Vers des spécifications de la connaissance exécutables

Boris Charreton, Jean-Louis Ermine

Vers des spécifications de la connaissance exécutables

B. Charreton, J-L Ermine

Version française non publiée

Article paru en partie dans :
Sous le titre : " From knowledge specification to executable specification, Knowledge Engineering and Modelling Languages
KEML'96, Paris, 15-16 January 1996
Sous le titre: Operationalization of a graphical knowledge representation language
EKAW'96, Nottingham, UK, May 1996, Poster Session, pp 2- 12

Vers des spécifications de la connaissance exécutables

Boris Charreton*, Jean-Louis Ermine *

** CEA/DIST*
Section Méthodes et Technologies de l'Information
Groupe Gestion des Connaissances
Centre d'Etudes de Saclay
91191 Gif sur Yvette CEDEX
tel: 69 08 61 14 / Fax: 69 08 26 69 /e-mail: bcharret@instn.saclay.cea.fr

RESUME: MOISE est une méthodologie de gestion de la connaissance qui comporte une étape de spécification de la connaissance, laquelle distingue la connaissance statique de la connaissance dynamique. Cette étape met en œuvre un langage de représentation de la connaissance graphique (LRC) qui combine un langage dédié à la représentation de la connaissance statique (réseaux sémantiques) et un autre à celle de la connaissance dynamique (langage de tâches). Les connaissances décrites dans le LRC ou spécifications de la connaissance (graphiques) constituent le document de référence, véritable point source de la description de la connaissance. Différentes transformations automatiques de ce document initial, sous d'autres formes, ont été envisagées. Sous une forme textuelle de type rapport (Winword), ce qui a déjà été réalisé. Sous une forme hypertextuelle permettant une navigation dans les spécifications graphiques, ce qui est en cours d'étude. Ou encore sous une forme exécutable permettant la validation de modèles ou même la génération de prototypes de systèmes experts, ce qui est en cours de réalisation. Cette dernière transformation ou opérationnalisation de modèles formels est l'étude présentée dans cet article.

MOTS CLE: langage de représentation des connaissances, ingénierie des connaissances, langage orienté objet, connaissance statique, connaissance dynamique, tâches, opérationnalisation.

1. Introduction

MOISE (Méthode Organisée pour l'Ingénierie des Systèmes Experts) est une méthodologie de gestion des connaissances élaborée par J-L. Ermine [Ermi93]. Elle comporte une étape dite de spécification de la connaissance dont l'objectif est de construire un modèle formalisé de l'ensemble de la connaissance contenue dans un problème et sa résolution. C'est une étape de modélisation au même titre que la construction du modèle conceptuel de la méthodologie KADS [Wiel92]. On distingue le niveau connaissance du niveau symbolique en accord avec les principes établis par Newel [Newe82].

A l'inverse des méthodologies KOD [Voge89] et KADS [Breu94] qui ont été développées pour une application industrielle, MOISE a été créée en 1989 pour une application en recherche. Elle a, toutefois, été appliquée avec succès dans plus de 30 projets industriels et dans un éventail très large de domaines tels que le diagnostic technique [Ermi91] ou les systèmes de centrales électriques [Ben95]. Divers outils logiciels (non commercialisés) ont été développés pour supporter la méthode. MOISE tend aujourd'hui à être intégrée dans une problématique plus générale de gestion des connaissances qui englobe et dépasse l'approche cognitive [Brun94].

L'étape de spécification des connaissances est l'étape de la méthode la plus fréquemment utilisée. Elle sépare la connaissance *statique* qui décrit la connaissance du domaine de la connaissance *dynamique* qui décrit la stratégie de l'expert en identifiant les tâches qui manipulent les objets du domaine. Elle met en œuvre un Langage de Représentation de la Connaissance graphique (LRC) qui combine deux sous langages graphiques. Le premier est un langage graphique de réseaux sémantiques qui permet de décrire les objets du domaine

structurés en *concepts* et qui est ainsi associé à la connaissance statique. Le second, appelé langage de tâches, est associé à la connaissance dynamique.

Les spécifications graphiques de la connaissance décrites dans le LRC, constituent le document de référence, véritable point source de la description de la connaissance. Différentes transformations automatiques de ce document initial, vers d'autres formats, sont à l'étude ou ont été réalisées. Une 1ère transformation vers une forme textuelle de type rapport (Winword) a déjà été réalisée. Une 2è vers une forme hypertextuelle [Chai95] permettant une navigation dans les spécifications graphiques est en cours d'étude. Une 3è vers une forme exécutable permettant la validation de modèles ou même la génération de prototypes de systèmes experts, est en cours de réalisation. Cette dernière transformation ou opérationnalisation de modèles formels est l'étude présentée dans cet article.

Dans la 1ère partie le LRC graphique est présenté. Puis le modèle objet, ayant permis le passage des spécifications formelles vers des spécifications exécutables, est détaillée dans la 2nde partie. Tout au long du document, nous nous appuierons sur le problème Sisyphus II pour illustrer nos explications. Pour mémoire, le problème Sisyphus II [Yost92] est un problème de configuration d'ascenseur déduit du code source du système expert de configuration d'ascenseur VT [Marc88]. L'objectif de Sisyphus II est de permettre la comparaison entre différentes approches d'ingénierie des connaissances sur un exemple non trivial.

2. Langage de représentation des connaissances

2.1 Langage graphique

L'approche MOISE couvre toutes les étapes méthodologiques depuis l'acquisition des connaissances jusqu'à la conception et l'implémentation d'un système. Les processus d'acquisition ou techniques d'interview d'experts ne sont pas discutés dans cet article qui se concentre uniquement sur le LRC.

Tout d'abord, nous pouvons nous interroger sur les qualités et vertus qu'il est souhaitable qu'un LRC possède. Généralement le LRC est un outil proposé à un expert au moyen duquel il va décrire son savoir faire d'une manière formalisée. L'expert qui ne connaît pas initialement le langage formel proposé, doit se l'approprier rapidement sans que cela devienne un nouvel obstacle. Le langage doit donc être simple, intuitif et expressif. D'un autre côté, il doit être suffisamment complet et précis pour s'adapter à tout type de domaine. Il concrétise un compromis entre simplicité et richesse de description et propose des primitives de modélisation nécessaires et suffisantes. Le LRC que nous présentons, combinaison d'un langage de réseaux sémantiques et d'un langage de tâches, a été défini en tenant compte, autant que possible, de ces considérations.

