Un système de maintenance de la vérité pour une représentation de connaissance centrée-objet

Jérôme Euzenat

Institut National Polytechnique de Grenoble.
DEA D'INFORMATIQUE - Profil Intelligence Artificielle.

Un système de maintenance de la vérité pour une représentation de connaissance centrée-objet.

Jérôme EUZENAT

Jury: J. CROWLEY
T. GRANIER
P. JORRAND
F. RECHENMANN
J. SIFAKIS
J.P. VERJUS

Juin 1987

Un système de maintenance de la vérité pour une représentation de connaissance centrée-objet.

Résumé.

L'utilisation d'objets pour la représentation des connaissances est de plus en plus répandue. C'est dire l'importance que prend la conception de bases de connaissance centrées-objet qui peuvent être manipulés de manière non monotone par divers systèmes informatiques tant pour y opérer des modifications que des consultations.

On se propose d'étudier des mécanismes permettant à la fois plus d'efficacité et de cohérence dans l'utilisation d'une représentation centrée-objet. Le mécanisme de caching introduit des problèmes liés à l'utilisation non monotone de la base. Dans le but de palier ces problèmes, les différents systèmes de maintenance de la vérité existant sont étudiés.

Un cadre général permettant la coopération des mécanismes de "caching" et de maintenance de la vérité au sein d'une représentation centrée-objet est proposé. On présente ensuite une réalisation effective des propositions sur le système de gestion de bases de connaissance centrées-objet Shirka.

Mots-clés.

représentation centrée-objet, maintenance de la vérité, TMS, raisonnement non monotone, caching.

Un système qui doit tenter de reproduire la connaissance d'un expert doit manipuler de grandes quantités de connaissance. Les systèmes de gestion de bases de connaissance sont donc destinés à se développer afin de faciliter l'écriture de programmes informatiques de plus en plus "intelligents" dans des domaines précis. C'est le but, entre autres, des représentations centrées-objet qui proposent une structure d'accueil unique: l'objet.

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 bien entendu d'agir sur la base de connaissance, aussi bien au niveau des connaissances factuelles que celles destinées au raisonnement.

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 entraine des gaspillages de temps considérables, c'est celle réalisée le plus souvent par les représentations centrées-objet existantes.

Il est envisageable de conserver les résultats obtenus afin de gagner du temps dans l'utilisation des bases de connaissance. Cela introduit cependant des problèmes liés à l'utilisation non monotone de telles bases. La gestion des phénomènes non monotones a été étudié dans le cadre des systèmes de maintenance de la vérité (truth maintenance systems ou TMS). Ces systèmes sont présentés indépendamment de tout formalisme de représentation de connaissance particulier.

Après avoir étudié séparemment, les systèmes de maintenance de la vérité proposés (essentiellement conçus pour des systèmes à base de règles) et les représentations de connaissance centrées-objet, la conception d'un TMS pour une représentation centrée-objet est envisagée. Il semble en effet intéressant de spécifier un TMS spécifiquement pour les représentations centrées-objet, afin de tenir compte des multiples particularités de tels systèmes.

Le cadre ainsi mis en évidence sera appliqué au système de gestion de bases de connaissance centrées-objet Shirka, ce qui fera l'objet de la dernière partie de ce mémoire.

première partie

Les systèmes de maintenance de la vérité

Au sein des systèmes de raisonnement, de grandes quantités de connaissance sont manipulées. 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. Dés lors que ces entités ont été précédement utilisées dans des inférences, les conclusions auxquelles a conduit leur utilisation devront être invalidées. Récursivement, les conclusions auxquelles ont contribué ces conclusions devront l'être à leur tour...

Le problème posé est de savoir à quelles entités, l'utilisation par le module de raisonnement d'une entité précise, a-t-elle mené. Pour ce faire, à la fin des années 70, on s'est avisé d'enregistrer les **dépendances** entre les objets représentant ces entités /DeKleer& 79, 82, Shrobe 79, Charniak& 85/.

L'autre problème est que l'ajout d'une connaissance dans une base peut rendre cette 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 interpréteurs Prolog, mais incluant le retrait des objets insérés 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, Shrobe 79/, où l'on ne remet pas en cause le dernier objet inféré, mais un objet qui a réellement contribué à une contradiction. /DeKleer 86a / propose une comparaison des différents systèmes de retour-arrière.

I. TMS à propagation.

Jon Doyle a le premier décrit un système de maintenance de la vérité complet. Il ne fit que généraliser à tout système de raisonnement ce qui était déjà présent dans le système ARS /Stallman& 77/. Il a isolé maintenance de la vérité et système de raisonnement et synthétisé toutes ces techniques dans un système appelé "truth maintenance system" mais qui devrait s'appeler "belief revision system" ou système de révision des croyances. On qualifiera ce système 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. L'essentiel du système est exposé dans /Doyle 79b/. On pourra, par ailleurs, trouver un résumé très clair des intentions de l'auteur dans /Doyle 79a/ et une introduction à la maintenance de la vérité dans /Thompson 79/.

Les systèmes de raisonnement agissent tous selon le principe suivant, appelé inférence: découvrir de nouvelles vérités à partir de vérités précédemment connues. Selon /Doyle 79a/, ce principe met à jour trois problèmes:

- le **problème du sens commun**: le raisonnement humain ne procède pas ainsi, il fait souvent des suppositions, qu'il remet en cause si des contradictions apparaissent. Ce n'est pas la manière de procéder des systèmes de raisonnement implémentés, en apparence du moins.

- le "**frame problem**": l'ensemble de nos croyances change de façon non-monotone (c'est-à-dire au cours du raisonnement, indépendement de celui-ci tel que défini plus haut). Savoir ce qui change et ce qui subsiste après une modification de la base est l'essentiel du problème. Face à de telles situations, les systèmes classiques sont obligés de refaire certaines inférences qui avaient déjà été faites.

- le **probleme du contrôle**: le problème est de ne pas choisir en aveugle l'inférence à entreprendre.

1. Proposition théorique.

Doyle redéfini 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 système de maintenance de la vérité 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.

Doyle propose donc les bases théoriques suivantes:
définition.1: Une **raison**, pour une attitude, est une paire d'ensembles d'attitudes. Le premier est appelé ensemble IN, le second ensemble OUT.
définition.2: Pour une attitude A, 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**.
définition.3: 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. Adaptation à l'implémentation du système.

Le système de maintenance de la vérité manipule et maintient deux sortes d'entités, les **nœuds**, qui représentent les attitudes et les **justifications** des nœuds. Il a trois actions fondamentales sur ces entités:

- **créer** de nouveaux nœuds.
- **ajouter** ou **supprimer** une justification pour un nœud.
- **marquer** un nœud comme contradictoire.

3. Les justifications.

A un nœud est associé un ensemble de justifications. Il y a deux sortes de justifications:

a) La **SL-justification** (pour support-list) représente la raison telle que présentée dans la partie théorique. Sa structure 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 dernière possibilité est un problème pour le système de maintenance de la vérité puisqu'elle autorise les paradoxes (nœud N justifié par (SL () (N)) c'est-à-dire $\neg N \Rightarrow N$) qu'il ne maitrise évidement pas.

b) La **CP-justification** (pour conditionnal-proof) est utilisée dans le système de maintenance de la vérité pour la remise en cause d'hypothèses en cas de contradiction. Elle se présente ainsi (CP <conséquence> <INhypliste> <OUThypliste>). Une CP-justification est dite valide ssi la conséquence est valide quand tous les nœuds de la INliste sont IN et tous les nœuds de la OUTliste sont OUT. La validité d'une CP-justification est très compliquée à vérifier, c'est pourquoi elle est utilisée le moins possible et toujours avec une OUTliste vide.

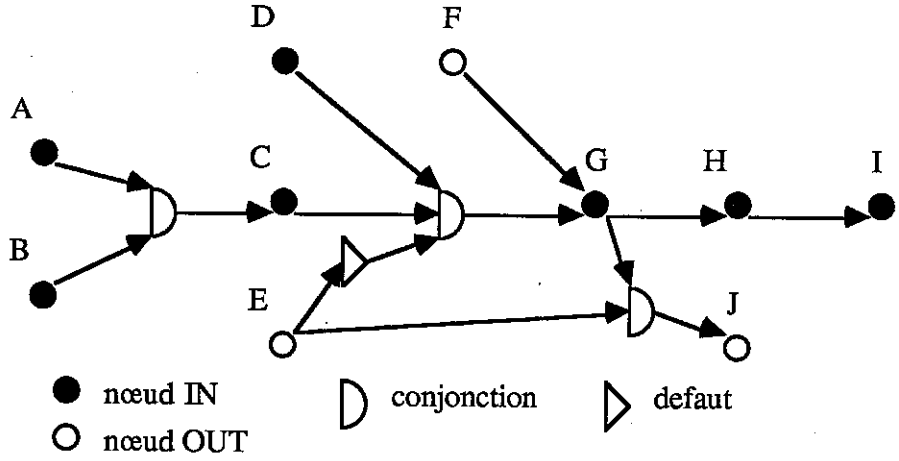
4. Les relations de dépendance.

A la structure de donnée représentant le nœud, Doyle va ajouter des attributs que l'on peut calculer en fonction des dépendances, et qui sont utiles à l'algorithme.

Soit à un moment donné, un ensemble de nœuds, avec leurs ensembles de justifications sous forme de SL-justifications, il est possible de mettre en évidence les attributs suivants de ces nœuds.

a) l'**état** (IN/OUT).
b) la **justification supportante** (Sup) est une justification valide d'un nœud IN.
c) les **nœuds supportants** (N-sup) sont pour un nœud IN INliste U OUTliste de la justification supportante. Pour un nœud OUT, un nœud OUT de la INliste ou un nœud IN de la OUTliste pour chaque justification du nœud.
d) les **antécédents** (Ant), sont l'ensemble des nœuds supportants d'un nœud IN.
e) la **fondation** (Fond) est la fermeture transitive des antécédents d'un nœud IN.
f) l' **ancètre** (Anc) est la fermeture transitive des nœuds supportants d'un nœud.
g) la **conséquence** (Csq) d'un nœud précis, est l'ensemble des nœuds qui l'utilisent dans une de leurs justifications.
h) la **conséquence effective** (Csq-ef) d'un nœud précis, est l'ensemble des nœuds qui l'utilisent comme nœud supportant.
i) la **IN-conséquence** (IN-Csq, pour believed consequences) est l'union des conséquences effectives des nœuds IN.
j) la **répercussion** (Rep), est la fermeture transitive des conséquences effectives d'un nœud.
k) la **IN-répercussion** (IN-Rep, pour believed repercussions) est la fermeture transitive des IN-conséquences d'un nœud.

Pour le réseau suivant, les attributs seraient les suivants:



Sup(G) = (SL (D,C) (E))
N-Sup(G) = {D,C,E}
Ant(G) = {D,C,E}
Fond(G) = {A,B,C,D,E}
Anc(G) = {A,B,C,D,E}
Csq(G) = {H,J}
Csq-Ef(G) = {H}
IN-Csq(G) = {H}
Rep(G) = {H,I}
IN-Rep(G) = {H,I}

N-Sup(J) = {E}

Anc(J) = {E}

Tout ce vocabulaire un peu barbare trouve sa justification dans le mécanisme de maintenance. Par ailleurs Doyle devance les critiques en signalant que si ces données occupent beaucoup de place en mémoire, il vaut mieux les enregistrer que les recalculer à chaque intervention du système de maintenance de la vérité

5. Le mécanisme de la maintenance.

Le système de raisonnement signale au système de maintenance de la vérité les inférences faites sous forme d'attitudes et de justifications pour ces attitudes. A chaque nouvelle attitude, le système de maintenance de la vérité fait correspondre un nœud. Tout le travail du système de maintenance de la vérité intervient quand c'est une justification qui lui est fournie.

Le système de maintenance de la vérité se charge alors d'ajouter la justification à celles du nœud en question (*1). Si cet ajout permet de valider le nœud (*2), alors cette validation est entérinée et propagée à toute la base (*345). Tous les nœuds qui peuvent être justifiés à l'aide du nœud en question et tous ceux qui peuvent être invalidés à cause de lui sont examinés.

Cette nouvelle justification peut amener à justifier un nœud marqué contradiction, le système de maintenance de la vérité se chargera alors, grâce à un retour-arrière dirigé par les dépendances, d'invalider un des nœuds justifiant la contradiction, et ainsi d'éliminer le nœud contradictoire de la base (*6DDBS).

Il est à noter que le système gère certaines circularités qui peuvent apparaître dans les dépendances entre nœuds (*5) à l'exception des paradoxes qui

font boucler le programme.

6. L'algorithme.

On cherche à ajouter la justification j au nœud N

* 1: ajouter une nouvelle justification
- AJOUTER(N,j)
- POURTOUT N' ∈ (OUTliste[j] U INliste[j])
  FAIRE Csq[N'] := Csq[N'] U {N}
- SI j est une CP-justification ALORS CPCSQ := CPCSQ U {j}
- SI Et[N] = IN ALORS ALLEREN *7
- SI j est invalide ALORS mise-à-jour de N-Sup[N] ; ALLEREN *7

* 2: mise à jour des croyances
- SI Csq-Ef[N] = ∅
  ALORS Et[N] := IN
  mise-à-jour de N-Sup[N], Ant[N], Fond[N], Anc[N] en considérant j comme justification supportante.
  ALLEREN *7
  SINON L := { (N',e) | N' ∈ Rep[N] & e = Et[N'] }

* 3: marquage des nœuds
- POURTOUT (N',e) ∈ L FAIRE Et[N']:= ()

* 4: évaluation des justifications des nœuds
- POURTOUT (N',e) ∈ L
  FAIRE SI Et[N'] = ()
  ALORS *4bcl:
  En commençant par les SL puis les CP
  SI ∃ j' valide pour N'
  ALORS
  Sup := CP2SL(j')
  mise-à-jour de N-Sup comme en *2
  Et[N'] := IN
  SINON
  Et[N'] := OUT
  POURTOUT N'' ∈ {N | N ∈ Csq[N'] & Et[N] = ()}
  FAIRE ALLEREN *4bcl AVEC N' := N''

```
* 5: relâchement des circularités
   - POURTOUT (N',e) ∈ L
     FAIRE SI Et[N'] = ()
           ALORS
            SI ∃ j' valide pour N' (en considérant () comme OUT)
            ALORS
               SI Csq-Ef[N'] = ∅
               ALORS
                 J-Sup[N'] := {j'}
                   mise-à-jour de N-Sup[N'] comme en *2
                   Et[N'] := IN
               SINON
                   POURTOUT N'' ∈ Csq-Ef[N']
                   FAIRE Et[N''] := ()
           SINON
               Et[N'] := OUT
               mise-à-jour de N-Sup[N'] comme en *1

* 6: vérifier les CP-justifications et les contradictions
   - POURTOUT (N',e) ∈ L
     FAIRE
     * 6.1:  SI Et[N'] = OUT OU CPCSQ = ∅
             ALORS ALLEREN *6.2
              SINON
                 POURTOUT j' valides ∈ CPCSQ
                 FAIRE
                    justifier N' avec CP2SL(j'')
                    si on a besoin du TMS alors commencer en *6
     * 6.2:  SI Et[N'] = IN & Contradiction[N']
             ALORS ALLEREN *DDBS

* 7: signaler les modifications
   - POURTOUT (N',e) ∈ L
     FAIRE SI Et[N'] ≠ e ALORS signaler au système de raisonnement.
```

CP2SL est une procédure qui transforme une CP-justification en SL-justification. Le principe de cette procédure est de trouver les nœuds supportant le <conséquent> tels que ceux-ci ne dépendent pas des hypothèses (c'est-à-dire <INhypliste> et <OUThypliste>). Ceci est résumé dans le nom que Doyle donne à sa procédure: **"Find Independant Support"**.

Il s'agit en fait de trouver

$\{ N \mid N \in Fond[<conséquent>]$

$\& \forall N' \in (<INhypliste> \cup <OUThypliste>)$

$N \notin Rep[N']$

$\& \forall N'' \in Rep[N']$

$N \notin Ant[N'']$

$\& N \notin Ant[<conséquent>] \}$

Les nœuds IN de cet ensemble formeront la INliste et les nœuds OUT la OUTliste de la SL-justification. On se rend bien compte que cette transformation est uniquement conjoncturelle puisqu'elle dépend de l'état des nœuds au moment de la construction.

7. Retour-arrière dirigé par les dépendances.

Le retour-arrière dirigé par les dépendances (DDBS pour "dependancy-directed backtracking system") est activé lorsqu'un nœud marqué contradiction est validé (IN). L'idée est de remettre en cause certaines suppositions afin d'éliminer les nœuds contradiction de l'ensemble des nœuds IN. Les nœuds remis en cause sont ceux ayant une OUTliste non vide, ainsi ce sont les assertions valides par défaut que l'on supprime de la base.

Ce retour-arrière se distingue du retour-arrière chronologique du fait qu'il ne va pas remettre en cause le dernier fait inféré par le système de raisonnement, mais un fait dont le nœud est antécédent de la contradiction.

L'algorithme comporte quatres étapes:

*1: trouver l'ensemble minimal des suppositions supportantes

- $S := \{ A \mid A \in Fond[N]$
  $\& Outlist[J\text{-}Sup[A]] \neq \emptyset$
  $\& \forall B \in S ; A \notin Fond[B]\}$
- SI $S = \emptyset$ ALORS signaler "CONTRADICTION INSOLUBLE"

*2: résumer les causes de l'inconsistance

- CREER(Pb)
- AJOUTER(Pb,(CP N S ()))

*3: sélectionner et rejeter un nœud fautif

- Choisir $A \in S$
- Choisir $D \in Outlist[J\text{-}Sup[A]]$
- AJOUTER(D,(SL ({X} U S \ {A}) (Outlist[J-Sup[A]] \ {D})))

*4: recommencer si nécessaire

- SI Et[C] = IN ALORS aller en *1

Ce retour-arrière, s'il est plus efficace que le retour-arrière chronologique, est cependant qualifié d'aveugle (p257). En effet les remises en cause se font dans un ordre théoriquement arbitraire (Choisir) puisqu'on manipule des ensembles. Dans la réalité, l'implémentation Lisp manipule des listes (inévitablement ordonnées), et Doyle fournit donc un certain nombre de méthodes (pp259-265) pour, connaissant l'implémentation, diriger plus radicalement le retour-arrière.

James Goodwin /Goodwin 82/ a fourni un algorithme de retour-arrière plus efficace que celui de Doyle ainsi que la preuve de son arrêt (s'il n'y a pas de paradoxes).

8. Extensions envisagées.

Le système de maintenance de la vérité a été utilisé dans différents programmes simulateurs de circuits électroniques, entre autres ceux de Stallman et Sussman /Stallman& 77/ et de De Kleer /DeKleer& 79/.

