Thèse

présentée par Jutta WILLAMOWSKI

pour obtenir le titre de docteur
de l'université Joseph Fourier - Grenoble 1

(arreté ministériel du 30 mars 1992)

Spécialité : informatique

Modélisation de tâches pour la résolution de problèmes en coopération système-utilisateur

Date de soutenance : 06 avril 1994

Composition du jury :

Président :	**Joëlle**	**COUTAZ**
Rapporteurs :	**Monique**	**THONNAT**
	Jean-Louis	**ERMINE**
Examinateurs :	**Stratis**	**GALLOPOULOS**
	Jean-Paul	**KRIVINE**
	Alain	**PAVE**
	François	**RECHENMANN**

Thèse préparée au sein du laboratoire LIFIA/IMAG

REMERCIEMENTS

Je remercie François Rechenmann pour son soutien et l'encadrement scientifique dont j'ai bénéficié tout au long de la préparation de cette thèse, ainsi que Joëlle Coutaz pour avoir accepté de présider le jury. Merci également à mes rapporteurs Monique Thonnat et Jean-Louis Ermine pour avoir accepté de lire et de juger mon travail et pour leurs conseils qui ont permis d'améliorer ce document. Je remercie également Stratis Gallopoulos, Jean-Paul Krivine et Alain Pavé de s'être intéressé à cette thèse et de participer à son jury.

Je remercie tout particulièrement Danielle Ziébelin pour ses conseils et son amitié tout au long de cette thèse, ainsi qu'Audrey qui, jusqu'au jour de sa naissance n'a pas cessé d'entendre parler des tâches. (On ne sait pas encore si elle va faire de l'informatique.) Je tiens à remercier aussi Jérôme Euzenat qui a su m'éclairer lors de nombreuses discussions, de l'introduction jusqu'à la conclusion et au-delà. Je remercie également Patrice Uvietta, lecteur patient et commentateur pertinent de tous les articles et thèses de l'équipe.

Je tiens aussi à remercier François Jean-Marie, ingénieur à Cap Gémini Innovation, que j'ai côtoyé lors de la conception et du développement de SCARP et avec lequel j'ai pu avoir aussi d'autres discussions lors de réunions et de voyages à Lyon. Pour finir, je remercie les utilisateurs de SCARP, et en particulier François Chevenet depuis hier docteur de l'université Claude Bernard, pour leurs coopérations fructueuses.

Je remercie l'ensemble du projet Sherpa, tous les doctorands, chargés de recherches, ingénieurs et stagiaires avec qui j'ai partagé plus que les repas du RU : Cécile, Claudine, Florence, Nathalie, Nina, Olga, Bruno, Christian, Gilles, Jérôme, Johannes, Mathias, Olivier, Pierre[2], Sylvain et Volker. De même, je tiens à remercier tous ceux du "vieux" temps d'Artemis, en particulier Agnès, Bernard et Noël.

RESUME

Un système coopératif d'aide à la résolution de problèmes doit être capable d'un côté de résoudre des problèmes de la manière la plus autonome et automatique possible et permettre de l'autre à son utilisateur d'intervenir ponctuellement dans le processus de résolution ou même de le diriger complètement. Pour permettre au système et à son utilisateur de coopérer, le raisonnement du système doit être facilement compréhensible par l'utilisateur et, inversement, l'utilisateur doit pouvoir aisément communiquer son propre raisonnement au système. Pour cela, une modélisation des connaissances et du processus de résolution de problèmes par planification hiérarchique est proposée. Cette modélisation repose essentiellement sur le concept de tâche. Une tâche permet de représenter un problème à résoudre sur différents niveaux d'abstraction et de lui associer une stratégie de résolution par décomposition récursive en sous-tâches de plus en plus élémentaires. Une telle modélisation permet en même temps :

- de faire gérer efficacement le processus de résolution par le système. La planification hiérarchique permet d'alterner des phases de planification et d'exécution. De cette façon, la stratégie appliquée peut être adaptée de façon opportuniste à l'état de résolution courant.
- d'établir une coopération entre le système et son utilisateur. Cette coopération peut s'établir sur chacun des niveaux d'abstraction et de décomposition introduit dans le raisonnement ; l'utilisateur peut intervenir dans le processus de résolution, prendre ou rendre le contrôle au système.

Le modèle proposé a donné lieu à un outil générique permettant de représenter les connaissances de résolution de problèmes et de les exploiter en coopération avec l'utilisateur. Cet outil, baptisé SCARP, a été expérimenté et validé par différentes applications. Deux exemples, en traitement du signal et en analyse de données, sont présentés dans ce document.

Mots clefs

- tâches
- environnement de résolution de problèmes
- coopération système-utilisateur

ABSTRACT

A cooperative problem solving environment should allow the user to intervene and steer the solution process. Control over the level of intervention should also be permitted. For instance, it should be possible for the user to fully steer the solution, or for the system to proceed with minimal or no user intervention. In order so enable such flexible system-user cooperation the reasoning process must be exploitable by the system and must be easily understandable by the user.

This thesis uses hierarchical planning to model the problem-solving process. The proposed model relies on the concept of a task. A task enables one to define problems at several levels of abstraction. A problem solving strategy can be associated to with each of these problem definitions. Such a strategy describes the recursive decomposition of a task into more elementary subtasks.

This model allows the system to efficiently manage the problem-solving process : hierarchical planning allows decomposition and execution steps to alternate. Cooperation can be initiated at every level of abstraction and decomposition : at each of these levels the user can intervene into the problem-solving process, can take control, or can give it back to the system. Hence, in the course of the solution process the problem solving strategy can change and adapt to the appropriate level of system-user cooperation.

This model has been implemented into a generic tool, called SCARP, which allows the representation and exploitation of the complete problem-solving knowledge in co-operation with the user. The tool has been used, tested, and validated in the context of different applications. This thesis presents the design of SCARP and describes its use, with examples from signal processing and data analysis.

Keywords

- tasks
- problem solving environment
- system-user cooperation

Table des matières

Introduction 1
Résolution de problèmes 1
Résolution coopérative de problèmes 1
Structure du document 3

1. Modélisation de la résolution coopérative de problèmes 4
1.1. Niveau de communication compréhensible et manipulable par l'utilisateur 6
1.2. La tâche : un niveau de description exploitable de façon automatique 15
1.3. Modélisation de la coopération système-utilisateur 25
1.4. Synthèse 31

2. SCARP - représentation des connaissances 33
2.1. Modèle à objets 33
2.2. Eléments d'une base de connaissances 38
2.3. Modèle de tâches 43
2.4. Contexte d'exécution d'une tâche 51
2.5. Bilan 58

3. SCARP - résolution de problèmes 61
3.1. Moteur de tâches 62
3.2. Gestionnaire de dialogues 72
3.3. Conception et fonctionnement de l'interface 80
3.4. Bilan 88

4. Développement et utilisation d'applications 90
4.1. Analyser un domaine pour développer une application SCARP 90
4.2. Résolution en coopération 101
4.3. Bilan 109

Conclusion 110
Bibliographie 113

INTRODUCTION

RESOLUTION DE PROBLEMES

Résoudre un problème consiste, en général, à choisir des actions appropriées parmi un ensemble d'actions possibles, à les organiser dans un plan d'actions et à les exécuter. La résolution de problèmes constitue souvent un processus d'essai-erreur, car l'exécution des actions identifiées comme adéquates peut échouer. Dans ce cas, il faut, au moins dans une certaine mesure, re-choisir, réorganiser et re-exécuter de nouvelles actions. Des systèmes à base de connaissances sont développés pour supporter ce processus de résolution de problèmes. Ce support peut se faire de différentes manières, correspondant aux différentes phases de résolution :

- Les systèmes de planification traitent surtout de la conception de plans d'actions. Ils construisent généralement un plan complet et détaillé pour satisfaire le but poursuivi, avant même de tenter, au moins en partie, son exécution. Ainsi il peut y avoir du travail perdu : si une des actions du plan échoue, celui-ci doit être modifié ou refait complètement.
- D'autres par contre ne se préoccupent pas du tout d'identifier l'ensemble des actions a priori nécessaires pour la résolution, et de l'organiser dans un plan. Ils choisissent une à une des actions qui semblent utiles dans la situation actuelle par rapport au but poursuivi et les exécutent, ceci jusqu'à la satisfaction du but. Si jamais une action choisie échoue ou aucune n'est possible, il y a immédiatement retour arrière et essai d'une action alternative. Ces systèmes réagissent donc tout de suite à des problèmes d'exécution. Mais, mené sans organisation globale, le processus de résolution risque de ne pas être très efficace ; beaucoup d'actions inutiles risquent d'être exécutées.

Pour résoudre ces problèmes, il faut trouver un moyen de se situer entre ces deux extrêmes, de planifier d'un côté globalement le processus de résolution, d'exécuter de l'autre, en même temps et au fur et à mesure de l'avancement de la planification, les parties suffisamment précisées du plan. Nous proposons pour cela une approche particulière de planification hiérarchique qui alterne dans son fonctionnement des phases de planification et des phases d'exécution. Une modélisation du raisonnement par planification hiérarchique décrit le raisonnement sur plusieurs niveaux d'abstraction et de décomposition. Les niveaux d'abstraction permettent de définir et de structurer le raisonnement à différents niveaux de précision ; les niveaux de décomposition permettent de décrire la résolution de problèmes par décomposition récursive en (sous-)problèmes de plus en plus élémentaires.