2.1.1 Réseaux sémantiques

Les réseaux sémantiques, utilisés pour représenter la connaissance statique, sont un langage bien connu puisque les logiciens du moyen âge s'en servaient. Ils sont intéressants pour représenter et structurer la connaissance du domaine et plus simplement pour « rendre claires nos idées » [Peir78]. Ils sont plus facilement compréhensibles que la logique des prédicats du 1er ordre [Sowa91] [Bläs89]. Différents réseaux sémantiques existent, proposant chacun sa propre terminologie ou représentation, et les nôtres sont présentés dans ce qui suit. Un réseau est un ensemble de noeuds interconnectés par des arcs. Un noeud représente un concept et un noeud terminal peut être un ensemble de valeurs (Réel, Entier, Booléen, Caractère,...). On distingue 5 arcs orientés différents (Figure 1). 3 arcs sont utilisés pour structurer le domaine d'une façon générale et 2 autres pour appliquer cette structuration à un cas spécifique.

Les arcs de structuration sont :

- **att** qui lie un concept et un concept **att**ribut qui le compose,
- **val** qui lie 2 concepts où le 1er prend ses **val**eurs dans le 2nd,
- **spec** qui lie 2 concepts où le 1er est **spéc**ialisé par le 2nd ; il permet la représentation hiérarchique d'une taxonomie.

Les arcs d'instanciation sont :

- **elt** qui lie un **él**émen**t** et le concept dans lequel il est classé,
- = qui donne la valeur d'un attribut pour un élément d'un concept ou parfois qui fixe la valeur d'un attribut pour le concept tout entier.

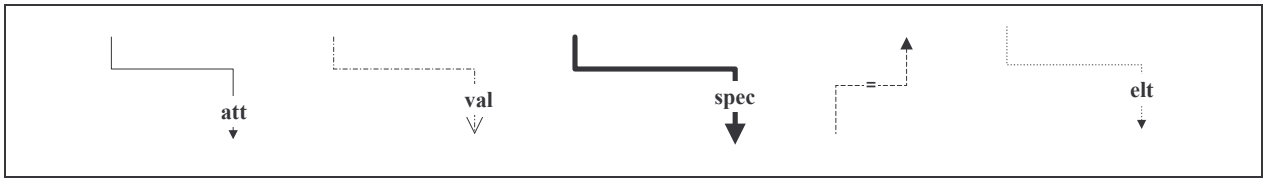

Figure 1: Les 5 arcs d'un réseau

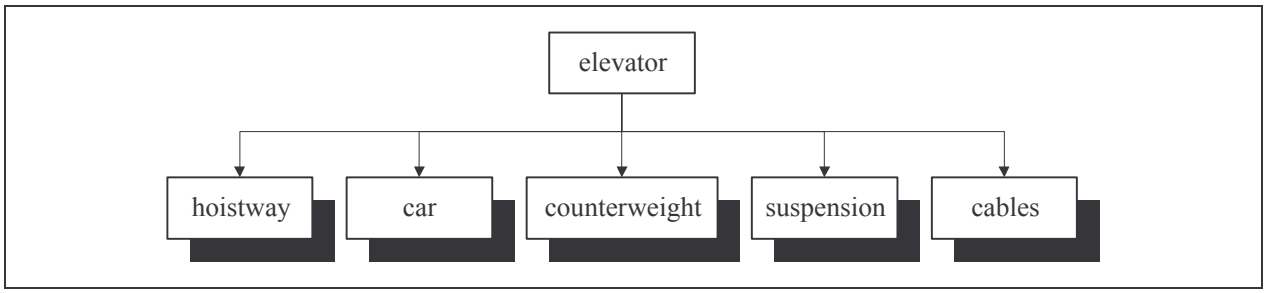

Figure 2: Le réseau d'un ascenseur

D'après l'étude du rapport Sisyphus II, un ascenseur (elevator system) peut être (partiellement) décrit par la Figure 2 d'une façon expressive. On en déduit qu'un ascenseur est composé de 5 sous-systèmes : la cage d'ascenseur (hoistway), la voiture (car), le contrepoids (counterweight), un système de suspension et d'un ensemble de câbles. 250 concepts ont été répertoriés pour Sisyphus II.

Nous allons « zoomer » sur le système d'amortisseur (buffer, Figure 3) qui est un sous-système de la voiture (car) et du contrepoids (counterweight). Un modèle d'amortisseur (model) prend ses valeurs dans l'ensemble défini par extension {OH-1, OM-14} tandis que la hauteur d'un amortisseur (height) prend ses valeurs dans l'ensemble des réels. Un amortisseur se spécialise en 2 types d'amortisseur (OM-14 buffer, OH-1 buffer) qui sont décrits par un certain type de modèle (OM-14, resp. OH-1), un certaine distance de compression (stroke = 14, resp. stroke = 8.25), une certaine hauteur (height = 38.5, height = 28.75), une certaine charge limite (load limit) minimum et maximum (minimum = 2900 et maximum = 11000 pour les 2).

