Une Sémantique Trivaluée pour un Langage de Règles de Mise à jour

Nicole Bidoit – Sofian Maabout

LIPN. CNRS URA1507. Université de Paris XIII. Institut Galilée.
Avenue J.B. Clément. 93430 Villetaneuse.

(Ce travail a été partiellement subventionné par le PRC : “Bases de Données”, Pôle: Gestion de la Dynamique.)

RÉSUMÉ. La sémantique des programmes de règles actives est définie généralement sous forme de modèle d'exécution. La complexité de ces modèles d'exécution se traduit par l'impossibilité de garantir, en général, la terminaison et la confluence des programmes. Nous proposons, pour une classe restreinte de règles actives appelées règles de mise à jour, une sémantique déclarative basée sur la sémantique bien fondée introduite en programmation logique. Le langage de règles de mise à jour ainsi défini est utilisé pour maintenir la cohérence d'une base de données en présence de contraintes statiques et de contraintes de transition. Les propriétés de changement minimal, de connaissance maximale et de stabilité sont examinées dans ce cadre.

ABSTRACT. The semantics of active rules is generally defined in terms of execution models. The complexity of these models makes properties like termination and confluence hard to decide. We propose for a restricted class of active rules called update rules a declarative semantics based on the well founded semantics of logic programs. We investigate the use of update rules for maintaining database consistency by considering both static and transition constraints. Classical properties such that minimal change, maximal knowledge and stability are investigated in this framework.

MOTS CLÉS : Mise à jour, sémantique bien fondée, contraintes d'intégrité.
KEY WORDS : Update, well founded semantics, integrity constraints.

1 Introduction

Les bases de données actives désignent des systèmes de gestion de bases de données offrant des langages de manipulation de données très particuliers. Ces langages de manipulation de données, appelés langages de règles actives de par leur proximité avec les langages de règles de production, sont destinés à permettre la détection d'événements, syntaxiques ou sémantiques (requête, mise à jour, horloge, commande système), qui vont déclencher l'exécution d'actions (mise à jour, requête, validation, retour arrière, message, etc.). Les programmes de règles actives ne sont donc pas directement destinés à l'utilisateur mais servent à "espionner" et à réagir à l'exécution des programmes d'application et des transactions lancés par l'utilisateur. Cette esquisse de description des bases de données actives laisse aisément entrevoir la complexité des langages de règles qui sont généralement présentés sous forme de règles ECA, (Evénement, Condition, Action).

L'engouement récent pour les langages ECA a été dominé par le développement de prototypes et de systèmes commercialisés (voir [WC96] pour un état de l'art). L'étude des concepts fondamentaux permettant de valider ces systèmes est moins riche. Ce type d'étude est rendu difficile, et même biaisé d'une certaine façon, parce que mené à posteriori en prenant pour référence les systèmes existants qui sont d'une grande diversité. Parmi ces travaux [FT95, Wid92, SAV95, PV95, PV97, CC95], l'étude de Picouet & Vianu [PV95] figure parmi les plus courageuses et les plus complètes. Elle permet une comparaison des puissances d'expression des prototypes existants.

Les descriptions des prototypes offrent en général une définition purement opérationnelle de la sémantique des langages ECA qu'ils implémentent sous forme de *modèle d'exécution*. Ces modèles d'exécution diffèrent les uns des autres par un nombre important de critères : spécification/expression des règles, détection des événements, évaluation des conditions, couplage de la détection des événements avec l'évaluation des conditions et/ou l'exécution des actions, exécution ensembliste ou par instance des règles, exécution parallèle ou séquentielle des règles, ordre pré-défini sur les règles pour gérer l'exécution en cascade, etc.

Parmi les nombreux problèmes posés par la définition de ces langages, nous pouvons citer : la non terminaison de l'exécution de règles, le non-déterminisme dû à l'ordre d'exécution des règles et les mises à jour conflictuelles. Ces problèmes sont discutés dans [SAV95, AHW95, BCW93, AS91].

L'"argument de vente" utilisé par les promoteurs de ces systèmes actifs s'articule autour des nombreuses applications qui peuvent bénéficier de ce nouveau mode de programmation. Cet argument (les applications) est pourtant actuellement remis en cause de par l'impraticabilité des systèmes existants. Par exemple, l'une des applications ciblées est la maintenance des contraintes d'intégrité : les règles actives sont destinées à réagir aux mises à jour et sont chargées d'effectuer les actions nécessaires pour rétablir la cohérence de la base de données. Dans l'état actuel des prototypes existants, en dehors de cas triviaux, il est reconnu qu'il est pratiquement impossible de prédire l'effet des règles sur les transactions, il est donc impossible de garantir la fiabilité des règles écrites pour maintenir la cohérence de la base de données.

Une règle de mise à jour (m.à.j) est un cas restreint de règle ECA : les événements sont des mises à jour, les conditions sont des requêtes sur la base de données et l'action est une mise à jour. Intuitivement, à partir d'une mise à jour initiale soumise à la base de données, un programme de règles de mise à jour sert à déduire d'autres mises à jour.

La synergie entre programmation logique et bases de données a été extrêmement productive dans le cadre de la définition des systèmes de bases de données déductives. Elle a en particulier offert un cadre formel d'investigation pour les définitions des sémantiques déclaratives et opérationnelles de langages de requêtes récursives tel que Datalog$^\neg$ [Bid91, Ull89]. D'autre part, les liens entre les bases de données déductives, la programmation logique et le raisonnement non monotone [Rei80, McC80, Moo85, MT93] sont bien établis et différentes modélisations ont été mises en œuvre, sous des formes simplifiées, pour définir la sémantique de Datalog$^\neg$. La mise à jour d'une base de données, ou plus généralement, l'évolution d'une base de données comporte intrinsèquement un caractère non monotone portant sur les informations déductibles de la base de données. Tout naturellement, la communauté de programmation logique s'intéresse actuellement à l'introduction des règles de m.à.j [MT95, RL96, Bar94, MW88, Fit95, ALS95, PT95, SVV92, BK94].

Néanmoins, les études présentées dans la littérature se prêtent à un certain nombre de critiques : les règles de mises à jour considérées ne permettent de traiter que des applications "académiques", la sémantique proposée, si elle présente l'avantage d'être déclarative, n'est pas toujours couplée à une sémantique opérationnelle ou alors au prix de restrictions sévères. Enfin, le problème de la production de mises à jour contradictoires est résolu de façon drastique par le rejet en bloc de toutes les mises à jour.

Le but de cet article est de proposer un langage de règles de mises à jour. Le langage proposé est, syntaxiquement parlant, identique à Datalog$^\neg$. Ceci induit la prise en compte d'"événements négatifs" dans le corps des règles de mises à jour. Intuitivement, la négation d'un événement est détectée si l'événement n'a pas lieu. Un programme de mise à jour étant un programme logique, la sémantique bien fondée [vGRS91] est étudiée pour définir la dérivation de mises à jour à partir d'une mise à jour initiale, d'une instance de la base de données en présence de règles de mise à jour. Bien que nous n'abordons pas dans cet article l'aspect opérationnel, le choix de la sémantique bien fondée garantit l'existence d'algorithmes d'évaluation polynômiaux. Une validation du travail a été amorcée en utilisant le prototype de [SSW94] implémentant une méthode d'évaluation en chaînage arrière couplée avec des techniques de tabulation (memoing technics).

Dans le but d'essayer d'apporter une réponse plus "souple" au problème des m.à.j contradictoires, nous choisissons de considérer des bases de données incomplètes. Intuitivement, la présence de faits inconnus dans la base de données est interprétée de façon inhabituelle dans notre approche, par l'occurrence au cours de l'évolution de la base de mises à jour contradictoires concernant ces faits.

Préliminairement à la présentation de notre langage, deux groupes de propriétés sont examinées indépendamment de toute sémantique particulière et servent ultérieurement à évaluer notre approche. Les premières propriétés portent sur les mises à jour déduites à partir d'un programme et nous considérons que toute sémantique de règles de mises à jour doit satisfaire ces propriétés. Le deuxième groupe de propriétés a pour cible les transitions induites par un programme de mise à jour. Une transition est caractérisée par un couple d'instances : l'instance initiale et l'instance résultat de la mise à jour. Contrairement aux premières propriétés, celles-ci ne sont pas requises a priori. C'est le contexte de l'application qui déterminera si certaines de ces propriétés doivent être vérifiées ou non.

L'exposé de notre langage sera poursuivi par l'examen des propriétés de la sémantique que nous proposons dans le cadre de l'utilisation des règles de mise à jour pour maintenir des contraintes d'intégrité. A la fin de cet article, notre approche est comparée au travail de Marek & Truszczyńsky [MT95] proposant un langage de règles de révision.

Certains résultats présentés ici ont été publiés dans [BM96, BM97]. Les preuves sont disponibles dans [Maa97].

1.1 Exemples

Les exemples suivants introduisent quelques problèmes bien connus dans le contexte des règles de m.à.j.

Exemple 1.1 Soit la base $\Delta = \{p, q\}$ à laquelle est associé le programme suivant :

$$\mathcal{P}_1 = \left\{ \begin{array}{llcl} \rho_1 : & +s, p & \longrightarrow & +r \\ \rho_2 : & +t, q & \longrightarrow & -r \end{array} \right\}$$

Intuitivement, ρ_1 exprime que lorsque s est inséré, si p est dans la base initiale alors r doit être inséré. Supposons maintenant qu'un utilisateur veuille insérer s et t dans Δ. Nous avons donc la m.à.j utilisateur $\delta = \{+s, +t\}$. Puisque les prémisses des deux règles sont vérifiées, les deux règles ρ_1 et ρ_2 sont “exécutables” simultanément. L'exécution en parallèle des deux règles génère un conflit de m.à.j : ρ_1 doit insérer r alors que ρ_2 doit le supprimer. La question est donc : que faut-il faire à propos de r ?

Examinons maintenant ce qu'il advient lorsque les règles sont exécutées séquentiellement, par exemple considérons le cas où ρ_1 est exécutée avant ρ_2. Son action consiste à insérer r. A ce stade, ρ_2 est toujours exécutable et son action supprime r. Les prémisses de ρ_1 restant valides, elle doit donc être exécutée une fois de plus, et ainsi de suite. L'exécution séquentielle des deux règles ne termine pas.

Suivant notre sémantique, cet exemple est traité en considérant r *inconnu* dans la nouvelle instance de la base. □

Exemple 1.2 Considérons la base vide Δ, la m.à.j $\delta = \{+s, +t\}$, et les règles

$$\mathcal{P}_2 = \left\{ \begin{array}{llcl} \rho_1 : & +s & \longrightarrow & -t \\ \rho_2 : & +t & \longrightarrow & -s \end{array} \right\}$$

L'application naïve de la m.à.j δ sur la base vide Δ produit une nouvelle base $\Delta' = \{s, t\}$. Remarquons que les règles ρ_1 et ρ_2 sont toutes les deux exécutables. Si ρ_1 et ρ_2 sont exécutées simultanément à partir de Δ', alors la nouvelle base est vide, i.e. on revient à la base initiale Δ. Le fait t (resp. le fait s) est supprimé, en exécutant ρ_1 (resp. ρ_2), car s (resp. t) est inséré. Finalement, la *justification* de la suppression de t, i.e. l'insertion de s, est perdue dans la base finale (s n'est pas inséré). Ainsi, nous dirons que la suppression de t n'est pas justifiée. La même remarque s'applique à la suppression de s.

L'exécution de ρ_1 seule supprime t et donc ensuite ρ_2 n'est plus exécutable puisque t n'est plus inséré. Ainsi, la base résultante contient s mais pas t. Symétriquement, si ρ_2 est exécutée en premier, alors la base finale contient t mais pas s. Nous sommes donc en présence d'un cas de non confluence : l'état final dépend de l'ordre d'exécution des règles. □

2 Préliminaires

Dans ce qui suit, nous utiliserons la terminologie de la programmation logique. Nous considérons un ensemble de variables $\mathcal{V}ar = \{x, y, z, \dots\}$, un ensemble non vide de constantes $\mathcal{C}onst = \{a, b, c, \dots\}$ et un ensemble de prédicats de base $\Pi_b = \{P, Q, R \dots\}$. Π_b est subdivisé en deux ensembles distincts : les prédicats intensionnels et les prédicats exentsionnels. A chaque prédicat extensionnel P de Π_b, sont associés deux prédicats, $+P$ et $-P$ de même arité que P appelés prédicat de m.à.j. $+P$ est un +prédicat (prédicat d'insertion) et $-P$ est un $-$prédicat (prédicat de suppression). Les définitions d'atome, littéral et fait sont les définitions usuelles. Un littéral de m.à.j est un littéral construit à partir d'un prédicat de m.à.j, de même qu'un atome et un fait de m.à.j. Si I est un ensemble de littéraux, alors $\neg.I$ désigne l'ensemble $\{\neg\ell : \ell \in I\}$.

