Maintenance de la vérité dans les systèmes à base de connaissance centrée-objet.

Truth maintenance in object-centered knowledge-based systems.

Jérôme EUZENAT, François RECHENMANN (INRIA)

RÉSUMÉ -

Le raisonnement non monotone est souvent une conséquence de la connexion des systèmes à base de connaissance à des systèmes informatiques extérieurs. Ces derniers sont en effet susceptibles d'agir sur les données et les connaissances de la base. Les systèmes de maintenance de la vérité (truth maintenance systems) possèdent certaines fonctionnalités requises pour gérer la non monotonie. Ils sont évalués dans le contexte d'une utilisation des représentations centrées-objet. Les caractéristiques de ces dernières (héritage, attachement procédural, valeurs par défaut, attributs multi-valués), et en particulier du modèle retenu dans le système Shirka, amènent à des solutions spécifiques.

Mots clés : maintenance de la vérité, TMS, raisonnement non monotone, représentations centrées-objet.

ABSTRACT -

Non-monotonic reasoning is often a consequence of the connection between knowlege-based systems and application programs. These programs can make changes on data and knowledge contained in the base. Truth maintenance systems have been specifically developped to manage non-monotonic reasoning. They are evaluated regarding their use in the context of object-centered representations. Specific solutions are proposed in order to take into account the characteristics of these representations in general (inheritance, procedural attachment, default values, multi-valued slots) and of the representation model used in the Shirka system in particular.

Keywords : truth maintenance, TMS, non-monotonic reasoning, object-centered knowledge representations.

Laboratoire ARTEMIS/Imag, BP 68
38402 - Saint Martin d'Hères Cedex, France
Tel. 76 51 46 04

[1] Jérôme Euzenat, François Rechenmann
Maintenance de la vérité dans les systèmes à base de connaissance centrée-objet
Actes 6ième congrès AFCET-INRIA «Reconnaissance des Formes et Intelligence Artificielle», Antibes (FR), 16-20 novembre1987, pp1095-1109 (Dunod, Paris (FR), ISBN 2-04-013480-8)

Maintenance de la vérité dans les systèmes à base de connaissance centrée-objet.

Jérôme EUZENAT, François RECHENMANN (INRIA)

Introduction.

Les systèmes à base de connaissance sont amenés à s'ouvrir sur l'extérieur. De plus en plus fréquemment en effet, ces systèmes sont connectés à des programmes d'application, des modules de calcul numérique ou de visualisation graphique, des bases de données ou des capteurs. Ces environnements sont susceptibles d'agir sur la base de connaissances, aussi bien au niveau des connaissances factuelles que des connaissances destinées à l'inférence. Se pose alors le problème de la cohérence des inférences avant et après de telles modifications. Les résultats obtenus avant une modification peuvent-ils être conservés tels quels ou doivent-ils être annulés et réinférés dans le nouvel état de la base? La réponse prudente est positive mais entraîne des gaspillages de temps considérables.

La solution de ce problème de gestion des raisonnements dits non monotones consiste fondamentalement à maintenir un réseau de justifications des résultats obtenus. C'est ce que font les systèmes de maintenance de la vérité (ou TMS pour «truth maintenance systems»). Ces systèmes sont présentés indépendamment de tout formalisme de représentation des connaissances particulier. Il est cependant intéressant de les étudier dans le cadre d'une représentation particulière, en espérant par exemple améliorer ses performances en tenant compte des spécificités de cette représentation.

1. Les représentations de connaissance centrées-objet.

Les représentations de connaissance qui utilisent le concept d'objet peuvent être présentées à travers un certain nombre de caractéristiques communes, telles que la notion de classe et d'instance, de treillis de classes et d'héritage, d'attachement procédural et de valeur par défaut. Très vite cependant des différences d'interprétation apparaissent d'un modèle à l'autre. A des fins de concrétisation, le modèle retenu dans le système Shirka sert de support à cet exposé.

Shirka est un outil de développement de systèmes à base de connaissances [Rechenmann 85]. Son modèle de représentation s'inspire des «frames» dont il reprend certains mécanismes élémentaires, tels que l'attachement procédural ou l'héritage. Les connaissances y sont décrites dans des unités interdépendantes appelées *schémas*. A la différence de nombreux modèles actuels utilisant la notion d'objet, Shirka ne fait appel à aucun autre type d'unité de description. En particulier, il ne mélange pas règles et objets, car ses propres mécanismes d'inférence suffisent. Il est donc particulièrement adapté pour étudier l'adéquation des techniques de maintenance de la vérité aux représentations centrées-objet.

1.1. Schémas de classe et schémas d'instances.

Un schéma rassemble toutes les connaissances sur une famille d'objets (*schéma de classe*) ou sur un objet particulier d'une classe (*schéma d'instance*). Dans les deux cas, il possède des *attributs*. Dans un schéma de classe, les attributs sont définis à l'aide de *facettes*, la première étant la facette de typage qui introduit le type de l'attribut. Ce type peut être simple (entier, réel, booléen, chaîne de caractères ou symbole) ou faire référence à une autre classe par le nom du schéma qui la définit. Dans une instance, les attributs reçoivent des valeurs. Quand le type n'est pas simple, la valeur est le nom d'une instance de la classe définissant le type. Une instance est généralement incomplète: seuls certains attributs ont reçu une valeur. Les valeurs des attributs indéterminées peuvent alors être inférées à l'aide des connaissances renfermées dans le schéma de classe à laquelle appartient l'instance.

« Exemple 1.1.

On peut ainsi définir la classe **change** représentant une opération financière. Elle est composée de trois attributs monovalués, dont le type de l'un est défini par une autre classe (**devise**). L'objet **frais-honolulu** est une instance incomplète de **change**. Elle exprime le change de 23450 francs en dollars.

```
{change
   sorte-de        =       opération-finance ;
   sortie-franc    $un     entier ;
   devise          $un     devise ;
   entrée-devise   $un     entier }

{frais-honolulu
   est-un          =       change ;
   sortie-franc    =       23450 ;
   devise          =       dollar } »
```

En fait, les schémas de classe sont eux-mêmes des instances de *méta-schémas*, le plus souvent du méta-schéma **schéma**, instance de lui-même. De même, les attributs et les facettes sont des instances respectives des schémas **attribut** et **facette**, ou de l'une de leurs spécialisations. Cette homogénéité de l'implémentation est source de nombreux avantages, en particulier pour l'extension ou la particularisation du système.