RESOLUTION COOPERATIVE DE PROBLEMES

Dans les approches classiques de planification et de résolution de problèmes, l'objectif est le développement de systèmes autonomes, fonctionnant pratiquement sans intervention de l'utilisateur. Ces systèmes sont destinés à remplacer l'expert dans un domaine et peuvent être consultés par des non-experts pour résoudre des problèmes.

L'interaction entre l'utilisateur et le système est dans ce cas très limitée : l'utilisateur est réduit au rôle de fournisseur de données. Pendant le processus de résolution seul le système a l'initiative, et ce n'est qu'une fois le résultat trouvé que l'utilisateur peut intervenir et avoir la possibilité de demander des explications.

Mais, envisager une résolution automatique de problèmes par des systèmes à base de connaissances est en général irréaliste. Premièrement, il existe des problèmes dont la résolution nécessite impérativement une intervention de l'utilisateur, l'interprétation visuelle de graphiques par exemple. Deuxièmement, les décisions prises pendant la résolution d'un problème ont souvent un caractère hypothétique. Elles doivent pouvoir être remises en cause si le besoin s'en fait sentir. Troisièmement, dans des domaines complexes, une base de connaissances est forcément incomplète. En particulier, les stratégies de résolution sont souvent mal connues et difficiles à formaliser. Ainsi, un utilisateur peut avoir des connaissances plus complètes que le système à base de connaissances lui-même et doit pouvoir les mettre en œuvre. C'est pour répondre à l'ensemble de ces besoins que nous proposons de concevoir un système *coopératif* d'aide à la résolution de problèmes.

Dans un système coopératif d'aide à la résolution de problèmes, le processus de résolution n'est pas géré de façon entièrement autonome par le système, mais en coopération avec l'utilisateur. Pour les problèmes avec une stratégie de résolution bien connue, le système peut gérer le processus de résolution de manière autonome, prenant en compte simplement les interactions obligatoires avec l'utilisateur. Mais, comme indiqué auparavant, l'utilisateur peut avoir *plus* de connaissances que le système. Etre coopératif signifie ici permettre à l'utilisateur d'intervenir à tout moment dans le processus de résolution pour en modifier les caractéristiques. Un système coopératif d'aide à la résolution de problèmes doit ainsi avoir un fonctionnement souple entre deux extrêmes. Il doit, d'un côté, être capable de gérer le processus de résolution de la manière la plus autonome et automatique possible. De l'autre, il doit permettre à l'utilisateur d'intervenir à tout moment dans le processus de résolution, et même de le guider complètement.

Pour résoudre des problèmes en coopération entre un système à base de connaissances et son utilisateur, d'un côté le raisonnement du système doit être facilement compréhensible (et modifiable) par l'utilisateur, de l'autre l'utilisateur doit pouvoir aisément communiquer son propre raisonnement au système. La modélisation du raisonnement par planification hiérarchique est pour cela particulièrement intéressante parce qu'elle introduit différents niveaux d'abstraction et de décomposition dans le raisonnement. Nous allons montrer dans cette thèse comment une telle modélisation du raisonnement rend possible la coopération : sur chacun de ces niveaux, l'utilisateur peut intervenir dans le processus de résolution, prendre ou rendre le contrôle au système.

Nous proposons avec SCARP un outil qui permet de développer des Systèmes Coopératifs d'Aide à la Résolution de Problèmes[1]. Pour cela, nous avons d'abord développé un modèle de représentation des connaissances adéquat. Ce modèle ne permet pas seulement de représenter les différents types de connaissances nécessaires pour la résolution de problèmes, mais intègre aussi la modélisation des différents types d'interactions avec l'utilisateur qui peuvent être nécessaires pendant le processus de résolution. Ensuite nous avons conçu le système SCARP lui-même, en séparant dans son architecture le raisonnement sur la résolution proprement dite et le raisonnement sur la coopération avec l'utilisateur. Ceci permet de gérer le processus de résolution de manière coopérative, c'est-à-dire souple et facilement modifiable par l'utilisateur.

---

[1] Le système SCARP a été développé en coopération avec Cap Gémini Innovation et cofinancé par le Ministère de la Recherche et de la Technologie.

STRUCTURE DU DOCUMENT

Notre objectif est de permettre au système et à l'utilisateur de coopérer pendant la résolution de problèmes. La précondition pour une coopération est la compréhension mutuelle des deux partenaires. Dans la première partie de ce mémoire, nous étudions donc tout d'abord les approches existantes, qui tentent d'identifier un niveau adéquat de communication du point de vue des deux partenaires. Elles nous conduisent à la notion de tâches et à une modélisation basée sur une planification hiérarchique. Ensuite, nous discutons de différentes approches, développées pour modéliser la coopération proprement dite. Elles adoptent une modélisation équivalente du processus de résolution de problèmes.

Dans la deuxième partie, nous exposons le modèle de représentation de connaissances que nous avons conçu pour SCARP. Nous avons adopté une approche de représentation de connaissances par objets, car elle est particulièrement appropriée pour modéliser la résolution de problèmes par planification hiérarchique. Après avoir introduit le modèle à objets, nous identifions les différents types de connaissances que nous considérons nécessaires pour modéliser le processus de résolution. Essentiellement la tâche va servir à représenter l'ensemble des connaissances attachées à un problème. Nous allons ainsi présenter de façon détaillée notre modèle de tâche, et montrer comment il intègre la formalisation des interactions avec l'utilisateur, qui sont nécessaires pendant le processus de résolution.

Dans la troisième partie, nous montrons comment est réalisée la résolution de problèmes en coopération système-utilisateur dans SCARP. Nous exposons d'abord comment est gérée la résolution proprement dite par le système lui-même. En effet, sauf pour les interactions obligatoires, le système est capable de gérer une résolution de problèmes de façon pratiquement autonome. Il passe pour cela par différentes phases, et prend dans chaque phase un ensemble de décisions importantes. Pour rendre le système coopératif, nous avons d'abord donné à l'utilisateur la possibilité de prendre le contrôle de ces différentes décisions. Nous montrons ensuite comment l'utilisateur peut, a posteriori, remettre en cause toutes les décisions prises et finalement comment il peut explorer de façon interactive la résolution de problèmes nouveaux, non définis dans la base de connaissances du système. Du fait que, pour la coopération, toutes les interactions entre le système et l'utilisateur passent par l'interface du système, la conception de cette interface a une grande importance pour la réalisation d'une résolution coopérative ; elle est détaillée à la fin de cette troisième partie.

Finalement, dans la quatrième partie, nous expliquons comment concevoir et utiliser des applications avec SCARP. Nous montrons en particulier deux exemples d'application, la première dans le domaine du traitement du signal, la deuxième dans celui de l'analyse de données. La première est limitée en taille et en complexité ; aussi elle nous permet de montrer comment un domaine d'application est analysé pour modéliser et exploiter les connaissances avec SCARP. De plus, dans cette application, la résolution des tâches nécessite beaucoup d'interaction avec l'utilisateur. Cette application nous permet ainsi de montrer l'intérêt de modéliser ces interactions et de gérer la résolution avec SCARP. La deuxième application s'attaque à un domaine beaucoup plus complexe. Elle nous permet aussi d'illustrer comment un utilisateur peut exploiter toutes les capacités coopératives de SCARP.

1. MODELISATION DE LA RESOLUTION COOPERATIVE DE PROBLEMES

L'objectif de cette thèse est de concevoir un système coopératif d'aide à la résolution de problèmes, c'est-à-dire un système capable de coopérer avec son utilisateur pour mener à bien une résolution. Il nous semble, tout d'abord, nécessaire de préciser ce que nous entendons par *résolution coopérative de problèmes*. Ceci en particulier pour faire la distinction avec les approches d'intelligence artificielle distribuée. Ces approches, plus précisément les approches de type *tableau noir* [Lâas&89] ou *langages d'acteurs* [Ferb89], [Masi&89], traitent aussi de la coopération, mais à un niveau différent de ce qui nous intéresse ici.

Leur objectif est avant tout de permettre une modularisation des connaissances. Les connaissances sont distribuées parmi plusieurs spécialistes indépendants, des *sources de connaissances* ou des *acteurs* respectivement, avec tous les avantages que cela peut apporter (modularité, adaptabilité, facilité de conception et d'extension, possibilité de parallélisation, tolérance aux pannes, etc.). Pour résoudre des problèmes, ces spécialistes sont, bien sûr, obligés de travailler ensemble, de coopérer. Mais la notion de coopération se réduit ici à la définition d'une architecture logicielle qui intègre des moyens de communication entre spécialistes : il s'agit d'un côté du tableau noir qui sert de support pour l'échange d'informations entre les différentes sources de connaissances, et de l'autre côté de l'envoi de messages entre agents. Il s'agit donc plutôt de la définition d'un canal de communication ; rien n'est spécifié sur la forme et sur le contenu des informations échangées, c'est-à-dire sur le sujet de la coopération.