Définition 1 (Règles) *La syntaxe d'une règle ρ est :*

$$\rho \ : \ E_1, \dots, E_n, \ L_1, \dots, \ L_m \ \longrightarrow \ A \qquad (1)$$

où $n \geq 0, m \geq 0$, les E_i sont des littéraux de m.à.j, les L_i des littéraux de base et A un atome. ρ est une règle de m.à.j ssi A est un atome de m.à.j. Sinon, ρ est dite règle déductive. Nous notons :

$$\begin{array}{ll} Event(\rho) = \{E_1, \dots, E_i\} & Cond(\rho) = \{L_1, \dots, \ L_m\} \\ Corps(\rho) = Event(\rho) \cup Cond(\rho) & T\hat{e}te(\rho) = \{A\} \end{array}$$

Un programme $\mathcal{P}$ est un ensemble fini de règles.

Exemple 2.1 L'ensemble des règles suivant constitue un programme :

$$\left\{ \begin{array}{lll} \rho_1 : & +Emp(x,y), \neg Emp(x,y) & \longrightarrow \quad Nouv\_Emp(x,y) \\ \rho_2 : & Nouv\_Emp(x,y), \neg Dept(y), \neg{+}Dept(y) & \longrightarrow \quad +Prob\_Dept\_Emp(x) \end{array} \right\}$$

La signification intuitive de ρ_1 est : si un nuplet (x, y) est inséré dans la relation (prédicat) *Emp* et si ce même nuplet n'existait pas déjà dans cette relation, alors <u>*déduire*</u> que x est un nouvel employé du département y. Remarquons que ρ_1 est une règle *déductive*.

La lecture intuitive de ρ_2 est : si (i) x est un nouvel employé du département y, (ii) le nuplet y n'existe pas dans la relation *Dept* et (iii) il n'est pas non plus inséré dans

$Dept$, alors l'<u>*insérer*</u> dans la relation $Prob\_Dept\_Emp$. Remarquons que ρ_2 est une règle de m.à.j.

Comme indiqué dans l'introduction, la *négation* d'un événement est satisfaite si il n'y a pas occurrence de cet événement. Ainsi, dans ρ_2, $\neg{+}Dept(y)$ signifie : l'insertion de (y) dans $Dept$ n'est pas requise. □

Une interprétation partielle I est un ensemble de littéraux clos tel que, si $pos(I)$ et $neg(I)$ représentent respectivement l'ensemble des littéraux positifs et négatifs de I, alors $pos(I) \cap \neg.neg(I) = \emptyset$. Une interprétation I satisfait un littéral clos ℓ (noté $I \models \ell$) ssi $\ell \in I$. De même, I satisfait une règle close ρ (noté $I \models \rho$) ssi l'implication suivante est vraie : $Corps(\rho) \subseteq I \Rightarrow T\hat{e}te(\rho) \subseteq I$. $Inst(\mathcal{P})$ désigne l'ensemble de toutes les instanciations possibles des règles de $\mathcal{P}$ en utilisant les constantes qui apparaissent dans $\mathcal{P}$. I est un modèle de $\mathcal{P}$ ssi I satisfait toutes les règles de $Inst(\mathcal{P})$.

Modèle bien fondé Nous présentons brièvement ci-dessous la *sémantique bien fondée* des programmes Datalog$^\neg$. Le lecteur pourra se référer à [vGRS91] pour une présentation plus détaillée. Notons ici que la syntaxe des programmes de m.à.j est identique à celle des programmes Datalog$^\neg$.

La sémantique d'un programme Datalog $\mathcal{P}$ est définie par l'intermédiaire d'un opérateur de point fixe WF, qui se compose lui-même de deux opérateurs, $T^{\in}$ (opérateur de conséquence fondée) et NF (opérateur de conséquence non fondée). Soit une interprétation partielle I.

- $T^{\in}_{\mathcal{P}}(I) = \{T\hat{e}te(\rho) | \rho \in Inst(\mathcal{P})$ et $\forall \ell \in Corps(\rho) : \ell \in I\}$
- Un ensemble de faits U est un *ensemble de faits non-fondés*, respectivement à I, ssi :
  $\forall f \in U, \forall \rho \in Inst(\mathcal{P}) : T\hat{e}te(\rho) = f \quad \Rightarrow \exists \ell \in Corps(\rho)$, tel que $\neg \ell \in I$ ou $\ell \in U$.
  $NF_{\mathcal{P}}(I)$ est défini comme étant le plus grand ensemble de faits non fondés respectivement à I.

Considérons maintenant la séquence d'interprétations $(I_i)_{(i \geq 0)}$ définie comme suit:

$$\begin{aligned} I_0 &= \emptyset \\ I_{i+1} &= WF_{\mathcal{P}}(I_i) = T^{\in}_{\mathcal{P}}(I_i) \cup \neg.NF_{\mathcal{P}}(I_i) \end{aligned}$$

[vGRS91] montre que la suite $I_{(i \geq 0)}$ est monotone donc, l'ensemble des constantes étant fini, il existe un entier k tel que $I_k = I_i$ pour tout $i \geq k$. Nous notons ce point fixe par $WF(\mathcal{P})$ qui est, par définition, le modèle bien fondé de $\mathcal{P}$. □

Les bases de données considérées dans ce travail sont incomplètes. Leur définition utilise la logique trivaluée [Fit85] de façon très élémentaire.

Nous spécifions une instance incomplète de base de données par le couple des faits vrais et des faits inconnus. Cette représentation est une alternative de la représentation classique qui considère les parties positives et négatives.

Définition 2 (Base de données) *Une instance de base de données* Δ *est un couple* $< V(\Delta), I(\Delta) >$ *où :* $V(\Delta)$ *et* $I(\Delta)$ *sont deux ensembles disjoints de faits. Les faits* *de $V(\Delta)$ (resp. $I(\Delta)$) sont les faits vrais (resp. inconnus) de la base. L'ensemble des faits qui ne sont ni dans $I(\Delta)$ ni dans $V(\Delta)$ représente l'ensemble des faits faux et il est noté par $F(\Delta)$. Δ est totale (ou bivaluée) ssi $I(\Delta) = \emptyset$. Si $\mathcal{B}$ est l'ensemble de tous les faits que l'on peut construire à partir des prédicats de Π_b et des constantes dans $\mathcal{C}onst$, alors $F(\Delta) = \mathcal{B} \backslash [V(\Delta) \cup I(\Delta)]$.*

Intuitivement, $I(\Delta)$ représente l'ensemble des faits pour lesquels des m.à.j contradictoires ont eu lieu au cours de l'évolution de la base de données. Nous verrons ultérieurement comment cet ensemble est géré.

3 Mises à jour

Une m.à.j représente une *spécification partielle* d'un nouvel état de la base [Che91]. Si l'utilisateur insère p, ceci signifie qu'il veut faire transiter la base vers un état où p est *vrai*. Dans le cadre du modèle relationnel, l'insertion est réalisée tout simplement en ajoutant p à l'état initial pour construire le nouvel état. Le contenu de la base initiale (en dehors de p) est "préservé par défaut". En présence de règles de m.à.j, la construction du nouvel état de la base n'est pas aussi simple. En effet, ces règles sont utilisées pour dériver d'autres mises à jour que celles proposées par l'utilisateur et/ou en annuler une partie. Par exemple, si l'utilisateur veut insérer p dans une base qui contient les faits q et r en présence de la règle $\rho : \; +p, t \longrightarrow -q$, alors le nouvel état est obtenu en insérant non seulement p mais aussi en supprimant q. La suppression de q est une *m.à.j dérivée*.

De façon à clarifier l'exposé, nous allons commencer par décrire le processus général de mise à jour d'une base de données en présence de règles de mise à jour. Ceci est fait indépendamment de toute sémantique et servira à introduire les propriétés portant sur la dérivation de mises à jour d'une part et sur les instances mises à jour d'autre part.

Le mécanisme de mise à jour commence par l'acquisition d'un ensemble de mises à jour initiales (insertions et suppressions) auprès de l'utilisateur ou à partir de l'exécution d'un programme d'application. Cet ensemble d'insertions et de suppressions peut être vu comme l'*effet net* de l'exécution séquentielle d'une transaction lancée par un utilisateur. Dans la suite, cet ensemble de m.à.j. est appelé m.à.j. utilisateur. Ensuite, étant données une instance de la base et une m.à.j. utilisateur, les règles sont utilisées pour produire un ensemble de m.à.j. dites dérivées. L'application qui, à partir d'un état de la base, d'un programme de règles et d'une m.à.j utilisateur rend un nouvel ensemble de m.à.j. est noté `Up_Der`. L'effet de la m.à.j. utilisateur sur l'état initial de la base est obtenu par application des m.à.j. dérivées produisant ainsi une nouvelle instance. `Apply` désigne l'application qui prend en entrée une instance et un ensemble de m.à.j. et rend une instance de la base de données. L'application définissant la sémantique des programmes de mise à jour est notée `Up_Sem`, elle prend en entrée une instance de la base de données, une m.à.j. utilisateur et un programme de règles de m.à.j. et rend une instance de la base de données. L'application `Up_Sem` est définie par composition des applications `Up_Der` et `Apply`. Le schéma suivant illustre le processus de mise à jour.

$$
\begin{array}{lcccccc}
 & & & \text{m.à.j dérivées} & & \text{instance} \\
 & & & \downarrow & & \downarrow \\
\left.\begin{array}{ll} \text{instance} & \Delta \\ \text{m.à.j utilisateur} & \delta \\ \text{programme de règles} & \mathcal{P} \end{array}\right\} & & \xrightarrow{\texttt{Up\_Der}} & \delta' & \xrightarrow{\texttt{Apply}} & \Delta' \\
 & & & & & \\
 & & & \texttt{Up\_Sem} & &
\end{array}
$$

Fig1. Processus de mise à jour

3.1 Propriétés des m.à.j dérivées

Nous introduisons dans cette section certaines propriétés des m.à.j dérivées. La première de ces propriétés exprime tout simplement le fait que les m.à.j dérivées ne doivent pas être contradictoires.

Définition 3 (Mise à jour) *Une m.à.j est une interprétation δ qui ne contient que des littéraux de m.à.j clos. δ^+ (resp. δ^-) représente l'ensemble des +littéraux (resp. $-$littéraux) de δ. δ est* non contradictoire *si $pos(\delta^+) \cap pos(\delta^-) = \emptyset$. Une m.à.j utilisateur est une m.à.j. δ non contradictoire telle que $neg(\delta) = \emptyset$.*

L'hypothèse faite qu'une m.à.j. utilisateur n'est pas contradictoire est compatible avec ce que nous avons dit précédemment. En effet, si la m.à.j utilisateur est considérée comme étant l'effet net d'une transaction, alors elle ne peut pas être contradictoire : toutes les m.à.j d'une transaction sont "exécutées" d'une manière séquentielle. Ainsi, une insertion suivie d'une suppression du même nuplet a un effet net qui est nul. De même, la suppression d'un nuplet (qui existe) suivie de l'insertion du même nuplet n'a pas d'effet sur la base. Dans la suite, nous exigeons que $\texttt{Up\_Der}(\Delta, \delta, \mathcal{P})$ soit non contradictoire.

La deuxième propriété présentée ici exprime que les m.à.j dérivées doivent être *bien fondées* : une insertion (resp. suppression) d'un fait n'est réalisable que si la "preuve" de la non occurrence de la suppression (resp. insertion) peut être faite.

Définition 4 (M.à.j bien fondée) *Une m.à.j δ est bien fondée ssi pour chaque atome de m.à.j clos $+p$ (resp. $-p$) dans δ, $\neg{-p}$ (resp. $\neg{+p}$) appartient à δ.*

Nous requierons dans la suite que `Up_Der` soit définie de sorte que $\texttt{Up\_Der}(\Delta, \delta, \mathcal{P})$ soit bien fondée.

Le *changement durable*(en anglais : *durable change*) a été introduit dans [Zan95] pour exprimer que tout événement qui en déclenchant une règle induit des m.à.j dérivées ne doit pas être invalidée au cours de la m.à.j de la base. En d'autres termes, toute m.à.j déduite doit être justifiée par l'état final de la base (c.f. Exemple 2).

Cette propriété a été introduite en programmation logique sous différentes variantes : interprétation *supportée* [ABW88], interprétation *justifiée* [BH89]. Nous adoptons ici la notion d'interprétation *bien supportée* due à [Fag91].

Définition 5 (Bien-Supportée) *Soit $\mathcal{P}$ un programme logique et I une interprétation. I est* bien supportée *par $\mathcal{P}$ si il existe un ordre $<$ sur la base de Herbrand $B_{\mathcal{P}}$ de $\mathcal{P}$ tel que pour tout fait p, $p \in I$ implique $\exists \rho \in Inst(\mathcal{P})$: $(Corps(\rho) \subseteq I$ et $q < p$ pour tout $q \in pos(Corps(\rho))$.*

Nous demandons dans la suite que `Up_Der` soit définie de sorte que `Up_Der`$(\Delta, \delta, \mathcal{P})$ soit bien supportée par $\mathcal{P}$.

