta_{cho}(P_{15}, I_2, \{r_1, r_2\}) \\
I_3 & = \langle \{n(1), n(2), \mathbf{a(1)}\}, \{\bot, \mathbf{b(1)}, \mathbf{b(2)}\} \rangle \\
\\
r_4 & = c(1) \leftarrow n(1), not\ b(2). \in \Delta_{pro}(P_{15}, I_3, \{r_1, r_2, r_3\}) \\
I_4 & = \langle \{n(1), n(2), a(1), \mathbf{c(1)}\}, \{\bot, b(1), b(2)\} \rangle \\
\\
 & \Delta_{pro}(P_{15}, I_4, \{r_1, r_2, r_3, r_4\}) = \emptyset \\
r_5 & = a(2) \leftarrow n(2), not\ b(2), not\ b(3). \in \Delta_{cho}(P_{15}, I_4, \{r_1, r_2, r_3, r_4\}) \\
I_5 & = \langle \{n(1), n(2), a(1), c(1), \mathbf{a(2)}\}, \{\bot, b(1), b(2), \mathbf{b(3)}\} \rangle \\
\\
r_6 & = c(2) \leftarrow n(2), not\ b(3). \in \Delta_{pro}(P_{15}, I_5, \{r_1, r_2, r_3, r_4, r_5\}) \\
I_6 & = \langle \{n(1), n(2), a(1), c(1), a(2), \mathbf{c(2)}\}, \{\bot, b(1), b(2), b(3)\} \rangle \\
\\
 & \Delta_{pro}(P_{15}, I_6, \{r_1, r_2, r_3, r_4, r_5, r_6\}) = \emptyset \\
 & \Delta_{cho}(P_{15}, I_6, \{r_1, r_2, r_3, r_4, r_5, r_6\}) = \emptyset \\
I_7 & = I_6
\end{array}$$

Cette `ASPeRiX` computation converge en l'ensemble $\{n(1), n(2), a(1), c(1), a(2), c(2)\}$ qui est un answer set pour $P_{15}$.

Lors d'une `ASPeRiX` computation, les étapes de (Propagation) s'effectuent jusqu'à ce que plus aucune règle ne puisse être déclenchée par $\Delta_{pro}$. Ensuite, l'étape (Choix de règle) permet de sélectionner une règle de $\Delta_{cho}$ applicable et de la déclencher. De nouvelles étapes de (Propagation) et de (Choix de règle) s'effectueront par la suite jusqu'à ce que plus aucune règle ne puisse être déclenchée. De manière pratique, cette stratégie correspond à la construction d'un arbre de recherche où les nœuds se font sur l'application ou non d'une règle de $\Delta_{cho}$.

Chaque ASPeRiX computation d'un programme logique permet de déterminer un answer set de ce dernier. En effet, lorsque le critère de convergence est respecté, l'ensemble $IN$ forme alors un answer set comme cela est exprimé à travers le théorème suivant.

**Théorème 2.** (D'après [31]) Soit $P$ un programme logique. Soit $X$ un ensemble de littéraux. $X$ est un answer set de $P$ si et seulement s'il existe une ASPeRiX computation $\langle R_i, I_i \rangle_{i=0}^{\infty}$, $I_i = \langle IN_i, OUT_i \rangle$, pour $P$ tel que $IN_\infty = X$.

On peut améliorer le mécanisme de propagation en intégrant des littéraux *must-be-true*[2]. Ces derniers correspondent à des littéraux qui doivent être dans l'ensemble $IN$ pour éviter une contradiction mais qui n'ont pas encore prouvé leur appartenance à cet ensemble.

**Exemple 16.** Soit $(\bot \leftarrow not\ b.)$ une contrainte dont le corps contient seulement un littéral $not\ b$ avec $b \notin IN \cup OUT$. Le littéral $b$ doit appartenir à l'ensemble $IN$ pour empêcher la contrainte d'être applicable. Si $b$ n'a pas encore été prouvé, on peut conclure que $b$ doit être vrai pour obtenir un answer set, donc $b$ est un littéral must-be-true.

Les littéraux must-be-true sont utilisés durant la phase de propagation pour réduire l'espace de recherche.

**Exemple 17.** Soit $(c \leftarrow a, b.)$ une règle avec $a \in IN$ et $b \notin IN$ mais où $b$ a été précédemment reconnu comme un littéral must-be-true. La règle peut être déclenchée durant la phase de propagation et la tête de règle $c$ devient alors un littéral must-be-true (car $b$ n'a pas encore été prouvé comme appartenant à l'ensemble $IN$).

Les littéraux must-be-true peuvent également permettre de réduire la taille de $\Delta_{cho}$.

**Exemple 18.** Soit $(c \leftarrow a, not\ b.)$ une règle avec $a \in IN$ et $b \notin IN$ mais où $b$ a été précédemment reconnu comme un littéral must-be-true. La règle peut d'ores et déjà être considérée comme un règle bloquée même si $b$ n'a pas encore prouvé son appartenance à l'ensemble $IN$. Ainsi, la règle peut être exclue de $\Delta_{cho}$.

Les littéraux must-be-true sont utilisés pour améliorer la recherche d'answer set lors des propagations et des points de choix mais doivent absolument prouver leur appartenance à l'ensemble $IN$ durant la computation pour obtenir un answer set.

Les notions d'interprétation partielle et d'ASPeRiX computation peuvent être modifiées pour intégrer les littéraux must-be-true.

**Définition 14** (Interprétation partielle mbt)**.** Soit $P$ un programme logique. Une *interprétation partielle mbt* pour $P$ est un triplet $\langle IN, MBT, OUT \rangle$ d'ensembles disjoints de littéraux inclus dans la base de Herbrand de $P$.

**Définition 15** (Type de règle)**.** Soit $r$ une règle, $\sigma$ une fonction de substitution et $I = \langle IN, MBT, OUT \rangle$ une interprétation partielle mbt. On dit que :

- $\sigma(r)$ est *supportée* si et seulement si $corps^+(\sigma(r)) \subseteq IN$ ;
- $\sigma(r)$ est *faiblement supportée* si et seulement si $corps^+(\sigma(r)) \subseteq (IN \cup MBT)$ et $corps^+(\sigma(r)) \nsubseteq IN$ ;
- $\sigma(r)$ est *bloquée* si et seulement si $corps^-(\sigma(r)) \cap (IN \cup MBT) \neq \emptyset$ ;

[2] Ce terme est apparu pour la première fois dans [18].

- $\sigma(r)$ est *débloquée* si et seulement si $corps^{-}(\sigma(r)) \subseteq OUT$ ;
- $\sigma(r)$ est (faiblement) *déclenchable* si et seulement si $\sigma(r)$ est (faiblement) supportée et débloquée ;
- $\sigma(r)$ est *applicable* si et seulement si $\sigma(r)$ est supportée et non bloquée.

La phase de propagation peut alors effectuer des *Mbt-propagations* qui correspondent à des déclenchements de règles qui sont débloquées et seulement faiblement supportées (au moins un littéral du corps positif appartient à l'ensemble $MBT$) vis à vis de l'interprétation partielle mbt $\langle IN, MBT, OUT\rangle$. La tête de règle peut alors être ajoutée à l'ensemble $MBT$ (voir exemple 17). De plus, l'ensemble de règles qui peuvent être choisies lors d'un point de choix $\Delta_{cho}$ est réduit à l'ensemble des règles supportées et non bloquées de l'interprétation partielle mbt (voir exemple 18).

**Définition 16** ($\Delta_{pro}$,$\Delta_{pro_mbt}$ et $\Delta_{cho_mbt}$)**.** Soit $P$ un programme logique, $I = \langle IN, MBT, OUT\rangle$ une interprétation partielle mbt et $R$ un ensemble de règles instanciées.

- $\Delta_{pro}(P, I, R) = \{\sigma(r) \mid r \in P, \sigma$ est une fonction de substitution telle que $\sigma(r)$ est déclenchable par rapport à $I$ et $\sigma(r) \notin R\}$.
- $\Delta_{pro_mbt}(P, I, R) = \{\sigma(r) \mid r \in P, \sigma$ est une fonction de substitution telle que $\sigma(r)$ est faiblement déclenchable par rapport à $I$ et $\sigma(r) \notin R\}$.
- $\Delta_{cho_mbt}(P, I, R) = \{\sigma(r) \mid r \in P, \sigma$ est une fonction de substitution telle que $\sigma(r)$ est applicable par rapport à $I$ et $\sigma(r) \notin R\}$.

Une mbt computation est une `ASPeRiX` computation qui contient en plus des Mbt-propagations ainsi que la possibilité de bloquer une règle de $\Delta_{cho_mbt}$ au lieu de l'appliquer (Exclusion de règle). Pour bloquer une règle, on ajoute une contrainte ($\bot \leftarrow not\ b_1, \ldots, not\ b_m.$) avec les littéraux du corps négatif de la règle. Si le corps négatif ne contient qu'un seul littéral $b$, on ajoute ce littéral à l'ensemble $MBT$ au lieu de créer une contrainte (voir l'exemple 16). Ces possibilités réduisent les choix de règles dans $\Delta_{cho_mbt}$ et ainsi interdisent certaines computations : si une règle $r$ est bloquée, la computation peut converger uniquement vers un answer set dont les règles génératrices ne contiennent pas la règle $r$. De plus, le principe de convergence vérifie qu'à la fin de la mbt computation, aucune contrainte n'est applicable et que chaque littéral de l'ensemble $MBT$ a bien été prouvé et basculé dans l'ensemble $IN$.

Dans la définition d'une mbt computation, $K_i$ est l'ensemble des contraintes ajoutées au programme lors des (Exclusion de règle). De plus, $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl}\rangle$ est composée de trois ensembles de règles instanciées. Tout d'abord, $R_i^{app}$ qui contient l'ensemble des règles déclenchées par $\Delta_{pro}$ lors d'une (Propagation) ou appliquées par $\Delta_{cho_mbt}$ lors d'un (Choix de règle). Ensuite, l'ensemble $R_i^{mbt}$ qui contient l'ensemble des règles déclenchées par $\Delta_{pro_mbt}$ lors de (Mbt-propagation). Enfin, l'ensemble de règles $R_i^{excl}$ qui contient l'ensemble des règles bloquées lors d'un point de choix par (Exclusion de règle).

**Définition 17** (mbt computation)**.** Soit $P$ un programme logique. Une *mbt computation* pour $P$ est une suite $\langle K_i, R_i, I_i\rangle_{i=0}^{\infty}$ avec $K_i$ un ensemble de règles instanciées, $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl}\rangle$ un triplet d'ensembles de règles instanciées et $I_i = \langle IN_i, MBT_i, OUT_i\rangle$ une interprétation partielle mbt qui satisfont les conditions suivantes :

- $K_0 = \emptyset$, $R_0 = \emptyset$ et $I_0 = \langle \emptyset, \emptyset, \{\bot\}\rangle$,

- (Révision) $\forall i \geq 1$,

  (Propagation) $K_i = K_{i-1}$,
  $R_i = \langle R^{app}_{i-1} \cup \{r_i\}, R^{mbt}_{i-1}, R^{excl}_{i-1}\rangle$ avec $r_i \in \Delta_{pro}(P, I_{i-1}, R^{app}_{i-1})$
  et $I_i = \langle IN_{i-1} \cup \{tête(r_i)\}, MBT_{i-1} \setminus \{tête(r_i)\}, OUT_{i-1}\rangle$

  ou (Mbt-propagation) $K_i = K_{i-1}$, $R_i = \langle R^{app}_{i-1}, R^{mbt}_{i-1} \cup \{r_i\}, R^{excl}_{i-1}\rangle$
  avec $r_i \in \Delta_{pro_mbt}(P, I_{i-1}, R^{app}_{i-1} \cup R^{mbt}_{i-1})$
  et $\begin{cases} I_i = \langle IN_{i-1}, MBT_{i-1} \cup \{tête(r_i)\}, OUT_{i-1}\rangle & \text{si } tête(r_i) \notin IN_{i-1} \\ I_i = I_{i-1} & \text{sinon} \end{cases}$

  ou (Choix de règle) $\Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R^{app}_{i-1}) = \emptyset$,
  $\Delta_{pro_mbt}(P \cup K_{i-1}, I_{i-1}, R^{app}_{i-1} \cup R^{mbt}_{i-1}) = \emptyset$,
  $K_i = K_{i-1}$,
  $R_i = \langle R^{app}_{i-1} \cup \{r_i\}, R^{mbt}_{i-1}, R^{excl}_{i-1}\rangle$ avec $r_i \in \Delta_{cho_mbt}(P, I_{i-1}, R^{app}_{i-1} \cup R^{excl}_{i-1})$
  et $I_i = \langle IN_{i-1} \cup \{tête(r_i)\}, MBT_{i-1} \setminus \{tête(r_i)\}, OUT_{i-1} \cup corps^-(r_i)\rangle$

  ou (Exclusion de règle) $\Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R^{app}_{i-1}) = \emptyset$,
  $\Delta_{pro_mbt}(P \cup K_{i-1}, I_{i-1}, R^{app}_{i-1} \cup R^{mbt}_{i-1}) = \emptyset$,

  $K_i = K_{i-1}$, $R_i = \langle R^{app}_{i-1}, R^{mbt}_{i-1}, R^{excl}_{i-1} \cup \{r_i\}\rangle$
  et $I_i = \langle IN_{i-1}, MBT_{i-1} \cup corps^-(r_i), OUT_{i-1}\rangle$
  avec $r_i \in \Delta_{cho_mbt}(P, I_{i-1}, R^{app}_{i-1} \cup R^{excl}_{i-1})$ et $|corps^-(r_i)| = 1$

  ou $K_i = K_{i-1} \cup \{\bot \leftarrow \cup_{b \in corps^-(r_i)} not\ b.\}$,
  $R_i = \langle R^{app}_{i-1}, R^{mbt}_{i-1}, R^{excl}_{i-1} \cup \{r_i\}\rangle$ et $I_i = I_{i-1}$
  avec $r_i \in \Delta_{cho_mbt}(P, I_{i-1}, R^{app}_{i-1} \cup R^{excl}_{i-1})$ et $|corps^-(r_i)| > 1$

  ou (Stabilité) $K_i = K_{i-1}$, $R_i = R_{i-1}$ et $I_i = I_{i-1}$,

- (Convergence) $\exists i \geq 0,\ \Delta_{cho_mbt}(P \cup K_i, I_i, R_i) = \emptyset$ et $MBT_i = \emptyset$.

**Exemple 19.** Soit $P_{19}$ le programme logique suivant :

$$\left\{ \begin{array}{l} a(1). \\ b(X) \leftarrow a(X), not\ c(X). \\ c(X) \leftarrow a(X), not\ b(X). \\ d(X) \leftarrow c(X). \end{array} \right\}$$

La suite suivante $\langle K_i, R_i, I_i\rangle_{i=0}^{6}$ avec $R_i = \langle R^{app}_i, R^{mbt}_i, R^{excl}_i\rangle$ et $I_i = \langle IN_i, MBT_i, OUT_i\rangle$ est un préfixe de mbt computation pour $P_{19}$ :

$I_0 \quad = \langle \emptyset, \emptyset, \{\bot\}\rangle$
$R_0 \quad = \langle \emptyset, \emptyset, \emptyset\rangle$
$K_0 \quad = \emptyset$

Au début, seule la règle $(a(1).)$ est déclenchable.
(Propagation)
$r_1 \quad = a(1). \in \Delta_{pro}(P_{19}, I_0, R^{app}_0)$
$I_1 \quad = \langle \{\mathbf{a(1)}\}, \emptyset, \{\bot\}\rangle$
$R_1 \quad = \langle \{\mathbf{r_1}\}, \emptyset, \emptyset\rangle$
$K_1 \quad = K_0$

Plus aucune règle n'est déclenchable ($\Delta_{pro}(P_{19} \cup K_i, I_1, R_1^{app}) = \emptyset$ et $\Delta_{pro_mbt}(P_{19} \cup K_i, I_1, R_1^{app} \cup R_1^{mbt}) = \emptyset$). Les seules (Révision) disponibles sont (Choix de Règle), (Exclusion de règles) ou (Stabilité).

(Exclusion de règle)

$r_2 = b(1) \leftarrow a(1), not\ c(1). \in \Delta_{cho_mbt}(P_{19}, I_1, R_1^{app} \cup R_1^{excl})$
$I_2 = \langle\{a(1)\}, \{\mathbf{c(1)}\}\}, \{\bot\}\rangle$
$R_2 = \langle\{r_1\}, \emptyset, \{\mathbf{r_2}\}\rangle$
$K_2 = K_1$

Après avoir ajouté $c(1)$ dans l'ensemble mbt, la règle $(d(1) \leftarrow c(1).)$ est déclenchable.

(Mbt-propagation)

$r_3 = d(1) \leftarrow c(1). \in \Delta_{pro_mbt}(P_{19}, I_2, R_2^{app} \cup R_2^{mbt})$
$I_3 = \langle\{a(1)\}, \{c(1), \mathbf{d(1)}\}\}, \{\bot\}\rangle$
$R_3 = \langle\{r_1\}, \{\mathbf{r_3}\}, \{r_2\}\rangle$
$K_3 = K_2$

À nouveau, plus aucune règle n'est déclenchable et les seules (Révision) disponibles sont (Choix de règle), (Exclusion de règle) ou (Stabilité).

(Choix de règle)

$r_4 = c(1) \leftarrow a(1), not\ b(1). \in \Delta_{cho_mbt}(P_{19}, I_3, R_3^{app} \cup R_3^{excl})$
$I_4 = \langle\{a(1), \mathbf{c(1)}\}, \{d(1)\}, \{\bot, \mathbf{b(1)}\}\rangle$
$R_4 = \langle\{r_1, \mathbf{r_4}\}, \{r_3\}, \{r_2\}\rangle$
$K_4 = K_3$

Après avoir ajouté $c(1)$ dans l'ensemble mbt, la règle $(d(1) \leftarrow c(1).)$ est déclenchable.

(Propagation)

$r_5 = d(1) \leftarrow c(1). \in \Delta_{pro}(P_{19}, I_4, R_4^{app})$
$I_5 = \langle\{a(1), c(1), \mathbf{d(1)}\}, \emptyset, \{\bot, b(1)\}\rangle$
$R_5 = \langle\{r_1, r_4, \mathbf{r_5}\}, \{r_3\}, \{r_2\}\rangle$
$K_5 = K_4$

Plus aucune règle ne peut être déclenchée ou appliquée, la seule action disponible pour le reste de la suite est (Stabilité).

(Stabilité)

$I_6 = I_5$

L'ensemble $\{a(1), c(1), d(1)\}$ est un answer set du programme $P_{19}$.

Les mbt computations permettent de déterminer les answer sets d'un programme logique normal. Lorsque le critère de convergence est respecté pour une mbt computation, l'ensemble $IN$ correspond à un answer set. C'est ce qu'exprime le théorème suivant.

**Théorème 3.** (D'après [31]) Soit $P$ un programme logique et $X$ un ensemble de littéraux. $X$ est un answer set de $P$ si et seulement s'il existe une mbt computation $\langle K_i, R_i, I_i\rangle_{i=0}^{\infty}$, $I_i = \langle IN_i, MBT_i, OUT_i\rangle$, pour $P$ tel que $IN_\infty = X$.

En théorie, l'ajout des littéraux must-be-true et la possibilité de bloquer une règle de $\Delta_{cho_mbt}$ au lieu de l'appliquer ne réduisent pas le nombre d'étapes d'une computation. En pratique, ils permettent toutefois de réduire l'espace de recherche en permettant de déterminer plus rapidement les branches qui vont échouer lorsqu'on parcourt l'arbre de recherche. Cette particularité peut être observée dans la partie suivante qui présente les algorithmes au cœur du solveur ASPeRiX.

# 2 Aspect algorithmique

## 2.1 Déroulement de l'algorithme principal

L'algorithme principal d'`ASPeRiX` est basé sur la construction de deux ensembles disjoints de littéraux, $IN$ et $OUT$, au fur et à mesure de la recherche d'answer set 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) de littéraux qui nécessitent d'appartenir à l'answer set pour éviter une contradiction. Cet ensemble contient des éléments qui devront appartenir à l'ensemble $IN$ mais n'ont pas encore prouvé leur appartenance. 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 triées selon les composantes fortement connexes ($CFC$) du graphe des dépendances de $P$. On note $pred(a)$ le symbole de prédicat d'un littéral $a$. Cette notation est étendue à des ensembles de littéraux. Les noeuds du graphe des dépendances d'un programme $P$ sont ses symboles de 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 note $composante(p)$ la composante à laquelle appartient le symbole de prédicat $p$. On dit qu'une règle $r$ appartient à une CFC $C$ si son prédicat de tête est inclus dans la composante $C$. Les contraintes ne sont pas réellement concernées par l'ordonnancement des règles mais sont considérées appartenir à une composante de plus grand numéro que la dernière $CFC$ du graphe des dépendances de $P$. Cela signifie que si $C_n$ est la dernière composante alors les contraintes appartiennent à $C_{n+1}$.

**Exemple 20.** Reprenons le programme $P_{15}$ 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 $P_{15}$ sont $C_1 = \{n\}$, $C_2 = \{a, b\}$ et $C_3 = \{c\}$ (voir figure 4.1).

L'algorithme de recherche d'answer set parcourt 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 propagation ni point de choix sur les règles de la CFC courante, on dit que les prédicats de la CFC sont *résolus*. Cela signifie que l'on ne peut plus rien déduire à propos 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 littéral $a$ tel que $pred(a) = p, a \notin IN \Rightarrow a \in OUT$. Elles ne sont pas ajoutées explicitement à $OUT$ car l'ensemble des instanciations d'un prédicat n'est pas calculé et est donc inconnu.

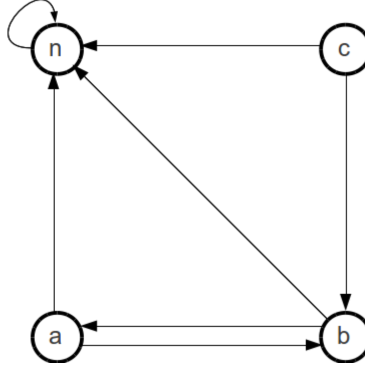



FIGURE 4.1 – Graphe des dépendances de $P_{15}$

Comme évoqué dans la section précédente, 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 d'`ASPeRiX` par les fonctions $\gamma_{pro}$ et $\gamma_{cho}$ (qui sont des fonctions de sélection dans les ensembles $\Delta_{pro} \cup \Delta_{pro_mbt}$ et $\Delta_{cho_mbt}$ de la mbt `ASPeRiX` computation). La fonction $\gamma_{pro}$ recherche une instance de règle (faiblement) 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 $IN$, $MBT$ et $OUT$. La fonction $\gamma_{cho}$ 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 $IN \cup MBT$.

Par ailleurs, les règles instanciées par les fonctions $\gamma_{pro}$ et $\gamma_{cho}$ sont stockées dans un ensemble $Subst = \{\ subst(r_1)\ , ...,\ subst(r_n)\}$ au cours de la recherche d'answer set. Cet ensemble contient toutes les instanciations qui ont été sélectionnées lors des phases de propagation et de point de choix pour chaque règle $r_i$ du programme $P$. Pour chaque règle au premier ordre $r_i$, on note $subst(r_i, \gamma_{pro})$ (respectivement $subst(r_i, \gamma_{cho})$) l'ensemble de toutes les substitutions $\theta$ telles que $\theta(r_i)$ a été déclenchée (respectivement choisie). L'ensemble $subst(r_i) = subst(r_i, \gamma_{pro}) \cup subst(r_i, \gamma_{cho})$ représente l'ensemble de toutes les substitutions $\theta$ telles que $\theta(r_i)$ a été déclenchée ou choisie. De plus, $subst_rule(r_i, \gamma_{pro}) = \bigcup_{\theta \in subst(r_i, \gamma_{pro})} \{\theta(r_i)\}$ (respectivement $subst_rule(r_i, \gamma_{cho}) = \bigcup_{\theta \in subst(r_i, \gamma_{cho})} \{\theta(r_i)\}$) représente l'ensemble des règles instanciées obtenues à partir des substitutions $subst(r_i, \gamma_{pro})$ (respectivement $subst(r_i, \gamma_{cho})$) et $subst_rule(r_i) = \bigcup_{\theta \in subst(r_i)} \{\theta(r_i)\}$ représente l'ensemble des règles instanciées obtenues à partir des substitutions $subst(r_i)$.

Les fonctions $\gamma_{pro}$ et $\gamma_{cho}$ sont définies de la façon suivante :

- $\gamma_{pro}(P, IN, MBT, OUT, CFC, Subst)$ : fonction non déterministe qui sélectionne une règle (ou une contrainte) instanciée $\theta(r)$ appartenant à une CFC supérieure ou égale à la CFC courante dans le graphe des dépendances de $P$ telle que $corps^+(\theta(r)) \subseteq IN \cup MBT$, $corps^-(\theta(r)) \subseteq OUT$, $r \in P$ et $\theta(r) \notin subst_rule(r, \gamma_{pro})$ ou retourne $NULL$ si aucune règle ne satisfait ces conditions.

- $\gamma_{cho}(P, IN, MBT, OUT, CFC, Subst)$ : fonction non déterministe qui sélectionne une règle instanciée $\theta(r)$ appartenant à la CFC courante dans le graphe des dépendances de $P$ telle que $corps^+(\theta(r)) \subseteq IN$, $corps^-(\theta(r)) \cap (IN \cup MBT) = \emptyset$, $r \in P$ et $\theta(r) \notin subst_rule(r)$ ou retourne $NULL$ si aucune règle ne satisfait ces conditions.

La fonction $solve(P_R,\ P_K,\ IN,\ MBT,\ OUT,\ CFC,\ Subst)$ (voir algorithme 3) décrit le déroulement de l'algorithme de recherche d'un answer set pour un programme $P$. $P_K$ représente l'ensemble des contraintes de $P$ et $P_R$ les autres règles. Par défaut, $\perp$ est inclus dans l'ensemble $OUT$. Ainsi, une contradiction est immédiatement détectée si une contrainte est déclenchée car $\perp$ serait alors ajouté à l'ensemble $IN$ ce qui rendrait les ensembles $IN$ et $OUT$ non disjoints. À noter que l'algorithme de la fonction $solve$ présenté ici décrit le calcul d'un answer set (ou aucun si le programme est inconsistant) grâce au booléen $stop$ qui arrête la recherche dès lors qu'un answer set a été trouvé. Cet algorithme peut être étendu aisément pour le calcul d'un nombre arbitraire d'answer set. Ici, la fonction $solve$ retournera soit un ensemble représentant un answer set (s'il existe) soit une constante $no_answer_set$ si le programme n'admet aucun answer set.

Initialement, $IN = \emptyset$, $MBT = \emptyset$, $OUT = \{\perp\}$, $Subst = \emptyset$ 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 $\gamma_{pro}(P_R \cup P_K, IN,\ MBT,\ OUT,\ CFC,\ Subst)$ qui sélectionne et instancie une unique règle du programme pouvant être déclenchée (ligne 4). Si une telle règle $r_0$ existe, son littéral de tête doit appartenir à l'answer set que l'on recherche. Il est alors ajouté à l'ensemble $IN$ (ligne 9) si le corps positif de la règle est inclus dans l'ensemble $IN$ ou bien ajouté à l'ensemble $MBT$ (ligne 7) lorsqu'au moins un littéral $a$ 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 littéral de tête qui est ajouté à $IN$ doit être retiré de l'ensemble $MBT$ s'il y apparaît puisque une preuve de son appartenance à l'answer set a été trouvée (ligne 10-11). 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 (ligne 13) et on passe à la phase de choix si aucune contradiction n'est détectée.

L'étape de choix force ou interdit le déclenchement d'une règle applicable. Elle applique une instance de règle supportée et non bloquée $r_0$ grâce à la fonction $\gamma_{cho}(P_R,\ IN,\ MBT,\ OUT, CFC,\ Subst)$ qui sélectionne et instancie une unique règle applicable de $P_R$ dont la tête de la règle appartient à la CFC courante (ligne 16). 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 (ligne 17). 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 $r_0$ (lines 19-25). 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 littéral alors celui-ci est ajouté à l'ensemble $MBT$ (ligne 22). Sinon, on ajoute une contrainte qui empêche l'ensemble des littéraux du corps négatif d'appartenir à $OUT$[3] (ligne 24).

Lorsque plus aucun point de choix n'est possible, la CFC courante peut être résolue (ligne 27). Avant de passer à la résolution des CFC suivantes (ligne 30), on s'assure qu'aucun élément de $MBT$ n'est une instance d'un prédicat de la CFC courante (ligne 28). 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 ajouté 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 (ligne 36). Lorsque la dernière CFC est résolue, l'ensemble $IN$ représente un answer set de $P$ si aucune contrainte ne peut être appliquée lors de l'appel à la fonction $\gamma_{check}$

[3]Ici on considère seulement les littéraux 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 $OUT$

(ligne 31) définie de la manière suivante :

- $\gamma_{check}(P, IN, MBT, OUT, CFC)$ : fonction qui vérifie s'il existe une contrainte $c \in ground(P)$ tel que $corps^+(c) \subseteq IN$, $\{a \in corps^-(c) \mid composante(pred(a)) \leq CFC\} \cap IN = \emptyset$ et $\{a \in corps^-(c) \mid composante(pred(a)) > CFC\} \subseteq OUT$.

Notons que dans l'usage qui en est fait ici, $CFC$ est la dernière composante et que l'on recherche donc simplement une contrainte applicable. La fonction sera réutilisée dans le chapitre 6 sur le backjumping et l'usage qui en sera alors fait sera plus général. Il nécessitera de distinguer les éléments du corps négatif des composantes déjà résolues des éléments des autres composantes. On vérifie que les premiers n'appartiennent pas à $IN$ et donc appartiennent implicitement à $OUT$ alors que les seconds doivent appartenir explicitement à $OUT$. On vérifie donc que, étant donné les ajouts implicites à l'ensemble $OUT$, la contrainte est déclenchable.

**Exemple 21.** Le déroulement de l'algorithme d'`ASPeRiX` pour le programme $P_{15}$ (voir exemple 20) est représenté par l'arbre de la figure 4.2. 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 (Point de choix $CP1$). On force alors la règle à être déclenchée en ajoutant $b(1)$ et $b(2)$ à l'ensemble $OUT$ (branche gauche de $CP1$). 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 (point de choix $CP2$) et force le déclenchement de $(a(2) \leftarrow n(2), not\ b(2), not\ b(3).)$ en ajoutant le littéral $b(3)$ à l'ensemble $OUT$ (branche gauche de $CP2$). 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 answer set $\{a(1), a(2), c(1), c(2), n(1), n(2)\}$.

Si l'on souhaite trouver de nouveaux answers sets, on revient au dernier point de choix $(a(2) \leftarrow n(2), not\ b(2), not\ b(3).)$ de la composante $C_2$ (point de choix $CP2$) et on bloque la règle en ajoutant une contrainte $(\bot \leftarrow not\ b(2), not\ b(3).)$ dans $P_K$ (branche droite de $CP2$). Un nouveau point de choix est effectué (point de choix $CP3$) et force le déclenchement de $(b(2) \leftarrow n(2), not\ a(2).)$ en ajoutant le littéral $a(2)$ à l'ensemble $OUT$ (branche gauche de $CP3$). 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 $C_2$ (point de choix $CP3$) et on bloque la règle en ajoutant $a(2)$ à l'ensemble $MBT$ (branche droite de $CP3$). Plus aucun point de choix ne peut être effectué mais l'ensemble $MBT$ contient un littéral 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 $C_2$ (point de choix $CP1$), bloque la règle et recherche à nouveau d'éventuels answer sets (branche droite de $CP1$). Ce parcours continue jusqu'à ce que l'arbre de recherche soit complètement parcouru dans le cas où l'on recherche tous les answer sets du programme.

**Algorithme 3** : *solve*

```
1  Fonction solve(P_R, P_K, IN, MBT, OUT, CFC, Subst);
2  // Recherche d'un answer set pour le programme P = P_R ∪ P_K
3  repeat //Phase de propagation
4      r_0 ← γ_pro(P_R ∪ P_K, IN, MBT, OUT, CFC, Subst);
5      if r_0 ≠ NULL then
6          if (corps⁺(r_0) ∩ MBT) ≠ ∅ then
7              MBT ← MBT ∪ {tête(r_0)};
8          else
9              IN ← IN ∪ {tête(r_0)};
10             if (tête(r_0) ∈ MBT) then
11                 MBT ← MBT \ {tête(r_0)};
12 until r_0 = NULL;
13 if ((IN ∪ MBT) ∩ OUT ≠ ∅) then //Contradiction détectée
14     return no_answer_set;
15 else
16     r_0 ← γ_cho(P_R, IN, MBT, OUT, CFC, Subst);
17     if r_0 ≠ NULL then //Point de Choix
18         stop ← solve(P_R, P_K, IN, MBT, OUT ∪ corps⁻(r_0), CFC, Subst);
19         if stop = no_answer_set then
20             literals ← {a | a ∈ corps⁻(r_0), pred(a) ∈ pred(CFC)};
21             if (|literals| = 1) then
22                 MBT ← MBT ∪ literals;
23             else
24                 P_K ← P_K ∪ {⊥ ← ∪_{a_i ∈ literals} not a_i};
25             stop ← solve(P_R, P_K, IN, MBT, OUT, CFC, Subst);
26         return stop ;
27     else// La CFC est résolue
28         if pred(MBT) ∩ pred(CFC) = ∅ then
29             if ¬last(CFC) then
30                 return solve(P_R, P_K, IN, MBT, OUT, CFC + 1, Subst);
31             else
32                 if γ_check(P_K, IN, MBT, OUT, CFC) then // une contrainte est violée
33                     return no_answer_set;
34                 else// Un answer set est trouvé
35                     return IN;
36         else// Un littéral MBT ne peut plus être prouvé
37             return no_answer_set;
```

## 2.2 Les fonctions $\gamma$

Les fonctions $\gamma$ ont un rôle majeur dans les deux étapes importantes de la recherche d'answer set. La fonction $\gamma_{pro}$ 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

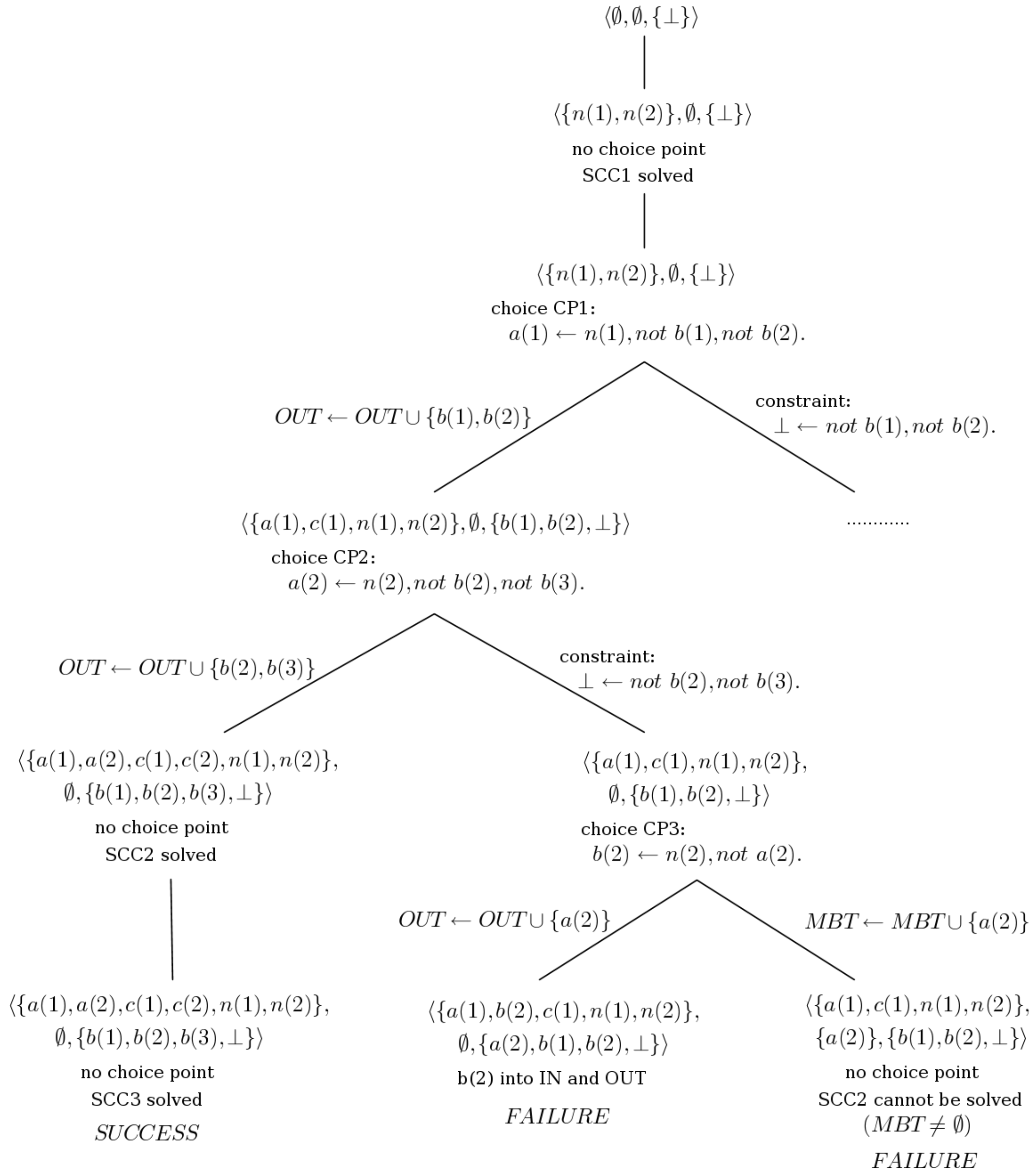



FIGURE 4.2 – Arbre de recherche d'answer set pour $P_{15}$.

$MBT$). La fonction $\gamma_{cho}$ 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 $\gamma_{check}$ 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 d'answer set, les fonctions $\gamma$ requièrent des appels à une fonction permettant de réaliser des instanciations des règles avec variables susceptibles d'être déclenchées ou appliquées. La fonction $instantiateRule$ (voir section 2.6) réalise cela et recherche

une substitution pour les littéraux d'une règle ASP compatible avec le critère de sélection de règle de la fonction $\gamma$ qui réalise son appel (règle déclenchable pour $\gamma_{pro}$ ou applicable pour $\gamma_{cho}$ et $\gamma_{check}$).

## 2.3 $\gamma_{pro}$

La fonction $\gamma_{pro}$ recherche une règle à déclencher à partir des éléments déjà présents dans les ensembles $IN$, $MBT$ et $OUT$. Pour cela, elle recherche une instanciation complète telle que le corps positif soit inclus dans $IN \cup MBT$ et le corps négatif soit inclus dans $OUT$ parmi un ensemble $R$ de règles contenant les prédicats des *littéraux à propager* c'est-à-dire les littéraux récemment ajoutés aux ensembles $IN$ et $OUT$ qui n'ont pas encore été utilisés pour la phase de propagation. Ainsi, lorsque un littéral $a$ est ajouté à l'ensemble $IN$ ou $MBT$ (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 $\gamma_{pro}$ afin de propager ce littéral et d'obtenir de nouvelles règles déclenchables (s'il en existe). Lors du premier appel à la fonction $solve$, l'ensemble $R$ contient toutes les règles qui possèdent un prédicat d'un littéral 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 littéraux 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 4 de la fonction $\gamma_{pro}$ choisit une règle $r$ parmi l'ensemble $R$ (la première règle de cet ensemble, ligne 5) 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 $r$ si elle existe (ligne 6). Dans le cas où la règle $r$ ne contient plus aucune instanciation déclenchable (ligne 7), $\gamma_{pro}$ 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 $R$. Lorsqu'une règle admet une substitution (ligne 10), cette dernière est enregistrée dans $Subst$ afin d'en trouver d'autres au prochain appel à $\gamma_{pro}$.