Notre objectif, par contre, est de permettre au système et à son utilisateur de coopérer pour résoudre des problèmes. Les deux partenaires doivent ici s'accorder sur la manière de résoudre des problèmes. Il est donc avant tout important de trouver une modélisation de la résolution de problèmes qui rende possible cette négociation. Une coopération peut seulement s'établir si les deux partenaires sont capables de se comprendre mutuellement. Il faut donc tout d'abord identifier un niveau adéquat de communication : une modélisation de la résolution de problèmes, compréhensible à la fois par un système et par son utilisateur. Ensuite, il faut définir les fonctionnalités nécessaires pour la coopération proprement dite et les moyens pour la mettre en œuvre.

Le premier but de ce chapitre est donc l'identification d'un niveau de description de la résolution de problèmes, compréhensible à la fois par l'utilisateur et par le système (figure 1). Dans la première partie de ce chapitre, nous allons exposer le point de vue de l'utilisateur et montrer que deux aspects sont importants pour celui-ci : il doit d'une part pouvoir comprendre et suivre le raisonnement mené par le système, et d'autre part avoir la possibilité de lui communiquer ce qu'il souhaite lui faire faire. Dans la deuxième partie, nous allons présenter le point de vue du système et montrer qu'il est important d'identifier, de décrire et de structurer toutes les connaissances afin qu'il puisse les exploiter pour gérer la résolution de façon autonome.

Le second but de ce chapitre est d'identifier les besoins particuliers pour réaliser la coopération système-utilisateur pendant la résolution de problèmes. Il s'agit ici de définir en quoi consiste cette coopération et comment elle peut être réalisée. En effet, pour coopérer il ne suffit pas de se comprendre. Une résolution coopérative est une résolution menée conjointement par le système et par l'utilisateur, c'est-à-dire que d'une part le système doit être capable de mener la résolution de la façon la plus autonome possible,

mais que d'autre part l'utilisateur puisse intervenir à tout moment pour (re-)diriger le processus de résolution. Ceci requiert une modélisation des connaissances, du raisonnement et de l'interaction, permettant de négocier dynamiquement le contrôle du processus de résolution. Nous en discutons dans la troisième partie de ce chapitre.

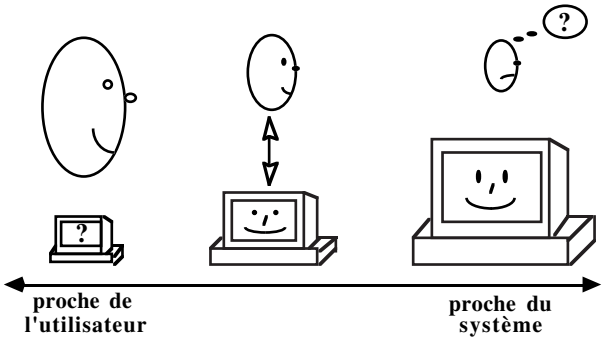


Figure 1 : Niveau de communication adéquat. Pour permettre une communication entre l'utilisateur et le système, il faut tout d'abord identifier un niveau de description des connaissances compréhensible pour les deux à la fois. Cette description doit d'un côté être suffisamment proche de l'utilisateur pour être immédiatement manipulable par celui-ci, de l'autre suffisamment précise pour être exécutable par le système.

La structure du chapitre est donc la suivante :

La première partie discute de différentes approches existantes qui tentent d'identifier un niveau approprié de communication du point de vue de l'utilisateur. Elles ont été développées dans deux domaines : dans le contexte des systèmes à base de connaissances qui ont pour but de gérer de façon autonome, mais compréhensible par l'utilisateur, le processus de résolution de problèmes, et dans le domaine de la modélisation de l'interaction homme-machine, où il s'agit de traduire les intentions d'un utilisateur en interaction (par exemple via un langage de commande) avec un système donné. Nous allons montrer que ces approches proposent une notion commune pour décrire les connaissances nécessaires à la résolution de problèmes : la notion de tâche. Cette notion correspond à la description d'un problème et de sa résolution par décomposition récursive en sous-problèmes de plus en plus simples, qui, finalement, peuvent être résolus directement par l'exécution d'actions élémentaires.

Dans la deuxième partie, nous examinons la pertinence de la notion de tâche du point de vue du système, c'est-à-dire dans l'objectif de gérer de façon autonome le processus de résolution de problèmes. Nous allons montrer que cette notion correspond en fait à une approche de modélisation du processus de résolution par planification hiérarchique. Cette approche, issue du domaine de la planification, est aussi utilisée pour la gestion intelligente de bibliothèques de programmes, c'est-à-dire pour gérer la résolution de problèmes par un enchaînement adéquat de programmes.

Dans la troisième partie de ce chapitre, nous discutons des fonctionnalités nécessaires pour permettre effectivement au système et à l'utilisateur de coopérer pour résoudre des problèmes. Nous y présenterons différentes approches existantes qui traitent de la coopération. D'un côté, elles s'attachent à établir une distribution fixe des responsabilités pour la résolution entre le système et l'utilisateur. De l'autre, elles définissent des mécanismes d'interaction pour négocier dynamiquement cette distribution.

1.1. NIVEAU DE COMMUNICATION COMPREHENSIBLE ET MANIPULABLE PAR L'UTILISATEUR

Notre premier objectif est d'identifier un niveau adéquat de communication entre un système informatique et son utilisateur. Pour cela, la condition première est que l'utilisateur soit capable de comprendre et de manipuler les connaissances à ce niveau. Dans le contexte de la résolution de problèmes, cet aspect a été traité dans deux domaines particuliers et pour des raisons différentes :

- dans le domaine du développement de systèmes à base de connaissances pour résoudre les problèmes de l'acquisition des connaissances et de l'explication du raisonnement à l'utilisateur,
- dans le domaine de la modélisation de l'interaction homme-machine pour décrire la relation entre les intentions d'un utilisateur et les interactions possibles avec un système informatique donné.

Nous allons présenter ici les approches adoptées dans ces domaines afin de dégager leurs caractéristiques communes.

1.1.1. Communication dans les systèmes à base de connaissances

Le premier groupe d'approches est issu de la problématique du développement de systèmes experts (SE). Un SE est un système de résolution automatique de problèmes, conçu pour atteindre les performances d'experts humains dans des domaines limités. Pour cela, il exploite un ensemble de connaissances acquises auprès de ces experts [Hato&91]. Un SE est destiné à pouvoir remplacer les experts humains, à être consulté par des utilisateurs non-experts pour résoudre leurs problèmes. L'interaction entre le SE et son utilisateur est en général très limitée : pendant le processus de résolution seul le SE a l'initiative, et ce n'est qu'une fois le résultat trouvé que l'utilisateur peut avoir la possibilité de demander des explications.

Les premiers SE avaient la structure suivante (figure 2) : ils séparaient la base de connaissances qui réunissait les connaissances nécessaires à la résolution, et le moteur d'inférence, le mécanisme de raisonnement qui permet d'exploiter ces connaissances pour résoudre des problèmes. La base de connaissances était elle-même séparée en deux parties : la base de faits et la base de règles. La base de faits décrivait l'état du monde, tandis que la base de règles définissait les déductions possibles sous forme de règles SI <condition> ALORS <conclusion>. Le moteur d'inférence était chargé d'appliquer ces règles de production aux faits pour déduire de nouveaux faits jusqu'à la résolution du problème posé. Le but principal de la recherche était d'obtenir, aussi efficacement que possible, les mêmes résultats qu'un expert.

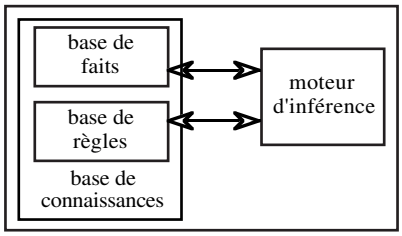


Figure 2 : Structure des premiers systèmes expert. Ils sont structurés en deux parties, la *base de connaissances* et le *moteur d'inférence*. Le moteur d'inférence choisit et applique des règles de la *base de règles* sur la *base des faits* déjà établis, pour déduire de nouveaux faits jusqu'à la résolution du problème posé.

Mais, avec ces premiers SE, trois problèmes sont apparus qui mettent en évidence leurs limites [Horn91] :

- les SE ne (re-)connaissent pas les bornes de leurs propres connaissances. Il n'ont en particulier pas de connaissances de "bon sens". Ainsi ils sont en général capables de trouver les bonnes solutions pour des problèmes typiques du domaine, mais, pour d'autres problèmes, les solutions proposées peuvent être complètement fausses.
- les SE ne sont pas capables d'expliquer de manière satisfaisante leur raisonnement à l'utilisateur. La qualité de l'explication est naturellement limitée par le formalisme de représentation des connaissances utilisé. Pour les systèmes basés sur des règles de production, l'explication naturelle est la trace des règles appliquées. Mais celle-ci est insuffisante puisqu'elle mélange différents types de connaissances et ne montre pas explicitement les stratégies de résolution appliquées [Clan&84].
- le processus d'acquisition, c'est-à-dire d'obtention des connaissances de la part de l'expert et leur formalisation, n'est pas suffisamment étudié et structuré. La distance entre la façon dont pense et agit un expert et les règles de production, c'est-à-dire le formalisme avec lequel les connaissances dans un SE sont représentées et exploitées, est trop grande.