Un résolveur de problèmes (AMORD) a été conçu autour du système de maintenance de la vérité et utilisé dans différents programmes /DeKleer& 79, Shrobe 79, Doyle 79c/.

Doyle signale diverses utilisations possibles de son système, dont on peut parler brièvement:

a) Raisonnement par défaut (pp261-262)

On l'a vu en I.3.a, une SL-justification composée d'une INliste vide et d'une OUTliste non vide, justifie un fait par l'absence d'autres faits. Le TMS modélise donc complètement le raisonnement par défaut tel que défini dans /Reiter 80/. Doyle étend sa représentation en expliquant comment exprimer des "cascades" de défauts.

b) Modélisation des croyances de différents agents (pp258-259)

Doyle propose d'utiliser le TMS chargé de maintenir les croyances d'un système informatique, à la maintenance des croyances de différents agents. L'idée proposée est de représenter la croyance de U en N par deux nœuds UB(N) et ¬UB(N) et de justifier ¬UB(N) par défaut. Pour chaque SL-justification j = (SL IN OUT) de N dans le système de croyance de U, on crée un nœud UB(j) et on justifie UB(N) par (SL INliste ()) où:

$$\text{INliste} = \{ UB(M) \mid M \in IN \} \cup \{ \neg UB(M) \mid M \in OUT \} \cup \{ UB(j) \}$$

Cette représentation a le défaut de supposer que les agents ont le même système de raisonnement et la même manière de maintenir leurs connaissances, on rejoint ici les logiques modales et les modèles à la Kripke /Turner 84/.

9. Tentatives de formalisation.

Dans /Doyle 79b/ on trouve une justification des actions du système de maintenance de la vérité décrites comme un comportement dialectique (selon

Hegel: mouvement de la pensée qui progresse vers une synthèse en s'efforçant continuellement de résoudre les oppositions entre chaque thèse et son antithèse). Doyle montre comment construire un système de raisonnement, qui produira thèse et antithèse, et comment le système de maintenance de la vérité fera à chaque étape la synthèse entre les deux aspects.

La logique sous-jacente au système de maintenance de la vérité sera plus complètement et plus sérieusement justifiée dans les articles sur les logiques non-monotones de Doyle et McDermott /McDermott& 80, McDermott 82/. Cette façon de voir a été récement critiquée dans /Brown 85/.

D'autre part une formalisation mathématique du système de maintenance de la vérité est fournie par Doyle dans /Doyle 83/. L'algorithme présenté ci-dessus est défini en termes mathématiques - à l'exeption du DDBS - ce qui simplifie de beaucoup les notations et évite de définir de nombreuses structures de données. Cette formalisation permet de faire la preuve des résultats du système de maintenance de la vérité

10. Critiques.

Toutes les critiques que l'on peut faire au système de Doyle ont été énumérées dans un article de De Kleer /DeKleer 83/. Elles ont toutes leur intérêt et peuvent être reprises ici.

a) Le problème de l'état unique.

Le système de maintenance de la vérité quand il résoud une contradiction amène le système dans un état particulier et unique. Or il existe peut-être d'autres moyens de résoudre cette contradiction qui mèneraient à d'autres états plus intéressants pour le module de raisonnement.

b) La remise en cause trop zélée.

Le DDBS va remettre en cause arbitrairement l'un des fondements d'une contradiction. Or il ne sait pas quel est le fondement qui entraine "véritablement" la contradiction. C'est-à-dire que plus tard on devra remettre en cause un autre fondement parce qu'il a entrainé une nouvelle contradiction, et on ne réintroduira pas le premier fondement mis en cause.

c) La difficulté de passer d'un état à un autre.

Il est très compliqué avec le système de Doyle, d'agir sur les hypothèses pour constater les changements d'état. En effet, ôter ou réactiver une hypothèse relance tout le travail de propagation.

d) La dominance des justifications.

De Kleer reproche à Doyle de trop se soucier des justifications, au lieu de s'intéresser aux suppositions (on verra plus loin que De Kleer veut ne raisonner que sur les clôtures sans conserver les justifications).

e) La lourdeur du mécanisme de retour-arrière.

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 système de maintenance de la vérité en devient très lent.

f) La désinvalidation ("unouting")

Quand une donnée a été remise en cause (OUT), et qu'elle est de nouveau considérée comme IN, le système de maintenance de la vérité parcourt tout le réseau de dépendances pour retrouver ce qui avait déjà été inféré. Cela est trop long.

II. TMS à contextes.

Partant de ces critiques, De Kleer, qui s'était déjà occupé de maintenance de la vérité avec Doyle, s'est attaché à construire son propre système, le **système de maintenance de la vérité basé sur des suppositions** (ATMS pour Assumption-based TMS). On le trouve à l'état embryonaire dans /DeKleer 83/, et de manière plus finie dans /DeKleer 86a/. C'est sur ce second article que s'appuie l'exposé de ce système.

1. L'idée de l'ATMS.

L'idée de De Kleer est que le système de maintenance de la vérité 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 système de maintenance de la vérité de décider de l'espace mais au module de raisonnement. Afin de permettre au module de raisonnement d'avoir accès à tous les états possibles, De Kleer propose d'enregistrer les ensembles d'hypothèses sous lesquelles un nœud est IN plutôt que le fait qu'un nœud soit IN. On construira alors un système permettant de raisonner simultanément avec divers ensembles de suppositions, système inspiré en partie de /Martins& 83/.

Ce système est qualifié de TMS à contextes, car au lieu d'enregistrer la validité d'une entité, il enregistre les contextes dans lequel cette entité est valide.

2. Données manipulées.

De Kleer définit les différents ensembles qui lui permettent d'exprimer la validité d'un nœud, ainsi:

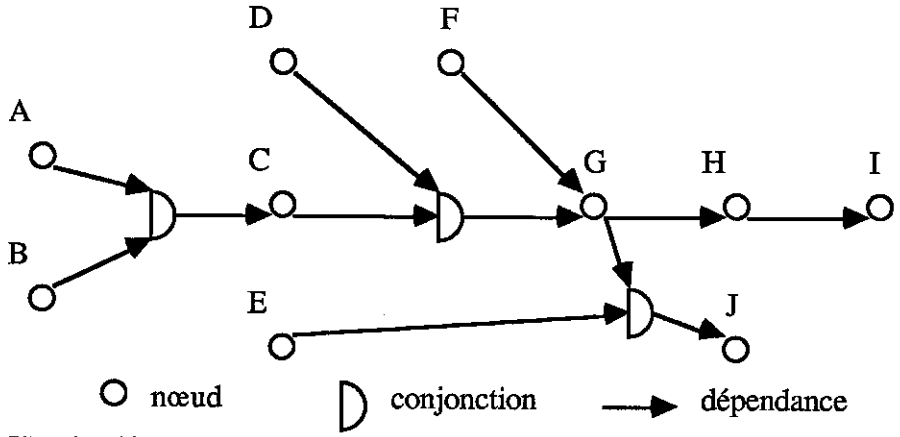
a) La **supposition** (assumption) est la fondation initiale qui peut supporter chaque donnée. La supposition peut cependant être dérivée.
b) Un **nœud** de l'ATMS représente une donnée du module de raisonnement.
c) Une **justification** signale comment dériver un nœud d'autres nœuds.
d) Un **environnement** est un ensemble de suppositions.
e) Un **contexte** est un ensemble formé de tous les nœuds d'un environnement consistant et de tous les nœuds dérivables de cet environnement.
f) Un **environnement caractéristique** d'un contexte est un ensemble de suppositions duquel tous les nœuds d'un contexte peuvent être dérivés.

g) Un nœud N est **valide** dans un environnement E s'il peut être dérivé de E et de l'ensemble courant de justifications J. On note:

$E,J \mapsto N$

h) Un **label** de N est un ensemble d'environnements consistants ayant la propriété que $E,J \mapsto N$. On notera L(N) pour un label de N.

Sur le même graphe que précédement, les données seraient les suivantes:



Une justification de G est {C,D}, une autre est {F}.
{A,B,D} est un environnement, un autre est {A,B,E,F}.
Un contexte de {A,B,D} est {A,B,C,D,G,H,I}, un contexte de {A,B,E,F} est {A,B,C,E,F,G,H,I,J}. L'environnement caractéristique du premier est {A,B,D}, celui du second est {A,B,E,F}.

Le système manipule pour chaque nœud la structure suivante:
nœud: <Donnée, label, Justifications>
La donnée et les justifications sont internes au module de raisonnement et ne sont pas interprétées par l'ATMS qui ne se soucie donc que des labels.
Quatre sortes de nœuds seront distinguées suivant leur label:

- **prémisse**: <p,{{}},{()}>
- **supposition**: <A,{{A}},{(A)}>
- **nœud assumé**: <a,{{A}},{(A)}>
- **nœud dérivé**: <n,{{A,B},{C},{E}},{(b),(c,d)}>

On aura donc dans l'exemple précédent:

donnée	label	justification
A	-	-
B	-	-
C	{A,B}	(A,B)
D	-	-
E	-	-
F	-	-
G	{F} {D,A,B}	(F) (D,C)
H	{F} {D,A,B}	(G)
I	{F} {D,A,B}	(H)
J	{F,E} {D,A,B,E}	(G,E)

3. Principe du système.

A un moment donné, le module de raisonnement a fourni au système de maintenance de la vérité un certain nombre de nœuds et un certain nombre de justifications. L'évolution du raisonnement est représentée 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 celui-ci a produit.

La tâche du système de maintenance de la vérité est de déterminer avec efficacité les contextes. Pour ce faire il manipule les labels en considérant les critères de complétude ($\forall E'$; $J,E' \mapsto N$ $\exists E \in L(N); E \subset E'$), minimalité ($\forall E \in L(N)$, $\forall E' \in L(N)$, $E \not\subseteq E'$), consistance ($\forall E \in L(N)$, $\neg(E \mapsto \bot)$) et correction ($\forall E \in L(N)$, $J,E \mapsto N$). La représentation par label permet de savoir très facilement à un moment donné et pour tous contextes si le nœud est **dans** le contexte ou non.

Les états d'un nœud dans un contexte sont définis par:

a) **nécessairement présent** si un environnement de son label est dans le contexte (inclusion ensembliste).

b) **nécessairement absent** si son addition au contexte causerait la dérivation de $\bot$.

c) **couramment absent** si le nœud peut devenir présent ou absent suivant une nouvelle justification.

Le module de raisonnement, suivant le contexte dans lequel il veut travailler (un ensemble de suppositions qu'il fait), obtient l'état de ses données. Ce principe permet de pallier la plupart des défauts d'un système de maintenance de la vérité à propagation: On explore tous les états possibles à la fois, il est facile de passer d'un contexte à l'autre (il suffit de le dire), les justifications ne dominent plus du tout et les problèmes de retour-arrière et de désinvalidation n'apparaissent plus.

4. Mécanisme.

Le protocole de communication entre le système de maintenance de la vérité et le module de raisonnement est complètement défini dans /DeKleer 86c/. Le système raisonneur fournit au TMS les justifications qu'il a inférées, en échange l'ATMS lui fournit l'état des croyances sans interpréter les symboles du système de raisonnement. Les conjonctions inconsistantes sont fournies à l'ATMS comme justification du nœud:

$N_\bot$: <⊥,{},{...}>

Il y a trois opérations de base: **création d'un nœud, création d'une supposition, ajout d'une justification**. Seule la troisième demande du travail au système de maintenance de la vérité.

Pour chaque nouvelle justification, le système s'assure que l'intersection des contextes des antécédents est égale aux contextes du conséquent.
Soit N justifié par j = {N1,...,Nn} Ni ayant pour label Li = {Ei1,...,Eim}
Pour ce, on fait tout d'abord l'union

$$LU = \{ E \mid E = E1i\ U \ldots U\ Enj \}$$

LU est complet et correct, il est rendu consistant en supprimant les ensembles incluant un sous-ensemble inconsistant:

$$LU' = \{ E \mid E \in LU \ \&\ \forall E' \subseteq E\ \neg NoGood(E) \}$$

LU' est rendu minimal en supprimant les ensembles inclus dans d'autres:

$$LU'' = \{ E \mid E \in LU' \ \&\ \forall E' \in LU'\ E \not\subset E' \}$$

Si le nœud est N⊥, on signale alors que chaque environnement de LU" doit être NoGood, et on supprime tout environnement inconsistant (c'est-à-dire un environnement contenant un environnement de LU) de la base.

De Kleer signale nombre d'astuces qui permettent d'augmenter la rapidité du programme.

5. Extensions.

De Kleer a signalé dans /DeKleer 86b/ les extensions possibles de son système vers les disjonctions d'antécédents, le raisonnement par défaut et la justification non-monotone.

6. Critiques.

On peut reprocher à De Kleer d'avoir, contrairement à Doyle, négligé les justifications au détriment des suppositions. En effet, les problèmes qui chez Doyle étaient de revenir sur un raisonnement déjà fait subsistent. Ici, le raisonnement est cristallisé dans les justifications et il est impossible de revenir sur celles-ci. De Kleer a prévu la remarque et donne quelques techniques pour supprimer des justifications, techniques qui de son propre aveu sont lourdes.

Une autre critique peut être faite aux systèmes de Doyle et De Kleer, c'est que leurs systèmes de maintenance de la vérité n'enregistrent que des faits, c'est-à-dire des données qui ne sont pas des règles du module de raisonnement. Or le module de raisonnement peut utiliser des règles ou des méthodes qui varient au cours du raisonnement comme on le verra plus loin. Cela rejoint le problème soulevé par /Doyle 79b/ quand il dit que sa modélisation des croyances de différents agents suppose que ceux-ci utilisent le même moyen de maintenance de la vérité.

Conclusion.

Les deux types de systèmes de maintenance de la vérité ont été présentés. Il s'agit de deux modèles opposés. Le TMS à propagation mémorise la validité d'une entité, alors que le TMS à contextes 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'un entité vérifier sa validité dans le contexte courant.

seconde partie

Représentations de connaissance centrées-objet

La notion d'objet s'est façonnée au cours des années soixante-dix à partir du langage de programmation Simula /Dahl& 66/. Simula proposait comme structure de programmation des objets ("processes") regroupés en classes ("activities") et héritant leurs caractéristiques et leur code de leur classe.

Au début des années 70, Alan Kay a systématisé cette idée, en inventant la notion de programmation orientée-objet au travers de Smalltalk-72. L'idée de base est que toutes les entités sont des objets qui communiquent entre eux au moyen d'une unique structure de contrôle: le message.

Marvin Minsky /Minsky 75/ proposa ensuite un cadre pour la représentation des connaissances en intelligence artificielle. Partant du constat que toute situation peut être reliée à une situation déja connue, il propose de représenter ces situations par des "frames", entités que l'on peut définir ainsi

nom-de-frame -> (slot valeur)*

et de lier ces "frames" entre eux au moyen des "slots".

L'article de Minsky a eu un grand retentissement dans la communauté IA qui a adopté ces idées. De multiples interprétations de ces "frames" sont alors apparues.

Nous distinguerons représentation centrée-objet et langage orienté-objet. Dans un langage orienté-objet, toutes les entités sont des objets qui communiquent entre eux par l'intermédiaire de messages.

Le but des représentations centrées-objet n'est pas d'imposer ce style de programmation, mais de proposer une structure d'accueil simple pour une base de connaissance. Les programmes manipulant cette connaissance pouvant être indépendamment procéduraux, fonctionnels ou logiques.

I. Principes généraux.

Nous entendons donner ici un aperçu des principes adoptés dans toute représentation centrée-objet. La syntaxe utilisée dans les exemples sera celle de Shirka, elle peut aisément être transformée en une autre syntaxe.

1. Modèle théorique.

Le modèle minimal des systèmes à base d'objets est décrit en trois propositions de base. Ce modèle est enrichi de trois propositions secondaires adoptées par tous les systèmes mais qui peuvent être modifiées.

B1: Tout est **objet**.

La structure d'objet est donc la structure unique et fondamentale de la représentation de connaissance centrée-objet.

a) Structure d'objet.

Un objet est une entité possédant des attributs qui permettent de stocker ses caractéristiques. Une caractéristique d'un objet est représentée par la valeur d'un **attribut**. La **valeur** d'un attribut est un objet (puisque tout est objet), un attribut peut être mono ou multivalué.

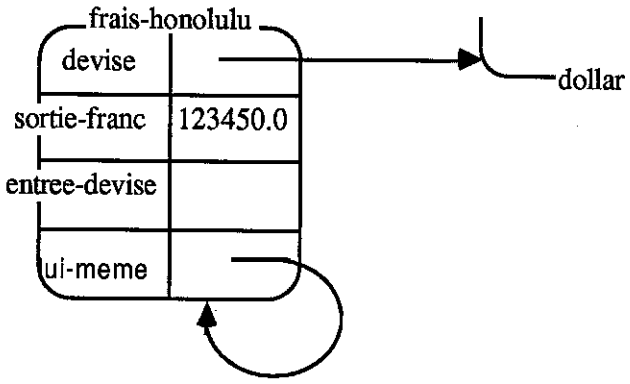
Tout objet peut être désigné par son nom, valeur de l'attribut **lui-même**.

◊ Exemple II.1

Une opération de change de devise peut être définie par divers attributs: la somme à changer, la devise à acquérir et l'équivalent en devises. Ainsi si les frais du congrès d'Honolulu s'élèvent à 123 450 francs, le schéma se présentera ainsi:

```
{ frais-honolulu
   lui-meme        =      frais-honolulu
   devise          =      dollar
   sortie-franc    =      123450.0 }
```

ce qui sera représenté par:



◊

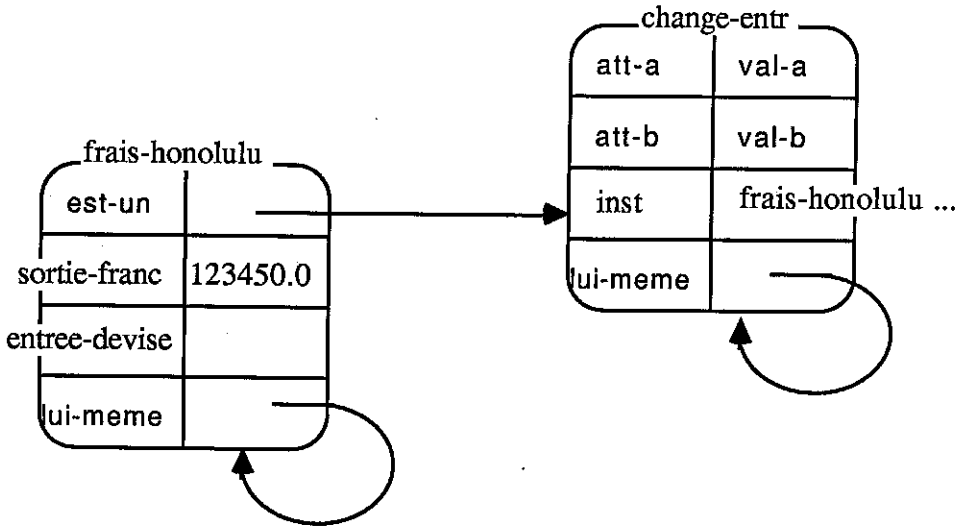
b) Classes et instances.

