FORMALISATION, PROTOTYPAGE ET VALIDATION DES CONNAISSANCES[1]

Par Gilbert Paquette et Lucien Roy

SOMMAIRE

[1] Ce texte est extrait en majeure partie du volume de G.Paquette et L. Roy, « Systèmes à base de connaissances, Télé-université et Beauchemin, pp.245-294

Introduction

Dans un texte précédent, nous avons passé en revue un ensemble de techniques permettant d'acquérir les connaissances d'un domaine en vue de les implanter dans une base de règles. Nous passons à l'étape de la réalisation d'une application informatique pour rendre utilisables, par des usagers, les connaissances acquises. Cela veut dire traduire les connaissances en règles et les rendre opérationnelles à l'aide d'un **environnement de développement**, soit un langage de programmation, une coquille de système expert ou un atelier d'ingénierie des connaissances. La démarche présentée dans ce chapitre est en continuité avec celle introduite précédemment. Elle consiste à procéder par raffinement progressif pour mettre au point un prototype. On améliore les versions successives du **prototype** en raffinant notre modèle des connaissances et en corrigeant les erreurs détectées lors des tests. Par prototype, on entend un ensemble opérationnel constitué de la base de règles, de l'interface usager et, s'il y a lieu, de la programmation rendue nécessaire par les traitements spécifiques de l'**application** désirée.

Le choix judicieux et l'exploitation efficace d'un environnement de développement sont des conditions essentielles au succès de l'implantation d'un SBR. C'est pourquoi, nous ferons d'abord un survol de ce type de logiciel sur le marché. Nous nous attarderons à leurs fonctionnalités communes et à leurs particularités pour en montrer la diversité et en identifier les outils susceptibles de combler des besoins spécifiques. Cela nous permettra de constater que les environnements de développement de SBR sont assez faciles à maitriser si l'on connait leurs fonctions et les concepts qu'ils permettent d'implanter. Puis, nous donnerons quelques conseils sur le prototypage, c'est-à-dire sur la construction rapide d'une application de type SBR.

L'étape de la **validation** des connaissances d'un SBR fait intervenir les experts, les cogniticiens et les programmeurs qui soumettent le prototype en développement à des tests. Nous accorderons une attention particulière à la conception, par le cogniticien, de jeux de tests auprès des experts et des usagers. On constatera les effets rétroactifs des résultats des tests : reformulation de certaines règles, amélioration de l'interface usager et parfois révision du modèle des connaissances. Comme toutes les autres phases importantes du développement d'un système à base de règles, la validation s'inscrit dans un *processus de raffinement progressif du prototype* de l'application à livrer aux usagers.

1. Cycle de formalisation, de prototypage et de validation

Ces trois dernières étapes du développement d'un système à base de règles forment un cycle par lequel on rend concret et opérationnel le modèle des connaissances conçu aux étapes précédentes. Le but est de fournir à l'usager une base de règles complète et cohérente pour résoudre les problèmes identifiés. Le tableau 1 donne un sommaire des tâches à réaliser.

Tableau 1 – Les tâches de formalisation, de prototypage et de validation des connaissances

Étapes	Tâches
Formalisation	Identification des contraintes Structuration des combinaisons attributs-valeurs Rédaction des règles
Prototypage	Codification des règles dans un formalisme Choix des mécanismes d'inférence Programmation de l'interface usager
Validation	Test des résultats produits par le système

À ces étapes, on poursuit une démarche de raffinement progressif : les résultats des tests obligent à revenir à la rédaction des règles pour les raffiner. Il s'agit donc d'un processus de *rétroaction,* puisque l'effet de la validation implique un retour sur les étapes de formalisation et de prototypage, comme le montre la figure 1.

1.1 Intervenants et leur rôle

Comme aux étapes précédentes de développement, le nombre d'intervenants dépend de l'importance du système. Pour un petit système (jusqu'à quelques centaines de règles), une même personne peut tenir les rôles d'expert, de cogniticien et de programmeur. Au-delà de cette taille, des connaissances approfondies du sujet ainsi que des compétences particulières en modélisation des connaissances et en programmation exigent un nombre plus grand de personnes spécialisées pour chacune de ces fonctions. Mais quel que soit le nombre de personnes dans l'équipe de développement, elles doivent travailler en collaboration pour livrer à l'usager un système qui répond à ses besoins.

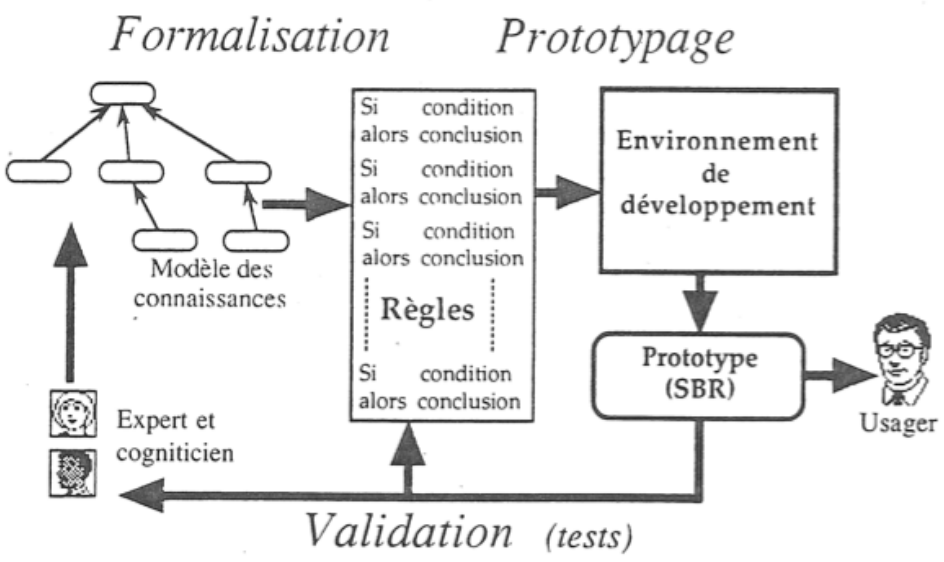

Figure 1 Le cycle de formalisation, de prototypage et de validation d'un SBR.

La **formalisation** des connaissances est la tâche du cogniticien. Elle consiste à traduire en règles le modèle des connaissances déjà mis au point aux étapes précédentes, tout en le

précisant. Il s'agit d'un travail de structuration définitive et de formulation explicite des connaissances pour constituer une base de règles.

Lors du *prototypage,* le programmeur doit transposer dans un environnement informatique la base de règles conçue par le cogniticien. Il s'agit d'un travail de **codification**, voire de programmation, selon la puissance et les fonctionnalités supportées par l'environnement choisi. En somme, le programmeur doit réaliser une application informatique opérationnelle, c'est-à-dire qu'il doit coder une base de règles, s'assurer de sa consistance, programmer l'interface usager et sélectionner les modes d'inférence appropriés. Le cogniticien fournit au programmeur la matière première à ces tâches : contraintes entre attributs, arbres de décisions, règles, choix d'un mode d'inférence, besoins et exigences de l'usager quant à l'interface.

Tous les intervenants du développement d'un SBR sont susceptibles de participer à la *validation.* D'abord l'usager qui, lors des tests, suggère des améliorations au fonctionnement du système et détecte des résultats douteux ou irréalistes. L'équipe de développement doit absolu- ment tenir compte des avis de l'usager pour rendre le système conforme à ses besoins. Ensuite, le cogniticien et l'expert qui évaluent les résultats du système pour proposer les correctifs appropriés. Enfin, le programmeur qui doit corriger les lacunes détectées et traduire en termes informatiques les remarques des autres intervenants. Le cogniticien est responsable de la validation. Il doit planifier les opérations, concevoir les jeux de. tests, enregistrer les résultats des tests, identifier les améliorations à apporter au système et veiller à ce qu'elles soient effectuées.

1.2 Organisation et structuration des connaissances

Un texte précédent nous a montré comment induire des règles d'une grille objets-attributs. En plus d'être systématique, la méthode suggérée est efficace pour un petit nombre de cas. Cependant, quand le domaine à décrire est plus vaste, on doit diviser les cas en sous-ensembles. L'arbre des contraintes entre attributs permet justement de subdiviser un domaine en petits ensembles. En fait, l'arbre des contraintes conduit à des blocs de règles ou des *modules indépendants* si l'on s'astreint à la consigne empirique suivante : limiter chaque contrainte à environ cinq attributs, ce qui correspond à peu près au nombre d'éléments que l'on peut gérer en mémoire à court terme (MCT). Ainsi, on peut construire progressivement une base de règles dont la résultante est une fusion de plusieurs sous-ensembles indépendants. Par ailleurs, on doit tendre à *structurer les règles sous forme hiérarchique* pour faciliter la gestion de vastes ensembles de connaissances. Or, il existe un outil largement répandu permettant de disposer graphiquement les attributs et les valeurs d'un objet : l'**arbre de décisions**.

À partir d'un exemple simple, voyons comment l'arbre des contraintes et l'arbre de décisions peuvent représenter des connaissances et faciliter leur codification sous forme de règles. Il s'agit de déduire le taux d'imposition qu'un contribuable doit payer, selon sa situation personnelle. Un expert en déclaration des revenus nous informe de la marche à suivre. Au lieu de nous fournir plusieurs cas en exemples, il nous présente le tableau 2.

Tableau 2 – Table simplifiée des taux d'imposition (fictifs)

	Sans conjoint	Avec conjoint sans enfant	Avec conjoint un enfant	Avec conjoint plus d'un enfant
moins de 5000 $	4 %	1 %	0 %	0 %
de 5000 $ à 15 000 $	9 %	6 %	4 %	1 %
plus de 15 000 $	16 %	12 %	9 %	6 %

Lorsque les connaissances sont déjà synthétisées de cette manière, il ne reste qu'à les structurer pour se rapprocher de leur expression sous forme de règles. Dans ce cas, il n'y a qu'une seule contrainte, le taux d'imposition. La figure 2, l'arbre de la contrainte Taux d'imposition, montre les relations de dépendance entre les attributs.