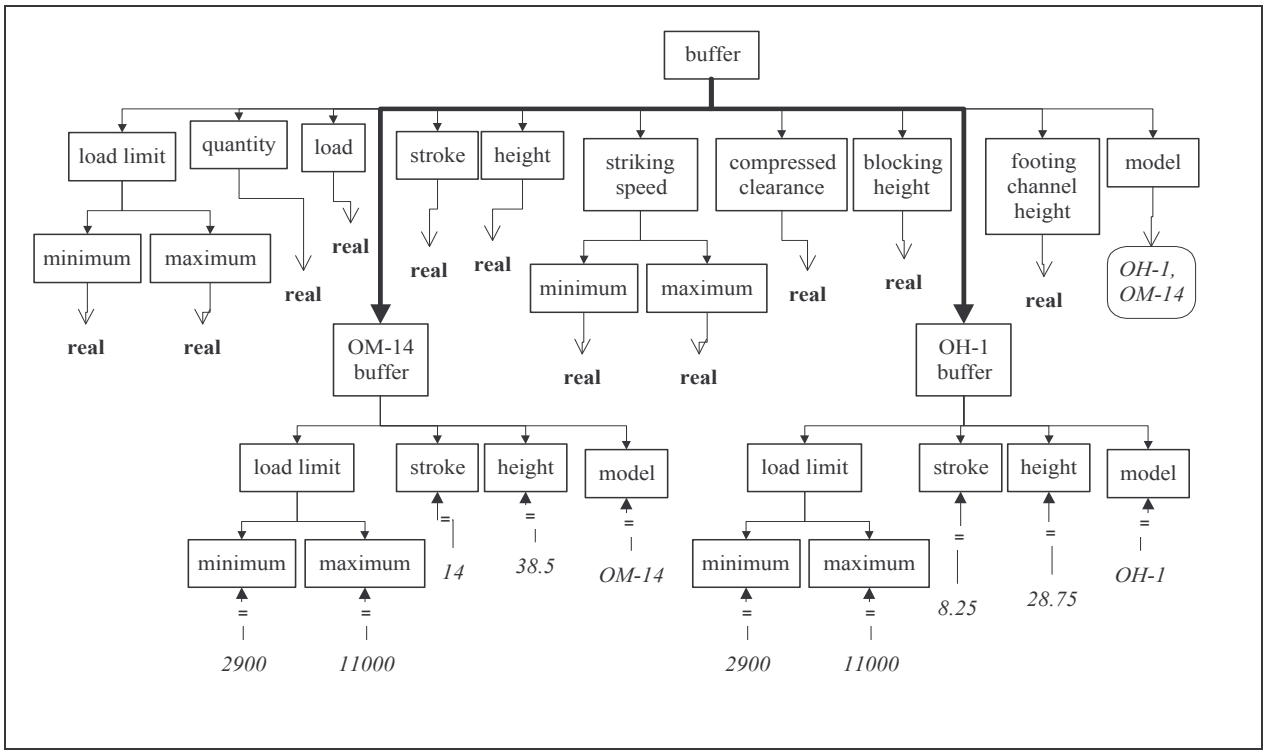

Figure 3: Le réseau d'un amortisseur

2.1.2 Langage de tâches

Le langage de tâches permet de représenter graphiquement la connaissance dynamique et a été défini en tenant compte d'études en psychologie cognitive appliquées à l'ergonomie [Scap89]. On distingue 2 types de tâches: les tâches spécifiques et les tâches génériques. Une tâche spécifique est décrite par un nom et des paramètres d'entrée/sortie, et encapsule des sous-tâches contrôlées par une tâche générique.

La Figure 4 indique que la tâche 'commencer à configurer un ascenseur' (start configuring elevator) permet de configurer en grande partie l'ascenseur (paramètre de sortie : 'mainly configured elevator') à partir des informations client nécessaires (paramètre d'entrée : 'customer required information'). Pour cela, elle doit effectuer en séquence (cf contrôle séquentiel Figure 5) les sous tâches de configuration de la cage d'ascenseur ('configure hoistway'), puis de la voiture ('start configuring car'), puis du câble de contrôle ('configure cable control'), puis du contrepoids ('start configuring counterweight'), puis du système de suspension ('start configuring suspension') et enfin des cables ('start configuring cables').

Lorsque nous qualifions une tâche de générique, nous entendons qu'elle contient une méthode de résolution de problème adaptable à différents problèmes. Ainsi, le résolveur de problème Propose&Revise introduit dans la méthode VITAL [Mott94] constitue une tâche générique puisqu'elle peut résoudre divers problèmes et notamment calculer des valeurs devant vérifier une contrainte dans Sisyphus II.

Il existe des contrôles génériques de base (Figure 5) prédéfinis et prêts à l'emploi et pour lesquels le terme de 'tâche générique' semble trop fort et celui de 'tâche de contrôle' mieux approprié car trop rudimentaires. Cependant ces tâches génériques sont des briques de base permettant la définition de tâches dont le caractère générique (applicabilité/réutilisabilité) est plus évident, ce que nous illustrons dans ce qui suit.

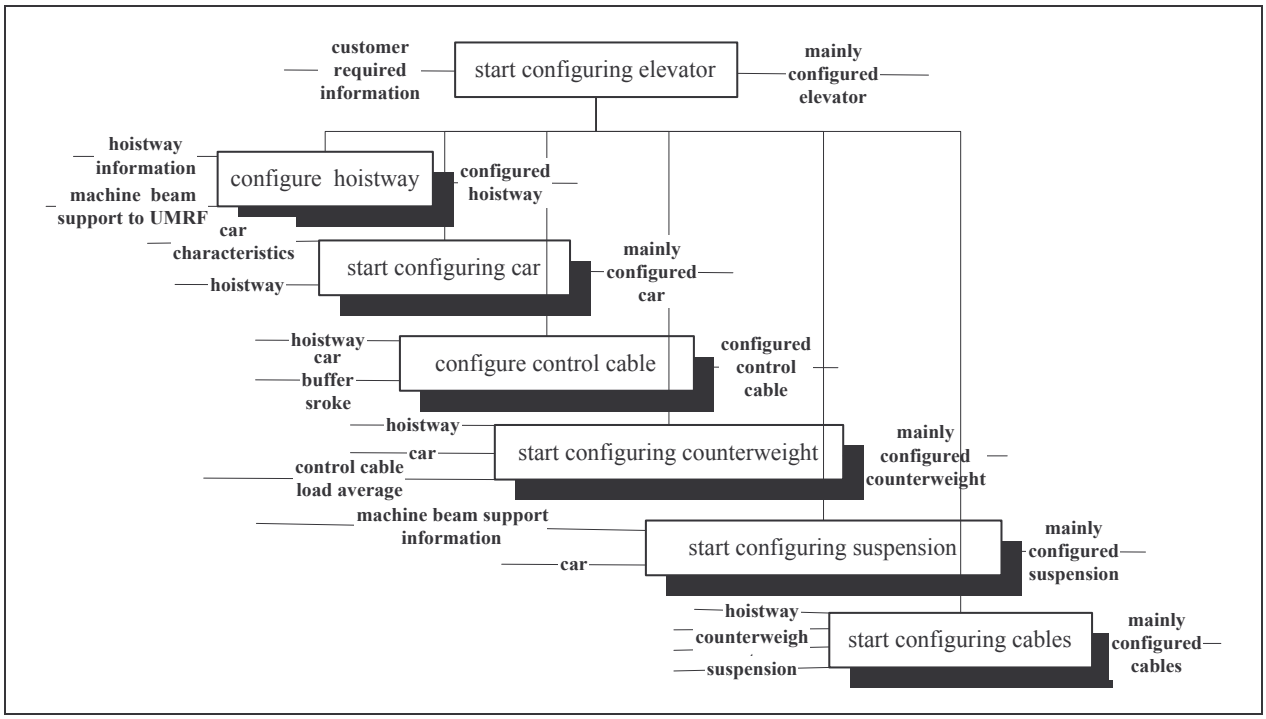

Figure 4 : Tâche spécifique 'commencer à configurer un ascenseur'