3.2 Propriétés des bases mises à jour

Dans la section précédente, nous avons présenté quelques propriétés qui sont directement liées à l'application `Up_Der`. Dans cette section, nous nous intéressons aux propriétés de l'application `Up_Sem`. Rappelons que cette application est la composition des applications `Up_Der` et `Apply`.

Les propriétés qui suivent ne doivent pas obligatoirement être vérifiées. Le contexte d'utilisation des programmes de règles déterminera la considération de leur validité.

Les instances de base de données considérées ici sont incomplètes. Il est donc naturel de comparer celles-ci en termes de "quantité" d'information contenue par chacune d'elles. Ainsi, étant données deux bases satisfaisant les mêmes propriétés (par exemple des contraintes d'intégrité), il est parfois utile de choisir celle qui contient le plus d'informations.

Définition 6 (Ordre de connaissance) *Soient Δ et Δ' deux instances. Δ est plus* informative *que Δ' , noté $\Delta \succeq \Delta'$, ssi $I(\Delta) \subseteq I(\Delta')$.*

Remarque 1 *Il est à remarquer que la relation $\succeq$ est différente des relations d'ordre $\leq_t$ (ordre de vérité) et $\leq_k$ (appelé aussi ordre de connaissance) introduits dans [Fit93]. En effet, soient deux interprétations partielles I et J.*

$$I \leq_t J \text{ ssi } pos(I) \subseteq pos(J) \text{ et } neg(I) \supseteq neg(J).$$
$$I \leq_k J \text{ ssi } pos(I) \subseteq pos(J) \text{ et } neg(I) \subseteq neg(J).$$

Considérons l'univers $U = \{a, b, c\}$, $I = \{a, \neg b\}$ et $J = \{\neg a, b, \neg c\}$. La partie positive de I (resp. de J) est $\{a\}$ (resp. $\{b\}$) et sa partie négative est $\{b\}$ (resp. $\{a, c\}$). I et J sont incomparables selon les ordres $\leq_t$ et $\leq_k$, alors que selon la définition 6, nous avons $J \succeq I$, puisque la partie inconnue de J est vide, alors que c est inconnu dans I.

La propriété de *changement minimal* est une propriété classique dans le cadre de l'étude des mises à jour [Win86, KM91]. Pour définir cette notion, nous avons besoin d'introduire une notion de distance entre instances de bases de données. La définition qui suit est une extension triviale au cas des instances incomplètes de la définition classique.

Définition 7 (Distance) *Soient Δ, Δ' et Δ'' des instances de bases de données. $\div$ est la différence symétrique ensembliste. La distance entre Δ' et Δ'', notée $|\Delta',\ \Delta''|$ est définie par*

$$|\Delta',\ \Delta''| = [V(\Delta') \div V(\Delta'')] \bigcup [I(\Delta') \div I(\Delta'')].$$

Δ' est "plus proche" de Δ que Δ'' ne l'est, ssi $|\Delta, \Delta'| \subseteq |\Delta, \Delta''|$.

La distance entre deux bases de données représente l'ensemble des faits qui changent d'état pour passer de l'une à l'autre.

Exemple 3.1 Soit $\Delta =< \{a, b\}; \{c\} >$ et $\Delta' =< \{a, c\}; \{b, d\} >$. $|\Delta, \Delta'| = \{b, c\} \cup \{b, c, d\} = \{b, c, d\}$. Pour passer de Δ à Δ', b (qui est vrai) et d (qui est faux) doivent *devenir* inconnus et c (qui est inconnu) doit devenir vrai. Nous voyons donc que la distance représente une notion de "quantité de changement". $\square$

L'application naïve d'une m.à.j δ sur une base initiale Δ_{init} produit une base Δ_{user} obtenue en rajoutant à Δ_{init} tout fait p tel que $+p \in \delta$ et en supprimant chaque fait p tel que $-p \in \delta$. La base Δ_{user} ainsi obtenue reflète l'intention initiale de l'utilisateur. La base Δ_{user} est définie précisément ci-dessous.

Définition 8 *Soit Δ_{init} une base et δ une m.à.j. Δ_{user} est la base définie comme suit :*

- $V(\Delta_{user}) = (V(\Delta_{init}) \cup \delta^+\ ) \setminus \delta^-$.
- $I(\Delta_{user}) = I(\Delta_{init}) \setminus (\ \delta^+\ \cup \delta^-\ )$.

Exemple 3.2 Si $\Delta_{init} =< \{r\}; \{p, q\} >$ et δ est la mise à jour $\{+p, -q, -r\}$, alors la base $\Delta_{user} =< \{p\}; \emptyset >$. $\square$

Supposons que les instances valides de la base de données doivent appartenir à une famille $\mathcal{F}$ d'instances qui peut par exemple être caractérisée par des contraintes. Alors il n'y a aucune garantie que la base Δ_{user} appartienne à $\mathcal{F}$.

La propriété de changement minimal a pour objectif d'exprimer tout simplement que, dans ce cas, le résultat de la mise à jour n'est pas Δ_{user}, mais une instance Δ appartenant à $\mathcal{F}$ qui soit aussi proche que possible de la base Δ_{user}. Formellement,

Définition 9 (Changement Minimal) *L'instance Δ_{min} élément de $\mathcal{F}$ satisfait la propriété du changement minimal, respectivement à l'instance initiale Δ_{init} et la m.à.j δ ssi : $\forall \Delta \in \mathcal{F} : |\Delta_{user}, \Delta| \subseteq |\Delta_{user}, \Delta_{min}| \Rightarrow |\Delta_{user}, \Delta| = |\Delta_{user}, \Delta_{min}|$.*

La propriété suivante appelée priorité de la m.à.j. initiale exprime que la m.à.j. utilisateur doit être visible dans le nouvel état de la base. La définition formelle prend en compte le fait que dans notre contexte de travail, nous manipulons des instances incomplètes et que la m.à.j. utilisateur peut être en conflit avec les m.à.j. dérivées.

Définition 10 (Priorité des m.à.j initiales) *L'application* `Up_Sem` *donne priorité aux m.à.j. initiales ssi pour tout* $+p \in \delta$ *(resp.* $-p \in \delta$*) soit* $p \in V(\Delta')$ *(resp.* $p \in F(\Delta')$*) soit* $p \in I(\Delta')$ *où* Δ'=`Up_Sem`($\Delta, \delta, \mathcal{P}$)*.*

D'une manière faible il est vrai, cette propriété implique qu'une m.à.j. utilisateur n'est jamais rejetée : si p est l'objet d'une insertion initiale alors le statut de p dans la base mise à jour est soit vrai soit inconnu mais p **ne peut être faux**.

Nous introduisons maintenant la propriété de *stabilité*. Une sémantique de m.à.j est *stable* pour un programme de m.à.j $\mathcal{P}$ ssi l'application d'une m.à.j. vide après une m.à.j. quelconque δ ne vient en rien modifier le résultat de la m.à.j. δ. Plus précisément :

Définition 11 (Stabilité) *L'application* `Up_Sem` *est stable relativement à un programme* $\mathcal{P}$ *ssi :* `Up_Sem`$(\Delta, \delta, \mathcal{P}) = \Delta' \Rightarrow$ `Up_Sem`$(\Delta', \emptyset, \mathcal{P}) = \Delta'$.

Etant donnée une définition de `Up_Sem`, l'application peut être stable pour un programme et non stable pour un autre. La stabilité est motivée en partie comme suit. Une transition de la base, d'une instance Δ vers une instance Δ', est représentée par le couple (Δ, Δ'). Une famille de transitions $\mathcal{T}$ est dite stable ssi pour chaque transition (Δ, Δ') de $\mathcal{T}$, les transitions (Δ, Δ) et (Δ', Δ') sont aussi dans $\mathcal{T}$. Par exemple, si $\mathcal{C}$ est un ensemble de contraintes statiques, alors les transitions (Δ, Δ') telles que Δ et Δ' satisfont toutes les deux $\mathcal{C}$ forment un ensemble stable de transitions. Nous verrons par contre dans la section 4.2, qu'un ensemble de contraintes de transitions ne définit pas nécessairement une famille stable de transitions.

Donc, si un programme de m.à.j maintient la cohérence d'une base de données en présence de contraintes statiques, alors la stabilité de `Up_Sem` pour ce programme devra être vérifiée. Par contre, si le programme maintient la cohérence en présence de contraintes de transition, alors la stabilité de `Up_Sem` pour ce programme ne sera pas suhaitable.

L'*indépendance du domaine* est une notion importante des langages d'interrogation de bases de données. Intuitivement, une requête est indépendante du domaine ssi son résultat ne dépend que des constantes apparaissant dans la base et dans la requête elle-même (i.e. domaine actif). Cette notion est transposée dans notre cadre, en requérant que le calcul de la base résultant d'une m.à.j utilisateur sur une instance initiale en présence de règles soit indépendant du domaine.

Définition 12 (Indépendance du Domaine) *Soit* $\mathcal{H}$ *l'ensemble des constantes apparaissant dans* $\mathcal{P} \cup \delta \cup \Delta$*,* $D_1 \supseteq \mathcal{H}$ *et* $D_2 \supseteq \mathcal{H}$*,* $\mathcal{P}_1$ *et* $\mathcal{P}_2$ *les instanciations de* $\mathcal{P}$ *à partir de, respectivement,* D_1 *et* D_2*. La sémantique de m.à.j.* `Up_Sem` *est indépendante du domaine ssi :* $\forall D_1, D_2 \supseteq \mathcal{H}$*:* `Up_Sem`$(\Delta, \delta, \mathcal{P}_1) =$ `Up_Sem`$(\Delta, \delta, \mathcal{P}_2)$.

3.3 Sémantique bien fondée

Dans cette section, nous montrons que la sémantique bien fondée peut être utilisée pour définir le mécanisme de dérivation de m.à.j (i.e. l'application `Up_Der`). Nous montrons de plus que cette définition de l'application `Up_Der` satisfait les propriétés introduites à la section 3.1.

Nous commençons par procéder à une réécriture des règles d'un programme de mise à jour. Cette modification a pour objectif de garantir que les m.à.j dérivées soient *bien fondées*. Elle est issue d'une lecture naturelle d'une règle de m.à.j ρ de la forme :

$$Corps(\rho) \longrightarrow +p$$

comme une règle de défaut [Rei80] :

- Si le *pré-requis* $Corps(\rho)$ est vrai, et
- si il est consistent de supposer la *justification* $\neg{-p}$, alors
- déduire la *conséquence* $+p$.

Il est reconnu que la sémantique bien fondée est un des formalismes adéquat d'approximation du raisonnement par défaut [BF91]. Ceci constitue l'une des motivations qui nous ont fait examiner l'usage de cette sémantique.

Ainsi, dans la suite du papier, nous supposons que pour chaque règle ρ telle que $T\hat{e}te(\rho) = +P(\vec{x})$ (resp. $-P(\vec{x})$)[1], $Corps(\rho) \ni \neg -P(\vec{x})$ (resp. $\neg + P(\vec{x})$).

Etant donnée une requête spécifiée par un programme Datalog $\mathcal{P}$, afin de définir le résultat de cette requête sur une instance Δ de la base de données, il est procédé de la manière suivante : la base Δ est "plongée" dans le programme $\mathcal{P}$ produisant ainsi un nouveau programme ; le résultat de la requête $\mathcal{P}$ sur Δ est alors extrait du modèle du programme augmenté.

Nous allons procéder de manière tout à fait similaire dans le contexte des mises à jour dérivées d'un programme de règles $\mathcal{P}$ à partir d'une instance Δ et d'une m.à.j. utilisateur δ : le programme est augmenté par des règles plongeant la base de données Δ et la m.à.j utilisateur δ dans $\mathcal{P}$. Ces règles sont :

1. pour chaque $p \in V(\Delta)$, ajouter la règle $\longrightarrow p$,
2. pour chaque $p \in I(\Delta)$, ajouter la règle $\neg p \longrightarrow p$,
3. pour chaque $+p \in \delta$, ajouter la règle $\neg{-p} \longrightarrow +p$ et
4. pour chaque $-p \in \delta$, ajouter la règle $\neg{+p} \longrightarrow -p$.

Le programme ainsi obtenu est noté $\mathcal{P}_{\delta,\Delta}$.

Pour comprendre le choix que nous avons fait pour prendre en compte la partie inconnue de Δ, il suffit de remarquer que si un atome p est défini par la seule règle

1. $\vec{x}$ est un vecteur de termes.

$\neg p \longrightarrow p$, alors p est inconnu dans le modèle bien fondé du programme en question. Ici, si l'atome p est inconnu, la règle $\neg p \longrightarrow p$ est la seule définissant p introduite dans le programme augmenté.

La définition de l'opérateur de dérivation des m.à.j `Up_Der` proposée ci-dessous utilise directement la notion de modèle bien fondé.