Les deux derniers points concernent en fait des problèmes de communication avec un utilisateur humain : l'explication du raisonnement à l'utilisateur et l'acquisition des connaissances auprès de l'expert (figure 3). La cause de ces problèmes de communication est que le niveau de représentation des connaissances, et par conséquent aussi celui du raisonnement du système qui exploite cette représentation, n'était pas approprié. Il n'était pas suffisamment proche de l'utilisateur. Les formalismes de représentation des connaissances utilisés étaient trop pauvres. Ils ne permettaient de toute façon pas de structurer clairement les différents types de connaissances nécessaires à la résolution de problèmes.

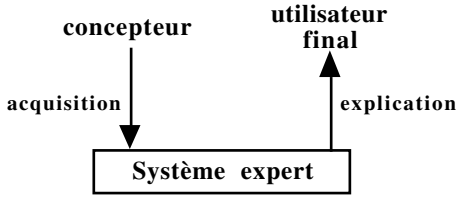


Figure 3 : Communication avec un utilisateur dans un SE. Une communication entre un utilisateur et un SE est nécessaire dans deux situations : pour l'*acquisition* des connaissances, en général dans la phase initiale de conception de la base de connaissances, et pendant l'exploitation du système, pour *expliquer* son raisonnement à l'utilisateur.

Les remèdes proposés pour résoudre les problèmes d'acquisition et d'explication consistent en général à mieux structurer les connaissances nécessaires à la résolution de problèmes. Ils ont essentiellement les mêmes caractéristiques ([Horn91], [Stee90]) : c'est la définition de modèles profonds du domaine, l'explicitation de structures d'inférences et l'identification de différents types de tâches génériques, c'est-à-dire de tâches correspondant aux différents types de problèmes existants.

Pour l'acquisition des connaissances ceci signifie qu'on n'essaye plus d'implémenter directement les connaissances de l'expert. A la place, on commence par la définition d'un modèle à un niveau plus élevé, indépendant de l'implémentation, mais plus proche de

l'utilisateur (cf. le "knowledge level" défini par [Newe82]). Les connaissances décrites à ce niveau ne sont plus directement exploitables par le SE.

Pour passer au niveau de l'implémentation, Steels propose de passer ensuite par un niveau intermédiaire ("knowledge use level") qui précise quand et comment on va utiliser quelles connaissances pendant le processus de résolution de problèmes [Stee90]. Mais la description des connaissances effectuée à un niveau plus élevé doit être conservée et accessible. Elle est utile aussi pendant la phase d'exploitation du SE : étant donné qu'elle se situe par définition à un niveau compréhensible pour l'utilisateur, elle sert directement à lui expliquer le raisonnement du SE.

Deux types d'approches principales nous semblent intéressants dans notre contexte :

- la modélisation fonctionnelle des connaissances, qui passe par l'identification de tâches génériques, c'est-à-dire de différents types de problèmes existants. Elle propose en fait de représenter les connaissances de façon différente pour chaque tâche générique et en fonction de celle-ci ;
- la modélisation conceptuelle des connaissances, qui met l'accent sur l'identification des différents éléments de connaissances nécessaires à la résolution de problèmes et de leurs relations entre eux.

1.1.1.1. Modélisation fonctionnelle

La modélisation fonctionnelle part du fait que, très souvent, les paradigmes de représentation de connaissances ne permettent pas de représenter de façon adéquate les caractéristiques du problème à résoudre. Elle critique le fait que le concepteur d'un système est forcé d'adapter la représentation des problèmes aux outils disponibles, plutôt que de modifier l'outil pour qu'il reflète correctement la structure du problème [Chan86]. La modélisation fonctionnelle constate que, lorsqu'un problème donné et sa résolution sont décrits indépendamment d'une implémentation, un vocabulaire spécifique et approprié au problème est utilisé. L'idéal serait de disposer d'un formalisme de représentation et de résolution qui corresponde exactement à ce vocabulaire. Il pourrait ainsi directement être utilisé pour l'acquisition des connaissances et pour l'explication du raisonnement à l'utilisateur.

Pour cela, la modélisation fonctionnelle propose la notion de *tâche générique*. Une tâche générique élémentaire correspond à un type de problème de base et la manière dont il est résolu [Chan87]. Elle est générique dans la mesure où elle est indépendante du domaine et où elle apparaît dans la résolution de différents types de problèmes plus complexes. Selon ce point de vue, tout problème, décrit à un niveau élevé d'abstraction, correspond lui-même à une tâche générique, soit élémentaire, soit composée d'autres tâches génériques plus élémentaires. Il serait donc intéressant d'identifier et de formaliser tout d'abord l'ensemble des tâches génériques élémentaires existantes. Ensuite, cet ensemble pourrait servir de boîte à outils pour modéliser des tâches génériques plus complexes lors de la construction de SE.

Le représentant principal de cette approche est Chandrasekaran [Chan&87]. Il propose d'identifier trois caractéristiques pour chaque tâche générique : sa spécification, c'est-à-dire le type de problème résolu et la nature de ses informations d'entrée et de sortie, ses connaissances, plus précisément la représentation et l'organisation de celles-ci adaptées à l'utilisation que la tâche doit en faire, et le contrôle qui permet de résoudre la tâche. La figure 4 résume la signification de ces caractéristiques et donne un exemple de tâche générique : la classification.

Chandrasekaran propose donc de représenter les connaissances de façon différente, selon la tâche qui les utilise. Il parle de l'uniformité de la représentation et de l'inférence.

Pour lui, l'intérêt de raisonner au niveau de tâches génériques est que ceci permet de toujours utiliser le vocabulaire approprié à la tâche. Ainsi l'explication pourrait être donnée de manière naturelle, dans les termes appropriés. Enfin, étant donné que chaque problème pourrait être décomposé en tâches génériques, leur identification pourrait aussi faciliter le processus d'acquisition des connaissances.

caractéristiques	*signification*	*exemple : la classification*
Spécification	**type de problème, nature des entrées / sorties**	**classification d'une situation dans une hiérarchie de classes**
Connaissances	**représentation des connaissances adaptée à l'utilisation qu'en fait la tâche**	**hiérarchie de classes et hiérarchie correspondante de spécialistes, capables de déterminer l'évidence de confirmation ou de rejet pour l'hypothèse d'appartenance à une classe**
Contrôle	**stratégie de contrôle / d'inférence approprié**	**de haut en bas dans la hiérarchie : établir et raffiner les hypothèses**

Figure 4 : Caractéristiques d'une tâche générique selon Chandrasekaran. Une tâche générique est caractérisée par sa *spécification*, la description des *connaissances* qu'elle manipule et le mécanisme de *contrôle* qu'elle utilise pour les exploiter. Cette figure donne un exemple de tâche générique, la *classification*.

Mais, si on va jusqu'au bout des idées de Chandrasekaran, deux problèmes se posent (figure 5) :

- l'hypothèse que, pour chaque tâche donnée, il existe un seul type de connaissance et une seule organisation appropriée de cette connaissance, et que ceux-ci dépendent du type de raisonnement associé à la tâche. Si on admet qu'un même problème peut être résolu de façons multiples, d'autres modes de représentation et de raisonnement devraient être pris en compte en parallèle [Dekk92].
- le problème de la communication entre les différentes tâches génériques. Ce problème résulte du fait que chacune aurait une représentation de connaissances différente. Si différentes tâches utilisent les mêmes connaissances, celles-ci sont représentées plusieurs fois, différemment pour chacune d'entre elles. Pour pouvoir les combiner, c'est-à-dire pour définir des tâches plus complexes, des moyens de communication entre les différentes tâches génériques doivent être introduits, ce qui limite l'intérêt de la modélisation fonctionnelle.

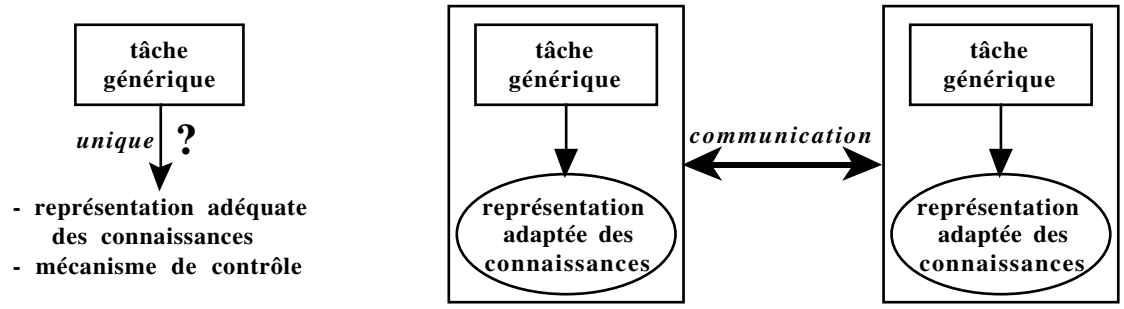


Figure 5 : Problèmes de la modélisation fonctionnelle. Une tâche générique est supposée avoir une seule représentation des connaissances adéquate et un seul mécanisme de contrôle associé. De plus, la représentation différente des connaissances pour chaque tâche générique pose le problème de la communication entre tâches génériques.