1.2. Les mécanismes d'inférence.

Un attribut est en général défini par plusieurs autres facettes. Les *facettes de restriction* permettent par exemple de lui associer des intervalles, ou une liste de valeurs admissibles, ainsi que des prédicats que ses valeurs potentielles dans les instances doivent satisfaire (facette **a-vérifier**).

Mais d'autres facettes permettent d'attacher à l'attribut différents moyens de déterminer sa valeur dans une instance donnée si elle y est inconnue.

Ainsi, la facette **valeur** définit la valeur de l'attribut pour toutes les instances de la classe. Dans une instance particulière, cet attribut aura la valeur qu'elle a introduite. La facette **défaut** joue un rôle similaire, mais la valeur associée peut être redéfinie dans une instance.

L'*attachement procédural*, classique dans tous les systèmes faisant appel à la notion de «frame», est présent dans Shirka. Il présente cependant diverses originalités. La procédure attachée à l'attribut doit elle-même être définie d'un point de vue externe par un schéma de classe, dont les attributs sont ses paramètres. D'un point de vue interne, c'est une fonction Lisp dont le nom figure derrière la facette **valeur** de l'attribut **nom-fct** du schéma de classe la définissant. Appeler cette procédure c'est donc instancier ce schéma de classe et transmettre l'instance ainsi créée à la fonction. Cette dernière peut alors accéder à ses paramètres d'entrée et effectuer les calculs requis. Si l'attachement procédural est employé dans la facette **sib-exec**, la fonction complète l'instance par les valeurs des attributs associés à ses paramètres de sortie. Ces valeurs sont alors propagées à d'autres attributs du schéma faisant appel à cet attachement procédural par le biais de variables, elles-mêmes attachées aux attributs par des facettes spécifiques.

« Exemple 1.2.

On peut redéfinir la classe **change** de façon à ce qu'elle permette le calcul du montant de devise changé à partir de la somme en francs. Cet attachement procédural utilise la méthode **calc-change** qui elle-même emploie la fonction Lisp **div**.

```
{ change
   sorte-de        =       opération-finance ;
   sortie-franc    $un     entier ;
   devise          $un     devise ;
   entrée-devise   $un     entier
                   $sib-exec
                           {calc-change
                                   francs          $var<-          sortie-franc ;
                                   parité          $valeur         7.10 ;
                                   montant         $var->          entrée-devise }}
```

```
{calc-change
  sorte-de      =       méthode ;
  nom-fct       $valeur div ;
  francs        $un     réel ;
  parité        $un     réel ;
  montant       $un     réel } »
```

Ce mécanisme, apparemment lourd, présente des avantages liés au processus d'instanciation. En effet, lors de l'instanciation, les valeurs des attributs du schéma de classe de la procédure sont recherchées, en utilisant les différents mécanismes d'inférence de Shirka, dont l'attachement procédural. Un paramètre peut ainsi recevoir une valeur par défaut ou être à son tour calculé. Des restrictions peuvent être introduites, comme dans n'importe quel schéma de classe. De plus, si les valeurs des attributs qui correspondent aux paramètres d'entrée de la procédure ne peuvent être obtenues, la procédure n'est pas appelée. Si une autre procédure figure à sa suite dans la facette d'attachement procédural, elle est alors à son tour essayée.

Le *filtrage* est un mécanisme d'inférence propre à Shirka. Il consiste à rechercher des instances satisfaisant une description donnée sous la forme d'un schéma de classe et à extraire des instances trouvées des valeurs qui deviendront, par le biais de variables, les valeurs des attributs indéterminés. Le filtrage est mis en œuvre par la facette **sib-filtre** qui peut contenir plusieurs filtres essayés séquentiellement.

« Exemple 1.3.

Une nouvelle définition de la classe **change** utilise un filtre pour obtenir le cours d'une devise . Le filtre recherche une instance de la classe **taux-vente** dont l'attribut **monnaie** correspond à l'attribut **devise** de la classe **change** et ramène pour valeur de **parité** la valeur de son attribut **taux**.

```
{change
  sorte-de      =       opération-finance ;
  sortie-franc  $un     entier ;
  devise        $un     devise ;
  entrée-devise $un     entier
                $sib-exec
                        {calc-change
                                francs  $var<-  sortie-franc ;
                                parité  $sib-filtre
                                                {taux-vente
                                                        monnaie         $var<-  devise ;
                                                        taux            $var->  parité } ;
                                montant $var-> entrée-devise }}
```

Dans notre exemple **frais-honolulu**, la mise en correspondance sera faite avec l'instance **taux-dollar**, et la valeur utilisée par l'attachement procédural **calc-change** sera 7.10.

```
{taux-dollar
  est-un        =       taux-vente ;
  monnaie       =       dollar ;
  taux          =       7.10 } »
```

1.3. Le treillis de spécialisation et l'héritage.

Les classes d'une base de connaissances Shirka sont organisées en une structure de *demi-treillis*, dont l'élément supérieur est la classe **objet**. Une classe domine dans ce treillis une ou plusieurs autres sous-classes qui la spécialisent. L'ensemble des instances potentielles des sous-classes est plus restreint que l'ensemble des instances potentielles de leur classe supérieure. Au niveau des attributs, cette spécialisation se traduit par le fait qu'une sous-classe peut posséder des attributs que la classe supérieure ne possédait pas, mais que les attributs existants ne peuvent être que précisés et non redéfinis. Une même classe peut en fait spécialiser simultanément plusieurs autres classes, suivant les mêmes conditions.