Définition 13 (`Up_Der_WF`) *L'application* `Up_Der_WF` *est définie par :*
`Up_Der_WF`*(Δ, δ, $\mathcal{P}$) est la plus grande m.à.j incluse dans le modèle bien fondé du programme $\mathcal{P}_{\delta,\Delta}$.*

Exemple 3.3 Soient la base $\Delta =< \{p\}, \{q\} >$, la m.à.j $\delta = \{+r, -q\}$ et le programme $\mathcal{P} = \{\rho : +r, q, \neg{+p} \longrightarrow -p\}$. $\mathcal{P}_{\delta,\Delta}$ est le programme :

$$\left\{ \begin{array}{rcl} & \longrightarrow & p \\ \neg q & \longrightarrow & q \\ \neg{-r} & \longrightarrow & +r \\ \neg{+q} & \longrightarrow & -q \\ +r, q, \neg{+p} & \longrightarrow & -p \end{array} \right\}$$

Le modèle bien fondé de ce programme est

$$WF(\mathcal{P}_{\delta,\Delta}) = \{p, \neg{+p}, -q, \neg{+q}, +r, \neg{-r}\}$$

Ainsi, la plus grande m.à.j qui puisse en être extraite est

$$\texttt{Up\_Der\_WF}(\Delta, \delta, \mathcal{P}) = \{\neg{+p}, -q, \neg{+q}, +r, \neg{-r}\}$$ □

Toutes les propriétés introduites dans la section 3.1 sont satisfaites par cette définition de la dérivation de m.à.j.

Proposition 1 `Up_Der_WF`*(Δ, δ, $\mathcal{P}$) est non contradictoire, bien fondée, est bien supportée par $\mathcal{P}_{\delta,\Delta}$.*

Nous devons définir maintenant l'effet d'une m.à.j (dérivée) sur une base de données, donc définir l'application `Apply`.

Définition 14 *Soit δ une m.à.j non contradictoire et Δ une instance.* `Apply`$(\delta, \Delta) = \Delta'$*, où :*

$$\begin{array}{l} V(\Delta') = \{p | +p \in \delta\} \cup \{p |\, p \in V(\Delta) \textit{ et } \neg{-p} \in \delta\} \\ I(\Delta') = \{p | p \notin V(\Delta') \cup DEL(\delta)\} \textit{ où} \\ DEL(\delta) = \{p | -p \in \delta\} \cup \{p |\, p \in F(\Delta) \textit{ et } \neg{+p} \in \delta\} \end{array}$$

Remarque 2 *Du fait que* `Up_Der_WF` *est non contradictoire, sa composition avec* `Apply`*, notée* `Up_Sem_WF`*, est bien définie.*

Dans ce qui suit, nous étudions les propriétés de la sémantique de m.à.j `Up_Sem_WF`. Cette étude va nous conduire à proposer des variantes de la proposition initiale (i.e. des variantes de `Up_Der`) afin de “garantir” certaines propriétés.

La première modification du processus de dérivation des m.à.j que nous proposons a trait à la prise en compte de la partie inconnue de la base initiale et à son plongement dans le programme. Elle est motivée par la maximisation de la connaissance.

Cette modification utilise l'observation suivante : étant donnée une interprétation I et une règle close ρ, ρ est satisfaite par I dès que le corps de ρ n'est pas satisfait par I (i.e. faux ou inconnu). Cette propriété n'est pas utilisée dans la définition standard du modèle bien fondé.
Nous illustrons cette discussion par l'exemple suivant.

Exemple 3.4 Soient $\delta = \{+p\}$, $\Delta =< \emptyset, \{q\} >$ et $\mathcal{P} = \{+p, q, \neg{-r} \longrightarrow +r\}$.

$$\mathcal{P}_{\delta,\Delta} = \left\{ \begin{array}{llcl} \rho_1 : & \neg{-p} & \longrightarrow & +p \\ \rho_2 : & \neg q & \longrightarrow & q \\ \rho_3 : & +p, q, \neg{-r} & \longrightarrow & +r \end{array} \right\}$$

Le corps de la règle ρ_3 contient le fait q qui est inconnu dans la base. Le fait que q soit inconnu implique que $+r$ est lui aussi inconnu dans le modèle bien fondé $\mathcal{M}$ de $\mathcal{P}_{\delta,\Delta}$. Ainsi,

$$\texttt{Up\_Sem\_WF}(\Delta, \delta, \mathcal{P}) =< \{p\}; \{q, r\} >.$$

Notons que r est inconnu dans la nouvelle base car $+r$ est inconnu dans $\mathcal{M}$.

L'interprétation I obtenue à partir de $\mathcal{M}$ en considérant $+r$ comme étant faux est un modèle de $\mathcal{P}_{\delta,\Delta}$. De plus, $\texttt{Apply}(I, \Delta) =< \{p\}, \{q\} >$ est un résultat de la m.à.j $+p$ en présence de $\mathcal{P}$. □

Définition 15 (`Up_Der_WFU`) *Soit $\mathcal{P}^{\mathcal{K}}_{\delta,\Delta}$ le programme de m.à.j obtenu à partir de $\mathcal{P}_{\delta,\Delta}$ en supprimant toute règle ρ telle qu'il existe un littéral ℓ dans $Corps(\rho)$ avec $I(\Delta) \cap \{\ell, \neg\ell\} \neq \emptyset$.*

L'application `Up_Der_WFU` *est définie par :* $\texttt{Up\_Der\_WFU}(\Delta, \delta, \mathcal{P})$ *est la plus grande m.à.j incluse dans le modèle bien fondé de $\mathcal{P}^{\mathcal{K}}_{\delta,\Delta}$. La sémantique de m.à.j* `Up_Sem_WFU` *est définie par la composition de* `Up_Der_WFU` *et* `Apply`.

Les m.à.j dérivées avec l'application `Up_Der_WFU` ne sont pas contradictoires (ce qui implique que leur composition avec `Apply` est bien définie), elles sont bien fondées et bien supportées. De plus, l'application `Up_Der_WFU` est "meilleure" que `Up_Der_WF` du point de vue de la maximisation des connaissances.

Proposition 2 *L'instance de base de données* $\texttt{Up\_Sem\_WFU}(\Delta, \delta, \mathcal{P})$ *est plus informative que l'instance* $\texttt{Up\_Sem\_WF}(\Delta, \delta, \mathcal{P})$.

Exemple 3.4 (Suite) Le programme simplifié est

$$\mathcal{P}^{\mathcal{K}}_{\delta,\Delta} = \{\rho_1 : \neg{-p} \longrightarrow +p\}$$

Les règles ρ_2 et ρ_3 sont supprimées du programme car toutes les deux contiennent des littéraux inconnus : $\neg q$ pour ρ_2 et q pour ρ_3. Ainsi, nous avons $\texttt{Up\_Sem\_WFU}(\Delta, \delta, \mathcal{P})$= $<\{p\}; \{q\}>$ qui est plus informative que la base $\texttt{Up\_Sem\_WF}(\Delta, \delta, \mathcal{P})$. □

Lemme 1 *Les deux sémantiques de m.à.j* `Up_Sem_WF` *et* `Up_Sem_WFU` *donnent priorité aux m.à.j utilisateur.*

Dans un contexte où maximiser l'information est primordial, il peut être souhaitable de rejeter une m.à.j. initiale lorsque celle-ci introduit dans la base résultante une information de type inconnue i.e. lorsque la m.à.j. initiale n'est pas réalisable. Nous proposons maintenant une variante de l'application de dérivation de m.à.j. qui induit une sémantique de m.à.j. ne donnant pas priorité aux m.à.j. initiales.

Définition 16 (`Up_Der++`) *Soit* `Up_Der` *l'une des applications* `Up_Der_WF` *ou bien* `Up_Der_WFU`. *Soit* β=`Up_Der`$(\Delta, \delta, \mathcal{P}) \cap \delta$. *L'application* `Up_Der++` *est définie par :*

`Up_Der++`$(\Delta, \delta, \mathcal{P}) =$ `Up_Der`$(\Delta, \beta, \mathcal{P})$.

La m.à.j β est un sous ensemble de la m.à.j initiale δ. L'idée motivant la définition de `Up_Der++` est la suivante : les m.à.j dérivées sont calculées, seules les m.à.j initiales qui sont confirmées (dérivées) par cette étape sont reprises pour effectuer une nouvelle étape de dérivation de m.à.j.

Les m.à.j dérivées avec les applications `Up_Der_WF++` et `Up_Der_WFU++` sont non conflictuelles, bien fondées et bien supportées.

Dans ce qui suit, `Up_Sem_WF++` (resp. `Up_Sem_WFU++`) désigne la sémantique de m.à.j basée sur `Up_Der_WF++` (resp. `Up_Der_WFU++`). Nous utilisons la notation `Up_Sem++` pour établir les propriétés qui sont satisfaites par les deux sémantiques.

Proposition 3

1. *Soit β la m.à.j définie comme ci-dessus.* `Up_Der++`$(\Delta, \delta, \mathcal{P}) \cap \beta = \beta$.
2. `Up_Der++` *ne donne pas la priorité aux m.à.j initiales.*
3. *Soit* Δ' =`Up_Sem++`$(\Delta, \delta, \mathcal{P})$. *Soit $+p$ (resp. $-p$) un élément de δ tel que $p \in I(\Delta')$. Soit $\delta' = \delta - \{+p\}$ (resp. $\delta - \{-p\}$). Alors, $p \in I(\Delta'')$ où* Δ''=`Up_Der++`$(\Delta, \delta', \mathcal{P})$.
4. `Up_Sem_WFU++`$(\Delta, \delta, \mathcal{P}) \succeq$ `Up_Sem_WFU`$(\Delta, \delta, \mathcal{P})$.

Le point 1. ci-dessus montre que toutes les mises à jours initiales qui sont confirmées lors du premier calcul du modèle se retrouvent confirmées dans la deuxième étape. En d'autres termes, ceci montre qu'il n'est pas nécessaire de répéter le calcul des mises à jour confirmées plus d'une fois.

Le point 3. de la proposition dit tout simplement qu'un fait sujet à une mise à jour par l'utilisateur (ex: $+p \in \delta$) et qui se retrouve inconnu dans la base finale `Up_Sem++`$(\Delta, \delta, \mathcal{P})$, sera nécessairement inconnu dans la base obtenue à partir de la m.à.j $\delta' = \delta \setminus \{+p\}$.

La sémantique de m.à.j `Up_Sem_WFU++` présente une propriété très intéressante : elle préserve les instances totales à la seule condition que le programme soit stratifiable [ABW88]. Nous voudrions insister sur le fait que cette propriété est indépendante des données dans le sens où la condition de stratification requise porte sur le programme initial et non pas sur le programme augmenté pour plonger l'instance et la m.à.j initiales.

Proposition 4 *Si le programme $\mathcal{P}$ est stratifiable et si l'instance Δ est totale, alors l'instance* `Up_Sem++`$(\Delta, \delta, \mathcal{P})$ *est totale.*

La stabilité est une propriété liée à un programme de m.à.j. donné. Nous proposons une condition suffisante portant sur un programme $\mathcal{P}$ pour que l'application `Up_Sem++` soit stable pour ce programme.

Proposition 5 *Soit ρ une règle de m.à.j. L'expansion de ρ, notée Exp(ρ), est l'ensemble des règles obtenu en remplaçant chaque littéral de base $P(\vec{x})$ (resp. $\neg P(\vec{x})$) dans $Corps(\rho)$ par $(P(\vec{x}) \wedge \neg{-}P(\vec{x})) \vee {+}P(\vec{x})$ (resp. par $(\neg P(\vec{x}) \wedge \neg{+}P(\vec{x})) \vee {-}P(\vec{x})$) puis en dépliant la règle pour éliminer les disjonctions.* `Up_Sem_WFU++` *est stable pour $\mathcal{P}$ si :*

1. $\nexists \rho \in \mathcal{P} : Cond(\rho) = \emptyset$ *et* $neg(Event(\rho)) = Event(\rho)$*, et*
2. $\forall \rho \in \mathcal{P} : neg(Event(\rho)) = Event(\rho)$ *et* $Cond(\rho) \neq \emptyset \Rightarrow Exp(\rho) \subseteq \mathcal{P}$.

Exemple 3.5 Soit le programme

$$\mathcal{P} = \left\{ \begin{array}{ccc} {+}p, \neg{-}r & \longrightarrow & {+}r \\ \neg{+}q, \neg{+}r & \longrightarrow & {-}r \end{array} \right\}$$

Considérons la base de données $\Delta =< \emptyset, \emptyset >$ et la mise à jour $\delta = \{{+}p\}$. Soit $\Delta' =$`Up_Sem++`$(\Delta, \delta, \mathcal{P})$. $\Delta' =< \{p\}; \{r\} >$, alors que la base `Up_Sem++`$(\Delta', \emptyset, \mathcal{P})$ est égale à $< \{p\}; \emptyset >$. La deuxième règle de $\mathcal{P}$ ne satisfait pas la première condition de la proposition ci-dessus.