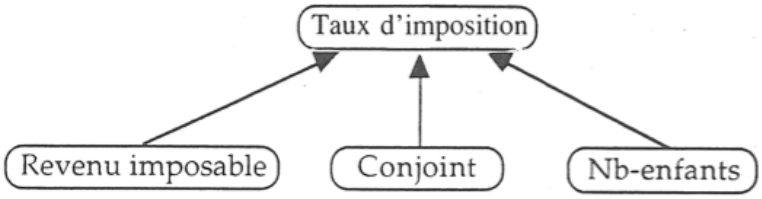

Figure 2 La contrainte Taux d'imposition (version simplifiée).

Ensuite, il faut inventorier les combinaisons de valeurs possibles. L'arbre de décisions nous facilite grandement cette tâche. La figure 3 nous montre comment on peut représenter les connaissances de cette table d'imposition avec un arbre de décisions.

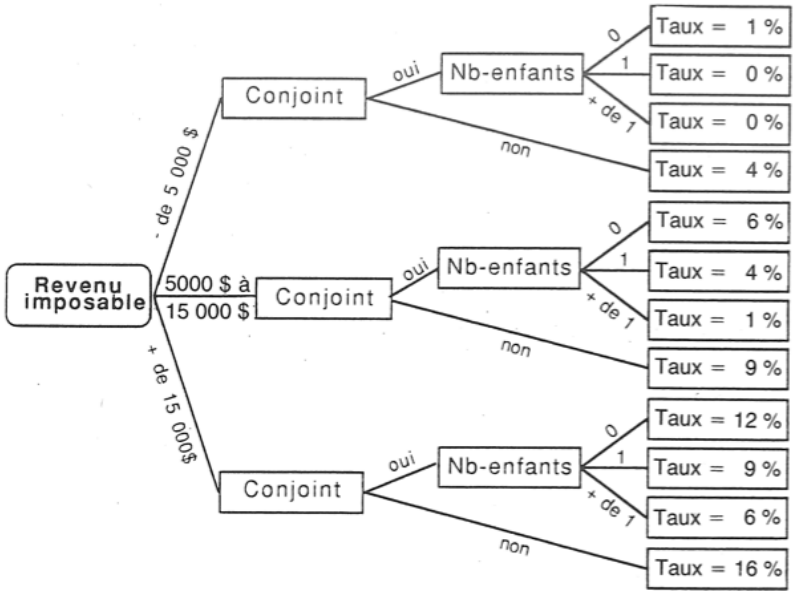

Figure 3 Arbre de décisions Imposition (version simplifiée).

Un tel diagramme détaille les connaissances déjà représentées dans l'arbre des contraintes. Pour le construire, on commence d'abord par *ordonner les attributs selon leur importance.* On

peut appliquer l'algorithme ID3 pour ordonner les attributs selon leur capacité à classer les cas. Ensuite, on identifie les valeurs possibles de chaque attribut, puis on identifie les combinaisons possibles de valeurs d'attributs. Pour construire l'arbre de décisions, on dispose *les attributs dans les « feuilles » et les valeurs sur les « branches »*. *Les feuilles terminales contiennent les conclusions* auxquelles on arrive si les combinaisons de valeurs d'attributs qui précèdent, surviennent.

De l'arbre de décisions, on peut passer directement à l'écriture des règles. Dans ce cas simple, pour écrire une règle, on commence par la « racine » de l'arbre en suivant une « branche » pour finir avec le taux d'imposition en conclusion. C'est la manière de procéder dans le cas où le moteur d'inférence applique les règles en chainage avant, c'est-à-dire des conditions vers la conclusion. Voici deux exemples :

Si	revenu imposable < 5 000 $ et
	conjoint = oui et
	nombre d'enfants > 1
Alors	taux d'imposition = 0 %

Si	revenu imposable > 15 000 $ et
	conjoint = non
Alors	taux d'imposition = 16 %

Cette méthode systématique permet donc d'écrire sans difficulté les douze règles correspondant à autant de conclusions différentes de l'arbre. La même démarche s'applique à un domaine de connaissances plus vaste. Ainsi, on peut imaginer des règles pour obtenir le revenu imposable en plus des règles que l'on peut déduire de l'arbre ci-dessus. On a alors deux sous-ensembles de règles : un ayant le revenu imposable comme conclusion et l'autre ayant le taux d'imposition. Il suffit de construire un autre arbre de décisions pour représenter les nouvelles connaissances relatives au revenu imposable.

L'intérêt de l'arbre de décisions est qu'il permet une synthèse graphique d'un ensemble de connaissances. Il permet aussi de réduire le nombre de règles dans les cas où des combinaisons différentes de valeurs d'attributs mènent à une même conclusion. Dans notre exemple, deux situations d'un contribuable nous permettent de conclure que le taux d'imposition est nul. On peut traduire ce fait par la règle suivante :

Si	revenu imposable > 5 000 $ et
	conjoint = oui et
	nombre d'enfants > 0
Alors	taux d'imposition = 0 %

Par ailleurs, l'arbre de décisions permet une *représentation modulaire des connaissances,* car chaque nœud de l'arbre peut constituer la racine ou la terminaison d'un sous-arbre.

1.3 Prototypage rapide

Un bon moyen de créer un climat de confiance dans une équipe de développement et d'acquérir une crédibilité auprès des experts et des usagers consiste à produire une version

préliminaire du SBR projeté, le plus tôt possible. Le but est de montrer à l'équipe une version expérimentale dès les premiers stades du développement et de la raffiner progressivement avec le processus d'acquisition des connaissances. Ainsi, la première version du prototype du système ne contient que la structure essentielle des connaissances et la manière typique dont l'expert s'y prend pour résoudre des problèmes. On fait une démonstration et des tests sommaires auprès des usagers et des experts, on corrige les erreurs et on améliore le prototype. Cela permet d'intégrer l'avis des experts et des usagers tout au long du développement et d'augmenter l'interaction et la motivation de tous les intervenants.

Le prototypage rapide d'un SBR a plusieurs avantages. D'abord, il fournit très tôt un aperçu du futur système. Un prototype est un excellent *moyen de vérifier des idées.* Il permet aux usagers de comparer le fonctionnement d'une version préliminaire avec l'idée qu'ils ont de ce que devrait faire le système, une fois terminé. D'autre part, pour les experts et les cogniticiens, le prototype est un véhicule pour mettre à l'essai et améliorer leurs connaissances d'un problème.

Le prototypage rapide est bien adapté aux particularités des systèmes à base de règles. Dans le cas des systèmes conventionnels, on peut compléter l'analyse et la conception *avant* de commencer la programmation, car il est plus facile d'avoir une bonne connaissance des problèmes posés et d'en fournir une représentation complète. L'expression « concevoir avant de commencer » s'applique dans la plupart des situations. Or, dans le cas des SBR il est difficile et parfois impossible de construire une représentation définitive d'un domaine de connaissances sans en vérifier la validité. Cela vient probablement du fait que, très souvent, les experts ne possèdent qu'une connaissance informelle et intuitive de leur sujet sans pouvoir l'expliciter dans un premier jet; d'où la nécessité de construire d'abord une ébauche que l'on raffinera par la suite.

Par ailleurs, on peut d'abord mettre à l'essai un sous-ensemble d'une base de règles avant de coder une base dans sa totalité. La *nature modulaire des règles* comme expression des connaissances facilite leur codification, leur vérification et leur modification par groupe. La facilité de modification et de raffinement des règles découle également du fait qu'elles sont dissociées du moteur d'inférence. Des changements apportés aux règles n'impliquent pas une modification ou une recompilation des programmes qui les exploitent. C'est là un avantage des SBR.

1.4 Définition de l'interface usager

L'interface usager est la fenêtre d'accès aux connaissances implantées dans une base de règles. Dans la plupart des cas, la force d'un système est celle de son interface usager. Le soin apporté à sa conception et la souplesse de l'environnement utilisé déterminent la *convivialité* du SBR. Or, la convivialité d'une application est tout aussi importante que ses résultats. La généralisation des interfaces graphiques et du contrôle avec la souris, sur la plupart des ordinateurs, a largement contribué à faciliter et à enrichir l'interaction des usagers avec leurs applications.

En premier lieu, on doit identifier le type d'usagers du système. Quel est leur niveau de connaissance du sujet? Quelles sont leurs occupations? À quelles fins vont-ils utiliser le système? La manière appropriée de répondre aux questions des usagers détermine le *vocabulaire* à employer. En somme, il faut que la terminologie soit adaptée au niveau des usagers. La partie la plus importante de l'interface usager d'un système à base de règles est le *dialogue* que le système doit entretenir avec l'usager. La richesse, la transparence et l'accessibilité d'un dialogue sont déterminées par :

- la justesse et la précision du vocabulaire employé dans la formulation des questions et l'identification des entités de l'application;
- la pertinence, la clarté et l'ordre des *questions* que le SBR pose à l'usager pour cheminer dans son raisonnement;
- l'aspect des écrans : disposition des informations, accessibilité des commandes, fluidité dans l'enchainement, équilibre de l'information textuelle et picturale;
- la puissance du module d'explication du SBR.

La conception du dialogue SBR-usager revient au cogniticien. Celui-ci doit porter une attention particulière aux termes choisis qui doivent être accessibles à l'usager tout en traduisant correctement les connaissances du contexte de l'application. Il faut que le dialogue avec l'usager soit vivant et non répétitif pour éviter la monotonie et la banalisation des questions du système. Par exemple, on doit éviter qu'une même séquence de questions soit posée à l'usager dans des circonstances différentes. Par ailleurs, les possibilités d'explication du raisonnement par le moteur d'inférence sont à considérer. C'est là un aspect spécifique des SBR qu'il faut exploiter. À cet égard, on choisit un environnement de développement qui supporte un module d'explication convivial.

1.5 Planification des tests

La validation fait partie intégrante du processus de raffinement progressif du prototype. Le cogniticien doit planifier soigneusement cette étape du développement d'un SBR. Il prépare un jeu de tests pour chaque version du prototype, note les résultats et propose les correctifs appropriés. Les tests les plus significatifs sont certainement ceux qui concernent l'exactitude des résultats déduits par le système. Toutefois, ce ne sont pas les seuls à faire. Ainsi, on peut classifier les tests selon les aspects évalués du SBR et les personnes qui les effectuent :

- de la part des experts pour vérifier l'exactitude des résultats;
- de la part des cogniticiens pour vérifier que le prototype reflète bien le modèle des connaissances;
- de la part des usagers pour vérifier la résistance et la convivialité de l'interface usager.