**Exemple 22.** Reprenons le déroulement de l'algorithme de recherche d'answer set pour le programme $P_{15}$ (voir exemple 20) représenté par l'arbre de la figure 4.2. Après le premier point de choix $(a(1) \leftarrow n(1), not\ b(1), not\ b(2).)$ (point de choix $CP1$), les littéraux $b(1)$ et $b(2)$ sont ajoutés à l'ensemble $OUT$ pour forcer le déclenchement de la règle (branche gauche du point de choix $CP1$). Lors de la phase de propagation qui suit ce point de choix, plusieurs appels à la fonction $\gamma_{pro}$ sont exécutés. Lors du premier appel, les littéraux 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 arbitrairement la première règle de l'ensemble $R$ et on trouve une instanciation déclenchable $(a(1) \leftarrow n(1), not\ b(1), not\ b(2).)$. $\gamma_{pro}$ renvoie alors l'instanciation de la règle et l'algorithme de recherche d'answer set (fonction $solve$) se charge d'ajouter $a(1)$ dans $IN$. Lors du prochain appel à $\gamma_{pro}$, l'ensemble $R$ doit contenir, en plus de ses règles précédentes, toutes les règles contenant le prédicat $a$ du littéral 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. $\gamma_{pro}$ 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 $R$. $\gamma_{pro}$ recherche ensuite

**Algorithme 4** : $\gamma_{pro}$

```
1  Fonction γ_pro(P, IN, MBT, OUT, CFC, Subst);
2  R ← Ensemble de règles (contraintes inclus) contenant les symboles de prédicats à
   propager;
3  if R ≠ ∅ then
4      repeat
5          r ← first(R);
           /* Recherche d'une instanciation de la règle r avec
              corps+(r) ⊆ IN ∪ MBT et corps−(r) ⊆ OUT                    */
6          θ ← instantiateRule(r, γ_pro, IN, MBT, OUT, subst(r));
7          if θ = NULL then
8              R ← R\{r};
9      until θ ≠ NULL or R = ∅ ;
10     if θ ≠ NULL then
           /* Une règle instanciée supportée et débloquée est
              trouvée                                                    */
11         subst(r, γ_pro) ← subst(r, γ_pro) ∪ {θ};
12         return θ(r);
13     else
14         return NULL;
15 else
16     return NULL;
```

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 d'answer set qui ajoute $c(1)$ dans $IN$. Lors de l'appel suivant à la fonction $\gamma_{pro}$, les règles contenant dans leur corps positif le prédicat $c$ du littéral 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. $\gamma_{pro}$ renvoie alors la valeur $NULL$ et la phase de propagation de l'algorithme de recherche d'answer set se termine.

## 2.4 $\gamma_{cho}$

La fonction $\gamma_{cho}$ 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 le littéral 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 $IN \cup MBT$.

L'algorithme 5 de la fonction $\gamma_{cho}$ présente des similitudes avec celui de $\gamma_{pro}$. $\gamma_{cho}$ recherche une règle applicable parmi un ensemble $R$ de règles de la CFC courante ayant dans leur corps négatif un ou plusieurs symboles de prédicats de la CFC courante[4] (les prédicats non résolus).

[4]Si tous les symboles de prédicats du corps négatif appartiennent à des CFC précédentes, ils sont déjà résolus et la règle est utilisable seulement pour la propagation.

Pour cela, $\gamma_{cho}$ 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 $\gamma_{pro}$, 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 $R$.

**Algorithme 5** : $\gamma_{cho}$

```
1  Fonction γ_cho(P, IN, MBT, OUT, CFC, Subst);
2  R ← Ensemble de règles de la CFC courante telles que le corps négatif contient au moins
   un symbole de prédicat non résolu;
3  if R ≠ ∅ then
4      repeat
5          r ← first(R);
           /* Recherche d'une instanciation de la règle r avec
              corps+(r) ⊆ IN et corps-(r) ∩ (IN ∪ MBT) = ∅                  */
6          θ ← instantiateRule(r, γ_cho, IN, MBT, OUT, subst(r));
7          if θ = NULL then
8              R ← R\{r};
9      until θ ≠ NULL or R = ∅ ;
10     if θ ≠ NULL then
           /* Une règle instanciée applicable est trouvée                   */
11         subst(r, γ_cho) ← subst(r, γ_cho) ∪ {θ};
12         return θ(r);
13     else
14         return NULL;
15 else
16     return NULL;
```

**Exemple 23.** Reprenons le déroulement de l'algorithme de recherche d'answer set pour le programme $P_{15}$ (voir exemple 20) représenté par l'arbre de la figure 4.2. 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 $\gamma_{cho}$. 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 $R$. $\gamma_{cho}$ recherche ensuite arbitrairement 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 d'answer set $solve$ (point de choix $CP1$). Après une phase de propagation, $\gamma_{cho}$ 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 d'answer set (point de choix $CP2$). Après une nouvelle phase de propagation, $\gamma_{cho}$ 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 retiré de l'ensemble $R$ et $\gamma_{cho}$ 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 $\gamma_{cho}$ renvoie $NULL$ à l'algorithme de recherche d'answer set pour signifier que plus aucun point de choix ne peut être effectué sur la CFC courante.

## 2.5 $\gamma_{check}$

La fonction $\gamma_{check}$ 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 symboles de prédicats de la dernière $CFC$ ne peut être appliquée afin de déterminer si l'ensemble $IN$ obtenu est un answer set.

L'algorithme 6 de $\gamma_{check}$ est similaire à celui de $\gamma_{cho}$. Il recherche une contrainte instanciée applicable parmi un ensemble $C$ de contraintes dont le corps négatif contient un ou plusieurs des symboles de prédicats de la dernière $CFC$. $\gamma_{check}$ 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 answer set du programme. Si une contrainte est applicable, l'algorithme renvoie vrai ce qui signifie qu'un échec est constaté sur la branche.

**Algorithme 6 :** $\gamma_{check}$

```
1  Fonction γcheck(P, IN, MBT, OUT, CFC);
2  C ← Ensemble de contraintes telles que le corps négatif contient au moins un symbole
   de prédicat non résolu;
3  if C ≠ ∅ then
4      repeat
5          c ← first(C);
           /* Recherche d'une instanciation de la contrainte c
              tel que corps+(c) ⊆ IN
              {a ∈ corps−(c) | composante(pred(a)) ≤ CFC} ∩ IN = ∅ et
              {a ∈ corps−(c) | composante(pred(a)) > CFC} ⊆ OUT          */
6          θ ← instantiateRule(c, γcheck, IN, MBT, OUT, ∅);
7          if θ = NULL then
8              C ← C\{c};
9      until θ ≠ NULL or C = ∅ ;
10     if θ ≠ NULL then
           /* Une contrainte instanciée applicable est trouvée */
11         return true;
12     else
13         return false;
14 else
15     return false;
```

## 2.6 Instanciation d'une règle

L'instanciation d'une règle est une étape qui intervient lors de chaque utilisation de $\gamma_{pro}$, $\gamma_{cho}$ et $\gamma_{check}$. Elle s'inspire de travaux réalisés sur le solveur DLV [16, 44] basés sur une technique d'optimisation semi-naïve [52]. Elle a pour objectif de trouver une substitution pour l'ensemble des littéraux du corps d'une règle $r$ compte tenu des éléments présents dans $IN$, $MBT$ et dans $OUT$. Pour cela, une substitution partielle $\theta$ est construite au fur et à mesure que l'on trouve une substitution pour les littéraux du corps de la règle $r$. Ainsi, les littéraux $a_1, a_2, \ldots, a_n$ du corps de

la règle $r$ sont ordonnés par une liste $[a_1, a_2, \ldots, a_n]$ où $premierLitteral(r)$ (respectivement $dernierLitteral(r)$) correspond à $a_1$ (respectivement $a_n$) et $precedentLitteral(r)$ (respectivement $prochainLitteral(r)$) correspond au littéral qui précède (respectivement qui suit) celui qu'on est en train d'étudier dans la liste. La recherche d'une substitution pour un littéral $a$ d'une règle $r$ se fait à l'aide des fonctions $firstMatch$ et $nextMatch$ qui recherchent une substitution du littéral qui ne mène pas à une substitution déjà présente dans l'ensemble $subst(r)$ contenant toutes les substitutions $\theta$ de la règle $r$ déclenchées au cours de la recherche d'answer set.

Les fonctions $firstMatch$ et $nextMatch$ sont définies de la manière suivante :

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

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

La substitution d'un littéral respecte les propriétés de la fonction $\gamma$ utilisée. Ainsi, lorsqu'un littéral appartient au corps positif, $firstMatch$ et $nextMatch$ recherchent une substitution telle que le littéral substitué appartient à l'ensemble $IN$ (ou $IN \cup MBT$). Lorsqu'il s'agit d'un littéral du corps négatif, il y a deux cas de figure selon le critère de sélection de la fonction $\gamma$ (règle déclenchable pour $\gamma_{pro}$ ou règle applicable pour $\gamma_{cho}$ et $\gamma_{check}$). $firstMatch$ et $nextMatch$ recherchent une substitution telle que le littéral substitué appartient à $OUT$ dans le cas de $\gamma_{pro}$ ou bien une substitution telle que le littéral substitué n'appartient pas à $IN$ pour $\gamma_{cho}$ et $\gamma_{check}$.

Pour une règle $r$, on appelle variable libre d'un littéral $a$, une variable $X$ qui apparaît pour la première fois dans le corps de $r$ lorsque l'on parcourt $a$. Autrement dit, aucun littéral qui précède $a$ dans le corps de $r$ ne possède la variable $X$.

**Exemple 24.** 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 :

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

Lors de l'instanciation d'une règle, on recherche la prochaine substitution possible pour toutes les variables libres de chaque littéral que l'on parcourt et on conserve les substitutions des variables précédemment calculées. Si un littéral ne possède aucune variable libre, on vérifie simplement que sa substitution est valide par rapport au critère de sélection de la fonction $\gamma$ utilisée c'est à dire que le littéral substitué respecte les propriétés de la fonction $\gamma$ utilisée. Ainsi, lorsque toutes les variables du corps positif de la règle ont été substituées, on se contente

**Algorithme 7** : $instantiateRule$

```
1  Function instantiateRule(r, γ, IN, MBT, OUT, subst);
2  θ ← dernierеSubstitution(r, γ);
3  if θ = ∅ then
       /* Recherche une première substitution possible pour le
          premier littéral                                      */
4      a ← premierLitteral(r);
5      matchFound ← firstMatch(a, θ, γ, INMBT, OUT, subst);
6  else
       /* Recherche la prochaine substitution possible pour le
          dernier littéral                                      */
7      a ← dernierLitteral(r);
8      θ ← θ\substitutionVariablesLibres(a);
9      matchFound ← nextMatch(a, θ, γ, IN, MBT, OUT, subst);
10 while true do
11     if matchFound then
12         if a ≠ dernierLitteral(r) then
13             a ← prochainLitteral(r);
14             matchfound ← firstMatch(a, θ, γ, IN, MBT, OUT, subst);
15         else
               /* Une substitution complète est trouvée        */
16             return θ;
17     else
           /* Aucune substitution pour le littéral a, on revient
              sur le littéral précédent (s'il existe) pour
              trouver sa prochaine substitution possible        */
18         if a ≠ premierLitteral(r) then
19             a ← precedentLitteral(r);
20             θ ← θ\substitutionVariablesLibres(a);
21             matchFound ← nextMatch(a, θ, γ, IN, MBT, OUT, subst);
22         else
23             return NULL;
```

alors de vérifier que les substitutions des littéraux restants dans le corps de la règle (notamment ceux du corps négatif[5]) sont valides par rapport au critère de sélection de la fonction $\gamma$.

La fonction $instantiateRule$ de l'algorithme 7 présente le principe d'instanciation d'une règle pour des ensembles $IN$, $MBT$ et $OUT$ constants. Elle débute avec pour valeur de la substitution partielle $\theta$ la dernière substitution réalisée sur la règle $r$ lors d'un précédent appel à la fonction $\gamma$ correspondante si elle existe grâce à la fonction $derniereSubstitution(r, \gamma)$ (ligne 2). S'il s'agit de la première tentative d'instanciation de la règle $r$ alors $\theta$ est vide (ligne 3) et $instantiateRule$ recherche une première substitution pour le premier littéral du corps de la règle à l'aide de la fonction $firstMatch$. Dans le cas contraire, une substitution complète de $r$ a déjà été calculée lors d'un précédent appel (ligne 6) et la fonction recherche alors une nouvelle

[5]Comme les règles d'un programme ASP doivent être sûres, toutes les variables apparaissant dans le corps négatif de la règle apparaissent également dans le corps positif.

substitution possible pour la règle $r$. Pour cela, elle recherche la prochaine instance valide du dernier littéral de la règle $r$ en supprimant de $\theta$ les substitutions apportées par les variables libres de ce littéral (grâce à la fonction $substitutionVariablesLibres$) et en faisant appel à la fonction $nextMatch$.

Lors de l'exécution de la boucle principale, elle vérifie tout d'abord qu'une substitution a bien été trouvée pour le littéral courant $a$ (ligne 11). Si c'est le cas, elle recherche une première substitution pour le littéral suivant du corps de la règle respectant la substitution partielle $\theta$. Lorsque tous les littéraux ont été parcourus, une substitution complète est trouvée (ligne 15). Elle renvoie alors à la fonction $\gamma_{pro}$, $\gamma_{cho}$ ou $\gamma_{check}$ la substitution complète. Lorsque l'instanciation d'un littéral échoue (aucune substitution possible), elle revient sur le littéral précédent (ligne 18) puis met à jour $\theta$ en supprimant les substitutions des variables libres de ce littéral. Ensuite, elle fait appel à la fonction $nextMatch$ recherchant la prochaine instanciation possible pour ce littéral. L'instanciation d'une règle $r$ échoue lorsque plus aucune substitution n'est possible pour le premier littéral (ligne 23).

En réalité, l'algorithme d'instanciation d'une règle est légèrement plus compliqué que cela notamment pour $\gamma_{pro}$ car il faut tenir compte des littéraux ajoutés aux ensembles $IN$, $MBT$ et $OUT$ durant les phases de propagation et de points de choix. Ainsi, ASPeRiX met en place un ensemble de littéraux dits *récursifs* dans le corps des règles à propager. Ceux ci sont les littéraux avec variables du corps de la règle étudiée qui ont le même prédicat que le littéral que l'on souhaite propager. Ces littéraux sont placés en premier dans la règle $r$ étudiée. Dans un premier temps, le premier littéral récursif est *marqué* et ne peut prendre comme valeur que celle du littéral à propager. Les autres littéraux récursifs qui le suivent peuvent prendre n'importe quelle valeur du littéral qui précède celle du littéral à propager dans l'instanciation partielle $\langle IN, OUT\rangle$ y compris ce dernier. Ensuite, lorsque l'instanciation du premier littéral échoue, on marque le littéral récursif suivant comme littéral récursif courant et on recommence l'instanciation de la règle en débutant par le littéral marqué. Les littéraux récursifs qui précèdent le littéral récursif courant peuvent alors prendre comme valeurs toutes les valeurs du prédicat du littéral à propager apparues avant ce dernier dans l'ensemble $IN$ ou $OUT$ tandis que le littéral récursif courant ne peut alors prendre comme valeur que celle du littéral à propager. Lorsque l'instanciation du premier littéral échoue et qu'il n'y a plus d'autres littéraux récursifs, l'instanciation de la règle se termine.

**Exemple 25.** Soit le programme $P_{25}$ suivant :

$$\left\{\begin{array}{l} a(1) \leftarrow b(1,1). \\ a(X+Y) \leftarrow a(X), b(X,Y), a(Y). \\ b(1,1). \\ b(1,2). \end{array}\right\}$$

Au départ, les faits sont propagés et ajoutés à l'ensemble $IN$. Le fait $b(1,1)$ permet de déclencher la règle $a(1) \leftarrow b(1,1)$. À ce moment là, $IN = \{a(1), b(1,1), b(1,2)\}$.

Ensuite, on souhaite propager le littéral $a(1)$ en instanciant la règle $r_2 : a(X+Y) \leftarrow a(X), b(X,Y), a(Y)$. Les littéraux récursifs du corps de la règle sont $a(X)$ et $a(Y)$, ils sont placés en tête de la règle (voir Table 4.1). Le littéral $a(X)$ est alors marqué et peut prendre comme valeur uniquement le littéral à propager $a(1)$ lors de son instanciation. $X$ est donc substitué par la valeur 1 dans $\theta$. Ensuite, le littéral suivant du corps de la règle, $a(Y)$, devient le littéral courant et prend comme valeur la première valeur instanciée du littéral présente dans l'ensemble $IN$ qui est également $a(1)$. $Y$ est substitué par la valeur 1 dans $\theta$. On passe alors au dernier littéral, $b(X,Y)$, qui ne possède aucune variable libre donc on vérifie simplement que le littéral $b(1,1)$ obtenu en substituant $b(X,Y)$ par les valeurs de $X$ et de $Y$ dans la substitution

TABLE 4.1 – Décomposition de l'instanciation de la règle $r_2$ pour le littéral à propager $a(1)$ (avec $IN = \{a(1), b(1,1), b(1,2)\}$)

| $a(X+Y)$ | $\leftarrow$ | $a(X)$, | $a(Y)$, | $b(X,Y)$ | |
|---|---|---|---|---|---|
| | | *** Premier parcours *** | | | |
| | | $\mathbf{a(1)}$ | - | - | ($a(X)$ marqué) |
| | | $\mathbf{a(1)}$ | $a(1)$ | - | |
| | | $\mathbf{a(1)}$ | $a(1)$ | $b(1,1)$ | $\Rightarrow$ Instanciation complète ! |
| | | *** Deuxième parcours *** | | | |
| | | $\mathbf{a(1)}$ | $a(1)$ | NO | |
| | | $\mathbf{a(1)}$ | NO | - | |
| | | **NO** | - | - | $\Rightarrow$ échec |
| | | - | $\mathbf{a(1)}$ | - | ($a(Y)$ marqué) |
| | | NO | $\mathbf{a(1)}$ | - | |
| | | - | **NO** | - | $\Rightarrow$ échec |

$\theta$ appartient bien à $IN$. Il ne reste plus de littéral à parcourir donc une substitution complète est trouvée. Le littéral de tête $a(X+Y)$ prend alors les valeurs de la substitution $\theta$. Ainsi, $a(2)$ est ajouté à $IN$ et devra être propagée.

Ensuite, une nouvelle tentative d'instanciation de la règle $r_2$ pour le littéral à propager $a(1)$ est réalisée. On repart de la dernière substitution de la règle avec $\theta = \{X/1, Y/1\}$ et on recherche une nouvelle substitution pour le littéral $b(X,Y)$. Comme $b(X,Y)$ ne possède aucune variable libre, il ne peut avoir d'autres substitutions que la substitution actuelle. On revient alors sur le littéral $a(Y)$ qui lui aussi ne possède aucune autre substitution ($a(2)$ a été déterminé après $a(1)$ et ne peut pas être utilisé). Le littéral $a(X)$ ne pouvant prendre que la valeur $a(1)$ échoue également. Comme le premier littéral a échoué, on marque alors le littéral récursif $a(Y)$ comme littéral récursif courant et on l'instancie avec la valeur du littéral à propager $a(1)$ ($\theta = \{Y/1\}$). Le littéral $a(X)$ ne pouvant prendre comme valeur que les littéraux apparus avant le littéral à propager $a(1)$, aucune substitution n'est possible. Comme $a(Y)$ ne peut prendre comme valeur que $a(1)$, on échoue à nouveau sur le premier littéral. Étant donné qu'il ne reste plus de littéraux récursifs à marquer, l'instanciation de la règle se termine.

Le littéral $a(2)$ doit à présent être propagé (voir Table 4.2). La seule règle susceptible de permettre une propagation est une nouvelle fois la règle $r_2$ et les littéraux $a(X)$ et $a(Y)$ se retrouvent à nouveau littéraux récursifs. On commence l'instanciation de la règle avec le littéral $a(X)$ qui est le littéral récursif courant. $X$ est substitué par la valeur 2 car seul la valeur du littéral à propager $a(2)$ est autorisée. On passe alors au littéral suivant $a(Y)$ où $Y$ peut être substitué par la valeur 1 car $a(1) \in IN$. Le littéral $b(X,Y)$ ne possède pas de variables libres et comme le littéral $b(2,1)$ respectant la substitution partielle de $\theta = \{X/2, Y/1\}$ n'appartient pas à l'ensemble $IN$, le littéral $b(X,Y)$ n'a aucune substitution possible. On tente alors d'instancier $a(Y)$ avec sa prochaine valeur possible 2 (car $a(2) \in IN$). De nouveau, le littéral $b(2,2)$ respectant la substitution partielle de $\theta = \{X/2, Y/2\}$ n'appartient pas à l'ensemble $IN$ et le littéral $b(X,Y)$ n'a aucune substitution possible. On revient alors à $a(Y)$ qui n'a plus aucune valeur possible. On revient à $a(X)$ qui à son tour n'a plus de valeur possible étant donné que sa seule valeur autorisée était la valeur 2 du littéral à propager $a(2)$.

Le premier littéral ayant échoué, on reprend l'instanciation en marquant le second littéral récursif $a(Y)$ comme littéral récursif courant et on l'instancie avec la valeur du littéral à propager $a(2)$ ($\theta = \{Y/2\}$). Le littéral $a(X)$ maintenant démarqué peut prendre les valeurs qui précèdent le littéral à propager $a(2)$ dans l'ensemble $IN$. $X$ est donc substitué par la valeur

TABLE 4.2 – Décomposition de l'instanciation de la règle $r_2$ pour le littéral à propager $a(2)$ (avec $IN = \{a(1), a(2), b(1,1), b(1,2)\}$)

| $a(X+Y)$ | $\leftarrow$ | $a(X),$ | $a(Y),$ | $b(X,Y)$ | |
|---|---|---|---|---|---|
| | | *** Premier parcours *** | | | |
| | | $\mathbf{a(2)}$ | - | - | ($a(X)$ marqué) |
| | | $\mathbf{a(2)}$ | $a(1)$ | - | |
| | | $\mathbf{a(2)}$ | $a(1)$ | NO | |
| | | $\mathbf{a(2)}$ | $a(2)$ | - | |
| | | $\mathbf{a(2)}$ | $a(2)$ | NO | |
| | | $\mathbf{a(2)}$ | NO | - | |
| | | **NO** | - | - | $\Rightarrow$ échec |
| | | - | $\mathbf{a(2)}$ | - | ($a(Y)$ marqué) |
| | | $a(1)$ | $\mathbf{a(2)}$ | - | |
| | | $a(1)$ | $\mathbf{a(2)}$ | $b(1,2)$ | $\Rightarrow$ Instanciation complète ! |
| | | *** Deuxième parcours *** | | | |
| | | $a(1)$ | $\mathbf{a(2)}$ | NO | |
| | | - | $\mathbf{a(2)}$ | - | |
| | | - | **NO** | - | |

1. Le littéral $b(X,Y)$ n'a aucune variable libre et comme $b(1,2)$ respectant la substitution partielle $\theta = \{X/1, Y/2\}$ appartient à $IN$ une substitution complète est trouvée. Le littéral de tête $a(X+Y)$ prend alors les valeurs de la substitution $\theta$. Ainsi, $a(3)$ est ajouté à $IN$ et devra être propagé.

Ensuite, une nouvelle tentative d'instanciation de la règle $r_2$ pour le littéral à propager $a(2)$ est réalisée. On repart de la dernière substitution de la règle avec $\theta = \{X/1, Y/2\}$ et on recherche une nouvelle substitution pour le littéral $b(X,Y)$. Comme $b(X,Y)$ ne possède aucune variable libre, il ne peut avoir d'autres substitutions que la substitution actuelle. Le littéral $a(X)$ échoue également car plus aucune valeur n'est disponible (il ne peut pas prendre pour valeur celle du littéral à propager $a(2)$ ni celle de $a(3)$ car il est apparu après $a(2)$ dans $IN$). On revient alors sur le littéral $a(Y)$ qui lui aussi ne possède aucune autre substitution car le littéral récursif courant n'accepte que la valeur 2 du littéral à propager $a(2)$. Étant donné qu'il ne reste plus de littéraux récursifs, l'instanciation de la règle se termine. Le littéral $a(2)$ n'a plus aucune règle permettant de le propager. On passe alors à la propagation du littéral $a(3)$ en tentant à nouveau d'instancier la règle $r_2$. Cette instanciation ne donnera rien et comme plus aucune règle ne peut être déclenchée ou appliquée, l'ensemble $IN = \{a(1), a(2), a(3), b(1,1), b(1,2)\}$ est un answer set de $P_{25}$.

# 3 Langage d'ASPeRiX et extensions

Le solveur ASPeRiX est capable de traiter des programmes logiques normaux contenant des variables, des symboles de fonction et des calculs arithmétiques. La seule restriction est que les règles soient sûres ce qui signifie que toute variable de chaque règle doit apparaître dans le corps positif.

## 3.1 Syntaxe

### 3.1.1 Terme

Un terme peut être une constante numérique, une constante symbolique, une chaîne de caractère constante, un terme fonctionnel, une variable ou une expression arithmétique.

**Constante numérique** Une constante numérique est un entier.
**Exemples :** $0$, $512$, -28, $\dots$

**Constante symbolique** Une constante symbolique est une chaîne comprenant des lettres, des chiffres et des tirets bas qui commence par une lettre minuscule.
**Exemples :** $toto$, $a1b2$, $a_B$, $\dots$

**Chaîne de caractère constante** Une chaîne de caractère constante est une suite de caractères entre guillemets.
**Exemples :** "$toto$", "$XyZ$", "$1Ab$", "*$z$2-*", $\dots$

**Terme fonctionnel** Un terme fonctionnel est un symbole de fonction (constante symbolique ou chaîne de caractère constante) suivi d'une liste de termes entre parenthèses.
**Exemples :** $f(1,2,3)$, $f2(a,2+X)$, "$f$"$(g(h([1,2])))$, $\dots$

**Variable** Une variable est une chaîne comprenant des lettres, des chiffres et des tirets bas qui commence par une lettre majuscule.
**Exemples :** $X$, $X_2$, $X_y_Z$, $\dots$

**Expression arithmétique** Une expression arithmétique est construite à partir de constantes numériques, de variables, des opérateurs binaires $+, -, *, /$, de l'opérateur modulo $mod$, de l'opérateur de valeur absolue $abs$ et de parenthèses.
**Exemples :** $4 * 3$, $(6\, mod\, 2)$, $X + 3$, $\dots$

### 3.1.2 Atome

Un atome est soit une formule atomique, soit un atome relationnel ou bien une affectation.

**Formule atomique** Une formule atomique est un symbole de prédicat (constante symbolique ou chaîne de caractère constante), suivi éventuellement d'une liste de termes entre parenthèses.
**Exemples :** $toto$, $f(X)$, $f(g(titi,2))$, $\dots$

**Suite d'atomes** Une tête de règle peut contenir une suite d'atomes. Celle-ci est représentée par un symbole de prédicat (constante symbolique ou chaîne de caractère constante), suivi d'une liste de termes entre parenthèses $t_1..t_n$ où $t_1$ et $t_n$ sont des constantes numériques ou des variables.
**Exemple :** Le fait $n(1..5)$. est équivalent à $n(1)$. $n(2)$. $n(3)$. $n(4)$. $n(5)$.

**Atome relationnel** Un atome relationnel est une comparaison entre deux termes. Les opérateurs autorisés sont $==$, !=, $<$, $<=$, $>$, $>=$. Les atomes relationnels apparaissent seulement dans le corps positif des règles.
**Exemples :** $4 >= -7$, *toto* != 2, $\ldots$

**Atome d'affectation** Un atome d'affectation est de la forme $variable = terme$. Les atomes d'affectation apparaissent seulement dans le corps positif des règles.
**Exemples :** $X = 2$, $Y = f(X)$, $\ldots$

### 3.1.3 Littéral

Un littéral est une formule atomique $a$ ou la négation forte d'une formule atomique -$a$.
**Exemples :** -$toto$, $f(a)$ -$f(a)$ $\ldots$

### 3.1.4 Règle

ASPeRiX autorise 3 types de règles :

- Fait : $tete.$
- Règle normale : $tete : -corps$
- Contrainte : $: -corps$

où $tete$ est un littéral et $corps$ une liste d'élements séparés par des virgules. Chaque élément peut être un littéral, éventuellement précédé par $not$, un atome relationnel ou un atome d'affectation.
**Exemples :**

- $p(1)$.
- $q(Y + 1)$ :- $p(Y)$, $Y$ != 3, $not$ $q(Y)$.
- :- -$q(2)$.

## 3.2 Respect d'ASP-Core2 et ordonnancement des termes

ASP-Core-2[6] est une standardisation du langage ASP qui a pour but d'unifier les différentes syntaxes et sémantiques proposées par les solveurs. Le solveur ASPeRiX a donc été modifié pour se conformer aux spécificités d'ASP-Core-2. La syntaxe et la sémantique d'ASP-Core-2 étant très complètes, la conformité se limite aux notions traitées par le solveur ASPeRiX. Cela exclut notamment les disjonctions en tête de règles ASP, les règles de choix ou bien encore les agrégats acceptés par des solveurs traditionnels tels que Clasp et DLV.

Le solveur ASPeRiX a subi diverses modifications au niveau de la grammaire du langage devant être respectée par un programme ASP. Certains éléments acceptés auparavant par la grammaire d'ASPeRiX comme la négation d'un atome en tête de règle ont été retirés et d'autres en revanche ont été ajoutés comme la possibilité de comparer entre eux différents types de termes.

Pour cela, un ordonnancement total $\preceq$ des termes a été effectué de manière à se conformer au standard ASP-Core2. Ainsi, l'ordre des termes respecte la hiérarchie suivante : constante numérique $\preceq$ constante symbolique $\preceq$ chaîne de caractère constante$\preceq$ symbole de fonctions. De plus,

[6]https ://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf

- les constantes numériques sont comparées par ordre croissant ce qui signifie que $a \preceq b$ si et seulement si $a \leq b.$ ;
- les constantes symboliques sont ordonnées de manière lexicographique ;
- les chaînes de caractères constantes sont ordonnées de manière lexicographique ;
- Soit $t = f(t_1, \ldots, t_m)$ et $u = g(u_1, \ldots, u_n)$ deux termes fonctionnels. Ces derniers sont ordonnés par les conditions suivantes :
  1. par l'arité de leur symbole de fonction, ce qui signifie que si $m < n$ alors $t \preceq u$ ;
  2. si les arités sont identiques, alors les noms des symboles de fonction sont ordonnés de manière lexicographique ;
  3. si les arités et les noms des symboles de fonction sont identiques, alors les arguments sont comparés entre eux ce qui signifie que si $t_1 < u_1$ alors $t \preceq u$ ;

**Exemple 26.** Voici quelques exemples d'ordonnancement des termes :

- Constantes numériques : $2 \preceq 7,\ \ 2 \preceq toto,\ \ 2 \preceq f(a)$
- Constantes symboliques et chaînes de caractères constantes : $titi \preceq toto,\ \ toto \preceq "titi",$ $"titi" \preceq "toto",\ \ titi \preceq f(a)$
- Termes fonctionnels : $q(a,b) \preceq p(a,b,c),\ \ p(a,b) \preceq q(a,b),\ \ p(a,b) \preceq p(b,b)$

## 3.3 Intégration des listes

Une gestion des listes a été intégrée dans ASPeRiX. Elle s'appuie sur la même représentation que les listes du langage Prolog. Ainsi, une liste de termes est représentée sous la forme $[t_1, ..., t_n]$ où $t_1, ..., t_n$ sont des termes ou bien alors sous la forme $[t|l]$ où $t$ est un terme et $l$ est une liste de termes.

**Quelques exemples de listes :** $[],\ \ [a,b,c],\ \ [a|[b,c]],\ \ \ldots$

Des prédicats externes ont été ajoutés pour manipuler plus facilement les listes à l'aide d'une bibliothèque dynamique $lists.so$ disponible dans la version $0.2.5$ d'ASPeRiX. Dans les prédicats affichés ci-dessous, le symbole "+" devant un argument représente un paramètre d'entrée tandis que le symbole "-" représente un paramètre de sortie.

- $\#append(+Liste1, +Liste2, -Liste3)$ est vrai si $Liste3$ est la concaténation des listes $Liste1$ et $Liste2$.
- $\#del(+Liste1, +Terme, -Liste2)$ est vrai si $Liste2$ est la liste $Liste1$ privée de toutes les occurrences de $Terme$.
- $\#delFirst(+Liste1, +Terme, -Liste2)$ est vrai si $Liste2$ est la liste $Liste1$ privée de la première occurrence de $Terme$.
- $\#delNth(+Liste1, +N, -Liste2)$ est vrai si $Liste2$ est la liste $Liste1$ privée du terme à la position $N$.
- $\#head(+Liste, -Terme)$ est vrai si $Terme$ est le premier terme de la liste $Liste$.

- $\#insLast(+Liste1, +Terme, -Liste2)$ est vrai si $Liste2$ est la liste obtenue en ajoutant $Terme$ à la fin de la liste $Liste1$.
- $\#insNth(+Liste1, +Terme, +N, -Liste2)$ est vrai si $Liste2$ est la liste obtenue en ajoutant $Terme$ à la liste $Liste1$ en position $N$.
- $\#last(+Liste, -Terme)$ est vrai si $Terme$ est le dernier élément de la liste $Liste$.
- $\#length(+Liste, -Taille)$ est vrai si $Taille$ est la longueur de la liste $Liste$.
- $\#list(-Liste)$ est vrai si $Liste$ est une liste vide.
- $\#list(+Terme_1, ..., +Terme_n, -Liste)$ $(n > 0)$ est vrai si $Liste$ est la liste contenant les éléments $Terme_1, ..., Terme_n$ dans le même ordre.
- $\#member(+Terme, +Liste)$ est vrai si la liste $Liste$ contient l'élement $Term$.
- $\#memberNth(+Liste, +N, -Terme)$ est vrai si $Terme$ est l'élément à la position $N$ de la liste $Liste$.
- $\#range(+Debut, +Fin, +Ecart, -Liste)$ est vrai si $Liste$ est la liste des entiers entre $Debut$ et $Fin$ avec un intervalle de $Ecart$.
- $\#reverse(+Liste1, -Liste2)$ est vrai si $Liste2$ est la liste renversée de $Liste1$ ;
- $\#subList(+Liste1, +Liste2)$ est vrai si $Liste1$ est une sous-liste de $Liste2$.
- $\#tail(+Liste1, -Liste2)$ est vrai si $Liste2$ représente la liste $Liste1$ privée de son premier élément.

Par ailleurs, il est possible d'omettre le paramètre de sortie d'un prédicat externe. Dans ce cas, le prédicat externe sera substitué par la valeur du paramètre de sortie durant le calcul.

**Exemples :** $\#length([a, b, c]) > 2$, $f(\#tail([a, b, c]))$, $X = \#del([a, b, b, c], b)$

Dans cette section, nous avons étudié en détail le solveur `ASPeRiX` tant d'un point de vue théorique, avec l'utilisation des computations, que pratique, avec la présentation des algorithmes utilisés dans le solveur. Nous avons montré l'originalité et l'intérêt du traitement de programmes ASP basé sur les règles avec instanciation à la volée. Dans la suite de cette thèse, nous continuons à nous pencher sur la recherche des answer sets guidée par les règles en cherchant à expliquer pourquoi un littéral appartient (ou non) à l'interprétation partielle en cours de construction, ou pourquoi un calcul échoue. Pour cela, nous étudions la notion de justification dans le cadre des computations puis nous nous intéressons à la mise en œuvre de ces justifications pour améliorer la recherche d'answer sets dans le solveur `ASPeRiX` grâce au backjumping.

# 5

# Justifications des computations

Dans ce chapitre, la notion de *justification* est introduite. Les justifications en programmation logique ont pour but de fournir des informations sur la raison pour laquelle une propriété est vraie lors de la construction d'un answer set. Elles peuvent par exemple exprimer la raison pour laquelle un atome appartient ou n'appartient pas à un answer set. Les justifications offrent donc des moyens de mieux comprendre comment se déroule l'application des règles d'un programme et peut permettre de réaliser du débogage sur ce même programme.

Au cours de ces dernières années, plusieurs travaux sur la justification [45, 48, 54] et le débogage [23, 13, 9] sont apparus. Les travaux sur la justification abordent essentiellement la raison pour laquelle une interprétation est un answer set. [45] utilise une approche à base d'atomes en utilisant la sémantique bien-fondée tandis que [48] utilise une approche argumentative. Les travaux sur le débogage quant à eux permettent d'expliquer pourquoi une interprétation ne mène pas à un answer set et de proposer des réparations sur le programme initial pour obtenir le résultat escompté.

Ici, on s'interesse aux justifications des `ASPeRiX` computations qui utilisent une approche guidée par les règles. Une justification, appelée *raison* par la suite, sera représentée par un ensemble de règles instanciées numérotées obtenu lors des étapes d'une computation. Une raison sera associée à chaque atome pour justifier sa présence dans une interprétation partielle ainsi qu'à chaque règle ajoutée au programme au cours d'une computation. Cela permettra par la suite de pouvoir justifier des propriétés particulières d'une computation notamment la raison pour laquelle un atome est resté indéterminé ou bien encore la raison pour laquelle une computation ne mène pas à un answer set. Cette dernière raison peut notamment être utile dans une optique de backjumping qui sera présenté dans le chapitre suivant. En effet, connaître la raison d'un échec permet de pouvoir remonter plus facilement dans l'arbre de recherche et éviter d'explorer certaines branches menant à ce même échec.

# 1 Raison des littéraux et des règles

## 1.1 Les raisons des littéraux et des contraintes

Lors de la recherche d'answer set d'un programme $P$, on introduit la notion de *raison* d'un littéral ou d'une règle. Celle-ci va permettre de justifier la présence d'un littéral dans une interprétation partielle mbt $I$ et la présence d'une règle dans $P \cup K$ où $K$ représente l'ensemble des contraintes ajoutées à $P$ lors de la recherche d'answer set.

Les différentes raisons sont exprimées à partir des règles instanciées numérotées d'une computation. Ainsi, la raison d'un littéral présent dans une interprétation partielle mbt $I = \langle IN, MBT, OUT \rangle$ sera représentée par un ensemble de règles instanciées numérotées responsable de l'ajout du littéral dans $IN$, $MBT$ ou $OUT$. De la même façon, la raison d'une règle présente dans $P \cup K$ sera représentée par un ensemble de règles instanciées numérotées responsable de l'ajout de la règle dans $P \cup K$. En pratique, seules des contraintes sont ajoutées en cours de recherche mais, afin d'uniformiser le traitement des contraintes et des autres règles, des raisons seront définis pour toutes les règles du programme.