Sur cette structure de treillis s'applique un mécanisme d'*héritage* classique. Une classe hérite de tous les attributs des classes supérieures et pour un attribut donné des facettes que possède cet attribut dans ces sur-classes. Cependant, l'héritage fonctionne différemment selon les facettes. Par exemple,

lors de la détermination de la valeur d'un attribut dans une instance, les facettes **valeur**, **sib-filtre**, **sib-exec** et **défaut** sont essayées dans cet ordre. Si toutes échouent, le même ordre d'essais est appliqué aux classes supérieures suivant un parcours en profondeur d'abord. Par contre, les prédicats des facettes **a-vérifier** se cumulent: une valeur d'attribut doit satisfaire tous les prédicats attachés à l'attribut de même nom dans les classes supérieures.

1.4. Problèmes de performances et solutions.

Dans Shirka, quand la valeur d'un attribut d'instance est indéterminée et qu'elle est obtenue par inférence, elle n'est pas rangée dans l'instance. Une autre demande de la valeur du même attribut d'instance provoquera donc une nouvelle inférence de la valeur. La raison en est qu'entre temps la base a pu être modifiée, au niveau des instances ou des classes, et que les conditions d'inférence de la valeur ont pu changer. L'inefficacité vient du fait qu'il est tout aussi possible que la base n'ait pas changé, auquel cas la valeur précédente est toujours valable. Tout au contraire, le procédé dit de «value caching», par analogie avec les mémoires cache, consiste à conserver au sein de l'attribut la valeur inférée comme une valeur normale. Ce procédé est inadapté à toute utilisation *non monotone* de la base de connaissance c'est-à-dire quand la modification (remplissage, suppression, remplacement) de la valeur d'un attribut entraîne la remise de valeurs d'autres attributs [AI 80]. L'exemple suivant illustrera le problème:

« Exemple 1.4.

Supposons un fragment de base qui peut calculer les sorties de devises nécessaires à une entreprise. Il est capable, étant donné un nombre de francs et le taux de change, d'inférer ces sorties. La classe **change** est définie comme précédemment.

Un appel de **(val? frais-honolulu entrée-devise)** va entraîner le calcul nécessaire du fait de la facette d'attachement procédural **sib-exec** associée à l'attribut **entrée-devise** de la classe **change**, à laquelle appartient l'instance **frais-honolulu**. La réponse (3302.81) sera fournie à l'utilisateur. Le mécanisme de «caching» permettrait de conserver cette valeur au sein de l'instance **frais-honolulu**. Malheureusement, le taux de change est variable et la valeur valide un jour ne l'est plus le lendemain. »

Il est envisageable de conserver les valeurs inférées à l'aide d'une facette $si-succes et de maintenir la cohérence grâce aux facettes réflexe. Cependant la construction d'une base deviendrait beaucoup trop compliquée, sans compter qu'à l'insertion d'une nouvelle classe il y a risque d'oubli d'un certain nombre de dépendances.

C'est pourquoi il faut considérer le problème de la non monotonie à l'échelle du système. Le contrôle du raisonnement non monotone a déjà été étudié indépendamment d'un système de raisonnement particulier. Il s'agit des *systèmes de maintenance de la vérité* . Ces travaux doivent être évalués dans l'optique d'une adaptation possible au problème du «caching» dans les représentations centrées-objet.

2. Les systèmes de maintenance de la vérité.

Au cours d'un raisonnement non monotone, il se peut que des entités manipulées par le module de raisonnement soient ôtées ou ajoutées à la base de connaissance. Dès lors que ces entités ont été précédemment utilisées dans des inférences, les conclusions auxquelles ont conduit leur utilisation devront être invalidées. Récursivement, les conclusions auxquelles ont contribué ces conclusions devront à leur tour être invalidées.

Le problème posé est de savoir à quelles entités l'utilisation par le module de raisonnement d'une entité précise a mené. Pour ce faire, à la fin des années 70, on s'est avisé d'enregistrer les *dépendances* entre les objets représentant ces entités [De Kleer 79].

L'autre problème est que l'ajout d'une granule de connaissance dans une base peut rendre la base inconsistante (ce qu'exclut le raisonnement monotone). On a donc cherché à remettre en cause les connaissances insérées dynamiquement. L'idée de base est de faire un retour-arrière dans le raisonnement, tel qu'on le trouve par exemple dans les interprètes Prolog, mais incluant le retrait des objets inférés au cours du raisonnement.

Le mécanisme de retour-arrière chronologique étant très coûteux en temps, on a alors cherché à l'optimiser et ainsi est né le *retour-arrière dirigé par les dépendances* [Stallman 77], où l'on ne remet pas en cause le dernier objet inféré, mais un objet qui a réellement contribué à une contradiction.

[De Kleer 86a] propose une comparaison des différents systèmes de retour-arrière.

2.1. TMS à propagation.

Jon Doyle a décrit le premier un TMS complet que l'on qualifiera de TMS à propagation car il enregistre l'état des entités manipulées par un système et propage la validité ou l'invalidité d'une entité à chaque modification. Il a introduit l'idée d'un TMS indépendant des systèmes de raisonnement en généralisant des idées déja implémentées dans ARS [Stallman 77]. Le système est exposé ici en s'appuyant sur [Doyle 79b].

Le principe du TMS à propagation est d'enregistrer pour chaque fait inféré une justification de l'inférence ainsi que les dépendances entre cette justification, ses antécédents (faits qui ont permis l'inférence) et son résolvant. A chaque modification opérée sur la base de faits la modification est propagée dans tout le graphe de dépendance.

2.1.1. Proposition théorique.

Doyle redéfinit le raisonnement comme le fait de trouver les raisons de chaque attitude (croyance, but, action). Ainsi un système de maintenance des croyances ne doit pas s'occuper des attitudes mais des raisons. La conséquence de cette affirmation est que le TMS ne doit pas s'occuper de la vérité des assertions, c'est-à-dire de la sémantique propre au système de raisonnement, mais des raisons de croire en ces assertions.

Il propose donc les bases théoriques suivantes: une *raison* pour une attitude A est une paire d'ensembles d'attitudes. Le premier est appelé ensemble IN, le second ensemble OUT. Si A a au moins une raison acceptable, A fait partie de l'ensemble des croyances courantes; on dira que A est *IN*. Sinon (A n'a pas de raison acceptable), A sera *OUT*. Une raison est *acceptable* ssi toutes les attitudes de son ensemble IN sont IN et toutes les attitudes de son ensemble OUT sont OUT.