2. Outils de développement informatique

De la même manière que l'on a recours à des logiciels comme Excet dBASE ou FileMaker pour développer des applications de gestion de données, on utilise des environnements de développement pour réaliser des systèmes à base de connaissances. L'originalité de ces systèmes, par rapport aux environnements conventionnels, est qu'ils contiennent déjà des modules programmés qui se chargent d'une bonne partie de l'aspect procédural du traitement des connaissances. Dans le cas des systèmes à base de règles, un moteur d'inférence effectue des déductions. Dans un environnement à base de schémas, un mécanisme d'échange de messages gère la communication entre les objets structurés alors que l'on retrouve un mécanisme de résolution ou d'unification dans le paradigme logique. Par ailleurs, la plupart des environnements de développement fournissent des facilités d'édition, de visualisation, de structuration et de mise au point du contenu des bases de connaissances.

2.1 Types d'environnements de développement

Ce sont des familles de logiciels qui donnent une indication de leur diversité d'utilisation, des connaissances requises pour développer une application et du matériel sur lequel ils fonctionnent.

Coquilles de systèmes experts ou systèmes essentiels. Ce sont des progiciels généralement constitués d'un moteur d'inférence, d'une interface usager, d'un éditeur de connaissances et d'un dialogue. Il s'agit d'outils assez faciles à se procurer en vue du développement d'applications sur des micro-ordinateurs. La plupart supportent des représentations à base de règles, d'autres plus sophistiqués, à base d'objets structurés et certains offrent ces deux représentations. Dans cette catégorie de logiciels, on différencie les *coquilles bas de gamme,* généralement limitées aux règles, des *coquilles haut de gamme* qui offrent en plus la possibilité d'intégrer les règles à des schémas. Les micro-ordinateurs de faible puissance supportent les coquilles bas de gamme. La plupart sont offerts à moins de 500 $. Par ailleurs, les coquilles haut de gamme supportent des interfaces assez conviviales et disposent souvent d'outils d'acquisition des connaissances. C'est pourquoi, elles nécessitent plus de ressources informatiques telles la puissance de traitement du processeur et les mémoires internes et sur disques.

- *Ateliers d'ingénierie de la connaissance.* Ce sont des environnements intégrés qui supportent plusieurs représentations et qui offrent des facilités de structuration et de mise au point des connaissances avec une interface graphique conviviale. Ils nécessitent une puissance de traitement assez élevée.
- *Environnements de programmation.* Ce sont des environnements basés sur des langages dédiés à l'lA tels LISP, PROLOG, SMALL- TALK ou des langages conventionnels comme le C. Ils permettent une très grande *souplesse* dans la réalisation d'applications personnalisées et performantes. Le temps de développement d'une application avec ces outils est plus long qu'avec les coquilles ou les ateliers. Par ailleurs, ils exigent une bonne compétence en programmation

2.2 Capacités de structuration des connaissances

Ce sont des fonctionnalités relatives aux types de connaissances traitées, aux relations à établir entre les connaissances et aux facilités d'organisation, notamment à leur structuration hiérarchique.

Traitement des connaissances non structurées :

- données factuelles,
- assertions,
- règles.

Traitement des connaissances structurées :

- arbres de décisions,
- schémas ou objets structurés,
- représentations hybrides.

On reconnaît la plupart des représentations des connaissances décrites dans le texte sur la représentation des connaissances. De plus, les facilités d'organisation et de traitement des connaissances d'un outil de développement se manifestent sous trois aspects : par ses mécanismes d'inférence, par ses possibilités d'édition des connaissances de l'interface de

développement et par ses facilités d'extraction de l'interface usager. Ainsi, plusieurs logiciels de ce type permettent de modifier graphiquement la structure d'une base de connaissances, d'en voir une partie ou de mettre en évidence un sous-ensemble par un mécanisme de *zooming.*

En somme, les caractéristiques essentielles des environnements de développement sont :

- la diversité offerte quant aux modes de représentation des connaissances;
- la convivialité des interfaces usager et expert;
- la puissance des outils d'acquisition et d'induction;
- les ressources informatiques requises pour les utiliser.

2.3 Exemples d'outils de développement de SBC

On trouvera des outils de développement aux adresses suivantes :

Clips	CLIPS 6.3 Beta for Windows Release 3 (http://clipsrules.sourceforge.net/Version63Beta.html)
	CLIPS 6.3 Beta for Mac OS X Release 1 (http://clipsrules.sourceforge.net/Version63MacBeta.html)
	CLIPS Java Native Interface 0.3 Beta (http://clipsrules.sourceforge.net/CLIPSJNIBeta.html)
EXSYS	http://www.exsys.com/
Vanguard	http://www.vanguardsw.com/products/knowledge-automation-system/
XpertRule	http://www.xpertrule.com/pages/authoring-studio.htm

3. Formalisation du premier prototype

Le but de cette étape est la rédaction d'une première version des règles que nous codifierons ultérieurement dans le formalisme de l'outil choisi pour implanter l'application. Pour y arriver, nous allons identifier les attributs et leurs contraintes, faire un inventaire de leurs valeurs possibles, construire des arbres de décisions et les traduire en règles. L'essentiel du travail consiste à *nommer, classer et structurer* les entités qui décrivent un domaine du savoir à codifier dans une base de connaissances. Le but : en arriver à un modèle structuré qui puisse être traduit facilement en règles.

La difficulté de la formalisation réside dans la *complexité* inhérente à certains problèmes causée principalement par le grand nombre d'éléments d'informations et de relations· à gérer. Cela nous conduit à suivre une *démarche qui facilite la gestion de la complexité.* À cette fin, nous adoptons une stratégie très souvent éprouvée en informatique : la subdivision d'un problème en sous-problèmes et la disposition hiérarchique de ses éléments. Il s'agit donc d'une démarche de *développement modulaire et hiérarchique* d'une base de connaissances. Or, l'arbre des contraintes entre attributs et l'arbre de décisions sont des outils taillés sur mesure pour une telle approche. Pour montrer comment on applique cette méthode basée sur la hiérarchie et sur une approche modulaire, nous allons travailler à l'aide d'un exemple concret.

Exemple : Dépannage auto

Nous voulons réaliser une application qui peut guider un usager dans l'identification de certains problèmes de fonctionnement d'une automobile, en particulier ceux reliés au moteur et au système électrique. L'identification d'un problème peut se faire par un dialogue avec l'usager : le programme pose des questions à l'usager concernant des symptômes de fonctionnement de l'auto, puis utilise les connaissances dont il dispose, pour identifier les problèmes qui sont la cause de ces symptômes. Le but de l'application envisagée est donc de produire un *diagnostic* de fonctionnement d'une automobile par l'identification de symptômes.

Or, pour ce genre de situation, un système à base de règles est particulièrement approprié. En effet, on peut formuler les connaissances de cette application sous forme de règles du type :

Si tels symptômes identifiés

Alors tels problèmes de fonctionnement appréhendés

Le volume des connaissances requises à cette tâche est assez imposant pour justifier une approche modulaire de la situation. Par ailleurs, ces connaissances sont facilement accessibles mais éparpillées dans la documentation copieuse fournie par plusieurs manufacturiers. Il faut donc procéder à leur classification et à leur structuration. La première étape consiste à identifier des *modules conceptuels* qui, une fois reliés logiquement, formeront un ensemble cohérent. Une première analyse nous conduit au découpage de la figure 4.

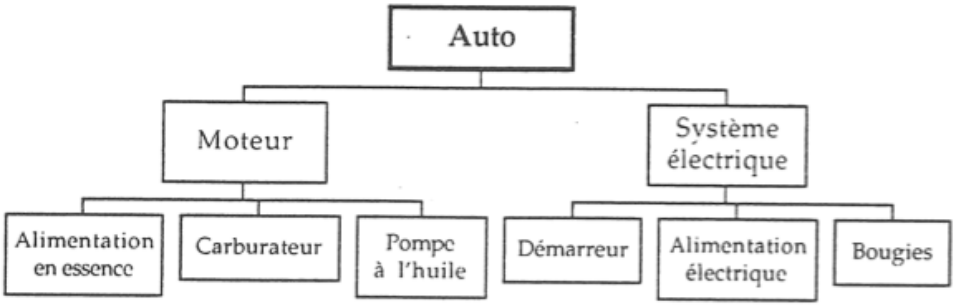

Figure 4 Un modèle simplifié d'une automobile.

Cette approche modulaire fait ressortir les *concepts* à traiter. D'autre part, elle suggère qu'il est possible de développer la base par parties : on peut mettre au point des petites bases de règles, c'est-à-dire des modules indépendants que l'on peut intégrer progressivement à la base globale Dépannage auto. Cette approche suppose aussi qu'il nous faut des règles qui font des *liens entre chacune des bases* en plus des règles spécifiques à chaque objet pour lequel on soupçonne un problème de fonctionnement.

3.1 Choix d'un outil de prototypage

Il est important de choisir le plus tôt possible l'outil avec lequel nous allons implanter la base de règles, ce qui nous guidera dans la manière de rédiger nos règles. Les principaux critères de choix sont de deux types : ceux relatifs à nos *besoins,* la plupart du temps dictés par l'application à développer, et ceux qui découlent de nos *moyens* selon le budget, l'environnement informatique et le niveau de compétence de l'équipe de développement. Dans la plupart des cas, le meilleur choix est celui qui résulte du *compromis* le mieux éclairé.