**Exemple 27.** Soit $P_{27}$ le programme composé des deux règles suivantes :

$$\left\{ \begin{array}{l} a(1). \\ b(X) \leftarrow a(X). \end{array} \right\}$$

La suite suivante $\langle K_i, R_i, I_i \rangle_{i=0}^{3}$ avec $I_i = \langle IN_i, MBT_i, OUT_i \rangle$ est un préfixe de mbt ASPeRiX computation pour $P_{27}$ :

$$\begin{array}{ll}
I_0 & = \langle \emptyset, \emptyset, \{\bot\} \rangle \\
R_0 & = \emptyset \\
K_0 & = \emptyset \\
\\
r_1 & = a(1). \in \Delta_{pro}(P_{27}, I_0, R_0) \\
I_1 & = \langle \{\mathbf{a(1)}\}, \emptyset, \{\bot\} \rangle \\
R_1 & = \langle \{\mathbf{r_1}\}, \emptyset, \emptyset \rangle \\
K_1 & = K_0 \\
\\
r_2 & = b(1) \leftarrow a(1). \in \Delta_{pro}(P_{27}, I_1, \{R_1\}) \\
I_2 & = \langle \{a(1), \mathbf{b(1)}\}, \emptyset, \{\bot\} \rangle \\
R_2 & = \langle \{r_1, \mathbf{r_2}\}, \emptyset, \emptyset \rangle \\
K_2 & = K_1 \\
\\
I_3 & = I_2 \\
R_3 & = R_2 \\
K_3 & = K_2
\end{array}$$

Dans ce préfixe de mbt ASPeRiX computation, la raison de l'ajout du littéral $a(1)$ à l'ensemble $IN$ de l'interprétation partielle $I_1$ est uniquement le déclenchement de la règle $r_1$. La raison du littéral $a(1)$ est donc $\{r_1\}$. En revanche, la raison de l'ajout du littéral $b(1)$ à l'ensemble $IN$ de l'interprétation partielle $I_2$ est le déclenchement de la règle $r_2$ ainsi que l'ensemble des règles qui ont permis de déduire les littéraux du corps de la règle de $r_2$ auparavant, c'est à dire la raison de la présence du littéral $a(1)$ dans l'ensemble $IN$ de l'interprétation partielle $I_1$. Ainsi, la raison du littéral $b(1)$ est $\{r_1, r_2\}$. Plus de détails concernant les raisons seront évoquées dans la définition 22.

La définition suivante présente quelques notions sur des couples qui seront utilisées par la suite.

**Définition 18.** Soit $E$ un ensemble de couples, on définit :

- $first(E) = \bigcup_{(a,b)\in E} \{a\}$
- $E \sqcup \{(a,b)\} = \begin{cases} E & \text{si } a \in first(E) \\ E \cup \{(a,b)\} & \text{sinon} \end{cases}$
- $E \ominus a = \{(a_1, b_1) \in E \mid a_1 \neq a\}$

**Notations.** Dans la suite, on note $r_i$ une règle de numéro $i$ dans une computation. Par abus de notation, les règles numérotées et non numérotées ne sont généralement pas distinguées.
On note $r_0$ une constante de numéro $0$. Elle est utilisée pour représenter une raison indépendante de toute révision effectuée lors d'une computation.

La définition suivante permet d'établir quelques propriétés sur les ensembles de règles numérotées.

**Définition 19.** Soit $R = \{r_{a_1}, \ldots, r_{a_n}\}$ un ensemble de règles numérotées, on a :

- $\forall r_i, r_j \in R, r_i < r_j$ si et seulement si $i < j$
- $max(R) = r_j$ tel que $\forall r_i \in R, i \leq j$.
- $min(R) = r_j$ tel que $\forall r_i \in R, i \geq j$.
- $R_{<r_i} = \{r \in R \mid r < r_i\}$
- $R_{\leq r_i} = \{r \in R \mid r \leq r_i\}$
- $R_{>r_i} = \{r \in R \mid r > r_i\}$
- $R_{\geq r_i} = \{r \in R \mid r \geq r_i\}$

Afin d'introduire les raisons d'un littéral dans une `ASPeRiX` computation, la notion d'interprétation partielle mbt est étendue en une *interprétation justifiée* où chaque second élément de couples ($litteral$, $raison$) représente la raison de la présence de $litteral$ dans l'interprétation partielle mbt justifiée. De la même manière, l'ensemble de règles $K$ correspondant aux contraintes ajoutées au programme est étendue en un *ensemble de règles justifiées* $\mathbb{K}$ où le premier élément de chaque couple de $\mathbb{K}$ est une contrainte ajoutée au programme $P$ au cours de la recherche d'answer set et le second élément est la raison pour laquelle la contrainte a été ajoutée au programme.

**Définition 20.** Soit $P$ un programme logique au premier ordre :

- Une *interprétation justifiée* pour $P$ est un triplet $\mathbb{I} = \langle \mathbb{IN}, \mathbb{MBT}, \mathbb{OUT} \rangle$ d'ensembles de couples $C = (litteral, raison)$ tel que $litteral$ appartient à la base de Herbrand de $P$ et $raison$ est un ensemble de règles instanciées numérotées inclus dans $ground(P)$. Par ailleurs, chaque couple $C$ présent dans $\mathbb{E} = \mathbb{IN} \cup \mathbb{MBT} \cup \mathbb{OUT}$ est tel que si $(a_1, b_1) \in \mathbb{E}$ et $(a_2, b_2) \in \mathbb{E}$ alors $a_1 \neq a_2$. Le triplet $I = \langle first(\mathbb{IN}), first(\mathbb{MBT}), first(\mathbb{OUT}) \rangle$ est une *interprétation partielle mbt* de $P$.

- Un *ensemble de règles justifiées* noté $\mathbb{K}$ est un ensemble de couples chacun composé d'une règle instanciée $r$ et d'un ensemble de règles instanciées numérotées inclus dans $ground(P)$.

**Notations.** Pour une interprétation justifiée $\mathbb{I} = \langle \mathbb{IN}, \mathbb{MBT}, \mathbb{OUT} \rangle$ et un ensemble de règles justifiées $\mathbb{K}$, on note $IN = first(\mathbb{IN})$, $MBT = first(\mathbb{MBT})$, $OUT = first(\mathbb{OUT})$, $I = \langle IN, MBT, OUT \rangle$ et $K = first(\mathbb{K})$.

La définition suivante permet d'extraire la raison d'un littéral et d'une règle à partir d'une interprétation partielle mbt justifiée et d'un ensemble de règles justifiées.

**Définition 21** (Raison d'un littéral et d'une règle)**.** Soient $P$ un programme logique au premier ordre, $\mathbb{I} = \langle \mathbb{IN}, \mathbb{MBT}, \mathbb{OUT} \rangle$ une interprétation justifiée, $\mathbb{K}$ un ensemble de règles justifiées, $a$ un littéral et $r$ une règle instanciée, on définit :

- $reason(a, \mathbb{I}) = Reas$ tel que $(a, Reas) \in \mathbb{IN} \cup \mathbb{MBT} \cup \mathbb{OUT}$[1]
- $reason(r, P, \mathbb{K}) = \begin{cases} \{r_0\} & \text{si } r \in ground(P) \\ Reas & \text{si } \exists Reas \text{ tel que } (r, Reas) \in \mathbb{K} \end{cases}$

Les raisons sont définies comme suit, relativement à une interprétation justifiée $\langle \mathbb{IN}, \mathbb{MBT}, \mathbb{OUT} \rangle$ :

**Les règles.** A chaque instance de règle et de contrainte du programme initial, on associe la raison $\{r_0\}$ correspondant à ce qui est indépendant de tout choix effectué lors de la recherche.

Les seules règles qui peuvent être ajoutées au cours de la procédure de recherche sont les contraintes ajoutées pour bloquer les règles lors d'une (Exclusion de règle) de la computation. À une contrainte générée pour bloquer une règle $r$ à l'étape $n$, on associe la raison $\{r_n\}$, le choix arbitraire justifiant à lui seul l'ajout.

**Les littéraux de $IN$.** Lors de l'étape de (Propagation) de la computation, on ajoute dans $IN$ la tête d'une instance de règle $r$ supportée et débloquée. Cet ajout est justifié par le fait que la règle est déclenchable : tous les littéraux du corps de $r$ sont déterminés avec les littéraux du corps positif qui sont dans $IN$ et les littéraux du corps négatif dans $OUT$. La raison de l'ajout de $tête(r)$ dans $IN$ est donc l'ensemble des raisons pour lesquelles les littéraux du corps positif (resp. négatif) sont dans $IN$ (resp. $OUT$), auquel on ajoute la règle elle-même. De la même façon, lors de l'étape de (Choix de règle) de la computation, on ajoute dans $IN$ la tête d'une instance de règle $r$ supportée et non bloquée. La raison de l'ajout de $tête(r)$ dans $IN$ est calculée de la même façon que pour l'étape de (Propagation) . La seule différence est que certains littéraux dans $OUT$ seront déterminés durant cette même étape.

**Les littéraux de $OUT$.** Lors d'un (Choix de règle) d'une computation à l'étape $n$, où une instance de règle $r$ est choisie pour être appliquée, les littéraux du corps négatif de la règle sont ajoutés dans $OUT$ avec pour seule justification que l'on a décidé d'appliquer la règle à cette étape.

**Les littéraux de $MBT$.** De la même façon que pour les littéraux de $IN$, lors de l'étape de (Mbt-propagation) de la computation, on ajoute dans $MBT$ la tête de toute instance de règle $r$ faiblement supportée et débloquée. Cet ajout est justifié par le fait que la règle est faiblement déclenchable : tous les littéraux du corps de $r$ sont déterminés avec les littéraux du corps positif qui sont dans $IN \cup MBT$ et les littéraux du corps négatif dans $OUT$. La raison de l'ajout

[1] Par la suite, le littéral appartiendra nécessairement à l'interprétation justifiée lors de l'utilisation de $reason(a, \mathbb{I})$. Par conséquent, aucun appel à $reason(a, \mathbb{I})$ ne pourra mener à un cas non défini.

de $t\hat{e}te(r)$ dans $MBT$ est l'ensemble des raisons pour lesquelles les littéraux du corps positif (resp. négatif) sont dans $IN \cup MBT$ (resp. $OUT$), auquel on ajoute la règle elle-même.

Lors d'une (Exclusion de règle) d'une computation à l'étape $n$, un littéral $l$ est ajouté à l'ensemble $MBT$ si on décide de bloquer arbitrairement une instance de règle $r$ dont le corps négatif contient uniquement ce littéral. La justification de cet ajout est uniquement ce choix de bloquer la règle à cette étape.

La définition d'une *computation justifiée* ci-après enrichit la définition d'une *mbt* `ASPeRiX` *computation* en y intégrant les raisons pour lesquelles on ajoute les littéraux dans l'interprétation partielle ainsi que les raisons pour lesquelles on ajoute des contraintes au programme. Par ailleurs, une computation justifiée n'aboutit pas nécessairement à un answer set contrairement à une mbt `ASPeRiX` computation ce qui signifie que le critère de convergence n'est plus imposé. Autrement dit, une computation justifiée correspond à une tentative de construction d'un answer set. Si la computation justifiée converge, un answer set est déterminé, sinon il s'agit d'un échec.

**Définition 22** (Computation justifiée)**.** Soit $P$ un programme logique au premier ordre. Une *computation justifiée* pour $P$ est une suite $\langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{\infty}$ avec $\mathbb{K}_i$ un ensemble de règles justifiées, $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ un triplet d'ensembles de règles instanciées et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$ une interprétation justifiée qui satisfont les conditions suivantes :

- $\mathbb{K}_0 = \emptyset$, $R_0 = \langle \emptyset, \emptyset, \emptyset \rangle$, $\mathbb{I}_0 = \langle \emptyset, \emptyset, \{(\bot, \{r_0\})\} \rangle$ ,
- (Revision) $\forall i \geq 1$,

(Propagation) $\mathbb{K}_i = \mathbb{K}_{i-1}$, $R_i = \langle R_{i-1}^{app} \cup \{r_i\}, R_{i-1}^{mbt}, R_{i-1}^{excl} \rangle$
avec $r_i \in \Delta_{pro}(P, I_{i-1}, R_{i-1}^{app})$
et $\mathbb{I}_i = \langle \mathbb{IN}_{i-1} \sqcup \{(t\hat{e}te(r_i), Reason)\}, \mathbb{MBT}_{i-1} \ominus t\hat{e}te(r_i), \mathbb{OUT}_{i-1} \rangle$
avec $Reason = \bigcup_{l \in corps(r_i)} reason(l, \mathbb{I}_{i-1}) \ \cup \ \{r_i\}$

ou (Mbt-propagation) $\mathbb{K}_i = \mathbb{K}_{i-1}$, $R_i = \langle R_{i-1}^{app}, R_{i-1}^{mbt} \cup \{r_i\}, R_{i-1}^{excl} \rangle$
avec $r_i \in \Delta_{pro_mbt}(P, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{mbt})$
et $\begin{cases} \mathbb{I}_i = \langle \mathbb{IN}_{i-1}, \mathbb{MBT}_{i-1} \sqcup \{(t\hat{e}te(r_i), Reason)\}, \mathbb{OUT}_{i-1} \rangle & \text{si } t\hat{e}te(r_i) \notin IN_{i-1} \\ \mathbb{I}_i = \mathbb{I}_{i-1} & \text{sinon} \end{cases}$
avec $Reason = \bigcup_{l \in corps(r_i)} reason(l, \mathbb{I}_{i-1}) \ \cup \ \{r_i\}$

ou (Choix de règle) $\Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app}) = \emptyset$,
$\Delta_{pro_mbt}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{mbt}) = \emptyset$,
$\mathbb{K}_i = \mathbb{K}_{i-1}$, $R_i = \langle R_{i-1}^{app} \cup \{r_i\}, R_{i-1}^{mbt}, R_{i-1}^{excl} \rangle$
avec $r_i \in \Delta_{cho_mbt}(P, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{excl})$
et $I_i = \langle \mathbb{IN}_{i-1} \sqcup \{(t\hat{e}te(r_i), Reason)\}, \mathbb{MBT}_{i-1} \ominus t\hat{e}te(r_i), \mathbb{OUT}_{i-1} \sqcup_{l \in corps^-(r_i)} \{(l, \{r_i\})\} \rangle$
avec $Reason = \bigcup_{l \in corps^+(r_i) \cup (corps^-(r_i) \cap OUT_{i-1})} reason(l, \mathbb{I}_{i-1}) \ \cup \ \{r_i\}$

ou (Exclusion de règle) $\Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app}) = \emptyset$,
$\Delta_{pro_mbt}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{mbt}) = \emptyset$,

$\mathbb{K}_i = \mathbb{K}_{i-1}$, $R_i = \langle R_{i-1}^{app}, R_{i-1}^{mbt}, R_{i-1}^{excl} \cup \{r_i\} \rangle$
$\mathbb{I}_i = \langle \mathbb{IN}_{i-1}, \mathbb{MBT}_{i-1} \sqcup \{(l, \{r_i\})\}, \mathbb{OUT}_{i-1} \rangle$
avec $r_i \in \Delta_{cho_mbt}(P, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{excl})$, $l \in corps^-(r_i)$ et $|corps^-(r_i)| = 1$

ou $\mathbb{K}_i = \mathbb{K}_{i-1} \sqcup \{(\bot \leftarrow \bigcup_{b \in corps^-(r_i)} not\ b., \{r_i\})\}$,
$R_i = \langle R^{app}_{i-1}, R^{mbt}_{i-1}, R^{excl}_{i-1} \cup \{r_i\}\rangle, \mathbb{I}_i = \mathbb{I}_{i-1}$
avec $r_i \in \Delta_{cho_mbt}(P, I_{i-1}, R^{app}_{i-1} \cup R^{excl}_{i-1})$ et $|corps^-(r_i)| > 1$

ou (Stabilité) $\mathbb{K}_i = \mathbb{K}_{i-1}$, $R_i = R_{i-1}$ et $\mathbb{I}_i = \mathbb{I}_{i-1}$,

Une computation justifiée *converge* si $\exists i \geq 0,\ \Delta_{cho_mbt}(P \cup K_i, I_i, R^{app}_i \cup R^{excl}_i) = \emptyset$ et $MBT_i = \emptyset$.

Ici, la règle $r_i$ sélectionnée lors de la (Revision) à une étape $i$ de la séquence $\langle \mathbb{K}_i, R_i, \mathbb{I}_i, \rangle^{\infty}_{i=0}$ est considérée porter le numéro $i$ dans les raisons.

**Exemple 28.** Soit $P_{28}$ le programme suivant :

$$\left\{\begin{array}{l} a(1). \\ f(1). \\ b(X) \leftarrow a(X), not\ c(X). \\ c(X) \leftarrow a(X), not\ b(X). \\ d(X) \leftarrow c(X). \\ e(X) \leftarrow f(X), not\ b(X). \end{array}\right\}$$

La suite suivante $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i\rangle^{8}_{i=0}$ avec $R_i = \langle R^{app}_i, R^{mbt}_i, R^{excl}_i\rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i\rangle$ est un préfixe de computation justifiée pour $P_{28}$ :

$\mathbb{I}_0 = \langle \emptyset, \emptyset, \{(\bot, \{r_0\})\}\rangle$
$R_0 = \langle \emptyset, \emptyset, \emptyset\rangle$
$\mathbb{K}_0 = \emptyset$

Au début, les faits $(a(1).)$ et $(f(1).)$ sont déclenchables.

(Propagation)
$r_1 = a(1). \in \Delta_{pro}(P_{28}, I_0, R^{app}_0)$
$\mathbb{I}_1 = \langle \{(\mathbf{a(1)}, \{\mathbf{r_1}\})\}, \emptyset, \{(\bot, \{r_0\})\}\rangle$
$R_1 = \langle \{\mathbf{r_1}\}, \emptyset, \emptyset\rangle$
$\mathbb{K}_1 = \mathbb{K}_0$

(Propagation)
$r_2 = f(1). \in \Delta_{pro}(P_{28}, I_1, R^{app}_1)$
$\mathbb{I}_2 = \langle \{(a(1), \{r_1\}), (\mathbf{f(1)}, \{\mathbf{r_2}\})\}, \emptyset, \{(\bot, \{r_0\})\}\rangle$
$R_2 = \langle \{r_1, \mathbf{r_2}\}, \emptyset, \emptyset\rangle$
$\mathbb{K}_2 = \mathbb{K}_1$

Plus aucune règle n'est déclenchable ($\Delta_{pro}(P_{28} \cup K_i, I_2, R^{app}_2) = \emptyset$ et $\Delta_{pro_mbt}(P_{28} \cup K_i, I_2, R^{app}_2 \cup R^{mbt}_2) = \emptyset$). Les seules (Revision) disponibles sont (Choix de règle), (Exclusion de règle) ou (Stabilité).

(Exclusion de règle)
$r_3 = b(1) \leftarrow a(1), not\ c(1). \in \Delta_{cho_mbt}(P_{28}, I_2, R^{app}_2 \cup R^{excl}_2)$
$\mathbb{I}_3 = \langle \{(a(1), \{r_1\}), (f(1), \{r_2\})\}, \{(\mathbf{c(1)}, \{\mathbf{r_3}\})\}, \{(\bot, \{r_0\})\}\rangle$
$R_3 = \langle \{r_1, r_2\}, \emptyset, \{\mathbf{r_3}\}\rangle$
$\mathbb{K}_3 = \mathbb{K}_2$

Après avoir ajouté $c(1)$ dans l'ensemble $MBT$, la règle $(d(1) \leftarrow c(1).)$ est déclenchable.

(Mbt-propagation)
$r_4 = d(1) \leftarrow c(1). \in \Delta_{pro_mbt}(P_{28}, I_3, R_3^{app} \cup R_3^{mbt})$
$\mathbb{I}_4 = \langle\{(a(1),\{r_1\}),(f(1),\{r_2\})\},\{(c(1),\{r_3\}),(\mathbf{d(1)},\{\mathbf{r_3},\mathbf{r_4}\})\},\{(\bot,\{r_0\})\}\rangle$
$R_4 = \langle\{r_1,r_2\},\{\mathbf{r_4}\},\{r_3\}\rangle$
$\mathbb{K}_4 = \mathbb{K}_3$

À nouveau, plus aucune règle n'est déclenchable et les seules (Revision) disponibles sont (Choix de règle), (Exclusion de règle) ou (Stabilité).

(Choix de règle)
$r_5 = c(1) \leftarrow a(1), not\ b(1). \in \Delta_{cho_mbt}(P_{28}, I_4, R_4^{app} \cup R_4^{excl})$
$\mathbb{IN}_5 = \{(a(1),\{r_1\}),(f(1),\{r_2\}),(\mathbf{c(1)},\{\mathbf{r_1},\mathbf{r_5}\})\}$
$\mathbb{MBT}_5 = \{(d(1),\{r_3,r_4\})\}$
$\mathbb{OUT}_5 = \{(\bot,\{r_0\}),(\mathbf{b(1)},\{\mathbf{r_5}\})\}$
$R_5 = \langle\{r_1,r_2,\mathbf{r_5}\},\{r_4\},\{r_3\}\rangle$
$\mathbb{K}_5 = \mathbb{K}_4$

Après avoir ajouté $c(1)$ dans l'ensemble $IN$, la règle $(d(1) \leftarrow c(1).)$ est déclenchable.

(Propagation)
$r_6 = d(1) \leftarrow c(1). \in \Delta_{pro}(P_{28}, I_5, R_5^{app})$
$\mathbb{IN}_6 = \{(a(1),\{r_1\}),(f(1),\{r_2\}),(c(1),\{r_1,r_5\}),(\mathbf{d(1)},\{\mathbf{r_1},\mathbf{r_5},\mathbf{r_6}\})\}$
$\mathbb{MBT}_6 = \emptyset$
$\mathbb{OUT}_6 = \{(\bot,\{r_0\}),(b(1),\{r_5\})\}$
$R_6 = \langle\{r_1,r_2,r_5,\mathbf{r_6}\},\{r_4\},\{r_3\}\rangle$
$\mathbb{K}_6 = \mathbb{K}_5$

Après avoir ajouté $b(1)$ dans l'ensemble $OUT$, la règle $(e(1) \leftarrow f(1), not\ b(1).)$ est déclenchable.

(Propagation)
$r_7 = e(1) \leftarrow not\ b(1). \in \Delta_{pro}(P_{28}, I_6, R_6^{app})$
$\mathbb{IN}_7 = \{(a(1),\{r_1\}),(f(1),\{r_2\}),(c(1),\{r_1,r_5\}),(d(1),\{r_1,r_5,r_6\}),(\mathbf{e(1)},\{\mathbf{r_2},\mathbf{r_5},\mathbf{r_7}\})\}$
$\mathbb{MBT}_7 = \emptyset$
$\mathbb{OUT}_7 = \{(\bot,\{r_0\}),(b(1),\{r_5\})\}$
$R_7 = \langle\{r_1,r_2,r_5,r_6,\mathbf{r_7}\},\{r_4\},\{r_3\}\rangle$
$\mathbb{K}_7 = \mathbb{K}_6$

Plus aucune règle ne peut être déclenchée ou appliquée, la seule action disponible pour le reste de la suite est (Stabilité).
$\mathbb{I}_8 = \mathbb{I}_7$
$R_8 = R_7$
$\mathbb{K}_8 = \mathbb{K}_7$

Lorsqu'un littéral est présent dans l'ensemble $IN_i$ (respectivement $MBT_i$ et $OUT_i$) d'un préfixe de computation justifiée, la règle de plus grand numéro de sa raison est comprise dans l'intervalle $[0..i]$. En effet, chaque raison est construite à partir des raisons déjà calculées auparavant. Le lemme suivant expose cette propriété.

**Lemme 1.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i\rangle_{i=0}^{n}$ un préfixe de computation justifiée pour $P$ avec $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i\rangle$. Soit $i \in [0..n]$.
$\forall a \in IN_i \cup MBT_i \cup OUT_i, max(reason(a, \mathbb{I}_i)) \leq r_i$.

*Démonstration.* (du lemme 1) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i\rangle_{i=0}^{n}$ un préfixe de computation justifiée pour $P$ avec $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i\rangle$.

On montre par récurrence sur $i \in [0..n]$ que :
$\forall a \in IN_i \cup MBT_i \cup OUT_i, max(reason(a, \mathbb{I}_i)) \leq r_i$.

- Si $i = 0$, l'unique littéral appartenant à $IN_0 \cup MBT_0 \cup OUT_0$ est $\perp$ d'après la définition d'une computation justifiée et $reason(\perp, \mathbb{I}_0) = r_0$. Donc $\forall a \in IN_0 \cup MBT_0 \cup OUT_0$, $max(reason(a, \mathbb{I}_0)) \leq r_0$.

- On suppose qu'à l'étape i $(H)$ $\forall a \in IN_i \cup MBT_i \cup OUT_i, max(reason(a, \mathbb{I}_i)) \leq r_i$ et on montre que la propriété reste vraie à l'étape $i+1$. D'après la définition d'une computation justifiée :

  – $\forall a \in IN_{i+1} \backslash IN_i, reason(a, \mathbb{I}_{i+1}) = \bigcup_{l \in corps(r_{i+1}) \cap (IN_i \cup OUT_i)} reason(l, \mathbb{I}_i) \cup \{r_{i+1}\}$. D'après l'hypothèse de récurrence $(H), \forall l \in IN_i \cup MBT_i \cup OUT_i, max(reason(l, \mathbb{I}_i))$ $\leq r_i$ donc $max(reason(a, \mathbb{I}_{i+1})) = r_{i+1} \leq r_{i+1}$.

  – $\forall a \in MBT_{i+1} \setminus MBT_i$,

    $reason(a, \mathbb{I}_{i+1}) = \bigcup_{l \in corps(r_{i+1})} reason(l, \mathbb{I}_i) \cup \{r_{i+1}\}$. D'après l'hypothèse de récurrence $(H)$, $\forall l \in IN_i \cup MBT_i \cup OUT_i$, $max(reason(l, \mathbb{I}_i)) \leq r_i$ donc $max(reason(a, \mathbb{I}_{i+1})) = r_{i+1} \leq r_{i+1}$.

    OU $reason(a, \mathbb{I}_{i+1}) = \{r_{i+1}\}$ donc $max(reason(a, \mathbb{I}_{i+1})) = r_{i+1} \leq r_{i+1}$.

  – $\forall a \in OUT_{i+1} \setminus OUT_i$, $reason(a, \mathbb{I}_{i+1}) = \{r_{i+1}\}$ donc $max(reason(a, \mathbb{I}_{i+1})) = r_{i+1} \leq r_{i+1}$.

  De plus, $\forall a \in IN_i \cup MBT_i \cup OUT_i$, $max(reason(a, \mathbb{I}_i)) \leq r_i$ d'après l'hypothèse de récurrence $(H)$ et $r_i \leq r_{i+1}$. Par conséquent, $\forall a \in IN_{i+1} \cup MBT_{i+1} \cup OUT_{i+1}$, $max(reason(a, \mathbb{I}_{i+1})) \leq r_{i+1}$.

$\square$

Le théorème suivant met en avant quelques propriétés d'une computation justifiée.

**Théorème 4.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soit $a$ un littéral.

- Les ensembles $R^{app}$, $R^{mbt}$ et $R^{excl}$ sont croissants.

- si $a \in IN_n$ alors $a$ est obtenu par (Propagation) ou (Choix de règle) et $\exists i \leq n$ tel que $r_i \in R_i^{app} \backslash R_{i-1}^{app}, tête(r_i) = a, max(reason(a, \mathbb{I}_i)) = r_i$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$.

- si $a \in MBT_n$ alors :

    $a$ est obtenu par (Mbt-propagation) et $\exists i \leq n$ tel que $r_i \in R_i^{mbt} \setminus R_{i-1}^{mbt}$, $tête(r_i) = a$, $max(reason(a, \mathbb{I}_i)) = r_i$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$.

  OU $a$ est obtenu par (Exclusion de règle) et $\exists i \leq n$ tel que $r_i \in R_i^{excl} \backslash R_{i-1}^{excl}$, $corps^-(r_i) = \{a\}$, $max(reason(a, \mathbb{I}_i)) = r_i$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$.

- si $a \in OUT_n$ alors :

$a = \bot$

OU $a$ est obtenu par (Choix de règle) et $\exists i \leq n$ tel que $r_i \in R_i^{app} \setminus R_{i-1}^{app}$, $a \in corps^-(r_i)$, $max(reason(a, \mathbb{I}_i)) = r_i$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$.

*Démonstration.* (du théorème 4) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soit $a$ un littéral.

- Les ensembles $R^{app}$, $R^{mbt}$ et $R^{excl}$ sont croissants par définition d'une computation justifiée.

- si $a \in IN_n$ alors d'après la définition d'une computation justifiée, $a$ est obtenu par (Propagation) ou (Choix de règle) à une étape $i \in [1..n]$. Ainsi, $\exists i \leq n$ tel que $r_i \in R_i^{app} \setminus R_{i-1}^{app}$ et $tête(r_i) = a$. De plus, d'après la définition d'une computation justifiée, $reason(a, \mathbb{I}_i) = \bigcup_{l \in corps(r_i) \cap (IN_{i-1} \cup OUT_{i-1})} reason(l, \mathbb{I}_{i-1}) \cup \{r_i\}$. Or, d'après le lemme 1, $\forall l \in IN_{i-1} \cup MBT_{i-1} \cup OUT_{i-1}$, $max(reason(l, \mathbb{I}_{i-1})) \leq r_{i-1}$. Par conséquent, $max(reason(a, \mathbb{I}_i)) = r_i$. Par ailleurs, comme $\mathbb{IN}$ est croissant et monotone, on a $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$.

- si $a \in MBT_n$ alors d'après la définition d'une computation justifiée, $a$ est obtenu par (Mbt-propagation) ou (Exclusion de règle) à une étape $i \in [1..n]$. Ainsi, d'après la définition d'une computation justifiée, $\exists i \leq n$ tel que :

  $r_i \in R_i^{mbt} \setminus R_{i-1}^{mbt}$, $tête(r_i) = a$ et $max(reason(a, \mathbb{I}_i)) = r_i$ par le même raisonnement que pour $a \in IN_n$.

  OU $r_i \in R_i^{excl} \setminus R_{i-1}^{excl}$ et $corps^-(r_i) = \{a\}$. De plus, $reason(a, \mathbb{I}_i) = \{r_i\}$ donc $max(reason(a, \mathbb{I}_i)) = r_i$.

  En outre, comme $a \in MBT_n$, on a, d'après la définition d'une computation justifiée, $\forall j$ tel que $j \geq i$ et $j \leq n$, $a \in MBT_j$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$.

- si $a \in OUT_n$ alors d'après la définition d'une computation justifiée :

  $a = \bot$ car $\bot \in OUT_0$ et $OUT$ est croissant,

  OU Par le même raisonnement que pour $a \in IN_n$, $a$ est obtenu par (Choix de règle) et $\exists i \leq n$ tel que $r_i \in R_i^{app} \setminus R_{i-1}^{app}$, $a \in corps^-(r_i)$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$. De plus, $reason(a, \mathbb{I}_i) = \{r_i\}$ d'après la définition d'une computation justifiée donc $max(reason(a, \mathbb{I}_i)) = r_i$.

□

## 1.2 Raison des littéraux indéterminés

Lors de la construction d'un answer set par une computation, un littéral peut rester indéterminé (ni dans $IN$ ni dans $OUT$) jusqu'à la fin de la construction. La définition suivante permet de déterminer une *raison* pour laquelle un littéral est resté indéterminé en vérifiant pourquoi chaque instance de règle susceptible de déduire le littéral (c'est à dire chaque instance de règle ayant pour tête le littéral indéterminé) n'a pu être déclenchée. Une instance de règle est déclenchable

si son corps positif est inclus dans l'ensemble $IN \cup MBT$ et si son corps négatif est inclus dans $OUT$.

Plusieurs cas peuvent empêcher une instance de règle susceptible de déduire le littéral indéterminé d'être déclenchée. Tout d'abord, il peut s'agir d'un (ou plusieurs) littéral du corps positif qui appartient à l'ensemble $OUT$ (cas $(i)$ de la définition 24) ou est resté indéterminé (cas $(v)$ de la définition 24). Ensuite, il peut s'agir d'un (ou plusieurs) littéral du corps négatif qui appartient à l'ensemble $IN$ ou $MBT$ et bloque la règle (cas $(ii)$ et $(iii)$ de la définition 24). Enfin, il peut s'agir d'une instance de règle bloquée arbitrairement lors d'un point de choix par l'ajout d'une contrainte dans le programme (cas $(iv)$ de la définition 24) qui empêche l'ensemble du corps négatif d'appartenir à $OUT$. Chacun de ces cas va permettre de déterminer une raison pour chacune des instances de règles susceptibles de déduire le littéral indéterminé et d'ainsi calculer une raison de l'indétermination de ce littéral.

Dans la définition suivante, une suite $\langle R_i, Litteraux_i, Reas_i \rangle_{i=0}^{\infty}$ représentera les différentes étapes du calcul d'une raison de l'indétermination d'un littéral $a$ où $R_i$ caractérisera l'ensemble des règles susceptibles de déduire le littéral indéterminé déjà traitées à l'étape $i$, $Litteraux_i$ caractérisera les littéraux indéterminés responsables de l'indétermination de $a$ (y compris $a$ lui même) et $Reas_i$ caractérisera une raison de l'indétermination de $a$ calculée par la suite. Le calcul d'une raison de l'indétermination d'un littéral se termine lorsque toutes les règles susceptibles de déduire le littéral indéterminé ont été traitées. Il est à noter que plusieurs raisons différentes peuvent être calculées pour un même littéral indéterminé.

**Exemple 29.** Considérons l'interprétation partielle $\langle \{x\}, \emptyset, \{c, d\} \rangle$, et supposons que $a$ n'est pas prouvable dans cette interprétation et que les deux seules règles (instanciées) concluant $a$ sont $(r_1 = a \leftarrow y,\ not\ c.)$ et $(r_2 = a \leftarrow x,\ not\ b,\ not\ d.)$.

$a$ n'est pas prouvable parce que, d'une part, $r_1$ n'est pas supportée, donc pas déclenchable ($y$ n'appartient pas à l'ensemble $IN$) et, d'autre part, $r_2$ n'a pas été déclenchée (alors qu'elle est pourtant applicable). $r_2$ a donc nécessairement été bloquée par (Exclusion de règle) en ajoutant une contrainte $(\bot \leftarrow not\ b,\ not\ d.)$. La raison pour laquelle $a$ n'est pas dans $IN$ sera l'union de la raison pour laquelle $y$ est indéterminé (car $y$ est ni dans $IN$ ni dans $OUT$) empêchant la règle $r_1$ d'être déclenchée et de la raison pour laquelle la règle $r_2$ a été bloquée par (Exclusion de règle).

La définition suivante de $hrule$ permet de récupérer toutes les règles instanciées d'un programme ayant pour tête le littéral $a$.

**Définition 23.** Soient $P$ un programme logique au premier ordre et $a$ un atome, $hrule(a, P) = \{r \in ground(P) \mid tête(r) = a\}$.

**Définition 24** (Raison des littéraux indéterminés)**.** Soient $P$ un programme logique au premier ordre, $\mathbb{I} = \langle \mathbb{IN}, \mathbb{MBT}, \mathbb{OUT} \rangle$ une interprétation justifiée, $\mathbb{K}$ un ensemble de règles justifiées et $a$ un littéral. On définit une *raison de l'indétermination d'un littéral*, notée $reason_{ind}(a, \mathbb{I}, P, \mathbb{K})$, grâce à une suite $\langle R_i, Litteraux_i, Reas_i \rangle_{i=0}^{\infty}$ où, pour chaque $i$, $R_i$ est un ensemble de règles instanciées, $Litteraux_i$ est un ensemble de littéraux, et $Reas_i$ est un ensemble de règles instanciées, telle que :

- $R_0 = \emptyset, Litteraux_0 = \{a\}, Reas_0 = \{r_0\}$
- $\forall i > 0$

$(i)$ $\exists l \in (corps^+(r_i) \cap OUT)$,

ou $(ii)$ $\exists l \in (corps^-(r_i) \cap IN)$,

ou $(iii)$ $\exists l \in (corps^-(r_i) \cap MBT)$,
avec $r_i \in \bigcup_{at \in Litteraux_{i-1}} hrule(at, P) \setminus R_{i-1}$,
$R_i = R_{i-1} \cup \{r_i\}$,
$Litteraux_i = Litteraux_{i-1}$,
$Reas_i = Reas_{i-1} \cup reason(l, \mathbb{I})$

ou $(iv)$ $\exists const \in first(\mathbb{K})$ tel que $const = (\bot \leftarrow \cup_{b \in corps^-(r_i)} not\ b.)$
et $reason(const, P, \mathbb{K}) = \{r_i\}$
avec $r_i \in \bigcup_{at \in Litteraux_{i-1}} hrule(at, P) \setminus R_{i-1}$,
$R_i = R_{i-1} \cup \{r_i\}$,
$Litteraux_i = Litteraux_{i-1}$,
$Reas_i = Reas_{i-1} \cup reason(const, P, \mathbb{K})$

ou $(v)$ $\exists l \in (corps^+(r_i) \setminus (IN \cup OUT))$,
avec $r_i \in \bigcup_{at \in Litteraux_{i-1}} hrule(at, P) \setminus R_{i-1}$,
$R_i = R_{i-1} \cup \{r_i\}$,
$Litteraux_i = Litteraux_{i-1} \cup \{l\}$,
$Reas_i = Reas_{i-1}$

ou $(vi)$ $R_i = R_{i-1}$,
$Litteraux_i = Litteraux_{i-1}$,
$Reas_i = Reas_{i-1}$

- (Convergence) $\exists i \geq 0,\ R_i = \bigcup_{at \in Litteraux_{i-1}} hrule(at, P)$

On dit que la suite $\langle R_i, Litteraux_i, Reas_i \rangle_{i=0}^{\infty}$ converge avec :

- $R_\infty = \bigcup_{i=0}^{\infty} R_i$,
- $Litteraux_\infty = \bigcup_{i=0}^{\infty} Litteraux_i$,
- $Reas_\infty = \bigcup_{i=0}^{\infty} Reas_i$.

On a alors $reason_{ind}(a, \mathbb{I}, P, K) = Reas_\infty$

# 2 Ensemble compatible et ensemble de blocage

Les règles présentes dans une raison peuvent avoir différents statuts selon les étapes de (Révision) effectuées lors de la computation. En effet, une règle peut être présente dans une raison parce qu'elle a été déclenchée ou choisie ou encore bloquée à une étape de la computation. La définition suivante va permettre de distinguer le statut des règles d'une raison $Reason$ pour un préfixe de computation justifiée $S$ :

- $Reason_S^{bloq}$ représente les règles de $Reason$ dont le fait qu'elles soient bloquées justifie leur présence dans $Reason$.
- $Reason_S^{nbloq}$ représente les règles $r_i$ de $Reason$ dont le fait de les avoir rendu applicables lors d'un (Choix de règle) en ajoutant le corps négatif dans $OUT$ justifie leur présence dans $Reason$. $Reason$ contient la raison d'au moins un littéral du corps négatif de $r_i$.
- $Reason_S^{decl}$ représente les règles $r_i$ de $Reason$ dont le déclenchement justifie leur présence dans $Reason$. $Reason$ contient la raison de l'ajout de la tête de chaque $r_i$ dans l'ensemble $IN$ ou $MBT$.