2.1.2. Implémentation du système.

Le TMS manipule et maintient deux sortes d'entités: les *nœuds*, qui représentent les attitudes et les *justifications* des nœuds. A un nœud est associé un ensemble de justifications. La structure des justifications est la suivante (SL <INliste> <OUTliste>). Une SL-justification est dite valide ssi tous les nœuds de sa INliste sont IN et tous les nœuds de sa OUTliste sont OUT.

On peut associer divers noms à des nœuds soutenus par des justifications précises, ainsi (SL () ()) correspondra à une *prémisse* du raisonnement, (SL INliste ()) correspond à une *déduction* classique et (SL () OUTliste) est une assertion valide par *défaut*. Cette possibilité est un problème pour le TMS puisqu'elle autorise les paradoxes (nœud N justifié par (SL () (N)) i.e. $\neg N \Rightarrow N$) qu'il ne maîtrise évidement pas.

Le système a trois actions fondamentales sur ces entités: *créer* de nouveaux nœuds, *ajouter* ou *supprimer* une justification pour un nœud, *marquer* un nœud comme contradictoire.

2.1.3. Le mécanisme de la maintenance.

L'algorithme utilisé par Jon Doyle procède en profondeur d'abord, il est muni d'un retour-arrière dirigé par les dépendances. Ce retour-arrière est utile au cas où les justifications fournies au système permettent de justifier des contradictions.

Le système de raisonnement signale au TMS les inférences faites sous forme d'attitudes et de justifications pour ces attitudes. A chaque nouvelle attitude, le TMS fait correspondre un nœud. Tout le travail du TMS intervient quand c'est une justification qui lui est fournie. Il se charge alors d'ajouter la justification à celles du nœud N en question. Si cet ajout permet de valider le nœud, cette validation est entérinée et propagée à toute la base. Tous les nœuds qui peuvent être justifiés à l'aide du nœud N et tous ceux qui peuvent être invalidés à cause de lui sont examinés.

Cette nouvelle justification peut amener à justifier un nœud marqué comme étant contradictoire. Le TMS se chargera alors, grâce à un retour-arrière dirigé par les dépendances, d'éliminer de la base un des nœuds justifiant la contradiction, et par là-même de faire disparaître la contradiction de la base.

A noter que le système gère certaines circularités qui peuvent apparaître dans les dépendances entre nœuds, à l'exception des paradoxes qui font boucler le programme.

2.1.4. Limitations.

Les limitations attachées à ce système sont essentiellement fondées sur la lourdeur du mécanisme de retour-arrière. En effet, celui-ci va remettre en cause arbitrairement l'un des fondements d'une contradiction. Or il ne sait pas quel est le fondement qui entraîne véritablement la contradiction. Plus tard, il est possible qu'un autre fondement soit mis en cause parce qu'il a entraîné une nouvelle contradiction, et que le premier ne soit pas réhabilité. D'autre part, le mécanisme de retour-arrière dirigé par les dépendances est très lourd dans son action. Il change parfois inconsidérément l'état des nœuds et provoque de nouvelles contradictions au cours de sa résolution. Le TMS en devient très lent. Il s'en suit que le raisonnement hypothétique n'est pas vraiment naturel, car passer d'un état à l'autre relance tout le travail de propagation.

Toutes les critiques du système de Doyle ont été détaillées dans [De Kleer 83].

2.2. TMS à contextes.

Johan De Kleer a donc été amené à proposer son propre système de maintenance de la vérité. Ce système appelé ATMS (pour «assumption-based TMS») constitue l'archétype des TMS à contextes. Partant des critiques formulées plus haut, son système permet de raisonner avec plusieurs ensembles d'hypothèses simultanément. Nous qualifierons ce système de TMS à contextes, car au lieu d'enregistrer la validité d'une entité, il enregistre le contexte dans lequel cette entité est valide.

2.2.1. L'idée de l'ATMS.

L'idée de De Kleer est que le TMS de Doyle, en raisonnant sur les ensembles d'états, limite l'espace de recherche du module de raisonnement (qui n'utilisera pas de nœuds OUT). Pour lui, ce n'est pas au TMS de décider de l'espace, mais au module de raisonnement. Afin de permettre à celui-ci d'avoir accès à tous les états possibles, il propose d'enregistrer les ensembles d'hypothèses sous lesquelles un nœud est IN plutôt que le fait qu'un nœud soit IN. Le système envisagé permettra de raisonner simultanément avec divers ensembles de suppositions, en s'inspirant en partie de [Martins 83].

2.2.2. Données manipulées.

Le principe du système est qu'au lieu d'enregistrer l'état de chaque fait (IN/OUT), on enregistre les ensembles d'hypothèses nécessaires sous lesquelles un fait est valide. Ces ensembles sont appelés environnements. La liste des environnements validant un fait est appelé *label*.. Le système manipule pour chaque nœud la structure <Donnée, Label, Justifications>. Le label ayant les propriétés suivantes:
- complétude (quelque soit un ensemble E de faits permettant de dériver un fait F, il existe un sous-ensemble de E dans le label de F).
- minimalité (un environnement d'un label ne peut en inclure un autre).
- consistence (un environnement d'un label ne peut supporter de contradiction).
- correction (un environnement d'un label permet de dériver le fait du label).

2.2.3. Principe du système.

A un moment donné, le module de raisonnement a fourni à l'ATMS un certain nombre de nœuds et un certain nombre de justifications. Le temps du raisonnement est représenté par l'ensemble de justifications J que le système de raisonnement a fourni à l'ATMS. Cet ensemble représente en effet toutes les inférences que le système a produit.

L'algorithme utilisé par l'ATMS est un algorithme en largeur d'abord qui permet de connaître tous les environnements soutenant chaque fait. L'essentiel du travail est à exécuter lors de l'établissement des labels, ceux-ci sont modifiés à chaque nouvelle justification fournie au système par le module de raisonnement.

Le système de raisonnement vérifie la validité d'un fait dans son contexte courant (ensemble d'axiomes) en faisant une mise en correspondance du contexte avec les environnements du label de ce fait.