1.1.1.2. Modélisation conceptuelle

Tandis que la modélisation fonctionnelle consiste essentiellement dans l'identification de tâches génériques, la modélisation conceptuelle tente d'établir un modèle conceptuel des connaissances nécessaires à la résolution de problèmes. Elle a été adoptée initialement pour répondre au problème de l'acquisition des connaissances et perçoit le développement de SE avant tout comme une activité de modélisation : un SE représente le modèle opérationnel des phénomènes observés dans la réalité. La conception de SE passe ici d'abord par un modèle conceptuel, indépendant d'une implémentation. Celui-ci doit ensuite être transformé en une architecture appropriée pour le SE, tout en conservant sa structure complète. Par conséquent, ce modèle pourra aussi être utilisé pour expliquer le raisonnement effectué par le système résultant (figure 6).

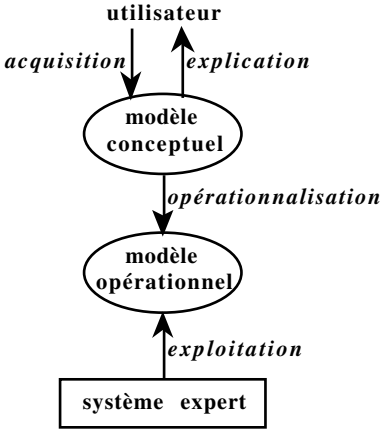


Figure 6 : Le modèle conceptuel comme moyen de communication entre un SE et son utilisateur. L'*acquisition* et l'*explication* du raisonnement du SE passent par un *modèle conceptuel*. Ce modèle décrit les connaissances à un niveau proche de l'utilisateur mais indépendant de l'implémentation. Il doit être "*opérationnalisé*" pour être *exploitable* par le système.

Nous présentons par la suite le représentant principal de cette approche, KADS (Knowledge Acquisition and Documentation Structuring) [Wile&86] [Breu&88]. La modélisation conceptuelle dans KADS est fondée sur l'identification de différents types génériques de connaissances en fonction des différents rôles qu'elles peuvent jouer dans le processus de résolution de problèmes. Le terme générique n'a ici pas la même signification que dans la modélisation fonctionnelle : il exprime l'indépendance non seulement du domaine d'application mais aussi du type de problème à résoudre. Pour cela les connaissances sont d'abord structurées en quatre couches successives avec des interactions limitées : *domaine, inférence, tâche* et *stratégie* (figure 7).

couche	*relation*	*objets manipulés*
stratégie		**plans, réparation**
	contrôle	
tâche		**buts, tâches**
	applique	
inférence		**inférences, métaclasses**
	décrit	
domaine		**concepts, propriétés, relations**

Figure 7 : Structuration des connaissances en couches (dans KADS). Quatre couches successives sont distinguées dans la représentation des connaissances au niveau du modèle conceptuel. A chacune de ces couches correspond un ensemble d'objets génériques manipulés avec des relations limitées.

Ces couches ont la signification suivante :

- La couche domaine contient les *concepts*, les *relations* et les *structures statiques* du domaine d'application. A l'opposé de la modélisation fonctionnelle, cette représentation est indépendante de la tâche qui doit les utiliser.
- La couche inférence décrit de façon abstraite les *inférences* praticables sur la couche domaine. Elle définit une fonction abstraite dans le processus de résolution de problèmes. Ses arguments sont appelés *méta-classes*. Celles-ci définissent le rôle que jouent des concepts de la couche domaine pour la fonction correspondante. Un même concept peut jouer différents rôles dans différentes structures d'inférence. Les inférences définies ne préjugent ni du moment où elles seront activées, ni de la façon dont elles seront utilisées. A chacune est attachée une méthode qui décrit comment elle est réalisée.
- La couche tâche définit comment la connaissance exprimée sur les deux couches inférieures est utilisée dans un contexte de résolution, pour obtenir un *but* particulier. Elle décrit sous forme de *tâches* comment les inférences doivent être activées et combinées. L'interaction des couches inférence et tâche est montrée dans la figure 8.
- La couche stratégie doit permettre de combiner des tâches pour établir des *plans*. Si des problèmes imprévus apparaissent, elle permet de modifier le contrôle en cours de route. Ceci revient en général à sélectionner une structure de tâches alternative pour la résolution [Hick&88]. Cette couche n'est en fait pas souvent utilisée, ou alors seulement de manière rudimentaire [Schr&88]. Soit elle n'est pas nécessaire, parce que statique et modélisée au niveau des tâches, soit elle est trop difficile à modéliser.

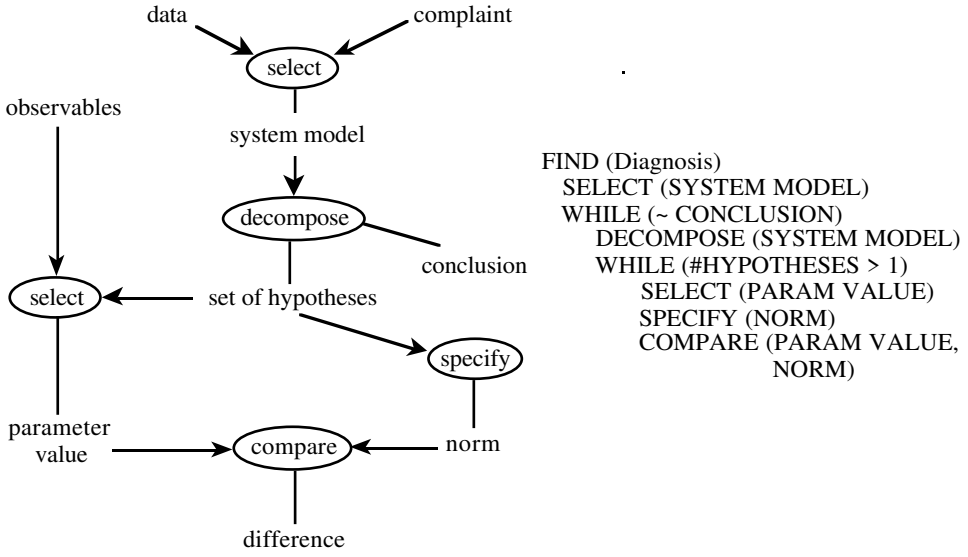


Figure 8 Les couches inférence et tâche, d'après [Breu&88]. La partie gauche montre un exemple d'une structure d'inférence. Elle décrit des fonctions abstraites (*select, specify*, ...) avec leurs arguments (*observables*, ...). Ces arguments ou méta-classes sont des rôles que doivent remplir les concepts du domaine. La partie droite décrit une tâche qui définit comment ces inférences doivent être utilisées et enchaînées.

En fait, seule la modélisation des connaissances sur la première couche doit dépendre entièrement du domaine. Pour les autres couches, KADS propose de définir des modèles prototypiques, indépendants d'un domaine particulier d'application. Ceux-ci pourraient ensuite être raffinés pour être instanciés dans un domaine cible. Ils seraient modifiables

dans le cas où ils ne correspondraient pas exactement aux besoins. Parmi ces modèles prototypiques, il y a aussi des "tâches génériques" comme structures de raisonnement élémentaires. La modélisation conceptuelle d'un domaine avec KADS peut alors consister à effectuer l'analyse des tâches du domaine, afin d'identifier les tâches génériques utilisées et leur interaction. Ainsi la modélisation d'un domaine peut passer par un raffinement des modèles initiaux présents, en les modifiant ponctuellement si nécessaire. Mais cet aspect nous intéresse moins, puisqu'il semble plutôt rare de retrouver dans un domaine d'application exactement (ou presque) des modèles prototypiques pré-définis.

Ce qui nous semble intéressant dans KADS est plutôt la tentative d'identifier différents types de connaissances nécessaires à la résolution de problèmes et leur signification pour le processus de résolution. Surtout les éléments des trois premières couches, domaine, inférence et tâche, et leurs inter-relations nous semblent pertinents :

- la définition de concepts, de propriétés et de relations entre concepts,
- la définition d'inférences et de méta-classes qui définissent le rôle des concepts pour ces inférences,
- et la définition de tâches qui organisent et ordonnent l'exécution des inférences (et de tâches plus élémentaires) pour atteindre des buts.

1.1.2. Traduction des intentions d'un utilisateur en interaction avec la machine

Entre un SE et son utilisateur, une communication est nécessaire à la fois pour acquérir des connaissances et pour expliquer à l'utilisateur le raisonnement suivi par le système. Un SE résout des problèmes de façon autonome. L'utilisateur n'intervient pas dans le processus de résolution. Un autre domaine, la modélisation de l'interaction homme-machine, traite du fonctionnement inverse : il décrit comment l'utilisateur peut mener la résolution de problèmes par interaction avec un système informatique donné (figure 9).