Considérons maintenant le programme $\mathcal{P} = \{p, \neg{-}q \longrightarrow {+}q\}$, l'instance $\Delta = < \emptyset, \emptyset >$ et la m.à.j $\delta = \{{+}p\}$. Il est facile de voir que `Up_Sem++`$(\Delta, \delta, \mathcal{P})$= $\Delta' = < \{p\}, \emptyset >$ et `Up_Sem++`$(\Delta', \emptyset, \mathcal{P}) =< \{p, q\}, \emptyset >$. L'unique règle de $\mathcal{P}$ ne satisfait pas la deuxième condition de la proposition. □

L'examen de la propriété d'indépendance du domaine pour l'application `Up_Sem++` conclut cette section. En raison de la complexité du test d'indépendance du domaine, la littérature propose, pour les requêtes, un certain nombre de conditions suffisantes pour qu'un programme soit indépendant du domaine. Nous procédons de manière similaire ici en présentant une condition suffisante sur un programme de m.à.j $\mathcal{P}$, garantissant que `Up_Sem++` soit indépendante du domaine pour $\mathcal{P}$.

Définition 17

– *Une règle ρ est* semi_autorisée *(resp.* autorisée*) si chaque variable qui apparaît dans ρ apparaît dans un littéral positif de ρ (resp. dans le corps de ρ). Ces deux propriétés sont généralisées pour un ensemble de règles, i.e. un programme.*

– *$\mathcal{P}^-$ désigne les règles de $\mathcal{P}$ qui ont un $-$littéral dans leurs têtes.*

– $\mathcal{P}$ *est* **restreint** *ssi*

1. *$\mathcal{P}^-$ est semi_autorisé,*
2. *$\forall \rho \in \mathcal{P} \backslash \mathcal{P}^-$ ρ est autorisé, et*
3. *pour toute $\rho \in \mathcal{P}^-$, si ρ n'est pas autorisée et si $Tete(\rho) = -P(\vec{z})$, alors pour toute $\hat{\rho} \in \mathcal{P} \backslash \mathcal{P}^-$, si $-P(\vec{x}) \in Corps(\rho)$, alors toute variable dans $\vec{x}$ apparaît dans un littéral de la forme $+Q(\vec{y})$ ou $Q(\vec{y})$.*

L'idée intuitive sur cette restriction consiste à autoriser la dérivation de la suppression de nuplets qui ne sont pas dans le domaine actif, mais en même temps interdire à ce que ces suppressions servent à dériver des insertions de noueaux nuplets.

Proposition 6 *Si le programme de m.à.j $\mathcal{P}$ est restreint, alors* `Up_Sem++` *est indépendante du domaine pour $\mathcal{P}$.*

Exemple 3.6 $\mathcal{P} = \{\neg P(x,y,z), R(z), \neg + Q(x,y,t) \longrightarrow -Q(x,y,t)\}$ est restreint. Notons que le même programme vu comme une requête, n'est pas indépendant du domaine. □

La proposition suivante ordonne les bases obtenues par les différentes sémantiques de m.à.j (`Up_Sem_*`) et ce selon la relation d'ordre $\leq_k$ de Fitting [Fit93] (voir remarque 1).

Proposition 7 *Soient Δ, δ et $\mathcal{P}$. Nous avons la relation suivante :*

$$\begin{array}{c} \texttt{Up\_Sem\_WF}(\Delta, \delta, \mathcal{P}) = \Delta_{WF} \\ \leq_k \\ \texttt{Up\_Sem\_WFU}(\Delta, \delta, \mathcal{P}) = \Delta_{WFU} \\ \leq_k \\ \texttt{Up\_Sem\_WFU++}(\Delta, \delta, \mathcal{P}) = \Delta_{WFU++} \end{array}$$

Remarquons que la relation $\leq_k$ est plus forte que la relation $\preceq$ dans le sens où $I \leq_k J$ implique $I \preceq J$. Ainsi, La proposition ci-dessus implique la propositions 2 et le point 4. de la proposition 3.

4 Contraintes d'intégrité & dérivation de règles

Cette section est destinée à valider le langage de règles de mise à jour introduit précédemment. Pour ce faire nous avons choisi une application classique : la maintenance des contraintes d'intégrité par des programmes de règles de mise à jour. Dans ce contexte, intuitivement les règles ont pour rôle de détecter les mises à jour susceptibles de violer une contrainte, de tester si effectivement la contrainte est violée et dans ce cas d'effectuer un traitement permettant de réparer l'incohérence temporaire de la base.

Les deux sous-sections suivantes exposent deux méthodes **non déterministes** générant automatiquement un ensemble de règles à partir d'un ensemble de contraintes : la première méthode a pour cible la maintenance de contraintes statiques, la deuxième méthode a pour objet la maintenance de contraintes de transition.

Même si la gestion de la cohérence est l'une des applications "fétiches" des systèmes actifs, les études ayant porté sur la dérivation de règles de m.à.j à partir de spécifications déclaratives de contraintes [CW90, Gre93, Ger94, UD92, CFPT94, BCP94] adoptent une démarche pragmatique et/ou analysent les propriétés des règles proposées pour maintenir les contraintes (terminaison, minimalité, confluence, etc).

4.1 Contraintes statiques

Une contrainte statique est une propriété qui doit être satisfaite par chaque état de la base. Dans cette section, nous nous intéressons aux contraintes qui sont exprimées par des formules du premier ordre de la forme :

$$\forall \vec{x} : [P_1(\vec{x}_{P_1}) \wedge \cdots \wedge P_n(\vec{x}_{P_n}) \Rightarrow \exists \vec{y} : (Q_1(\vec{x}_{Q_1}, \vec{y}_{Q_1}) \vee \cdots \vee Q_m(\vec{x}_{Q_m}, \vec{y}_{Q_m}))]$$

où $\vec{x} = \bigcup_{i=1}^{n} \vec{x}_{P_i}$, $\vec{y} = \bigcup_{i=1}^{m} \vec{y}_{Q_i}$, $\bigcup_{j=1}^{n} \vec{x}_{Q_j} \subseteq \vec{x}$.

L'instance Δ viole l'ensemble de contraintes $\mathcal{C}$ ssi il existe une contrainte $c \in \mathcal{C}$, et une substitution θ telles que $\Delta \models \neg c\theta$. L'instance Δ est cohérente vis à vis de $\mathcal{C}$ ssi aucune contrainte $c \in \mathcal{C}$ n'est violée par Δ.

Exemple 4.1 Soit $\Delta =< \{P(a)\}, \{Q(a)\} >$ et $\mathcal{C} = \{c : \forall x(P(x) \Rightarrow Q(x))\}$. Δ ne viole pas c car l'implication n'est pas fausse ; elle est inconnue (Rappelons qu'une implication $A \Rightarrow B$ est fausse ssi A est vrai et B est faux.). $\square$

La dérivation des règles de m.à.j à partir d'un ensemble $\mathcal{C}$ de contraintes statiques est définie par les quatre étapes suivantes :

Première étape :
Chaque contrainte est réécrite sous forme de *denial* (tout comme dans [SK88]):

$$\rho_c \equiv P_1(\vec{x}_{P_1}), \ldots, P_n(\vec{x}_{P_n}), \neg Q_1(\vec{x}_{Q_1}, \vec{y}_{Q_1}), \ldots, \neg Q_m(\vec{x}_{Q_m}, \vec{y}_{Q_m}) \longrightarrow$$

Notons que la règle ci-dessus peut ne pas être restreinte : les variables de $\vec{y}_{Q_j}$ n'apparaissent dans aucun littéral positif. Afin de pallier ceci, pour chaque littéral négatif $\neg Q_j(\vec{x}_{Q_j}, \vec{y}_{Q_j})$, nous introduisons la règle "déductive" :

$$Q_j(\vec{x}_{Q_j}, \vec{y}_{Q_j}) \longrightarrow Q'_j(\vec{x}_{Q_j})$$

et nous remplaçons ρ_c par :

$$P_1(\vec{x}_{P_1}), \ldots, P_n(\vec{x}_{P_n}), \neg Q'_1(\vec{x}_{Q_1}), \ldots, \neg Q'_m(\vec{x}_{Q_m}) \longrightarrow$$

Q'_j est un prédicat auxiliaire. Le programme de règles ainsi obtenu est noté $\mathcal{P}_1$.

Exemple 4.2 Soient les prédicats $Emp(N, S, D)$, $Dept(D, V)$ où N, S, D, V représentent, respectivement, le Nom, le Salaire, le Département et la Ville. Considérons la contrainte référentielle classique qui dit : si un employé est affecté à un département, alors ce dernier doit exister dans la base. Cette contrainte est exprimée par :

$$c : \forall x, y, z \ : \ [Emp(x, y, z) \Rightarrow \exists t \ : \ Dept(z, t)]$$

La contrainte c est réécrite sous forme de denial comme suit:

$$\begin{array}{rcl} Dept(z,t) & \longrightarrow & Dept'(z) \\ Emp(x,y,z), \neg Dept'(z) & \longrightarrow & \end{array}$$

$\square$

Deuxième étape:
Pour chaque denial $\rho_c = \ell_1, \ldots, \ell_n, \neg\ell_{n+1}, \ldots, \neg\ell_m \longrightarrow$ dans $\mathcal{P}_1$, nous dérivons une règle de m.à.j ρ qui est l'une des deux règles suivantes :

$$\begin{array}{lcll} \ell_1, \ldots, \ell_n, \neg\ell_{n+1}, \ldots, \neg\ell_{i-1}, \neg\ell_{i+1}, \ldots, \neg\ell_m, \neg - \ell_i & \longrightarrow & +\ell_i & \text{ou} \\ \ell_1, \ldots, \ell_{i-1}, \ell_{i+1}, \ldots, \ell_n, \neg\ell_{n+1}, \ldots, \neg\ell_m, \neg + \ell_i & \longrightarrow & -\ell_i & \end{array}$$

Le programme $\mathcal{P}_2$ est l'ensemble des règles déductives de l'étape 1 et des règles de m.à.j de l'étape 2.

Exemple 4.2 (Suite) Nous pouvons dériver l'une des deux règles de m.à.j suivantes:

$$\begin{array}{lrcll} \rho_1 : & Emp(x,y,z) & \longrightarrow & +Dept'(z) & \text{ou bien} \\ \rho_2 : & \neg Dept'(z) & \longrightarrow & -Emp(x,y,z) & \end{array}$$

Comme nous venons de le voir, à partir de chaque denial dont le corps contient m littéraux, il est possible de dériver m règles de mise à jour. Le choix de l'une parmi ces m règles peut être confié soit au concepteur de la base, ou bien par un algorithme non déterministe.

Troisième étape:
Remarquons que certaines règles de $\mathcal{P}_2$ peuvent avoir dans leurs têtes des insertions sur des prédicats auxiliaires (ex. $+Q'(\vec{x})$). Aussi ces insertions doivent être traduites en des insertions sur des prédicats de base.

Pour ce faire, une constante spéciale $\perp$ est introduite. Pour chaque $\rho \in \mathcal{P}_2$ telle que $T\hat{e}te(\rho) = +Q'(\vec{x})$ et Q' est un prédicat auxiliaire, $T\hat{e}te(\rho)$ est remplacé par $+Q(\vec{x}, \vec{\perp})$ et $\neg - Q'(\vec{x})$ dans $Corps(\rho)$ est remplacé par $\neg - Q(\vec{x}, \vec{\perp})$.

Le programme ainsi obtenu est noté $\mathcal{P}_3$.

Exemple 4.2 (Suite) Si nous choisissons la règle ρ_1, alors celle-ci est remplacée par :

$$\rho_1 : \quad Emp(x,y,z) \quad \longrightarrow \quad +Dept(z, \perp)$$

Quatrième étape :
Chaque règle $\rho \in \mathcal{P}_3$ est remplacée par son expansion $Exp(\rho)$ (voir Proposition 5 pour la définition de $Exp(\rho)$). Le programme final ainsi obtenu est noté $\mathcal{P}_{\mathcal{C}}$.

Exemple 4.2 (Suite) Si la règle ρ_1 ci-dessus est choisie, alors elle est transformée en les règles

$$\begin{array}{lrcl} \rho_1 : & Emp(x,y,z), \neg - Emp(x,y,z) & \longrightarrow & +Dept(z, \perp) \\ \rho_2 : & +Emp(x,y,z) & \longrightarrow & +Dept(z, \perp) \end{array}$$

4.2 Contraintes de transition

Une contrainte de transition exprime une propriété qui lie deux instances consécutives d'une base. Une transition est spécifiée par un couple (Δ,Δ') où Δ est l'instance initiale et Δ' est l'instance finale. Nous considérons les contraintes de transition exprimées par les formules suivantes de la logique du premier ordre :

$$\forall \vec{x} : \bigwedge_{i=1}^{m_1} P_i(\vec{x}_{P_i}) \bigwedge_{j=1}^{m_2} \vec{Q}_j(\vec{x}_{Q_j}) \Rightarrow \exists \vec{y} : \bigvee_{k=1}^{n_1} R_k(\vec{x}_{R_k}, \vec{y}_{R_k}) \bigvee_{l=1}^{n_2} \vec{S}_l(\vec{x}_{S_l}, \vec{y}_{S_l})$$

Dans l'expression ci-dessus, les symboles de prédicats *fléchés* (surlignés par une flèche) sont évalués sur l'état final d'une transition.