La différence entre un TMS à propagation et un TMS à contextes réside essentiellement dans le fait que le premier mémorise la validité d'une entité, alors que le second mémorise les contextes dans lesquels une entité est valide. Il en découle que le premier est obligé de propager les invalidités à chaque changement de la base alors que le second doit à chaque utilisation d'une entité vérifier si celle-ci est valide dans le contexte courant.

3. Application aux représentations centrées-objet.

Le point de départ de notre recherche est d'utiliser le mécanisme d'un TMS dans une représentation de connaissances centrée-objet afin d'en améliorer les performances. Dans un premier temps, les avantages et inconvénients des TMS classiques pour les représentations de connaissance centrées-objet ont été évalués, puis les réalisations de différents systèmes permettant d'implémenter le «caching» ont été étudiées.

3.1. TMS classiques et représentation de connaissances centrée-objet.

Il est donc nécessaire d'identifier les différences entre les systèmes de raisonnement classiques, essentiellement à base de règles, et les représentations de connaissance centrées-objet, afin de comprendre les aménagements à apporter aux mécanismes précédemment décris . Après avoir choisi un type de TMS, nous verrons que les représentations de connaissance centrées-objet obligent à la maintenance des règles de raisonnement et que par contre deux gros problèmes des TMS disparaissent.

3.1.1. Suppositions et contextes.

L'ATMS de De Kleer est attrayant. En effet, il est intéressant de pouvoir changer facilement de contexte, d'avoir constamment sous la main tous les mondes minimaux, consistants et complets.

Certaines représentations de connaissance centrées-objet (dernières versions de ART et KEE par exemple) utilisent un TMS à contextes. Ce n'est pas pour l'amélioration des performances du système, mais pour la gestion cohérente et plus économe en mémoire des différents mondes («views») possibles dans lesquels évolue chaque objet. Un TMS à propagation serait inadapté à ce problème, qui est tout à fait en rapport avec les contextes.

Cependant un aspect de l'ATMS pose un problème vis-à-vis de l'objectif que nous nous fixons d'emblée. Nous voulons que l'utilisateur ne se rende pas compte de la présence du TMS et que celui-ci donne des réponses avec une rapidité accrue. Or, l'ATMS, pour vérifier la validité d'une valeur, cherche à savoir si elle est dans le contexte (i.e. si ses antécédents sont valides). Cette vérification prend un certain temps, parfois autant que le calcul lui-même, alors que la réponse est instantanée si l'on conserve l'ensemble des états valides.

D'autre part, Shirka n'intégrant pas la notion de mondes possibles, les contextes ne sont pas aussi utiles que dans le cas des systèmes multimondes. C'est pourquoi, au TMS à contextes nous préférerons un TMS à propagation comme celui de Doyle.

3.1.2. Maîtrise du système de raisonnement.

Dans les systèmes de Doyle et De Kleer, le système de raisonnement est séparé du TMS. Les systèmes de raisonnement considérés sont supposés figés, c'est-à-dire qu'ils utilisent toujours les même règles. Ils n'ont donc besoin d'enregistrer que des dépendances entre données. Nous considérons par contre que les règles d'inférence sont susceptibles d'évoluer comme les données et par conséquent qu'il est utile de les maintenir. Au sein des représentations de connaissance centrées-objet telles que Shirka, les inférences sont dirigées par les facettes, qui définissent les moyens de calculer une valeur. Ces facettes sont susceptibles d'être modifiées au cours du temps. Aussi serons-nous amenés à proposer des solutions pour maintenir les facettes qui sont les règles de notre système de raisonnement.

3.1.3. Justifications par défaut et paradoxes.

Les représentations de connaissance centrées-objet ayant leur propre mécanisme de valeur par défaut, il est inutile d'ajouter cette fonction au TMS. Cela nous conduit à considérer les justifications dans les représentations de connaissance centrées-objet comme des listes de données (INliste). Cette absence d'OUTliste dans nos dépendances évite d'introduire des paradoxes que le TMS ne saurait gérer.

3.1.4. Les contradictions dans les représentations de connaissance centrées-objet.

Trois situations peuvent être perçues comme introduisant des contradictions: on tente d'affecter une valeur à un attribut déja valué, on tente d'affecter une valeur à un attribut ayant une valeur fixe (déclarée par la facette $valeur), deux méthodes ne rendent pas le même résultat. Cependant, en y regardant de plus près, ces contradictions n'apparaissent jamais dans une représentation centrée-objet classique, ou bien ne sont pas considérées comme des contradictions, c'est-à-dire qu'elles sont gérées par le système. Dans les trois cas le système a en effet un comportement prédéfini:

a) Si on désire remplacer une valeur par une autre, tout se passe bien. Si on désire ajouter une valeur à celle qui est déjà présente, tout est prévu dans le système: le concepteur de la base aura pris soin de signaler l'attribut comme étant multivalué.
b) Le second cas caractérise une mauvaise définition de la base par son concepteur. En effet, il aurait du utiliser une facette *defaut* à la place de *valeur*, ce qui aurait permis de modifier la valeur de l'attribut.
c) Enfin le dernier cas n'apparaît jamais dans les représentations centrées-objet. Les méthodes de calcul sont en effet invoquées dans un ordre dépendant du graphe d'héritage et de la priorité entre les différents types de méthode et le calcul s'arrête au premier résultat trouvé.

L'idée même de contradiction a donc été bannie des représentations centrées-objet. Le processus qui permet d'attribuer une valeur à un attribut y est en effet déterministe et ne pose pas de problème. Cependant, les réponses données ici mettent en évidence l'importance de la conception de la base qui fige ce déterminisme. Si un TMS est utile dans une représentation de connaissance centrée-objet, ce ne sera donc pas par sa faculté à résoudre les contradictions.

On peut se demander si le nom de TMS est toujours adapté à un système qui ne gère plus les contradiction ni les valeurs par défaut. Cependant parce que nous nous plaçons dans la lignée des travaux de Doyle et par analogie avec son système d'enregistrement et de propagation nous conserverons ce nom.

3.2. «Caching» et maintenance.