B2: Tout objet est **instance** d'une **classe**.

Cette classe est un objet contenant la définition de la structure des objets qui en dépendent (liste et spécification des attributs). Toutes les instances d'une classe ont donc la même structure. Une instance est reliée à sa classe par la valeur de son attribut **est-un**.

◊ Exemple II.2

L'instance frais-honolulu est donc une instance de l'objet change-entr, ce qui sera représenté ainsi:



◊

Une classe étant elle-même un objet, elle est aussi instance d'une classe que l'on appellera **métaclasse**. Cette métaclasse contient donc la définition de la structure des classes. Le problème des métaclasses est un problème qui reste ouvert au sein de la communauté des objets. Il est tranché dans Shirka en posant la définition suivante:

B3: Toute classe est instance de **schéma.**

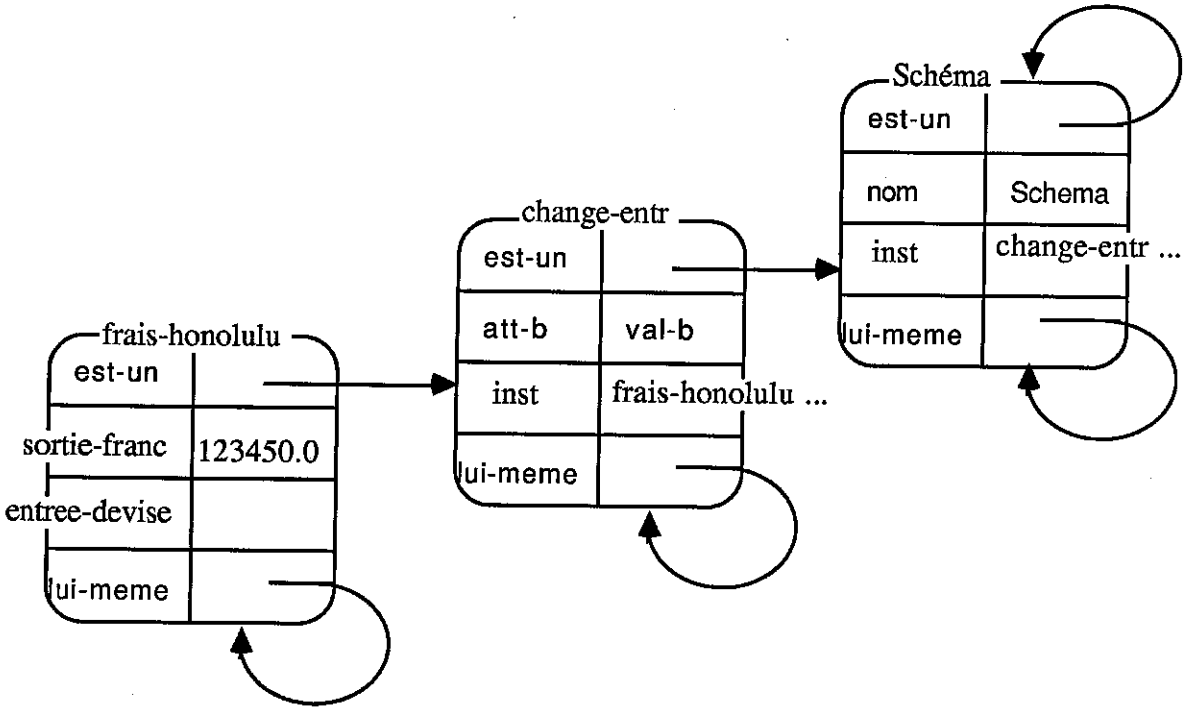
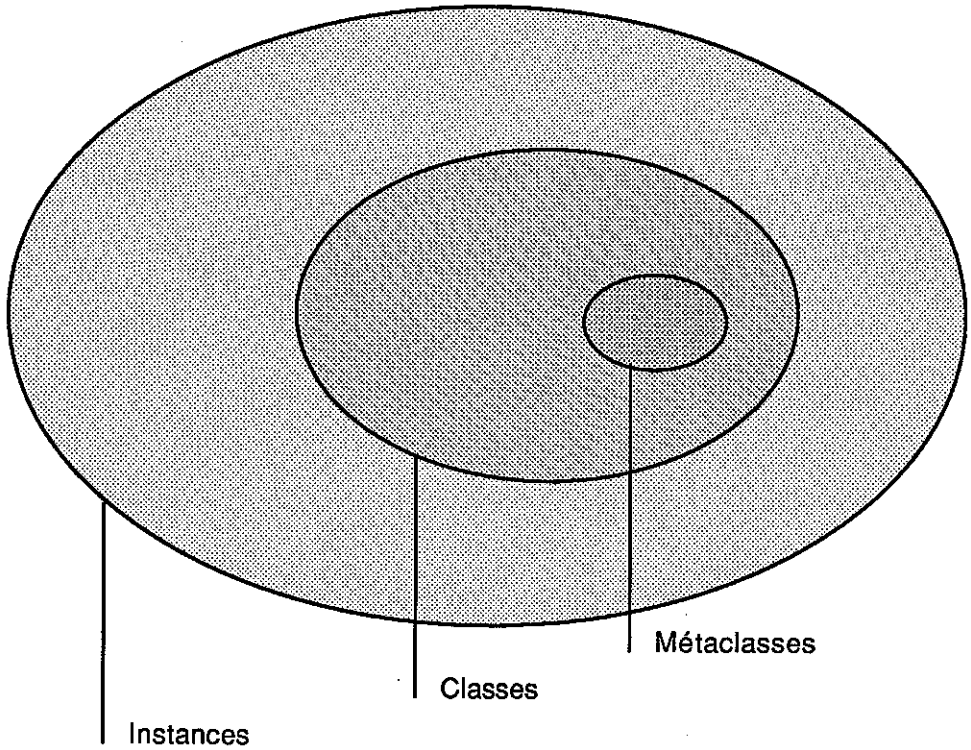
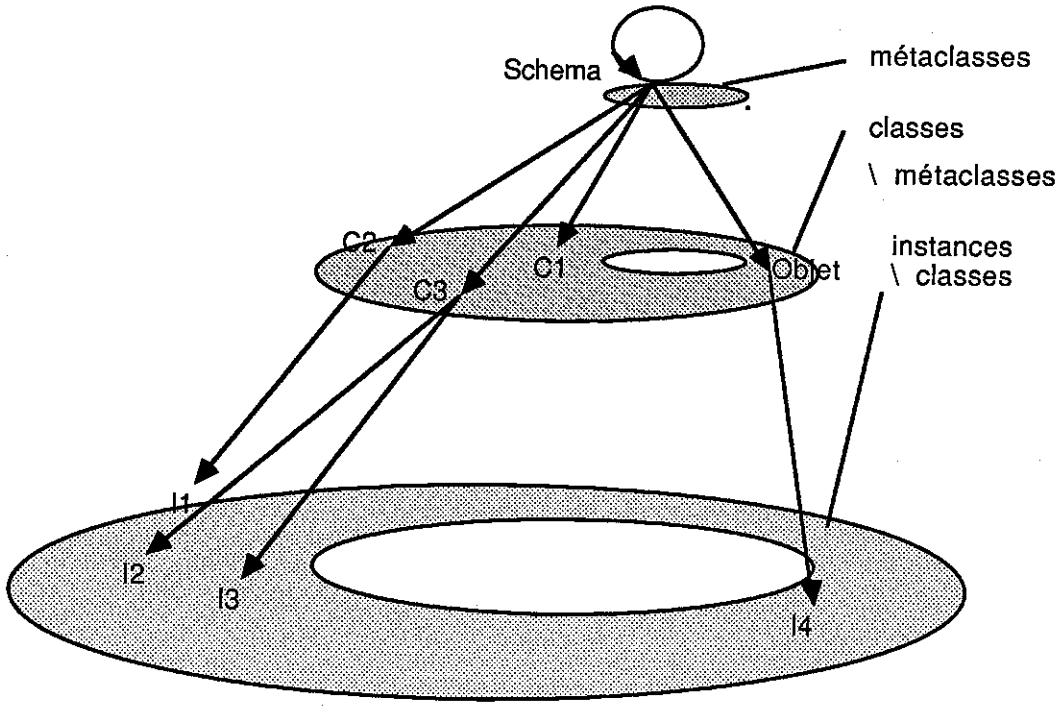


Schéma est donc métaclasse du système et est instance d'elle-même. Cette métaclasse est compatible avec tout ce qui a été dit précédemment, c'est un

objet qui contient la description de sa propre structure. On obtient alors les figures suivantes représentant instance (terminale), classe et métaclasse ainsi que l'**arbre d'instanciation:**





2. Spécification des attributs.

Nous avons dit que les classes contenaient la spécification des attributs de leurs instances. Ceci est réalisé en déclarant une classe comme le moule des instances dont les valeurs des attributs utilisent des **facettes** prédéfinies. Nous

décrivons ici les différentes facettes permettant de spécifier un attribut, suivant la fonction de cette facette.

a) Typage.

Chaque attribut est typé, son type est une classe d'objets. L'attribut peut être mono ou multivalué. Deux facettes permettent d'exprimer le type et la nature de l'attribut, il s'agit de **$un** et **$liste-de**. Puisque tout est objet, un attribut pouvant prendre n'importe quelle valeur sera typé ($un objet).

◊ Exemple II.3

La classe change-entr définira le type de ses instance de la façon suivante:

```
{ change-entr
  est-un            =       schema
  devise            $un     devise
   entree-devise    $un     entier
   sortie-franc     $un     entier }
```

Où devise est une classe d'objet définie ainsi:

```
{ devise
  est-un        =           schema
   utilisee       $liste-de symbole }
```

Entier et symbole étant des classes prédéfinies du système.

◊

b) Restrictions.

Une fois un attribut typé, il est possible de restreindre son type à certaines instances caractéristiques. Ainsi diverses facettes permettent de réduire le type à une valeur (**$valeur** <val>), une liste de valeurs précises (**$domaine** <val>*) ou pour les entiers, une liste d'intervalles (**$intervalle** (<valmin> <valmax>)*).

◊ Exemple II.4

Il est possible de restreindre la somme à changer ainsi que les devises possibles. La classe change-entr sera alors définie ainsi:

```
{ change-entr
   est-un            =           schema
   devise            $un         devise ;
                          $domaine    [lire livre peseta bolivar dollar]
    entree-devise    $un          entier
    sortie-franc     $un          entier ;
                        $intervalle  [0.0 500000.0] }
```

◊

c) Valuation.

Les facettes introduisent aussi au niveau des classes des "méthodes" permettant de connaître la valeur d'un attribut si celle-ci n'est pas présente dans l'instance. Ainsi la facette **$valeur** déja citée permet de savoir par contrainte quelle est la valeur d'un attribut. La facette **$sib-exec** permet d'activer une procédure chargée de calculer la valeur de l'attribut quand celle-ci est inconnue. Enfin, la facette **$defaut** permet de spécifier la valeur par défaut d'un attribut .

◊ Exemple II.5

La classe change, classe de frais-honolulu, peut permettre de valuer les attributs de façon intéressante:

```
{change
  est-un          =          schema
  devise          $un        devise ;
                   $defaut   dollar
   sortie-franc   $un        reel
   entree-devise $un         reel ;
                  $sib-exec
                   { calc-change
                      monnaie   $var<-   devise
                      parite    $valeur  7.12
                       francs    $var<-   sortie-franc
                       montant   $var->   entree-devise } }
```

La devise dans laquelle on changera sera, sauf indication du contraire, le dollar. Le calcul de la somme se fera automatiquement grace à l'attachement procédural calc-change dont la parité est fixée. Cet attachemant sera défini ainsi:

```
  { calc-change
    est-un      =          schema
    sorte-de    =          methode
     nom-fct      $valeur  calc-change
    monnaie     $un        objet
    parite      $un        reel
    francs      $un        reel
    montant     $un        reel }
(de calc-change (oper)
  (aj-valeur
   oper
    'montant
```

```
    (divide (valeur? 'francs oper) (valeur? 'parite oper))
  t))
```

◊

En cas de présence, au niveau d'une classe, de plusieurs facettes de valuation, un ordre est défini dans le système qui permet de décider quelle méthode activer avant quelle autre.

3. Organisation hiérarchique des classes.

Afin de mieux structurer les bases d'objets, on développe une méthode de hiérarchisation des classes.

a) L'arbre des classes.

Les systèmes d'objets sont batis autour du concept d'objet. La proposition S1 est donc introduite:

S1: La classe **Objet** préexiste à toutes les autres.

L'enrichissement du système se fait par affinement et précision du concept. Ceci est permis grâce à la proposition suivante:

S2: La création d'une classe se fait par **spécialisation** d'une classe préexistante.

La spécialisation est matérialisée par l'attribut **sorte-de** de chaque classe (donc défini au niveau de Schéma), qui accepte comme valeur une classe.
L'ensemble des classes munies de la relation sorte-de est donc organisé en un arbre dont la racine est Objet, qui est appelé **arbre d'héritage**.

b) Le mécanisme d'héritage.

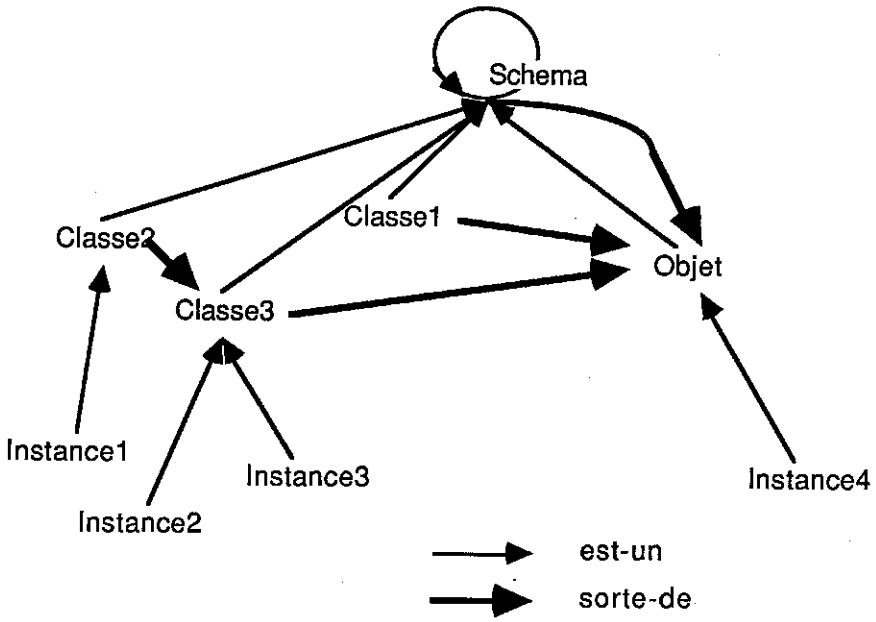
Le mécanisme d'héritage est basé sur la proposition suivante:

S3: SI S est une spécialisation de C alors toute instance de S est une instance de C

Il en résulte que la structure d'une instance I de S est définie non seulement dans S mais aussi dans toutes les classes C sur le chemin entre Objet et S. On dira que I **hérite** de ces objets C. I possèdera alors tous les attributs définis entre Objet et S. Les méthodes attachées à l'attribut A de I seront toutes celles définies pour A entre Objet et S. Ces méthodes s'appliqueront dans l'ordre de rencontre entre S et Objet (du plus spécialisé vers le plus général).
Une spécialisation S a donc trois moyens d'agir sur ses instances:
- rajouter un attribut n'existant pas dans la structure de la classe mère.
- restreindre les possibilités de valuation d'un attribut existant.
- enrichir la liste des méthodes disponibles pour un attribut existant.

L'héritage est un mécanisme simple pour factoriser la connaissance et classer les objets suivant leurs caractéristiques. On remarquera, d'autre part, que S3 implique que toute instance (y compris les classes qui sont instances de Schema elle-même spécialisation d'Objet) sont des instances d'Objet. La proposition B1 a donc été réalisée: tout est objet.

Tout ce qui a été dit sur les différents graphes d'une représentation de connaissance centrée-objet se résume par la figure suivante:



II. Présentation de Shirka.

Shirka /Rechenmann 85/ est un système de gestion de bases de connaissances centrées-objet écrit en Le_Lisp /Chailloux& 86/. La réalisation concrète d'un système de maintenance de la vérité se fera sur ce système. Les spécificités de Shirka par rapport au modèle standard sont donc présentées ici.

1. Restrictions sur les listes.

Deux facettes **$card-max** et **$card-min** permettent de spécifier le nombre maximum et minimum de valeurs d'un attribut multivalué.

◊ Exemple II.6

Il est ainsi posible d'obliger une devise à être utilisée dans au moins un pays. Il suffit d'utiliser la facette $card-min dans la déclaration de la classe devise:

```
{ devise
  est-un        =          schema
   utilisee      $liste-de  symbole ;
```

```
                $card-min 1 }
```

◊

2. Le filtrage.

Le filtrage est un puissant mécanisme d'inférence permettant d'obtenir une valeur au travers d'autres objets. Concrètement, un filtrage permet de sélectionner les objets comprenant certaines particularités, sur les valeurs de leurs attributs, et de ramener la valeur d'un de ces attributs.

Il est introduit à l'aide de la facette **$sib-filtre** suivit d'une classe d'objets, nommée filtre. Le système sélectionnera toutes les instances de la classe dont le filtre est une spécialisation pouvant être instances du filtre. Un des attributs du filtre contiendra la facette **$var->** ou **$var-liste->** qui permettra de donner à l'attribut désigné par la variable la valeur de cet attribut.