Revenons aux critères d'évaluation des environnements de développement suggérés à la section 2.2. Il suffit de les raffiner et de ne retenir que ceux qui s'appliquent spécifiquement aux coquilles de systèmes experts. En supposant que l'on limite notre choix aux coquilles de systèmes à base de règles, voici quelques caractéristiques à évaluer lorsque l'on désire une coquille parmi plusieurs coquilles commercialisées :

Quelques caractéristiques des coquilles de systèmes à base de règles

Modes d'inférence
- Chaînage avant Chaînage arrière Chaînage mixte
- Largeur/profondeur

Format des règles
- Syntaxe des noms de variables
- Types de variables Opérateurs *et, ou* et *non* Incertitude
- Priorité des règles
- Classes de règles

Interface usager
- Explications
- Graphisme Fenêtres
- Souris

Capacité/limites
- Nombre de règles Nombre de prémisses Nombre de conclusions

Compléments
- Documentation
- Exemples de bases de règles

Prix des mises à jour

Matériel requis

Ouverture
- Chaînage de bases
- Accès à des fichiers textes, des bases de données, des tableurs, des images
- Appel à d'autres applications
- Appel au système d'exploitation
- Aides au développement
- Instructions du langage hôte de programmation
- Fonctions programmées
- Facilités d'épuration
- Structuration des règles
- Outils d'induction

Évaluation subjective
- Conviviale au développement
- Conviviale pour l'usager
- Respect de l'interface

Prix
- Prix de base
- Prix du module d'exécution autonome

En tenant compte de ces attributs, il faut choisir une coquille compatible avec nos moyens (matériel disponible, montant à investir, niveau de compétence requis, etc.) et susceptible de combler nos besoins (ampleur de l'application, nature des connaissances à implanter, interface souhaitée, etc.).

Si l'on décide de développer l'application « Dépannage auto » avec un système à base de règles, qu'en est-il de nos besoins et de nos moyens? Commençons par les *besoins.* À première vue, voici ce que l'on peut dire du problème à résoudre :

- Il s'agit d'un problème de *diagnostic* tout à fait approprié à l'utilisation d'un système à base de règles. Dans une telle situation, le mode d'inférence en *chaînage arrière* est préférable, puisque le système doit vérifier des faits *à priori* inconnus. En effet, c'est en interrogeant l'usager que le moteur d'inférence recueille les faits qui lui permettent de vérifier des hypothèses.
- Les connaissances à traiter sont homogènes, c'est-à-dire qu'elles peuvent toutes s'exprimer sous la même forme : Si tel symptôme Alors telle cause.
- Le problème est facilement décomposable en modules indépendants. De plus, il est extensible, car on peut ajouter des modules de diagnostic selon les besoins à combler.

- Le problème n'a pas beaucoup d'ampleur : au plus quelques centaines de règles suffiront à détecter la plupart des problèmes de fonctionnement.
- L'interface usager doit entretenir un dialogue familier et assez riche avec l'usager. Le système doit justifier ses recommandations par des textes compréhensibles et explicites et fournir des réponses imagées. Un support graphique serait un facteur de succès dans cette application.

Quant aux moyens, on les suppose modestes. Nous estimons que ce genre de problème se pose généralement dans une petite organisation. On y dispose de micro-ordinateurs et le personnel qui y travaille fait preuve d'une compétence de niveau technique. Par ailleurs, les ressources matérielles et les budgets sont la plupart du temps assez réduits.

Ces considérations nous conduisent à un choix qui s'impose de lui-même : une « *coquille bas de gamme » qui fonctionne sur micro-ordinateur* nous semble la solution optimale à notre équation des moyens et besoins. Enfin, il est préférable de disposer d'un micro-ordinateur aux possibilités d'affichage graphique. Dans un tel cas cependant, les connaissances et l'effort requis à la maîtrise de ces outils sont sans commune mesure avec la complexité de l'application à développer.

3.2 Attributs : contraintes et valeurs

Une contrainte illustre les relations de dépendance entre des attributs qui définissent un objet. Ces relations apparaissent suite à l'étude du sujet et sont imposées par le *but* de l'application à développer. Par ailleurs, on peut s'inspirer des concepts associés à l'objet étudié, pour concevoir les contraintes.

Revenons à l'exemple de Dépannage auto. La figure4 illustre la structure de l'objet Auto auquel on associe les concepts moteur, carburateur, démarreur, etc. Dans une perspective de diagnostic de fonctionnement d'une automobile, on peut dériver des attributs signifiants, selon la nature des problèmes détectables. De plus, on peut parvenir au but visé, c'est-à-dire à l'identification de problèmes, en procédant du général au particulier : on identifie la catégorie de problèmes possibles, puis on localise le problème.

Selon cette approche, nous avons identifié les *attributs* suivants :

- Problèmes auto, Problèmes localisés ou Problèmes étendus, Problèmes avertisseurs, Problèmes démarrage, etc.
- Problèmes carburateur, Problèmes électriques, Problèmes moteur, etc.

La dernière liste renvoie à des problèmes spécifiques d'une automobile. Voici donc les principales contraintes entre les attributs, c'est-à-dire leurs liens de dépendance.

L’arbre des contraintes pour les problèmes auto

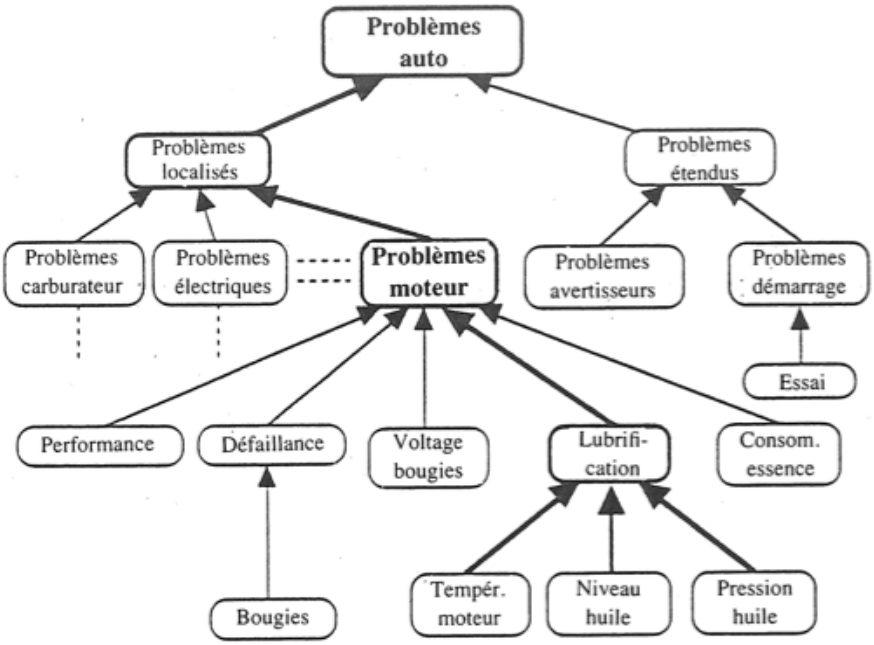

Figure 5 L’arbre des contraintes Problèmes auto.

On voit dans la figure 5 que chaque contrainte constitue un sous-arbre intégré à l’arbre des contraintes Problèmes auto. Cette segmentation de l’arbre des contraintes en sous-arbres nous permet de constituer des *modules de connaissances indépendantes.*

Cet arbre des contraintes est incomplet, car il ne représente que les attributs correspondant à quelques problèmes de fonctionnement. On peut le compléter si on désire diagnostiquer d’autres problèmes possibles. C’est d’ailleurs ce qu’indiquent les lignes pointillées. D’autre part, le prototype que nous développons en exemple est constitué des règles déduites des contraintes illustrées en caractères gras dans la figure 5. Plus précisément, notre prototype résulte de la traduction en règles des quatre contraintes suivantes : Problèmes auto, Problèmes localisés, Problèmes moteur et Lubrification. Nous laissons au lecteur le soin de compléter les autres contraintes en consultant une documentation appropriée.

Les valeurs des attributs

Voici les valeurs possibles de chaque attribut pour les quatre contraintes de notre prototype :

- *Problèmes auto :* spécifiques, généraux, aucun
- Problèmes localisés : défaillance carburateur, défaillance alimentation électrique, défaillance moteur, aucun
- Problèmes étendus : avertissements, démarrage laborieux, aucun
- Problèmes carburateur : à développer, aucun
- Problèmes électriques : à développer, aucun
- Problèmes moteur : mauvaise performance, raté, état des bougies, mauvaise lubrification, consommation, aucun
- Performance : correcte, médiocre
- Défaillance : à développer, aucune

- Voltage des bougies : normal, bas
- Lubrification : température, niveau, pression, normale
- Température du moteur : normale, élevée
- Niveau d'huile : normal, bas
- Pression d'huile : normale, basse
- Consommation d'essence : normale, élevée

Lorsque nous indiquons « à développer », cela signifie que cette partie de l'arbre des contraintes ne sera pas développée pour le moment. Par exemple, pour l'attribut défaillance, on pourrait examiner plusieurs attributs (constante, à haute vitesse, au ralenti, erratique, etc.) qui permettent de déterminer la nature de la défaillance.

3.3 Des contraintes aux arbres de décisions

Nous disposons de toutes les valeurs que peut prendre chaque attribut. Cependant, pour construire une base de règles, on doit *combiner les valeurs d'attributs différents.* De plus, nous devons structurer ces combinaisons pour faire ressortir le lien entre la partie condition et la partie conclusion d'un ensemble de règles. Or, l'arbre de décisions illustre un ensemble de combinaisons. Il est donc un outil approprié.

D'autre part, l'arbre de décisions nous aide à *gérer la complexité* découlant de l'explosion combinatoire lors de l'inventaire des triplets objets-attributs-valeurs. L'augmentation du nombre des cas possibles avec le nombre d'attributs d'un objet peut s'évaluer quantitativement. En général, le nombre de *combinaisons possibles* est égal au *produit* des nombres de valeurs différentes que peut prendre chaque attribut. Ainsi, un objet défini par six attributs ne prenant chacun que trois valeurs distinctes peut générer 3^6 combinaisons, c'est-à-dire 729 situations différentes. Il s'agit pourtant d'un modèle très simple. De ce nombre, seulement une partie ne sera retenue à cause de sa signification concrète. L'arbre de décisions nous aide à identifier les *combinaisons significatives* parmi toutes les combinaisons possibles. De ce fait, il permet de réduire la taille d'un problème et donc, sa complexité. L'arbre de décisions facilite l'épuration d'un inventaire de combinaisons objets-attributs-valeurs à cause de l'aspect visuel d'ensemble qu'il procure.