Exemple 4.3 L'exemple le plus couramment utilisé pour illustrer ces contraintes est la règle qui dit que le salaire d'un employé ne doit pas décroître. Ainsi, une transition (Δ_1, Δ_2) satisfait cette contrainte ssi les salaires des employés dans l'instance Δ_2 ne sont pas inférieurs aux salaires de ceux-ci dans l'instance Δ_1. Cette contrainte peut s'exprimer par la formule :

$$\forall x, s : Emp(x, s), \vec{Emp}(x, s'), s' < s \Rightarrow \vec{Emp}(x, s) \vee Emp(x, s')$$

En adoptant cette notation, la satisfaction de contraintes de transition peut se définir comme suit :

Définition 18 *Soit c une contrainte de transition. La transition* (Δ, Δ') *viole c ssi :*

$$\begin{aligned} \exists \theta : \quad & \Delta \models \bigwedge_{i=1}^{m_1} P_i(\vec{x}_{P_i}).\theta,\ \Delta' \models \bigwedge_{j=1}^{m_2} \vec{Q}_j(\vec{x}_{Q_j}).\theta, \\ & \Delta \models \neg \bigvee_{k=1}^{n_1} R_k(\vec{x}_{R_k}, \vec{y}_{R_k}).\theta \textbf{ et } \Delta' \models \neg \bigvee_{l=1}^{n_2} \vec{S}_l(\vec{x}_{S_l}, \vec{y}_{S_l}).\theta \end{aligned}$$

Comme pour les contraintes statiques, le processus de génération d'un programme de m.à.j s'effectue en quatre étapes que nous décrivons ci-dessous.

Première étape : Elle consiste à traduire les formules de premier ordre en denials. La méthode est identique à celle décrite pour les contraintes statiques.

Seconde étape : Pour chaque denial de $\mathcal{P}_1$

$$\vec{E_1}, \ldots, \vec{E_m}, \neg\vec{F_1}, \ldots, \neg\vec{F_n}, B_1, \ldots, B_n \longrightarrow$$

où B_i est un littéral de base E_i et F_j sont des atomes de base, nous générons la règle de m.à.j ρ qui est l'une des deux règles ci-dessous :

$$+E_1, \ldots, +E_{i-1}, +E_{i+1}, \ldots, +E_m, -F_1, \ldots, -F_k, B_1, \ldots, B_n, \neg + E_i \longrightarrow -E_i$$

ou

$$+E_1, \ldots, +E_m, -F_1, \ldots, -F_{i-1}, -F_{i+1}, -F_k, B_1, \ldots, B_n, \neg - F_i \longrightarrow +F_i$$

Notons que les m.à.j que nous avons mises en tête des règles ne portent que sur des prédicats "fléchés". Ceci est dicté par le fait que "rétablir" la cohérence d'une transition ne peut se faire qu'en agissant sur le "futur" nouvel état de la base. Le programme obtenu est noté $\mathcal{P}_2$.

Troisième étape :
(identique à l'étape 3 exposée pour les contraintes statiques)

Quatrième étape :
Pour chaque $\rho \in \mathcal{P}_3$, $+E_i$ (resp. $-F_j$) est remplacé par $+E_i \vee (E_i \wedge \neg - E_i)$ (resp. $-F_j \vee (\neg F_j \wedge \neg + F_j)$).

Notons que, les littéraux de base ne sont pas transformés (l'expansion des règles n'est pas utilisée ici). Le programme final est noté $\mathcal{P}_\mathcal{C}$.

Exemple 4.4 Considérons la contrainte qui dit : *une personne ne peut être engagée comme directeur(trice) du département d que si elle était déjà employée de ce département.* Celle-ci est exprimée par la formule :

$$\forall x : [\overrightarrow{Dir}(x,d) \Rightarrow \exists y : Emp(x,y,d)]$$

Cette expression est transformée en :

$$\begin{array}{rcl} Emp(x,y,d) & \longrightarrow & Emp'(x,d) \\ \neg Emp'(x,d), \overrightarrow{Dir}(x,d) & \longrightarrow & \end{array}$$

Le programme final que nous obtenons est

$$\begin{array}{rcl} Emp(x,y,d) & \longrightarrow & Emp'(x,d) \\ \neg Emp'(x,d), \neg + Dir(x,d) & \longrightarrow & -Dir(x,d) \end{array}$$

□

4.3 Résultats

Dans ce qui suit, $\mathcal{C}$ représente un ensemble de contraintes statiques et/ou de transition et $\mathcal{P}_\mathcal{C}$ le programme associé à $\mathcal{C}$.

Proposition 8 *Soit* $\Delta' =$`Up_Sem++`$(\Delta, \delta, \mathcal{P}_\mathcal{C})$.

- $\mathcal{P}_\mathcal{C}$ *est un programme restreint.*
- Δ' *satisfait l'ensemble des contraintes statiques* $\mathcal{C}$.
- *La transition* (Δ, Δ') *satisfait l'ensemble des contraintes de transition* $\mathcal{C}$.

Le lecteur peut noter que dans le précédent résultat, nous n'avons pas pris pour hypothèse que la base initiale est cohérente. Rappelons que dans le cadre des systèmes actifs, cette hypothèse est en général faite car elle permet d'avorter une transaction lorsque le système détecte une exécution de règles qui ne termine pas. Ainsi, on est sûr de la cohérence de l'état final qui alors est l'état initial de la base.

Notre approche a, de ce point de vue, l'avantage de pouvoir s'appliquer lorsque le schéma de la base de données évolue et que de nouvelles contraintes sont ajoutées, cas où il est alors impossible de supposer que l'état initial de la base est cohérent.

Proposition 9 *Si $\mathcal{C}$ est un ensemble de contraintes statiques, alors la sémantique de m.à.j* `Up_Sem++` *est stable pour $\mathcal{P}_{\mathcal{C}}$.*

Remarquons que l'application `Up_Sem++` n'est pas stable pour tous les programmes générés à partir de contraintes de transition, ce qui est attendu : les contraintes de transitions ne définissent pas toujours des familles de transitions stables.

Exemple 4.5 Soit la contrainte de transition exprimée par le denial suivant :

$$c : \vec{p}, q \longrightarrow$$

A partir de cette contrainte, nous dérivons le programme

$$\mathcal{P}_c = \{q, \neg{+p} \longrightarrow {-p}\}$$

C'est l'unique règle qu'on peut dériver à partir de c. Considérons la base initiale Δ=$< \emptyset; \emptyset >$ et la m.à.j utilisateur $\delta = \{+p, +q\}$.
`Up_Sem_WFU++`$(\Delta_0, \delta, \mathcal{P}_c) =< \{p, q\}; \emptyset >= \Delta'$. Il est clair que (Δ, Δ') satisfait c. Notons néanmoins que la transition (Δ', Δ') viole la contrainte c. Remarquons aussi que si la m.à.j vide est appliquée à Δ', la base résultante est différente de Δ' : en effet nous avons `Up_Sem_WFU++`$(\Delta', \emptyset, \mathcal{P}_c) =< \{q\}; \emptyset >\neq \Delta'$. Ainsi, `Up_Sem_WFU++` n'est pas stable pour $\mathcal{P}_c$. □

La proposition suivante examine les propriétés de changement minimal et de maximisation des connaissances. Intuitivement, en se restreignant à des instances totales, nous montrons que les bases satisfaisant $\mathcal{C}$ qui réalisent moins de changement que la base $\Delta_{\mathcal{P}}$ obtenue à partir de $\mathcal{P}$ sont nécessairement moins informatives que $\Delta_{\mathcal{P}}$. Formellement,

Proposition 10 *Soit Δ_{init} une instance et δ une m.à.j. Soit $\Delta_{\mathcal{P}}$ la base qui désigne* `Up_Sem++`$(\Delta_{init}, \delta, \mathcal{P}_{\mathcal{C}})$. *Si Δ_{init} et $\Delta_{\mathcal{P}}$ sont toutes les deux totales, et s'il n'y a pas d'occurrence de la constante $\perp$ dans $\mathcal{P}$, alors pour toute base Δ qui satisfait les contraintes statiques de $\mathcal{C}$ (resp. telle que (Δ_{init}, Δ) satisfait les contraintes de transitions de $\mathcal{C}$), nous avons : $|\Delta_{user}, \Delta| \subset |\Delta_{user}, \Delta_{\mathcal{P}}| \Rightarrow \Delta_{\mathcal{P}} \succ \Delta$.*

En conséquence de la proposition ci-dessus :

Corollaire 1 *Si Δ_{init} et $\Delta_{\mathcal{P}}$ sont totales et si $\Delta_{user} \in \mathcal{C}$ (resp. $(\Delta_{init}, \Delta_{user}) \in \mathcal{C}$) alors $\Delta_{\mathcal{P}} = \Delta_{user}$.*

En considérant les programmes dérivés à partir de contraintes, la proposition 7 permet de comparer les différentes sémantiques de m.à.j (`Up_Sem`) en termes de "quantité de changement".

Corollaire 2 *Soient les bases Δ_{WF}, Δ_{WFU} et Δ_{WFU++} définies dans la proposition 7.*

1. $p \in |\Delta_{user}, \Delta_{WFU}| \setminus |\Delta_{user}, \Delta_{WF}| \Rightarrow p \in I(\Delta_{WF})$.

2. $p \in |\Delta_{user}, \Delta_{WFU++}| \setminus |\Delta_{user}, \Delta_{WFU}| \Rightarrow p \in I(\Delta_{WFU})$.

Le premier point du corollaire signifie tout simplement que : si le passage de Δ_{user} à Δ_{WFU} nécessite plus de changement que celui nécessaire pour le passage de Δ_{user} à Δ_{WF}, alors la différence ne concerne que des faits inconnus dans Δ_{WF} qui sont soit vrais soit faux dans Δ_{WFU}.

5 Les Programmes de Révision

Marek et Truscszinski [MT95] ont proposé un langage de règles de révision propositionnel.

Tout d'abord, remarquons que [MT95] ne considère que des bases de données *totales*. Ainsi, une instance est considérée comme étant un ensemble de faits (vrais). Dans la suite de cette section, nous nous restreignons donc aux instances totales pour comparer la sémantique de m.à.j présentée dans ce papier et la sémantique de révision de [MT95].

Les règles de révision peuvent être de l'une des deux formes suivantes :

$$in(q_1), \ldots, in(q_m), out(s_1), \ldots, out(s_n) \longrightarrow in(p)$$
$$in(q_1), \ldots, in(q_m), out(s_1), \ldots, out(s_n) \longrightarrow out(p)$$

Un programme de révision $\mathcal{P}^{\mathcal{R}}$ est un ensemble fini de règles de révision. Les expressions $in(p)$ et $out(p)$ expriment, respectivement, la *présence dans* et *l'absence de* la base du fait p. Etant donnée une base initiale D_I et un programme de révision $\mathcal{P}^{\mathcal{R}}$, [MT95] propose un algorithme qui *vérifie* si une base D_R est une "révision" de D_I selon $\mathcal{P}^{\mathcal{R}}$.

La première étape de l'algorithme consiste à simplifier le programme : soit D_I et D_R deux instances. $\mathcal{P}^{\mathcal{R}}$ est transformé comme suit :

1. Est Supprimée de $\mathcal{P}^{\mathcal{R}}$ toute règle telle que $q_i \notin D_R$ ou $s_i \in D_R$. Cette simplification produit le programme $\mathcal{P}^{\mathcal{R}}_{D_R}$.

2. Pour chaque règle $\rho \in \mathcal{P}^{\mathcal{R}}_{D_R}$, est supprimé de $Corps(\rho)$ chaque $in(q_i)$ tel que $q_i \in D_I$ et chaque $out(s_i)$ tel que $s_i \notin D_I$. Le programme obtenu est noté $\mathcal{P}^{\mathcal{R}}_{D_R|D_I}$.

$\mathcal{P}^{\mathcal{R}}_{D_R|D_I}$ est un programme positif. Il admet un plus petit modèle $\mathcal{M}$ qu'on peut subdiviser en $\mathcal{M}_{in} = \{p : \ in(p) \in \mathcal{M}\}$ et $\mathcal{M}_{out} = \{p : \ out(p) \in \mathcal{M}\}$.
Si $\mathcal{M}_{in} \cap \mathcal{M}_{out} = \emptyset$ et si $D_R = (D_I \cup \mathcal{M}_{in}) \backslash \mathcal{M}_{out}$, alors D_R est une révision $\mathcal{P}^{\mathcal{R}}$-justifiée de D_I.