◊ Exemple II.7

Le schéma change peut être encore amélioré en introduisant le filtrage sur le cours des devises.

```
{ change
   sorte-de          =        operation
   est-un            =        schema
   devise            $un      devise;
                      $defaut  dollar
   sortie-franc      $un      reel
   entree-devise     $un      reel ;
                     $sib-exec
                       { calc-change
                          monnaie   $var<-   devise
                            parite      $sib-filtre
                                        { cours
                                            titre          $var<-   monnaie
                                              prix-vente   $var->   parite }
                          francs    $var<-   sortie-franc
                          montant   $var->   entree-devise } }
```

La valeur parite nécessaire au calcul du $sib-exec est maintenant obtenue grâce à un filtre. Ce filtre cherche dans tous les objets cours celui qui se rapporte au titre devise et en ramène l'attribut prix-vente qui sera la parité de cette monnaie. Pour frais-honolulu, la mise en correspondance se fera avec l'objet:

```
{ cours-dollar
   est-un        =   cours
   titre         =   dollar
   prix-achat    =   6.95
   prix-vente    =   7.12 }
```

et rendra la valeur 7.12.

◊

3. Les variables.

La notion de variable est introduite au niveau de chaque objet. Ainsi, au sein du même objet ou entre différents objets imbriqués (filtres, méthodes) les valeurs d'attributs peuvent transiter. Pour cela on utilise différentes facettes:

- **$var-nom** permet d'attribuer à une variable la valeur d'un attribut. A noter que, par défaut, à chaque attribut est associé une variable dont le nom est celui de l'attribut.

- **$var<-** et **$var-liste<-** permettent de valuer un attribut avec la valeur de la variable désignée.

- **$var->** et **$var-liste->** permettent de valuer l'attribut désigné par la variable avec la valeur de l'attribut contenant ces facettes.

◊ Exemple II.8

L'exemple II.7 contient plusieurs utilisation de la facette $var<-. Elles permettent:
- Que l'attribut monnaie de calc-change soit valué avec l'attribut devise de l'instance de change qui le contiendra.
- Que l'attribut francs de calc-change soit valué avec l'attribut sortie-franc de l'instance de change qui le contiendra.
- Que l'attribut titre de cours soit valué avec l'attribut monnaie de l'instance de calc-change qui le contiendra.

De même les utilisations de $var-> permettent de transmettre:
- A l'attribut parite de calc-change la valeur de l'attribut prix-vente de cours.
- A l'attribut entree-devise de change la valeur de l'attribut montant de calc-change.

◊

4. La contrainte.

La notion de contrainte, qui permet d'exiger une cohérence entre différentes valeurs d'attributs d'un même objet est intégrée dans Shirka. La contrainte introduite par la facette **$a-vérifier**, dans l'attribut **lui-meme**, appelle un prédicat qui vérifiera que les valeurs d'attributs passées en argument vérifient certaines conditions. La contrainte ne concernant qu'un seul attribut peut être placée au niveau de cet attribut.

◊ Exemple II.9

Il est posible de contraindre le prix de vente d'un titre à être supérieur à son prix d'achat, par l'ajout d'un simple prédicat. Le schéma cours devient alors le suivant:

```
{ cours
    est-un        =          Schema
    sorte-de      =          Objet
     lui-meme      $a-verifier
                       { superieur
                            sup      $var<-   prix-vente
                            inf      $var<-   prix-achat }
     titre          $un       Objet
     prix-vente     $un        reel
     prix-achat     $un        reel  }
{ superieur
    est-un         =          Schema
    sorte-de       =           predicat
     nom-fct         $valeur   superieur
    sup             $un       reel
    inf             $un       reel }
(de superieur ( sch )
    (> (valeur? 'sup sch) (valeur? 'inf sch)))
```

◊

5. Les réflexes.

La notion de réflexe est présente dans nombre de langages objets, elle permet aux objets de réagir à un évènement. Dans Shirka, les réflexes permettent de réagir après ou avant l'ajout d'une valeur, la modification d'une valeur ou la suppression d'une valeur d'attribut. On a donc les facettes suivantes: **$avant-ajout, $après-ajout, $avant-modif, $après-modif, $avant-sup, $après-sup.** La réaction peut aussi être déclenchée en cas d'échec ou de réussite d'une inférence, on a alors les facettes **$si-echec** et **$si-succès.** Chaque facette réflexe appelle une procédure qui sera exécutée dans les circonstances spécifiées.

◊ Exemple II.10

Divers réflexes peuvent être introduits dans la classe change afin de rendre les mécanismes employées par le système plus compréhensibles et plus souples à l'utilisateur. Ainsi, une facette $si-echec permet d'indiquer à l'utilisateur l'échec du filtrage sur les cours de la monnaie:

```
{ change
   sorte-de          =          operation
```

```
est-un          =       schema
devise          $un     devise;
                  $defaut  dollar
 sortie-franc   $un      reel
 entree-devise  $un      reel ;
                  $sib-exec
                    { calc-change
                       monnaie  $var<-   devise
                        parite     $sib-filtre
                                   { cours
                                       titre        $var<-   monnaie
                                        prix-vente  $var->   parite } ;
                                $si-echec
                                   { pas-de-cours
                                       titre        $var<-   monnaie }
                       francs     $var<-    sortie-franc
                       montant    $var->    entree-devise } }

  { pas-de-cours
   est-un        =        methode
    nom-fct       $valeur   pas-de-cours
   titre         $un       devise }
(de  pas-de-cours  (schema)
   (print "Le cours du "
          (valeur? 'titre schema)
        " est suspendu."))
◊
```

6. Le multihéritage.

Le multihéritage est introduit dans le modèle objet. C'est-à-dire que l'attribut sorte-de est multivalué et qu'une classe devient spécialisation d'un ensemble non vide de classes. Il découle de cette proposition qu'Objet est l'élément maximal du demi-treillis supérieur qu'est l'ensemble des classes muni de la relation sorte-de. Ce demi-treillis est appelé **graphe d'héritage**. Cela modifie le sens de la proposition S3:

S3: SI S est une spécialisation de C alors toute instance de S est une instance de C

Il en résulte que la structure d'une instance I de S est définie non seulement dans S mais aussi dans toutes les classes C sur un chemin permettant d'aller de Objet à S.

Par conséquent, si l'attribut A est présent dans la structure induite par une des classes dont S est spécialisation alors l'attribut A est présent dans la

structure des instances de S. Le multihéritage est donc pratique pour synthétiser deux concepts en un seul.

Cependant il pose de nouveaux problèmes au niveau de l'héritage. En effet, l'ordre de l'héritage simple dépend de l'ordre de succession des classes dans l'arbre qui est unique. Le multihéritage nous oblige à définir un ordre de parcours du graphe pour déterminer quelle méthode activer avant quelle autre. Dans Shirka, les valeurs de l'attribut sorte-de sont ordonnées et le graphe est parcouru en profondeur d'abord.

7. Les primitives d'utilisation.

Un certain nombre de fonctions Lisp permettent à un utilisateur ou à un programme de manipuler la base de connaissance.
(**def-sh** <schema>) déclare un schéma qui est chargé dans la base.
(**sup-inst** <schema>) supprime le schéma <schema> de la base.
(**aj-valeur** <schema> <attribut> <valeur> <fin?>) ajoute la valeur <valeur> à la <fin?> de l'attribut <attribut> de schéma <schema>.
(**mod-valeur** <schema> <attribut> <valeur1> <valeur2>) substitue dans l'attribut <attribut> du schéma <schema> la valeur <valeur2> par la valeur <valeur1>.
(**sup-valeur** <schema> <attribut> <valeur>) supprime la valeur <valeur> de l'attribut <attribut> du schéma <schema>.
(**valeur?** <attribut> <nom-schema>) rend la valeur de l'attribut <attribut> du schéma de nom <nom-schema>. Cette fonction lance les inférences si celles-ci sont nécessaires.

Conclusion.

L'informatique et plus précisement l'intelligence artificielle, utilisent de de grandes quantités de connaissance très structurée et souvent remises à jour. Il semble que la représentation des connaissances sous forme d'objets constitue une bonne approche du problème en introduisant des concepts très utiles dans ces domaines (structuration de la base, factorisation des données, intégrité de l'objet...).

Il est bon de pouvoir disposer d'outils à la fois fiables et performants. La partie suivante présente une technique tentant d'améliorer les performances des systèmes de représentation de connaissance centrée-objet, moyennant l'introduction d'un système de maintenance de la vérité.

troisième partie

Maintenance de la vérité et représentations de connaissance centrées-objet

Les représentations de connaissance centrées-objet sont économes en mémoire parce qu'elles factorisent les données partagées par des objets d'une même classe. C'est leur grand avantage. Cependant, cet avantage implique une recherche et un calcul des données qui sont coûteux en temps. Dans le cas où cette factorisation est un moyen de structurer la base plutôt qu'une manière d'économiser de la place, il peut être utile de conserver chaque donnée inférée.

Ce procédé est appelé "caching" par analogie avec les mémoires cache. Il consiste à conserver au sein de l'attribut la valeur inférée comme une valeur normalement communiquée au système. Dans certaines implémentations, ce "caching" peut ne pas accroître la place mémoire utilisée.

Il est bien évident que le "caching" permet d'augmenter les performances du système en diminuant les temps de recherche et de calcul.

Mais le "caching" n'a pas que des avantages. Il est en effet inadapté à toute utilisation "non-monotone" de la base de connaissance c'est-à-dire une utilisation où la modification (remplissage, suppression, remplacement) de la valeur d'un attribut entraine la remise en cause de calculs de la valeur d'autres attributs.

Le point de départ de notre recherche est d'utiliser dans une représentation de connaissance centrée-objet le mécanisme d'un système de maintenance de la vérité afin de maintenir la validité permanente des valeurs cachées.

Dans un premier temps on évaluera les avantages et inconvénients des système de maintenance de la vérité classiques utilisés dans une représentation de connaissance centrée-objet. Puis on proposera un modèle de système de maintenance de la validité à plusieurs niveaux. L'implémentation du système proposé au sein d'une représentation de connaissance centrée-objet précise, Shirka, sera exposée dans la partie suivante.

I. TMS classiques et représentations de connaissance centrées-objet.

Il est 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 différents aménagements à apporter aux mécanismes précédement décrits.

1. Suppositions et contextes.

Le système 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 (ART et la dernière version de KEE par exemple) utilisent un système de maintenance de la vérité 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 évoluent chaque objet. Ce problème, qui est tout à fait en rapport avec les contextes, est en fait beaucoup trop compliqué pour un TMS à propagation.

Cependant, un aspect du TMS à contextes pose problème vis-à-vis de l'objectif que nous nous fixons. En effet, nous voulons que l'utilisateur ne se rende pas compte de la présence du système de maintenance de la vérité et que celui-ci donne des réponses avec une rapidité accrue. Or l'ATMS, pour vérifier la validité d'une valeur cachée, cherchera à savoir si celle-ci est dans le contexte, c'est-à-dire si ses antécédents sont valides. Cette vérification prend un certain temps, parfois autant que le calcul lui-même, alors que la réponse est instantannée si l'on maintient l'ensemble des états valides.

D'autre part, l'idée d'utiliser les seules suppositions pour justifier les nœuds est bonne si l'on considère un système de raisonnement classique nécessitant peu d'axiomes par rapport au nombre d'inférences. Dans une base de connaissance, les suppositions correspondent à toutes les données présentes dans la base au début du raisonnement. Cela représente beaucoup trop de contextes possibles.

Enfin, 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, un TMS à propagation est plus adapté aux représentations de connaissance centrées-objet qu'un TMS à contexte.

2. Maîtrise du système de raisonnement.

Dans les systèmes de Doyle et DeKleer, le système de raisonnement est séparé du système de maintenance de la vérité. Leurs systèmes de raisonnement sont figés, c'est-à-dire qu'ils utilisent toujours les mêmes règles, ils n'ont donc besoin d'enregistrer des dépendances qu'entre données. Or, les règles d'inférence sont aussi susceptibles d'évoluer que les données et par conséquent il est utile de les maintenir.

Au sein des représentations de connaissance centrées-objet, les inférences sont dirigées par les facettes, qui permettent de calculer une valeur. Ces facettes peuvent être modifiées au cours de l'utilisation de la base. Des solutions devront être proposées pour maintenir les facettes qui sont les règles de notre système de raisonnement.

3. Justifications par défaut et paradoxes.

Dans les représentations de connaissance centrées-objet toutes les facettes de valuation, à l'exeption de $valeur, sont des facettes agissant par défaut.

En effet, elles sont utilisées uniquement par défaut de valeur fournie (=) ou imposée ($valeur), et défaut d'autres méthodes. Il est inutile d'ajouter cette fonctionnalité au système de maintenance de la vérité. Les justifications dans les représentations de connaissance centrées-objet seront donc considérées comme des listes de données. Cette absence d'OUTliste dans les dépendances évite d'introduire dans les représentations de connaissance centrées-objet des paradoxes que le système de maintenance de la vérité ne saurait gérer.

4. Représentations de connaissance centrées-objet et contradictions.

Quand des contradictions apparaissent-elles dans une représentation centrée-objet ?

On peut avancer trois réponses à cette question:
a) Quand un attribut possède une valeur et que l'on veut lui en affecter une seconde.
b) Quand on veut affecter une valeur à un attribut alors qu'il est placé sous une facette $valeur.
c) Quand deux méthodes différentes appelées pour calculer une valeur ne rendent pas le même résultat.

Cependant, en y regardant de plus près, ces contradictions n'apparaissent jamais ou 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 un comportement prédéfini:
a) Si on désire remplacer la valeur initiale par une seconde, tout se passe bien. Si on désire ajouter la seconde valeur à celle qui est déja présente, ici encore 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 un $defaut à la place du $valeur ce qui lui aurait permis de modifier la valeur de l'attribut.
c) Enfin le dernier cas n'apparaît jamais dans les représentations centrées-objet, car les méthodes de calcul sont invoquées dans un ordre dépendant premièrement du graphe d'héritage, deuxièmement de la priorité entre les différents types de méthode, et le calcul s'arrête au premier résultat trouvé.

C'est donc l'idée même de contradiction qui a 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. Un pas de plus vers la détection et la suppression des "contradictions" serait de s'assurer que la base est construite en accord avec des objectifs. On aborde alors les problèmes de preuve de programme et d'aide à la définition d'une base de connaissance.

Par ailleurs, les réponses données ici mettent en évidence l'importance de la conception de la base qui fige ce déterminisme. Le système tel qu'il doit être conçu, fait tout pour empêcher l'émergence d'une contradiction. Si

le système de maintenance de la vérité est utile ce ne sera pas pour sa gestion des contradictions. C'est en étendant le système vers d'autres fonctionnalités, que l'on se rend compte de son intérêt.

II. Le "caching".

C'est en étudiant plus profondement le "caching" qu'on réalisera la pleine utilité du système de maintenance de la vérité. C'est en effet l'introduction du "caching" dans les représentations de connaissance 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 de connaissance centrée-objet intégrant le "caching".

1. Le "caching" sans maintenance.

Le "caching" est implémenté de manière brute dans les systèmes SRL (Scheme Representation Language /Wrigth& 83/) et RLL (Representation Language Language /Greiner& 80/). 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 III.1

Dans l'exemple présenté tout au long de la troisième partie, un appel de (valeur? entree-devise frais-honolulu) va lancer le calcul nécessaire. Le mécanisme de "caching" va mémoriser cette valeur au sein de l'objet frais-honolulu et la réponse ($ 17338.5) sera fournie à l'utilisateur. Si celui-ci a besoin une seconde fois de cette valeur, la réponse à sa demande se fera plus rapidement puisque la valeur inférée aura été conservée. L'objet frais-honolulu sera conservé ainsi:

```
{ frais-honolulu
  est-un          =    change-entr
  sortie-franc    =    123450.0
  devise          =    dollar
  entree-devise   =    17338.5 }
```

◊

Cependant, ce caching brut comporte quelques inconvénients, en effet, si entre deux appels à la fonction *valeur?* le contenu de la base à été modifié de telle sorte qu'un des paramètre entrant dans le calcul de la valeur la réponse rendu à l'utilisateur sera erronée.

◊ Exemple III.2

Les cours du change et des devises ont la propriété de... changer, ainsi si le cours du dollar est passé de 7.12 à 7.15 et que l'utilisateur de la base demande la valeur de (valeur? entree-devise

frais-honolulu), le lendemain, le système renverra 17338.5 au lieu de 17265.7.
◊

2. Maintenance sur les arguments.

Une solution à ce problème est implémentée dans ART (Advanced Reasonning 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, c'est typiquement la méthode proposée par Jon Doyle.

◊ Exemple III.3

L'exemple précédent donnerait à la fin du calcul:

```
{ frais-honolulu
  est-un          =     change-entr
  sortie-franc    =     123450.0
  devise          =     dollar
  entree-devise   =     17338.5 }

{ cours-dollar
  est-un          =     cours
  titre           =     dollar
                          [utilise-par frais-honolulu entree-devise]
  prix-achat      =     6.95
  prix-vente      =     7.12
                          [utilise-par frais-honolulu entree-devise] }
```

Ainsi, à la modification du cours du dollar, on invalidera toutes les valeurs conséquentes et donc celle de l'attribut entree-devise de frais-honolulu, et le schéma frais-honolulu redeviendra:

```
{ frais-honolulu
  est-un          =     change-entr
  sortie-franc    =     123450.0
  devise          =     dollar }
```

Et une nouvelle demande de (valeur? entree-devise frais-honolulu) entrainera un nouveau calcul et une nouvelle mémorisation.
◊

Cependant la maintenance des dépendances entre valeurs ne suffit pas. En effet, si l'on imagine qu'au lieu de modifier l'un des paramètres du calcul, c'est la méthode qui a été modifiée, la valeur calculée n'est plus valide et pourtant elle resterait encore "cachée" dans le cas présent.

◊ Exemple III.4

Si par exemple une nouvelle règlementation entre en vigueur, qui fait payer une pénalité de 0.05% aux gens qui changent plus de 20 000 francs, la méthode calc-change sera modifiée afin de tenir compte de cet aspect de l'opération. Par contre la valeur mémorisée ne sera pas invalidée, encore une fois le système répondra à une nouvelle requète de façon erronée.
◊

3. Maintenance sur les méthodes.

La méthode la plus simple est celle envisagée pour PAUL (/Hein 83/). Elle consiste en la mémorisation de la méthode utilisée de la même manière que les données.

◊ Exemple III.5

Dans l'exemple cité la mémorisation se ferait ainsi:

```
{ calc-change
  est-un      =          schema
  sorte-de    =          methode
   util-par   =             [ frais-honolulu entree-devise ]
   nom-fct    $valeur     calc-change
  monnaie     $un         objet
  parite      $un         reel
  francs      $un         reel
  montant     $un         reel }

{ cours-dollar
  est-un          =      cours
  titre           =      dollar
                           [utilise-par frais-honolulu entree-devise]
  prix-achat      =      6.95
  prix-vente      =      7.12
                           [utilise-par frais-honolulu entree-devise] }

{ frais-honolulu
  est-un          =      change-entr
   sortie-franc   =      123450.0
  devise          =      dollar
   entree-devise  =      17338.5 }
```

Ainsi, à la modification de la fonction calc-change, les calculs auxquels elle a participé seront invalidés et le schéma

frais-honolulu redeviendra:

{ frais-honolulu
est-un = change-entr
sortie-franc = 123450.0
devise = dollar }

Et une nouvelle demande de (valeur? entree-devise frais-honolulu) entrainera un nouveau calcul et une nouvelle mémorisation.