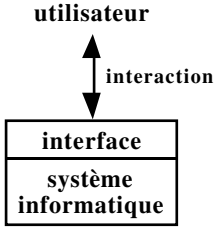


Figure 9 : Modélisation de l'interaction homme-machine. Ici, l'objectif est de modéliser comment un utilisateur doit interagir avec un système informatique pour résoudre ses problèmes.

La forme de communication système-utilisateur traitée ici, est donc la traduction des tâches d'un utilisateur en interactions élémentaires nécessaires pour leur résolution. Selon le système en question, les problèmes à résoudre et les interactions possibles peuvent être très divers. Des exemples sont : pour un problème, la recopie d'un ensemble de fichiers d'un répertoire dans un autre, pour des interactions, l'action-utilisateur *appuyer-sur-des-touches* ou encore l'action-système *afficher*.

La modélisation de l'interaction homme-machine, appelée généralement "analyse de tâches", peut être effectuée avec différents objectifs [Wils&88], le plus souvent pour analyser ou concevoir l'interface homme-machine d'un système par rapport aux traitements effectués par son utilisateur. Les résultats immédiats d'une analyse de tâches sont :

- la description des fonctions élémentaires du système qui sont à la disposition de l'utilisateur,
- la définition des tâches possibles de l'utilisateur,

- et la manière dont les fonctions élémentaires doivent être combinées pour résoudre ces tâches.

Les premiers outils pour l'analyse de tâches sont des grammaires formelles comme CLG [Mora81] et TAG [Payn&86]. D'autres, comme TAKD [John&91] et ETAG [Taub91] s'en sont inspiré. Ces outils sont en général basés sur une modélisation correspondant à la planification hiérarchique (cf. section 1.2.1.) : différents niveaux d'abstraction et de décomposition sont distingués [Sebi88]. Ils décrivent la décomposition progressive des tâches de l'utilisateur en tâches de plus en plus élémentaires qui sont finalement traduites en interactions avec le système. Cette description est ainsi structurée à des niveaux successifs d'abstraction, partant d'un niveau adéquat pour l'utilisateur et arrivant finalement à un niveau adéquat pour le système.

[Scap&89] critique le fait que ces outils restent souvent sur un niveau trop bas et sont purement descriptifs. TAG par exemple est volontairement restreint à la description de tâches simples. Des tâches complexes qui nécessitent de la coordination, une itération par exemple, ne peuvent pas être décrites. Avec MAD, [Scap&89] propose de modéliser la tâche comme objet décrivant de manière déclarative ses aspects fonctionnels et ses aspects opérationnels. Comme les approches antérieures, MAD est basée sur le principe de planification hiérarchique, mais les connaissances sont ici plus structurées et représentées de manière uniforme. C'est pourquoi nous allons détailler cette approche.

Son concept principal est la tâche, qui peut être, soit une action élémentaire, soit une tâche complexe avec une structure de décomposition en sous-tâches. La description d'une tâche dans MAD comporte les champs suivants :

- l'état initial : un ensemble d'objets qui définit les entrées de la tâche,
- l'état final : un ensemble d'objets qui définit les sorties de la tâche,
- le but : un sous-ensemble de l'état final qui indique l'objectif propre, recherché avec la tâche,
- les pré-conditions : un ensemble de prédicats qui expriment les contraintes sur l'état initial qui doivent être satisfaites pour pouvoir exécuter la tâche. Un sous-ensemble de ces prédicats forme les conditions nécessaires qui permettent d'activer la tâche,
- les post-conditions : un ensemble de prédicats qui expriment les contraintes sur l'état final qui doivent être satisfaites après l'exécution de la tâche.

Si la tâche est élémentaire, un champ supplémentaire décrit l'action à exécuter, Si elle est complexe, un autre champ supplémentaire définit sa décomposition en sous-tâches, utilisant les constructeurs *séquence*, *parallélisation*, *alternative*, *boucle*, et *facultatif*.

Dans MAD, une tâche peut aussi être décrite sur plusieurs niveaux d'abstraction ; une action abstraite d'édition par exemple peut être précisée en l'une des actions suivantes : couper, effacer, coller ou copier. Ces liens de spécialisation entre tâches décrivent aussi un héritage des caractéristiques entre tâches plus générales et tâches plus spécifiques. Cet héritage concerne tous les champs énumérés ci-dessus, en particulier la stratégie de résolution par décomposition en sous-tâches (figure 10).

Scapin constate qu'en fait la frontière entre la modélisation par tâches de l'activité d'un utilisateur et la modélisation par tâches du raisonnement pour la résolution automatique de problèmes n'est pas toujours claire. Nous verrons dans la section 1.2.2. que la modélisation adoptée par MAD présente beaucoup de ressemblances avec celle du système SCAI, développé pour le pilotage de programmes en calcul scientifique, qui intègre aussi un moteur d'inférence pour gérer l'exécution de tâches.

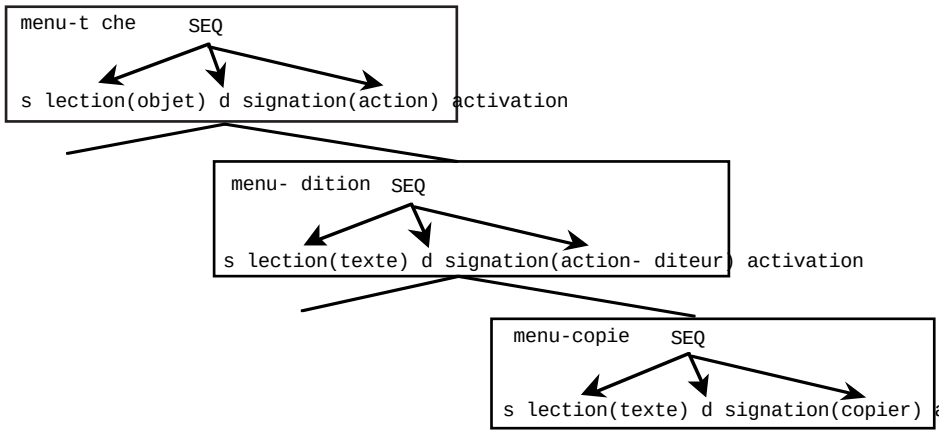


Figure 10 : Spécialisation de tâches dans MAD. Une tâche générale décrivant une action sur un menu consiste d'abord à sélectionner un objet, ensuite à désigner l'action à effectuer et finalement à activer cette action. Une première spécialisation restreint la tâche à l'édition de texte ; une deuxième spécialisation restreint à l'action précise de copier un texte.

1.1.3. Notion de tâche

Nous avons présenté ici un ensemble d'approches concernées par la nécessité de définir un niveau adéquat de communication système-utilisateur du point de vue de l'utilisateur. Elles sont issues de deux domaines, celui du développement de SE et celui de la modélisation de l'interaction homme-machine. Elles ont une notion en commun : la tâche. Une tâche correspond à la description de la résolution d'un problème par décomposition récursive en sous-problèmes de plus en plus simples, qui, finalement, peuvent être résolus directement par l'exécution d'actions élémentaires. Du point de vue de l'utilisateur, la notion de tâche semble donc constituer un niveau de description compréhensible et par conséquent adéquat pour la communication avec un système informatique.

Dans le domaine du développement de SE, les tâches ont été proposées sous deux points de vue :

- celui de la définition de tâches génériques. Selon ce point de vue, tout problème peut être décrit sur un niveau abstrait comme une tâche générique. Il peut directement correspondre à une tâche générique élémentaire, ou alors être composé de tâches génériques plus élémentaires. Le but est, d'identifier un ensemble de tâches génériques élémentaires qui constitueraient les briques de base exclusives pour la construction de tâches plus complexes.
- celui d'un élément générique de connaissances nécessaire à la résolution de problèmes. Une tâche est ici définie comme moyen pour atteindre un but par enchaînement de moyens d'inférence. Chaque tâche constitue elle-même récursivement un moyen d'inférence, mais plus complexe que ceux qu'elle utilise elle-même.

Le premier de ces deux points de vue nous semble moins pertinent, car nous ne sommes pas convaincus de la facilité d'identification de tâches génériques à partir d'un problème réel. Le second par contre paraît plus intéressant. Ici la description de tâches n'est pas limitée par un ensemble de tâches génériques élémentaires. La tâche est "seulement" identifiée comme un des types de connaissances essentiels pour la résolution de problèmes.

Dans le domaine de la modélisation de l'interaction homme-machine, les tâches ont été introduites pour décrire comment un utilisateur doit traduire ses intentions en interactions élémentaires avec le système. Les tâches de plus haut niveau correspondent en fait aux intentions de l'utilisateur, et sont ainsi immédiatement compréhensibles pour celui-ci. La décomposition indique comment elles doivent être traduites à des niveaux de plus en plus proches du système.

Les approches discutées ci-dessus n'ont pas été développées pour être exploitables par un système, dans le sens où celui-ci serait capable d'exploiter cette représentation pour gérer de façon autonome la résolution de problèmes. Mais [Payn&86] compare déjà les tâches simples décrites en TAG aux opérateurs élémentaires dans des systèmes de planification. Il suggère de coupler leur représentation avec un planificateur pour pouvoir décrire la résolution de tâches plus complexes. Mais il faut constater que la représentation choisie concerne essentiellement les actions à effectuer et les paramètres à spécifier pour ces actions. Il n'y a pas de notions de pré- ou de post-conditions dans le formalisme TAG. Il est donc nécessaire de préciser la représentation de tâches à l'aide d'autres informations, indispensables pour qu'un système puisse gérer leur exécution de façon correcte.