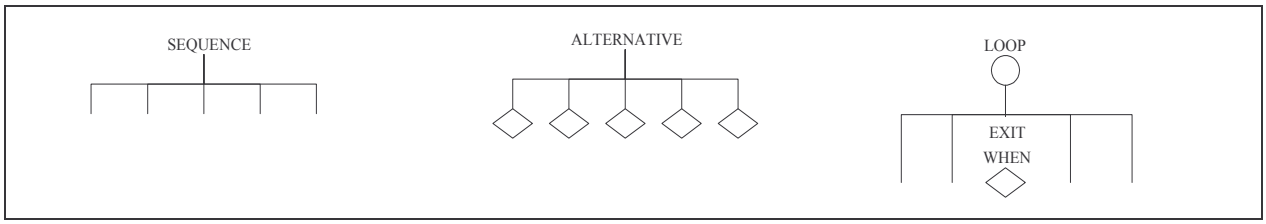

Figure 5 : 3 tâches génériques de base

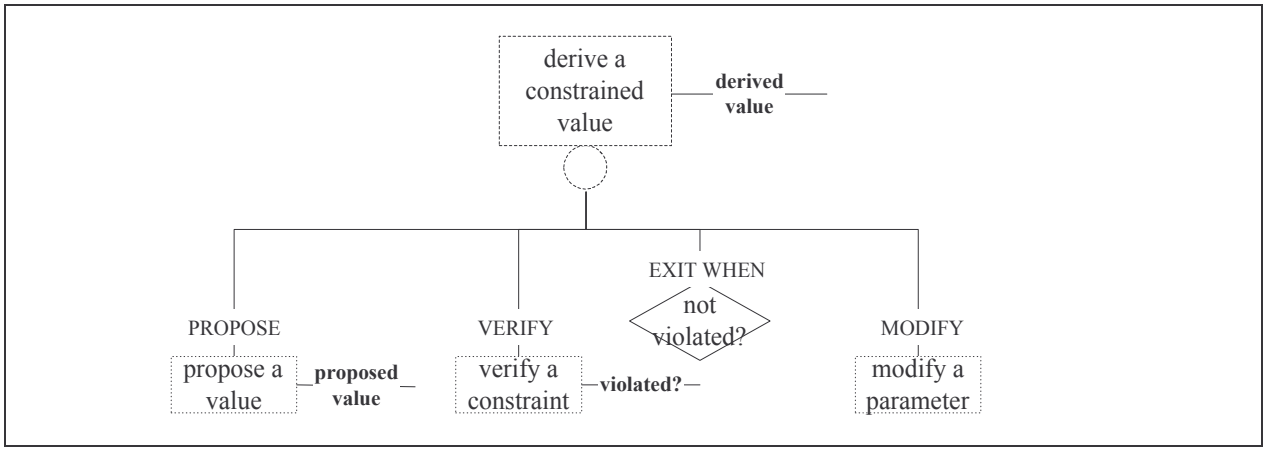

Figure 6 : Tâche générique 'calculer une valeur contrainte'

De manière à rester proche du mode de raisonnement d'un expert d'un domaine, il est possible de définir des tâches génériques personnalisées qui lui sont propres. Nous pouvons ainsi définir une tâche générique dédiée au calcul d'une valeur contrainte en analysant le document Sisyphus II. Concernant le calcul de valeur contrainte , G.R. Yost écrit : « Pour calculer les valeurs des variables de configuration [...], appliquez les contraintes dès que

possible. [...] La 1ère étape lors de l'application d'une contrainte est de voir si elle est violée ou non. Si elle ne l'est pas, plus d'autre application n'est nécessaire pour cette contrainte et vous pouvez poursuivre le calcul des valeurs d'autres variables de configuration et appliquer d'autres contraintes. Si la contrainte est violée, alors, vous devez immédiatement essayer de trouver les modifications de configuration qui remédieront à cette violation. » Ceci peut être reformulé de la manière suivante : Pour calculer une valeur contrainte, on propose d'abord une valeur, puis l'on regarde si la contrainte est violée ou non. Si elle ne l'est pas, arrêtons là. Sinon modifions les bons paramètres de configuration puis recommençons à nouveau les différentes étapes (proposer, vérifier, modifier). On peut ainsi définir la tâche générique 'calculer une valeur contrainte' ('derive a constrained value', Figure 6) à partir d'un contrôle de base de type 'boucler...sortir quand' ('loop...exit when'). Il est possible dans un 2è temps de réutiliser cette tâche générique pour définir le contrôle d'une tâche spécifique telle que 'calculer la charge de l'amortisseur de la voiture' ('derive a car buffer load value', Figure 7).

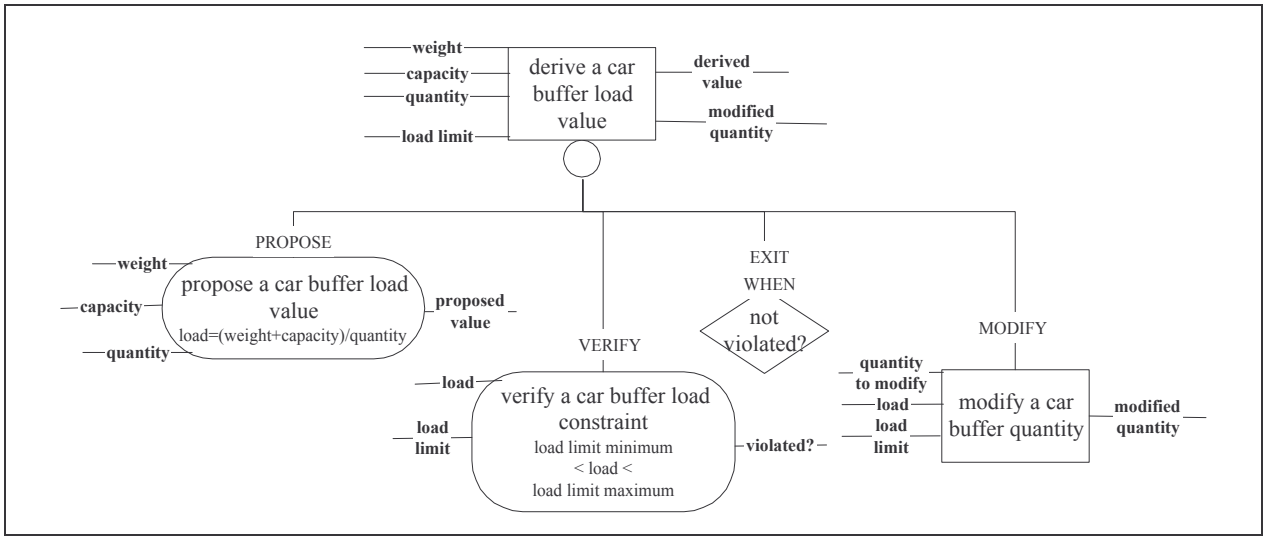

Figure 7 : Tâche spécifique 'calculer la charge de l'amortisseur de la voiture'