◊

Mais cette méthode pose un dernier problème soulevé dans /Hein 83/. Il est possible qu'entre deux calculs ce soit non plus un objet qui ait été modifié mais l'ordre d'activation des méthodes dans le graphe d'héritage.

◊ Exemple III.6

Si la règlementation est de nouveau modifiée dans un souci de ne pas décourager l'exportation, et que le seuil est placé à 100 000 francs et la pénalité à O.O3% pour les entreprises, une nouvelle méthode devra alors être insérée dans le graphe d'héritage au niveau de la classe change-entr.

Une fois de plus un appel à (valeur? entree-devise frais-honolulu) rendra une valeur erronée car le calcul ne doit plus être fait par la méthode calc-change mais la nouvelle méthode calc-change-entr.

◊

4. Maintenance sur le graphe d'héritage.

Il ne s'agit plus ici 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.

◊ Exemple III.7

Ainsi dans l'exemple précédent, l'algorithme remontera dans les classes supérieures à change-entr. Il rencontrera change dont l'attribut entree-devise utilise la méthode calc-change. Cette méthode ayant servi à valuer une instance de calc-change-entr (en l'occurence frais-honolulu), ce calcul sera invalidé.

Un nouvel appel à (valeur? entree-devise frais-honolulu), entrainera alors un nouveau calcul.

◊

III. Fonctionnalités d'un système de maintenance pour une représentation de connaissance centrée-objet.

1. Intérêt de chaque système.

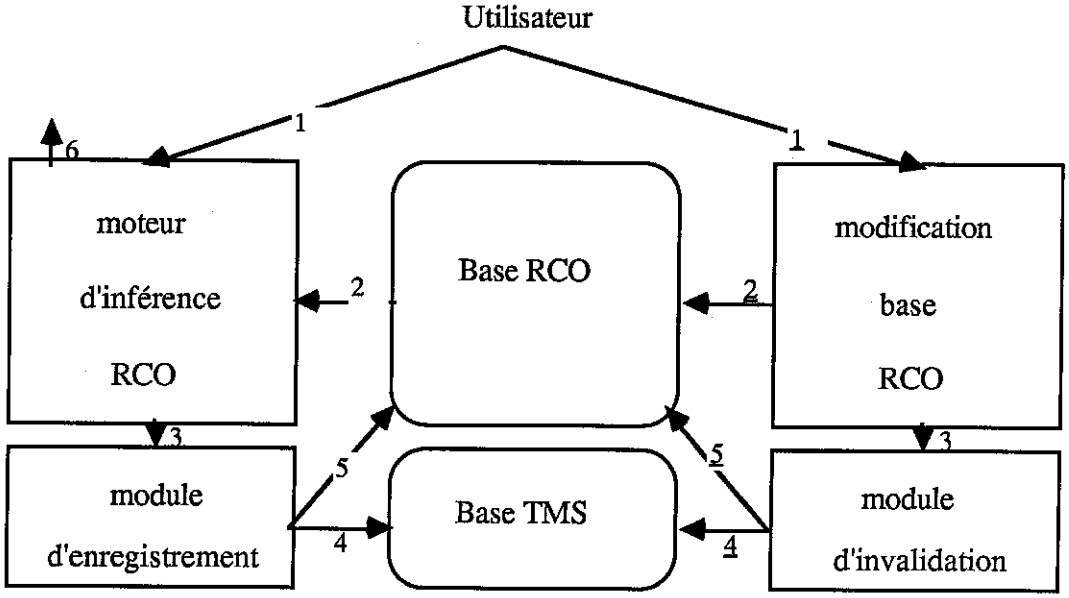
On peut se rendre compte que chacun des systèmes présentés en II a son intérêt dans un contexte bien précis, ainsi:
- Le caching sans maintenance est parfaitement adapté à l'utilisation monotone d'une base figée.
- La maintenance sur les valeurs est adaptée à une utilisation non-monotone d'une base de connaissance elle aussi figée.
- La maintenance sur les méthodes est intéressante pour la mise au point d'une base de connaissance dont le graphe d'héritage ne sera plus modifié.
- La maintenance sur le graphe d'héritage est utile en cas d'utilisation d'une base dont la structure risque d'évoluer ou de s'enrichir en 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 nous proposerons un système dans lequel chacun des niveaux est utilisable.

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

2. Architecture du système.

Il ressort donc que le système de maintenance de la vérité doit s'intégrer à une représentation centrée-objet de façon à enregistrer les inférences produites et invalider les dépendances. L'architecture suivante semble donc s'imposer pour un tel 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.

3. Description fonctionnelle.

Les fonctions que devra effectuer un système réalisant tout ce qui a été présenté plus haut sont présentées ici sous la forme d'un algorithme général pour chacune d'elle. La première fonction modifiée provient du module d'inférence. Les autres sont utilisées pour modifier la base d'objets.

Le caching sans maintenance est très simple à implémenter. Il s'agit d'une modification de la fonction **valeur?** 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 imédiatement rendue, comme si l'utilisateur l'avait lui-même fournie. Par ailleurs,la fonction devra mémoriser les inférences qu'elle fait et les valeurs inférées. C'est la base minimale du caching sans maintenance.

a) Demande d'une valeur.

valeur? [%attribut %instance] =
SI une valeur existe pour %attribut
ALORS -> cette valeur
SINON calculer le résultat.
enregistrer la justification.
mémoriser le résultat.
-> ce résultat.

La forme de **valeur?** présentée ici 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 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.

a') Demande d'une valeur.

```
valeur? [ %attribut %instance ] =
                SI une valeur existe pour %attribut
                ALORS -> cette valeur
                SINON SI une méthode utilisée par une justification est IN
                      ALORS recalculer les valeurs d'attributs
                                enregistrer la nouvelle justification
                                enregistrer le résultat
                                -> le résultat
                           SINON utiliser l'algorithme classique pour trouver la
                                 méthode.
                                 SI cette méthode est dans la liste des justifications
                                 ALORS mettre ces justifications à IN
                                           relancer cette fonction.
                                 SINON utiliser l'algorithme classique pour calculer le
                                       résultat.
                                           enregistrer la justification.
                                           mémoriser le résultat.
                                           -> ce résultat.
```

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.

b) Modification d'une valeur.

```
mod-valeur [ %instance %attribut %valeur ] =
                - POUR TOUT e dans la liste des conséquents de la valeur
                  FAIRE mettre l'état de e à OUT
                              propager les OUT de la même façon sous e
                - modifier la valeur de %instance.%attribut en %valeur
```

c) Suppression d'une valeur.

```
sup-valeur [ %instance %attribut ] =
                - POUR TOUT e dans la liste des conséquents de la valeur
                  FAIRE mettre l'état de e à OUT
                              propager les OUT de la même façon sous e
                - supprimer la valeur
```

d) Ajout d'une valeur.

```
aj-valeur [ %instance %attribut %valeur ] =
                - SI %instance.%attribut == valeur_mémorisée
                  ALORS POUR TOUT e dans la liste des conséquents de la valeur
```

FAIRE mettre l'état de e à OUT
propager les OUT de la même façon sous e
- modifier la valeur de %instance.%attribut en %valeur

e) Suppression d'une instance.

sup-inst [%instance] =
- POUR TOUT a attribut de %instance
FAIRE sup-valeur [%instance a]

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.

f) Modification d'une méthode.

mod-fac [%instance %attribut %facette %methode] =
- POUR TOUT c dans la liste des conséquents de %méthode
FAIRE détruire la justification c
propager les invalidations sous c

Comme pour les méthodes, le graphe d'héritage est géré par une unique fonction, mais celle-ci est plus complexe. L'algorithme 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.

g) Ajout d'une méthode.

aj-fac [%classe %attribut %methode] =
POUR TOUT c ; %classe sorte-de c
FAIRE POUR TOUT f = facette [c %attribut]
FAIRE POUR TOUT i.a = dépendance [f]
FAIRE SI i est-un %classe
ALORS supprimer [i a]
Ces différentes fonctions doivent remplir pleinement l'office de du TMS décrit.

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 plusieurs fois la même inférence.

Dans ce contexte, un mécanisme de maintenance de la vérité a son utilité et permettra d'assurer la validité des connaissances stockées dans le système. Il offrira même un complément au "caching" puisqu'il permet d'effectuer plus facilement les calculs en réutilisant des parties de calculs déja effectués.

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). On peut d'ores et déjà se rendre compte des avantages apportés par un TMS en contrepartie d'un code minimal. Cet accroissement des performances est bien sûr obtenu au prix d'une plus grande place mémoire occupée par la base.

quatrième partie

Réalisation d'un système de maintenance

de la vérité pour Shirka

La réalisation d'un système de maintenance de la vérité pour Shirka repose sur les deux actions principales précédemment mises en évidence: mémorisation des inférences et propagation des invalidités.

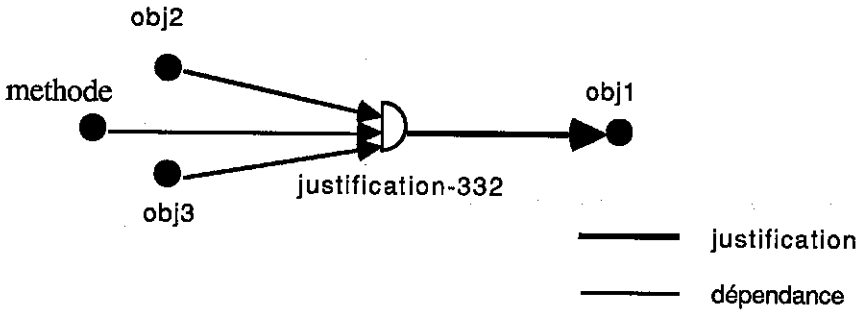
La présentation du TMS de Shirka sera donc développée suivant ces deux axes. La structure de donnée choisie pour la mémorisation des inférences, puis les différents programmes construits seront présentés.

Tout comme Shirka, la programmation a été faite en Le_Lisp 15.1. On y utilise amplement les primitives internes au système Shirka.

I. Structure de donnée pour les enregistrements.

Le premier travail du TMS est la mémorisation des inférences. Des justifications permettant de lier entre eux les objets dont dépendent un calcul sont insérées au sein de la base.

Pour chaque justification on a le diagramme suivant:



L'enregistrement des justifications au sein des objets nécessite des structures d'accueil. Shirka étant déjà un logiciel fini il faut insérer ces structures d'accueil dans le logiciel. D'autre part, il faut déterminer la structure qui accueillera la justification, entité qui n'a pas été bien précisée au sein d'une représentation de connaissance centrée-objet.

1. Ce qui existe sous Shirka.

Shirka mémorise les valeurs au sein des instances de la façon suivante:

schema ::= #[<est-un> <util-par> <nom-inst> <valeur-d-attribut>*]

◊ Exemple IV.1

Le schéma frais-honolulu est représenté ainsi:

#[change-entr () frais-honolulu dollar 123450.0 ()]

◊

Un objet est donc un vecteur contenant au moins trois cases, ou "slots". Chaque case contient la valeur de l'attribut qui y correspond quelqu'en soit son type. Une cellule plus complexe doit donc être implantée sur cette case.

2. Ce qui doit être enregistré.

Trois sortes d'entités sont susceptibles d'apparaître dans cette cellule:
- la valeur si elle est connue.
- les justifications de valeurs calculées dans le cas contraire.
- la liste des attributs dépendants de la valeur.

Afin d'accéder le plus rapidement possible aux valeurs stockées, il est préférable de conserver une case pour la valeur. Une seconde case sera ménagée pour recevoir justifications et dépendances. On peut donc proposer la structure suivante:

#[<est-un> <util-par> <nom-inst>
<valeur> <mémorisation> <valeur> <mémorisation>...]
où <mémorisation> ::= (<justifications> . <dépendances>)

La taille des vecteurs sera doublée afin de disposer de deux cases au lieu d'une par attribut. La structure destinée à recevoir dépendances et justifications sera un doublet mais elle pourrait aussi bien être un vecteur.

La première valeur du doublet concerne les justifications, c'est-à-dire les justifications qui soutiennent, ou ont soutenu, la, ou les, valeurs cachées. La structure est construite ainsi:

#[(INJust) (OUTArgs) (OUTMeth)]

Il s'agit d'un vecteur contenant trois cases, une première contenant les justifications actives, c'est-à-dire celles qui soutiennent la ou les valeurs cachées (c'est cette liste qu'il faut consulter pour savoir si la valeur qui est dans la case <valeur> est une valeur cachée ou fixe). La seconde case contient une liste de justifications momentanément invalidées à cause d'un de leurs arguments c'est-à-dire qu'il suffit de recalculer les arguments pour avoir les nouvelles valeurs. La troisième case contient une liste de justifications invalidées à cause des méthodes qui ont été masquées. Cette structure permet donc clairement le recalcul de la valeur tel qu'il a été exposé dans le second algorithme de *valeur?*.

La seconde valeur du doublet est celle correspondant aux dépendances, il s'agit d'une liste des justifications dans lesquelles la valeur intervient. Ainsi, à l'invalidation de la valeur, on pourra invalider les inférences soutenues par les justifications enregistrées.

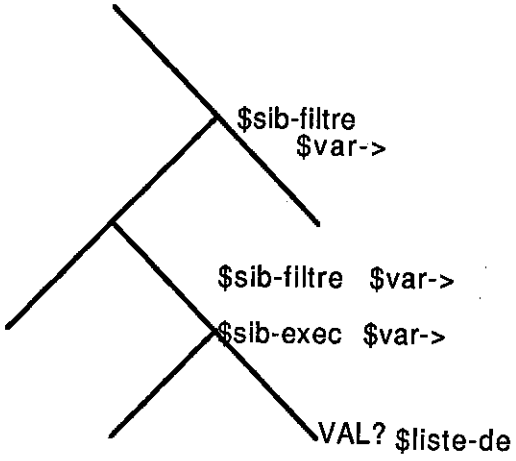
◊ Exemple IV.2

A la fin du calcul de (valeur? entree-devise frais-honolulu), le schéma frais-honolulu serait donc ainsi:

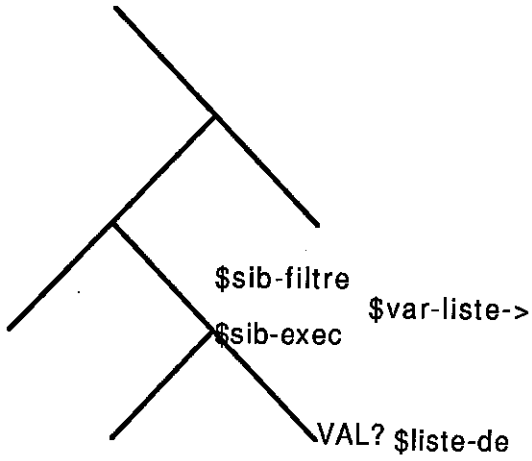
```
#[change-entr () () () frais-honolulu () dollar (() %just-var<-108)
      123450.0 (() %just-var<-109) 17338.5 (#[(%just-sib-110) () ()])]
```

◊

Les justifications sont mémorisées au sein de listes, car les attributs multivalués peuvent être calculés de différentes manières dans Shirka. Ils peuvent être obtenus sous forme d'une liste déjà composée, ou élément par élément:



les trois méthodes doivent entrer en action et on fait l'union des résultats.



une méthode est activée, rendant une liste.

Une première idée, en ce qui concerne les attributs multivalués, est de mémoriser la valeur comme les autres valeurs, c'est-à-dire classiquement sous la forme de la liste obtenue. Cette idée est justifiée dans le cas d'une liste obtenue d'un seul bloc, mais des problèmes apparaissent si la liste est obtenue élément par élément:

- Les éléments sont obtenus à partir de différentes méthodes, il est donc impossible de mémoriser toute la liste sous une seule méthode comme dans la structure de donnée classique.
- Les éléments sont obtenus grâce à différentes valeurs indépendantes les unes des autres qui ne peuvent être stockées comme telles dans la structure de donnée classique.

Il apparaît donc que si les éléments sont obtenus un par un, il faudra en tenir compte dans la mémorisation. De plus, enregistrer les éléments un par un comporte différents avantages:

- La possibilité d'invalider un seul élément au lieu de remettre en cause toute la liste (moins de calcul au second appel).
- La possibilité d'invalider les seuls éléments obtenus par une méthode précise, quand celle-ci est remise en cause (toujours moins de calculs).

Les éléments obtenus seront enregistrés un par un, en utilisant la structure classique pour tous les éléments, c'est ce qui justifie que les cases du vecteur contiennent des listes.

Pour résumer, un schéma sera représenté dans Shirka par le vecteur:

#[<est-un> () <util-par> () <nom-inst> ()
{<valeur> (#[({<just>}*) ({ <just> }*) ({ <just> }*)] . { <just> }*) }*
]

3. Structure générale des justifications.

Une fois déterminé l'emplacement des enregistrements, il reste à déterminer leur forme. Chaque justification dépend d'un certain nombre de paramètres:

- valeur justifiée.
- type de méthode.
- nom et localisation de la méthode.
- instance filtrée pour les $sib-filtre.
- instance de la méthode pour $sib-exec.
- localisation du résultat.

Au regard de la liste des attributs d'une justification il apparaît qu'un objet est la structure de donnée adaptée pour les représenter. Ce n'est pas la seule, en effet une structure en cascade:

```
( { (<IN/OUT> <meth-adr> { <arg-adr> }*
     { (<IN/OUT> <valeur>
          { (<IN/OUT> <arg-val>) }* )
       }* )
  }* )
```

où <IN/OUT> est un booléen indiquant la validité de la méthode, de la valeur inférée, ou de la valeur des arguments permet de retrouver facilement et avec le minimum de tests les méthodes, valeurs ou arguments valides.

Cependant on a choisi d'utiliser les objets de Shirka comme structure de justification car ceci permet d'utiliser les primitives de Shirka pour accéder aux données et de ne pas avoir à développer de code et de structures parallèles. Ce choix va plus loin dans l'orthogonalité et obéit au premier précepte des objets: tout est objet... même les justifications.