Exemple 5.1 Soient $D_I = \{p, q\}$ et $\mathcal{P}^{\mathcal{R}} = \{in(p) \longrightarrow out(q); \ in(q) \longrightarrow out(p)\}$. Soit $D_R = \{p\}$. $\mathcal{P}^{\mathcal{R}}_{D_R} = \{in(p) \longrightarrow out(q)\}$ et $\mathcal{P}^{\mathcal{R}}_{D_R|D_I} = \{\longrightarrow out(q)\}$. Le plus petit modèle de $\mathcal{P}^{\mathcal{R}}_{D_R|D_I}$ est $\mathcal{M} = \{out(q)\}$ et $(D_I \cup \mathcal{M}_{in}) \backslash \mathcal{M}_{out} = D_R$. Donc, D_R

est une révision $\mathcal{P}^{\mathcal{R}}$-justifiée de D_I. Le lecteur peut vérifier que la base vide $D' = \emptyset$ n'est pas une révision $\mathcal{P}^{\mathcal{R}}$-justifiée de D_I. □

Selon cette sémantique de mise à jour, une base peut admettre une, plusieurs ou aucune révision.

Remarque 3 *Dans [MT95], les m.à.j utilisateur sont plongées dans le programme par des règles dont le corps est vide : l'insertion de p est modélisée par le rajout de la règle $\longrightarrow in(p)$, et la suppression de p est modélisée par la règle $\longrightarrow out(p)$.*

5.1 Comparaison

Comme les programmes de révision et les programmes de m.à.j ne partagent pas la même syntaxe, nous définissons dans ce qui suit une application $\mathcal{U}$ qui associe à chaque programme de révision un programme de m.à.j.

Définition 19 *Soit $\mathcal{P}^{\mathcal{R}}$ un programme de révision. $\mathcal{U}(\mathcal{P}^{\mathcal{R}})$ est le programme de révision obtenu en transformant chaque règle ρ de $\mathcal{P}^{\mathcal{R}}$ comme suit :*

1. *Si $Tête(\rho) = in(p)$, alors $Tête(\rho)$ est remplacée par $+p$ et le littéral $\neg{-p}$ est rajouté dans $Corps(\rho)$.*
2. *Si $Tête(\rho) = out(p)$, alors $Tête(\rho)$ est remplacée par $-p$ et le littéral $\neg{+p}$ est rajouté dans $Corps(\rho)$.*
3. *Si $in(p) \in Corps(\rho)$, alors il est remplacé par $(p \wedge \neg{-p}) \vee +p$.*
4. *Si $out(p) \in Corps(\rho)$, alors il est remplacé par $(\neg p \wedge \neg{+p}) \vee -p$.*

Exemple 5.1 **(Suite)** Pour $\mathcal{P}^{\mathcal{R}} = \{in(p) \longrightarrow out(q); in(q) \longrightarrow out(p)\}$, nous avons

$$\mathcal{U}(\mathcal{P}^{\mathcal{R}}) = \left\{ \begin{array}{rcl} p, \neg{-p}, \neg{+q} & \longrightarrow & -q \\ +p, \neg{+q} & \longrightarrow & -q \\ q, \neg{-q}, \neg{+p} & \longrightarrow & -p \\ +q, \neg{+p} & \longrightarrow & -p \end{array} \right\}$$

Le programme $\mathcal{U}(\mathcal{P}^{\mathcal{R}})$ est obtenu en utilisant la définition ci-dessus puis en éliminant les disjonctions du corps des règles par dépliage. □

L'exemple suivant illustre le lien établi par la proposition 11.

Exemple 5.1 **(Suite)** Nous pouvons facilement vérifier que D_I admet deux révisions ; $D_{R1} = \{p\}$ et $D_{R2} = \{q\}$. Considérons maintenant $\mathcal{P} = \mathcal{U}(\mathcal{P}^{\mathcal{R}})$.

$$\mathcal{P}_{\emptyset, D_I} = \left\{ \begin{array}{lrcl} \rho_{11}: & p, \neg{-p}, \neg{+q} & \longrightarrow & -q \\ \rho_{12}: & +p, \neg{+q} & \longrightarrow & -q \\ \rho_{21}: & q, \neg{-q}, \neg{+p} & \longrightarrow & -p \\ \rho_{22}: & +q, \neg{+p} & \longrightarrow & -p \\ \rho_{31}: & & \longrightarrow & p \\ \rho_{32}: & & \longrightarrow & q \end{array} \right\}$$

$\mathcal{P}_{\emptyset,D_I}$ admet deux modèles stables [GL88], $\mathcal{M}_1 = \{p, q, -p\}$ et $\mathcal{M}_2 = \{p, q, -q\}$. Les modèles stables étant des interprétations totales, nous les représentons donc par leurs parties positives. Par exemple, dans $\mathcal{M}_2$, $+q$ est faux. Notons que D_{R1} peut être obtenue en appliquant $\mathcal{M}_1$, i.e. $D_{R1} = \texttt{Apply}(\mathcal{M}_1, D_I)$ et $D_{R2} = \texttt{Apply}(\mathcal{M}_2, D_I)$. La proposition suivante généralise cette observation. $\square$

Remarque 4 *Quand [MT95] parle de programmes de révisions, les programmes considérés sont ceux où les m.à.j utilisateurs sont "déjà" incluses. Donc, pour comparer la révision d'une base Δ par $\mathcal{P}^{\mathcal{R}}$ et sa m.à.j par le programme $\mathcal{P} = \mathcal{U}(\mathcal{P}^{\mathcal{R}})$, nous comparons les révisions $\mathcal{P}^{\mathcal{R}}$-justifiées et le résultat de la m.à.j de la base Δ par le programme $\mathcal{P}$ en présence de la m.à.j vide.*

Proposition 11 *Soit Δ une base totale, $\mathcal{P}^{\mathcal{R}}$ un programme de révision et $\mathcal{P} = \mathcal{U}(\mathcal{P}^{\mathcal{R}})$ le programme de m.à.j associé. Soit $\mathcal{D}_{\mathcal{R}}$ l'ensemble des révisions $\mathcal{P}^{\mathcal{R}}$-justifiées de Δ. Alors, pour toute instance $D \in \mathcal{D}_{\mathcal{R}}$, il existe une interprétation $\mathcal{M}$ de $\mathcal{P}_{\emptyset,\Delta}$ telle que :*

- $\texttt{Apply}(\mathcal{M}, \Delta) = D$ *et*
- *$\mathcal{M}$ est un modèle stable [GL88] de $\mathcal{P}_{\emptyset,\Delta}$.*

La réciproque de cette proposition n'est pas vraie : certains modèles stables de $\mathcal{P}_{\emptyset,\Delta}$ ne définissent pas des révisions de Δ. Ceci est illustré par l'exemple suivant.

Exemple 5.2 Soit $D = \emptyset$ et $\mathcal{P}^{\mathcal{R}} = \{\longrightarrow in(p);\ \longrightarrow out(p)\}$. D n'admet aucune révision. Considérons maintenant $\mathcal{P} = \mathcal{U}(\mathcal{P}^{\mathcal{R}})$ le programme de m.à.j associé :

$$\mathcal{P}_{\emptyset,D} = \left\{ \begin{array}{llll} \rho_1 : & \neg{-p} & \longrightarrow & +p \\ \rho_2 : & \neg{+p} & \longrightarrow & -p \end{array} \right\}$$

Ce programme admet deux modèles stables : $\mathcal{M}_1 = \{+p\}$, et $\mathcal{M}_2 = \{-p\}$.
Néanmoins, $\texttt{Apply}(\mathcal{M}_1, D) = \{p\}$ n'est pas une révision de D. De même pour $\mathcal{M}_2$, $\texttt{Apply}(\mathcal{M}_2, D) = \emptyset$. $\square$

La proposition suivante donne une condition nécessaire et suffisante pour que le modèle stable du programme de m.à.j associé au programme de révision, définisse une révision.

Proposition 12 *Soit Δ une base totale, $\mathcal{P}^{\mathcal{R}}$ un programme de révision et $\mathcal{P} = \mathcal{U}(\mathcal{P}^{\mathcal{R}})$. Soit $\mathcal{M}$ un modèle stable de $\mathcal{P}_{\emptyset,\Delta}$.*

$\mathcal{M}$ définit une une révision $\mathcal{P}^{\mathcal{R}}$-justifiée de Δ ssi c'est un modèle stable du programme $\vec{\mathcal{P}}_{\emptyset,\Delta}$, où $\vec{\mathcal{P}}_{\emptyset,\Delta}$ est le programme obtenu à partir de $\mathcal{P}_{\emptyset,\Delta}$ en supprimant du corps de chaque règle ρ le littéral $\neg{+p}$ (resp. $\neg{-p}$) si $Tête(\rho) = -p$ (resp. $+p$).

Exemple 5.2 (Suite)

$$\vec{\mathcal{P}}_{\emptyset,D} = \left\{ \begin{array}{llll} \rho'_1 : & & \longrightarrow & +p \\ \rho'_2 : & & \longrightarrow & -p \end{array} \right\}$$

Nous pouvons voir que ni $\mathcal{M}_1$ ni $\mathcal{M}_2$ ne sont des modèles stables de $\vec{\mathcal{P}}_{\emptyset,D}$. □

La proposition 11 induit l'unicité de la révision justifiée par $\mathcal{P}^{\mathcal{R}}$ dès lors que le modèle bien fondé du programme de m.à.j qui lui est associé est total (puisqu'il est l'unique modèle stable).

Corollaire 3 *Soit $\mathcal{M}$ le modèle bien fondé de $\mathcal{P}_{\emptyset,\Delta}$. Si $\mathcal{M}$ est un modèle bivalué, alors* $\texttt{Apply}(\mathcal{M},\Delta)$ *est l'unique révision $\mathcal{P}^{\mathcal{R}}$-justifiée de Δ.*

La proposition suivante montre que toutes les révisions d'une base étendent la base que nous obtenons avec notre sémantique de m.à.j. Notre approche offre une approximation de la sémantique de m.à.j de [MT95]. Cette approximation présente l'avantage d'être calculable en temps polynômial alors que le calcul de la sémantique de [MT95] est exponentiel en temps.

Proposition 13 *Soit D_I une base, $\mathcal{P}^{\mathcal{R}}$ un programme de révision et $\mathcal{P} = \mathcal{U}(\mathcal{P}^{\mathcal{R}})$. Soit $\mathcal{D}_{\mathcal{R}}$ l'ensemble des révisions $\mathcal{P}^{\mathcal{R}}$-justifiées de D_I. Soit $\Delta = \texttt{Up\_Sem\_WF}(\emptyset, D_I, \mathcal{P})$. Alors, $\forall D_R \in \mathcal{D}_{\mathcal{R}}$: $V(\Delta) \subseteq D_R$ et $F(\Delta) \cap D_R = \emptyset$*

Marek & Truscszinski [MT95] ont caractérisé deux classes de programmes, à savoir les programmes sains (en anglais : *safe*) et les programmes MT_stratifiables[2], qui à toute base de données initiale D_I associent une révision “unique” D_R. De plus, pour tout programme appartenant à l'une de ces deux classes de programmes, il existe un algorithme en temps polynômial qui calcule D_R. Nous montrons qu'un programme $\mathcal{P}^{\mathcal{R}}$ est MT_stratifiable ssi $\mathcal{U}(\mathcal{P}^{\mathcal{R}})$ est stratifiable (dans le sens classique de [ABW88]).

Définition 20 (Programmes sains) *$\mathcal{P}^{\mathcal{R}}$ est sain ssi pour toute règle ρ, Tête$(\rho) = \alpha \Rightarrow \alpha^D \notin Var(\mathcal{P}^{\mathcal{R}})$. Où $\alpha^D = in(p)$ (resp. $out(p)$) si $\alpha = out(p)$ (resp. $in(p)$) et $Var(\mathcal{P}^{\mathcal{R}})$ est l'ensemble des littéraux clos apparaissant dans $\mathcal{P}^{\mathcal{R}}$.*

Il est important de remarquer ici que cette notion de programme sain est dépendante des m.à.j utilisateur. Nous rappelons qu'un programme de révision inclue les m.à.j utilisateur (voir remarque 4). Notre approche identifie clairement, programme de m.à.j d'une part et m.à.j utilisateur d'autre part. Donc, un programme de révision (avec m.à.j utilisateur) peut être sain mais le “même” programme avec une autre m.à.j utilisateur peut ne pas l'être (voir l'exemple 5.3 ci-dessous).

La condition de la proposition 4 portant sur les programmes de m.à.j est quant à elle indépendante des m.à.j initiales. Elle est donc, d'un point de vue pratique, bien plus satisfaisante que la condition ci-dessus portant sur les programmes de révision.