**Définition 25** (Types de raison)**.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{n}$ un préfixe de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$. Soit $Reason$ un ensemble de règles instanciées numérotées, on définit :

- $Reason_S^{bloq} = Reason \cap R_n^{excl}$
- $Reason_S^{nbloq} = \{r_i \in Reason \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \nsubseteq Reason\}$
- $Reason_S^{decl} = (Reason \cap R_n^{mbt}) \cup \{r_i \in Reason \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \subseteq Reason\}$

**Exemple 30.** En reprenant le programme $P_{28}$ de l'exemple 28 :

- Soit le préfixe $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{8}$ vu auparavant. Soit $ReasonE_1$ la raison de la présence du littéral $e(1)$ dans l'interprétation $\mathbb{I}_8$. On a $ReasonE_1 = reason(e(1), \mathbb{I}_8) = \{r_2, r_5, r_7\}$ avec $r_2 \in R_8^{app}$, $r_5 \in R_8^{app}$ et $r_7 \in R_8^{app}$.
  Ici, on a :
  - $ReasonE_{1S}^{bloq} = \emptyset$,
  - $ReasonE_{1S}^{nbloq} = \{r_5\}$ car $reason(head(r_5), \mathbb{I}_5) = reason(c(1), \mathbb{I}_5) = \{r_1, r_5\} \nsubseteq ReasonE_1$,
  - $ReasonE_{1S}^{decl} = \{r_2, r_7\}$ car $reason(head(r_2), \mathbb{I}_2) = reason(f(1), \mathbb{I}_2) = \{r_2\} \subseteq ReasonE_1$ et $reason(head(r_7), \mathbb{I}_7) = reason(e(1), \mathbb{I}_7) = \{r_5, r_7\} \subseteq ReasonE_1$.

  Intuitivement $e(1)$ est ajouté à l'ensemble $IN$ car la règle $r_7$ : $(e(1) \leftarrow f(1), not\, b(1).)$ a été *déclenchée*, $f(1)$ est dans $IN$ et $b(1)$ est dans $OUT$. Or, $f(1)$ est dans $IN$ car la règle $r_2$ : $(f(1).)$ a été *déclenchée* et $b(1)$ est dans $OUT$ car la règle $r_5$ : $(c(1) \leftarrow a(1), not\, b(1).)$ a été *débloquée* par (Choix de règle).

- Si l'on prend pour préfixe $S_1 = S_{i=0}^{4}$ et $ReasonD_1$ la raison de la présence du littéral $d(1)$ dans l'interprétation $\mathbb{I}_4$, on a $ReasonD_1 = reason(d(1), \mathbb{I}_4) = \{r_3, r_4\}$ avec $r_3 \in R_4^{excl}$ et $r_4 \in R_4^{mbt}$.
  Dans ce cas :
  - $ReasonD_{1S_1}^{bloq} = \{r_3\}$ car $r_3 \in R_4^{excl}$,
  - $ReasonD_{1S_1}^{nbloq} = \emptyset$,
  - $ReasonD_{1S_1}^{decl} = \{r_4\}$ car $r_4 \in R_4^{mbt}$.

  Intuitivement $d(1)$ est ajouté à l'ensemble $MBT$ car la règle $r_4$ : $(d(1) \leftarrow c(1).)$ a été *déclenchée* et $c(1)$ est dans $MBT$. Or, $c(1)$ est dans $MBT$ car la règle $r_3$ : $(b(1) \leftarrow a(1), not\, c(1).)$ a été *bloquée* par (Exclusion de règle).

Le lemme suivant établit des propriétés d'un sous-ensemble de règles $Reas$ d'une raison $Reason$. Il est à noter que certaines règles $r_i$ de l'ensemble $Reason_S^{decl}$ d'un préfixe de computation justifiée $S$ peut appartenir à l'ensemble $Reas_S^{nbloq}$. Cela est lié au fait que $Reas$ ne comprend qu'une partie des règles de $Reason$ qui ne satisfont pas nécessairement la condition pour appartenir à l'ensemble $Reas_S^{decl}$.

**Exemple 31.** En reprenant le programme $P_{28}$ de l'exemple 28, le préfixe $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{8}$ et la raison $ReasonE_1 = reason(e(1), \mathbb{I}_8) = \{r_2, r_5, r_7\}$ de l'exemple précédent.
Soit $ReasE_1 = \{r_2, r_7\} \subseteq ReasonE_1$.
Dans ce cas, la règle $r_7$ appartient à ${ReasE_1}_S^{nbloq}$ car $reason(head(r_7), \mathbb{I}_7) = reason(e(1), \mathbb{I}_7) = \{r_2, r_5, r_7\} \nsubseteq ReasE_1$ alors qu'elle appartient à ${ReasonE_1}_S^{decl}$ (d'après l'exemple précédent).
En revanche, la règle $r_2$ appartient à ${ReasE_1}_S^{decl}$ ainsi qu'à ${ReasonE_1}_S^{decl}$.

**Lemme 2.** Soit $P$ un programme logique au premier ordre. Soient $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{n}$ un préfixe de computation justifiée pour $P$ et $Reason$ un ensemble de règles instanciées numérotées. Soit $Reas \subseteq Reason$, alors :

- $Reas_S^{bloq} \subseteq Reason_S^{bloq}$
- $Reas_S^{nbloq} \subseteq Reason_S^{decl} \cup Reason_S^{nbloq}$
- $Reas_S^{decl} \subseteq Reason_S^{decl}$

*Démonstration.* (du lemme 2) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{n}$ un préfixe de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$. Soit $Reason$ un ensemble de règles instanciées numérotées. Soit $Reas \subseteq Reason$, alors d'après la définition 25 :

- $Reas_S^{bloq} = Reas \cap R_n^{excl}$ donc $Reas_S^{bloq} \subseteq Reason \cap R_n^{excl}$. Donc $Reas_S^{bloq} \subseteq Reason_S^{bloq}$.
- $Reas_S^{nbloq} = \{r_i \in Reas \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \nsubseteq Reas\}$ donc $Reas_S^{nbloq} \subseteq Reas \cap R_n^{app} \subseteq Reason \cap R_n^{app} \subseteq Reason \cap (R_n^{app} \cup R_n^{mbt})$. Donc, d'après la définition 25, $Reas_S^{nbloq} \subseteq Reason_S^{decl} \cup Reason_S^{nbloq}$.
- $Reas_S^{decl} = (Reas \cap R_n^{mbt}) \cup \{r_i \in Reas \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \subseteq Reas\}$. Or, $Reas \cap R_n^{mbt} \subseteq Reason \cap R_n^{mbt}$ et $\{r_i \in Reas \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \subseteq Reas\} \subseteq \{r_i \in Reason \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \subseteq Reason\}$. Donc, d'après la définition 25, $Reas_S^{decl} \subseteq Reason_S^{decl}$.

□

Lors d'une computation justifiée qui converge en un answer set $X$, les règles de $R^{app}$ correspondent aux règles génératrices du programme par rapport à $X$. Ainsi, un answer set est *compatible* avec une raison $Reas$ si :

- les règles de $R^{app}$ qui sont dans la raison car elles ont été déclenchées (appartenant à $Reas^{decl}$) sont des règles génératrices de $X$,
- les règles de $R^{app}$ qui sont dans la raison car elles ont été débloquées (appartenant à $Reas^{nbloq}$) ne sont pas bloquées par rapport à $X$,
- les règles de $R^{excl}$ qui sont dans la raison car elles ont été bloquées (appartenant à $Reas^{bloq}$) sont bloquées par rapport à $X$.

**Définition 26** (ensemble compatible)**.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{k}$ un préfixe de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$.
Soit $Reas$ un ensemble de règles instanciées tel que $Reas \subseteq R_k^{app} \cup R_k^{mbt} \cup R_k^{excl} \cup \{r_0\}$.
Un ensemble d'atomes $X$ est *compatible* avec $Reas$ si

$$\begin{cases} Reas_S^{decl} \subseteq GR_P(X) \\ \forall r \in Reas_S^{nbloq}, corps^-(r) \cap X = \emptyset \\ \forall r \in Reas_S^{bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$$

Une raison est une véritable raison d'une propriété de computation si chaque answer set compatible avec la raison satisfait la propriété. Dans la définition suivante, une raison d'une computation justifiée est un *ensemble de blocage* s'il ne peut aboutir à un answer set ce qui signifie qu'il n'existe aucun answer set compatible avec la raison.

**Définition 27** (ensemble de blocage)**.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{k}$ un préfixe de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$. Soit $Reas$ un ensemble de règles instanciées tel que $Reas \subseteq R_k^{app} \cup R_k^{mbt} \cup R_k^{excl} \cup \{r_0\}$.
On dit que $Reas$ est un *ensemble de blocage* s'il n'existe pas d'answer set $X$ pour $P$ compatible avec $Reas$.

# 3 Raison d'échec

Une computation peut être vue comme une branche d'un arbre de recherche où les nœuds correspondent aux choix d'appliquer une règle par (Choix de règle) ou de l'exclure par (Exclusion de règle). Dans cette partie, la raison pour laquelle une computation ne converge pas sera déterminée. Dans un premier temps, cela correspondra à déterminer pourquoi une branche de l'arbre de recherche mène à un échec. Ensuite, dans un second temps, il sera possible de déterminer pourquoi les branches gauche et droite d'un nœud de l'arbre mènent à un échec en combinant des computations de mêmes préfixes. Le fait de connaître la raison pour laquelle une computation ne converge pas permettra en particulier de comprendre pourquoi certaines branches de l'arbre de recherche se terminent en échec et d'éviter d'explorer des branches qui auront nécessairement le même échec à l'aide de la technique de backjumping (voir chapitre suivant).

## 3.1 Computation bloquée

La définition suivante introduit la notion de préfixe *bloqué* de computation justifiée $S$ pour un programme $P$. Intuitivement, un préfixe de $S$ est bloqué à une étape $k$ lorsque la seule action possible à partir de $k$ est (Stabilité) et que le critère de convergence n'est pas respecté. Cette situation intervient lorsque tous les déclenchements de règles susceptibles d'être effectués ne respectent plus la disjonction des ensembles $IN$, $MBT$ et $OUT$ (cas (Propagation-failure) et (Mbt-Propagation-failure) de la définition 28) ou bien lorsque les seules règles applicables susceptibles d'être déclenchées sont des contraintes ajoutées lors de la computation justifiée (cas ($\Delta_{cho_mbt}$ Convergence-failure) de la définition 28) ou encore lorsqu'un littéral $MBT$ ne peut plus être prouvé ce qui signifie qu'il ne peut plus être ajouté dans $IN$ (cas (Mbt-Convergence-failure) de la définition 28).

**Exemple 32.** (Propagation-failure)
Soit $r$ la règle $(c \leftarrow a, not\ b.)$ d'un programme $P$ avec $a \in IN$, $b \in OUT$ et $c \in OUT$. Le déclenchement de cette règle $r$ entraînerait la non-disjonction des ensembles $IN$ et $OUT$ car le littéral $c$ appartiendrait alors aux deux ensembles. La règle ne pourra donc jamais être déclenchée et restera dans $\Delta_{pro}$.

**Définition 28** (Préfixe bloqué)**.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{k}$ un préfixe de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. On dit que $S$ est *bloqué* si $S$ vérifie au moins une des conditions suivantes :

(Propagation-failure) $\Delta_{pro}(P \cup K_k, I_k, R_k^{app}) \neq \emptyset$,
$\forall r \in \Delta_{pro}(P \cup K_k, I_k, R_k^{app}), head(r) \in OUT_k$

ou (Mbt-Propagation-failure) $\Delta_{pro_mbt}(P \cup K_k, I_k, R_k^{app} \cup R_k^{mbt}) \neq \emptyset$,
$\forall r \in \Delta_{pro_mbt}(P \cup K_k, I_k, R_k^{app} \cup R_k^{mbt}), head(r) \in OUT_k$

ou ($\Delta_{cho_mbt}$ Convergence-failure) $\Delta_{pro}(P \cup K_k, I_k, R_k^{app}) = \emptyset$,
$\Delta_{pro_mbt}(P \cup K_k, I_k, R_k^{app} \cup R_k^{mbt}) = \emptyset$,
$\Delta_{cho_mbt}(P, I_k, R_k^{app} \cup R_k^{excl}) = \emptyset$,
$\Delta_{cho_mbt}(K_k, I_k, R_k^{app} \cup R_k^{excl}) \neq \emptyset$

ou (Mbt-Convergence-failure) $\Delta_{pro}(P \cup K_k, I_k, R_k^{app}) = \emptyset$,
$\Delta_{pro_mbt}(P \cup K_k, I_k, R_k^{app} \cup R_k^{mbt}) = \emptyset$,
$\Delta_{cho_mbt}(P, I_k, R_k^{app}) = \emptyset$,
$\mathbb{MBT}_k \neq \emptyset$

À partir des différentes situations de préfixe bloqué, on peut déterminer une raison d'*échec* qui correspond à l'ensemble des règles appliquées ou exclues lors de la computation justifiée qui sont responsables du fait que le préfixe est bloqué. S'il s'agit d'une propagation de règle $r$ ne respectant plus la disjonction des ensembles $IN$, $MBT$ et $OUT$ (cas (Propagation-failure) et (Mbt-Propagation-failure)), la raison d'échec sera la raison pour laquelle la tête de $r$ se retrouve à appartenir à ces différents ensembles. S'il s'agit d'une contrainte applicable ajoutée lors de la computation justifiée (cas ($\Delta_{cho_mbt}$ Convergence-failure)), la raison d'échec sera composée de chaque raison des littéraux du corps de la règle qui ont permis de rendre la règle applicable ainsi que la raison de l'ajout de la contrainte lors de la computation justifiée. Enfin, lorsque un littéral $MBT$ ne peut plus être prouvé (cas (Mbt-Convergence-failure)), la raison d'échec sera une raison pour laquelle le littéral est resté indéterminé ainsi que la raison qui a permis de déterminer que le littéral appartient à l'ensemble $MBT$. Par ailleurs, plusieurs raisons sont possibles pour un même préfixe bloqué car plusieurs règles ou plusieurs littéraux $MBT$ peuvent être responsables du préfixe bloqué.

**Définition 29** (Raison d'échec d'un préfixe bloqué)**.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{k}$ un préfixe bloqué de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. On définit une *raison d'échec de* $S$ par une règle instanciée $rf$, notée $reason_{Failure}(S, rf)$, pour les cas suivants :

Si $S$ vérifie (Propagation-failure)
$reason_{Failure}(S, rf) = \bigcup_{l \in corps(rf)} reason(l, \mathbb{I}_k) \ \cup$
$reason(rf, P, \mathbb{K}_k) \ \cup \ reason(head(rf), \mathbb{I}_k)$
avec $rf \in \Delta_{pro}(P \cup K_k, I_k, R_k^{app})$

ou Si $S$ vérifie (Mbt-Propagation-failure)
$reason_{Failure}(S, rf) = \bigcup_{l \in corps(rf)} reason(l, \mathbb{I}_k) \ \cup$
$reason(rf, P, \mathbb{K}_k) \ \cup \ reason(head(rf), \mathbb{I}_k)$
avec $rf \in \Delta_{pro_mbt}(P \cup K_k, I_k, R_k^{app} \cup R_k^{mbt})$

ou Si $S$ vérifie ($\Delta_{cho_mbt}$ Convergence-failure)
$reason_{Failure}(S, rf) = \bigcup_{l \in (corps^-(rf) \cap OUT_k)} reason(l, \mathbb{I}_k) \cup \bigcup_{l \in (corps^-(rf) \setminus OUT_k)} reason_{ind}(l, \mathbb{I}_k, P, \mathbb{K}_k) \cup reason(rf, P, \mathbb{K}_k)$
avec $rf \in \Delta_{cho_mbt}(K_k, I_k, R_k^{app} \cup R_k^{excl})$

De la même manière, on définit une raison d'échec de $S$ par un littéral $l$, notée $reason_{Failure}(S, l)$, pour le cas suivant :

Si $S$ vérifie (Mbt-Convergence-failure)
$reason_{Failure}(S, l) = reason_{ind}(l, \mathbb{I}_k, P, \mathbb{K}_k) \cup reason(l, \mathbb{I}_k)$
avec $l \in MBT_k$

## 3.2 Combinaison d'échec

Lorsque deux computations justifiées de même préfixe se retrouvent bloquées lors d'un point de choix sur une règle $r$, l'une en appliquant la règle par (Choix de règle) et l'autre en la bloquant par (Exclusion de règle), il est possible de combiner leurs raisons d'échecs respectives pour déterminer les règles des raisons d'échecs responsables du blocage. La règle du point de choix ne pouvant ni être appliquée ni exclue ne peut être en cause dans cet échec. L'échec est donc avéré avant d'effectuer ce point de choix. Cette combinaison permet ainsi de déceler les règles qui étaient déjà responsables du blocage avant même que le point de choix sur la règle $r$ ne soit fait.

**Exemple 33.** Soit $P_{33}$ le programme suivant :

$$\left\{ \begin{array}{ll} a \leftarrow not\ b. & e \leftarrow not\ f. \\ b \leftarrow not\ a. & f \leftarrow not\ e. \\ c \leftarrow not\ d. & \bot \leftarrow a, e. \\ d \leftarrow not\ c. & \bot \leftarrow a, f. \end{array} \right\}$$

La suite suivante $S_{i=0}^{3} = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{3}$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$ est un préfixe de computation justifiée pour $P_{33}$ :

$$\begin{array}{ll}
\mathbb{I}_0 & = \langle \emptyset, \emptyset, \{(\bot, \{r_0\})\} \rangle \\
R_0 & = \langle \emptyset, \emptyset, \emptyset \rangle \\
\mathbb{K}_0 & = \emptyset \\
 & \text{(Choix de règle)} \\
r_1 & = a \leftarrow not\ b. \in \Delta_{cho_mbt}(P_{33}, I_0, R_0^{app} \cup R_0^{excl}) \\
\mathbb{I}_1 & = \langle \{(\mathbf{a}, \{\mathbf{r_1}\})\}, \emptyset, \{(\bot, \{r_0\}), (\mathbf{b}, \{\mathbf{r_1}\})\} \rangle \\
R_1 & = \langle \{\mathbf{r_1}\}, \emptyset, \emptyset \rangle \\
\mathbb{K}_1 & = \mathbb{K}_0 \\
 & \text{(Choix de règle)} \\
r_2 & = c \leftarrow not\ d. \in \Delta_{cho_mbt}(P_{33}, I_1, R_1^{app} \cup R_1^{excl}) \\
\mathbb{I}_2 & = \langle \{(a, \{r_1\}), (\mathbf{c}, \{\mathbf{r_2}\})\}, \emptyset, \{(\bot, \{r_0\}), (b, \{r_1\}), (\mathbf{d}, \{\mathbf{r_2}\})\} \rangle \\
R_2 & = \langle \{r_1, \mathbf{r_2}\}, \emptyset, \emptyset \rangle \\
\mathbb{K}_2 & = \mathbb{K}_1 \\
 & \text{(Choix de règle)} \\
r_3 & = e \leftarrow not\ f. \in \Delta_{cho_mbt}(P_{33}, I_2, R_2^{app} \cup R_2^{excl}) \\
\mathbb{I}_3 & = \langle \{(a, \{r_1\}), (c, \{r_2\}), (\mathbf{e}, \{\mathbf{r_3}\})\}, \emptyset, \{(\bot, \{r_0\}), (b, \{r_1\}), (d, \{r_2\}), (\mathbf{f}, \{\mathbf{r_3}\})\} \rangle \\
R_3 & = \langle \{r_1, r_2, \mathbf{r_3}\}, \emptyset, \emptyset \rangle \\
\mathbb{K}_3 & = \mathbb{K}_2
\end{array}$$

Ici, $\Delta_{pro}(P_{33} \cup K_3, I_3, R_3^{app}) = \{(\bot \leftarrow a, e.)\} \neq \emptyset$ et $\forall r \in \Delta_{pro}(P_{33} \cup K_3, I_3, R_3^{app}), head(r) \in OUT_3$ donc le préfixe est bloqué par (Propagation-failure) et sa raison d'échec est $\{r_0, r_1, r_3\}$. Si on revient sur le dernier (Choix de règle) et que l'on applique une (Exclusion de règle) à la place on obtient :

$$\begin{array}{ll} & \text{(Exclusion de règle)} \\ r_{3'} & = e \leftarrow not\ f. \in \Delta_{cho_mbt}(P_{33}, I_2, R_2^{app} \cup R_2^{excl}) \\ \mathbb{I}_{3'} & = \langle\{(a, \{r_1\}), (c, \{r_2\})\}, \{(\mathbf{f}, \{\mathbf{r_{3'}}\})\}, \{(\bot, \{r_0\}), (b, \{r_1\}), (d, \{r_2\})\}\rangle \\ R_{3'} & = \langle\{r_1, r_2\}, \emptyset, \{\mathbf{r_{3'}}\}\rangle \\ \mathbb{K}_{3'} & = \mathbb{K}_2 \end{array}$$

Ici, $\Delta_{pro_mbt}(P_{33} \cup K_3, I_3, R_3^{app} \cup R_3^{mbt}) = \{(\bot \leftarrow a, f.)\} \neq \emptyset$ et $\forall r \in \Delta_{pro_mbt}(P_{33} \cup K_3, I_3, R_3^{app} \cup R_3^{mbt}), head(r) \in OUT_3$ donc le préfixe est bloqué par (Mbt-propagation-failure) et sa raison d'échec est $\{r_0, r_1, r_3\}$.

En combinant les échecs, on observe que le (Choix de règle) $r_1$ : $(a \leftarrow not\ b.)$ est la dernière règle responsable des échecs en $r_3$ et $r_{3'}$. En effet, l'échec du (Choix de règle) $r_3$ et de l'(Exclusion de règle) $r_{3'}$ sont liés au fait que $a$ a été ajouté à l'ensemble $IN$ par le (Choix de règle) $r_1$. Le (Choix de règle) $r_2$ est totalement indépendant de ces deux échecs. On est donc assuré qu'aucune computation justifiée de préfixe $S_{i=0}^1$ ne converge.

La définition suivante permet d'extraire les règles de *point de choix* d'une computation justifiée. Ces règles correspondent aux nœuds de l'arbre de recherche d'`ASPeRiX`. Dans une computation, il s'agit simplement des étapes de (Choix de règle) ou d'(Exclusion de règle).

**Définition 30.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$. Soit $Reason$ un ensemble de règles numérotées :

- $ruleChoicePoints(S) = \{r_i \in (R_i^{app} \cup R_i^{excl}) \mid i \in [1 \ldots n], \Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app}) = \emptyset, \Delta_{pro_mbt}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{mbt}) = \emptyset\}$ est l'ensemble des règles utilisées lors des points de choix de $S$.
- $ruleChoicePoints(Reason, S) = reason \cap ruleChoicePoints(S)$ est l'ensemble des règles de $Reason$ utilisées lors des points de choix de $S$.
- $lastRuleChoicePoint(Reason, S) = max(ruleChoicePoints(Reason, S))$ est la dernière règle de $Reason$ utilisée lors d'un point de choix de $S$ (la règle de plus grand numéro).

La définition suivante introduit la notion de *préfixe échec* de computation justifiée qui correspond à un préfixe qui ne peut aboutir à un answer set (aucune convergence possible avec ce préfixe). Ce préfixe correspond soit à un échec d'une branche de l'arbre de recherche dans le cas où il s'agit d'un préfixe bloqué, soit à la combinaison des échecs des branches gauche et droite de l'arbre.

**Définition 31** (Préfixe échec)**.** Soit $P$ un programme logique au premier ordre. On définit un *préfixe échec* de computation justifiée pour $P$ et une raison d'échec associée à ce préfixe de la manière suivante :

(1) un préfixe bloqué est un préfixe échec avec une raison d'échec telle que définie dans la définition 29.

(2) Soient $S_1 = \langle \mathbb{K}_{1i}, R_{1i}, \mathbb{I}_{1i} \rangle_{i=0}^{n}$ et $S_2 = \langle \mathbb{K}_{2i}, R_{2i}, \mathbb{I}_{2i} \rangle_{i=0}^{m}$ deux préfixes échec de computation justifiée pour $P$ avec $R_{1i} = \langle {R_1}_i^{app}, {R_1}_i^{mbt}, {R_1}_i^{excl} \rangle$ et $R_{2i} = \langle {R_2}_i^{app}, {R_2}_i^{mbt}, {R_2}_i^{excl} \rangle$ et soient $RF_1$ et $RF_2$ deux raisons d'échec respectivement pour $S_1$ et pour $S_2$ tels que :

- $lastRuleChoicePoint(RF_1, S_1) = lastRuleChoicePoint(RF_2, S_2) = r_{lc}$
- $\langle S_{1i} \rangle_{i=0}^{lc-1} = \langle S_{2i} \rangle_{i=0}^{lc-1}$
- Pour $S1$, $r_{lc} \in {R_1}_{lc}^{app}$ et pour $S2$, $r_{lc} \in {R_2}_{lc}^{excl}$

Soit $RaisonEchec = RF_{1<r_{lc}} \cup RF_{2<r_{lc}}$ et soit $r_k = max(RaisonEchec)$.
$\langle S_{1i} \rangle_{i=0}^{k} = \langle S_{2i} \rangle_{i=0}^{k}$ est un préfixe échec et $RaisonEchec$ est une *raison d'échec* pour ce préfixe.

**Définition 32.** Soit $P$ un programme logique au premier ordre. On définit le *nombre de combinaisons d'échecs* d'un préfixe échec $S$ de computation justifiée pour $P$ de la manière suivante :

(1) si $S$ est un préfixe bloqué alors le nombre de combinaisons d'échecs de $S$ vaut $0$.

(2) si $S$ est obtenu à partir de la combinaison d'échecs de deux préfixes échecs de computations justifiées $S_1$ et $S_2$ ayant respectivement un nombre de combinaisons d'échecs $c_1$ et $c_2$ alors le nombre de combinaisons d'échecs de $S$ vaut $c_1 + c_2 + 1$.

## 3.3 Propriétés des computations

Précédemment, des raisons, matérialisées par des ensembles de règles, ont été construites afin de justifier certaines propriétés notamment la raison pour laquelle un littéral est présent dans $IN$, la raison pour laquelle un littéral est indéterminé ou bien encore la raison pour laquelle une computation ne converge pas. Dans cette partie, on montrera que les raisons calculées auparavant sont de véritables raisons de la satisfaction d'une propriété de computation. Cela signifie que chaque answer set compatible avec une raison satisfait la propriété.

Le théorème suivant montre que l'ensemble des règles d'une raison d'indétermination d'un littéral d'une computation justifiée $S$ ne peut aboutir à aucun answer set contenant ce même littéral. Cela signifie qu'il n'existe aucune computation qui converge où les règles de cet ensemble possèdent le même statut et tel que le littéral appartient à l'answer set calculé.

**Théorème 5.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{k}$ un préfixe de computation justifiée pour $P$. Soit $l$ un atome tel que $l \notin (IN_k \cup OUT_k)$ et $R_{ind}$ une raison d'indétermination de $l$. Pour tout answer set $X$ compatible avec $R_{ind}$, $l \notin X$.

Quelques notions sur les *ensembles non fondés* ci-dessous vont permettre de réaliser la preuve du théorème 5.

**Définition 33.** (D'après [53]) Soit $E$ un ensemble d'atomes et $X$ un answer set d'un programme $P$. On dit que $E$ est un *ensemble non fondé* pour $P$ par rapport à $X$ si $\forall r \in ground(P)$ tel que $tête(r) \in E$, au moins une des conditions suivantes est vérifiée :

(1) $corps^+(r) \nsubseteq X$

(2) $corps^-(r) \cap X \neq \emptyset$

(3) $corps^+(r) \cap E \neq \emptyset$

**Définition 34.** (D'après [35]) Soit $X$ un answer set pour un programme $P$, $X$ est *unfounded-free* si $X \cap E = \emptyset$ pour chaque ensemble non fondé $E$ pour $P$ par rapport à $X$.

**Théorème 6.** (D'après [35]) Soit $X$ un ensemble de littéraux pour un programme $P$. Si $X$ est un answer set pour $P$ alors $X$ est unfounded-free.

*Démonstration.* (du théorème 5) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R'_i, \mathbb{I}_i \rangle_{i=0}^{k}$ un préfixe de computation justifiée pour $P$ avec $R'_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soit $l$ un littéral tel que $l \notin (IN_k \cup OUT_k)$. Soit $R_{ind}$ une raison d'indétermination de $l$ définie grâce à la suite $T = \langle Rules_i, Litteraux_i, Reas_i \rangle_{i=0}^{\infty}$. On suppose Il existe un answer set $X$ compatible avec $R_{ind}$ ce qui signifie que :
$$\begin{cases} {R_{ind}}_S^{decl} \subseteq GR_P(X) \\ \forall r \in {R_{ind}}_S^{nbloq}, corps^-(r) \cap X = \emptyset \\ \forall r \in {R_{ind}}_S^{bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$$
On montre dans un premier temps que $Litteraux_\infty$ est un ensemble non fondé pour $P$ par rapport à $X$ (définition 33) ce qui signifie que $\forall r \in ground(P)$ tel que $tête(r) \in Litteraux_\infty$, au moins une des conditions suivantes doit être vérifiée :

(1) $corps^+(r) \nsubseteq X$

(2) $corps^-(r) \cap X \neq \emptyset$

(3) $corps^+(r) \cap Litteraux_\infty \neq \emptyset$.

D'après la définition 24, $Rules_\infty = \bigcup_{a \in Litteraux_\infty} hrule(a, P) = \{r \in ground(P) \mid tête(r) \in Litteraux_\infty\}$, donc $\forall r \in ground(P)$ tel que $tête(r) \in Litteraux_\infty$, $\exists j > 0$ dans la suite $T$ tel que $r = r_j$ et $r_j \in Rules_j \setminus Rules_{j-1}$.
D'après la définition 24, cinq cas sont possibles à l'étape $j$ dans $T$ :

$(i)$ $\exists a \in (corps^+(r_j) \cap OUT_k)$ et $reason(a, \mathbb{I}_k) \subseteq R_{ind}$.
$a \in OUT_k$ et $a \neq \bot$ car $a \in corps^+(r_j)$ donc, d'après le théorème 4, $a$ est obtenu par (Choix de règle) et $\exists i \leq k$ tel que $r_i \in R_i^{app} \setminus R_{i-1}^{app}$, $r_i \in R_k^{app}$, $a \in corps^-(r_i)$ et $max(reason(a, \mathbb{I}_i)) = r_i$. Donc $r_i \in reason(a, \mathbb{I}_i)$.

- Si $reason(tête(r_i), \mathbb{I}_i) \subseteq R_{ind}$ alors $r_i \in {R_{ind}}_S^{decl}$ d'après la définition 25. Par hypothèse, ${R_{ind}}_S^{decl} \subseteq GR_P(X)$. On a donc $r_i \in GR_P(X) = \{r \in ground(P) \mid corps^+(r) \subseteq X$ et $corps^-(r) \cap X = \emptyset\}$. Or, $a \in corps^-(r_i)$ donc $a \notin X$. Par conséquent, $corps^+(r_j) \nsubseteq X$ et la condition (1) de la définition 33 d'un ensemble non fondé est vérifiée.
- Si $reason(tête(r_i), \mathbb{I}_i) \nsubseteq R_{ind}$ alors $r_i \in {R_{ind}}_S^{nbloq}$ d'après la définition 25. Par hypothèse, $\forall r \in {R_{ind}}_S^{nbloq}, corps^-(r) \cap X = \emptyset$. Or, $a \in corps^-(r_i)$ donc $a \notin X$. Par conséquent, $corps^+(r_j) \nsubseteq X$ et la condition (1) de la définition 33 d'un ensemble non fondé est vérifiée.

$(ii)$ $\exists a \in (corps^-(r_j) \cap IN_k)$ et $reason(a, \mathbb{I}_k) \subseteq R_{ind}$.
$a \in IN_k$ donc, d'après le théorème 4, $a$ est obtenu par (Propagation) ou (Choix de règle) et $\exists i \leq k$ tel que $r_i \in R_i^{app} \setminus R_{i-1}^{app}$, $r_i \in R_k^{app}$, $tête(r_i) = a$ et $max(reason(a, \mathbb{I}_i)) = r_i$. Donc $r_i \in reason(a, \mathbb{I}_i)$ et $r_i \in {R_{ind}}_S^{decl}$ d'après la définition 25. Par hypothèse, ${R_{ind}}_S^{decl} \subseteq GR_P(X)$. On a donc $r_i \in GR_P(X)$, et $X = tête(GR_P(X))$ d'après le théorème 1, donc $a \in X$. Par conséquent, $corps^-(r_j) \cap X \neq \emptyset$ et la condition (2) de la définition 33 d'un ensemble non fondé est vérifiée.

$(iii)$ $\exists a \in (corps^-(r_j) \cap MBT_k)$ et $reason(a, \mathbb{I}_k) \subseteq R_{ind}$.
On a $a \in MBT_k$ donc, d'après le théorème 4,

- Si $a$ est obtenu par (Mbt-propagation) dans le préfixe de computation justifiée $S$ alors $\exists i \leq k$ tel que $r_i \in R_i^{mbt} \backslash R_{i-1}^{mbt}$, $r_i \in R_k^{mbt}$, $t\hat{e}te(r_i) = a$ et $max(reason(a, \mathbb{I}_i)) = r_i$. En appliquant le même raisonnement que pour $(ii)$, la condition (2) de la définition 33 d'un ensemble non fondé est vérifiée.
- Si $a$ est obtenu par (Exclusion de règle) dans le préfixe de computation justifiée $S$ alors $\exists i \leq k$ tel que $r_i \in R_i^{excl} \setminus R_{i-1}^{excl}$, $r_i \in R_k^{excl}$, $corps^-(r_i) = \{a\}$ et $max(reason(a, \mathbb{I}_i)) = r_i$. Donc $r_i \in reason(a, \mathbb{I}_i)$ et $r_i \in {R_{ind}}_S^{bloq}$ d'après la définition 25. Par hypothèse, $\forall r \in {R_{ind}}_S^{bloq}, corps^-(r) \cap X \neq \emptyset$. Or, $a$ est l'unique littéral du corps négatif de $r_i$ donc $a \in X$. Par conséquent, $corps^-(r_j) \cap X \neq \emptyset$ et la condition (2) de la définition 33 d'un ensemble non fondé est vérifiée.

$(iv)$ $\exists const \in first(\mathbb{K})$ tel que $const = (\perp \leftarrow \cup_{b \in corps^-(r_j)} not\ b.)$, $reason(const, P, \mathbb{K}) = \{r_j\}$ et $reason(const, P, \mathbb{K}) \subseteq R_{ind}$.
Donc, d'après la définition d'une computation justifiée, $const$ obtenu par (Exclusion de règle) et $r_j \in R_k^{excl}$. Donc $r_j \in {R_{ind}}_S^{bloq}$ d'après la définition 25. Par hypothèse, $\forall r \in {R_{ind}}_S^{bloq}$, $corps^-(r) \cap X \neq \emptyset$. Par conséquent, $corps^-(r_j) \cap X \neq \emptyset$ et la condition (2) de la définition 33 d'un ensemble non fondé est vérifiée.

$(v)$ $\exists a \in (corps^+(r_j) \setminus (IN_k \cup OUT_k))$ et $a \in Litteraux_\infty$.
Donc, $corps^+(r_j) \cap Litteraux_\infty \neq \emptyset$ et la condition (3) de la définition 33 d'un ensemble non fondé est vérifiée.

Donc $Litteraux_\infty$ est un ensemble non fondé.

Par hypothèse, $X$ est un answer set, donc, d'après le théorème 6, $X \cap Litteraux_\infty = \emptyset$. Or, d'après la définition 24, $l \in Litteraux_0$ et $Litteraux_0 \subseteq Litteraux_\infty$. Donc, $l \notin X$. Par conséquent, le théorème 5 est vérifié.

□

Le lemme suivant fournit quelques propriétés des answer sets compatibles avec une raison d'échec qui seront utiles par la suite.

**Lemme 3.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^k$ un préfixe échec de computation justifiée pour $P$ avec $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soient $RF$ une raison d'échec pour $S$ et $a$ un littéral de $P$.
S'il existe un answer set X compatible avec $RF$ alors :

- si $a \in IN_k$ et $reason(a, \mathbb{I}_k) \subseteq RF$ alors $a \in X$.
- si $a \in MBT_k$ et $reason(a, \mathbb{I}_k) \subseteq RF$ alors $a \in X$.
- si $a \in OUT_k$ et $reason(a, \mathbb{I}_k) \subseteq RF$ alors $a \notin X$.
- si $a \notin IN_k \cup OUT_k$ et $reason_{ind}(a, \mathbb{I}_k, P, \mathbb{K}_k) \subseteq RF$ alors $a \notin X$.

*Démonstration.* (du lemme 3) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^k$ un préfixe échec de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soient $RF$ une raison d'échec pour $S$ et $a$ un littéral de $P$.
$(A)$ Soit $X$ un answer set compatible avec $RF$,

ce qui signifie que : $\begin{cases} RF_S^{decl} \subseteq GR_P(X) \\ \forall r \in RF_S^{nbloq}, corps^-(r) \cap X = \emptyset \\ \forall r \in RF_S^{bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$.

- On suppose $a \in IN_k$ et $reason(a, \mathbb{I}_k) \subseteq RF$ et on montre que $a \in X$.
  Si $a \in IN_k$ alors, d'après le théorème 4, $a$ est obtenu par (Propagation) ou (Choix de règle) à une étape $i \leq k$ : $\exists i \leq k$ tel que $r_i \in R_i^{app} \setminus R_{i-1}^{app}$, $r_i \in R_k^{app}$, $tête(r_i) = a$, $max(reason(a, \mathbb{I}_i)) = r_i$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$. Donc, comme $reason(a, \mathbb{I}_k) \subseteq RF$, $r_i \in RF_S^{decl}$ d'après la définition 25. D'après $(A)$, $r_i \in GR_P(X)$, et $X = tête(GR_P(X))$ d'après le théorème 1, donc $a \in X$.

- On suppose $a \in MBT_k$ et $reason(a, \mathbb{I}_k) \subseteq RF$ et on montre que $a \in X$.
  Si $a \in MBT_k$ alors, d'après le théorème 4 :

  $a$ est obtenu par (Mbt-propagation) à une étape $i \leq k$ : $\exists i \leq k$ tel que $r_i \in R_i^{mbt} \setminus R_{i-1}^{mbt}$, $r_i \in R_k^{mbt}$, $tête(r_i) = a$, $max(reason(a, \mathbb{I}_i)) = r_i$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$. En appliquant le même raisonnement que précédemment, $a \in X$.

  ou $a$ obtenu par (Exclusion de règle) à une étape $i \leq k$ : $r_i \in R_i^{excl} \setminus R_{i-1}^{excl}$, $r_i \in R_k^{excl}$, $corps^-(r_i) = \{a\}$, $max(reason(a, \mathbb{I}_i)) = r_i$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$. Donc, comme $reason(a, \mathbb{I}_k) \subseteq RF$, $r_i \in RF_S^{bloq}$ d'après la définition 25. D'après $(A)$, $\forall r \in RF_S^{bloq}, corps^-(r) \cap X \neq \emptyset$. Comme $a$ est l'unique littéral appartenant au corps négatif de $r_i$, $a \in X$.