C'est en étudiant plus profondément le «caching» qu'on réalisera la pleine utilité du TMS. C'est en effet l'introduction du «caching» dans les représentations centrées-objet qui nécessite la maintenance des croyances du système et non la représentation elle-même. On va donc exposer ici les différentes manières de maintenir une représentation des connaissances centrée-objet intégrant le «caching».

3.2.1. Le «caching» sans maintenance.

Le «caching» est implémenté de manière brute dans le système SRL (Scheme Representation Langage [Wrigth 83]). Un drapeau indique au système s'il doit ou non conserver la valeur inférée. Une telle utilisation du «caching» est très utile dans le cas où les valeurs inférées sont définitives.

« Exemple 3.1.

Dans l'exemple 1.1, un appel de **(val? frais-honolulu entrée-devise)** va lancer le calcul nécessaire. La réponse (3302.81) sera fournie à l'utilisateur. Si celui-ci a de nouveau besoin du montant en dollars des francs à changer, la réponse à sa demande se fera plus vite puisque la valeur déjà calculée se trouve maintenant directement dans le schéma d'instance **frais-honolulu**:

```
{frais-honolulu
      est-un          =     change ;
      sortie-franc    =     23450 ;
      devise          =     dollar ;
      entrée-devise   =     3302.81 } »
```

Cependant, ce «caching» brutal présente quelques inconvénients. En effet, le cours du change a la propriété de... changer, ainsi si le lendemain notre utilisateur veut savoir ce qu'il va lui en coûter, la réponse fournie (3302.81) aura toutes les chances d'être fausse. Il va donc devenir nécessaire de maintenir la validité des valeurs inférées.

3.2.2. Maintenance sur les arguments.

Une solution à ce problème est implémentée dans la première version de ART (Advanced reasoning Tool [Williams 84]) et SYPRUC (Système pour la représentation et l'utilisation des

connaissances [Chehire 86]). Elle consiste à enregistrer les dépendances entre les données utilisées à chaque inférence et à propager les invalidations quand une valeur est modifiée. Il s'agit donc typiquement de la méthode proposée par Doyle. L'exemple précédent donnerait à la fin du calcul:

« Exemple 3.2.

```
{taux-dollar
        est-un          =       taux-vente ;
        monnaie         =       dollar ;
        taux            =       7.10
                /utilisé-par frais-honolulu.entrée-devise/ }

{frais-honolulu
        est-un          =       change ;
        sortie-franc    =       23450 ;
        devise          =       dollar ;
        entrée-devise   =       3302.81 }
```

Ainsi à la modification de **taux-dollar.taux** en 7.05, la valeur de **frais-honolulu.entrée-devise** sera invalidée:

```
{frais-honolulu
        est-un          =       change ;
        sortie-franc    =       23450 ;
        devise          =       dollar }
```

et une seconde demande **(val? frais-honolulu entrée-devise)** entrainera un nouveau calcul des francs en dollars. »

Cependant la maintenance des dépendances entre valeurs ne suffit pas. En effet, si l'on imagine qu'au lieu de modifier **taux-dollar.taux**, c'est la méthode **calc-change** qui a été modifiée parce qu'une nouvelle réglementation fait payer des pénalités aux personnes qui changent plus de 20000 francs, la valeur calculée n'est plus valide et pourtant elle resterait encore valeur de **frais-honolulu.entrée-devise**.

3.2.3. Maintenance sur les méthodes.

La solution la plus simple est celle envisagée dans PAUL [Hein 83]. Elle consiste en la mémorisation de la méthode utilisée, de la même manière que les données. Ce qui donnera pour notre exemple:

« Exemple 3.3.

```
{calc-change
        sorte-de        =       méthode ;
        util-par        =       frais-honolulu.entrée-devise ;
        nom-fct         =       div ;
        francs          $un     réel ;
        parité          $un     réel ;
        montant         $un     réel }

{taux-dollar
        est-un          =       taux-vente ;
        monnaie         =       dollar ;
        taux            =       7.10
                /utilisé-par frais-honolulu.entrée-devise/ }

{frais-honolulu
        est-un          =       change ;
        sortie-franc    =       23450 ;
        devise          =       dollar ;
        entrée-devise   =       3302.81 }
```

ainsi, si le schéma **calc-change** subit une modification, on invalidera les valeurs qu'il a servi à calculer. Une fois de plus on aura

```
{frais-honolulu
  est-un        =       change ;
  sortie-franc  =       23450 ;
  devise        =       dollar }
```

et une seconde demande de **(val? frais-honolulu entrée-devise)** entrainera un nouveau calcul de **entrée-devise.** »

Mais cette méthode pose un dernier problème soulevé dans [Hein 83]. Il est possible qu'entre deux calculs ce soit non plus une méthode qui ait été modifiée mais la précédence des méthodes dans le graphe d'héritage. En voici un exemple:

« Exemple 3.4.

Si la réglementation a été modifiée de nouveau afin de faciliter l'exportation et que le seuil soit placé à 100000 francs et la pénalité à 0.03% pour les entreprises, on réagira, soit en modifiant l'arbre d'héritage et en ayant:

```
{change-entr
        sorte-de        =       change ;
        entrée-devise   $sib-exec
                                (calc-change-entr
                                        francs  $var<-  sortie-franc ;
                                        parité  $sib-filtre
                                                        (taux-vente
                                                                monnaie         $var<-  devise ;
                                                                taux            $var->  parité } ;
                                        montant $var-> entrée-devise }}

{frais-honolulu
        est-un          =       change-entr ;
        sortie-franc    =       23450 ;
        devise          =       dollar }

{calc-change-entr
        sorte-de        =       méthode ;
        nom-fct         =       calc-change-entr ;
        francs          $un     réel ;
        parité          $un     réel ;
        montant         $un     réel }
```

soit en insérant la méthode **calc-change-entr** dans le graphe déja existant. Une fois de plus, la valeur enregistrée dans **frais-honolulu.entrée-devise** ne sera pas invalidée, et donc un nouvel appel rendrait une valeur fausse. »

3.2.4. Maintenance sur le graphe d'héritage.

Il ne s'agit plus d'enregistrer une quelconque structure, mais d'un algorithme qui, à l'insertion d'une nouvelle méthode dans le graphe d'héritage, va invalider les résultats obtenus par l'utilisation d'une méthode moins prioritaire. Cet algorithme oblige à enregistrer pour toute valeur inférée la méthode grâce à laquelle elle l'a été. Dans l'exemple précédent on aurait donc eu la mémorisation suivante:

« Exemple 3.5

```
{calc-change
        sorte-de        =       méthode ;
        util-par        =       frais-honolulu.entrée-devise ;
        nom-fct         $valeur div ;
        francs          $un     réel ;
        parité          $un     réel ;
        montant         $un     réel }
```

{taux-dollar
est-un = taux-vente ;
monnaie = dollar ;
taux = 7.10
/utilisé-par frais-honolulu.entrée-devise/ }

{frais-honolulu
est-un = change ;
sortie-franc = 23450 ;
devise = dollar ;
entrée-devise = 3302.81
/calculé-par change.calc-change/ }

Ainsi à l'installation de la nouvelle méthode **calc-change-entr**, on peut descendre l'arborescence et invalider toutes les valeurs qui ont été obtenues à l'aide d'une méthode invalidée par cette dernière. On invalidera donc la valeur inférée grâce à calc-change. »

4. Proposition de réalisation

4.1. Intérêt de chaque système.

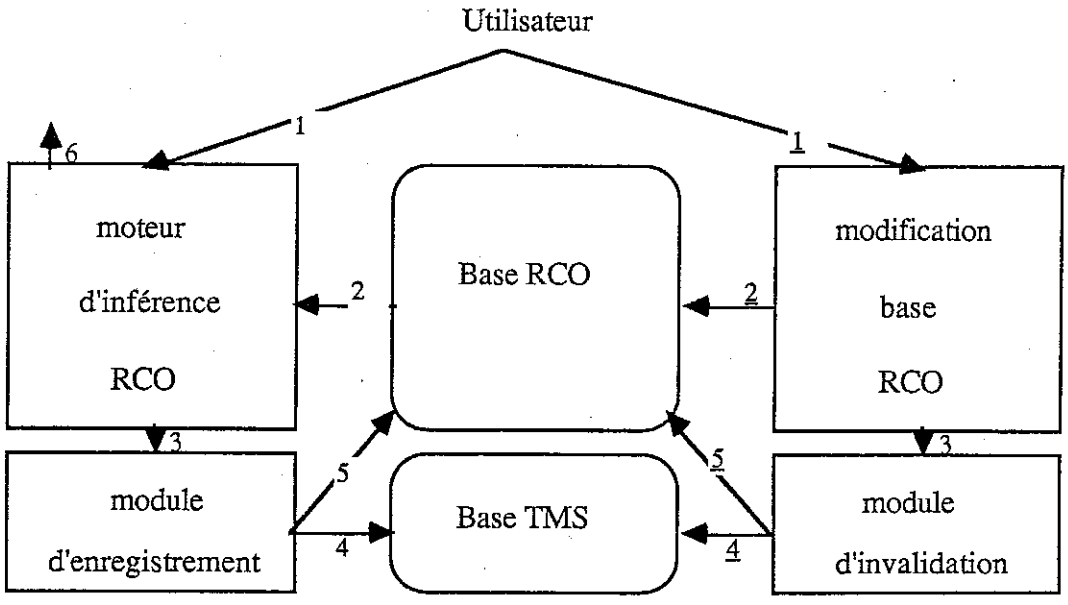
On peut se rendre compte que chacun des systèmes présentés a son intérêt dans un contexte bien précis, ainsi:

- Le «caching» sans maintenance est parfaitement adapté à l'utilisation monotone d'une base figée.
- La maintenance sur les valeurs est adaptée à une utilisation non monotone sur une base elle aussi figée.
- La maintenance sur les méthodes est intéressante pour la mise au point d'une base de connaissance dont on désire encore modifier les méthodes de calcul.
- La maintenance sur le graphe d'héritage est utile pour l'utilisation d'une base dont la structure (graphe d'héritage) risque d'évoluer ou de s'enrichir de nouvelles méthodes.

Au vu de ces différentes utilisations possibles du système il semble raisonnable de penser que c'est à l'utilisateur de décider de la maintenance adéquate. Aussi proposons nous un système dans lequel chacun des niveaux est utilisable et commandable à l'aide d'un drapeau.

D'autre part, l'absence de contradiction dans les représentations de connaissance centrées-objet évite d'utiliser le coûteux retour-arrière de Doyle. Le système n'aura donc que deux fonctions: **enregistrement des dépendances** et **propagation des invalidités**.

4.2. Architecture du système.



1- Interrogation de la base par la demande d'une valeur.
2- Lecture de la base (méthodes, valeurs...)
3- Communication des inférences produites.
4- Mémorisation de l'inférence.

5- Mémorisation des valeurs inférées.
6- Réponse à l'utilisateur.

1- Mise à jour ou saisie de la base.

2- Modification de la base.
3- Communication des inférences à invalider.
4- Suppression de l'inférence mémorisée sous forme de justification.
5- Suppression des valeurs inférées.

L'architecture proposée ici permet d'intégrer le système de maintenance de la vérité de façon modulaire à une représentation de connaissance centrée-objet. Toutes les opérations d'enregistrement et d'invalidation sont présentes pour les deux types d'actions sur la base.

4.3. Description fonctionnelle.

Nous décrivons ici toutes les fonctionnalités du système s'il veut réaliser tout ce qui a été présenté plus haut.

4.3.1 «Caching» sans maintenance.

Le «caching» sans maintenance est très simple à implémenter. Il s'agit d'une modification de la fonction **val?** chargée de déterminer la valeur d'un attribut. Le gain de temps apporté est évident. A la première instruction, si une valeur (IN) a été stockée dans l'instance, elle est immédiatement rendue, comme si l'utilisateur l'avait lui-même fournie. C'est la base minimale du «caching» sans maintenance.

La forme de **val?** qui a été implémentée est plus complexe puisqu'elle intègre les fonctionnalités utilisables avec la maintenance. Si aucune valeur n'est stockée, le programme regarde si une méthode déjà utilisée est toujours valide. Dans ce cas il ne perd pas de temps en recherche de la méthode, il ne fait que recalculer les arguments. Enfin, si aucune méthode n'est conue comme valide, le programme va chercher la méthode à utiliser, et tente une dernière fois de réutiliser les anciens résultats.