2.2 Langage formel

Les spécifications graphiques de la connaissance sont ensuite converties sous une forme textuelle (Winword) dans un langage dit formel. Les spécifications obtenues dites formelles, conserve une bonne lisibilité pour l'utilisateur et lui permet d'enrichir les spécifications avec des éléments qui ne figurent pas dans la représentation graphique tels que des contraintes sémantiques supplémentaires ou tels que le flot de données. De plus, la forme textuelle de ces spécifications formelles les rend interprétables par un analyseur syntaxique et permettra par la suite d'établir une passerelle vers le langage de programmation.

2.2.1 Langage formel statique

Voici le résultat de la transformation du réseau sémantique de la Figure 3 dans le langage formel statique :

net
buffer **att** model **val** {OH-1, OM-14}**;**
buffer **att** quantity **val** real**;**
buffer **att** load **val** real**;**
buffer **att** stroke **val** real**;**
buffer **att** height **val** real**;**
buffer **att** footing channel height **val** real**;**
buffer **att** blocking height **val** real**;**

buffer **att** compressed clearance **val** real**;**
buffer **att** striking speed **att** minimum **val** real**;**
buffer **att** striking speed **att** maximum **val** real**;**
buffer **att** load limit **att** minimum **val** real**;**
buffer **att** load limit **att** maximum **val** real**;**
net
OM-1 buffer **spec** buffer **att** model = OH-1**;**
OM-1 buffer **spec** buffer **att** height = 28.75**;**
OM-1 buffer **spec** buffer **att** stroke = 8.25**;**
OM-1 buffer **spec** buffer **att** load limit **att** minimum = 2900**;**
OM-1 buffer **spec** buffer **att** load limit **att** minimum = 11000**;**
net
OM-14 buffer **spec** buffer **att** model = OM-14**;**
OM-14 buffer **spec** buffer **att** height = 38.5**;**
OM-14 buffer **spec** buffer **att** stroke = 14**;**
OM-14 buffer **spec** buffer **att** load limit **att** minimum = 2900**;**
OM-14 buffer **spec** buffer **att** load limit **att** minimum = 11000**;**

2.2.2 Langage formel dynamique

Voici la description de la tâche générique ‘calculer une valeur contrainte’ (‘derive a constrained value, Figure 6) et celle de la tâche spécifique ‘calculer la charge d’un amortisseur de voiture’ (‘derive a car buffer load value’, Figure 7) dans le langage formel dynamique.

generic task
derive a constrained value
input
output
derived value **elt** real**,**
task
propose : propose a value**,**
verify : verify a constraint**,**
modify : modify a parameter**,**
control
loop...exit when
exit condition <= not violated?
task list <= (propose, verify, exit when, modify)

La description formelle de la tâche ‘derive a car buffer load value’ inclut le flot de données qui n’est pas représenté graphiquement pour des raisons de clarté. Par exemple, la ligne (a) exprime le flot de données du paramètre de sortie ‘proposed value’ de la sous tâche ‘propose a car buffer load value’ vers le paramètre de sortie ‘derived parameter’ de la tâche principale ‘derive a car buffer load value’. La ligne (b) décrit le flot de données du paramètre de sortie de la sous tâche ‘propose a car buffer load value’ vers le paramètre d’entrée ‘load’ de la sous tâche ‘verify a car buffer load constraint’.

task
derive a car buffer load value
input
weight **elt** real
capacity **elt** real
quantity **elt** real
load limit **elt** load limit
output
derived parameter **elt** real
modified quantity **elt** real
control
derive a constrained value
propose <=

```
propose a car buffer load value
        weight <=
        derive a car buffer load value.weight
        capacity <=
        derive a car buffer load value.capacity
        quantity <=
        derive a car buffer load value.quantity,
        modify a car buffer quantity.modified quantity
        proposed value =>
        derive a car buffer load value.derived parameter                (a)
verify <=
verify a car buffer load constraint
        load <=
        propose a car buffer load value.proposed value                  (b)
        load limit <=
        derive a car buffer load value.load limit
        violated? =>
modify <=
modify a car buffer quantity
        quantity to modify <=
        derive a car buffer load value.quantity,
        modify a car buffer quantity.modified quantity
        load <=
        propose a car buffer load value.proposed value
        load limit <=
        derive a car buffer load value.load limit
        modified quantity =>
        derive a car buffer load value.modified quantity
```

2.2.3 Nouvelles contraintes

Il est possible de rajouter de nouvelles contraintes, non représentées graphiquement, au niveau des spécifications formelles. Par exemple le document Sisyphus II mentionne la contrainte suivante : « La voiture et le contrepoids sont tous deux constitués de plusieurs amortisseurs (buffer) dont les modèles sont les mêmes à la fois pour la voiture et le contrepoids, dans un système d'ascenseur. » Ceci s'écrit :

elevator system **att** car assembly **att** buffer **att** model = elevator system **att** counterweight assembly **att** buffer **att** model

« La charge limite minimum est inférieure à la charge limite maximum », ce qui s'écrit:

load limit **att** minimum < load limit **att** maximum

3. Langage de programmation objet

Pour obtenir des spécifications exécutables, nous avons choisi le langage de programmation de haut niveau SPIRAL [Lorr92] [Lorr93] qui est un langage de programmation logique utilisé pour le développement de systèmes à base de connaissance. Le modèle de conception d'accueil des spécifications statiques et dynamiques est présenté dans cette partie. Ce modèle objet découle de la définition d'une sémantique dénotationnelle associée au LRC [Char96].