Un autre avantage de l'arbre de décisions vient du fait qu'il nous aide à ordonner les attributs selon leur importance logique dans la description. Cela veut dire qu'il nous facilite la découverte d'une *hiérarchie* des objets et de leurs attributs. D'ailleurs, on peut considérer un arbre de décisions comme un diagramme hiérarchique. Cette circonstance rend possible l'élaboration modulaire d'un arbre. On peut donc s'intéresser, pendant un moment, aux combinaisons découlant d'un seul attribut, c'est-à-dire ne développer qu'une « ramification » de l'arbre en délaissant les autres « branches ».

Enfin, l'arbre de décisions est un outil de l'approche modulaire, car il permet de centrer notre attention sur une partie d'un problème et de reporter temporairement l'analyse des autres modules. Il est très facile d'intégrer l'arbre de décisions dans notre démarche de conception d'une base de règles, une fois les modules conceptuels et les contraintes défi-nies. En effet, il ne reste qu'à expliciter les contraintes en représentant les combinaisons objets-attributs-valeurs. Le passage des contraintes à l'arbre de décisions se fait en respectant la règle suivante :

> *À une contrainte, correspond un arbre de décisions.*

L'application de cette consigne très simple nous amène naturellement à réaliser des *bases de règles modulaires* où chaque module est de taille réduite et donc, facilement compréhensible. Les tests et les modifications en sont facilités. Enfin, la syntaxe des arbres de décisions peut se résumer aux règles suivantes :

- on place les *attributs* dans des « feuilles » représentées par des rectangles;
- on place les *valeurs* des attributs sur les « arcs » représentés par des droites qui relient les feuilles.

Revenons à la base Dépannage auto. Les matériaux servant à la construction des arbres de décisions sont les contraintes et. les valeurs des attributs que l'on combine en tenant compte de notre compréhension du contexte. Les figures 6 à 9 nous montrent les arbres de décisions correspondant à l'arbre des contraintes Problèmes auto.

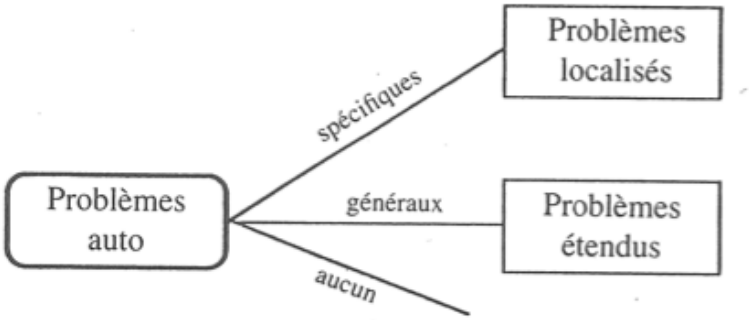

Figure 6 Arbre de décisions de la contrainte Problèmes auto.

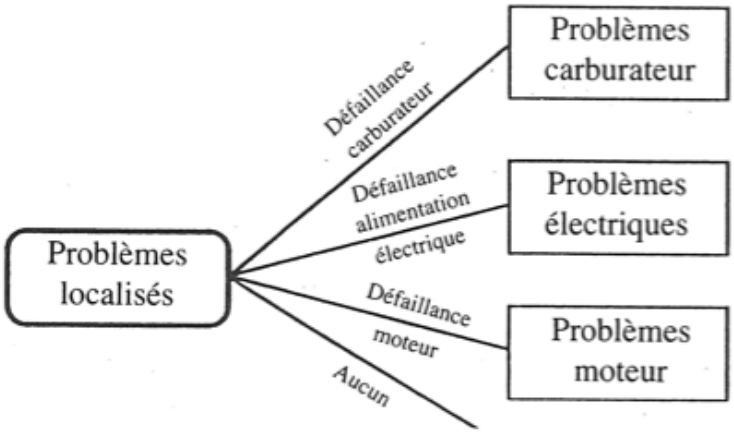

Figure 7 Arbre de décisions de la contrainte Problèmes localisés.

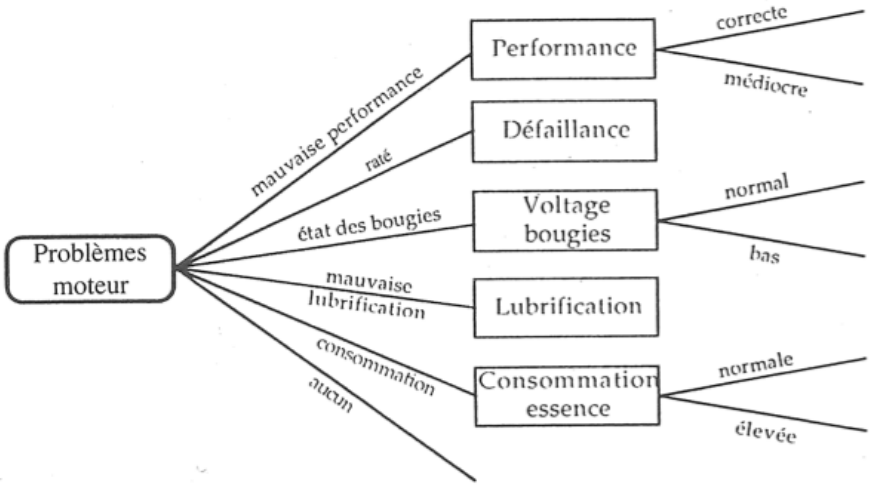

Figure 8 Arbre de décisions de la contrainte Problèmes moteur.

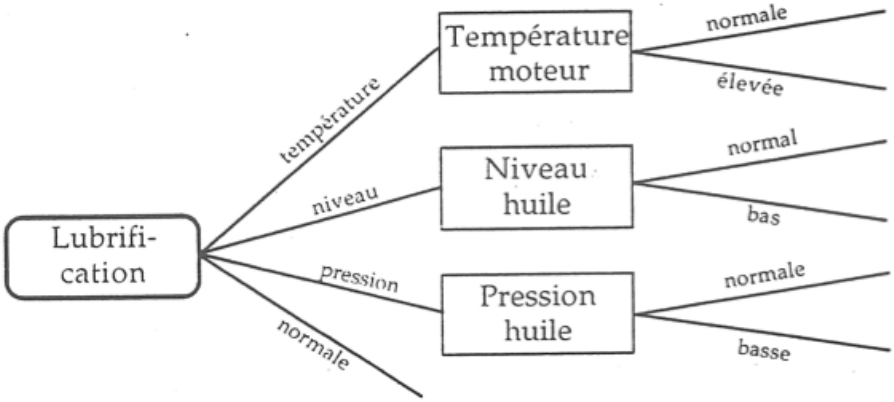

Figure 9 Arbre de décisions de la contrainte Lubrification.

Comme on peut le constater, l'arbre de décisions est un diagramme qui donne une vue d'ensemble des connaissances d'un sujet. C'est un outil d'intégration et de synthèse des connaissances. Il permet de faire ressortir l'essentiel et de délaisser le secondaire. De plus, il met en évidence des erreurs de conception et des redondances, et il fait ressortir des structures.

Puisque nous prévoyons appliquer les règles en chainage arrière, la « racine » d'un arbre correspond à la conclusion que l'on peut déduire si les combinaisons de valeurs qui suivent sont vérifiées. Par ailleurs, on constate que certains attributs se retrouvent dans deux arbres distincts : ce sont les attributs « racine » d'un arbre et « feuille » de l'arbre de niveau supérieur. C'est ainsi que s'opère la jonction entre deux arbres de décisions.

3.4 De l'arbre de décisions aux règles

Une fois l'arbre de décisions terminé, on peut passer à une étape de *codification* des connaissances. Il s'agit du pas qui mène tout droit à l'implantation d'un modèle conceptuel dans un environnement informatique. En effet, les contraintes et l'arbre de décisions sont bel et bien des *modèles conceptuels* des connaissances, car ils sont trop généraux pour être

supportés par la plupart des systèmes à base de règles. En conséquence, nous devons nous rapprocher des représentations fonctionnelles et organiques qui sont effectivement réalisées sur ordinateur.

La rédaction des règles est une étape que l'on peut systématiser pour s'assurer qu'une base de règles soit conforme au modèle que nous avons conçu et tel qu'on l'a représenté par les contraintes et les arbres de décisions. À cette fin, nous suggérons les consignes suivantes·:

- Donner des *noms d'attributs significatifs* et qui sont identiques ou dérivés directement de ceux déjà employés dans les contraintes et les arbres de décisions.
- Rédiger un ensemble de règles pour chaque arbre de décisions pour respecter l'approche modulaire. On peut étiqueter les règles pour savoir à quel arbre elles sont associées. Certaines coquilles permettent de les *numéroter.* Lorsque cela est possible, nous suggérons de numéroter les règles par centaine, la même centaine s'applique à toutes les règles d'une contrainte. Une telle convention de numérotation traduit l'approche modulaire d'une base de règles.
- Rédiger d'abord les règles des modules inférieurs, ensuite celles qui contrôlent les autres modules.
- Rédiger d'abord les règles les plus probables, c'est-à-dire celles qui sont susceptibles d'être le plus fréquemment appliquées.

La rédaction des règles s'effectue en tenant compte du mode d'inférence. En chaînage avant, on part de la « racine » de l'arbre de décisions pour suivre une « branche » et coder toutes les conditions qui conduisent à une conclusion dans une feuille terminale. En chaînage arrière, on procède en sens inverse : on part des feuilles pour remonter le long d'une branche jusqu'à la racine.

Voici donc les bases de règles qui correspondent aux quatre contraintes de notre prototype Dépannage auto.