De nouveaux objets, indépendants du calcul, seront donc destinés à conserver les justifications. La structure générale d'une justification telle qu'elle a été retenue sera la suivante:

```
(justification
  est-un        =          schema
  sorte-de      =          ( objet )
  facette       $un        facette          ; facette utilisée
  class-meth    $un        methode          ; méthode ou filtre
  obj-fac       $un        objet            ; localisation de la méthode utilisée
  att-fac       $un        symbole
  inst-meth     $un        methode          ; instance de la méthode
  obj-val       $un        objet            ; localisation de la valeur inférée
  att-val       $un        symbole
  obj-res       $liste-de  objet            ; localisation des attributs cibles
  att-res       $un        symbole
  resultat      $liste-de  objet )          ; valeur inférée
```

A noter la facette $liste-de pour l'attribut obj-res qui permet de factoriser des résultats. En effet, en cas de réutilisation d'une méthode (deux frères ont les mêmes ancêtres), la justification sera réutilisée.

Ce schéma est bien sûr modulable en fonction de la facette qui a permi de déterminer la valeur justifiée.

4. Les justifications facette par facette.

Les facettes n'utilisent pas le même nombre, ni le même type d'arguments, elles n'utiliseront donc pas les mêmes attributs de la justification. Les différents aspects des justifications sont ici présentés suivant la facette utilisée pour le calcul.

En exemple sont présentés les différentes mémorisations effectuées lors de la demande *(valeur? entrée-devise frais-honolulu)*. Les dessins utilisent les symboles définis dans la seconde partie pour représenter les objets. La partie valeur est maintenant divisée en trois parties: la première contient la valeur proprement dite, la seconde les justifications des valeurs inférées, la troisième les dépendances avec d'autres justifications. Les flèches figurant sur ces dessins sont des pointeurs correspondant dans la réalité à deux attributs des justifications (nom de l'objet pointé, nom de l'attribut pointé). Ces flèches sont en gras quand il s'agit de justifications, normales quand il s'agit de dépendances et en pointillé pour les valeurs.Les objets à valuer sont toujours présentés à gauche.

a) $defaut et $valeur.

Ces facettes peuvent apparaître partout dans une instance, elles contribuent à donner une valeur unique à un attribut et ne prennent pas d' argument. Il est nécessaire de savoir où elles ont été rencontrées pour pouvoir connaître leur priorité dans le graphe d'héritage.

Elles seront conservées sous la forme suivante:

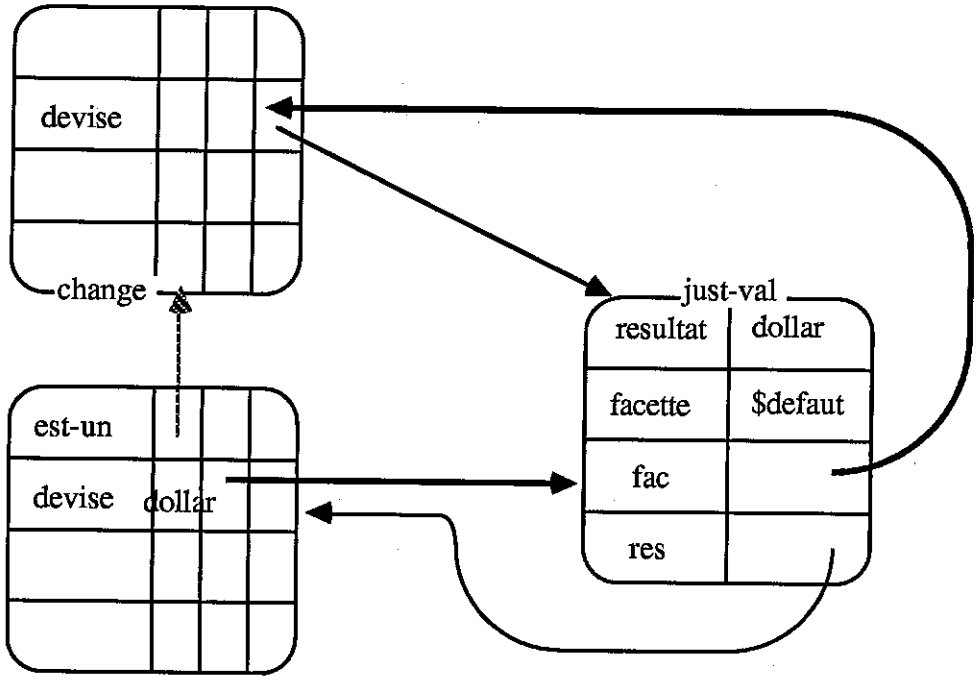
```
(just-val
  est-un      =          schema
  sorte-de    =          ( justification )
  facette     $un        facette                    ; facette utilisée
              $domaine   ( $valeur $defaut )
  obj-fac     $un        objet                      ; localisation de la méthode utilisée
  att-fac     $un        symbole
  obj-res     $liste-de  objet                      ; localisation des attributs cibles
  att-res     $un        symbole
  resultat    $liste-de  objet                      ; valeur inférée
```

L'attribut de la localisation de la méthode n'est pas conservé puisqu'il s'agit forcément de l'attribut recherché.

◊ Exemple IV.3

Lors de l'appel de (valeur? entree-devise frais-honolulu), le système a besoin de connaître la valeur de l'attribut devise de l'objet frais-honolulu. Cette valeur est obtenue grâce à la facette $defaut qui se situe dans la classe change. Ce qui donne le réseau de dépendances

suivant:



◊

b) $var<- et $var-liste<-.

Ce type de facette peut se trouver partout. Il peut avoir été utilisé en référence directe à un attribut ou en utilisant une variable. Une telle facette prend un argument: l'adresse d'une valeur. Il est donc nécessaire de mémoriser avec elle l'endroit où elle a été trouvée, l'adresse à laquelle a été trouvée la valeur demandée (en résolvant si nécessaire les références aux variables) et cette valeur.

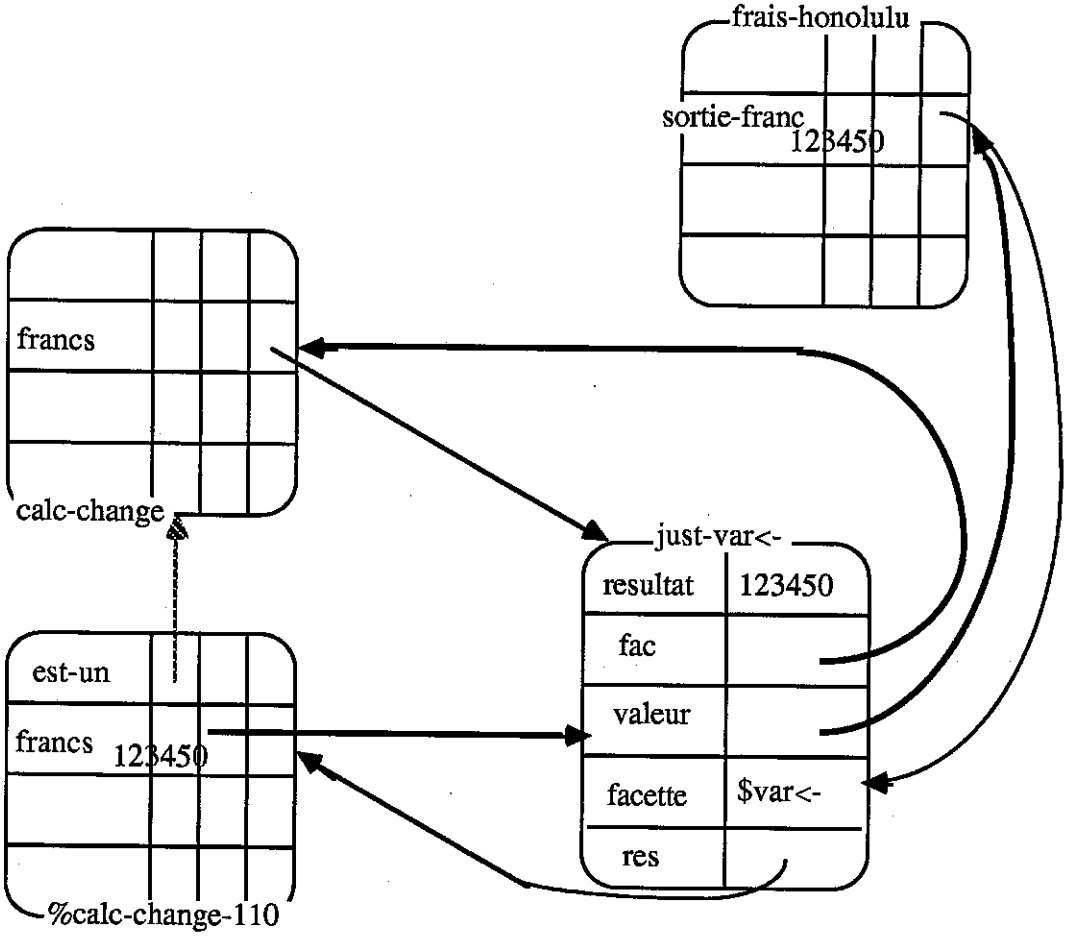
On construit donc la justification suivante:

```
(just-var<-
  est-un      =          schema
  sorte-de    =          ( justification )
  facette     $un        facette                    ; facette utilisée
              $domaine   ( $var<- $var-liste<- )
  obj-fac     $un        objet                      ; localisation de la méthode utilisée
  att-fac     $un        symbole
  obj-val     $un        objet                      ; localisation de la valeur inférée
  att-val     $un symbole
  obj-res     $liste-de  objet                      ; localisation des attributs cibles
  att-res     $un        symbole
  resultat    $liste-de  objet )                    ; valeur inférée
```

◊ Exemple IV.4

Dans l'exemple, plusieurs facettes $var<- sont utilisées. Par exemple, la méthode invoquée a besoin de la somme à changer en francs. Cette valeur qui se trouve dans l'attribut sortie-francs de l'objet

frais-honolulu va, grâce à la facette $var<- située dans la classe représentant l'attachement procédural, valuer l'attribut francs de l'instance d'attachement procédural %calc-change-110. Ce qui sera mémorisé ainsi:



◊

c) $sib-exec.

Les filtres et méthodes se présentent comme des schémas. A la déclaration d'une classe les contenant ils sont compilés en des classes spéciales qui seront instanciées à chaque appel. Ces classes permettent de remplir les schémas-méthode (représentés par les instances) et d'inférer la valeur demandée.

Les schémas-méthode peuvent eux-même contenir des appels à d'autres facettes pour remplir certains de leurs attributs. Il faudra donc maintenir (de manière régressive) les instances générées par les méthodes. Ces instances seront conservées et reliées au reste de la base par les justifications qui permettront de connaître la provenance des valeurs inférées. C'est ce qui a été présenté dans les exemples IV.4 et IV.6.

Pour pouvoir connaître une méthode $sib-exec, il faut conserver son adresse (<obj> <att>), le nom de la classe représentant cette méthode compilée, et

le nom de l'instance de l'objet méthode qui a été créée pour contenir les arguments de la méthode.

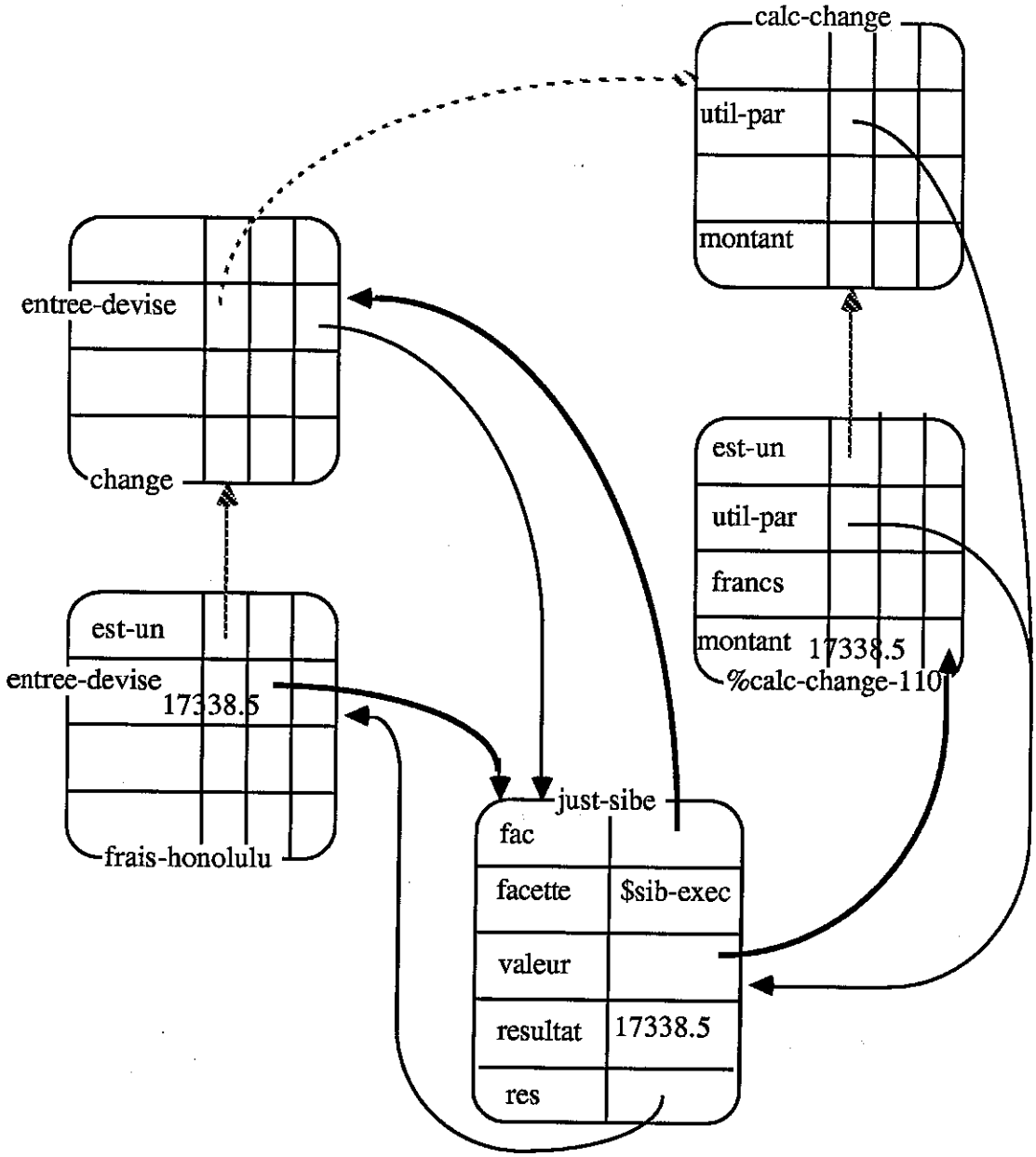
Le résultat quant à lui sera représenté par sa valeur, le nom de l'instance de la méthode compilée, et l'attribut dans lequel se trouve le résultat. On utilisera donc le type de cellule suivant:

```
(just-sib
   est-un        =          schema
   sorte-de      =          ( justification )
   facette       $un        facette                    ; facette utilisée
                 $domaine   ( $sib-exec $sib-filtre )
   class-meth $un           methode                    ; méthode
   obj-fac       $un        objet                      ; localisation de la méthode utilisée
   att-fac       $un        symbole
   inst-meth     $un        methode                    ; instance de la méthode
   obj-val       $un        objet                      ; localisation de la valeur inférée
   att-val       $un        symbole
   obj-res       $liste-de  objet                      ; localisation des attributs cibles
   att-res       $un        symbole
   resultat      $liste-de  objet )                    ; valeur inférée
```

Une factorisation semble possible car pour une même méthode, c'est toujours le même attribut qui rend la valeur.

◊ Exemple IV.5

L'attachement procédural calc-change, est utilisé pour trouver la valeur de l'attribut entree-devise de l'objet frais-honolulu. La classe représentant cette méthode a donné lieu à une instance %calc-change-110 qui a été valuée pour pouvoir faire la calcul. Le résultat de ce calcul a été rangé dans l'attribut montant de %calc-change-110. C'est lui qui va valuer l'attribut entree-devise de frais-honolulu. Cette dépendance est enregistrée ainsi:



◊

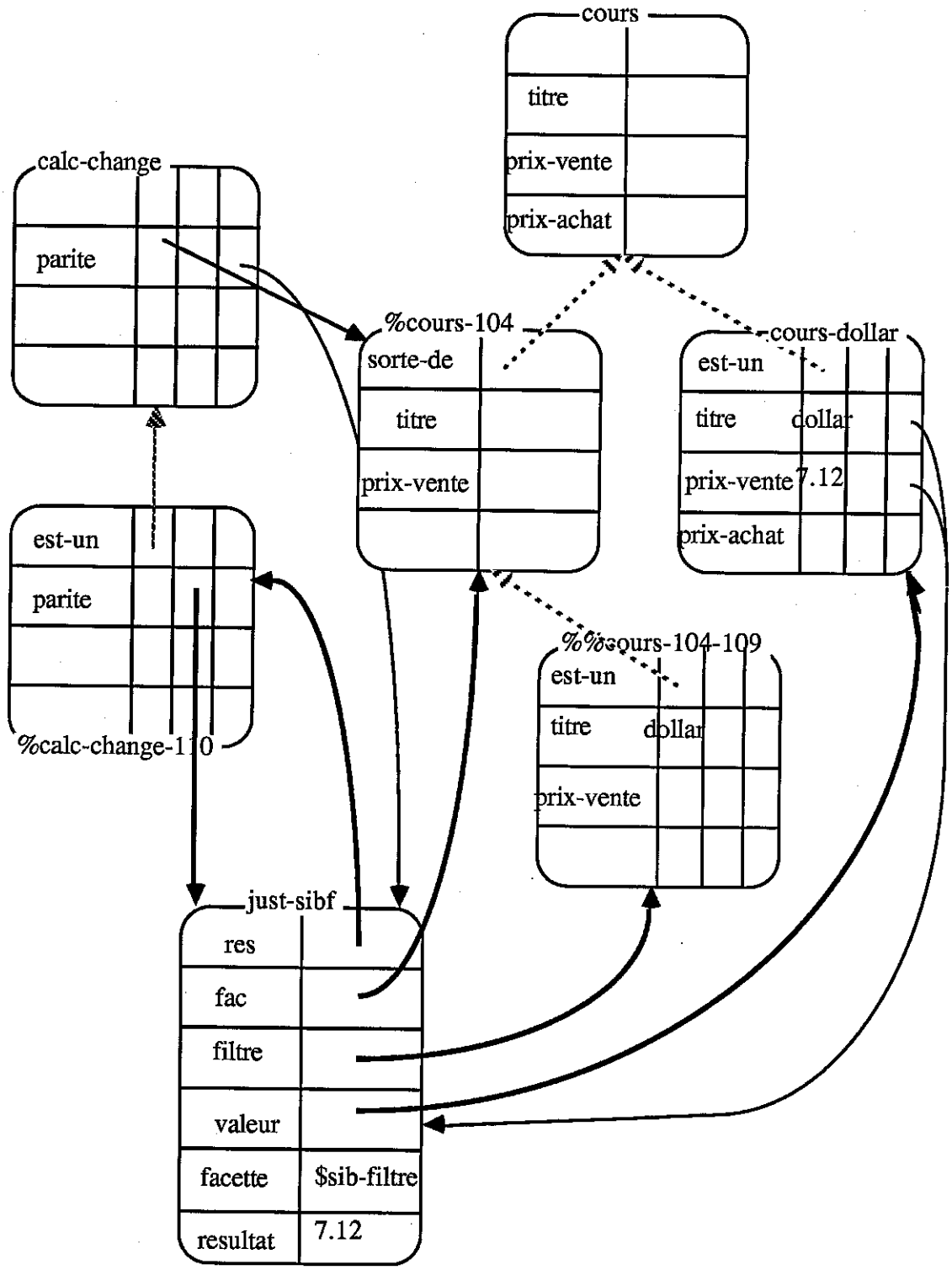
d) $sib-filtre.