3.1 Objets statiques

Le modèle de classe prédéfini de SPIRAL est suffisamment complet pour décrire la plupart des objets statiques. Un attribut d'une classe possède les facettes :

- *domaine (domain)* qui permet de définir un ensemble en extension,
- *valeur (value)* qui permet de fixer la valeur d'un attribut

- *attribut (attribute)* qui indique si un champ est monovalué ou multivalué.

Un *concept* est décrit par une *classe*. Toute classe est créée par la méta-classe 'classe' ('class') et est sous classe de la classe 'objet' ('object'). Le concept d'un amortisseur (buffer, Figure 3) est décrit ci-dessous :

```
ask(class,create,buffer
      ,(superclasses,object)
      ,(fields
            ,(model,identifier,(domain,OH-1,OM-14),(attribute,monovalue))
            ,(quantity,real,( attribute,monovalue))
            ,(load,real,( attribute,monovalue))
            ,(stroke,real,( attribute,monovalue))
            ,(height,real,( attribute,monovalue))
            ,(footing channel height,real,( attribute,monovalue))
            ,(blocking height,real,( attribute,monovalue))
            ,(compressed clearance,real,( attribute,monovalue))
            ,(striking speed, striking speed,( attribute,monovalue))))
            ,(load limit, load limit,( attribute,monovalue)))));

ask(class, create, load limit
      ,(superclasses,object)
      ,(fields
            ,(minimum,real,( attribute,monovalue))
            ,(maximum,real,( attribute,monovalue))));

ask(class,create,OH-1 buffer
      ,(superclasses, buffer)
      ,(fields
            ,(model,identifier,(value,OH-1))
            ,(height,real,(value,28.75))
            ,(stroke,real,(value,8.25))
            ,( load limit,load limit,(value,OH-1 load limit))));

ask(load limit,create, OH-1 load limit
            ,(minimum,2900)
            ,(maximum,11000));
```

3.2 Objets dynamiques

Dans le modèle objet de SPIRAL sont définis 3 niveaux d'abstraction: le *niveau méta-classe*, le niveau *classe* et le niveau *instance*. Une méta-classe est une classe qui crée d'autres classes (les instances d'une méta-classe sont des classes) et une classe simple crée des instances. La notion de tâche est définie aux 3 niveaux d'abstraction, rebaptisés niveaux *méta-tâche* (*méta-classe*), *abstraction de tâche* (*classe*) et *application de tâche* (*instance*).

méta-classe	méta-tâche
classe	abstraction de tâche
instance	application de tâche

3.2.1 Création de tâches

La notion de tâche (abstraction de tâche) spécifique introduite en 2.1.2 est vue comme une classe d'objets particulière possédant des paramètres d'entrée/sortie (E/S) par opposition à celle de tâche générique vue comme une classe possédant en plus des paramètres tâche (E/S/T). On définit les 2 méta-tâches *méta-tâche* (ou *méta-tâche spécifique*) et *méta-tâche générique* (Figure 8) qui permettent de créer des tâches *spécifiques* respectivement *génériques*. On pourra par exemple (cf 3.2.2) demander à *méta-tâche* de créer la tâche spécifique 'derive a car buffer load value' avec les paramètres *weight*, *capacity*, *quantity* en entrée et *proposed value* en sortie et demander à *méta-tâche générique* de créer la tâche

générique ‘derive a constrained value’ avec les paramètres *derived value* en sortie et *propose, modify, verify* en tâche.

3.2.2 Hiérarchie de tâches

Puisque les tâches sont représentées par des classes, nous allons utiliser le mécanisme d’héritage propre à tout langage objet et donc à SPIRAL pour définir une hiérarchie de tâches. Une tâche est caractérisée par des *paramètres* et par un *contrôle* qui définit son mode opératoire. Une tâche avec des paramètres d’E/S est dite *spécifique* et avec des paramètres d’E/S/T est dite *générique*. Une tâche dont le contrôle est un programme prédéfini (expression) est dite *terminale* et dont le contrôle est défini par une tâche générique est dite *composée*. La racine hiérarchique des tâches comporte ainsi les tâches abstraites *tâche* (ou *tâche spécifique*, racine de toutes les tâches), *tâche terminale* (ou *tâche spécifique terminale*), *tâche composée* (ou *tâche spécifique composée*), et les tâches *tâche générique*, *tâche générique terminale*, *tâche générique composée*. Cette possibilité de définir une taxonomie de tâches, non utilisée par le LRC mais en revanche offerte par le langage objet, possède un véritable intérêt conceptuel [Rech85][Will94].