Les règles de la contrainte Lubrification

401	Si	Température moteur (Auto) = normale et
		Niveau huile (Auto) = normal et
		Pression huile (huile) = normale
	Alors	Lubrification (Auto) = normale
402	Si	Température moteur (Auto) = élevée et
		Niveau huile (Auto) = normal et
		Pression huile (huile) = normale
	Alors	Lubrification (Auto) = température
403	Si	Température moteur (Auto) = normale et
		Niveau huile (Auto) = bas et
		Pression huile (huile) = normale
	Alors	Lubrification (Auto) = niveau
404	Si	Température moteur (Auto) = normale et

Niveau huile (Auto) = normal et
Pression huile (huile) = basse
Alors Lubrification (Auto) = pression

Les règles de la contrainte Problèmes moteur

301 Si Performance (Auto) = correcte et
Défaillance (Auto) = aucune et
Voltage bougies (Auto) = normal et
Lubrification (Auto) = normale et
Consommation essence (Auto) = normale
Alors Problèmes moteur (Auto) = aucun

302 Si Performance (Auto) = médiocre et
Défaillance (Auto) = aucune et
Voltage bougies (Auto) = aucunl et
Lubrification (Auto) = normale et
Consommation essence (Auto) = normale
Alors Problèmes moteur (Auto) = mauvaise performance

303 Si Performance (Auto) = normale et
Défaillance (Auto) = à développer et
Voltage bougies (Auto) = aucun et
Lubrification (Auto) = normale et
Consommation essence (Auto) = normale
Alors Problèmes moteur (Auto) = manque

304 Si Performance (Auto) = normale et
Défaillance (Auto) = aucune et
Voltage bougies (Auto) = bas et
Lubrification (Auto) = normale et
Consommation essence (Auto) = normale
Alors Problèmes moteur (Auto) = état des bougies

305 Si Performance (Auto) = normale et
Défaillance (Auto) = aucune et

		Voltage bougies (Auto) = normal et
		Lubrification (Auto) ≠ aucune et
		Consommation essence (Auto) = élevée
	Alors	Problèmes moteur (Auto) = consommation

306	Si	Performance (Auto) = normale et
		Défaillance (Auto) = aucune et
		Voltage bougies (Auto) = normal et
		Lubrification (Auto) ≠ aucune et
		Consommation essence (Auto) = élevée
	Alors	Problèmes moteur (Auto) = consommation

Les règles de la contrainte Problèmes localisés

201	Si	Problème carburateur (Auto) = aucun et
		Problème électriques (Auto) = aucun et
		Problème moteur (Auto) = aucun
	Alors	Problèmes localisés (Auto) = aucun

202	Si	Problème carburateur (Auto) = à développer et
		Problème électriques (Auto) = aucun et
		Problème moteur (Auto) = aucun
	Alors	Problèmes localisés (Auto) = défaillance carburateur

203	Si	Problème carburateur (Auto) = aucun et
		Problème électriques (Auto) = à développer et
		Problème moteur (Auto) = aucun
	Alors	Problèmes localisés (Auto) = défaillance alimentation électrique

204	Si	Problème carburateur (Auto) = aucun et
		Problème électriques (Auto) = aucun et
		Problème moteur (Auto) ≠ aucun
	Alors	Problèmes localisés (Auto) = défaillance moteur

Les règles de la contrainte Problèmes auto

101	Si	Problème localisés (Auto) = aucun et

		Problème étendus (Auto) = aucun
	Alors	Problèmes auto (Auto) = aucun
102	Si	Problème localisés (Auto) ≠ aucun et
		Problème étendus (Auto) = aucun
	Alors	Problèmes auto (Auto) = spécifiques
103	Si	Problème localisés (Auto) ≠ aucun et
		Problème étendus (Auto) = aucun
	Alors	Problèmes auto (Auto) = généraux

On rédige les règles des autres modules en suivant la même démarche. Pour chaque contrainte ou module conceptuel d'une base de règles, on suit la séquence suivante :

- définition des contraintes entre les attributs d'un objet;
- inventaire des valeurs des attributs;
- construction de l'arbre de décisions;
- rédaction des règles à partir de l'arbre de décisions.

4. Prototypage

Le prototypage consiste à implanter une base de règles avec un outil de développement. C'est un travail de codification et de programmation qui demande une certaine familiarité avec la coquille, notamment son interface de développement ou interface expert. Il s'agit d'un module qui a généralement l'aspect d'un *éditeur* pour les attributs et leurs valeurs, les règles et le dialogue avec l'usager.

4.1 Mise au point de la base de règles

La réalisation d'une base de règles implique la définition des attributs et de leurs valeurs ainsi que la codification des règles. Selon les coquilles, on peut définir implicitement les attributs en codifiant les règles ou les définir explicitement. De plus, la codification suppose un respect du formalisme de la coquille choisie, c'est-à-dire de la syntaxe des attributs et des règles. En général, l'éditeur d'attributs et de règles, supporté par la plupart des coquilles, facilite le respect de la syntaxe en imposant un cadre d'écriture relativement rigide. La figure 10 représente une fenêtre d'édition d'attributs. Il s'agit d'un prototype fictif qui illustre néanmoins l'aspect typique d'une partie de l'interface de développement de la plupart des coquilles commercialisées.

On peut obtenir un écran de ce type en activant un article dans une barre de menu ou en effectuant un choix dans un menu de fonctions supportées par l'interface de développement de la coquille. Cet écran comporte des zones d'entrée de texte, comme le nom de l'attribut et des fenêtres, pour définir ses valeurs ainsi que le libellé de la question à poser à l'usager. De plus, on a accès à des zones sensibles dans lesquelles il suffit de « cliquer » pour effectuer un choix. Il en est ainsi du type de l'attribut et des flèches se déplaçant vers la gauche ou vers la droite/ nous permettant d'accéder à l'attribut précédent ou suivant ou de définir un nouvel

attribut. Une fois les attributs définis, on peut effectuer une sauvegarde et passer à l'édition des règles.

La figure 10 illustre les principaux attributs d'un attribut : son nom, son type, les valeurs qu'il peut prendre et, s'il y a lieu, une question à poser à l'usager, lorsque le moteur d'inférence tente d'affecter une valeur à l'attribut. Comme dans la plupart des langages de programmation conventionnels, on doit respecter des normes relatives à la longueur des noms et des valeurs, à leurs types et à leur syntaxe. On comprend que les valeurs possibles sont les seules qui sont acceptées lors du dialogue avec l'usager, ce qui assure une forme de validation. Dans l'exemple de Dépannage auto, tous les attributs sont de type chaîne de caractères.

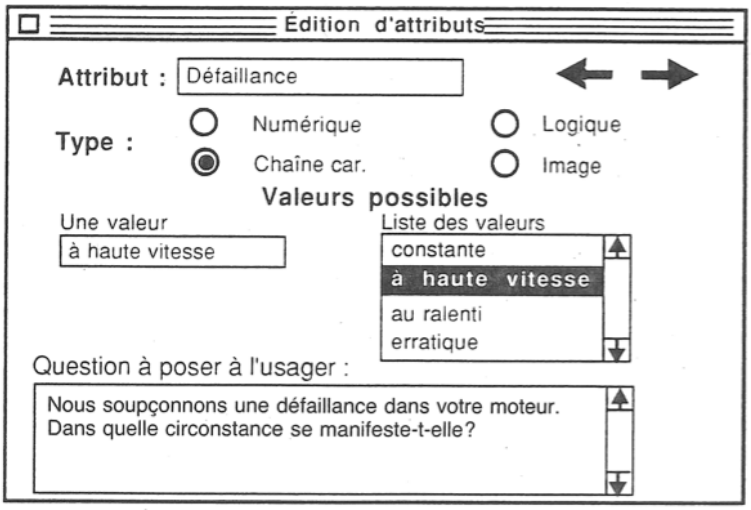

Figure 10 Un prototype de fenêtre d'édition d'attributs.

Quant aux règles, on peut les éditer à l'aide d'un écran conçu à cette fin. En voici un prototype à la figure 11.

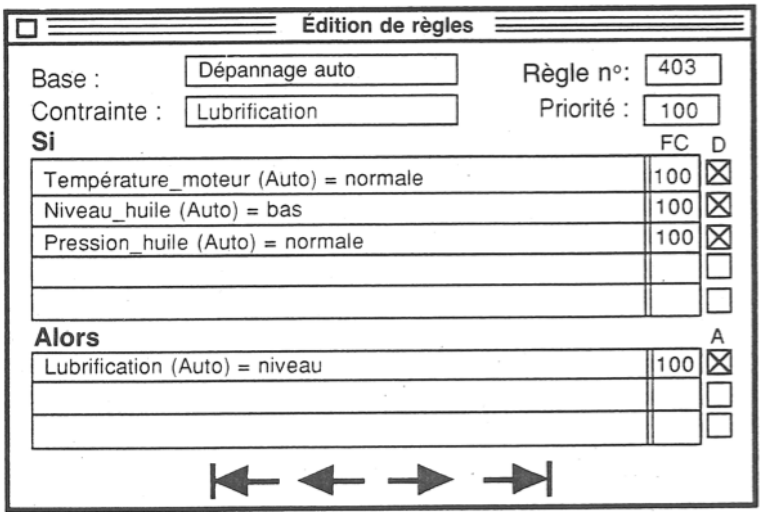

Figure 11 Un prototype de fenêtre d'édition de règles.

On reconnaît ici les attributs standards d'une règle :

- son numéro;
- sa contrainte ou le module de la base auquel appartient la règle;
- sa priorité, c'est-à-dire l'ordre dans lequel peut être appliquée la règle;
- sa partie condition, où une condition est un énoncé littéral ou une expression faisant intervenir un attribut;
- sa partie conclusion, où une conclusion est une expression ou une action qui peut être l'exécution d'une fonction supportée par la coquille ou un appel d'une autre base de règles ou d'une procédure externe.

Certaines coquilles permettent d'associer un ·facteur de confiance (FC) à chaque fait et offrent le choix de demander (D) ou non une · question ou d'afficher (A) ou non une réponse lors du dialogue avec l'usager. L'interface fictive illustrée ci-dessus suppose des limites au nombre de conditions (5) et de conclusions (3). Il n'en est pas toujours ainsi. Enfin, le seul opérateur de relation *et* entre les conditions et les conclusions est implicite. Certaines coquilles offrent en plus les opérateurs *ou* et *non*.