1.2. LA TACHE : UN NIVEAU DE DESCRIPTION EXPLOITABLE DE FAÇON AUTOMATIQUE

Pour l'utilisateur, la notion de tâche semble constituer un niveau adéquat de communication avec un système informatique. Il convient à examiner si ce niveau est aussi compréhensible par le système, c'est-à-dire si celui-ci peut être capable de l'exploiter pour gérer la résolution. La résolution de problèmes passe, en général, par une étape initiale de planification. Celle-ci sert à choisir des actions appropriées parmi un ensemble d'actions possibles et à les organiser dans un plan, qui peut ensuite être exécuté. C'est pourquoi nous allons étudier d'abord le domaine de la planification. Ceci va nous permettre d'établir de fortes ressemblances entre la notion de tâche et un type spécifique de planification hiérarchique.

La planification est surtout concernée par la conception de plans d'actions, et moins par la gestion de leur exécution. Un domaine qui traite en même temps de leur représentation et de leur exécution est celui de la gestion intelligente de bibliothèques de programmes. Nous allons montrer que des approches développées dans ce domaine appliquent également des techniques de planification hiérarchique pour gérer l'exécution de programmes. La notion de tâche représente par conséquent aussi un niveau de représentation exploitable pour le système.

1.2.1. Tâche et planification

La planification consiste à concevoir un plan d'actions pour atteindre un état but à partir d'un état initial. Les connaissances décrites en planification sont l'état du monde, le problème à résoudre (le but ou les caractéristiques de l'état but) et l'ensemble des opérateurs ou des actions possibles, qui permettent de modifier l'état du monde (figure 11).

En général, l'état du monde est représenté par un ensemble de faits. Souvent des planificateurs sont construits à partir de systèmes à règles de production: la base de faits correspond à la représentation de l'état du monde, la base de règles à l'ensemble des opérateurs. Chaque règle de production modélise un des opérateurs disponibles. La partie SI décrit les pré-conditions pour l'application de l'opérateur (les faits nécessaires), tandis que la partie ALORS définit les modifications qu'elle entraîne dans l'état du monde (la

liste des faits enlevés, modifiés et rajoutés). Un opérateur peut être primitif, c'est-à-dire directement exécutable dans le monde, ou alors de plus haut niveau, c'est-à-dire décomposable en opérateurs plus simples. Planifier consiste à déterminer une suite d'opérateurs primitifs qui permet de faire passer le monde de l'état initial à un état qui satisfait le but à atteindre [Regn90].

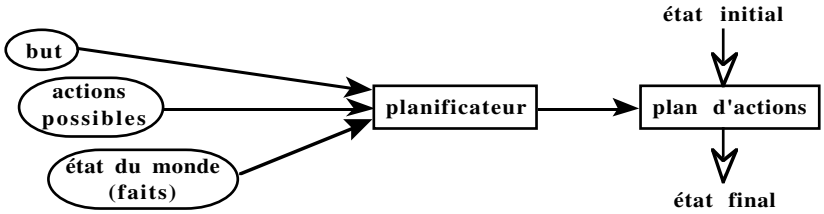


Figure 11 : Planification. En fonction d'un but, un planificateur exploite l'ensemble des actions possibles et une représentation de l'état du monde pour construire un plan d'actions adéquat. Celui-ci doit permettre de transformer l'état initial du monde dans un état final qui satisfait le but.

1.2.1.1. Différents types de planification

La recherche d'une solution en planification peut s'effectuer dans trois différents types d'espace : dans un espace d'états, dans un espace de sous-buts ou dans un espace de plans [Regn90]. Nous donnons ici les caractéristiques essentielles de chaque type de recherche :

- la recherche d'une solution dans un espace d'états part de l'état initial (et/ou de l'état but) et applique successivement des opérateurs possibles. Ainsi un graphe d'états est développé, ceci jusqu'à ce que l'un de ces états satisfasse le but requis (respectivement corresponde à l'état initial). La suite des opérateurs qui a permis de lier l'état initial et l'état but constitue la solution. Cette méthode de recherche conduit facilement à un nombre important d'états développés,. Celui-ci peut être limitée à l'aide d'heuristiques qui permettent de choisir soit l'état à développer soit l'opérateur à appliquer. Un exemple est le choix de l'opérateur à l'aide de l'analyse des fins et des moyens dans GPS [Newe&63], [Erns&69].
- la recherche de solutions dans l'espace des sous-problèmes repose sur la définition de règles de décomposition de but en sous-buts. La recherche de la solution se fait par application de ces règles, en supposant a priori que les buts sont indépendants. Mais, ceci n'est pas toujours garanti et l'interaction entre buts est possible : la résolution d'un but peut influer sur la résolution des autres. Il peut arriver par exemple, qu'un fait accompli pour la résolution d'un premier but doive être détruit pour la résolution d'un second. Un exemple de ce type de planification est STRIPS [Fike&71].
- la recherche dans un espace de plans utilise des plans partiels de résolution connus et pré-définis pour les différents problèmes possibles. Ces plans sont partiels dans la mesure où, soit des opérateurs sont non-primitifs, soit des variables sont non-valuées, soit l'ordre d'exécution des opérateurs n'est pas complètement spécifié. Ce plan partiel est précisé au fur et à mesure, jusqu'à ce qu'on arrive à un plan complètement spécifié, composé d'opérateurs primitifs.

Le dernier type de planification est le plus structuré et nous semble le plus proche du raisonnement humain. Etant donné que notre objectif global est de modéliser la résolution de problèmes en coopération système-utilisateur, nous allons restreindre la suite de la

discussion à la planification par recherche dans un espace de plans. Deux approches importantes et non-exclusives existent ici : la planification non-linéaire et la planification hiérarchique.

La planification non-linéaire essaye d'imposer toujours le minimum de contraintes au plan développé. Elle ne présuppose en particulier aucun ordre d'exécution des opérateurs. L'ordre d'exécution des actions du plan est déterminé au fur et à mesure de l'avancement du processus de planification (figure 12). L'avantage de ce type de planification est qu'en général le plan résultant n'est que partiellement ordonné, et peut donc contenir des actions qui peuvent être exécutées en parallèle. Le désavantage est qu'étant donné que l'ordre définitif d'exécution des actions peut changer jusqu'au dernier moment, la phase de planification doit être complètement terminée avant de pouvoir entamer son exécution [Regn90].

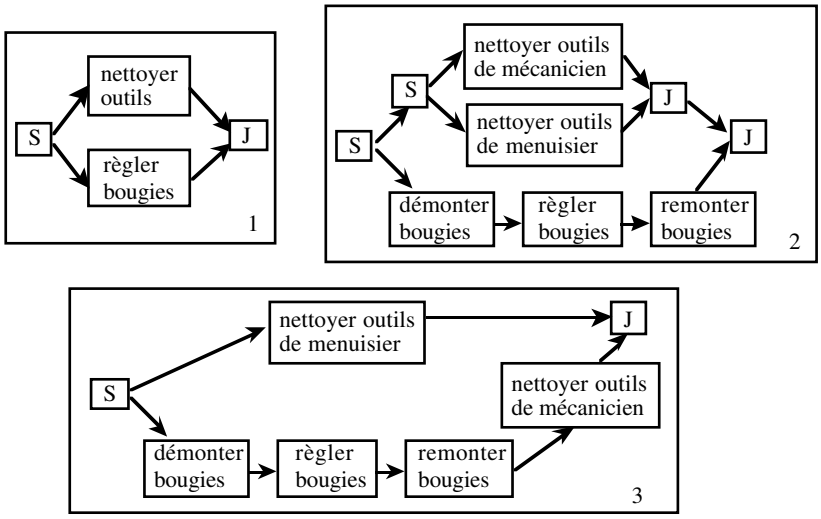


Figure 12 : Exemple de planification non-linéaire. Dans NOAH [Sace75], un plan est représenté par un réseau dirigé d'actions. Des états initiaux (finaux) communs à plusieurs actions sont représentés par des noeuds S (J respectivement). Le plan 1 consiste à nettoyer les outils et, en parallèle, à régler les bougies. Dans la première phase de planification, chacune de ces actions est précisée. Ainsi on obtient le plan 2. Dans la deuxième phase de planification, grâce aux listes d'ajouts et de retraits (qui ne sont pas représentées ici), le planificateur s'aperçoit que les actions concernant les bougies utilisent et salissent les outils de mécanicien. Ceux-ci doivent donc être nettoyés en dernier (3).

Cependant, pour une résolution de problèmes menée de façon coopérative entre un système et son utilisateur, il nous semble important de pouvoir intercaler des phases de planification et des phases d'exécution. Ceci permet par exemple à l'utilisateur d'interpréter des résultats partiels obtenus par une première phase d'exécution, et de remettre en cause, en fonction de ceux-ci, la suite du plan, qui peut être seulement partiellement établie (cf. section 1.2.2.). Par conséquent, nous n'allons pas discuter plus en détail ici de la planification non-linéaire.