- On suppose $a \in OUT_k$ et $reason(a, \mathbb{I}_k) \subseteq RF$ et on montre que $a \notin X$.
  Si $a \in OUT_k$ alors, d'après le théorème 4,

  $a = \perp$ donc $a \notin X$.

  ou $a$ est obtenu par (Choix de règle) à une étape $i \leq k$ : $\exists i \leq k$ tel que $r_i \in R_i^{app} \setminus R_{i-1}^{app}$, $r_i \in R_k^{app}$, $a \in corps^-(r_i)$, $max(reason(a, \mathbb{I}_i)) = r_i$ et $reason(a, \mathbb{I}_i) = reason(a, \mathbb{I}_j)$ $\forall j$ tel que $j \geq i$ et $j \leq n$. Donc, comme $reason(a, \mathbb{I}_k) \subseteq RF$, $r_i \in RF_S^{decl} \cup RF_S^{nbloq}$ d'après la définition 25. D'après $(A)$, $\forall r \in RF_S^{decl}, r \in GR_P(X) = \{r' \in P \mid corps^+(r') \subseteq X \text{ and } corps^-(r') \cap X = \emptyset\}$ et $\forall r \in RF_S^{nbloq}, corps^-(r) \cap X = \emptyset$ donc, dans les deux cas, $corps^-(r_i) \cap X = \emptyset$ et $a \notin X$.

- On suppose $a \notin IN_k \cup OUT_k$ et $reason_{ind}(a, \mathbb{I}_k, P, \mathbb{K}_k) \subseteq RF$ et on montre que $a \notin X$.
  Si $reason_{ind}(a, \mathbb{I}_k, P, \mathbb{K}_k) \subseteq RF$ alors par $(A)$ et par le lemme 2,
  $\begin{cases} reason_{ind}(a, \mathbb{I}_k, P, \mathbb{K}_k)_S^{decl} \subseteq RF_S^{decl} \subseteq GR_P(X) \\ \forall r \in reason_{ind}(a, \mathbb{I}_k, P, \mathbb{K}_k)_S^{nbloq}, r \in RF_S^{decl} \cup RF_S^{nbloq} \text{ donc } corps^-(r) \cap X = \emptyset \\ \forall r \in reason_{ind}(a, \mathbb{I}_k, P, \mathbb{K}_k)_S^{bloq}, r \in RF_S^{bloq} \text{ donc } corps^-(r) \cap X \neq \emptyset \end{cases}$.
  Donc, d'après le théorème 5, $a \notin X$

□

Lorsqu'un préfixe de computation justifiée $S$ d'un programme $P$ est bloqué, les règles présentes dans la raison d'échec $RF$ forment un ensemble de blocage. Il est ainsi impossible de trouver une computation qui converge où les règles de $RF$ ont le même statut que dans $S$. C'est ce qu'énonce le théorème suivant :

**Théorème 7.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{k}$ un préfixe bloqué de computation justifiée pour $P$. Soit $RF$ une raison d'échec de $S$.
$RF$ est un ensemble de blocage.

*Démonstration.* (du théorème 7) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{k}$ un préfixe bloqué de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soit $RF$ une raison d'échec de $S$.
On suppose $\exists$ answer set $X$ compatible avec $RF$,
ce qui signifie que : $\begin{cases} RF_S^{decl} \subseteq GR_P(X) \\ \forall r \in RF_S^{nbloq}, corps^-(r) \cap X = \emptyset \\ \forall r \in RF_S^{bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$
et on démontre que ce n'est pas possible.

D'après les définitions 28 d'un préfixe bloqué et 29 d'une raison d'échec d'un préfixe bloqué, on a quatre possibilités :

- Si $S$ vérifie (Propagation-Failure)
  $RF = \bigcup_{l \in corps(rf)} reason(l, \mathbb{I}_k) \cup reason(rf, P, \mathbb{K}_k) \cup reason(head(rf), \mathbb{I}_k)$
  avec $rf \in \Delta_{pro}(P \cup K_k, I_k, R_k^{app})$ et $tête(rf) \in OUT_k$. On a donc $corps^+(rf) \subseteq IN_k$ et $corps^-(rf) \subseteq OUT_k$ d'après la définition 16. On montre dans un premier temps que $rf \in GR_P(X)$.

  - $\forall a \in corps^+(rf)$, $a \in IN_k$. De plus, $reason(a, \mathbb{I}_k) \subseteq RF$ donc, d'après le lemme 3, $a \in X$.
  - $\forall b \in corps^-(rf), b \in OUT_k$. De plus, $reason(b, \mathbb{I}_k) \subseteq RF$ donc, d'après le lemme 3, $b \notin X$.

  Donc $rf \in GR_P(X) = \{r \in ground(P) \mid corps^+(r) \subseteq X$ et $corps^-(r) \cap X = \emptyset\}$ et $tête(rf) \in X$ (car $X = tête(GR_P(X))$ d'après le théorème 1).

  Par ailleurs, $tête(rf) \in OUT_k$ et $reason(head(rf), \mathbb{I}_k) \subseteq RF$, donc, d'après le lemme 3, $tête(rf) \notin X$.

  Une contradiction est détectée ($tête(rf) \in X$ et $tête(rf) \notin X$).

- Si $S$ vérifie (Mbt-Propagation-Failure)
  $RF = \bigcup_{l \in corps(rf)} reason(l, \mathbb{I}_k) \cup reason(rf, P, \mathbb{K}_k) \cup reason(head(rf), \mathbb{I}_k)$
  avec $rf \in \Delta_{pro_mbt}(P \cup K_k, I_k, R_k^{app} \cup R_k^{mbt}$ et $tête(rf) \in OUT_k$. On a donc $corps^+(rf) \subseteq (IN_k \cup MBT_k)$, $corps^+(rf) \nsubseteq IN_k$ et $corps^-(rf) \subseteq OUT_k$ d'après la définition 16.

  Par le même raisonnement que pour (Propagation-failure), $rf \in GR_P(X)$ et $tête(rf) \in X$. De plus, $tête(rf) \in OUT_k$ donc, par le même raisonnement que pour (Propagation-failure), $tête(rf) \notin X$.

  Une contradiction est détectée ($tête(rf) \in X$ et $tête(rf) \notin X$).

- Si $S$ vérifie ($\Delta cho_mbt$ Convergence-failure)
  $RF = \bigcup_{l \in (corps^-(rf) \cap OUT_k)} reason(l, \mathbb{I}_k) \cup$
  $\bigcup_{l \in (corps^-(rf) \setminus OUT_k)} reason_{ind}(l, \mathbb{I}_k, P, \mathbb{K}_k) \cup reason(rf, P, \mathbb{K}_k)$
  avec $rf \in \Delta_{cho_mbt}(K_k, I_k, R_k^{app} \cup R_k^{excl})$ . On a donc $corps^-(rf) \cap (IN_k \cup MBT_k) = \emptyset$ d'après la définition 16. On montre dans un premier temps que $corps^-(rf) \cap X = \emptyset$ :

  - $\forall b \in (corps^-(rf) \cap OUT_k), b \in OUT_k$. De plus, $reason(b, \mathbb{I}_k) \subseteq RF$ donc, d'après le lemme 3, $b \notin X$.

– $\forall b \in (corps^-(rf) \setminus OUT_k)$, $b \notin IN_k$ car $corps^-(rf) \cap (IN_k \cup MBT_k) = \emptyset$. De plus, $reason_{ind}(b, \mathbb{I}_k, P, \mathbb{K}_k) \subseteq RF$ donc, d'après le lemme 3, $b \notin X$.

Par ailleurs, $rf \in K_k$ donc, d'après la définition d'une computation justifiée, $rf$ est obtenu par (Exclusion de règle) à une étape $i \leq k$ : $\exists i \leq k$ tel que $corps^-(r_i) = corps^-(rf)$, $reason(rf, P, \mathbb{K}_i) = \{r_i\}$ et $r_i \in R_i^{excl}$. Or, d'après la définition d'une computation justifiée, $R^{excl}$ est croissant donc $r_i \in R_k^{excl}$. De plus, d'après la définition d'une computation justifiée, $\mathbb{K}$ est croissant et monotone donc $reason(rf, P, \mathbb{K}_i) = reason(rf, P, \mathbb{K}_k)$. Or, $reason(rf, P, \mathbb{K}_k) \subseteq RF$ donc $r_i \in RF_S^{bloq}$ d'après la définition 25. Par hypothèse, $\forall r \in RF_S^{bloq}, corps^-(r) \cap X \neq \emptyset$. Une contradiction est détectée.

- Si $S$ vérifie (Mbt-Convergence-Failure)
  $RF = reason_{ind}(l, \mathbb{I}_k, P, \mathbb{K}_k) \cup reason(l, \mathbb{I}_k)$
  avec $l \in MBT_k$.

  On a $l \in MBT_k$. De plus, $reason(l, \mathbb{I}_k) \subseteq RF$ donc, d'après le lemme 3, $l \in X$.

  Par ailleurs, $reason_{ind}(l, \mathbb{I}_k, P, \mathbb{K}_k) \subseteq RF$ et $l \notin IN_k \cup OUT_k$ donc, d'après le lemme 3, $l \notin X$ et une contradiction est détectée.

□

Le lemme suivant expose des propriétés des règles d'une raison d'échec ayant été déclenchées par (Propagation) ou (Mbt-propagation) lors d'une computation justifiée.

**Lemme 4.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe échec de computation justifiée pour $P$. Soit $RF$ une raison d'échec pour $S$. Si $r_i \in RF$ (avec $r_i \neq r_0$) et l'étape $i$ de $S$ est (Propagation) ou (Mbt-propagation) alors $r_i \in reason(tête(r_i), \mathbb{I}_i)$ et $reason(tête(r_i), \mathbb{I}_i) \subseteq RF$.

*Démonstration.* (du lemme 4) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe échec de computation justifiée pour $P$ avec $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soit $RF$ une raison d'échec pour $S$.
Par définition d'une computation justifiée, les raisons sont construites à chaque étape $i$ (pour $i \neq 0$) en utilisant des raisons déjà construites à l'étape $i-1$ et la règle courante $r_i$. De plus, les raisons d'échecs d'un préfixe de computation justifiée et les raisons d'indétermination d'un littéral sont construites en utilisant uniquement des raisons déjà construites auparavant. Donc :

(1) si une règle $r_i$ n'est pas utilisée à l'étape $i$ pour construire une raison alors elle ne sera jamais utilisée.

(2) si une règle $r_i$ est utilisée à l'étape $i$ pour construire une seule raison $R$ alors toute raison contenant $r_i$ contiendra $R$.

(3) Si l'étape $i$ de $S$ est (Propagation) alors :

  $r_i \in reason(tête(r_i), \mathbb{I}_i)$ si $tête(r_i) \notin IN_{i-1}$, par définition d'une computation justifiée.

  ou $r_i$ n'apparaît dans aucune raison, par (1)[2].

(4) Si l'étape $i$ de $S$ est (Mbt-propagation) alors :

[2]il s'agit bien ici de la règle de numéro $i$. La même règle avec un autre numéro peut éventuellement appartenir à une raison

$r_i \in reason(tête(r_i), \mathbb{I}_i)$ si $tête(r_i) \notin IN_{i-1} \cup MBT_{i-1}$, par définition d'une computation justifiée.

ou $r_i$ n'apparaît dans aucune raison, par (1).

Ainsi, si $r_i \in RF$ (avec $r_i \neq r_0$) et l'étape $i$ de $S$ est (Propagation) ou (Mbt-propagation) alors, par (3) et (4), $r_i \in reason(tête(r_i), \mathbb{I}_i)$ et, par (2), $reason(tête(r_i), \mathbb{I}_i) \subseteq RF$. □

Lorsqu'un préfixe de computation justifiée ne peut aboutir à un answer set, le préfixe s'arrêtant au dernier point de choix ne peut pas non plus aboutir à un answer set. En effet, toutes les propagations réalisées à la suite de ce point de choix sont directement liées à ce dernier.

**Exemple 34.** Soit $P_{34}$ le programme suivant :

$$\left\{ \begin{array}{ll} a \leftarrow not\ b. & d \leftarrow a, not\ c. \\ b \leftarrow not\ a. & e \leftarrow not\ d. \\ c \leftarrow a, not\ d. & \bot \leftarrow a, e. \end{array} \right\}$$

La suite suivante $\langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{3}$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$ est un préfixe de computation justifiée pour $P_{34}$ :

$$\begin{array}{ll}
\mathbb{I}_0 & = \langle \emptyset, \emptyset, \{(\bot, \{r_0\})\} \rangle \\
R_0 & = \langle \emptyset, \emptyset, \emptyset \rangle \\
\mathbb{K}_0 & = \emptyset \\
 & \text{(Choix de règle)} \\
r_1 & = a \leftarrow not\ b. \in \Delta_{cho_mbt}(P_{34}, I_0, R_0^{app} \cup R_0^{excl}) \\
\mathbb{I}_1 & = \langle \{(\mathbf{a}, \{\mathbf{r_1}\})\}, \emptyset, \{(\bot, \{r_0\}), (\mathbf{b}, \{\mathbf{r_1}\})\} \rangle \\
R_1 & = \langle \{\mathbf{r_1}\}, \emptyset, \emptyset \rangle \\
\mathbb{K}_1 & = \mathbb{K}_0 \\
 & \text{(Choix de règle)} \\
r_2 & = c \leftarrow a, not\ d. \in \Delta_{cho_mbt}(P_{34}, I_1, R_1^{app} \cup R_1^{excl}) \\
\mathbb{I}_2 & = \langle \{(a, \{r_1\}), (\mathbf{c}, \{\mathbf{r_1}, \mathbf{r_2}\})\}, \emptyset, \{(\bot, \{r_0\}), (b, \{r_1\}), (\mathbf{d}, \{\mathbf{r_2}\})\} \rangle \\
R_2 & = \langle \{r_1, \mathbf{r_2}\}, \emptyset, \emptyset \rangle \\
\mathbb{K}_2 & = \mathbb{K}_1 \\
 & \text{(Propagation)} \\
r_3 & = e \leftarrow not\ d. \in \Delta_{pro}(P_{34}, I_2, R_2^{app} \cup R_2^{excl}) \\
\mathbb{I}_3 & = \langle \{(a, \{r_1\}), (c, \{r_1, r_2\}), (\mathbf{e}, \{\mathbf{r_2}, \mathbf{r_3}\})\}, \emptyset, \{(\bot, \{r_0\}), (b, \{r_1\}), (d, \{r_2\})\} \rangle \\
R_3 & = \langle \{r_1, r_2, \mathbf{r_3}\}, \emptyset, \emptyset \rangle \\
\mathbb{K}_3 & = \mathbb{K}_2
\end{array}$$

Ici, $\Delta_{pro}(P_{34} \cup K_3, I_3, R_3^{app}) = \{(\bot \leftarrow a, e.)\} \neq \emptyset$ et $\forall r \in \Delta_{pro}(P_{34} \cup K_3, I_3, R_3^{app}), head(r) \in OUT_3$ donc le préfixe est bloqué par (Propagation-failure) et sa raison d'échec est $\{r_0, r_1, r_2, r_3\}$. Ici, la (Propagation) de $r_3$ est inévitable dès lors qu'on effectue le (Choix de règle) en $r_2$ donc l'échec est déjà avéré au moment où on effectue le (Choix de règle) en $r_2$.

Le théorème suivant exprime cette caractéristique.

**Théorème 8.** Soit $P$ un programme logique au premier ordre. Soient $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{n}$ un préfixe échec de computation justifiée et $RF$ une raison d'échec pour $S$.
Soit $r_{lc} = lastRuleChoicePoint(RF, S)$. Si $RF$ est un ensemble de blocage alors $RF_{\leq r_{lc}}$ est un ensemble de blocage également.

*Démonstration.* (du théorème 8) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^{n}$ un préfixe échec de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soit $RF$ une raison d'échec pour $S$. Soit $r_{lc} = lastRuleChoicePoint(RF, S)$. On suppose que $RF_{\leq r_{lc}}$ n'est pas un ensemble de blocage :

$(*1)$ il existe un answer set X tel que $\begin{cases} RF_{\leq r_{lc}S}^{\;decl} \subseteq GR_P(X) \\ \forall r \in RF_{\leq r_{lc}S}^{\;nbloq}, corps^-(r) \cap X = \emptyset \\ \forall r \in RF_{\leq r_{lc}S}^{\;bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$

et on prouve que $RF$ n'est pas un ensemble de blocage en montrant que $X$ vérifie

$$\begin{cases} (p_1)\; RF_S^{decl} \subseteq GR_P(X) \\ (p_2)\; \forall r \in RF_S^{nbloq}, corps^-(r) \cap X = \emptyset \\ (p_3)\; \forall r \in RF_S^{bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$$

afin de valider la contraposée du théorème 8.

On a $r_{lc} = lastRuleChoicePoint(RF, S) = max(RF \cap \{r_i \in (R_i^{app} \cup R_i^{excl}) \mid i \in [1 \ldots n], \Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app}) = \Delta_{pro_mbt}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{mbt}) = \emptyset\})$ d'après la définition 30. Donc, $\forall r_j \in RF_{>r_{lc}}$, on a :

$r_j \in R_n^{mbt}$ (car $\forall r \in RF$, $r \in R_n^{app} \cup R_n^{mbt} \cup R_n^{excl}$)

ou $r_j \in \Delta_{pro}(P \cup K_{j-1}, I_{j-1}, R_{j-1}^{app})$

ou $r_j \in \Delta_{pro_mbt}(P \cup K_{j-1}, I_{j-1}, R_{j-1}^{app} \cup R_{j-1}^{mbt})$

Or, d'après la définition 22 d'une computation justifiée, si $r_j \in R_n^{mbt}$ alors l'étape $j$ est une (Mbt-propagation) et $r_j \in \Delta_{pro_mbt}(P, I_{j-1}, R_{j-1}^{app} \cup R_{j-1}^{mbt})$.
Donc :
$(*2)$ $\forall r_j \in RF_{>r_{lc}}, j > 0, r_j \in \Delta_{pro}(P \cup K_{j-1}, I_{j-1}, R_{j-1}^{app}) \cup \Delta_{pro_mbt}(P \cup K_{j-1}, I_{j-1}, R_{j-1}^{app} \cup R_{j-1}^{mbt})$ et l'étape $j$ est (Propagation) ou (Mbt-propagation) dans $S$.

On montre par récurrence sur $j \in [1..n]$ que $\forall r_j \in RF$, $r_j$ satisfait les propriétés $(p_1)$ à $(p_3)$ :

- base : On montre que $\forall r_j \in RF$ tel que $j \leq lc$ les propriétés sont vérifiées. Trois cas sont possibles :

  - $r_j \in RF_S^{decl} = (RF \cap R_n^{mbt}) \cup \{r_i \in RF \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \subseteq RF\}$. Comme $r_j \in RF_{\leq r_{lc}} \subseteq RF$, on a $r_j \in (RF_{\leq r_{lc}} \cap R_n^{mbt}) \cup \{r_i \in RF_{\leq r_{lc}} \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \subseteq RF\}$. Or, $max(reason(head(r_j), \mathbb{I}_j)) \leq r_j$ d'après le lemme 1 et $r_j \leq r_{lc}$ donc $reason(head(r_j), \mathbb{I}_j) \subseteq RF_{\leq r_{lc}}$ et $r_j \in RF_{\leq r_{lc}S}^{\;decl}$ et par $(*1)$ la propriété $(p_1)$ est vérifiée.
  - $r_j \in RF_S^{nbloq} = \{r_i \in RF \cap R_n^{app} \mid reason(head(r_i), \mathbb{I}_i) \nsubseteq RF\}$. Comme $r_j \in RF_{\leq r_{lc}} \subseteq RF$, on a $reason(head(r_j), \mathbb{I}_j) \nsubseteq RF_{\leq r_{lc}}$. Donc $r_j \in RF_{\leq r_{lc}S}^{\;nbloq}$ et par $(*1)$ la propriété $(p_2)$ est vérifiée.
  - $r_j \in RF_S^{bloq} = RF \cap R_n^{excl}$. Comme $r_j \in RF_{\leq r_{lc}} \subseteq RF$, on a $r_j \in RF_{\leq r_{lc}} \cap R_n^{excl}$ donc $r_j \in RF_{\leq r_{lc}S}^{\;bloq}$ et par $(*1)$ la propriété $(p_3)$ est vérifiée.

  Donc, la propriété est vraie jusqu'à $r_{lc}$ : $\forall r_j \in RF$ tel que $j \in [1..lc]$, $r_j$ satisfait les propriétés $(p_1)$ à $(p_3)$.

- On suppose la propriété vraie jusqu'à $r_i \geq r_{lc}$ avec $r_i \in RF\ (HR)$.
  Soit $r_j = min(RF_{>r_i})$. On a $r_j > r_i \geq r_{lc}$ donc, d'après $(*2)$, l'étape $j$ est une (Propagation) ou une (Mbt-propagation) dans $S$ et $r_j \in R_j^{app} \cup R_j^{mbt}$ donc $r_j \in R_n^{app} \cup R_n^{mbt}$ car $R^{app}$ et $R^{mbt}$ sont croissants. De plus, $r_j \in RF$ donc, d'après le lemme 4, $r_j \in reason(tête(r_j), \mathbb{I}_j)$ et $reason(tête(r_j), \mathbb{I}_j) \subseteq RF$. Donc, d'après la définition 25, $r_j \in RF_S^{decl}$.

  Il faut donc montrer que $r_j$ satisfait la propriété $(p_1)$, c'est à dire, $r_j \in GR_P(X) = \{r \in ground(P) \mid corps^+(r) \subseteq X$ et $corps^-(r) \cap X = \emptyset\}$.
  On a $reason(tête(r_j), \mathbb{I}_j) = \bigcup_{l \in corps(r_j)} reason(l, \mathbb{I}_{j-1}) \ \cup\ \{r_j\}$, donc $\forall l \in corps(r_j)$, $reason(l, \mathbb{I}_{j-1}) \subseteq RF$. De plus, $r_j \in \Delta_{pro}(P \cup K_{j-1}, I_{j-1}, R_{j-1}^{app}) \cup \Delta_{pro_mbt}(P \cup K_{j-1}, I_{j-1}, R_{j-1}^{app} \cup R_{j-1}^{mbt})$ donc $corps^+(r_j) \subseteq IN_{j-1} \cup MBT_{j-1}$ et $corps^-(r_j) \subseteq OUT_{j-1}$.

  - $\forall a \in corps^+(r_j)$, $a \in IN_{j-1} \cup MBT_{j-1}$, deux cas sont possibles par définition d'une computation justifiée et d'après le théorème 4 :
    - $a$ est obtenu par (Propagation) ou (Mbt-propagation) : $\exists r_h < r_j$ tel que $tête(r_h) = a$, $r_h \in R_n^{app} \cup R_n^{mbt}$, $max(reason(a, \mathbb{I}_h)) = r_h$ et $reason(a, \mathbb{I}_h) = reason(a, \mathbb{I}_{j-1})$. Or, $reason(a, \mathbb{I}_{j-1}) \subseteq reason(tête(r_j), \mathbb{I}_j) \subseteq RF$. Donc, $r_h \in RF$ et, d'après la définition 25, $r_h \in RF_S^{decl}$. Par ailleurs, $r_h < r_j$ et $r_j = min(RF_{>r_i})$ donc $r_h \leq r_i$. L'hypothèse de récurrence $(HR)$ s'applique donc pour $r_h$. Par conséquent, $r_h \in GR_P(X)$, et $X = tête(GR_P(X))$ d'après le théorème 1, donc $a \in X$.
    - $a$ est obtenu par (Exclusion de règle) : $\exists r_h < r_j$ tel que $corps^-(r_h) = \{a\}$, $r_h \in R_n^{excl}$, $max(reason(a, \mathbb{I}_h)) = r_h$ et $reason(a, \mathbb{I}_h) = reason(a, \mathbb{I}_{j-1})$. Or, $reason(a, \mathbb{I}_{j-1}) \subseteq reason(tête(r_j), \mathbb{I}_j) \subseteq RF$. Donc, $r_h \in RF$ et, d'après la définition 25, $r_h \in RF_S^{bloq}$. Par ailleurs, $r_h < r_j$ et $r_j = min(RF_{>r_i})$ donc $r_h \leq r_i$. L'hypothèse de récurrence $(HR)$ s'applique pour $r_h$. Par conséquent, $corps^-(r_h) \cap X \neq \emptyset$ donc $a \in X$.
  - $\forall b \in corps^-(r_j)$, $b \in OUT_{j-1}$. Donc, par définition d'une computation justifiée et d'après le théorème 4, $b$ est obtenu par (Choix de règle) : $\exists r_h < r_j$ tel que $b \in corps^-(r_h)$, $r_h \in R_n^{app}$, $max(reason(b, \mathbb{I}_h)) = r_h$ et $reason(b, \mathbb{I}_h) = reason(b, \mathbb{I}_{j-1})$. Or, $reason(b, \mathbb{I}_{j-1}) \subseteq reason(tête(r_j), \mathbb{I}_j) \subseteq RF$. Donc $r_h \in RF$ et, d'après la définition 25, $r_h \in RF_S^{decl} \cup RF_S^{nbloq}$. Par ailleurs, $r_h < r_j$ et $r_j = min(RF_{>r_i})$ donc $r_h \leq r_i$. L'hypothèse de récurrence $(HR)$ s'applique pour $r_h$. Par conséquent, $corps^-(r) \cap X = \emptyset$ donc $b \notin X$.

  Donc $r_j \in GR_P(X) = \{r \in ground(P) \mid corps^+(r) \subseteq X$ et $corps^-(r) \cap X = \emptyset\}$ et vérifie $(p_1)$.
  La récurrence est vérifiée.

On a donc il existe un answer set X tel que $\begin{cases} RF_S^{decl} \subseteq GR_P(X) \\ \forall r \in RF_S^{nbloq}, corps^-(r) \cap X = \emptyset \\ \forall r \in RF_S^{bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$.

La contraposée est vérifiée.

□

De manière analogue à un préfixe bloqué de computation justifiée $S$ d'un programme $P$, les règles présentes dans la raison d'échec $RF$ d'un préfixe échec forment un ensemble de blocage.

**Théorème 9.** Soit $P$ un programme logique au premier ordre. Soient $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe échec de computation justifiée et $RF$ une raison d'échec pour $S$.
$RF$ est un ensemble de blocage.

*Démonstration.* (du théorème 9) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe échec de computation justifiée pour $P$ avec $\mathbb{K}_i$ un ensemble de règles justifiées, $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Soit $RF$ une raison d'échec pour $S$. On montre par récurrence sur le nombre de combinaisons d'échecs de $S$ que $RF$ est un ensemble de blocage :

$(A)$ il n'existe pas d'answer set X tel que $\begin{cases} RF_S^{decl} \subseteq GR_P(X) \\ \forall r \in RF_S^{nbloq}, corps^-(r) \cap X = \emptyset \\ \forall r \in RF_S^{bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$ .

Soit $nb_combi(S)$ le nombre de combinaisons d'échecs de $S$.

- Si $nb_combi(S) = 0$ alors $S$ est un préfixe bloqué et d'après le théorème 7, $RF$ est un ensemble de blocage donc la propriété $(A)$ est vérifiée.

- $(HR)$ On suppose que $RF$ est un ensemble de blocage pour tout préfixe échec $S$ tel que le nombre de combinaisons d'échecs est inférieur ou égal à $k$ (avec $k \geq 0$).
  Soit deux préfixes échecs de computations justifiées $S_1 = \langle \mathbb{K}_{1i}, R_{1i}, \mathbb{I}_{1i} \rangle_{i=0}^{m_1}$ et $S_2 = \langle \mathbb{K}_{2i}, R_{2i}, \mathbb{I}_{2i} \rangle_{i=0}^{m_2}$ avec pour raison d'échec $RF_1$ et $RF_2$ tels que :

  - $lastRuleChoicePoint(RF_1, S_1) = lastRuleChoicePoint(RF_2, S_2) = r_{lc}$
  - $\langle S_{1i} \rangle_{i=0}^{lc-1} = \langle S_{2i} \rangle_{i=0}^{lc-1}$
  - Pour $S1$, $r_{lc} \in R_{1lc}^{app}$ et pour $S2$, $r_{lc} \in R_{2lc}^{excl}$
  - $nb_combi(S_1) \leq k$ et $nb_combi(S_2) \leq k$
  - $nb_combi(S_1) + nb_combi(S_2) + 1 = k'$ avec $k' > k$.

  Soit $RF = (RF_1 \cup RF_2)_{<r_{lc}}$ et $r_n = max(RF)$. Alors le préfixe $S$ tel que $\langle S_i \rangle_{i=0}^n = \langle S_{1i} \rangle_{i=0}^n = \langle S_{2i} \rangle_{i=0}^n$ est un préfixe échec de computation justifiée obtenu par combinaison d'échecs de $S_1$ et $S_2$ avec pour raison d'échec $RF$. De plus, on a $nb_combi(S) = k'$.

  On montre que la propriété reste vraie pour $\langle S_i \rangle_{i=0}^n$ avec un nombre de combinaisons $k' > k$ :

  On suppose que $RF$ n'est pas un ensemble de blocage :

  $(H)$ il existe un answer set X tel que $\begin{cases} RF_S^{decl} \subseteq GR_P(X) \\ \forall r \in RF_S^{nbloq}, corps^-(r) \cap X = \emptyset \\ \forall r \in RF_S^{bloq}, corps^-(r) \cap X \neq \emptyset \end{cases}$
  et on montre que ce n'est pas possible.

  - On suppose tout d'abord que $corps^-(r_{lc}) \cap X \neq \emptyset$.
    $RF_2$ est une raison d'échec du préfixe échec $S_2$. Comme $nb_combi(S_2) \leq k$, l'hypothèse de récurrence $(HR)$ s'applique et $RF2$ est un ensemble de blocage, ce qui signifie :

    il n'existe pas d' answer set $X'$ tel que $\begin{cases} RF2_{S_2}^{decl} \subseteq GR_P(X') \\ \forall r \in RF2_{S_2}^{nbloq}, corps^-(r) \cap X' = \emptyset \\ \forall r \in RF2_{S_2}^{bloq}, corps^-(r) \cap X' \neq \emptyset \end{cases}$ .

    Donc, d'après le théorème 8, $RF_{2 \leq r_{lc}}$ est un ensemble de blocage. Ainsi,

$(P1)$ il n'existe pas d'answer set $X'$ tel que $\begin{cases} RF_{2\leq_{r_{lc}}S_2}^{decl} \subseteq GR_P(X') \\ \forall r \in RF_{2\leq_{r_{lc}}S_2}^{nbloq}, corps^-(r) \cap X' = \emptyset \\ \forall r \in RF_{2\leq_{r_{lc}}S_2}^{bloq}, corps^-(r) \cap X' \neq \emptyset \end{cases}$.

De plus, $r_{lc} = lastRuleChoicePoint(RF_2, S_2)$ donc $r_{lc} \in RF_{2\leq_{r_{lc}}}$. Par ailleurs, $r_{lc} \in R_{2lc}^{excl}$. Par conséquent, $r_{lc} \in RF_{2\leq_{r_{lc}}S_2}^{bloq}$ d'après la définition 25 et $r_{lc} \notin RF_{2\leq_{r_{lc}}S_2}^{decl} \cup RF_{2\leq_{r_{lc}}S_2}^{nbloq}$.

Donc, $RF_{2\leq_{r_{lc}}S_2}^{decl} = RF_{2<_{r_{lc}}S_2}^{decl}$, $RF_{2\leq_{r_{lc}}S_2}^{nbloq} = RF_{2<_{r_{lc}}S_2}^{nbloq}$ et $RF_{2\leq_{r_{lc}}S_2}^{bloq} = RF_{2<_{r_{lc}}S_2}^{bloq} \cup \{r_{lc}\}$.

La propriété $(P1)$ peut être réécrite de la manière suivante :

$(P1')$ il n'existe pas d'answer set $X'$ tel que $\begin{cases} RF_{2<_{r_{lc}}S_2}^{decl} \subseteq GR_P(X') \\ \forall r \in RF_{2<_{r_{lc}}S_2}^{nbloq}, corps^-(r) \cap X' = \emptyset \\ \forall r \in RF_{2<_{r_{lc}}S_2}^{bloq}, corps^-(r) \cap X' \neq \emptyset \\ \text{et } corps^-(r_{lc}) \cap X' \neq \emptyset \end{cases}$.

On a $RF_{2<_{r_{lc}}} \subseteq RF$ et d'après le lemme 2 :

* $RF_{2<_{r_{lc}}S_2}^{decl} \subseteq RF_S^{decl}$ donc, d'après l'hypothèse $(H)$, $RF_{2<_{r_{lc}}S_2}^{decl} \subseteq GR_P(X)$.
* $RF_{2<_{r_{lc}}S_2}^{nbloq} \subseteq RF_S^{decl} \cup RF_S^{nbloq}$ donc, d'après l'hypothèse $(H)$, $\forall r \in RF_{2<_{r_{lc}}S_2}^{nbloq}$, $r \in GR_P(X)$ ou $corps^-(r) \cap X = \emptyset$. Comme $GR_P(X) = \{r \in P \mid corps^+(r) \subseteq X$ et $corps^-(r) \cap X = \emptyset\}$ on a $\forall r \in RF_{2<_{r_{lc}}S_2}^{nbloq}$, $corps^-(r) \cap X = \emptyset$.
* $RF_{2<_{r_{lc}}S_2}^{bloq} \subseteq RF_S^{bloq}$ donc, d'après l'hypothèse $(H)$, $\forall r \in RF_{2<_{r_{lc}}S_2}^{bloq}$, $corps^-(r) \cap X \neq \emptyset$.

Donc, d'après la propriété $(P1')$, il n'est pas possible que $corps^-(r_{lc}) \cap X \neq \emptyset$. Une contradiction est détectée.

– On suppose à présent $corps^-(r_{lc}) \cap X = \emptyset$.
$RF_1$ est une raison d'échec du préfixe échec $S_1$. Comme $nb_combi(S_1) \leq k$, l'hypothèse de récurrence $(HR)$ s'applique et $RF1$ est un ensemble de blocage, ce qui signifie :

il n'existe pas d'answer set $X'$ tel que $\begin{cases} RF_{1S_1}^{decl} \subseteq GR_P(X') \\ \forall r \in RF_{1S_1}^{nbloq}, corps^-(r) \cap X' = \emptyset \\ \forall r \in RF_{1S_1}^{bloq}, corps^-(r) \cap X' \neq \emptyset \end{cases}$.

Donc, d'après le théorème 8, $RF_{1\leq_{r_{lc}}}$ est un ensemble de blocage. Ainsi,

$(P2)$ il n'existe pas d'answer set $X'$ tel que $\begin{cases} RF_{1\leq_{r_{lc}}S_1}^{decl} \subseteq GR_P(X') \\ \forall r \in RF_{1\leq_{r_{lc}}S_1}^{nbloq}, corps^-(r) \cap X' = \emptyset \\ \forall r \in RF_{1\leq_{r_{lc}}S_1}^{bloq}, corps^-(r) \cap X' \neq \emptyset \end{cases}$.

On a $RF_{1<_{r_{lc}}} \subseteq RF$ et d'après le lemme 2 :

* $RF_{1<_{r_{lc}}S_1}^{decl} \subseteq RF_S^{decl}$ donc, d'après l'hypothèse $(H)$, $RF_{1<_{r_{lc}}S_1}^{decl} \subseteq GR_P(X)$.
* $RF_{1<_{r_{lc}}S_1}^{nbloq} \subseteq RF_S^{decl} \cup RF_S^{nbloq}$ donc, d'après l'hypothèse $(H)$, $\forall r \in RF_{1<_{r_{lc}}S_1}^{nbloq}$, $r \in GR_P(X)$ ou $corps^-(r) \cap X = \emptyset$. Comme $GR_P(X) = \{r \in P \mid corps^+(r) \subseteq X$ et $corps^-(r) \cap X = \emptyset\}$ on a $\forall r \in RF_{1<_{r_{lc}}S_1}^{nbloq}$, $corps^-(r) \cap X = \emptyset$.
* $RF_{1<_{r_{lc}}S_1}^{bloq} \subseteq RF_S^{bloq}$ donc, d'après l'hypothèse $(H)$, $\forall r \in RF_{1<_{r_{lc}}S_1}^{bloq}$, $corps^-(r) \cap X \neq \emptyset$.

De plus, $r_{lc} = lastRuleChoicePoint(RF_1, S_1)$ donc $r_{lc} \in RF_{1\leq_{r_{lc}}}$. Par ailleurs,

$r_{lc} \in R_{1_{lc}}^{app}$. Par conséquent, $r_{lc} \in RF_{1 \leq_{r_{lc}} S_1}^{decl} \cup RF_{1 \leq_{r_{lc}} S_1}^{nbloq}$ d'après la définition 25 et $r_{lc} \notin RF_{1 \leq_{r_{lc}} S_1}^{bloq}$.

* Si $r_{lc} \in RF_{1 \leq_{r_{lc}} S_1}^{decl}$, alors $RF_{1 \leq_{r_{lc}} S_1}^{decl} = RF_{1 <_{r_{lc}} S_1}^{decl} \cup \{r_{lc}\}$, $RF_{1 \leq_{r_{lc}} S_1}^{nbloq} = RF_{1 <_{r_{lc}} S_1}^{nbloq}$ et $RF_{1 \leq_{r_{lc}} S_1}^{bloq} = RF_{1 <_{r_{lc}} S_1}^{bloq}$. La propriété $(P2)$ peut être réécrite de la manière suivante :

  $(P2')$ il n'existe pas d'answer set $X'$ tel que $\begin{cases} RF_{1 <_{r_{lc}} S_1}^{decl} \subseteq GR_P(X') \\ \forall r \in RF_{1 <_{r_{lc}} S_1}^{nbloq}, corps^-(r) \cap X' = \emptyset \\ \forall r \in RF_{1 <_{r_{lc}} S_1}^{bloq}, corps^-(r) \cap X' \neq \emptyset \\ corps^+(r_{lc}) \subseteq X' \\ \text{et } corps^-(r_{lc}) \cap X' = \emptyset \end{cases}$.

  De plus, comme $r_{lc} \in RF_{1 \leq_{r_{lc}} S_1}^{decl}$ alors $reason(head(r_{lc}), \mathbb{I}_{lc}) \subseteq RF_{1 \leq_{r_{lc}}}$ d'après la définition 25. Or, d'après la définition d'une computation justifiée, $reason(tête(r_{lc}), \mathbb{I}_{lc}) = \bigcup_{l \in corps(r_{lc}) \cap (IN_{lc-1} \cup OUT_{lc-1})} reason(l, \mathbb{I}_{lc-1}) \cup \{r_{lc}\}$. Donc $\forall l \in corps(r_{lc}) \cap (IN_{lc-1} \cup OUT_{lc-1}), reason(l, \mathbb{I}_{lc-1}) \subseteq RF_{1 \leq_{r_{lc}}}$.
  Or, $r_{lc} \in \Delta_{cho_mbt}(P, I_{lc-1}, R_{lc-1}^{app} \cup R_{lc-1}^{excl})$ donc $\forall a \in corps^+(r_{lc})$, $a \in IN_{lc-1}$.
  De plus, d'après le lemme 1, $max(reason(a, \mathbb{I}_{lc-1})) \leq r_{lc-1}$ donc $reason(a, \mathbb{I}_{lc-1}) \subseteq RF_{1 <_{r_{lc}}}$ et, par conséquent, $reason(a, \mathbb{I}_{lc-1}) \subseteq RF$. Donc, d'après le lemme 3, $a \in X$.
  Donc, d'après la propriété $(P2')$, il n'est pas possible que $corps^-(r_{lc}) \cap X = \emptyset$.
  Une contradiction est détectée.