Cette version du programme est très efficace puisqu'elle réutilise au maximum les données stockées.

4.3.2 Maintenance sur les arguments.

La maintenance sur les arguments requiert la propagation de l'invalidité quand une valeur est modifiée. Il s'agit de modifications ponctuelles dans le code, l'algorithme est très simple mais hautement récursif.

Pour toute valeur d'attribut modifiée - ajout, suppression ou modification - on invalide tous ses conséquents et les conséquents de ceux-ci. Cette opération est activée sur tous les attributs d'une instance que l'on détruit. La nature de l'attribut - mono ou multivalué - n'est pas prise en compte. En effet, pour les attribut multivalués, soit la valeur est obtenue en une seule fois, soit elle est obtenue élément par élément. Dans le premier cas elle est considérée comme une valeur unique et ne dépend que d'une justification, tous les algorithmes présentés portent alors simplement sur des listes de valeurs. Dans le second cas, on aura établi une justification pour chaque valeur trouvée et les programmes traiteront toutes les valeurs en série, ou une valeur en particulier, indépendamment des autres.

4.3.3 Maintenance sur les méthodes.

Pour la maintenance sur les méthodes, il n'y a qu'une action à effectuer: quand une méthode est modifiée, il faut invalider les justifications dans lesquelles cette méthode intervient et donc invalider toutes les valeurs «cachées» calculées avec cette méthode.

4.3.4. Maintenance sur le graphe d'héritage.

Comme pour les méthodes, le graphe d'héritage est géré par une unique fonction, mais celle-ci est plus complexe. Cette fonction remonte dans le graphe d'héritage et invalide tous les calculs permettant de valuer l'attribut considéré d'une instance de la classe considérée avec les méthodes rencontrées.

Ces différentes fonctions doivent remplir pleinement l'office de notre TMS. Même si certains aspects n'ont pas été traités ici (structure des justifications, définition d'une méthode masquée) parce qu'ils auraient rendu l'exposé trop ardu et trop spécifique, on peut d'ores et déjà se rendre compte des avantages apportés par un TMS en contrepartie d'un code minimal.

Conclusion.

Le mécanisme de «caching» dans une représentation de connaissance centrée-objet doit permettre d'en augmenter considérablement les performances en temps en évitant de refaire des inférences inutilement. Cet accroissement des performances est bien sûr obtenu au prix d'une plus grande place mémoire occupée par la base.

Le système de maintenance de la vérité présenté est beaucoup plus simple et concis que les TMS classiques. En effet, l'absence de valeurs par défaut, et donc de liste OUT, diminue grandement le travail du TMS, mais c'est l'absence de contradiction qui permet d'utiliser des algorithmes très simples (évitant le retour-arrière).

Mais l'implémentation d'un TMS dans une base de connaissances offre certains avantages additionnels. Tout d'abord, le fait d'avoir une structure de donnée mémorisant les inférences permet de les justifier et de les expliquer a posteriori sans aucun coût supplémentaire. Une part de l'explication négative (expliquer pourquoi telle valeur n'a pas été obtenue ou pourquoi telle ou telle méthode n'a pas été activée) est facilement envisageable. D'autre part, le système, tout comme ceux de Doyle et De Kleer, est tout à fait utilisable pour faire du raisonnement hypothétique. Il détruira les inférences occasionnées par une hypothèse abandonnée et cela sans retour-arrière.

Le système proposé a été implémenté dans le système de gestion de base de connaissance Shirka écrit en Le_Lisp de l'INRIA [Chailloux 86]. La description détaillée des choix d'implémentation se trouve dans [Euzenat 87].

Références.

[AI 80]
John McCARTHY, Raymond REITER, Drew McDERMOTT, Jon DOYLE
Special issue on non-monotonic logic
Artificial Intelligence 13-I, 1980

[Chailloux 86]
Jérôme CHAILLOUX
Le_Lisp 15.2 : manuel de référence
Rapport technique, INRIA, FR, 1986

[Chehire 86]
Wadih CHEHIRE
Sypruc : système pour la représentation et l'utilisation des connaissances
Actes Avignon-86-II (pp933-946), 1986

[De Kleer 79]
Johan DE KLEER, Jon DOYLE, Guy STEELE, Gerald SUSSMAN
Explicit control of reasoning
dans: WINSTON, BROWN
Artificial intelligence: an MIT perspective
The MIT press, Cambridge, MA, (pp95-116), 1979

[De Kleer 83]
Johan DE KLEER
Choices without backtracking
Actes 4th nationnal conference on artificial intelligence, Austin, TX (pp79-85), 1983

[De Kleer 86a]
Johan DE KLEER
An assumption-based TMS
Artificial Intelligence 28-II (pp127-162), 1986

[Doyle 79b]
Jon DOYLE
A truth maintenance system
Artificial Intelligence 12-III (pp231-272), 1979

[Euzenat 87]
Jérôme EUZENAT
Un système de maintenance de la vérité pour une représentation de connaissance centrée-objet
Rapport de DEA, INP Grenoble, FR, 1987

[Hein 83]
Uwe HEIN
PAUL: A programming langage for knowledge engineering applications
Rapport de Recherche IDA-R-80-04, Linköping university, SW, 1982

[Martins 83]
João MARTINS, Stuart SHAPIRO
Reasoning in multiple belief spaces
Actes IJCAI-83-I (pp370-373), 1983

[Rechenmann 85]
François RECHENMANN
Shirka : mécanismes d'inférence sur une base de connaissance centrée-objet
Actes RFIA-85-II (pp1243-1254), 1985

[Stallman 77]
Richard STALLMAN, Gerald SUSSMAN
Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis
Artificial Intelligence 9-II (pp135-196), 1977

[Williams 84]
Chuck WILLIAMS
ART: The advanced reasoning tool, conceptual overview
Inference Corp., Los Angeles, CA, 1984

[Wrigth 83]
J. WRIGHT, Mark FOX
SRL/1.5 user manual
Carnegie-Mellon University, Pittsburg, PA, 1983