Exemple 5.3 Soit $\mathcal{P}^{\mathcal{R}} = \{in(p) \longrightarrow out(q)\}$. $\mathcal{P}^{\mathcal{R}}$ est sain. Supposons maintenant que l'utilisateur veut insérer q. En intégrant la règle $\longrightarrow in(q)$, le nouveau programme de révision n'est plus sain. □

2. [MT95] utilise le terme stratifiable dont la définition est naturellement différente de celle introduite dans le contexte de la programmation logique.

Nous rappelons dans ce qui suit la définition des programmes MT_stratifiables qui représente une classe plus large de programmes de révision.

Définition 21 (MT_Stratifiabilité) *Un programme de révision $\mathcal{P}^{\mathcal{R}}$ est* MT_stratifiable *ssi il existe une partition $\{\mathcal{P}_i^{\mathcal{R}}\}_{0<i<n}$ de $\mathcal{P}^{\mathcal{R}}$ telle que*

- *$\mathcal{P}_i^{\mathcal{R}}$ est sain, et*
- *si $\rho \in \mathcal{P}_i^{\mathcal{R}}$ et $T\hat{e}te(\rho) = \alpha$, alors il n'y a aucune règle $\hat{\rho} \in \cup_{j<i}\mathcal{P}_j^{\mathcal{R}}$ telle que α ou $\alpha^D \in Var(\hat{\rho})$.*

Exemple 5.4 Le programme suivant est MT_stratifiable mais il n'est pas sain.

$$\mathcal{P}^{\mathcal{R}} = \left\{ \begin{array}{lccc} \rho_1 : & in(q) & \longrightarrow & in(p) \\ \rho_2 : & out(p) & \longrightarrow & in(r) \end{array} \right\}$$

La partition $\mathcal{P}_1^{\mathcal{R}} = \{\rho_1\}$ et $\mathcal{P}_2^{\mathcal{R}} = \{\rho_2\}$ satisfait les conditions de la définition 21 mais $\mathcal{P}^{\mathcal{R}}$ ne satisfait pas celles de la définition 20. Remarquons aussi que si nous voulons insérer p, alors le fait de rajouter la règle $\longrightarrow in(p)$ à $\mathcal{P}^{\mathcal{R}}$ va donner un nouveau programme qui n'est pas MT_stratifiable. □

Proposition 14 *Soit $\mathcal{P} = \mathcal{U}(\mathcal{P}^{\mathcal{R}})$. $\mathcal{P}^{\mathcal{R}}$ est* MT_stratifiable $\Leftrightarrow$ *$\mathcal{P}$ est* stratifiable *(dans le sens de [ABW88]).*

Tout programme stratifiable admet un modèle bien fondé total. Si $\mathcal{M}$ est le modèle bien fondé de $\mathcal{P}_{\emptyset,\Delta}$ alors, le corollaire 3 implique que $\texttt{Apply}(\mathcal{M}, \Delta)$ est l'unique révision justifiée de Δ. Ceci offre une autre preuve que les programmes MT_stratifiables définissent des révisions uniques. Il est par contre bien connu que la classe des programmes logiques admettant un modèle bien fondé total est plus large que la classe des programmes stratifiables.

6 Comparaison avec d'autres travaux

[RL96] définit une classe de programmes où les règles peuvent être ordonnées de telle sorte qu'un fait supprimé ne peut être réinséré. Par exemple, dans $\mathcal{P} = \{\rho_1 : p \longrightarrow -q;\ \rho_2 : s \longrightarrow +q\}$, ρ_2 doit être considérée avant ρ_1. [RL96] montre l'équivalence entre la sémantique opérationnelle (non déterministe) et “un des modèles stables” d'un programme logique associé au programme de m.à.j. Ce travail avait pour objectif d'assurer la terminaison, mais le problème de la maintenance de l'intégrité est traité de façon peu convaincante. En effet, les règles qui font une insertion et celles qui font une suppression ne sont donc pas traitées de la même manière : seules les règles de suppression sont supposées maintenir des contraintes (c'est ce qui explique leur ordonnancement). Les autres règles sont considérées comme des règles déductives.

[LSV90] propose une sémantique pour les programmes Datalog$^{\neg *}$, qui autorise la négation en tête de règles. En se basant sur ce travail, [SVV92] propose une sémantique de m.à.j en présence de règles Datalog$^{\neg *}$. L'approche est basée sur l'hypothèse que les faits récents ont priorité sur les faits existants déjà dans la base. Par exemple, si nous avons $\mathcal{P} = \{p \longrightarrow \neg q\}$, alors en insérant q dans la base qui contient p, nous obtenons une base où p et q sont vrais ; la nouvelle information q "écrase" l'ancienne information : $\neg q$. En procédant de la sorte, le traitement de l'information contradictoire (la règle ci-dessus déduit $\neg q$ et la nouvelle information est q) est résolue en faveur de la nouvelle information. Remarquons que l'approche ne permet pas de gérer tous les cas de contradiction. En effet, soit $\mathcal{P} = \{p \longrightarrow \neg q;\ r \longrightarrow q\}$ et la base vide. Il est facile de voir que le fait d'insérer p et r implique un conflit au niveau de q. La solution proposée consiste à annuler la m.à.j (i.e. revenir à l'état initial).

[ALS95], tout en donnant la priorité aux faits récents, propose une approche différente. En considérant le programme $\mathcal{P} = \{p \longrightarrow \neg q\}$ et la base qui contient p, l'insertion de q implique, en utilisant $\mathcal{P}$, la suppression de p. [ALS95] impose des restrictions très fortes sur la syntaxe des programmes : les règles considérées ne contiennent qu'un seul littéral dans leurs corps.

[CW90] propose un algorithme de dérivation de règles de production à partir de spécifications de contraintes sous forme de requêtes SQL. Une base est cohérente si toutes ces requêtes retournent des réponses vides. L'avantage de cette approche sur notre proposition est que les contraintes considérées peuvent utiliser les opérateurs d'aggrégation (`COUNT, MAX`, ...). [CW90] montre qu'après toute m.à.j émise par l'utilisateur, si l'exécution du programme dérivé se termine, alors la base est cohérente. Si par contre le programme ne termine pas, la solution proposée est celle d'avorter la transaction et de revenir à l'état initial. Cette approche suppose que l'état initial est donc cohérent. Cette hypothèse n'est pas forcément vérifiée dans un cadre où les contraintes évoluent. Rappelons que dans le cadre de notre travail, nous n'avons pas supposé la base initiale cohérente. Remarquons aussi que les règles de [CW90] sont totalement ordonnées. Il est facile de voir que ce faisant, la propriété du changement minimal n'est pas garantie. Le travail de [CFPT94] est directement lié à celui de [CW90]. La différence entre les deux approches est que l'exécution des règles actives n'est plus ensembliste mais par nuplet. La propriété de confluence n'est donc plus garantie. Pour la terminaison, les auteurs proposent d'arrêter l'exécution des règles dès qu'un nuplet est supprimé par une règle après avoir été inséré par une autre règle car dans ce cas il y a *risque* de non terminaison.

Remerciements : Nous remercions les rapporteurs pour leurs remarques pertinentes qui nous ont permis d'améliorer la présentation de ce travail.

7 Bibliographie

[ABW88] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–142. Morgan Kaufmann, 1988.

[AHW95] A. Aiken, J. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *TODS*, 20(1):3–41, 1995.

[ALS95] M. Halfeld Ferrari Alves, D. Laurent, and N. Spyratos. Update rules in datalog programs. In *Proceedings of LPNMR95, Logic Programming and Non Monotonic Reasoning*, LNAI 928. Springer Verlag, 1995.

[AS91] S. Abiteboul and E. Simon. Fundamental properties of deterministic and non deterministic extensions of datalog. *Theoretical Computer Science*, 78(1):137–158, 1991.

[Bar94] C. Baral. Rule based updates on simple knowledge bases. In *Proceedings of AAAI'94*, Seattle, USA, August 1994.

[BCP94] E. Baralis, S. Ceri, and S. Paraboschi. Declarative specification of constraint maintenance. In *Proceedings of ER'94*, 1994.

[BCW93] E. Baralis, S. Ceri, and J. Widom. Better termination analysis for active databases. In *Proceedings of ER'93 conference*, 1993.

[BF91] N. Bidoit and C. Froidevaux. General logic databases and programs: Default logic semantics and stratification. *Information and Computation*, 91:15–54, 1991.

[BH89] N. Bidoit and R. Hull. Minimalism, justification and non-monotonicity in deductive databases. *Journal of Computer and System Science*, 38(2):290–325, 1989.

[Bid91] N. Bidoit. Negation in rule based database languages: A survey. *Theoretical Computer Science*, 78:85–112, 1991.

[BK94] A. J. Bonner and M. Kiefer. Transaction logic programming (or, a logic of procedural and declarative knowledge). Technical Report CSRI-270, University of Toronto, 1994.

[BM96] N. Bidoit and S. Maabout. Update rule programs versus revision programs. In *Proc. of NMELP workshop held in conjunction with JICSLP'96. To appear in LNAI series of Springer*, Bonn, Germany, 1996.

[BM97] N. Bidoit and S. Maabout. A model theoretic approach to update rule programs. In *Proc. of ICDT Conference*, LNCS, Delphi, Greece, Jan. 1997. Springer-Verlag.

[CC95] T. Coupaye C. Collet. Denotational semantics for an active rule execution model. In *Proceedings of RIDS'95*. Springer Verlag, 1995.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboshi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM TODS*, 19(2):367–422, 1994.

[Che91] W. Chen. Declarative specification of database updates. In *Proc. of DOOD'91*. Springer, 1991.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[Fag91] F. Fages. A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. *New Generation Computing*, 9(4), 1991.

[Fit85] M. Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2:295–312, 1985.

[Fit93] M. C. Fitting. The family of stable models. *Journal of Logic Programming*, 17:197–225, 1993.

[Fit95] M. Fitting. Annotated revision specification programs. In *Proceedings of LPNMR95, Logic Programming and Non Monotonic Reasoning*, number 928 in LNAI. Springer Verlag, 1995.

[FT95] P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *ACM TODS*, 20(4):414–471, 1995.

[Ger94] M. Gertz. On specifying the reactive behavior on constraint violations. In *Proceedings of RIDE'94*. IEEE, 1994.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *Proceedings of the 5th international symposium on logic programming*. MIT Press, 1988.

[Gre93] P.W.P.J. Grefen. Combining theory and practice in integrity control: A declarative approach to the specification of a transaction modification subsystem. In *Proc. of VLDB*, 1993.

[KM91] H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledgebase and revisiting it. In *Proceedings of Intrl. Conf. on Knowledge Represention and Reasoning (KR'91)*, 1991.

[LSV90] E. Laenens, D. Saccà, and D. Vermeir. Extending logic programming. In *Proceedings of ACM PODS*, 1990.

[Maa97] S. Maabout. *Règles de mise à jour dans les bases de données*. PhD thesis, Universitéde Paris13, en préparation., 1997.

[McC80] J. McCarthy. Circumscription- a form of non monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[Moo85] R. Moore. Semantical considerations on non monotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[MT93] V. Marek and M. Truszczynski. *Non Monotonic Logics; Context-Dependent Reasoning*. Springer Verlag, Berlin, 1993.

[MT95] V. Marek and M. Truszczynski. Revision programming, database updates and integrity constraints. In *Proceedings of the 5th International Conference on Database Theory. ICDT*, Prague, Jan. 1995.

[MW88] S. Manchanda and D. S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan Kaufmann, 1988.

[PT95] T. C. Przymusinski and H. Turner. Update by means of inference rules. In *Proceedings of the International Conf. on Logic Programming and Non Monotonic Reasoning, LPNMR'95*, LNAI, 1995.

[PV95] P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *Proceedings of PODS*, 1995.

[PV97] P. Picouet and V. Vianu. Expressiveness and complexity of active databases. In *Proceedings of ICDT'97*, 1997.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[RL96] L. Raschid and J. Lobo. Semantics for update rule programs and implementations in a relational database management system. *ACM TODS*, 21(4):526–571, December 1996.

[SAV95] A. Siebes, J.V. Akker, and M.V. Voort. (Un)decidability results for trigger design theories. Technical Report 1995, CWI, 1995.

[SK88] F. Sadri and R. Kowalski. A theorem-proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman Publishers, 1988.

[SSW94] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proccedings of ACM Sigmod Conference*, 1994.

[SVV92] D. Saccà, B. Verdonk, and D. Vermeir. Evolution of knowledge bases. In *Proceedings of EDBT92*, 1992.

[UD92] S.D. Urban and M. Desideiro. Context: a connstraint explanation tool. *Data & Knowledge Enginieering*, 8:153–183, 1992.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 2. Computer Science Press, 1989.

[vGRS91] A. van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38, 1991.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufman, 1996.

[Wid92] J. Widom. A denotational semantics for the starburst production rule language. *SIGMOD Record*, 21(3), 1992.

[Win86] M. Winslett. A model-theoretic approach to updating logical databases. In *PODS'86*, 1986.

[Zan95] C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *Proceedings of DOOD'95*, 1995.