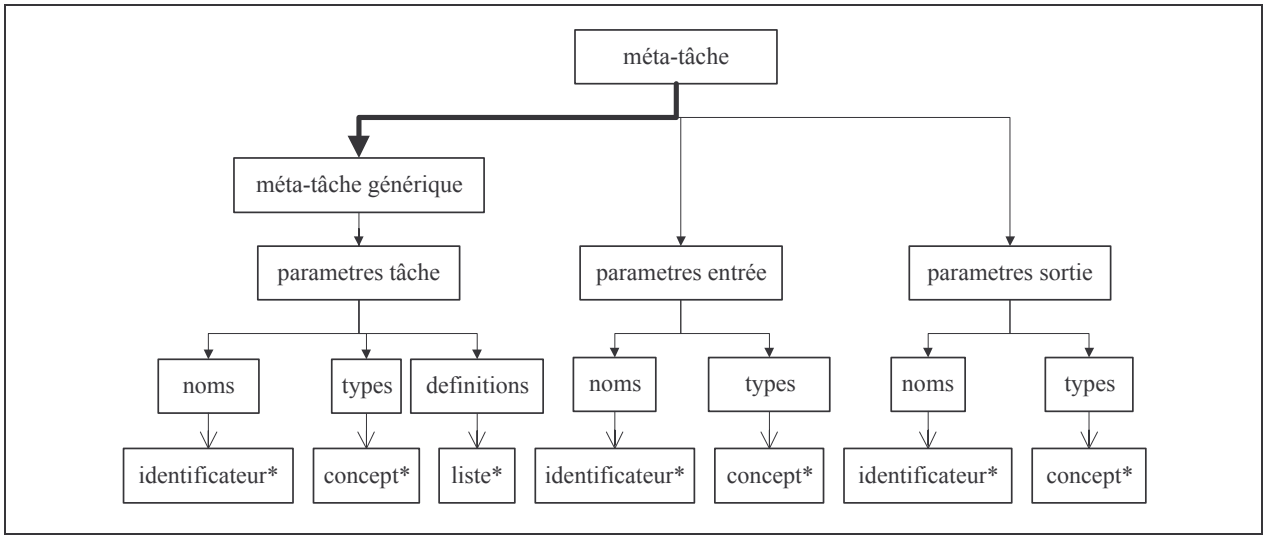

Figure 8: Réseau des méta-tâches

Les définitions en SPIRAL de la tâche générique ‘derive a constrained value’ et de la tâche spécifique ‘derive a car buffer load value’ s’écrivent :

```
ask(generic meta-task,task create,derive a constrained value
                ,(superclasses,composed generic task)
                ,(output parameter names,derived value)
                ,(output parameter types,real)
                ,(task parameter names,control,propose,verify,modify)
                ,(task parameter types,generic task,propose a value,verify a constraint,modify a parameter)
                ,(task parameter definitions
                ,(loop...exit when
                        ,(exit condition, violated COND)
                        ,(task list,(propose,verify,exit when,modify))
                )
                ,()
                ,()
                ,()));
ask(meta-task,task create,derive a car buffer load value
        ,(superclasses,composed task)
        ,(input parameter names,weight,capacity,quantity,load limit)
```

```
        ,(input parameter types,real, real, real,load limit)
        ,(output parameter names,derived value,modified quantity)
        ,(output parameter types,real, real)
        ,(task parameter names,control)
        ,(task parameter types,generic task)
        ,(task parameter definitions
    ,(derive a constrained value
        ,(propose
            ,(propose a car buffer load value
                ,(weight,(derive a car buffer load value,weight))
                ,(capacity,(derive a car buffer load value,capacity))
                ,(quantity
                    ,(derive a car buffer load value,quantity)
                    ,(modify a car buffer quantity,modified quantity))
                ,(proposed value,(derive a car buffer load value,derived value))))
        ,(verify
            ,(verify a car buffer load constraint
                ,(load,(propose a car buffer load value,proposed value))
                ,(load limit,(derive a car buffer load value,load limit))))
        ,(modify
            ,(modify a car buffer quantity
                ,(quantity to modify
                    ,(derive a car buffer load value,quantity)
                    ,(modify a car buffer quantity,modified quantity))
                ,(load,(propose a car buffer load value,proposed value))
                 ,(load limit,(derive a car buffer load value,load limit))
                ,(modified quantity,(derive a car buffer load value,modified quantity))))
    )
        ));
```

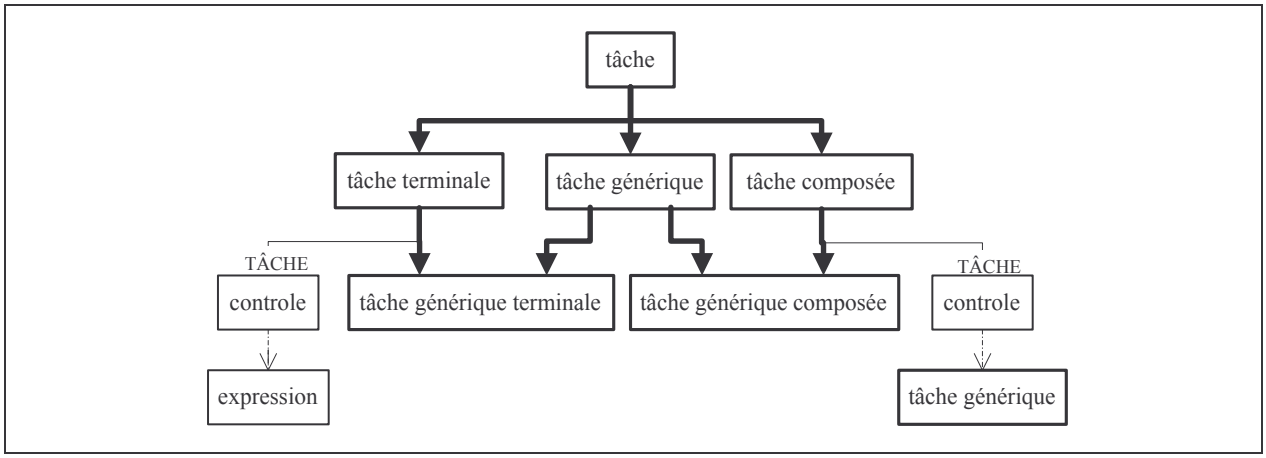

Figure 9 : Racine hiérarchique des tâches

3.2.3 Exécution de tâches

Toute tâche terminale ou composée possède une méthode (au sens objet) d'*exécution* qui permettra à leurs applications de s'exécuter. Le déroulement de la méthode d'*exécution* entraîne la valuation des paramètres de sortie à partir des paramètres d'entrée. En outre, l'exécution d'une application composée ('derive a car buffer load value#0') va créer puis exécuter une application générique ('loop...exit when#0') qui va contrôler la création et l'exécution de sous-applications ('propose a car buffer load value#0...n', 'verify a car buffer load constraint#0...n, ...) et ainsi de suite.