Dans l'exemple de la base Dépannage auto, nous avons considéré comme acquis que toutes les règles avaient la même priorité, 100 par exemple. Il en est de même du facteur de confiance dont nous n'avons pas tenu compte. En ce qui concerne la contrainte, nous avons la contrainte Problèmes auto pour les règles 101 à 103, Problèmes localisés pour les règles 201 à 204, Problèmes moteur pour celles numérotées de 301 à 306 et Lubrification de 401 à 404. Avec certaines coquilles, de tels groupes de règles forment une classe. Comme nous l'avons vu à la section 6 du chapitre 3, la notion de classe correspond à une contrainte, puisqu'elle désigne un ensemble de règles qui fait intervenir des attributs communs ou qui sont contraints par des liens de dépendance. Le regroupement des règles en contraintes permet de faciliter grandement le contrôle des règles. Ainsi, on peut codifier une règle de la façon suivante :

Si	Problèmes_carburareur (Auto) = aucun et
	Problèmes_électriques (Auto) = aucun et
	Problèmes_moteur (Auto) = mauvaise_lubrification
Alors	Activer contrainte Problèmes_moteur

La conclusion de la règle « Activer contrainte Problèmes_moteur » donne l'ordre au moteur d'inférence de sélectionner seulement les règles associées à la contrainte Problèmes moteur. C'est là un moyen très simple de contrôle du moteur.

4.2 Mise au point des mécanismes de contrôle

Lors de la conception d'une base de règles, le cogniticien doit songer à la manière dont le moteur va sélectionner puis appliquer les règles. Cela mène au choix du mode d'inférence approprié à la situation. Selon le niveau de sophistication du moteur dont on dispose, on peut imposer un chainage avant, arrière ou mixte et une recherche en largeur ou en profondeur. Il est possible avec certains environnements de changer de mode d'inférence en cours de dialogue avec l'usager.

Avec la base Dépannage auto, le mode chainage arrière est de mise, car les faits sont inconnus initialement. C'est en obtenant des réponses de l'usager que le moteur peut accumuler des faits qui lui permettent de déduire des conclusions.

Selon la souplesse et la puissance de l'environnement de développement, la *programmation* est parfois nécessaire pour réaliser des opérations spécifiques ou une interface très personnalisée. Les outils de développement comprennent une *librairie de fonctions* que le programmeur doit appeler pour exécuter des tâches spécifiques tels les calculs, les traitements de caractères, le graphisme ou l'accès et la gestion de données sur fichier.

4.3 Mise au point des interfaces

Cette étape du prototypage vise l'élaboration du dialogue avec l'usager et les liens à établir avec d'autres applications ou documents, s'il y a lieu. Le dialogue avec l'usager repose principalement sur les questions que le système pose à l'usager. Le cogniticien doit les formuler soigneusement pour s'assurer de la convivialité de l'application. D'autre part, on doit également penser aux commandes du dialogue accessibles à l'usager tels des boutons, des menus et des zones d'entrée de texte. La plupart des coquilles de systèmes experts offrent une fenêtre de dialogue avec l'usager dont le format s'apparente à celui de la figure 12.

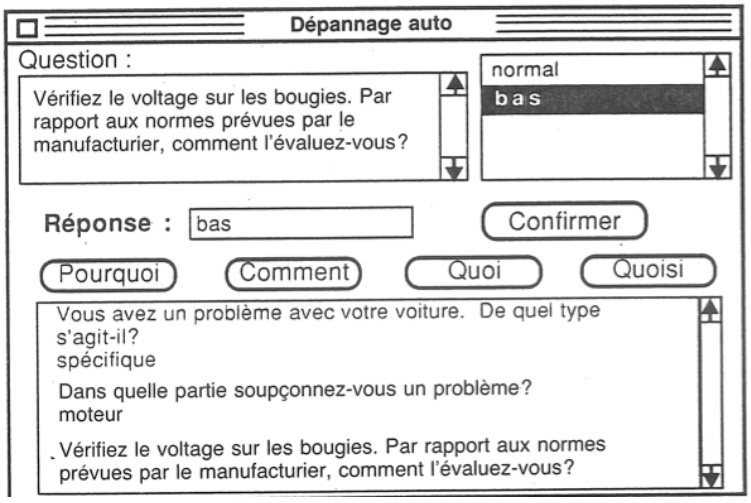

Figure 12 Un prototype de fenêtre de dialogue avec l'usager.

Avec un tel écran de dialogue, l'usager n'a qu'à cliquer dans des zones sensibles pour fournir ses réponses au programme. Ainsi, il sélectionne une réponse parmi celles fournies dans la fenêtre des choix possibles ou il demande des explications en cliquant sur l'un des quatre boutons prévus à cette fin, puis il confirme son choix pour que le moteur d'inférence passe à l'étape suivante du dialogue.

Une interface conviviale doit permettre une sélection parmi des choix possibles, sans qu'il soit nécessaire à l'usager d'entrer lui-même les réponses. Les possibilités d'explication (pourquoi, comment, etc.) sont toujours disponibles. Enfin, il est intéressant, pour l'usager, de disposer d'une fenêtre dont le contenu déroulant donne une trace du dialogue et peut être sauvegardé.

Les questions doivent être explicites et intelligibles à l'usager, de même que les réponses fournies. Cela est essentiel pour maintenir l'intérêt de l'usager à entretenir le dialogue avec le système.

Dans certaines applications, il peut être fort intéressant d'intégrer des données à un dialogue avec l'usager. Par exemple, dans le cas du Dépannage auto, en appliquant la règle 304, on peut afficher un croquis pour illustrer comment on vérifie *Y* état des bougies. De même, lors de l'application de la règle 404, un texte affiché peut expliquer comment se fait la lecture de la pression d'huile. Ainsi, l'application d'une règle peut amener l'affichage d'un texte ou d'une figure ou le traitement de données d'un tableur ou d'une base de données pour appuyer ou expliciter une conclusion. La plupart des coquilles offrent cette option. Il faut utiliser les fonctions appropriées du langage de programmation intégré à la coquille pour permettre ces opérations.

5. Validation du prototype

L'évaluation du fonctionnement d'un système à base de règles est une opération continue et progressive. Elle peut se faire à chaque étape et ses résultats alimentent le cycle de *rétroaction* typique du développement d'un système. Comme la formalisation et le prototypage, la validation doit être menée systématiquement, selon un plan conçu par le cogniticien. Nous décrivons trois aspects inhérents à la validation : les jeux de tests, les séances de vérification et leurs résultats ainsi que les correctifs à apporter.

5.1 Élaboration des jeux de tests

Les tests d'un SBR doivent être planifiés et effectués systématiquement comme pour les autres systèmes informatiques. Considérons d'abord la *stratégie de test.* Elle s'inspire des mêmes principes généraux que ceux appliqués aux étapes précédentes :

- Vérifier un SBR par partie ou module, c'est-à-dire mettre à l'essai chaque classe de règles séparément. Ensuite, on vérifie la cohésion ou les liens entre chaque partie.
- Identifier les situations à vérifier en se servant des arbres de décisions. Toutes les combinaisons de circonstances doivent y être représentées.

Les coquilles de systèmes experts supportent une *validation syntaxique* d'une base de règles. Ainsi, les valeurs d'attributs non définies sont automatiquement rejetées. En général, le choix des réponses possibles étant explicitement circonscrit dans l'interface usager, cela libère le programmeur d'une foule de validations typiques des systèmes conventionnels. Il reste les *validations sémantiques,* c'est-à-dire celles qui ont rapport à la valeur intrinsèque des résultats fournis par le système ou à la cohérence logique des règles. Certaines coquilles détectent des contradictions ou des redondances entre les règles. En tenant compte des éléments stratégiques énoncés ci-dessus, les erreurs possibles sont les suivantes :

- Consistance ou validité.
  - Le modèle des connaissances est faux, c'est-à-dire que les contraintes et les arbres de décisions conduisent à des conclusions qui ne correspondent pas à la réalité.
  - Certaines règles ne sont pas conformes aux arbres de décisions.
  - Des règles se contredisent.
  - La cohésion entre les modules de la base est douteuse.
- Complétude.
  - Le modèle des connaissances est incomplet, c'est-à-dire que les contraintes et les arbres de décisions ne tiennent pas compte de toutes les situations qui peuvent survenir.

- Il manque des règles, car toutes les situations représentées dans les arbres de décisions ne sont pas codifiées.

- Convivialité.
  - Le fonctionnement de l'interface usager conduit à des situations inacceptables.
  - Efficacité.
  - Le temps de réponse est trop élevé.

Nous venons d'identifier précisément les éléments à vérifier. Pour élaborer une *banque de tests,* on doit s'assurer que des tests permettent de déceler l'une ou plusieurs des sources d'erreurs identifiées plus haut. Les jeux de tests doivent être écrits sur des formulaires appropriés prévoyant une section pour enregistrer les résultats, notamment les erreurs détectées. Malgré toutes les précautions prises, il faut toujours se rappeler la limite des tests comme le mentionne Dijkstra [2] dans sa remarque : « Les tests démontrent la présence d'erreurs, ils n'en démontrent jamais leur absence. »

Pour faire un inventaire des tests possibles, on peut *s'inspirer du format de la grille objets-attributs*. Ces grilles servent de *formulaires* pour identifier les combinaisons de valeurs qui devraient correspondre à la plupart des circonstances possibles lors du dialogue avec l'usager. Le tableau 3 représente un jeu de tests possibles pour le module Problèmes moteur de la base Dépannage auto construit en exploitant cette idée. Nous y avons intégré les situations prévues dans l'arbre de décisions de la contrainte Problèmes moteur, figure 8.

Tableau 3 – Un formulaire pour un jeu de tests de la contrainte Problèmes moteur

Problèmes moteur	Perfor-mance	Défaillance	Voltage bougies	Lubrifi-cation	Consom. essence	Résultat test
perform.	correcte	–	–	–	–	
perform.	médiocre	–	–	–		
manque	–	aucune		–	–	
manque	–	constante		–	–	
manque	–	ralenti	–	–	–	
manque	–	erratique	–	–	–	
manque	–	haute vitesse	–	–	–	
état bougies	–	–	normal	–	–	
état bougies	–	–	bas	–		
lubrific.	–	–	–	normale		
lubrific.	–	–	–	tempér.		
lubrific.	–	–	–	niveau		
lubrific.	–	–	–	pression	–	
consom.	–	–	–	–	normale	
consom.	–	–	–	–	élevée	

[2] Un pionnier de la programmation structurée dont les contributions à ce sujet remontent aux années 70.