Comme les méthodes, les filtres sont compilés en des classes qui seront instanciées pour représenter le filtrage /Bloch 86/. Ces instances seront donc maintenues de la même manière que celles des méthodes.

Pour pouvoir connaître un filtre, il faut connaître son adresse (<obj> <att>), le nom de la classe représentant le filtre compilé et le nom de l'instance du filtre généré au cours du filtrage. L'objet filtré sera celui dans lequel on a trouvé le résultat. Les différents attributs intervenant dans le filtre sont conservés dans le filtre compilé, il n'est donc pas besoin de les enregistrer dans les justifications. Les justifications des filtres auront en fait la même structure que celles des attachements procéduraux.

◊ Exemple IV.6

L'attachement procédural %just-sib-110 pour pouvoir effectuer son calcul a besoin du cours du dollar. Cette valeur, située dans l'attribut prix-vente de l'objet cours-dollar, est obtenue à l'aide d'un filtre, compilé en une spécialisation de la classe cours et instancié en %%cours-104-109. Ce filtre sera mis en correspondance avec l'instance cours-dollar pour rendre la valeur désirée et valuer l'attribut parite de l'objet %calc-change-110. Ce qui occasionera la justification suivante:



II. Procédures.

La propagation des invalidités est effectuée par un groupe de fonctions insérées dans le code de Shirka. Suivant le type de modification que subit la base ce n'est théoriquement pas les mêmes fonctions qui sont utilisées. Ces fonctions sont présentées ici suivant le type d'invalidité à propager.

1. Obtention d'une valeur.

A la réussite de l'inférence d'une valeur causée par la fonction *valeur?*, le système procède ainsi: Il cré une justification de l'inférence grâce à la fonction **justification**, celle-ci construit les objets justification et enregistre les dépendances avec la valeur résultat et la facette qui a occasionné l'inférece si le degrés de maintenance l'exige. La fonction **mem** exécute le "caching" en plaçant dans la case correspondante la valeur ramenée par la fonction *valeur?*.

2. Invalidation d'une valeur.

Les fonctions Shirka *aj-valeur*, *sup-valeur*, *mod-valeur* et *sup-inst* suppriment des valeurs de la base d'objets. Il est donc nécessaire de maintenir l'état des croyances du système en propageant dans toute la base l'invalidité d'une valeur.

Une première fonction, **out-mem**, permet de supprimer une valeur mémorisée de la base. Elle appelle la fonction **out-just**, qui permet de supprimer de positionner la ou les justifications supportantes dans une case *OUTmeth* de la valeur cachée.

Si la valeur détruite avait servi à calculer une autre valeur, il va falloir ôter de la base la mémorisation de la valeur inférée et tout cela récursivement. La fonction **out-dep** se charge donc d'invalider les valeurs inférées, de positionner les justifications soutenues par elles dans les cases *OUTarg* des attributs qu'elles ont servi à valuer, et de propager récursivement l'invalidité à toute la base.

3. Invalidation d'une méthode.

Les fonctions Shirka *mod-fac* et *sup-fac* permettent de modifier les méthodes de la base d'objets. Une telle modification doit entrainer l'invalidation de toutes les valeurs calculées à l'aide de cette méthode.

La fonction **out-meth** se charge d'invalider ces valeurs en supprimant leurs justifications. Elle propage récursivement à toute la base l'invalidité des valeurs inférées grâce à la fonction *out-mem*.

4. Modification du graphe d'héritage.

L'introduction d'une nouvelle facette à l'aide de la fonction Shirka *aj-fac* doit entrainer l'invalidation de toutes les valeurs inférées à l'aide de méthodes masquées par cette nouvelle facette (confère partie3.III.3). La fonction **out-graph** réalise cette invalidation, elle remonte dans le graphe d'héritage jusqu'à la classe *Objet* afin d'invalider tous les calculs effectués à l'aide de méthodes

calcul complet. D'autre part des optimisations sont possibles dans le cas du filtrage puisque les instances candidates sont préfiltrées à leur déclaration, et re-préfiltrées à leur modification.

d) Gestion des chaînes de références.

Dans l'hypothèse où un schéma contient des filtres imbriqués, les références dénotées par les variables peuvent être plus longues qu'un simple accès. Les justifications telles qu'elles existent ne permettent pas d'enregistrer des chaînes de référence. Il en est de même pour les valeurs provenant de filtres imbriqués.

Il faudrait enregistrer toutes les références.

e) Le TMS en tant que réflexe.

Il est envisageable d'implémenter le code du TMS comme des réflexes valides par défaut sur tous les objets. En effet, les programmes composant le système de maintenance de la vérité agissent dans des points bien précis:
- avant-ajout: suppression des valeurs cachées.
- après-modif, après-ajout, après-sup: invalidation des dépendances.
- si-succès: enregistrement des justifications.

Les programmes, pourront être des méthodes de Shirka, déclarées au sein d'*Objet*, constituant ainsi, à l'image des méthodes de visualisation, des objets manipulables par le programmeur Shirka.

f) Compilation et justification.

Actuellement, Shirka compile toutes les méthodes. Ces compilations comprennent la sémantique de celles-ci. Une fois implémenté le recalcul dans Shirka, le système n'utilise plus les compilations mais les justifications pour inférer les valeurs. Les justifications contiennent donc toutes les informations présentes dans le premier calcul. Il est imaginable au cours de la compilation de créer des classes de justifications qui seraient instanciées pour donner lieu aux justifications actuelles, qui deviendraient plus simples à créer.

Conclusion.

L'implémentation effective d'un TMS dans Shirka permet un accroissement des performances appréciable. Cet accroissement varie cependant suivant l'utilisation qui est faite de la base. En effet si celle-ci est très souvent modifiée le gain de temps n'est pas très considérable car le système de maintenance de la vérité doit propager toutes les invalidations. Par contre, le gain est évident quand la base subit beaucoup de requêtes.

La réutilisation systématique des structures offertes par Shirka, permet au code consacré au TMS d'être très compact.

L'évolution du gain apporté au cours de l'utilisation de la base de connaissance n'a pas été étudiée. On peut cependant s'attendre à un comportement de type de celui relevé pour RLL /Greiner& 80/ : « Tant que la base de connaissance change, RLL doit supprimer des valeurs cachées qui ne sont plus valides. Ce qui veut dire en terme d'efficacité qu'au cours de ses premières minutes d'utilisation RLL travaille très lentement, mais que le système accélère graduellement jusqu'à ce qu'on ne remarque plus les calculs. La lenteur revient temporairement après chaque modification majeure du système.» /Hayes-Roth& 83/.

2. Améliorations envisageables et perspectives.

a) Gestion des filtres.

Dans l'actuelle version de Shirka, les instances de filtres et d'attachements procéduraux n'ont pas la structure d'objets, ils sont optimisés et manipulés par des fonctions différentes de celles de Shirka. Ainsi, certaines étapes de valuation n'utilisent pas la fonction *valeur?* et par conséquent ne sont pas maintenues par le TMS.

Ceci devrait être résolu dans une prochaine version de Shirka où filtres et attachements procéduraux devraient être de "vrais" objets.

b) Optimisation au chargement du code.

Beaucoup de tests sont de trop. Il s'agit de ceux destinés à savoir si le TMS doit être activé ou non en interrogeant le drapeau. Ceux-ci peuvent être aisément éliminés par l'utilisation des macro #- et #+ de Le_Lisp qui ne charge que les partie de code utile. Cette facilité n'a pas été implémenté car on envisage pour la version 3 de Shirka d'affecter la maintenance non plus au système globalement, mais à chaque attribut individuelement.

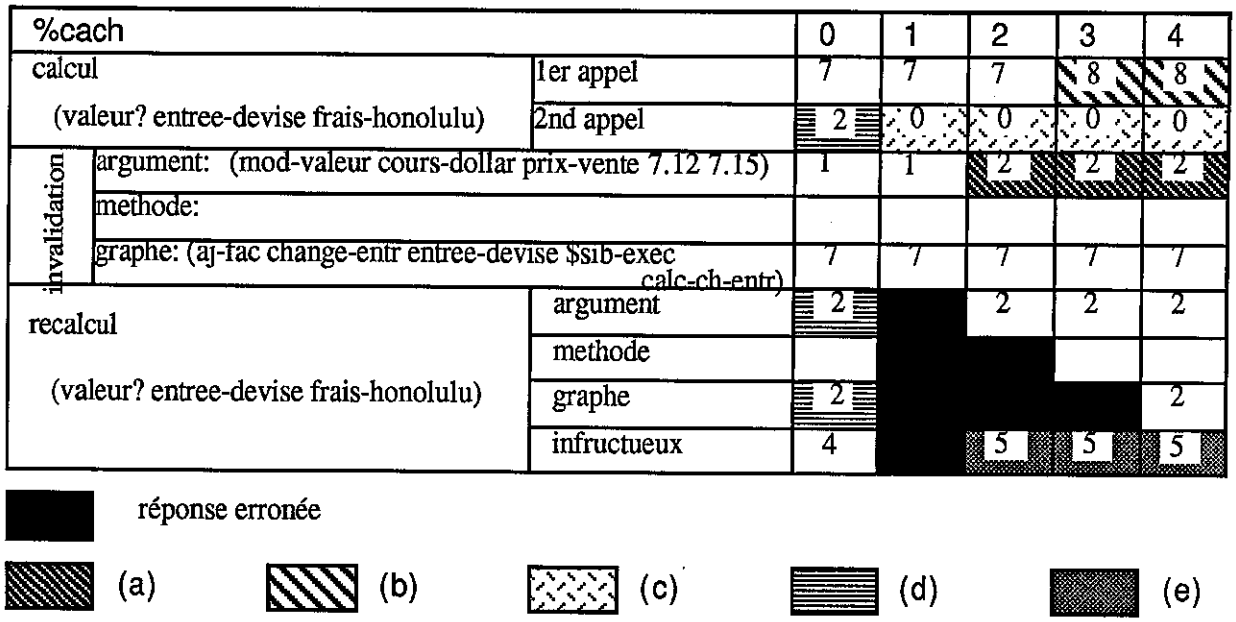
c) Necessité d'une case INConnu.

L'implémentation du TMS sur Shirka n'est pas satisfaisante. En effet, divers problèmes ne sont pas pris en compte:

- Si une valeur nécessaire à un calcul est inconnue, une seconde méthode moins prioritaire peut être activée et fournir un résultat. Cependant, ce résultat est à invalider si cette valeur est fournie ce que ne fait pas le TMS actuel.
- Dans le cas d'attribut multivalué, il en va de même pour un schéma nouvellement créé qui pourrait être mis en correspondance avec un filtre.
- Il en va de même dans le cas d'attribut multivalué dont la liste est obtenue élément par élément, si un nouvel élément de la base est candidat pour faire partie de la valeur.

La solution envisagée est d'introduire dans les vecteurs de justifications une case INConnu pour stocker les justifications de calculs avortés. Ainsi, avant de rendre la valeur cachée, le recalcul de cette justification sera tenté afin de s'assurer la validité de celle-ci. Cette solution a le grand désavantage de l'inefficacité, puisque la recherche infructueuse est la plus coûteuse en temps de calcul, cependant le temps de réponse devrait être toujours inférieur à celui du

représentative dans le sens où elle utilise toutes les facettes ($defaut, $var<-, $sib-filtre et $sib-exec). Cependant elle n'est pas représentative de la façon dont une base de connaissance est utilisée, en ce qui concerne la proportion de ces facettes.



Le système de maintenance de la vérité par lui-même n'améliore pas les performances d'une représentation de connaissance, c'est le "caching" qui diminue les temps de réponse. Son efficacité a été testée suivant les différents niveaux de maintenance, différentes constatations s'imposent:

a) Comme attendu, le temps utilisé dans les invalidations a augmenté. Cet accroissement est bien sûr dû à la propagation de l'invalidité.
b) Le temps de calcul a lui aussi augmenté légèrement. Ceci est dû à l'enregistrement des inférences.
c) Le "caching" permet un gain sensible au second appel d'une valeur.
d) Cependant le gain n'est pas aussi élevé que prévu du fait que Shirka conserve les instance de filtre et méthode qu'il réutilise, ainsi le calcul n'est pas refait. La conséquence de ceci est que l'avantage théorique du "caching" qui devrait être de 0 à 7 n'est que de 0 à 2 sur Shirka. Pour la même raison, le re-calcul implémenté avec le TMS se révèle ne pas améliorer les temps de réponse.
e) Le re-calcul infructueux prend plus de temps.

On a déja montré (confère troisième partie) la nécessité de ne pas utiliser certains niveaux de maintenance avec certaines utilisations de la base de connaissance (cases noires). Il ressort de l'étude des performances que le "caching" est intéressant si l'utilisateur redemande souvent les mêmes valeurs (c). Cet avantage n'est pas mis en valeur à cause des facilités de re-calcul implémentées dans Shirka. A première vue, le temps utilisé pour les invalidations (a) ne semble pas trop lourd pour le système (du moins par rapport à ce qu'en dit De Kleer /DeKleer 83/). Cependant les test ne permettent pas d'être formel sur ce point.

permettant de valuer des instances de la classe où est introduite la nouvelle facette.

5. Recalcul d'une valeur.

Il est très possible de construire un système de maintenance de la vérité sans modifier le calcul des inférences dans la fonction *valeur?*. Cela garantit la rapidité accrue apportée par le TMS. Un tel TMS n'a pas besoin de mémorisations aussi pousées que celles présentées, il suffit d'avoir

<mémorisation> ::= (INjust . dep)

au lieu de

<mémorisation> ::= (#[INjust OUTArg OUTmeth] . dep)

La version du TMS présentée ici réalise toutes les fonctionnalités proposées à la fin de la troisième partie. C'est-à-dire que *valeur?* a été modifiée dans le but d'accroître les performances du système même quand aucune valeur valide n'est cachée. Pour cela, les programmes de recherche d'une valeur de Shirka ont été modifiés afin d'y insérer un programme de re-calcul des inférences.

Le moteur d'inférence construit procède ainsi:

Une première fonction **moteur**, recherche d'abord l'ensemble des méthodes, applicables à la valeur recherchée. Ensuite, si la nature de l'attribut à valuer est $liste-de on appelle **inferer-tout**, sinon, on tente d'abord de recalculer des inférences précédentes invalidées pour défaut d'arguments à l'aide de la fonction **re-calculer**, et en cas d'échec on fait appel à la fonction **inferer**.

La fonction *inferer* tente d'appliquer dans l'ordre toutes les méthodes, et s'arrête au premier résultat trouvé. Au cas où elle a à appliquer une méthode dont une justification se trouve dans *OUTmeth*, elle tente d'abord de refaire le calcul de la justification en utilisant *re-calculer* sinon elle utilise l'algorithme classique de Shirka. La fonction *inferer-tout* tente d'abord de recalculer toutes les justifications de *OUTarg* grâce à *re-calculer* et en cas d'echec utilise toutes les méthodes qui lui sont fournies.

La fonction *re-calculer* est celle qui accélère réellement le calcul. Elle cherche d'abord toutes les valeurs nécessaires au calcul qui sont défaillantes, au cas où l'une d'elle fait toujours défaut elle abandonne le recalcul. Sinon, pour les facettes $var<- et $var-liste<- elle rend la valeur trouvée, pour $sib-filtre et $sib-exec elle relance les programmes de calcul de Shirka.

III. Performances et perspectives.

1. Performances en temps.

Les performances de Shirka muni du TMS ont été comparées à la version initiale de Shirka. Les résultats ont été obtenus sur LISA d'Apple, avec le timer de Le_Lisp qui n'a pas de plus grande précision que la seconde. Les tests ont été exécutés sur l'inférence présentée tout au long de ce rapport. Cette inférence est

Conclusion

A quoi sert la maintenance de la vérité dans une représentation centrée-objet ?

La maintenance de la vérité n'est envisageable dans une représentation des connaissances centrée-objet que liée au mécanisme de "caching". Celui-ci permet d'augmenter les performances du système dans des cas d'utilisation très précis (confère partie4.III). Deux sortes d'utilisations profitables peuvent être faites de ce TMS.

I. Facilités offertes par le système de gestion de bases de connaissance.

Ce premier type d'utilisation est profitable dans des cas qui ne se réclament pas ouvertement du raisonnement non-monotone mais plutôt de l'utilisation courante d'une base de connaissance. Il s'agit tout d'abord de la mise au point d'une base (confère partie3.III.1) qui grâce au TMS peut s'effectuer sans perdre les données inférées à chaque modification. Il s'agit ensuite de la rectification d'une erreur, qui peut être faite très longtemps après la saisie et la mise au point de la base. L'utilisation du "caching" a permis de mémoriser toutes les inférences faites, le TMS permet, à la rectification de l'erreur, de ne conserver que les inférences correctes.

II. Le raisonnement non-monotone.

L'utilisation non-monotone d'une base de connaissance se limite vraissemblablement au raisonnement hypothètique comme le montrent tous les exemples rencontrés: établissement d'emploi du temps par relaxation des assertions valides par défaut /Doyle 79b/, prospective politique /McDermott 83/, évolution des marchés financiers /Williams 84/. Cette limitation est assez vaste puisque le raisonnement hypothétique intègre divers types de raisonnements.

Tout d'abord le raisonnement par défaut /Reiter 80/ qui est à la base de nos raisonnements les plus quotidiens. Il semble bien illusoire de tenter de copier le raisonnement humain sans l'utiliser.

En second lieu, la prospective permettant de connaître les conséquences d'une hypothèse dans le temps. Le TMS au moment de l'invalidation d'une donnée du temps t invalidera toutes les données qu'elle a permis d'inférer aux temps t+x. Cette voie a été profondément explorée et récement exposée dans /Dean& 87/.