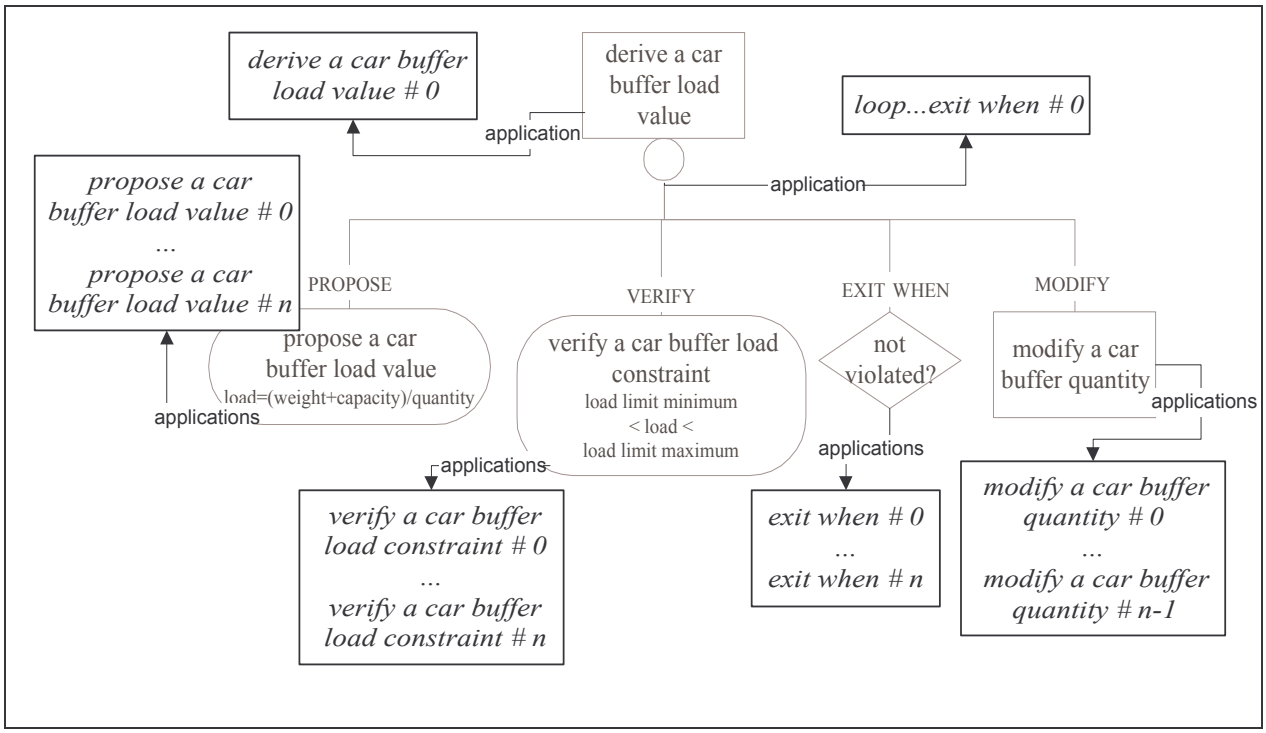

Figure 10: Applications résultant d'une exécution

4. Conclusion

L'objectif de l'étape de spécification de la méthode MOISE n'est pas seulement de résoudre un problème de configuration d'ascenseur, mais il est également de donner une représentation de la résolution du problème qui soit claire pour l'utilisateur et qui lui permette de comprendre ce que fait le système. Ce qui implique avant tout un gros travail de modélisation.

L'opérationnalisation de spécifications des connaissances (graphiques) correspond à une 1ère transformation vers des spécifications formelles (textuelles) puis à une 2nde vers des spécifications exécutables (SPIRAL). Les 2 outils, permettant ces transformations, restent à implémenter. Actuellement, l'exécution des tâches est supportée par un browser qui permet de lancer des exécutions puis de naviguer au sein des appels (applications de tâche) résultants via une interface. Il serait intéressant, en s'inspirant d'études d'ergonomie [Scap89], d'étudier la génération d'interfaces permettant à l'utilisateur d'interagir au cours de l'exécution.

Références

[Ben95] Z. Benzian, B. Bergeon, J-L Ermine, C.M. Falinower: "Knowledge Engineering for a Power Plant Operations Support System". LSS95 7th IFAC/IFORS/IMACS Symposium on Large Scale Systems: Theory and Applications, London (England), july 1995.

[Bläs89] K.H. Bläsius, U. Hedstück, C.-R Rollinger (eds): *Sorts and Types in Artificial Intelligence*. Lecture Notes in Artificial Intelligence, Springer-Verlag ed, 1989.

[Breu94] J. Breuker, W. Van de Velde (eds): *Comon KADS library for expertise modelling (reusable problem solving components)*. IOS Press ed, Amsterdam (Netherlands), 1994.

[Brun94] E. Brunet, J-L. Ermine: "Problématique de la gestion des connaissances des organisations". Ingénierie des systèmes d'information, AFCET-Hermes, vol 2, No 3, p 263-291, 1994.

[Chai95] M. Chaillot: " Conception d'une application hypermédia en gestion des connaissances pour la conduite des centrales nucléaires en mode dégradé.". Rapport interne SMTI/GGC/MC/95/026-CCC, 1995.

[Char96] M. Charreton, M. Ermine: " From knowledge specification to executable specification." KEML'96, jan 15-16, 1996.

[Ermi91] J-L. Ermine: "Un système à base de connaissance pour le diagnostique technique et la recherche documentaire". Convention IA91, Hermes ed , Paris (france), p 241-256, 1991.

[Ermi93] J-L. Ermine: *Génie logiciel et génie cognitif pour les systèmes à base de connaissances*. *Aspects méthodologiques*, Lavoisier ed, Tec et Doc, 1993.

[Lorr92] J-P. Lorre, J-M Evrard, E. Dorlet: *Bases de connaissances pour la modélisation d'installations industrielles,* Journées d'Avignon, Avignon, France, juin 1992.

[Lorr93] J-P. Lorre: *SPIRAL: un langage objet logique,* Journées CEA-CUIC 93 (theme: l'approche objet: vers une nouvelle technologie de developpement), Grenoble, France, juin 1993.

[Marc88] E. Motta, K. O'Hara, N. Shadbolt, A. Stutt, Z. Zdrahal: *A VITAL Solution to the Sisyphus II elevator Design Problem,* proceedings of the 8th Knowledge Acquisition for Knowledge-based System Workshop, vol 3, Banff, Canada, february 1994.

[Mott94] Marcus, Stout, Mc dermott: *VT: An expert elevator designer that uses knowledge-based backtracking,* AI magazine, spring 1988.

[Newe82] A. Newel: *The knowledge level,* AI magazine, vol 18, 1982.

[Peir78] C.S. Peirce: *How to make our ideas clear,*Popular Science Monthly, pp 286-302, january 1878.

[Rech85] F. Rechenmann: *SHIRKA:Mécanismes d'inférence sur une base de connaissances centrée-objet,* 5e congrès Reconnaissance de Formes et Intelligence Artificielle, pp 1243-1254, 1985.

[Scap89] D.L. Scapin, C. Pierret-Golbreich: "Towards a method for task description: MAD". Proceedings of Work with display units conference, Montreal (Canada), 1989.

[Sowa91] F. Sowa: *Principles of semantic networks*: J.F. Sowa ed, Morgan Kaufmann , 1991.

[Voge89] C. Vogel: *Génie cognitif*. Masson ed, 19893.

[Wiel92] B.J. Wielinga, A.Th. Schreiber, J.A Breuker: "KADS: a modelling approach to knowledge engineering". Proceedings of Knowledge Acquisition, vol 4, No 1, p 5-53, march 1992.

[Will94] J. Willamovski: *Modélisation de tâches pour la résolution coopérative de problèmes,* These, LIFIA, 1994.

[Yost92] Gregg R. Yost: *Configuring elevator systems,* Technical report, Digital equipment Co, Malboro, Massachussetts, 1992.