* Si $r_{lc} \in RF_{1 \leq_{r_{lc}} S_1}^{nbloq}$, alors $RF_{1 \leq_{r_{lc}} S_1}^{decl} = RF_{1 <_{r_{lc}} S_1}^{decl}$, $RF_{1 \leq_{r_{lc}} S_1}^{bloq} = RF_{1 <_{r_{lc}} S_1}^{bloq}$ et $RF_{1 \leq_{r_{lc}} S_1}^{nbloq} = RF_{1 <_{r_{lc}} S_1}^{nbloq} \cup \{r_{lc}\}$. La propriété $(P2)$ peut être réécrite de la manière suivante :
  $(P2'')$ il n'existe pas d'answer set $X'$ tel que
  $$\begin{cases} RF_{1 <_{r_{lc}} S_1}^{decl} \subseteq GR_P(X') \\ \forall r \in RF_{1 <_{r_{lc}} S_1}^{nbloq}, corps^-(r) \cap X' = \emptyset \\ \forall r \in RF_{1 <_{r_{lc}} S_1}^{bloq}, corps^-(r) \cap X' \neq \emptyset \\ \text{et } corps^-(r_{lc}) \cap X' = \emptyset \end{cases}.$$
  Donc, d'après la propriété $(P2'')$, il n'est pas possible que $corps^-(r_{lc}) \cap X = \emptyset$.
  Une contradiction est détectée.

Donc il est à la fois impossible que $corps^-(r_{lc}) \cap X \neq \emptyset$ et que $corps^-(r_{lc}) \cap X = \emptyset$. L'hypothèse $(H)$ n'est pas valide et $RF$ est un ensemble de blocage.

□

Le théorème suivant présente quelques propriétés des ensembles $R^{app}$, $R^{mbt}$ et $R^{excl}$ pour une computation qui converge vers un answer set.

**Théorème 10.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ une computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. Si $S$ converge vers $IN_\infty$ alors :

(1) $R_\infty^{app} = GR_P(IN_\infty)$

(2) $R_\infty^{mbt} \subseteq R_\infty^{app}$

(3) $R_\infty^{excl} \cap R_\infty^{app} = \emptyset$

*Démonstration.* (du théorème 10) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ une computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$ et $\mathbb{I}_i = \langle \mathbb{IN}_i, \mathbb{MBT}_i, \mathbb{OUT}_i \rangle$. On suppose que $S$ converge vers $IN_\infty$ et on montre les propriétés (1) à (3) :

(1) Cette propriété à déjà été démontrée dans la preuve du lemme 3 de [31].

(2) On a $S$ qui converge vers $IN_\infty$ donc, d'après la définition d'une computation justifiée, $\exists i \geq 0,\ \Delta_{cho_mbt}(P \cup K_i, I_i, R_i^{app} \cup R_i^{excl}) = \emptyset$ et $MBT_i = \emptyset$.

Soit $g \leq i$, on a $\forall a \in MBT_g$, $a \notin MBT_i$. Donc, par définition d'une computation justifiée, $\exists r_h$ tel que $g < h \leq i$ et l'étape $h$ est une (Propagation) ou (Choix de règle) avec $head(r_h) = a$ et $a \in IN_h$. Comme IN est croissant, $a \in IN_i$.
Par conséquent :
$(A)\ \forall g \leq i, MBT_g \subseteq IN_i$.

Soit $r_g \in R_\infty^{mbt}$, $r_g \in \Delta_{pro_mbt}(P, I_{g-1}, R_{g-1}^{app} \cup R_{g-1}^{mbt})$ donc $corps^+(r_g) \subseteq IN_{g-1} \cup MBT_{g-1}$. Donc, comme $IN$ est croissant et $(A)$ on a $corps^+(r_g) \subseteq IN_i$. De même, $corps^-(r_g) \subseteq OUT_{g-1} \subseteq OUT_i$ car $OUT$ est croissant. Donc $corps^-(r_g) \cap IN_i = \emptyset$. Donc comme $S$ converge, on a $IN_i = IN_\infty$ et $r_g \in GR_P(IN_\infty)$. Donc $r_g \in R_\infty^{app}$ d'après (1).

(3) On a $S$ qui converge vers $IN_\infty$ donc, d'après la définition d'une computation justifiée, $\exists i \geq 0,\ \Delta_{cho_mbt}(P \cup K_i, I_i, R_i^{app} \cup R_i^{excl}) = \emptyset$ et $MBT_i = \emptyset$.

Soit $r_g \in R_\infty^{excl}$, $r_g \in \Delta_{cho_mbt}(P, I_{g-1}, R_{g-1}^{app} \cup R_{g-1}^{excl})$ donc $corps^+(r_g) \subseteq IN_{g-1} \subseteq IN_\infty$ car $IN$ est croissant. Deux cas sont possibles d'après la définition d'une computation justifiée :

- $|corps^-(r_g)| = 1$, $l \in corps^-(r_g)$ et $l \in MBT_g$. D'après la propriété $(A)$, $l \in IN_i = IN_\infty$. Donc $r_g \notin GR_P(In_\infty) = R_\infty^{app}$.
- $|corps^-(r_g)| > 1$, $\exists c = (\perp \leftarrow \bigcup_{b \in corps^-(r_g)} not\ b.) \in K_g$. De plus, $K_g \subseteq K_i$ car $K$ est croissant et $\Delta_{cho_mbt}(P \cup K_i, I_i, R_i^{app} \cup R_i^{excl}) = \emptyset$. Donc $c \notin \Delta_{cho_mbt}(P \cup K_i, I_i, R_i^{app} \cup R_i^{excl})$ et $corps^-(r_g) \cap (IN_i \cup MBT_i) \neq \emptyset$. Or, $IN_i \cup MBT_i = IN_i = IN_\infty$ donc $corps^-(r_g) \cap IN_\infty \neq \emptyset$. Donc $r_g \notin GR_P(IN_\infty) = R_\infty^{app}$.

□

Un préfixe échec de computation justifiée ne peut aboutir à un answer set comme le montre le théorème suivant.

**Théorème 11.** Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe échec de computation justifiée pour $P$. Alors, pour toute computation justifiée $S' = \langle \mathbb{K}'_i, R'_i, \mathbb{I}'_i \rangle_{i=0}^\infty$ de préfixe $S$, $S'$ ne converge pas.

*Démonstration.* (du théorème 11) Soit $P$ un programme logique au premier ordre. Soit $S = \langle \mathbb{K}_i, R_i, \mathbb{I}_i \rangle_{i=0}^n$ un préfixe échec de computation justifiée pour $P$ avec $R_i = \langle R_i^{app}, R_i^{mbt}, R_i^{excl} \rangle$. Soit $RF$ une raison d'échec de $S$. Soit $S' = \langle \mathbb{K}'_i, R'_i, \mathbb{I}'_i \rangle_{i=0}^\infty$ avec $R'_i = \langle R'^{app}_i, R'^{mbt}_i, R'^{excl}_i \rangle$ et $\mathbb{I}'_i = \langle \mathbb{IN}'_i, \mathbb{MBT}'_i, \mathbb{OUT}'_i \rangle$ une computation justifiée pour $P$ de préfixe $S$ ,ce qui signifie que $\langle S_i \rangle_{i=0}^n = \langle S'_i \rangle_{i=0}^n$.
D'après la définition 25, $\forall r \in RF$,

- si $r \in RF_S^{decl}$ alors $r \in R_n^{app} \cup R_n^{mbt}$

- si $r \in RF_S^{nbloq}$ alors $r \in R_n^{app}$
- si $r \in RF_S^{bloq}$ alors $r \in R_n^{excl}$

Si $S'$ converge alors $IN'_\infty$ est un answer set. Or,

- $\forall r \in RF_S^{decl}$, $r \in R_n^{app} \cup R_n^{mbt}$ donc $r \in R'^{app}_n \cup R'^{mbt}_n$. Or, $R'^{app}$ et $R'^{mbt}$ sont croissants donc $R'^{app}_n \cup R'^{mbt}_n \subseteq R'^{app}_\infty \cup R'^{mbt}_\infty$. Par conséquent, $r \in R'^{app}_\infty$ par la propriété (2) du théorème 10 et $r \in GR_P(IN'_\infty)$ par la propriété (1) du théorème 10.
- $\forall r \in RF_S^{nbloq}$, $r \in R_n^{app}$ donc $r \in R'^{app}_n$. Or, $R'^{app}$ est croissant donc $R'^{app}_n \subseteq R'^{app}_\infty$. Par conséquent, $r \in GR_P(IN'_\infty)$ par la propriété (1) du théorème 10 et donc $corps^-(r) \cap IN'_\infty = \emptyset$.
- $\forall r \in RF_S^{bloq}$, $r \in R_n^{excl}$ donc $r \in R'^{excl}_n$. Or, $R'^{excl}$ est croissant donc $R'^{excl}_n \subseteq R'^{excl}_\infty$. Par conséquent, $r \notin R'^{app}_\infty$ par la propriété (3) du théorème 10 et $r \notin GR_P(IN'_\infty)$ par la propriété (1) du théorème 10. Or, $r \in R'^{excl}_n$ donc, d'après la définition d'une computation justifiée, $\exists j \leq n$ tel que $r \in \Delta_{cho_mbt}(P, I_{j-1}, R_{j-1}^{app} \cup R_{j-1}^{excl})$ et $r \in R'^{excl}_j$. Donc $corps^+(r) \subseteq IN'_{j-1} \subseteq IN'_\infty$. Or, $r \notin GR_P(IN'_\infty)$ donc $corps^-(r) \cap IN'_\infty \neq \emptyset$.

D'après le théorème 9, $IN'_\infty$ n'est pas un answer set. Donc $S'$ ne converge pas.

□

Dans ce chapitre, des justifications de computations pour une approche basée sur les règles ont été abordées. Des raisons ont été construites pour exprimer certaines propriétés des computations. Des théorèmes ont ensuite permis de valider le fait que les raisons calculées sont de véritables raisons qui justifient les propriétés. Par conséquent, les justifications de computations peuvent être utilisées pour diverses applications notamment de l'apprentissage et du débogage.

Dans le chapitre suivant, l'intégration du backjumping dans le solveur `ASPeRiX` est proposée à partir de ces justifications.

# 6

# Backjumping pour ASPeRiX

Précédemment, la procédure de recherche d'ASPeRiX a été détaillée. Cette dernière est basée sur la construction d'une interprétation grâce à des phases de propagation où des instances de règles sont déclenchées qui alternent avec des phases de points de choix où une instance de règle applicable est soit débloquée soit bloquée. Ainsi, la procédure de recherche d'ASPeRiX construit un arbre de recherche où chaque nœud correspond à un point de choix sur une instance de règle applicable du programme ASP traité. La branche gauche d'un point de choix correspond à débloquer l'instance de règle applicable tandis que la branche droite correspond au blocage de cette même règle.

Lorsqu'une branche gauche d'un nœud de l'arbre de recherche échoue, autrement dit elle n'aboutit pas à un answer set, on parcourt systématiquement sa branche droite en proposant de bloquer la règle du point de choix au lieu de la débloquer. De la même façon, lorsqu'une branche droite d'un nœud de l'arbre échoue, on remonte au nœud précédent (s'il existe) et l'on réitère ce processus jusqu'à ce que tous les nœuds de l'arbre soient parcourus. Cette technique de retour-arrière est appelée *backtracking* et est à la base de l'implémentation des algorithmes de recherche. Cependant, dans certains cas il est inutile de parcourir la branche droite d'un point de choix si l'on est certain qu'elle aboutira à un échec.

Le backtracking intelligent ou plus communément appelé *backjumping* est une technique issue de la résolution du problème SAT [41] qui consiste à éviter de parcourir des branches vouées à l'échec en remontant au nœud de l'arbre le plus récent qui participe à l'échec de la branche. Cette technique a déjà été adaptée avec succès dans des solveurs basés sur des atomes notamment dans DLV [47]. L'idée est donc d'adapter le backjumping aux spécificités du solveur ASPeRiX afin de parcourir plus efficacement l'arbre de recherche.

Pour pouvoir intégrer le backjumping à la procédure de recherche d'ASPeRiX, le principe est de proposer une structure capable de justifier la raison d'un échec d'une branche (ou d'un sous-arbre) de manière similaire aux justifications du chapitre précédent. Cette raison d'échec doit permettre de remonter au point de choix le plus récent en cause dans l'échec et ainsi d'éviter de parcourir les branches qui sont vouées au même échec.

Les raisons définies auparavant sont adaptées à cette problématique. Elles sont constituées d'un ensemble de règles responsables de l'échec. D'un point de vue pratique, ce qui nous intéresse c'est l'endroit dans l'arbre où ces règles ont été utilisées. Autrement dit, ce qui nous importe c'est le nœud de l'arbre auquel est lié chaque règle responsable de l'échec. Nous allons

donc redéfinir une version simplifiée des raisons constituée d'un ensemble de nœuds représentant les points de choix de l'arbre de recherche.

Dans ce chapitre, une présentation de l'adaptation du backjumping dans un solveur au premier ordre à base de règles est proposée. Dans un premier temps, on s'intéresse aux limites de l'approche par backtracking. Ensuite, les caractéristiques de la nouvelle représentation des raisons et le principe de fonctionnement du backjumping sont exprimés. Enfin, l'implémentation du backjumping dans le solveur ASPeRiX est détaillé.

# 1 Limites du backtracking

Le backtracking est une technique très répandue qui garantit de trouver l'ensemble des solutions d'un problème donné en explorant l'intégralité des possibilités. Cela correspond à parcourir tous les nœuds d'un arbre de recherche. Néanmoins, cette approche mène à l'exploration de branches parfois inutiles comme le montre l'exemple suivant.

**Exemple 35.** Soit le programme $P_{35}$ suivant qui code une 2-coloration d'un graphe à quatre sommets et deux arêtes.

$$P_{35} = \left\{ \begin{array}{l} s(1) \leftarrow ., s(2) \leftarrow ., s(3) \leftarrow ., s(4) \leftarrow ., \\ arete(1,3) \leftarrow ., \ arete(3,4) \leftarrow ., \\ vert(4) \leftarrow ., \\ rouge(X) \leftarrow s(X), \ not\ vert(X)., \\ vert(X) \leftarrow s(X), \ not\ rouge(X)., \\ \bot \leftarrow arete(X,Y), rouge(X), rouge(Y)., \\ \bot \leftarrow arete(X,Y), vert(X), vert(Y). \end{array} \right\}$$

Le principe de la recherche guidée par les règles est illustré en figure 6.3. Lorsque tous les faits sont déclenchés, plus aucune propagation n'est possible (voir figure 6.1). Une instance de règle supportée non bloquée est alors choisie. On force alors le déclenchement de la règle $(rouge(1) \leftarrow s(1),\ not\ vert(1).)$ en ajoutant $vert(1)$ dans l'ensemble $OUT$, ce qui permet de déduire ensuite que $rouge(1)$ appartient à l'ensemble $IN$. Puis une seconde instance de règle est choisie. De la même facon, on force le déclenchement de la règle $(rouge(2) \leftarrow s(2),\ not\ vert(2).)$ en ajoutant $vert(2)$ dans l'ensemble $OUT$, ce qui permet de déduire que $rouge(2)$ appartient à l'ensemble $IN$ (voir figure 6.2).

Et enfin, la règle $(rouge(3) \leftarrow s(3),\ not\ vert(3).)$ est choisie. On ajoute alors $vert(3)$ à l'ensemble $OUT$, et on déduit que $rouge(3)$ appartient à $IN$. Une contradiction est alors détectée car la contrainte $(\bot \leftarrow arete(1,3), rouge(1), rouge(3).)$ est déclenchée. On revient alors à la dernière règle choisie et on interdit son application en la bloquant. Comme le corps négatif de la règle ne contient qu'un seul littéral, on déduit simplement que $vert(3)$ doit être vrai en l'ajoutant à l'ensemble $MBT$. La contrainte $(\bot \leftarrow arete(3,4), vert(3), vert(4).)$ est alors déclenchée et cette branche se termine à son tour en échec.

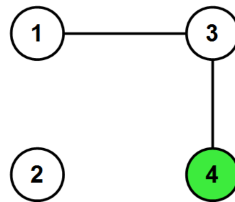



FIGURE 6.1 – Graphe du programme $P_{35}$ après la première phase de propagation

Après avoir parcouru les deux branches du point de choix 3, il devient impossible de colorer le sommet 3. Si l'on réalise un backtrack, on revient alors au point de choix 2 et on interdit le

FIGURE 6.2 – Graphe du programme $P_{35}$ après le premier point de choix (à gauche) puis après le deuxième point de choix (à droite)

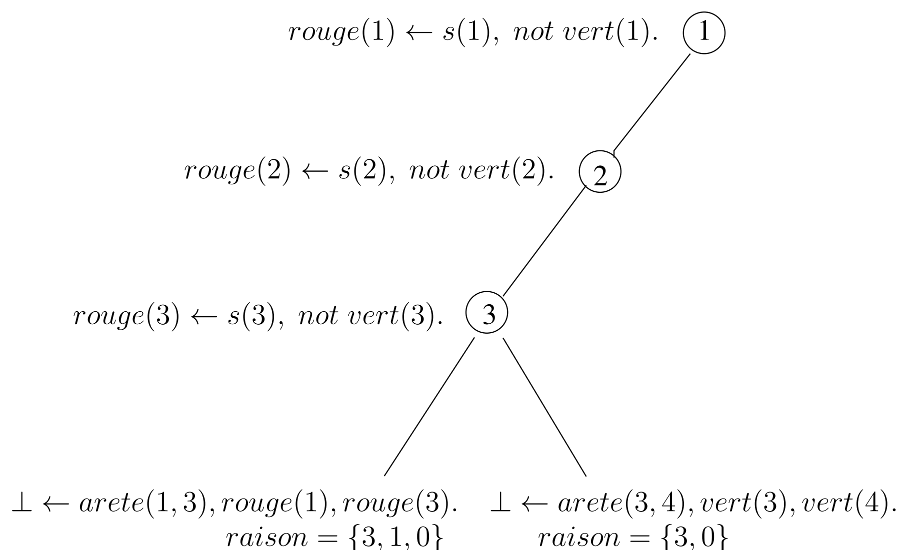



FIGURE 6.3 – Arbre de recherche du programme $P_{35}$

déclenchement de la règle $(rouge(2) \leftarrow s(2),\ not\ vert(2).)$ en bloquant la règle (en ajoutant $vert(2)$ à l'ensemble $MBT$). Néanmoins, ce sous-arbre mène irrémédiablement à un échec puisque la coloration du sommet 2 est totalement indépendante du problème rencontré pour colorer le sommet 3. En effet, l'échec au point de choix nº 3, provient des faits que le sommet 3 est relié aux sommets 1 et 4, qui n'ont pas la même couleur (le sommet 1 est de couleur rouge tandis que le sommet 4 est de couleur verte). Tant que ces données restent inchangées, aucune solution ne peut être trouvée. Ainsi, il est inutile de parcourir la branche droite du point de choix 2 qui porte sur la couleur du sommet 2. En revanche, le point de choix 1 qui porte sur la couleur du sommet 1 fait partie de l'échec de la coloration du sommet 3. On doit donc revenir sur ce point de choix et parcourir sa branche droite afin de déterminer les answers sets du programme $P_{35}$.

À travers l'exemple 35, on peut observer que le parcours systématique de tous les nœuds de l'arbre nous amène dans certains cas à reproduire les mêmes échecs dûs aux mêmes règles. Le *backjumping* présenté dans la suite de ce chapitre permet de revenir directement à l'endroit dans l'arbre où l'échec d'une branche est avéré. Il nécessite de connaître les raisons des échecs, ce qui permet de revenir au point de choix le plus proche susceptible de modifier les conditions ayant provoqué l'échec.

# 2 Les raisons

Le *backjumping* nécessite de justifier les échecs, afin d'éviter d'explorer des branches dans lesquelles on retrouvera ces mêmes échecs. Pour cela, il faut déterminer les littéraux et règles ayant provoqué les échecs. Par exemple, un échec provoqué par le déclenchement de la contrainte $(\perp \leftarrow a,\ not\ c.)$ est dû à la présence de la contrainte dans le programme et au fait qu'elle soit déclenchable ($a \in (IN \cup MBT)$ et $c \in OUT$). Il nous faut donc également déterminer les raisons pour lesquelles on ajoute des contraintes dans le programme, et les raisons pour lesquelles on met des littéraux dans $IN$, dans $OUT$ et dans $MBT$.

Pour intégrer le *backjumping*, ce qui nous intéresse dans les justifications ne sont pas directement les règles déclenchées, débloquées ou bloquées en cause, mais les nœuds (points de choix) dans l'arbre de recherche où ces informations ont été produites. Ainsi, contrairement aux justifications du chapitre précédent, les raisons des littéraux (respectivement des règles) comprendront uniquement les nœuds de l'arbre de recherche qui ont contribué à leur ajout dans l'interprétation partielle (respectivement dans le programme).

A chaque élément des ensembles $IN$, $MBT$ et $OUT$, et à chaque règle du programme (les contraintes en particulier), on va associer l'ensemble des points de choix qui ont participé à leur ajout. Dans le cas d'un choix arbitraire, le seul point de choix incriminé suffit à justifier l'ajout. Dans le cas d'une propagation, l'ensemble des raisons des littéraux du corps de la règle sont nécessaires.

Une *raison* est définie comme un ensemble de niveaux (numérotant les points de choix) dans l'arbre de recherche symbolisés par des entiers positifs ou nuls :

- le niveau 0 correspond à ce qui précède le premier choix (le programme initial et ce qui peut être déduit par propagation de celui-ci) ;
- les points de choix sont numérotés à partir de 1, par ordre chronologique ; un niveau $n$ correspond au $n$-ième point de choix, ainsi que ce qui peut en être directement déduit.

On note, pour un littéral $l$ :

- $R_{IN}(l)$ : la raison pour laquelle $l$ est dans $IN$ ;
- $R_{OUT}(l)$ : la raison pour laquelle $l$ est dans $OUT$ ;
- $R_{MBT}(l)$ : la raison pour laquelle $l$ est dans $MBT$ ;
- $R_{ind}(l)$ : la raison pour laquelle $l$ est indéterminée ($l \notin (IN \cup OUT)$) ;

et, pour une règle $r$, $R(r)$ est la raison pour laquelle $r$ a été ajoutée au programme.

## 2.1 Les raisons des littéraux et des contraintes

On définit dans cette section la façon de calculer les raisons pour lesquelles un littéral est ajouté dans $IN$, dans $OUT$ ou dans $MBT$, et celles pour lesquelles une règle est ajoutée dans le programme. En pratique, seules des contraintes sont ajoutées en cours de recherche mais, afin d'uniformiser le traitement des contraintes et des autres règles, nous définissons des raisons pour toutes les règles du programme.

Les raisons présentées ici sont similaires à celle du chapitre précédent sur les justifications. La seule différence réside dans le fait que les raisons sont constituées de nœuds de l'arbre plutôt que de règles instanciées.

Les raisons sont définies comme suit, relativement à une interprétation partielle $\langle IN, MBT, OUT\rangle$ :

**Les règles.** A chaque règle et contrainte $r$ du programme initial, on associe la raison $\{0\}$ correspondant à ce qui est indépendant de tout choix effectué lors de la recherche :

$$R(r) = \{0\}.$$

**Exemple 36.** En reprenant l'exemple $P_{35}$, les contraintes $(\bot \leftarrow arete(1,3), rouge(1), rouge(3).)$ et $(\bot \leftarrow arete(3,4), vert(3), vert(4).)$ déclenchées respectivement après les branches gauche et droite du point de choix nº 3 de la figure 6.3 ont pour raison $\{0\}$. En effet, il s'agit ici d'instances des règles $(\bot \leftarrow arete(X,Y), rouge(X), rouge(Y).)$ et $(\bot \leftarrow arete(X,Y), vert(X), vert(Y).)$ du programme initial.

A une contrainte générée pour bloquer une règle $r$ choisie lors d'un point de choix de niveau $n$, on associe la raison $\{n\}$, le choix arbitraire justifiant à lui seul l'ajout :

$$R(\bot \leftarrow corps^-(r)) = \{n\}.$$

**Les littéraux de $OUT$.** Lors d'un point de choix de niveau $n$, où une règle $r$ est choisie pour être débloquée (branche gauche), les littéraux du corps négatif de la règle sont ajoutés dans $OUT$ avec pour seule justification que l'on a décidé de débloquer la règle à ce niveau :

$$\forall l \in corps^-(r) \setminus OUT, \quad R_{OUT}(l) = \{n\}.$$

**Exemple 37.** En reprenant l'exemple $P_{35}$, au point de choix nº 1 de la figure 6.3, le littéral $vert(1)$ est ajouté à l'ensemble $OUT$ avec la raison $\{1\}$ dans la branche gauche. De même, le littéral $vert(2)$ (respectivement $vert(3)$) est ajouté à l'ensemble $OUT$ avec la raison $\{2\}$ (respectivement $\{3\}$) dans la branche gauche du point de choix nº 2 (respectivement du point de choix nº 3).

Notons qu'en pratique, lorsqu'une composante fortement connexe est résolue, les littéraux indéterminés de cette composante sont implicitement ajoutés à l'ensemble $OUT$. La raison de cet ajout est donc la raison pour laquelle le littéral est indéterminé, i.e., la raison pour laquelle il n'a pas pu être prouvé. Ces raisons sont détaillées dans la section 2.3.

**Les littéraux de $IN$.** La raison de l'ajout de $tête(r)$ dans $IN$ est l'ensemble des raisons pour lesquelles les littéraux du corps positif (resp. négatif) sont dans $IN$ (resp. $OUT$), auquel on ajoute la raison de la règle elle-même (qui est toujours $\{0\}$ si ce n'est pas une contrainte ajoutée au programme lors d'un point de choix) :

$$R_{IN}(tête(r)) = \bigcup_{l \in corps^+(r)} R_{IN}(l) \cup \bigcup_{l \in corps^-(r)} R_{OUT}(l) \cup R(r).$$

**Exemple 38.** En reprenant l'exemple $P_{35}$, dans la branche gauche du point de choix nº 1 de la figure 6.3, la règle $(rouge(1) \leftarrow s(1), not\ vert(1).)$ est déclenchable. Le littéral $rouge(1)$ est alors ajouté à $IN$. Sa raison contient la raison pour laquelle $s(1)$ est dans $IN$, la raison pour laquelle $vert(1)$ est dans $OUT$ ainsi que la raison de la règle elle même. Or $s(1)$ est dans $IN$ car il appartient aux faits du programme $P_{35}$ donc sa raison est $\{0\}$, $vert(1)$ est dans $OUT$ avec la raison $\{1\}$ (voir exemple 37) et la règle $(rouge(1) \leftarrow s(1), not\ vert(1).)$ a pour raison $\{0\}$. Le littéral $rouge(1)$ admet donc pour raison $\{0,1\}$. De la même façon, dans la branche gauche du point de choix nº 2 (respectivement du point de choix nº 3) de la figure 6.3, la règle $(rouge(2) \leftarrow s(2), not\ vert(2).)$ (respectivement $(rouge(3) \leftarrow s(3), not\ vert(3).)$) est déclenchable et admet pour raison $\{0,2\}$ (respectivement $\{0,3\}$).

**Les littéraux de $MBT$.** De la même façon que pour les littéraux de $IN$, lors de la phase de propagation, la raison de l'ajout de $tête(r)$ dans $MBT$ est l'ensemble des raisons pour lesquelles les littéraux du corps positif (resp. négatif) sont dans $IN \cup MBT$ (resp. $OUT$), auquel on ajoute la raison de la règle elle-même :

$$R_{MBT}(tête(r)) = \bigcup_{l \in corps^+(r) \cap IN} R_{IN}(l) \cup \bigcup_{l \in corps^+(r) \cap MBT} R_{MBT}(l) \cup \bigcup_{l \in corps^-(r)} R_{OUT}(l) \cup R(r).$$

Lors d'un point de choix de niveau $n$, un littéral $l$ est ajouté à l'ensemble $MBT$ si on décide de bloquer arbitrairement une règle $r$ (branche droite) dont le corps négatif contient uniquement ce littéral. La justification de cet ajout est uniquement ce choix de bloquer la règle à ce niveau :

$$R_{MBT}(l) = \{n\}.$$

**Exemple 39.** En reprenant l'exemple $P_{35}$, dans la branche droite du point de choix n° 3 de la figure 6.3, le littéral $vert(3)$ est ajouté à l'ensemble $MBT$ avec la raison $\{3\}$.

## 2.2 Les raisons des échecs

Trois formes d'échec sont possibles pour une branche et correspondent aux différents cas de préfixe bloqué de computation justifiée du chapitre précédent : soit on découvre une contradiction suite à la phase de propagation avec $(IN \cup MBT) \cap OUT \neq \emptyset$ (cas (Propagation-failure) et (Mbt-propagation-failure) d'un préfixe bloqué de computation justifiée), soit il n'y a plus aucune règle applicable mais il y a une contrainte non satisfaite, i.e., supportée et non bloquée (cas ($\Delta_{cho_mbt}$ Convergence-failure) d'un préfixe bloqué de computation justifiée) ou bien l'ensemble $MBT$ est non vide lorsque plus aucune règle n'est applicable (cas (Mbt-Convergence-failure) d'un préfixe bloqué de computation justifiée).

Dans le premier cas, il existe au moins un littéral $l \in IN \cap OUT$ (respectivement $l \in MBT \cap OUT$), la raison de la contradiction est simplement la raison pour laquelle $l$ est dans $IN$ (respectivement dans $MBT$) et la raison pour laquelle $l$ est dans $OUT$ :

$$(a) \quad R_{IN}(l) \cup R_{OUT}(l) \text{ (respectivement } R_{MBT}(l) \cup R_{OUT}(l)).$$

Le deuxième cas, où il existe au moins une contrainte non satisfaite, est spécifique à notre approche. En effet, la grande majorité des solveurs opérant leurs choix sur les littéraux, lorsque tous les choix sont faits, la valeur de chaque littéral est déterminée. Ce n'est pas le cas de notre approche où, les choix portant sur les règles, l'interprétation $\langle IN, MBT, OUT\rangle$ peut rester partielle jusqu'à la fin de la recherche ; les littéraux indéterminés (qui ne sont ni dans $IN$ ni dans $OUT$) sont des littéraux non prouvables (puisqu'il n'y a plus de règle qui peut être appliquée) et qui ne peuvent donc pas appartenir au modèle. $IN$ est un answer set si aucune contrainte n'est applicable relativement à l'interprétation partielle $\langle IN, MBT, OUT\rangle$. S'il existe une contrainte $c$ non satisfaite, la raison de l'échec est l'ensemble des raisons qui rendent la contrainte supportée et non bloquée :

$$(b) \quad \bigcup_{l \in corps^+(c)} R_{IN}(l) \cup \bigcup_{l \in corps^-(c) \cap OUT} R_{OUT}(l) \cup \bigcup_{l \in corps^-(c) \setminus OUT} R_{ind}(l) \cup R(c).$$

La difficulté ici est que la contrainte est non bloquée ($corps^-(c) \cap IN = \emptyset$) mais pas débloquée ($corps^-(c) \not\subseteq OUT$), sinon elle aurait été déclenchée lors d'une phase de propagation. Il y a

donc au moins un littéral du corps négatif dont le statut est resté indéterminé et donc pour lequel la raison de l'absence dans $IN$ est inconnue.

Dans le dernier cas, il existe au moins un littéral présent dans l'ensemble $MBT$ à la fin de la recherche. Il s'agit ici d'un littéral qui devrait appartenir à l'ensemble $IN$ d'après les règles déclenchées, appliquées ou bloquées sur la branche de l'arbre de recherche. Cependant, aucune règle permettant de déterminer son appartenance à l'ensemble $IN$ n'a pu être déclenchée. La raison de cet échec est donc la raison qui a permis de déterminer son appartenance à l'ensemble $MBT$ ainsi que celle de son indétermination :

$$(c) \;\; R_{MBT}(l) \cup R_{ind}(l).$$

Plusieurs raisons d'échecs sont possibles pour une branche. Le solveur `ASPeRiX` s'arrête à la première cause d'échec trouvée sans que cela ne garantisse que ce soit la plus efficace pour permettre au backjumping de remonter dans l'arbre. Ainsi, `ASPeRiX` s'arrête à la première contradiction de littéral trouvé, au premier littéral $MBT$ trouvé à la fin de la recherche ou à la première contrainte non satisfaite trouvée à la fin de la recherche.

La manière de calculer la raison de l'absence d'un littéral est détaillée dans la section suivante.

## 2.3 Les raisons des littéraux indéterminés

Un littéral est indéterminé s'il n'a pas prouvé son appartenance à l'ensemble $IN$ et n'a jamais été ajouté à l'ensemble $OUT$ lors d'un point de choix. On a vu dans la section précédente qu'il peut s'agir soit d'un littéral n'appartenant à aucun ensemble de l'interprétation partielle $\langle IN, MBT, OUT\rangle$ en bout de branche, soit d'un littéral qui est resté dans l'ensemble $MBT$ mais n'a jamais prouvé son appartenance à l'ensemble $IN$. Intuitivement, si un littéral $a_0$ n'est pas dans $IN$, c'est parce qu'aucune des règles instanciées concluant $a_0$ n'a été déclenchée. Il nous faut donc déterminer pourquoi une règle n'est jamais déclenchée tout au long d'une branche de l'arbre de recherche. La raison d'un littéral indéterminé est calculée de manière similaire à celle des justifications.

Soient $\langle IN, MBT, OUT\rangle$ une interprétation partielle, et $r$ une règle instanciée. On dit qu'un littéral $l$ du corps de la règle *neutralise* $r$ si $l \in corps^+(r)\backslash IN$ ou $l \in corps^-(r)\cap(IN\cup MBT)$. Un littéral neutralise donc une règle s'il empêche la règle de pouvoir être appliquée : c'est un littéral du corps positif qui n'est pas dans $IN$ et donc empêche la règle d'être supportée, ou un littéral du corps négatif qui est dans $IN$ et donc bloque la règle. On peut alors dire qu'une règle $r$ est non applicable si et seulement s'il existe un littéral $l$ qui neutralise $r$. Chaque littéral $l$ neutralisant $r$ est donc une cause, une raison pour laquelle la règle $r$ n'est pas applicable.

Un littéral $l_0$ n'a pas pu être prouvé dans une interprétation partielle $\langle IN, MBT, OUT\rangle$ si aucune règle qui conclut $l_0$ n'a pu être déclenchée. Notons $Rl_0$ l'ensemble des règles instanciées de tête $l_0$. Si aucune règle de $Rl_0$ n'a été déclenchée durant le recherche, c'est que chaque règle $r$ de $Rl_0$ soit est neutralisée par (au moins) un littéral, soit, si elle était applicable, a été choisie lors d'un point de choix et bloquée par l'ajout d'une contrainte dans le programme. Dans ce dernier cas, la raison du non-déclenchement de la règle est simplement ce choix arbitraire.

Pour déterminer les règles de $Rl_0$ qui sont bloquées par une contrainte lors d'un point de choix, on utilise l'ensemble $Subst$ des instances de règles choisies durant la recherche. Ces règles incluent toutes les règles déclenchées par propagation ainsi que toutes celles sélectionnées lors des points de choix. Ainsi, $Rl_0 \cap Subst$ sont les règles de $Rl_0$ qui ont été choisies et bloquées par une contrainte.

Notons *RNonDec(r)* la raison pour laquelle la règle $r$ n'est pas déclenchable. On a :

$$RNonDec(r) = \begin{cases} R_{OUT}(l) & \text{si } l \in (corps^{+}(r) \cap OUT) \\ R_{IN}(l) & \text{si } l \in (corps^{-}(r) \cap IN) \\ R_{MBT}(l) & \text{si } l \in (corps^{-}(r) \cap MBT) \\ R(\bot \leftarrow corps^{-}(r)) & \text{si } r \in Subst \\ R_{ind}(l) & \text{si } l \in (corps^{+}(r) \setminus (IN \cup OUT)) \end{cases}$$

Plusieurs raisons peuvent empêcher une règle d'être déclenchable. En effet, plusieurs littéraux peuvent neutraliser la règle et la règle peut également être bloquée par une contrainte ajoutée lors d'un point de choix. En pratique, le solveur ASPeRiX se contente de récupérer une raison pour éviter un coût calculatoire trop important. Néanmoins, ce choix ne garantit en aucun cas de récupérer la meilleure raison qui permettrait le backjumping le plus efficace.

L'ordre d'écriture dans lequel la raison d'une règle non déclenchable est établie dans la définition de $RNonDec(r)$ correspond au traitement effectué par le solveur ASPeRiX pour déterminer une raison pour laquelle une règle est non déclenchable. Dans un premier temps, il recherche un littéral du corps positif qui appartient à l'ensemble $OUT$. Puis, si aucun littéral n'est trouvé, il recherche un littéral du corps négatif qui appartient à $IN \cup MBT$. De la même façon, si aucun littéral n'est trouvé, il vérifie si la règle a été bloquée par point de choix. Enfin, il recherche un littéral indéterminé dans le corps positif de la règle si aucune autre raison n'a été trouvée. Cette manière de calculer la raison d'une règle non déclenchable permet ainsi de limiter le plus possible le calcul de littéraux indéterminés qui peut être très coûteux et boucler sous certaines conditions.

En conclusion, la raison d'un littéral indéterminé $l_0$ est donc une raison pour laquelle chaque règle concluant $l_0$ (chaque règle ayant pour tête $l_0$) n'est pas déclenchable :

$$R_{ind}(l_0) = \bigcup_{r \in Rl_0} RNonDec(r)$$

# 3 Le backjumping

## 3.1 Principe général

En cas d'échec sur une branche, dont la raison est $raison_echec$, on revient sur le point de choix le plus récent participant à l'échec, i.e., on remonte au niveau $MAX(raison_echec)$ avec

$$MAX(S) = x \text{ tel que } x \in S \text{ et } \forall y \in S, y \leq x$$

Le niveau $MAX(raison_echec)$ est le plus haut niveau auquel on peut remonter en ayant la garantie de ne pas couper des branches qui mènent à un answer set. Il s'agit du niveau le plus récent qui est impliqué dans l'échec de la branche.

Lorsque l'échec est dû à une contradiction (suite à la phase de propagation), le point de choix courant est toujours en cause dans l'échec, le *backjumping* se ramène alors à un simple *backtracking*.

## 3.2 Combinaison des échecs

La combinaison des échecs a déjà été abordée dans le chapitre précédent sur les justifications. Lorsque les branches gauche et droite d'un point de choix $n$ échouent et que $n$ appartient aux raisons de l'échec de chacune des deux branches, il est possible de remonter au point de choix

le plus récent hormis $n$ participant à au moins un des échecs. L'idée est donc de combiner les raisons d'échecs des deux branches et d'exclure le point de choix $n$ de cette nouvelle raison d'échec.

La notion de préfixe échec du chapitre précédent est définie dans ce cas de figure où l'on combine les raisons d'échecs de deux branches d'un même nœud ainsi que dans le cas d'une raison d'échec d'une seule branche.

**Exemple 40.** En reprenant l'exemple de la figure 6.3, au point de choix nº 3, l'échec de la branche gauche provient de la contrainte déclenchable ($\bot \leftarrow arete(1,3), rouge(1), rouge(3).$) dont la raison est {0} (voir exemple 36), des littéraux $arete(1,3)$ de raison {0}, $rouge(1)$ de raison {0,1} (voir exemple 37) et $rouge(3)$ de raison {0,3} (voir exemple 37). Étant donné que $\bot$ appartient déjà à $OUT$ par défaut avec une raison {0}, la raison d'échec est {0,1,3}.