Cependant le raisonnement hypothétique est beaucoup plus pratique à manipuler avec un TMS à contextes qui n'a pas à invalider les conséquences des hypothèses abandonnées.

III. L'explication des inférences.

Comme cela avait été relevé dans /Doyle 79b, 79c/ et /Chehire 86/, les systèmes de maintenance de la vérité permettent une implémentation simple de l'explication des raisonnements.

Les structures ajoutées à Shirka pour l'implémentation du TMS ont permis la construction d'un mécanisme d'explication des inférences. En effet, les

justifications mémorisent toutes les caractéristiques des inférences effectuées par Shirka. Il est donc aisé d'utiliser ces justifications pour exposer les inférences à l'utilisateur. Ce module d'explication (2 pages de Lisp contenant essentiellement des procédures d'affichage) ne nécessite pas de mémoire supplémentaire puisqu'il utilise les enregistrements fait pour le système de maintenance de la vérité.

Une fonction permet donc, à la demande de l'utilisateur, d'afficher les méthodes et données utilisées pour inférer une valeur. L'explication négative est aussi permise dans une certaine mesure, il s'agit d'afficher les méthodes ou données modifiées qui ont permis d'invalider des inférences antérieures.

Bibliographie

/Bloch 86/
Didier BLOCH
Filtrage dans une base de connaissance centrée-objet, optimisation et techniques de compilation
Rapport de DEA, INP Grenoble, FR, 1986

/Brown 85/
Allen BROWN
Modal logic propositionnal semantics for reason maintenance systems
Actes IJCAI85-I (p178), 1985

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

/Charniak& 85/
Eugene CHARNIAK, Drew MAC DERMOTT
Introduction to artificial intelligence
Addisson-Wesley, Reading, MA, 1985

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

/Dahl& 66/
Ole-Johan DAHL, Kristen NYGAARD
Simula - An Algol-based simulation language
Communication of the ACM 9-IX (pp671-678), 1966

/Dean& 87/
Thomas DEAN, Drew MAC DERMOTT
Temporal data base management
Artificial intelligence 32-I (pp1-55), 1987

/DeKleer& 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

/DeKleer& 82/
Johan DE KLEER, Jon DOYLE
Dependencies and assumptions
dans: BARR, FEIGENBAUM
The handbook of artificial intelligence
William Kaufmann, Los Altos, CA, (pp72-76),1982

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

/DeKleer 86a/
Johan DE KLEER
An assumption-based TMS
Artificial Intelligence 28-II (pp127-162), 1986

/DeKleer 86b/
Johan DE KLEER
Extending the ATMS
Artificial Intelligence 28-II (pp163-196), 1986

/DeKleer 86c/
Johan DE KLEER
Problem solving with the ATMS
Artificial Intelligence 28-II (pp197-224), 1986

/Doyle 79a/
Jon DOYLE
Truth maintenance system for problem solving
Actes IJCAI-79-I (p247), 1979

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

/Doyle 79c/
Jon DOYLE
A glimpse of truth maintenance
dans: WINSTON, BROWN
Artificial intelligence: an MIT perspective
The MIT press, Cambridge, MA, (pp121-135), 1979

/Doyle 83/
Jon DOYLE
The ins and outs of reason maintenance
Actes IJCAI-83-I (pp349-351), 1983

/Goodwin 82/
James GOODWIN
An improved algorithm for non-monotonic dependency net update
Rapport de recherche MAT-R-82-23, Linköping university, SW, 1982

/Greiner& 80/
Russell GREINER, Douglas LENAT
A representation language language
Actes AAAI-80 (pp165-169), 1980

/Hayes-Roth& 83/
Frederick HAYES-ROTH, Donald WATERMAN, Douglas LENAT
Building expert systems
Addison-Wesley, Reading, MA, 1983

/Hein 83/
Uwe HEIN
PAUL: A programming language 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

/McDermott& 80/
Drew MAC DERMOTT, Jon DOYLE
Non-monotonic logic I
Artificial Intelligence 13-I (pp41-72), 1980

/McDermott 82/
Drew MAC DERMOTT
Non-monotonic logic II: non-monotonic modal theories
Journal of the ACM 29-I (pp33-57), 1982

/McDermott 83/
Drew MAC DERMOTT
Contexts and data dependencies: a synthesis
IEEE Transactions on pattern analysis and machine intelligence 5-III (pp237-246), 1983

/Minsky 75/
Marvin MINSKY
A framework for representing knowledge
dans WINSTON
The psychology of computer vision
McGraw-Hill, New York, NY, (pp211-277), 1975

/Reiter 80/
Raymond REITER
On reasoning by default
Artificial Intelligence 13-I (pp81-132), 1980

/Rechenmann 85/
François RECHENMANN
SHIRKA : mécanismes d'inférence sur une base de connaissances centrée-objet
Actes RFIA-85-II (pp1243-1254), 1985

/Shrobe 79/
Howard SHROBE
Dependency-directed reasoning in the analysis of programs that modify complex data structures
Actes IJCAI-79-II (pp829-835), 1979

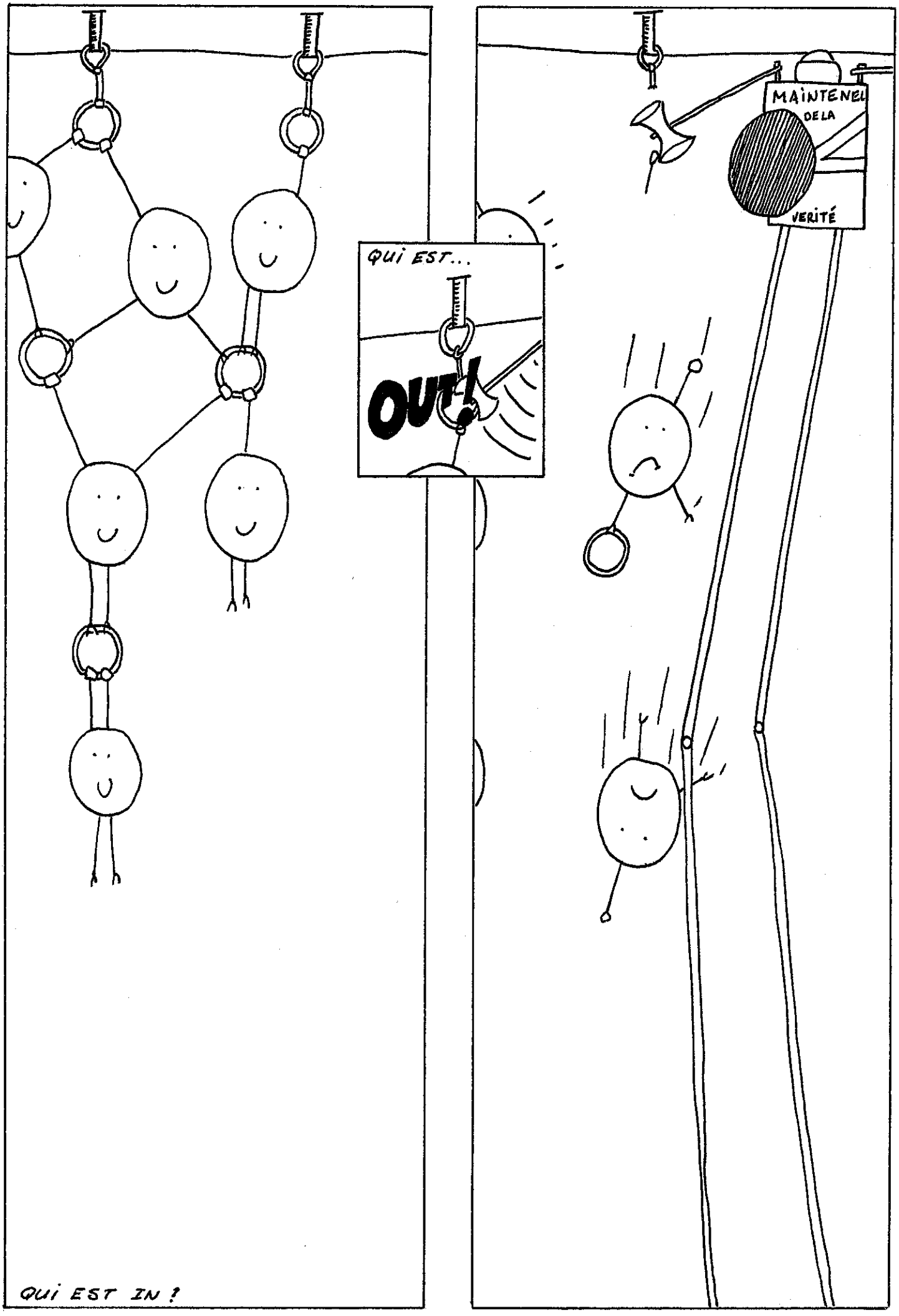
/Stallman& 77/
Richard STALLMAN, Gerald SUSSMAN
Forward reasoning and dependancy-directed backtracking in a system for computer-aided circuit analysis
Artificial Intelligence 9-II (pp135-196), 1977

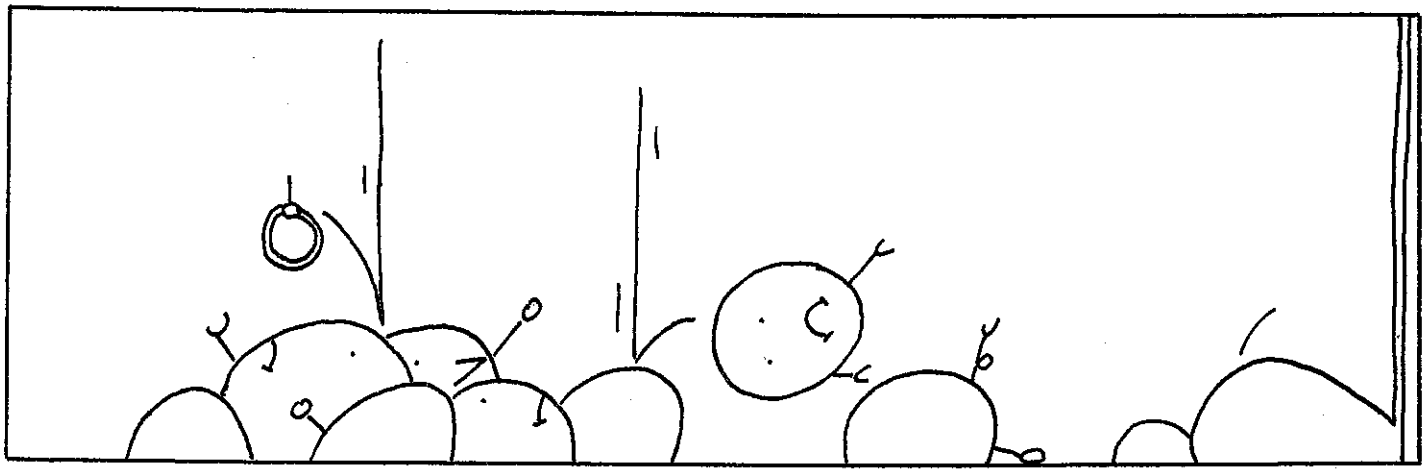
/Thompson 79/
Alan THOMPSON
Network truth maintenance system for deduction and modelling
Actes IJCAI-79-II (pp877-879), 1979

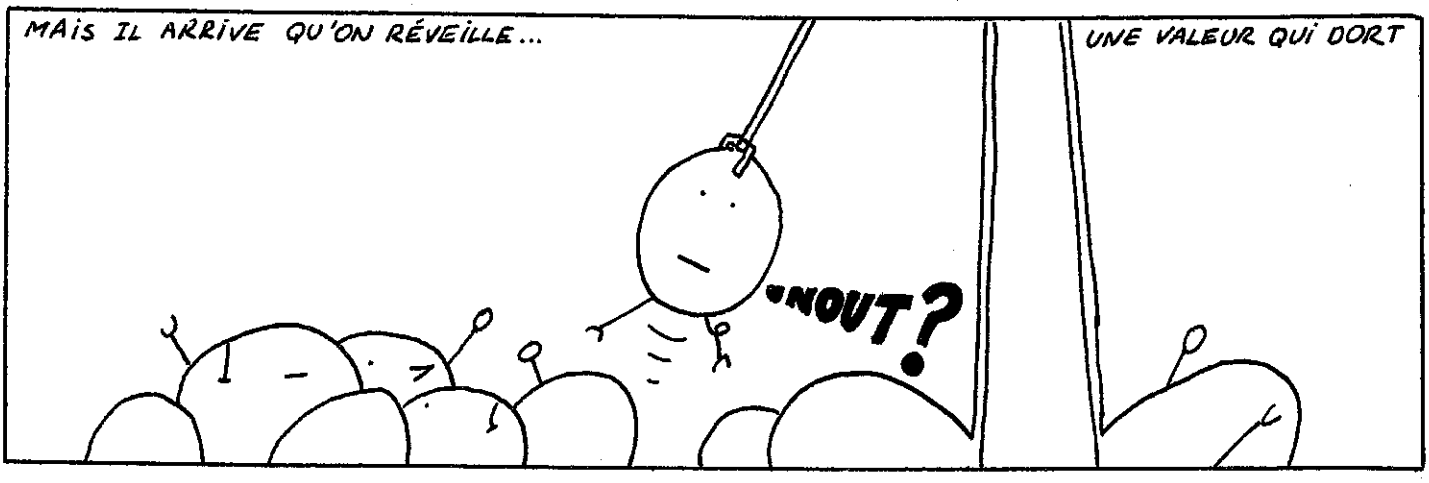
/Turner 84/
Raymond TURNER
Logiques pour l'intelligence artificielle
Masson, Paris, FR, 1984

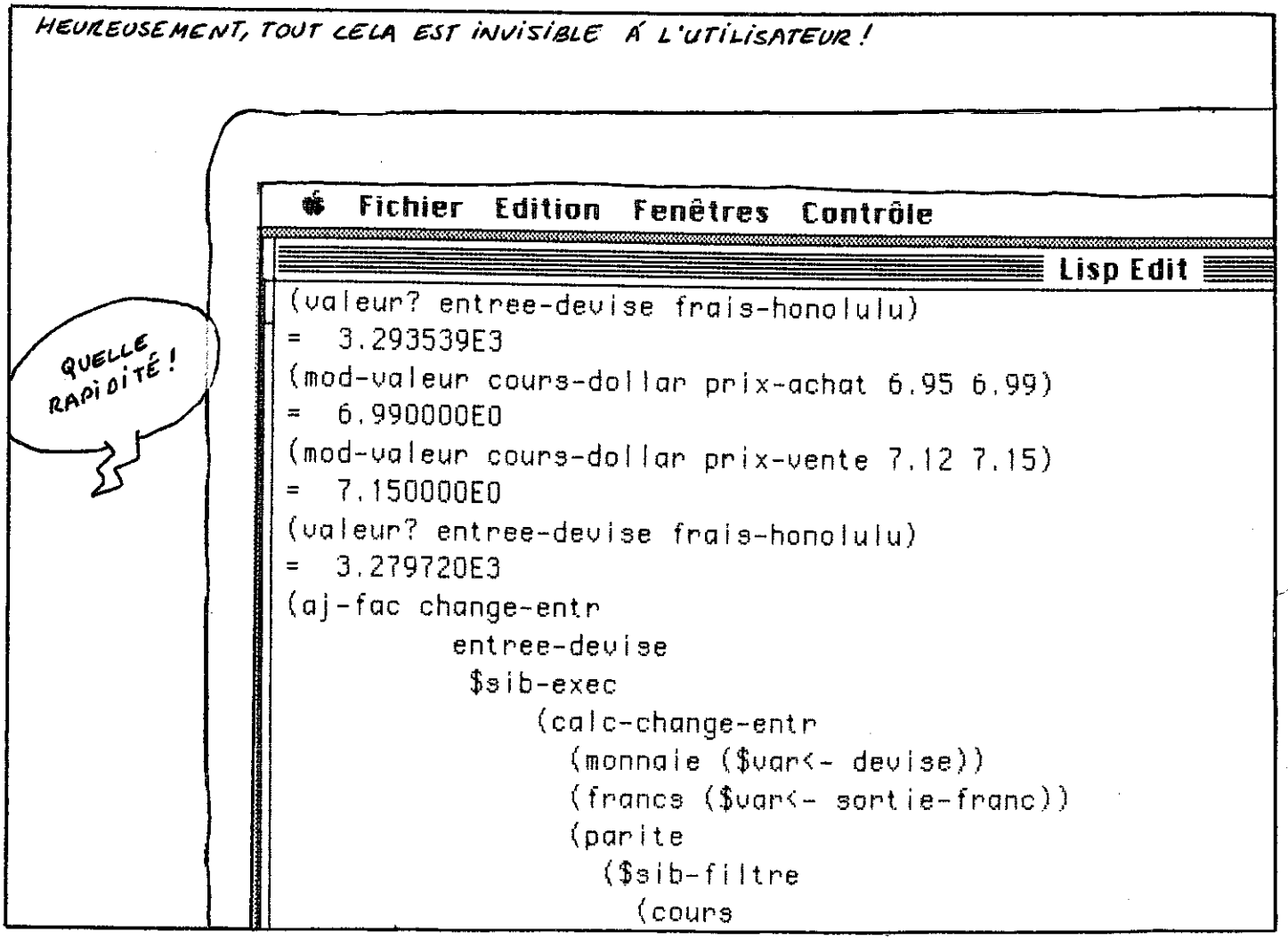
/Williams 84/
Chuck WILLIAMS
ART: The advanced reasoning tool, conceptual overview
Inference Corp., Los Angeles, CA, 1984

/Wrigth& 83/
J. WRIGHT, Mark FOX
SRL/1.5 user manual
Carnegie-Mellon university, Pittsburg, PA, 1983


QUI EST...
OUT!
MAINTENEL
DE LA
VERITÉ
QUI EST IN ?




MAIS IL ARRIVE QU'ON RÉVEILLE...
UNE VALEUR QUI DORT
•NOUT?


HEUREUSEMENT, TOUT CELA EST INVISIBLE À L'UTILISATEUR !
QUELLE RAPIDITÉ !
Fichier Edition Fenêtres Contrôle
Lisp Edit
(valeur? entree-devise frais-honolulu)
= 3.293539E3
(mod-valeur cours-dollar prix-achat 6.95 6.99)
= 6.990000E0
(mod-valeur cours-dollar prix-vente 7.12 7.15)
= 7.150000E0
(valeur? entree-devise frais-honolulu)
= 3.279720E3
(aj-fac change-entr
entree-devise
$sib-exec
(calc-change-entr
(monnaie ($var<- devise))
(francs ($var<- sortie-franc))
(parite
($sib-filtre
(cours