Pour valider complètement les règles de la contrainte Problèmes moteur, on met à l'essai les quinze combinaisons représentées ici. Le signe indique que l'attribut n'intervient pas dans cette circonstance ou qu'il peut prendre n'importe quelle valeur. Ainsi, la dernière ligne représente la situation où Problèmes moteur = consommation et Consommation essence = élevée, les autres attributs n'étant pas pris en compte dans cette situation.

5.2 Tests auprès des experts et des usagers

Les experts sont les personnes les plus qualifiées pour valider les résultats fournis par un SBR. Ils confrontent les conclusions d'un système avec celles qu'ils déduiraient dans des circonstances similaires. D'autre part, les usagers peuvent juger de la performance d'un SBR, notamment de son interface usager.

Les séances de tests doivent être préparées en fonction des personnes qui évalueront la performance d'un SBR. Pour la validation auprès des experts, on utilise des schémas, des contraintes, des arbres de décisions et des grilles objets-attributs. De ce fait, la validation est intégrée au processus d'acquisition des connaissances. Les tests doivent démontrer aux experts que le modèle des connaissances est correct et que sa transcription dans un formalisme informatique produit de bons résultats.

Une fois que le système fournit des résultats complets et exacts, on peut évaluer la manière de l'opérer. C'est la préoccupation des usagers qui peuvent demander que telle question soit reformulée, que telle réponse soit plus explicite ou que les commandes du programme soient réaménagées pour en rendre l'opération plus conviviale. À cette étape, les usagers peuvent donner une bonne appréciation de la qualité du dialogue avec le programme, notamment quant à la pertinence des questions et à leur ordonnancement. Une impression de monotonie ou de répétition de la même séquence de questions peut ennuyer les usagers et traduire une conception douteuse de la base de connaissances. Enfin, les usagers peuvent donner un avis pertinent sur la performance générale du système, notamment sur son temps de réponse. En somme, la validation auprès des usagers doit permettre le raffinement de l'interface usager et la performance générale de l'application.

5.3 Bilan des tests et correctifs à apporter

La précaution la plus importante à prendre est de *conserver les résultats des tests.* Pour ce faire, le cogniticien doit construire des documents appropriés pour la saisie des résultats de tests. Ensuite, il voit à ce que l'on donne suite aux séances de tests. Les correctifs à apporter à un prototype en développement dépendent des résultats des tests. Selon la nature des erreurs relevées, les modifications qui en découlent peuvent impliquer de simples corrections à l'interface usager, un raffinement de la formulation des règles ou même une révision du modèle des connaissances.

6. Du prototype au système final

Le système à base de règles; construit selon une démarche de raffinement progressif, est livré à ses destinataires, les usagers. On voit d'abord à *compléter le développement* pour s'assurer que tous les modules de la base fournissent les résultats attendus. Il en est de même des fonctionnalités prévues à l'interface usager. Ensuite, le BR est installé dans le milieu réel pour lequel on l'a conçu. C'est l'étape de l'implantation. De plus, on doit former les usagers en vue de son utilisation et assurer un support lors de son opération.

6.1 Implantation

La méthode de développement modulaire que nous avons appliquée offre un avantage fort intéressant : on peut, au besoin, ajouter des fonctionnalités à un SBR sans modifier les principes de sa conception ou les modules déjà implantés. Le découpage en modules d'une base de règles et la nature modulaire des règles favorisent un prolongement de la base sans modification des modules existants.

Ainsi, dans le cas du Dépannage auto, on peut envisager un ajout éventuel de modules de dépannage à ceux déjà existants. Cela implique *l'ajout d'attributs* ou de modules tels Freins, Transmission, Suspension, etc. Une modification au module principal est nécessaire pour y intégrer le contrôle des modules nouvellement ajoutés. La base de connaissances peut alors comporter les modules de la figure 13.

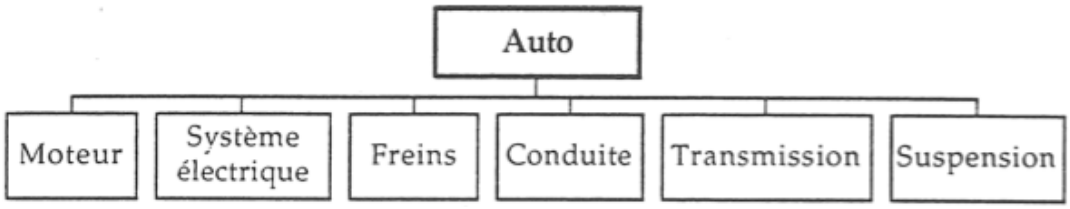

Figure 13 Les modules de la base étendue Dépannage auto.

Même à l'étape d'implantation, des modifications peuvent être requises pour satisfaire des demandes ponctuelles des usagers ou corriger des erreurs détectées in *extremis.* Mais l'implantation d'un SBR, comme tout autre système informatique, suppose une préparation du milieu dans lequel il est intégré et une *formation* de ses usagers. Cela implique une adaptation à cette technologie de la part des personnes touchées par l'implantation d'un SBR.

6.2 Documentation et formation

Les schémas, les tableaux, les grilles, les arbres et les règles sont des documents à l'intention des membres de l'équipe de *développement* d'un SBR, c'est-à-dire les experts, les cogniticiens et les programmeurs. D'autres documents sont destinés à l'équipe *d'opération,* c'est-à-dire les *usagers.* Ces documents décrivent le protocole et les instructions d'opération d'un système dont l'essentiel constitue un *manuel d'utilisation.* Voici un aperçu de son contenu.

- Présentation sommaire du domaine de l'application :
  - le sujet,
  - la fonction du SBR,
  - le rôle de l'usager dans son exploitation.
  - Le modèle des connaissances :(schémas, attributs, modules conceptuels, contraintes, arbres, etc.)
- Description de l'interface usager :
  - les données ou les réponses à fournir au système,
  - les résultats fournis par le système,
  - les images des écrans de dialogue avec l'usager.
- Les consignes d'opération :
  - la mise en marche,

- la sélection du but d'une séance de travail,
- les réponses aux questions du système,
- les demandes d'explications,
- la constitution et la sauvegarde d'une trace d'un dialogue.
- Quoi faire en cas de situation critique?
- Consignes de sécurité.
- Glossaire des termes utilisés.

Le manuel d'utilisation prend une forme procédurale : il décrit *comment* on opère le SBR. Son contenu est donc concis, sans détour et centré sur les modalités d'opération du SBR.

Conclusion

Ce texte a décrit une *méthodologie* conduisant à l'implantation d'un système à base de règles. Celle-ci se présente sous la forme d'une démarche systématique et structurée de réalisation d'une base de règles, de ses mécanismes de contrôle et de son interface usager. Les éléments essentiels de cette méthodologie sont :

- la conception modulaire et par raffinement progressif d'un prototype,
- l'emploi des arbres des contraintes et des arbres de décisions comme outils de structuration des connaissances,
- la place faite à la rétroaction dans la séquence de formalisation, de prototypage et de validation,
- le choix d'un environnement de développement qui résulte d'un
- compromis entre les besoins dictés par les caractéristiques du SBR à développer les moyens dont on dispose.

La méthodologie présentée s'apparente beaucoup à l'analyse structurée de systèmes conventionnels. Ainsi, le passage des contraintes à l'arbre de décisions et enfin aux règles est une séquence de structuration progressive des connaissances. La spécificité du vocabulaire et des outils employés caractérise le développement d'un SBR. L'utilisation des environnements de développement (coquilles) en est une autre particularité. La richesse de leur interface usager ainsi que leurs potentialités croissantes de représentation des connaissances en font des outils d'intégration de divers types d'applications.

À retenir

1. La formalisation, le prototypage et la validation des connaissances acquises consistent à transposer dans une représentation appropriée, notamment un système à base de règles, le modèle des connaissances que le cogniticien a conçu. Le résultat final est un prototype d'une application informatisée prévue pour répondre aux besoins de ses usagers.
2. Les principales étapes de réalisation d'un prototype sont la codification sous forme de règles d'un modèle des connaissances ou formalisation, le choix d'un environnement de développement, le prototypage avec la mise au point de l'interface usager ainsi que la planification et la réalisation des tests. On procède par raffinement progressif des versions successives du prototype à implanter.
3. Le prototypage rapide permet à l'équipe de développement d'un SBR de faire valider très tôt, par les usagers, les résultats et le mode de fonctionnement du prototype à implanter. Cette approche, basée sur une rétroaction continue entre les intervenants, peut faciliter la mise au point d'un système et faire en sorte que ses résultats soient mieux adaptés aux besoins des usagers.
4. On doit construire puis vérifier une base de règles de façon modulaire : pour chaque contrainte entre les attributs, on met au point un ensemble de règles complet, efficace et consistant.
5. Les environnements de développement de systèmes à base de connaissances commercialisés tendent à être ouverts à d'autres applications. Ils permettent des liens avec un système de gestion de base de données, un tableur ou un éditeur de texte. De plus, ils supportent une interface graphique.
6. Les principales fonctionnalités d'un environnement de développement de SBR (une coquille) sont les mécanismes de contrôle du moteur d'inférence, l'édition des règles et les options de structuration, les fonctions d'épuration de bases de règles, les facilités de développement d'une interface usager, les mécanismes d'ouverture à d'autres applications, les fonctions de traitement et les instructions du langage hôte de programmation.
7. Le choix d'un environnement de développement de SBR dépend du matériel disponible, du prix du logiciel par rapport à ses capacités et aux besoins, des ressources que l'on doit investir pour le maitriser ainsi que de la représentation choisie pour implanter les connaissances.
8. Les interfaces d'une application de type SBR doivent rendre transparent l'environnement de développement de SBR. L'expert et l'usager n'ont pas à connaitre les technicités de fonctionnement du système, mais doivent centrer leur attention sur les connaissances et la manière de les exprimer.