L'échec de la branche droite provient de la contrainte faiblement déclenchable ($\bot \leftarrow arete(3,4)$, $vert(3)$, $vert(4).$) dont la raison est {0} (voir exemple 36), des littéraux $arete(3,4)$ de raison {0}, $vert(3)$ de raison {3} (voir exemple 39) et $vert(4)$ de raison {0}. Étant donné que $\bot$ appartient déjà à $OUT$ par défaut avec une raison {0}, la raison d'échec est {0,3}.

Le point de choix le plus récent, hormis 3, participant à l'un au moins des deux échecs est 1, c'est le plus haut niveau auquel on peut revenir de façon sûre.

### 3.3 Lien avec les justifications

La procédure de recherche d'`ASPeRiX` suit le même mode opératoire que les computations du chapitre précédent. En effet, comme pour une computation justifiée, `ASPeRiX` déclenche toutes les instances de règles possibles durant la phase de propagation avant de réaliser un choix sur une instance de règle applicable. C'est pourquoi les propriétés sur les computations se vérifient également dans l'arbre de recherche d'`ASPeRiX`.

Ainsi, si en un point de l'arbre un ensemble de règles déclenchées, débloquées ou bloquées forment un ensemble de blocage (aucun answer set compatible avec cet ensemble), le prolongement des branches à partir de cet endroit dans l'arbre ne mènera à aucun answer set.

Dans la pratique, les seules informations présentes dans un ensemble de blocage sont les niveaux dans l'arbre de recherche dont les règles sont issues. Néanmoins, d'après le théorème 8, un ensemble de blocage est avéré au niveau du dernier point de choix de son ensemble de règles. Par conséquent, le fait de ne conserver que les niveaux dont les règles sont issues est suffisant pour caractériser un ensemble de blocage.

De plus, d'après les théorèmes 9 et 11, une raison d'échec est un ensemble de blocage et il n'existe aucun answer set à partir de l'endroit où l'échec est avéré. Cela permet de justifier le fait qu'aucun answer set ne peut être déterminé à partir du point de choix le plus récent d'une raison d'échec d'une branche ou d'une combinaison de branches. On est donc assuré que les parties non explorées de l'arbre ne contiennent pas de solution.

L'algorithme d' `ASPeRiX` étant juste et complet et les branches coupées par le backjumping ne menant à aucun answer set, l'algorithme avec backjumping est juste et complet.

## 4 Implémentation

Dans cette partie, on s'intéresse aux changements apportés par le backjumping sur l'algorithme d'`ASPeRiX`. Les calculs des raisons peuvent être effectués directement au cours de la recherche en chaînage avant à l'exception des raisons des littéraux indéterminés. Le calcul d'une raison

d'indétermination d'un littéral nécessitera d'effectuer une recherche en chaînage arrière sur les règles d'un programme ASP.

## 4.1 L'algorithme d'ASPeRiX

L'algorithme 8 reprend la fonction `Solve` de la section 2.1 en lui ajoutant un mécanisme de *backjumping*. Un nouveau paramètre, $niveau$, représente le numéro du point de choix courant, qui vaut 0 au premier appel. En cas d'échec, la fonction ne retourne plus seulement qu'aucun answer set n'a été trouvé mais lui adjoint une raison de l'échec. En cas de succès, elle retourne l'ensemble réponse trouvé (avec la raison $\emptyset$).

L'algorithme se focalise sur le backjumping proprement dit et omet les raisons des littéraux et des contraintes.

La phase de propagation est inchangée. Si une contradiction est détectée, la fonction retourne la constante $no_answer_set$ signifiant un échec sur la branche et calcule la raison de la contradiction (comme indiqué dans le cas $(a)$ de la section 2.2).

Lors de la phase de point de choix, si une règle applicable de $P_R$ de la composante fortement connexe ($CFC$) courante a été choisie, on crée un nouveau point de choix en incrémentant la variable $niveau$. On commence par forcer l'application de la règle. Si ce choix échoue, la raison de l'échec est calculée dans la variable $Raison_g$. Il faut alors examiner cette raison pour savoir s'il faut explorer la branche droite ou si on peut remonter directement à un niveau inférieur. Si le niveau de *backjumping*, $MAX(Raison_g)$ (qui représente alors le niveau du point de choix le plus récent impliqué dans l'inconsistance), est inférieur au niveau courant $niveau$, alors on « saute » le point de choix courant et on remonte au choix précédent. Sinon le niveau courant est impliqué dans l'échec, on explore donc la branche droite du point de choix. Si celle-ci mène elle aussi à un échec, on calcule la raison $Raison_d$ et on l'examine à son tour pour déterminer si le *backjumping* est possible. S'il n'est pas possible, cela signifie que le point de choix courant est impliqué dans les deux échecs, la raison globale de l'échec du nœud est alors l'union des deux raisons (à laquelle on enlève le numéro du point de choix courant, déjà exploré).

Lorsqu'il n'y a plus de règle de $P_R$ applicable dans la $CFC$ courante, 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 $MBT$, une raison d'échec est calculée de la manière présentée dans le cas $(c)$ de la section 2.2. De la même façon, on vérifie qu'aucune contrainte n'est applicable lorsqu'on ajoute implicitement les instances des prédicats de la $CFC$ courante qui ne sont pas dans $IN$ à l'ensemble $OUT$. Si une contrainte est violée, une raison d'échec est calculée de la manière présentée dans le cas $(b)$ de la section 2.2. Lorsque la dernière CFC est résolue, l'ensemble $IN$ représente alors un answer set de $P$.

## 4.2 Détails d'implémentation

Le calcul des raisons est effectué comme indiqué dans la section 2. La seule difficulté réside dans le calcul de la raison d'un littéral indéterminé. Il nécessite de calculer une raison pour laquelle chaque règle instanciée ayant pour tête le littéral indéterminé n'a pas été déclenchée. Cependant, comme le solveur `ASPeRiX` instancie directement les règles d'un programme au premier ordre à la volée, l'ensemble des règles instanciées ayant pour tête un littéral indéterminé n'est pas facilement calculable.

Une solution possible est de construire cet ensemble en effectuant une recherche en chaînage arrière sur les différentes règles d'un programme. Pour cela, on utilise le langage logique

**Algorithme 8** : *solve_backjump*

```
1  Fonction solve_backjump(P_R, P_K, IN, MBT, OUT, CFC, Subst, niveau);
2  // Recherche d'un answer set pour le programme P = P_R ∪ P_K
3  repeat //Phase de propagation
4      r_0 ← γ_pro(P_R ∪ P_K, IN, MBT, OUT, CFC, Subst);
5      if r_0 ≠ NULL then
6          if (corps⁺(r_0) ∩ MBT) ≠ ∅ then
7              MBT ← MBT ∪ {tête(r_0)};
8          else
9              IN ← IN ∪ {tête(r_0)};
10             if (tête(r_0) ∈ MBT) then
11                 MBT ← MBT\{tête(r_0)};
12 until r_0 = NULL;
13 if ((IN ∪ MBT) ∩ OUT ≠ ∅) then //Contradiction détectée
14     return (no_answer_set, raison_contradiction((IN ∪ MBT) ∩ OUT));
15 else
16     r_0 ← γ_cho(P_R, IN, MBT, OUT, CFC, Subst);
17     if r_0 ≠ NULL then //Point de Choix
18         niveau ← niveau + 1 ;
19         // branche gauche
20         (stop, Raison_g) ←
           solve_backjump(P_R, P_K, IN, MBT, OUT ∪ corps⁻(r_0), CFC, Subst, niveau);
21         if stop = no_answer_set then
22             if MAX(Raison_g) < niveau then // backjump
23                 return (no_answer_set, Raison_g);
24             else// branche droite
25                 literals ← {a|a ∈ corps⁻(r_0), pred(a) ∈ pred(CFC)};
26                 if (|literals| = 1) then
27                     MBT ← MBT ∪ literals;
28                 else
29                     P_K ← P_K ∪ {⊥ ← ∪_{a_i∈literals} not a_i};
30                 (stop, Raison_d) ←
                   solve_backjump(P_R, P_K, IN, MBT, OUT, CFC, Subst, niveau);
31                 if stop = no_answer_set then
32                     if MAX(Raison_d) < niveau then // backjump
33                         return (no_answer_set, Raison_d);
34                     else//combinaison des echecs
35                         return (no_answer_set, (Raison_g ∪ Raison_d) \ {niveau});
36     else// La CFC est résolue
37         if pred(MBT) ∩ pred(CFC) ≠ ∅ then // Un littéral MBT ne peut plus être prouvé
38             return
               (no_answer_set, raison_MbtNonVide(P_R ∪ P_K, IN, MBT, OUT, CFC, Subst));
39         else
40             if γ_check(P_R ∪ P_K, IN, MBT, OUT, CFC) then // une contrainte est violée
41                 return
                   (no_answer_set, raison_contrainte_nonSat(P_R, P_K, IN, MBT, OUT, CFC, Subst));
42             else
43                 if ¬last(CFC) then
44                     return solve(P_R, P_K, IN, MBT, OUT, CFC + 1, Subst);
45                 else// Un answer set est trouvé
46                     return (IN, ∅);
```

Prolog. On propose donc une traduction des règles ASP en clauses Prolog de manière à calculer une raison pour laquelle une règle $r$ d'un littéral indéterminé n'a pas été déclenchée. Cependant, plusieurs raisons de non déclenchement d'une règle sont possibles. On calcule la première raison trouvée selon l'ordre d'écriture établi pour $RNonDec(r)$ de la section 2.3.

L'invocation au sein du solveur ASPeRiX du calcul de la raison d'indétermination d'un atome $sp(a_1, \ldots, a_n)$ nécessite l'appel à la primitive metalogique $setof$ de Prolog. L'appel suivant calcule une raison $R$ pour chaque règle ayant pour tête le littéral $sp(a_1, \ldots, a_n)$ et l'ajoute à l'ensemble des raisons $Raisons$.

```
setof(Raison, sp(a_1,...,a_n,Raison,IN,OUT,CONT,Borne), Raisons)
```

On ajoute au littéral les arguments :

- $Raison$ la raison de non déclenchement de la règle ;
- $IN$ et $OUT$, les littéraux appartenant à l'interprétation partielle en cours ;
- $CONT$ les contraintes ajoutées au cours de la recherche[1] ;
- $Borne$ le nombre d'appels à des calculs de littéraux indéterminés (pour cette règle).

**Traduction de la tête de règle.**
On cherche la raison de non déclenchement d'une règle de tête $sp(a_1, \ldots, a_n)$.

$TradTete($`sp(a_1,...,a_n)`$) =$
`sp(a_1,...,a_n,Raison,IN,OUT,CONT,Borne)`

**Traduction d'un corps avec plusieurs littéraux.**
Si le premier littéral positif appartient à $OUT$, on retourne sa raison. Sinon s'il appartient à $IN$, on poursuit la recherche sur les littéraux suivants. Sinon on recherche pourquoi le premier littéral est indéterminé. [2]

$TradCorps($`(sp(a_1,...,a_n),b),h`$) =$
`member(sp(a_1,...,a_n,Raison),OUT);`
`member(sp(a_1,...,a_n,_),IN) *->`
$TradCorps($`b,h`$)$`;`
`sp(a_1,...,a_n,Raison,IN,OUT,CONT,Borne_)`

**Traduction d'un corps avec un seul littéral positif.**
Si le littéral appartient à $OUT$, on retourne sa raison. Sinon s'il n'appartient pas à $IN$, on recherche pourquoi le premier littéral est indéterminé.

---

[1] En pratique, $CONT$ contient uniquement les têtes de règles bloquées par une contrainte

[2] La traduction utilise un opérateur de “si alors sinon” appelé *softcut* dans gnu-prolog, noté `(_ *-> _; _)`, et qui permet a contrario du “si alors sinon” classique de Prolog, noté `(_ -> _; _)`, de ne pas couper les points de choix de la conditionnelle lorsque celle-ci obtient un succès. Son code est le suivant :

```
:- op(1050, xfy, '*->').
'*->'(C, G1;_G2) :-
    call(C),
    call(G1).
'*->'(C, _G1;G2) :-
    \+ call(C),
    call(G2).
```

Dans la traduction, l'appel sur la négation par l'échec s'effectue toujours sur un but complètement instancié.

```
TradCorps(sp(a_1,...,a_n),h) =
  member(sp(a_1,...,a_n,Raison),OUT);
  member(sp(a_1,...,a_n,_),IN) *->
  fail;
  sp(a_1,...,a_n,Raison,IN,OUT,CONT,Borne_)
```

**Traduction d'une règle.**

```
TradRegle(s(a_1,...,a_n).) =
  s(_,...,_,[0],_,_,_,_).

TradRegle((h :- b.)) =
  TradTete(h):-
  Borne < K, Borne_ is Borne + 1 ->
  TradCorps(b,h);
  bornesetof(Raison).
```

$K$ est une constante fixée interne au système correspondant au nombre d'appels maximum à des calculs de littéraux indéterminés (pour cette règle). Si cette borne est atteinte une raison $-1$ est renvoyée par le fait `Prolog`

```
bornesetof([-1]).
```

qui exprime que tout est raison. Cela signifie qu'on n'a pas réussi à trouver une raison de non déclenchement de la règle dans la limite du nombre d'appels récursifs autorisé.

**Exemple 41.** Exemple de traduction de la règle $(arc(X,Y) \leftarrow arc(Y,X).)$.

```
TradRegle(arc(X,Y) :- arc(Y,X)) =
arc(X,Y,Raison,IN,OUT,CONT,Borne) :-
Borne<K, Borne_ is (Borne + 1) ->
   ( member(arc(Y,X,Raison),OUT);
     (member(arc(Y,X,_),IN) *->
       fail;
       arc(Y,X,Raison,IN,OUT,CONT,Borne_))
     )
   );
   bornesetof(Raison).
```

**Traduction d'un corps avec plusieurs littéraux négatifs.**
Si le premier littéral négatif appartient à $IN$, on retourne sa raison. Sinon on poursuit la recherche sur les littéraux suivants du corps négatif.

```
TradCorps((not sp(a_1,...,a_n),b),h) =
  member(sp(a_1,...,a_n,Raison),IN) *->
  true;
  TradCorps(b,h)
```

**Calcul de la raison de la contrainte lors d'une exclusion de la règle.**
On recherche si une règle de tête `sp(a_1,...,a_n)` a été bloquée et appartient à $CONT$. On retourne alors sa raison.

$TradContrainte($`sp(a_1,...,a_n)`$) =$
`member(sp(a_1,...,a_n,Raison),CONT)`

**Traduction d'un corps avec un seul littéral négatif.**
Si le littéral négatif appartient à $IN$, on retourne sa raison. Sinon on recherche si la règle est bloquée par une contrainte de $CONT$.

$TradCorps($`not sp(a_1,...,a_n),h`$) =$
```
member(sp(a_1,...,a_n,Raison),IN) *->
true;
```
$TradContrainte($`h`$)$

**Exemple 42.** Exemple de traduction de la règle ayant deux éléments dans son corps négatif ($vert(X) \leftarrow s(X), not\ rouge(X), not\ bleu(X).$).

$TradRegle($`vert(X) :- s(X), not rouge(X), not bleu(X)`$) =$
```
 vert(X,Raison,IN,OUT,CONT,Borne) :-
 Borne<K, Borne_ is (Borne + 1) ->
   ( member(s(X,Raison),OUT);
     (member(s(X,_),IN) *->
       (member(rouge(X,Raison),IN) *->
         true;
         (member(bleu(X,Raison),IN) *->
           true ;
           member(vert(X,Raison),CONT)
         )
       );
       s(X,Raison,IN,OUT,CONT,Borne_))
     )
   );
   bornesetof(Raison).
```

**Traduction des règles du programme ASP**

$$TradProg(P) = \{\texttt{bornesetof([-1]).}\} \cup \bigcup_{r \in P} TradRegle(r)$$

Le programme ASP $P_{43}$ suivant traite un problème de circuit hamiltonien et illustre l'application de la transformation à un programme ASP entier.

**Exemple 43.** `% Problème de circuit hamiltonien`

```
% L'ensemble des sommets
s(1). s(2). s(3). s(4).

% L'ensemble des arcs
a(1,2). a(2,3). a(2,4). a(3,1). a(4,3).


% Sommet initial
init(1).
```

```
% Un arc a(X,Y) est dans (d) ou est hors (h) du circuit
d(X1,X2) :- a(X1,X2), v(X1), not h(X1,X2). % r1
h(X1,X2) :- a(X1,X2), v(X1), not d(X1,X2). % r2


% Chaque sommet doit être visité
v(X) :- init(X).
v(X) :- d(X1,X).
:- s(X), not v(X).

% Pas plus d'une visite par sommet
:- d(X,X1), d(X,X2), X1!=X2.
:- d(X1,X), d(X2,X), X1!=X2.

% Le circuit se clôt sur le point de départ
c :- init(X), d(X1,X).
:- not c.
```

Voici la traduction complète $TradProg(P_{43})$ du programme ASP pour le calcul du cycle hamitonien précédent.

```
%% d(X1,X2):- a(X1,X2), v(X1), not h(X1,X2).
d(X1,X2,Raison,IN,OUT,CONT,Borne) :-
  Borne<20, Borne_ is (Borne + 1) *->
  (member(a(X1,X2,Raison),OUT);
    (member(a(X1,X2,_),IN) *->
      (member(v(X1,Raison),OUT);
        (member(v(X1,_),IN) *->
          (member(h(X1,X2,Raison),IN) *->
            true ;
            member(d(X1,X2,Raison),CONT)
          );
          v(X1,Raison,IN,OUT,CONT,Borne_)
        )
      );
      a(X1,X2,Raison,IN,OUT,CONT,Borne_)
    )
  );
  bornesetof(Raison).

%% h(X1,X2) :- a(X1,X2), v(X1), not d(X1,X2).
h(X1,X2,Raison,IN,OUT,CONT,Borne) :-
  Borne<20, Borne_ is (Borne + 1) *->
  (member(a(X1,X2,Raison),OUT);
    (member(a(X1,X2,_),IN) *->
      (member(v(X1,Raison),OUT);
        (member(v(X1,_),IN) *->
          (member(d(X1,X2,Raison),IN) *->
```

```
            true;
            member(h(X1,X2,Raison),CONT)
          );
          v(X1,Raison,IN,OUT,CONT,Borne_)
        )
      );
      a(X1,X2,Raison,IN,OUT,CONT,Borne_)
    )
  );
  bornesetof(Raison).

%% v(X) :- init(X).
v(X,Raison,IN,OUT,CONT,Borne) :-
  Borne<20, Borne_ is (Borne + 1) *->
  (member(init(X,Raison),OUT);
    (member(init(X,_),IN) *->
      fail ;
      init(X,Raison,IN,OUT,CONT,Borne_)
    )
  );
  bornesetof(Raison).

%% v(X) :- d(X1,X).
v(X,Raison,IN,OUT,CONT,Borne) :-
  Borne<20, Borne_ is (Borne + 1) *->
  (member(d(X1,X,Raison),OUT);
    (member(d(X1,X,_),IN) *->
      fail ;
      d(X1,X,Raison,IN,OUT,CONT,Borne_)
    )
  );
  bornesetof(Raison).

%% c:-init(X), d(X1,X).
c(Raison,IN,OUT,CONT,Borne) :-
  Borne<20, Borne_ is (Borne + 1) *->
  (member(init(X,Raison),OUT);
    (member(init(X,_),IN) *->
      (member(d(X1,X,Raison),OUT);
        (member(d(X1,X,_),IN) *->
          fail ;
          d(X1,X,Raison,IN,OUT,CONT,Borne_)
        )
      );
      init(X,Raison,IN,OUT,CONT,Borne_)
    )
  );
  bornesetof(Raison).
```

```
a(_,_,[0],_,_,_,_).

init(_,[0],_,_,_,_).

s(_,[0],_,_,_,_).

bornesetof([-1]).
```

## 4.3 Limites de l'implémentation

L'implémentation et la traduction sont des preuves de concepts qui montrent plusieurs limites. La première qui porte sur la gestion des contraintes dans la traduction `Prolog` peut se révéler être une modification facile à réaliser ; la deuxième qui porte sur le traitement des exécutions infinies de `Prolog` est plus profonde car elle porte sur le fragment effectivement reconnu ; la troisième qui porte sur la gestion d'instances d'un prédicat dans $OUT$ et d'autres instances indéterminées ainsi que la quatrième qui porte sur le choix de `GNU-Prolog` ouvre la porte à une refonte de l'implémentation en remplaçant `Prolog` par une version adaptée de la machine abstraite de Warren (ou WAM [2]), sous jacente à de nombreuses implémentations `Prolog`.

### 4.3.1 Problème au niveau de la gestion des contraintes dans la traduction `Prolog`

Si une règle $r$ a été bloquée par une contrainte, celle ci est de la forme $(\bot \leftarrow \cup_{b \in corps^-(r)} not\ b.)$. Or, $CONT$ va contenir seulement la tête de la règle bloquée avec sa raison. Si deux règles de même tête ont été bloquées, l'ensemble $CONT$ contiendra deux occurrences de la tête avec deux raisons différentes. Les deux raisons seront récupérées lors de la recherche de la tête de $r$ dans l'ensemble $CONT$ par $member$. Donc la raison calculée pour une règle $r$ ne sera pas juste. Cependant, lors d'un calcul d'une raison d'indétermination, on s'intéresse toujours à l'ensemble de règles et la raison calculée sera exacte.

**Exemple 44.** Supposons deux uniques règles de tête $a$ : $(a \leftarrow b, not\ c_1, not\ c_2.)$ et $(a \leftarrow not\ d_1, not\ d_2.)$ avec $IN = \{b\}$ et $OUT = \emptyset$. On suppose également que ces deux règles ont été bloquées chacune par une contrainte $(\bot \leftarrow not\ c_1, not\ c_2.)$ avec la raison $\{2\}$ et $(\bot \leftarrow not\ d_1, not\ d_2.)$ avec la raison $\{4\}$. La raison d'indétermination de $a$ est la raison pour laquelle chacune des règles a été bloquée par une contrainte, soit $\{2, 4\}$. En pratique, la liste $CONT$ contient $a([2])$ et $a([4])$ correspondant à une règle bloquée de tête $a$ avec la raison $\{2\}$ et une règle bloquée de tête $a$ avec la raison $\{4\}$. La recherche de la raison pour laquelle la première règle a été bloquée va retourner les deux raisons soit $\{2, 4\}$. De la même façon, la recherche de la raison pour laquelle la deuxième règle a été bloquée va retourner $\{2, 4\}$. Donc, la raison d'indétermination de l'atome $a$ sera $\{2, 4\}$ ce qui est le résultat escompté.

### 4.3.2 Problème d'exécution infinie

S'il existe une infinité d'appels récursifs au calcul d'indétermination d'un littéral, le calcul s'arrête en échec et retourne $\{-1\}$.

**Exemple 45.** Soit le programme composé des deux règles $(p(X) \leftarrow q(X).)$ et $(q(X) \leftarrow q(f(X)).)$ avec $IN = \emptyset$ et $OUT = \emptyset$. Le calcul de l'indétermination de $p(a)$ se termine avec la raison $\{-1\}$ lorsque la borne $K$ est atteinte alors que la raison théorique attendue est $\{0\}$.

**Exemple 46.** Soit le programme composé des deux règles $(a(X) \leftarrow b(X).)$ et $(b(X) \leftarrow a(X).)$ avec $IN = \emptyset$ et $OUT = \emptyset$. Le calcul de l'indétermination de $a(1)$ se termine avec la raison $\{-1\}$ lorsque la borne $K$ est atteinte alors que la raison théorique attendue est $\{0\}$.

### 4.3.3 Gestion d'instances d'un prédicat dans $OUT$ et d'autres instances indéterminées

L'utilisation de `Prolog` pour le calcul des raisons se révèle non minimale dès lors qu'un atome est dans $OUT$ et qu'une raison indéterminée peut lui être aussi calculé comme le montre l'exemple suivant basé sur l'exemple 43 du circuit hamiltonien.

**Exemple 47** (Suite de l'exemple 43)**.** La figure 6.4 trace l'arbre de recherche d'`ASPeRiX` pour le programme de l'exemple 43. Un nœud est étiqueté par le niveau du point de choix, de la règle et son instance appliquée. Les arcs sont étiquetés en fils gauche pour les nouveaux $IN$ et $OUT$ et en fils droit par les nouveaux $MBT$. L'échec le plus à gauche dans l'arbre de recherche est dû au déclenchement de la contrainte $(\bot \leftarrow s(3), not\ v(3).)$.

La trace des appels du `setof` sur `v(3)` est détaillée ci-dessous. Les arguments `IN` et `OUT` sont fixés par l'appel du `setof` :

```
IN = {init(1,[0]),s(1,[0]),s(2,[0]),s(3,[0]),s(4,[0]),a(1,2,[0]),
a(2,3,[0]),a(2,4,[0]),a(3,1,[0]),a(4,3,[0]),v(1,[0]),
d(1,2,[0,1]),h(2,3,[0,1,2]),h(2,4,[0,1,3]),v(2,[0,1])}
OUT = {d(2,3,[2]),d(2,4,[3]),h(1,2,[1])}
```

La raison calculée par chaque branche du parcours du `setof` est capturée par une unification sur la variable `Raison`.

```
1.  Call: v(3,Raison,IN,OUT,[],0)
2.     Call: member(init(3,Raison),OUT) % Fail
3.     Call: member(init(3,_),IN) % Fail
4.     Call: init(3,Raison,IN,OUT,[],1)
5.     Exit: init(3,[0],IN,OUT,[],1)
6.  Exit: v(3,[0],IN,OUT,[],0)
7.  Raison = [0]
8.  Redo: v(3,Raison,IN,OUT,[],0)
9.     Call: member(d(X1,3,Raison),OUT) % X1=2
10. Exit: v(3,[2],IN,OUT,[],0)
11. Raison = [2]
12. Redo: v(3,Raison,IN,OUT,[],0)
13.    Call: member(d(X1,3,_),IN) % Fail
14.    Call: d(X1,3,Raison,IN,OUT,[],1)
15.       Call: member(a(X1,3,Raison),OUT) % Fail
16.       Call: member(a(X1,3,_),IN) % X1=2
17.       Call: member(v(2,Raison),OUT) % Fail
18.      Call: member(v(2,_),IN) % Succès
19.      Call: member(h(2,3,Raison),IN) % Succès
20.    Exit: d(2,3,[0,1,2],IN,OUT,[],1)
21. Raison = [0,1,2]
22.    Redo: d(2,3,Raison,IN,OUT,[],1)
23.       Call: member(a(X1,3,[0]),IN) % X1=4
24.       Call: member(v(4,Raison),OUT) % Fail
25.       Call: member(v(4,_),IN) % Fail
```

```
26.       Call: v(4,Raison,IN,OUT,[],2)
27.          Call: member(init(4,Raison),OUT) % Fail
28.          Call: member(init(4,_),IN) % Fail
29.          Call: init(4,Raison,IN,OUT,[],3)
30.          Exit: init(4,[0],IN,OUT,[],3)
31.       Exit: v(4,[0],IN,OUT,[],2)
32. Raison = [0]
33.       Redo: v(4,Raison,IN,OUT,[],2)
34.          Call: member(d(X1,4,Raison),OUT) % X1=2
35.       Exit: v(4,[3],IN,OUT,[],2)
36. Raison = [3]
37.       Redo: v(4,Raison,IN,OUT,[],2)
38.          Call: member(d(X1,4,_),IN) % Fail
39.          Call: d(X1,4,Raison,IN,OUT,[],3)
40.             Call: member(a(X1,4,Raison),OUT) % Fail
41.             Call: member(a(X1,4,_),IN) % X1=2
42.             Call: member(v(2,Raison),OUT) % Fail
43.             Call: member(v(2,_),IN) % Succès
44.             Call: member(h(2,4,Raison),IN) % Succès
45.          Exit: d(2,4,[0,1,3],IN,OUT,[],3)
46. Raison = [0,1,3]

Raisons = [[0],[0,1,2],[0,1,3],[2],[3]]
```

La raison de l'atome `d(2,3)` est calculée deux fois : aux lignes 9 et 14 qui correspondent à deux appels de la seconde clause du prédicat `v/6`. La première raison (ligne 9) provient de l'appartenance de `d(X1,3,Raison)` à $OUT$ tandis que la seconde (ligne 14) provient du calcul en cas d'indétermination, ce qui n'est pas ici le cas pour l'instance $X1 = 2$ puisqu'elle est dans $OUT$. La seconde raison est donc inutile mais n'en est pas pour autant fausse. Pour que le calcul devienne minimal, il faut que la seconde branche de la disjonction de la seconde clause du prédicat `v/6` prenne en compte l'instanciation réalisée lors du calcul de la première branche de la disjonction et l'interdise dans la seconde branche ce qui n'est pas possible dans un calcul classique en `Prolog`, les deux branches de la disjonction étant parfaitement indépendantes.

On peut néanmoins ici le simuler en injectant directement dans la seconde branche de la disjonction de la seconde clause du prédicat `v/6` l'instance capturée par la première branche sur la forme de la contrainte (au sens de `Prolog`) `dif((X1,X),(2,3))`. La seconde clause du prédicat `v/6` devient alors :

```
v(X,Raison,IN,OUT,CONT,Borne) :-
  Borne<20, Borne_ is (Borne + 1) ->
  (member(d(X1,X,Raison),OUT);
    (member(d(X1,X,_),IN) *->
      fail ;
      dif((X1,X),(2,3)),
      d(X1,X,Raison,IN,OUT,CONT,Borne_)
    )
  );
  bornesetof(Raison).
```

Avec cette modification, le calcul des raisons retourne

```
Raisons = [[0],[0,1,3],[2],[3]]
```

qui est minimal. La trace est identique, seules les lignes 17 à 23 incluses disparaissent et la ligne 16 est modifiée en :

```
16.    Call: member(a(X1,3,Raison),IN) % X1=4
```

Une autre solution, statique elle, qui est séduisante mais qui va se révéler fausse est de modifier la traduction pour interdire un quelconque appel pour le calcul en cas indétermination de la raison d'un atome si celle-ci a déjà été capturée par le $OUT$ (ici sur `d(X1,3,Raison)`). La seconde clause du prédicat `v/6` devient alors :

```
v(X,Raison,IN,OUT,CONT,Borne) :-
  Borne<20, Borne_ is (Borne + 1) ->
  (member(d(X1,X,Raison),OUT);
    (member(d(X1,X,_),IN) *->
      fail ;
      \+ member(d(X1,X,Raison),OUT),
      d(X1,X,Raison,IN,OUT,CONT,Borne_)
    )
  );
  bornesetof(Raison).
```

Avec cette modification, le calcul des raisons retourne

```
Raisons = [[0],[2]]
```

ce qui est faux : non seulement le calcul d'indéterminisation sur `d(2,3,Raison)` a été éliminé comme dans la solution précédente mais aussi celui sur `d(4,3,Raison)` perdant les raisons $\{3\}$ et $\{0,1,3\}$.

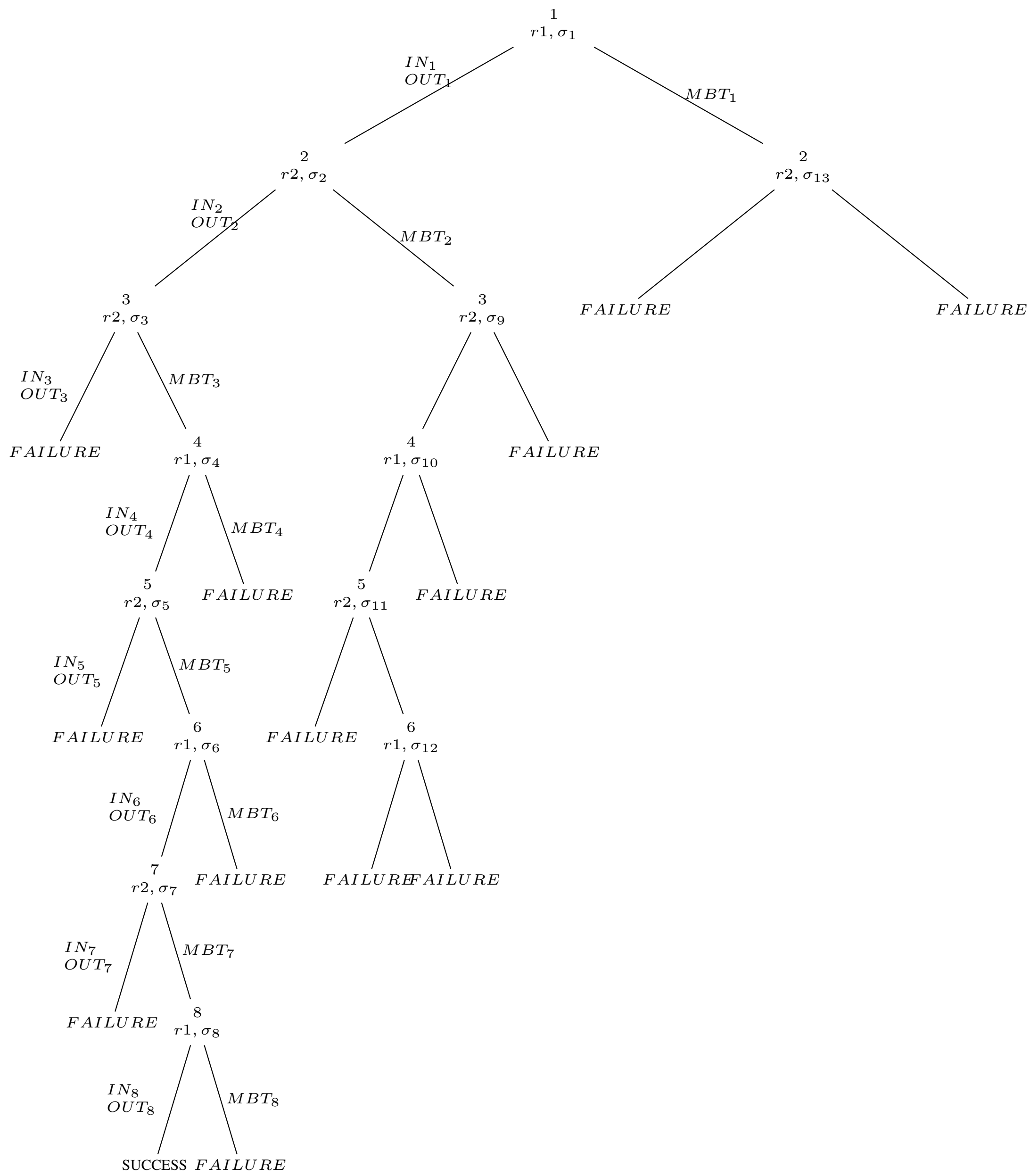
La solution d'un appel à `Prolog` à l'intérieur même d'`ASPeRiX` se révèle insuffisant pour le calcul des raisons d'indéterminisation mais l'approche en chaînage avant reste intéressante. Pour la rendre pertinente il faudrait prendre les mécanismes du chaînage avant de `Prolog` (comme par exemple une WAM) et les dédier au calcul des raisons.

### 4.3.4 Problème au niveau de la communication entre GNU-Prolog et ASPeRiX

`Prolog` est appelé en interne pour calculer en chaînage avant les raisons d'indéterminisation mais les paramètres de cet appel proviennent de la traduction en structure `GNU-Prolog` de la structure C d'`ASPeRiX` ce qui se révèle non seulement évidemment inefficace mais aussi problématique vis-à-vis de la gestion de la mémoire. Comme dans le paragraphe précédent, ce problème de communication entre `GNU-Prolog` et `ASPeRiX` plaide pour une intégration fine de la WAM (ou toute autre machine abstraite réalisant le chaînage avant) dans `ASPeRiX` pour en utiliser directement les structures.

## 5 Expérimentations

Dans cette partie, des résultats expérimentaux montrent l'intérêt du backjumping pour `ASPeRiX`. Ici, la version d'`ASPeRiX` avec backjumping est comparée à la version classique d'`ASPeRiX`.




$IN_0 = \{(init(1), \{0\}), (s(1), \{0\}), (s(2), \{0\}), (s(3), \{0\}), (s(4), \{0\}), (a(1,2), \{0\}), (a(2,3), \{0\}), (a(2,4), \{0\}), (a(3,1), \{0\}), (a(4,3), \{0\}), (v(1), \{0\})\}, OUT_0 = \emptyset, MBT_0 = \emptyset$

$\sigma_1 = [X1 = 1, X2 = 2], IN_1 = IN_0 \cup \{(v(2), \{0,1\}), (d(1,2), \{0,1\})\}, OUT_1 = OUT_0 \cup \{(h(1,2), \{1\})\}, MBT_1 = MBT_0 \cup \{(h(1,2), \{1\})\}$

$\sigma_2 = [X1 = 2, X2 = 3], IN_2 = IN_1 \cup \{(h(2,3), \{0,1,2\})\}, OUT_2 = OUT_1 \cup \{(d(2,3), \{2\})\}, MBT_2 = MBT_1 \cup \{(d(2,3), \{2\})\}$

$\sigma_3 = [X1 = 2, X2 = 4], IN_3 = IN_2 \cup \{(h(2,4), \{0,1,3\})\}, OUT_3 = OUT_2 \cup \{(d(2,4), \{3\})\}, MBT_3 = MBT_2 \cup \{(d(2,4), \{3\})\}$

$\sigma_4 = [X1 = 2, X2 = 4], IN_4 = IN_3 \cup \{(v(4), \{0,3\}), (d(2,4), \{3\})\}, OUT_4 = OUT_3 \cup \{(h(2,4), \{4\})\}, MBT_4 = MBT_3 \cup \{(h(2,4), \{4\})\}$

$\sigma_5 = [X1 = 4, X2 = 3], IN_5 = IN_4 \cup \{(h(4,3), \{0,3,5\})\}, OUT_5 = OUT_4 \cup \{(d(4,3), \{5\})\}, MBT_5 = MBT_4 \cup \{(d(4,3), \{5\})\}$

$\sigma_6 = [X1 = 4, X2 = 3], IN_6 = IN_5 \cup \{(v(3), \{0,5\}), (d(4,3), \{5\})\}, OUT_6 = OUT_5 \cup \{(h(4,3), \{6\})\}, MBT_6 = MBT_5 \cup \{(h(4,3), \{6\})\}$

$\sigma_7 = [X1 = 3, X2 = 1], IN_7 = IN_6 \cup \{(h(3,1), \{0,5,7\})\}, OUT_7 = OUT_6 \cup \{(d(3,1), \{7\})\}, MBT_7 = MBT_6 \cup \{(d(3,1), \{7\})\}$

$\sigma_8 = [X1 = 3, X2 = 1], IN_8 = IN_7 \cup \{(c, \{0,7\}), (d(3,1), \{7\})\}, OUT_8 = OUT_7 \cup \{(h(3,1), \{8\})\}, MBT_8 = MBT_7 \cup \{(h(3,1), \{8\})\}$

FIGURE 6.4 – Arbre de recherche pour le programme ASP du problème de circuit hamiltonien

Par ailleurs, étant donné que la version utilisant le backjumping ne propose aucune option d'optimisation dû à l'utilisation de structures de GNU-Prolog, elle sera également comparée à la version classique d'ASPeRiX sans optimisation. La comparaison avec d'autres solveurs existants est omise étant donné qu'ASPeRiX est lui même en un état de prototype et ne dispose pas à l'heure actuelle de structures autant optimisées que les autres solveurs.

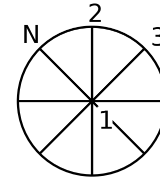
Dans chaque exemple, on calculera l'ensemble des answers sets afin d'observer si le backjumping permet de réduire le nombre de points de choix de l'arbre de recherche. La borne d'appels récursifs à des calculs de littéraux indéterminés de la traduction en clauses Prolog est fixée à 15.

Chaque test est réalisé à partir d'un ordinateur Intel Core i7-3520M avec 4 coeurs à 2.90GHz et environ 4GB de RAM sous Ubuntu Linux 12.04 64 bits. Pour chaque instance d'un problème, l'outil RunLim1.7 est utilisé pour limiter la mémoire à 1.000MB et le temps de calcul à 600 secondes.

La mention ND dans les tableaux de résultats indiquera que la mémoire utilisée n'a pas pu être déterminée[3]. La mention DT indiquera un dépassement du temps alloué pour le calcul des solutions au programme. La mention DM indiquera un dépassement de la mémoire allouée pour le calcul des solutions au programme. Enfin la mention PM indiquera un problème de mémoire lié à l'utilisation des structures GNU-Prolog.

**Problème de 3-coloration** Soit $P_{3col}$ le programme proposé dans [43] qui encode un problème de 3-coloration pour un graphe à $N$ sommets qui forme une roue de vélo. Dans ce programme, $s$ représente les *sommets*, $a$ les *arcs*, $c$ les *couleurs*, $col$ le fait qu'un sommet $s$ soit coloré par une couleur $c$ et $ncol$ le fait qu'un sommet $s$ ne soit pas coloré par une couleur $c$.

$$P_{3col} = \left\{ \begin{array}{l} s(1).,\ldots,s(N).,\ c(rouge).,\ c(bleu).,\ c(vert).,\\ a(1,2).,\ldots,a(1,N).,\\ a(2,3).,a(3,4).,\ldots,a(N,2).,\\ col(S,C) \leftarrow s(S),\ c(C),\ not\ ncol(S,C).,\\ ncol(S,C) \leftarrow col(S,D),\ c(C),\ C \neq D.,\\ \leftarrow a(S1,S2),\ col(S1,C),\ col(S2,C). \end{array} \right\}$$




Les résultats expérimentaux pour le problème de 3-coloration sont consignés dans le tableau 6.1. À travers ce problème de 3 coloration, on peut observer que la version classique d'ASPeRiX propose des résultats dans le temps imparti de 10 minutes jusqu'à 15 sommets mais pas au delà. Cela est dû au nombre de points de choix très conséquent en cas de simple backtracking. En revanche, la version utilisant le backjumping s'en sort nettement mieux et permet de trouver un résultat dans les 10 minutes imparties jusqu'à près de 200 sommets grâce à une baisse significative du nombre de points de choix. En effet, comme observé lors de l'exemple 35, la version avec backjumping va permettre en cas d'échec d'une branche de revenir à la raison pour laquelle deux sommets reliés par un arc ont la même couleur ou à la raison pour laquelle un sommet n'a pas pu être coloré. Cela permet d'éviter les nombreux backtrack de la version classique qui mènent plusieurs fois au même échec. D'un point de vue de la mémoire, le programme ne requiert pas énormément de mémoire mais la version avec backjumping demande légèrement plus d'espace.

**Problème de Schur** Le problème de Schur proposé dans [8] consiste en une partition de $N$ nombres dans $M$ ensembles telle que la condition suivante est respectée pour chaque ensemble : si $X$ et $Y$ appartiennent à l'ensemble alors $X+Y$ n'appartient pas à cet ensemble. Le problème $P_{Schur-4}$ présenté ici représente la partition de 4 nombres dans 3 ensembles.

[3]Cette situation intervient lorsque le temps d'exécution est très court (inférieur à 0,1 secondes).

| | | `ASPeRiX0.2.5` | `ASPeRiX_SansOpti` | `ASPeRiX_BJ` |
|---|---|---|---|---|
| $N = 5$ | temps en secondes | <0.1 | <0.1 | <0.1 |
| | mémoire en MB | ND | ND | ND |
| | nombre de points de choix | 225 | 225 | 55 |
| $N = 7$ | temps en secondes | <0.1 | 0.2 | <0.1 |
| | mémoire en MB | ND | 1.8 | ND |
| | nombre de points de choix | 1896 | 1896 | 129 |
| $N = 9$ | temps en secondes | 0.3 | 1.9 | <0.1 |
| | mémoire en MB | 1.6 | 1.8 | ND |
| | nombre de points de choix | 17625 | 17625 | 275 |
| $N = 11$ | temps en secondes | 3.5 | 18.9 | <0.1 |
| | mémoire en MB | 1.6 | 1.8 | ND |
| | nombre de points de choix | 174633 | 174633 | 307 |
| $N = 13$ | temps en secondes | 36 | 202 | 0.1 |
| | mémoire en MB | 1.6 | 1.8 | 4.2 |
| | nombre de points de choix | 1802412 | 1802412 | 444 |
| $N = 15$ | temps en secondes | 380 | DT | 0.1 |
| | mémoire en MB | 1.6 | - | 4.2 |
| | nombre de points de choix | 19063833 | - | 602 |
| $N = 17$ | temps en secondes | DT | - | 0.1 |
| | mémoire en MB | - | - | 4.2 |
| | nombre de points de choix | - | - | 784 |
| $N = 51$ | temps en secondes | - | - | 4 |
| | mémoire en MB | - | - | 10.7 |
| | nombre de points de choix | - | - | 7544 |
| $N = 101$ | temps en secondes | - | - | 48 |
| | mémoire en MB | - | - | 31 |
| | nombre de points de choix | - | - | 30088 |
| $N = 151$ | temps en secondes | - | - | 225 |
| | mémoire en MB | - | - | 65 |
| | nombre de points de choix | - | - | 67638 |
| $N = 191$ | temps en secondes | - | - | 559 |
| | mémoire en MB | - | - | 101 |
| | nombre de points de choix | - | - | 108478 |
| $N = 201$ | temps en secondes | - | - | DT |
| | mémoire en MB | - | - | - |
| | nombre de points de choix | - | - | - |

TABLE 6.1 – Résultats expérimentaux pour 3col

$$P_{Schur-4} = \left\{ \begin{array}{l} number(1).,\quad number(2).,\quad number(3).,\quad number(4).,\\ part(1).,\qquad part(2).,\qquad part(3).,\\ inpart(X,1) \leftarrow number(X),\ not\ inpart(X,2),\ not\ inpart(X,3).,\\ inpart(X,2) \leftarrow number(X),\ not\ inpart(X,1),\ not\ inpart(X,3).,\\ inpart(X,3) \leftarrow number(X),\ not\ inpart(X,1),\ not\ inpart(X,2).,\\ \leftarrow\ number(X),\ number(Y),\ part(P),\\ \quad inpart(X,P),\ inpart(Y,P),\ inpart(Z,P),\\ \quad T = Y+1,\ X < T,\ Z = X+Y. \end{array} \right\}$$

| | | | ASPeRiX0.2.5 | ASPeRiX_SansOpti | ASPeRiX_BJ |
|---|---|---|---|---|---|
| $N=1$ | $AS=3$ | temps en secondes | <0.1 | <0.1 | <0.1 |
| | | mémoire en MB | ND | ND | ND |
| | | nombre de points de choix | 3 | 3 | 3 |
| $N=2$ | $AS=6$ | temps en secondes | <0.1 | <0.1 | <0.1 |
| | | mémoire en MB | ND | ND | ND |
| | | nombre de points de choix | 12 | 12 | 12 |
| $N=3$ | $AS=18$ | temps en secondes | <0.1 | <0.1 | <0.1 |
| | | mémoire en MB | ND | ND | ND |
| | | nombre de points de choix | 51 | 51 | 46 |
| $N=4$ | $AS=30$ | temps en secondes | <0.1 | <0.1 | <0.1 |
| | | mémoire en MB | ND | ND | ND |
| | | nombre de points de choix | 162 | 162 | 131 |
| $N=5$ | $AS=66$ | temps en secondes | <0.1 | 0.1 | 0.1 |
| | | mémoire en MB | ND | 2 | 4.2 |
| | | nombre de points de choix | 597 | 597 | 383 |
| $N=6$ | $AS=120$ | temps en secondes | <0.1 | 0.4 | 0.2 |
| | | mémoire en MB | ND | 2 | 4.5 |
| | | nombre de points de choix | 1893 | 1893 | 897 |
| $N=7$ | $AS=258$ | temps en secondes | 0.2 | 1.6 | 0.6 |
| | | mémoire en MB | 2 | 2.1 | 6 |
| | | nombre de points de choix | 6831 | 6831 | 2328 |
| $N=8$ | $AS=288$ | temps en secondes | 0.9 | 5.5 | 1.3 |
| | | mémoire en MB | 1.9 | 2.1 | 7.8 |
| | | nombre de points de choix | 20202 | 20202 | 4491 |
| $N=9$ | $AS=546$ | temps en secondes | 3.5 | 21.7 | 3 |
| | | mémoire en MB | 1.9 | 2.1 | 12.5 |
| | | nombre de points de choix | 72825 | 72825 | 9808 |
| $N=10$ | $AS=300$ | temps en secondes | 11.2 | 69 | 4.4 |
| | | mémoire en MB | 1.9 | 2.1 | 15.2 |
| | | nombre de points de choix | 207102 | 207102 | 12569 |
| $N=11$ | $AS=186$ | temps en secondes | 40 | 252 | - |
| | | mémoire en MB | 2.2 | 2.3 | PM |
| | | nombre de points de choix | 700362 | 700362 | - |
| $N=12$ | $AS=114$ | temps en secondes | 128 | DT | - |
| | | mémoire en MB | 2.2 | - | PM |
| | | nombre de points de choix | 2044212 | - | - |
| $N=13$ | $AS=18$ | temps en secondes | 447 | DT | - |
| | | mémoire en MB | 2.2 | - | PM |
| | | nombre de points de choix | 6751332 | - | - |

TABLE 6.2 – Résultats expérimentaux pour Schur pour une partition de $N$ nombres dans 3 ensembles

Le tableau 6.2 présente les résultats expérimentaux pour le problème de Schur pour une partition de $N$ nombres dans 3 ensembles. Dans cet exemple, on observe que la version avec

backjumping est un peu plus lente que la version classique d'ASPeRiX pour les premières partitions. Cela est dû au manque d'optimisation de la version avec backjumping. On observe néanmoins que la version avec backjumping est plus rapide que la version classique sans optimisation grâce à un nombre de points de choix plus réduit. Ici, la raison pour laquelle deux nombres et la somme de ces 2 nombres appartiennent au même ensemble ou la raison pour laquelle un nombre ne peut appartenir à aucun ensemble permettent là encore de remonter dans l'arbre de recherche et ainsi d'éviter que cette situation se reproduise dans les branches avoisinantes. Du point de vue de la mémoire, la version avec backjumping croît davantage que la version classique compte-tenu du stockage des raisons, des nombreux appels à la primitive Prolog `setof` durant la recherche ainsi que la place en mémoire des listes $IN$, $OUT$ et $CONT$ nécessaire pour l'appel à cette primitive.

Pour la partition de 10 nombres dans 3 ensembles, la version avec backjumping réussit à être plus rapide que la version classique grâce à son nombre de points de choix revu considérablement à la baisse. Toutefois, la version avec backjumping ne propose pas les résultats du problème à partir de la partition des 11 premiers nombres à cause du trop grand nombre d'appels à des structures GNU-Prolog.

**N reines** Le problème des N reines consiste à placer $N$ reines sur un échiquier de taille $N$ sans qu'aucune de ces reines ne soit en conflit avec une autre. Cela signifie qu'il ne peut y avoir plus d'une reine par ligne, par colonne et par diagonale. Le programme suivant permet de calculer les solutions à ce problème pour $4$ reines.

$$P_{Nreines} = \left\{ \begin{array}{l} n(1).,\ldots,n(4).,\\ reine(X,Y) \leftarrow n(X),\ n(Y),\ not\ nreine(X,Y).,\\ nreine(X,Y) \leftarrow n(X),\ n(Y),\ not\ reine(X,Y).,\\ ligne(X) \leftarrow reine(X,Y).,\\ \leftarrow n(X),\ not\ ligne(X).,\\ \leftarrow reine(X,Y1),\ reine(X,Y2), Y1! = Y2.,\\ \leftarrow reine(X1,Y),\ reine(X2,Y), X1! = X2.,\\ \leftarrow reine(X1,Y1),\ reine(X2,Y2), X1! = X2, Y1! = Y2, X1+Y1 == X2+Y2.,\\ \leftarrow reine(X1,Y1),\ reine(X2,Y2), X1! = X2, Y1! = Y2, X1-Y1 == X2-Y2., \end{array} \right\}$$

| | | | ASPeRiX0.2.5 | ASPeRiX_SansOpti | ASPeRiX_BJ |
|---|---|---|---|---|---|
| $N=4$ | $AS=2$ | temps en secondes | <0.1 | 0.2 | 0.1 |
| | | mémoire en MB | ND | 1.9 | 4.2 |
| | | nombre de points de choix | 3091 | 3091 | 2107 |
| $N=5$ | $AS=10$ | temps en secondes | 0.4 | 2.1 | 0.8 |
| | | mémoire en MB | 1.7 | 1.9 | 8.1 |
| | | nombre de points de choix | 36314 | 36314 | 19122 |
| $N=6$ | $AS=4$ | temps en secondes | 4.9 | 23.6 | 6.3 |
| | | mémoire en MB | 1.7 | 1.9 | 38.5 |
| | | nombre de points de choix | 407259 | 407259 | 164802 |
| $N=7$ | $AS=40$ | temps en secondes | 61 | 290 | - |
| | | mémoire en MB | 1.7 | 1.9 | PM |
| | | nombre de points de choix | 5060840 | 5060840 | - |

TABLE 6.3 – Résultats expérimentaux pour Nreines

Les résultats expérimentaux pour le problème des Nreines sont présentés dans le tableau de résultats 6.3. Ici, le même phénomène est observé que pour l'exemple précédent. La version avec backjumping réduit le nombre de points de choix mais ne permet pas de résoudre le problème au delà de 6-reines à cause du trop grand nombre d'appels à des structures GNU-Prolog. On observe toutefois que la version classique d'ASPeRiX ne permet pas non plus de résoudre

ce problème pour un grand nombre de reines là encore dû à un nombre de points de choix très conséquent lors de simple backtrack. Au niveau du temps de calcul, la version avec backjumping est légèrement plus lente que la version classique. On observe également que son espace mémoire croît lorsqu'on augmente le nombre de reines dû là encore au stockage des raisons, aux nombreux appels à la primitive Prolog `setof` durant la recherche ainsi qu'à la place en mémoire des listes $IN$, $OUT$ et $CONT$ nécessaire pour l'appel à cette primitive.

**Problème des oiseaux** Le problème des oiseaux est un programme qui détermine si une sorte d'oiseaux vole ou non. Dans le programme ci dessous, $o$ représente un oiseau, $v$ le fait qu'un oiseau vole, $nv$ le fait qu'un oiseau ne vole pas, $a$ représente une autruche, $m$ représente un manchot et $sm$ un super manchot.

$$P_{vole} = \left\{ \begin{array}{lr} m(X) \leftarrow sm(X)., \quad o(X) \leftarrow m(X)., & o(X) \leftarrow a(X)., \\ v(X) \leftarrow o(X),\ not\ m(X),\ not\ a(X)., & v(X) \leftarrow sm(X)., \\ nv(X) \leftarrow m(X),\ not\ sm(X)., & nv(X) \leftarrow a(X). \end{array} \right\}$$

On ajoute à ce programme $N$ oiseaux dont $10\%$ d'autruches et $20\%$ de manchots dont la moitié sont des super manchots. Ce programme admet un seul answer set déterminé après une unique phase de propagation. Les résultats expérimentaux de ce programme sont répertoriés dans le tableau 6.4.

| | | ASPeRiX0.2.5 | ASPeRiX_SansOpti | ASPeRiX_BJ |
|---|---|---|---|---|
| $N = 100000$ | temps en secondes | 0.2 | 2 | 2.2 |
| | mémoire en MB | 41 | 42 | 84 |
| $N = 200000$ | temps en secondes | 0.5 | 4.3 | 4.6 |
| | mémoire en MB | 75 | 82 | 167 |
| $N = 300000$ | temps en secondes | 0.7 | 6.7 | 7.3 |
| | mémoire en MB | 121 | 123 | 248 |
| $N = 400000$ | temps en secondes | 1 | 9.1 | 9.8 |
| | mémoire en MB | 153 | 163 | 330 |
| $N = 500000$ | temps en secondes | 1.4 | 11.9 | 12.4 |
| | mémoire en MB | 197 | 204 | 412 |
| $N = 600000$ | temps en secondes | 1.7 | 14.1 | 15.2 |
| | mémoire en MB | 239 | 245 | 493 |
| $N = 700000$ | temps en secondes | 2 | 16.8 | 18 |
| | mémoire en MB | 278 | 286 | 577 |
| $N = 800000$ | temps en secondes | 2.4 | 19.2 | 20.3 |
| | mémoire en MB | 317 | 326 | 657 |
| $N = 900000$ | temps en secondes | 2.7 | 21.9 | 23.2 |
| | mémoire en MB | 358 | 366 | 738 |
| $N = 1000000$ | temps en secondes | 3 | 24.5 | 26 |
| | mémoire en MB | 399 | 406 | 820 |

TABLE 6.4 – Résultats expérimentaux pour Vole

Pour cet exemple, on constate que la mémoire utilisée par la version avec backjumping est environ deux fois supérieure à celle de la version classique d'ASPeRiX (avec ou sans optimisations). Cela est dû à la place en mémoire des listes $IN$, $OUT$ et $CONT$ réalisées à partir des structures GNU-Prolog. Le nombre élevé de littéraux montre également une augmentation légère du temps de calcul par rapport à la version classique d'ASPeRiX sans optimisations dû au traitement supplémentaire pour créer les listes Prolog.

**Circuit Hamiltonien** Soit $P_{CircuitHamiltonien}$ le programme, inspiré par celui présent dans [43], qui encode le problème de circuit hamiltonien dans un graphe complet orienté à $N$ som-

mets. Dans ce programme, $s$ représente les *sommets*, $a$ les *arcs*, $ch$ représente le fait qu'un arc entre 2 sommets *appartient au circuit hamiltonien*, $nch$ le fait qu'un arc entre 2 sommets *n'appartient pas au circuit hamiltonien*, $depart$ représente le *sommet de départ* (ici, le sommet de départ est le sommet 1) et $parcouru$ le fait qu'un *sommet a été parcouru*.

$$P_{CircuitHamiltonien} = \left\{ \begin{array}{l} depart(1)., \quad s(1)., \ldots, s(N)., \qquad a(X,Y) \leftarrow s(X),\ s(Y),\ X \neq Y., \\ ch(X,Y) \leftarrow a(X,Y),\ depart(X),\ not\, nch(X,Y)., \\ ch(X,Y) \leftarrow a(X,Y),\ parcouru(X),\ not\, nch(X,Y)., \\ nch(X,Y) \leftarrow a(X,Y),\ ch(X,Z),\ Y \neq Z., \\ nch(X,Y) \leftarrow a(X,Y),\ ch(Z,Y),\ X \neq Z., \\ parcouru(Y) \leftarrow ch(X,Y)., \\ \leftarrow s(X),\ not\, parcouru(X). \end{array} \right\}$$

| | | | ASPeRiX0.2.5 | ASPeRiX_SansOpti | ASPeRiX_BJ |
|---|---|---|---|---|---|
| $N=4$ | $AS=6$ | temps en secondes | <0.1 | <0.1 | <0.1 |
| | | mémoire en MB | ND | ND | ND |
| | | nombre de points de choix | 30 | 30 | 30 |
| $N=5$ | $AS=24$ | temps en secondes | <0.1 | <0.1 | 0.5 |
| | | mémoire en MB | ND | ND | 16 |
| | | nombre de points de choix | 128 | 128 | 128 |
| $N=6$ | $AS=120$ | temps en secondes | <0.1 | 0.1 | 4.9 |
| | | mémoire en MB | ND | 1.8 | 114 |
| | | nombre de points de choix | 650 | 650 | 650 |
| $N=7$ | $AS=720$ | temps en secondes | 0.1 | 0.6 | 45 |
| | | mémoire en MB | 1.7 | 1.8 | 752 |
| | | nombre de points de choix | 3912 | 3912 | 3912 |
| $N=8$ | $AS=5040$ | temps en secondes | 0.8 | 4.8 | - |
| | | mémoire en MB | 1.7 | 1.8 | DM |
| | | nombre de points de choix | 27398 | 27398 | - |

TABLE 6.5 – Résultats expérimentaux pour le programme $P_{CircuitHamiltonien}$

Dans cet exemple, la version avec backjumping ne permet pas de réduire le nombre de points de choix comme le montre le tableau de résultats 6.5. Toutefois, de nombreux appels à la primitive *setof* pour le calcul de littéraux indéterminés sont effectués dès lors qu'au moins un sommet n'a pas été parcouru avant de revenir au sommet de départ. En effet, lorsqu'un (ou plusieurs sommets) n'a pas été parcouru, la contrainte $(\leftarrow\ s(X),\ not\ parcouru(X).)$ est déclenchée et entraîne le calcul du littéral indéterminé $parcouru(s1)$ pour un sommet $s1$ non parcouru par le (faux) circuit hamiltonien. Le programme cherche alors pourquoi la règle $(parcouru(s1)\ \leftarrow\ ch(X,s1).)$ n'a pas été déclenchée. Cela l'amène à vérifier pourquoi les différentes instances de $ch(X,s1)$ sont indéterminés par le biais des règles $(ch(X,s1)\ \leftarrow a(X,s1), depart(X), not\, nch(X,s1).)$ et $(ch(X,s1) \leftarrow a(X,s1), parcouru(X), not\, nch(X,s1).)$.

Dans le cas où un autre sommet $s2$ n'a pas été parcouru, on recherchera alors pourquoi ce sommet n'a pas été parcouru à partir de la règle instanciée $(ch(s2,s1) \leftarrow a(s2,s1),\ parcouru(s2),\ not\, nch(s2,s1).)$. Or, celui-ci dépendra également du fait que le sommet $s1$ n'a pas été parcouru car le graphe est complet. Ainsi, des problèmes de boucles infinies font leur apparition et le calcul du littéral indéterminé se termine par un échec lorsque la borne fixée par la traduction en clauses `Prolog` est atteinte.

D'autres appels à la primitive setof interviennent pour le calcul de raison d'un littéral $MBT$ en bout de branche provoquant également des boucles infinies stoppées lorsque la borne est atteinte.

Ces nombreux appels à Prolog entraînent une hausse importante du temps de calcul et de la mémoire utilisée par la version d'ASPeRiX avec backjumping contrairement à la version classique d'ASPeRiX.

À travers ces quelques exemples, on peut constater que le backjumping proposé est utile pour des problèmes de décision où toutes les possibilités d'instanciations des règles non monotones ont été effectuées avant que l'échec ne se produise comme dans les exemples de coloration, du problème de Schur et des Nreines. En revanche, il apparaît inefficace lorsque les branches qui échouent sont liées à des littéraux indéterminés qui s'appellent mutuellement lors du calcul de leur raison.

Par ailleurs, les problèmes de mémoire liés à l'utilisation de structures GNU-Prolog dans la version actuelle d'ASPeRiX proposant le backjumping ainsi que son absence d'optimisation pénalisent nettement ses performances. Ces défauts pourrait être corrigés par une version intégrant directement une WAM qui permettrait également de corriger certaines limites de la traduction en clauses Prolog.

7

# Conclusion

Dans cette thèse, nous nous sommes intéressés aux justifications pour les solveurs ASP basés sur les règles. D'un point de vue théorique, nous avons étudié les justifications de plusieurs propriétés telles que la présence ou l'absence d'un littéral dans un answer set, ou la raison d'un échec lors de la recherche d'un answer set. Ce travail formel a permis de dégager des théorèmes sur les justifications avec leur preuve. Nous nous sommes particulièrement intéressés à la justification des échecs pendant le calcul des answer sets. Le concept principal est celui d'ensemble de blocage qui se caractérise par un sous-ensemble des règles appliquées, débloquées ou bloquées lors d'une computation et qui suffit à justifier l'échec du calcul. En pratique, connaître de telles justifications permet, en cas d'échec, de revenir directement à un point de l'arbre susceptible de remettre en cause l'ensemble de blocage et peut donc avoir des applications directes pour le backjumping. Ces résultats théoriques ont effectivement été mis en oeuvre d'un point de vue pratique avec l'implémentation du backjumping dans le solveur basé sur les règles `ASPeRiX`. Cette mise en oeuvre a été précédée d'une étude détaillée de ce solveur pour en décrire les algorithmes et d'une mise à jour du solveur pour lequel diverses corrections et extensions ont été implémentées. Les expérimentation faites ont montré que le backjumping pouvait réduire considérablement le nombre de points de choix parcourus dans l'arbre de recherche, même si l'implémentation, utilisant Prolog, n'a pas vocation à être efficace.

Ces travaux pourraient être poursuivis dans plusieurs directions. La première direction concerne l'implémentation proprement dite du solveur `ASPeRiX`. D'une part, l'implémentation actuelle est peu efficace et nécessiterait d'être améliorée afin que les techniques d'optimisation du parcours de l'arbre de recherche prennent tout leur sens. Il faudrait en particulier intégrer des techniques issues des bases de données pour la représentation et l'accès aux données. D'autre part, l'implémentation actuelle du backjumping souffre de quelques défauts qui n'ont pu être corrigés faute de temps.

Une suite naturelle de ce travail serait l'étude de l'apprentissage de lemmes pour les solveurs basés sur les règles : comment « compiler » un ensemble de blocage en des règles susceptibles d'élaguer l'arbre de recherche ? Une tentative préliminaire a été faite dans [55] où des règles au premier ordre (avec variables) sont apprises mais beaucoup de travail reste à réaliser dans ce cadre. En particulier, l'apprentissage consiste généralement à apprendre des contraintes (on apprend « ce qui ne va pas »). Or les contraintes sont mal exploitées dans les solveurs basés sur les règles : elles ne permettent pas de déduire de l'information mais seulement de vérifier

la cohérence d'une solution. L'apprentissage de contraintes risque donc d'être peu efficace. Il faudrait alors, soit apprendre des règles utilisables lors de la propagation, soit mieux gérer l'utilisation des contraintes pour en extraire des informations lors des étapes de propagation.

Le travail effectué sur les justifications peut avoir également d'autres applications telles que le débogage de programmes pour lequel l'approche par les règles nous semble bien adaptée, le fonctionnement du solveur pouvant être tracé plus facilement que dans le cas des solveurs basés sur les atomes. En effet, contrairement aux solveurs basés sur les atomes, le chemin permettant d'atteindre un answer set ou menant à un échec se ramène à la suite de règles effectivement déclenchées ou bloquées et les justifications sont les ensembles de règles impliquées dans la recherche. Il pourrait donc être plus aisé d'en tirer des informations pour corriger le programme. Par exemple, si le programme n'a pas d'answer set alors qu'il en était attendu, l'étude de régularités dans les ensembles de blocage pourrait permettre d'expliquer l'absence de solution, voire de suggérer des règles susceptibles d'être « fautives » dans cette absence de solution.

# Table des matières

**1 Introduction** **5**

**2 Answer Set Programming** **9**

**3 Solveurs ASP** **15**
1 Approche traditionnelle . . . 15
1.1 Phase d'instanciation (grounding) . . . 15
1.2 Phase de résolution . . . 16
1.3 Limites de cette approche . . . 19
2 Approche guidée par les règles . . . 21

**4 ASPeRiX** **23**
1 Aspect théorique . . . 23
2 Aspect algorithmique . . . 30
2.1 Déroulement de l'algorithme principal . . . 30
2.2 Les fonctions $\gamma$ . . . 34
2.3 $\gamma_{pro}$ . . . 36
2.4 $\gamma_{cho}$ . . . 37
2.5 $\gamma_{check}$ . . . 39
2.6 Instanciation d'une règle . . . 39
3 Langage d'ASPeRiX et extensions . . . 44
3.1 Syntaxe . . . 45
3.2 Respect d'ASP-Core2 et ordonnancement des termes . . . 46
3.3 Intégration des listes . . . 47

**5 Justifications des computations** **49**
1 Raison des littéraux et des règles . . . 50
1.1 Les raisons des littéraux et des contraintes . . . 50
1.2 Raison des littéraux indéterminés . . . 57
2 Ensemble compatible et ensemble de blocage . . . 59
3 Raison d'échec . . . 62
3.1 Computation bloquée . . . 62
3.2 Combinaison d'échec . . . 64
3.3 Propriétés des computations . . . 66

**6 Backjumping pour ASPeRiX** **81**
1 Limites du backtracking . . . 82
2 Les raisons . . . 84
2.1 Les raisons des littéraux et des contraintes . . . 84

2.2 Les raisons des échecs . . . . . 86
2.3 Les raisons des littéraux indéterminés . . . . . 87
3 Le backjumping . . . . . 88
3.1 Principe général . . . . . 88
3.2 Combinaison des échecs . . . . . 88
3.3 Lien avec les justifications . . . . . 89
4 Implémentation . . . . . 89
4.1 L'algorithme d'ASPeRiX . . . . . 90
4.2 Détails d'implémentation . . . . . 90
4.3 Limites de l'implémentation . . . . . 97
5 Expérimentations . . . . . 100

**7 Conclusion** **109**

# Bibliographie


[1] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP : A native ASP solver based on constraint learning. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, volume 8148 of *LNCS*, pages 55–67. Springer, 2013. 16

[2] Hassan Aït-Kaci. *Warren's abstract machine : a tutorial reconstruction*. Logic programming. Cambridge, Mass. MIT Press, 1991. 97

[3] Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with a-prolog. *Theory Pract. Log. Program.*, 3(4) :425–461, July 2003. 5

[4] Marcello Balduccini, Michael Gelfond, and Monica Nogueira. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 47(1) :183–219, 2006. 5

[5] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003. 5

[6] Keith L. Clark. Negation as failure. In Jack Minker, editor, *Logic and Data Bases*, volume 1, pages 293–322. Plenum Press, New York, London, 1978. 18

[7] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In P. Hill and D. Warren, editors, *Proceedings of the 25th International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*. Springer, 2009. 6

[8] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Gasp : Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3) :297–322, August 2009. 21, 102

[9] Carlos Viegas Damásio, João Moura, and Anastasia Analyti. Unifying justifications and debugging for answer-set programs. In Marina De Vos, Thomas Eiter, Yuliya Lierler, and Francesca Toni, editors, *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015.*, volume 1433 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015. 7, 49

[10] M. Dao-Tran, T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl. OMiGA : An open minded grounding on-the-fly answer set solver. In *Proceedings of the 13th European Conference on Logics in Artificial Intelligence (JELIA'12)*, volume 7519 of *LNAI*, pages 480–483. Springer, 2012. 6, 21

[11] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, July 1962. 17

[12] Daniel Diaz and Philippe Codognet. A minimal extension of the wam for clp(fd). In *Proceedings of the Tenth International Conference on Logic Programming on Logic Programming*, ICLP'93, pages 774–790, Cambridge, MA, USA, 1993. MIT Press. 21

[13] Thomas Eiter, Michael Fink, Peter Schüller, and Antonius Weinzierl. Finding explanations of inconsistency in multi-context systems. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning : Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press, 2010. 7, 49

[14] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming : A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and RenateA. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer Berlin Heidelberg, 2009. 16

[15] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12) :1495 – 1539, 2008. 5

[16] Wolfgang Faber, Nicola Leone, and Simona Perri. The intelligent grounder of DLV. In *Correct Reasoning*, pages 247–264. Springer, 2012. 15, 39

[17] Wolfgang Faber, Nicola Leone, Simona Perri, and Gerald Pfeifer. Efficient instantiation of disjunctive databases. Technical report, Tech. Rep. DBAI-TR-2001-44, TU Wien, Austria (November 2001), http ://www. dbai. tuwien. ac. at/local/reports/dbai-tr-2001-44. pdf, 2001. 15

[18] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pushing goal derivation in dlp computations. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, volume 1730 of *LNCS*, pages 177–191. Springer, 1999. 16, 26

[19] Charles L. Forgy. Expert systems. chapter Rete : A Fast Algorithm for the Many Pattern/-Many Object Pattern Match Problem, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990. 21

[20] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental asp solver. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 190–205, Berlin, Heidelberg, 2008. Springer-Verlag. 19

[21] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco : The Potsdam answer set solving collection. *Ai Communications*, 24(2) :107–124, 2011. 16

[22] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-Driven Answer Set Solving. In *IJCAI*, volume 7, pages 386–392, 2007. 5, 16

[23] Martin Gebser, Jörg Pührer, Torsten Schaub, and Hans Tompits. A meta-programming technique for debugging answer-set programs. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 448–453. AAAI Press, 2008. 7, 49

[24] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 266–271. Springer, 2007. 15

[25] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming (ICLP'88)*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press. 5, 12

[26] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4) :365–386, 1991. 5, 10, 14

[27] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4) :345–377, 2006. 18

[28] Keijo Heljanko and Ilkka Niemelä. Bounded ltl model checking with stable models. *Theory Pract. Log. Program.*, 3(4) :519–550, July 2003. 5

[29] Tomi Janhunen. Representing normal programs with clauses. In *In Proc. of the 16th European Conference on Artificial Intelligence*, pages 358–362. IOS Press, 2004. 18

[30] Kathrin Konczak, Thomas Linke, and Torsten Schaub. Graphs and colorings for answer set programming. *Theory and Practice of Logic Programming*, 6 :61–106, 1 2006. 13

[31] Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. ASPeRiX, a First Order Forward Chaining Approach for Answer Set Computing. *CoRR*, abs/1503.07717, 2015. 6, 21, 26, 29, 78

[32] Claire Lefèvre and Pascal Nicolas. A first order forward chaining approach for answer set computing. In *Logic Programming and Nonmonotonic Reasoning*, pages 196–208. Springer, 2009. 6, 21

[33] Claire Lefèvre and Pascal Nicolas. The first version of a new ASP solver : ASPeRiX. In *Logic Programming and Nonmonotonic Reasoning*, pages 522–527. Springer, 2009. 6, 21

[34] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3) :499–562, July 2006. 5, 16

[35] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models : Unfounded sets, fixpoint semantics, and computation. *Information and Computation*, 135(2) :69 – 112, 1997. 67

[36] V. Lifschitz. Answer set planning. In *International Conference on Logic Program- ming*, page 23–37, 1999. 5

[37] F. Lin and Y. Zhao. ASSAT : computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2) :115–137, 2004. 18

[38] L. Liu and M. Truszczynski. Pbmodels - software to compute stable models by pseudo-boolean solvers. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *LNCS*, pages 410–415. Springer, 2005. 18

[39] Lengning Liu, Enrico Pontelli, Tran Cao Son, and Miroslaw Truszczynski. Logic programs with abstract constraint atoms : The role of computations. *Artificial Intelligence*, 174(3-4) :295–315, 2010. 21, 23

[40] Marco Maratea, Francesco Ricca, Wolfgang Faber, and Nicola Leone. Look-back techniques and heuristics in DLV : Implementation, evaluation, and comparison to QBF solvers. *Journal of Algorithms*, 63(1-3) :70–89, January 2008. 16

[41] J.P. Marques-Silva and K.Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Confenrence on Computer-Aided Design (ICCAD'96)*, pages 220–227, 1996. 7, 81

[42] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4) :241–273, 1999. 5

[43] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4) :241–273, 1999. 20, 102, 106

[44] Simona Perri, Francesco Scarcello, Gelsomina Catalano, and Nicola Leone. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence*, 51(2-4) :195–228, December 2007. 39

[45] Enrico Pontelli, Tran Cao Son, and Omar El-Khatib. Justifications for logic programs under answer set semantics. *TPLP*, 9(1) :1–56, 2009. 7, 49

[46] R. Reiter. Readings in nonmonotonic reasoning. chapter A Logic for Default Reasoning, pages 68–93. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. 5

[47] Francesco Ricca, Wolfgang Faber, and Nicola Leone. A backjumping technique for disjunctive logic programming. *AI COMMUNICATIONS*, 19(2) :155, 2006. 7, 81

[48] Claudia Schulz and Francesca Toni. Justifying answer sets using argumentation. *TPLP*, 16(1) :59–110, 2016. 7, 49

[49] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2) :181–234, 2002. 5, 16

[50] T. Syrjänen. Implementation of local grounding for logic programs for stable model semantics. Technical report, Helsinki University of Technology, 1998. 15

[51] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *Logic Programming and Nonmotonic Reasoning*, pages 434–438. Springer, 2001. 16

[52] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989. 39

[53] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)*, 38(3) :619–649, 1991. 21, 66

[54] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. *Logic Programming and Nonmonotonic Reasoning : 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, chapter Justifications for Logic Programming, pages 530–542. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 7, 49

[55] Antonius Weinzierl. Learning non-ground rules for answer-set solving. In *2nd Workshop on Grounding and Transformations for Theories With Variables (GTTV'13)*, 2013. 21, 109

UNIVERSITE BRETAGNE LOIRE

université angers

# Thèse de Doctorat


## Christopher BÉATRIX


### Justifications dans les approches ASP basées sur les règles

Application au backjumping dans le solveur ASPeRiX

### Justifications in rule-based ASP computations

Application to backjumping in the ASPeRiX solver


**Résumé**

L'Answer Set Programming (ASP) est un formalisme capable de représenter des connaissances en Intelligence Artificielle à l'aide d'un programme logique au premier ordre pouvant contenir des négations par défaut. En quelques années, plusieurs solveurs performants ont été proposés pour calculer les solutions d'un programme ASP que l'on nomme answer sets.
Nous nous intéressons ici plus particulièrement au solveur ASPeRiX qui instancie les règles au premier ordre à la volée durant le calcul des answer sets. Pour réaliser cela, ASPeRiX applique un chaînage avant sur les règles à partir de littéraux précédemment déterminés.
L'étude de ce solveur nous amène notamment à considérer la notion de justification dans le cadre d'une approche de calcul d'answer sets basée sur les règles. Les justifications permettent d'expliquer pourquoi certaines propriétés sont vérifiées. Parmi celles-ci, nous nous concentrons particulièrement sur les raisons d'échecs qui justifient pourquoi certaines branches de l'arbre de recherche n'aboutissent pas à un answer set.
Cela nous conduit à implémenter une version d'ASPeRiX proposant du backjumping qui évite de parcourir systématiquement toutes les branches de l'arbre de recherche grâce aux informations fournies par les raisons d'échecs.



**Mots clés**

Logique du premier ordre, Answer Set Programming, Solveurs, Approche guidée par les règles, Justifications, Backjumping.



**Abstract**

Answer set programming (ASP) is a formalism able to represent knowledge in Artificial Intelligence thanks to a first order logic program which can contain default negations. In recent years, several efficient solvers have been proposed to compute the solutions of an ASP program called answer sets.
We are particularly interested in the ASPeRiX solver that instantiates the first order rules on the fly during the computation of answer sets. It applies a forward chaining of rules from literals previously determined.
The study of this solver leads us to consider the concept of justification as part of a rule-based approach for computing answer sets. Justifications enable to explain why some properties are true or false. Among them, we focus particularly on the failure reasons which justify why some branches of the search tree does not result in an answer set. This encourages us to implement a version of ASPeRiX with backjumping in order to jump to the last choice point related to the failure in the search tree thanks to information provided by the failure reasons.



**Key Words**

First order logic, Answer Set Programming, Solvers, Rule-based approach, Justifications, Backjumping.