La planification hiérarchique est caractérisée par le fait qu'elle construit le plan d'actions à des niveaux successifs de détail ou d'abstraction. La motivation principale, qui a conduit à cette approche, est la réduction de l'espace de recherche à chaque étape d'affinement du plan, donc un gain d'efficacité. Mais la notion d'abstraction est aussi considérée comme importante pour le processus de résolution de problèmes humain [Poly65]. La hiérarchisation, c'est-à-dire l'introduction de différents niveaux d'abstraction dans le processus de planification, peut se faire de différentes façons. Nous allons présenter rapidement les travaux les plus importants, afin de pouvoir identifier un

type d'approche qui nous semble particulièrement intéressant, car proche de la notion de tâche.

1.2.1.2. Tâches et planification hiérarchique

Nous avons identifié deux façons différentes d'introduire des niveaux d'abstraction dans le processus de planification :

- la hiérarchisation des buts à accomplir
- et la hiérarchisation des opérateurs disponibles.

La première a été mise en œuvre pour la première fois par ABSTRIPS [Sace74]. La hiérarchisation consiste à associer des valeurs numériques aux pré-conditions des règles qui modélisent les opérateurs (figure 13). Une valeur exprime pour chacune de ces pré-conditions sa difficulté à être satisfaite. Le concepteur de la base détermine cette valeur en fonction du nombre d'opérateurs disponibles pour réaliser la pré-condition.

```
( Pick_up (?x)
        Pré-conditions : (2 (on_table ?x)) (2 (clear ?x)) (1 ( hand_empty))
        Ajouts : (holding ?x)
)
```

Figure 13 : Hiérarchisation de la planification dans ABSTRIPS. Les buts à atteindre sont ordonnés par la difficulté de leur réalisation. La planification se fait dans ABSTRIPS en chaînage arrière. Le but *holding ?x* est, selon la règle présentée, remplacé par les pré-conditions associées qui constituent les nouveaux buts. Les valeurs listées avant les pré-conditions correspondent à l'effort à faire pour réaliser chacune d'entre elles. Ici, la pré-condition *hand_empty* est par exemple plus facile à réaliser que les deux autres.

La planification s'effectue ainsi selon différents niveaux : à chaque étape de planification, le système ne prend en compte que les pré-conditions les plus difficiles à satisfaire, c'est-à-dire du niveau le plus élevé. Le plan est ainsi complètement spécifié à ce niveau, comme si les pré-conditions de niveaux plus faibles n'existaient pas. Puis, la planification précise le plan pour les pré-conditions du niveau inférieur et ainsi de suite. Il n'y a ici pas de hiérarchie d'opérateurs, car, pouvant avoir plusieurs faits dans sa liste d'ajout, chaque opérateur peut être utilisé à des niveaux de planification différents.

La hiérarchisation des opérateurs constitue la seconde façon d'introduire des niveaux d'abstraction dans le processus de planification. Elle a été adoptée par exemple par NOAH [Sace75] et MOLGEN [Stef81a,b], [Frie&85]. Elle consiste à définir des opérateurs à différents niveaux de précision. Chaque opérateur fait référence à un ensemble d'opérateurs plus élémentaires, c'est-à-dire définis à un niveau plus précis. Seul les opérateurs de plus bas niveau sont directement exécutables. Avec cette approche, un but à résoudre peut être atteint par un (ou plusieurs) opérateur(s) d'un niveau d'abstraction élevé. Celui-ci n'est pas immédiatement exécutable. Il est remplacé au fur et à mesure de la planification par des opérateurs de plus en plus élémentaires, jusqu'à ce que le plan soit composé uniquement d'opérateurs de bas niveau.

Dans NOAH, la planification s'effectue en fait de façon hiérarchique et non-linéaire (cf. figure 12). Un plan est représenté par un réseau d'actions ou d'opérateurs plus ou moins détaillés et partiellement ordonnés. A chaque étape de résolution ce réseau est développé. Ceci consiste à préciser les actions une par une, en les remplaçant par un sous-réseau d'actions plus détaillées. A l'intérieur de chaque sous-réseau les actions sont partiellement ordonnées, de façon à pouvoir accomplir l'action qui a été remplacée. Dans

une deuxième phase de raisonnement, les interactions entre les actions du nouveau plan plus précis sont déterminées et analysées. Puis, en fonction de cette analyse on ajoute ou on élimine des actions ou on impose des contraintes supplémentaires d'ordonnancement aux actions.

Dans MOLGEN [Frie&85], la planification est hiérarchique et linéaire, c'est-à-dire que les plans d'actions sont toujours ordonnés. Son concept central est celui de squelettes de plan. Celui-ci définit une séquence d'étapes généralisées qui, une fois instanciées par des opérations spécifiques dans un contexte spécifique, vont permettre d'atteindre des buts. Il ordonne ces différentes étapes de résolution et il est supposé garantir une solution correcte si celles-ci peuvent être instanciées avec des techniques élémentaires. Un but principal et des modificateurs possibles pour ce but sont associés à chaque squelette de plan pour pouvoir réaliser aussi des buts légèrement différents.

Des squelettes de plan sont définis pour différents buts et à différents niveaux d'abstraction. Deux phases essentielles sont ici distinguées dans la planification : d'une part la recherche de squelettes de plan appropriés pour la résolution d'un problème et d'autre part leur raffinement, c'est-à-dire leur instanciation avec des techniques de résolution élémentaires du domaine. Si, pour une étape de résolution, aucune technique élémentaire correspondante n'existe, le système essaye d'appliquer un squelette de plan pour atteindre le but recherché. Le processus de raffinement d'un squelette peut donc passer par d'autres squelettes de plan avant d'arriver aux techniques de résolution élémentaires du domaine (figure 14).

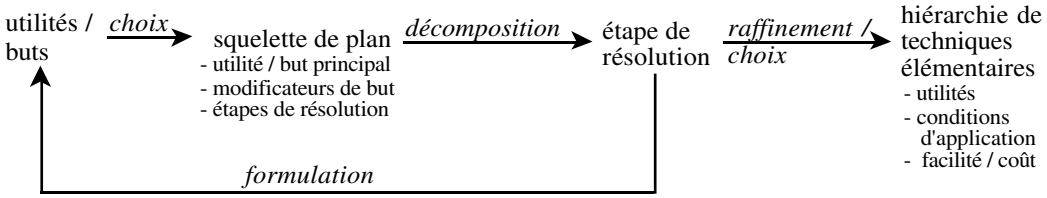


Figure 14 : Planification avec des squelettes de plan dans MOLGEN. A partir d'un but recherché le système choisit un squelette de plan qui répond directement à ce but ou qui peut être modifié pour y répondre. Pour l'exécution du plan chacune de ses étapes doit être raffinée. Le plus souvent une technique (ou une classe de techniques) appropriée répond directement aux besoins et est intégrée dans le plan. Si aucune technique n'est disponible, l'étape de résolution est transformée en un but et le système essaye d'identifier un squelette de plan qui permette de l'atteindre.

La planification hiérarchique avec introduction de niveaux d'abstraction par hiérarchisation des opérateurs correspond en fait à la notion de tâche que nous avons définie à la fin de la section 1.1. Une tâche correspond à un opérateur complexe qui fait référence à un plan d'actions (cf. le réseau d'actions de NOAH ou le squelette de plan de MOLGEN). Chaque action de ce plan peut elle-même récursivement correspondre à une tâche décomposable ou alors directement à une action exécutable. La résolution d'un problème est donc ici aussi modélisée par décomposition récursive de tâches en sous-tâches.

1.2.2. Tâche exécutable

Le contrôle de l'exécution d'un plan, qui est négligé dans le domaine de la planification, est traité dans un autre domaine, celui de la gestion intelligente de bibliothèques de programmes. Ici, il s'agit de résoudre des problèmes par enchaînement adéquat de programmes provenant d'une bibliothèque. Ceci nécessite une phase de planification pour déterminer la suite des programmes à exécuter.

La connaissance sur la sémantique d'un programme est en général limitée. C'est pourquoi il n'est pas possible d'utiliser des programmes comme actions élémentaires à partir desquelles un plan d'actions pourrait être *composé*. Ici, il s'agit plutôt de mettre en œuvre des stratégies existantes, c'est-à-dire de *décomposer* un problème en sous-problèmes, jusqu'au niveau élémentaire des programmes à enchaîner (figure 15). C'est pourquoi l'approche de planification hiérarchique a été adoptée. Elle permet justement la définition de stratégies de résolution bien connues à différents niveaux de complexité et de décomposition.

planification hiérarchique	***gestion intelligente de bibliothèques de programmes***
but	**problème**
opérateurs complexes	**stratégies bien identifiées**
opérateurs élémentaires	**programmes**

Figure 15 : Gestion intelligente de bibliothèques de programmes. Les buts de la planification hiérarchique correspondent aux problèmes à résoudre, les opérateurs complexes aux stratégies de résolution bien connues, et les opérateurs élémentaires aux programmes de la bibliothèque.

Le pilotage de programmes peut être intéressant dans deux contextes, d'une part pour le développement de systèmes réactifs, c'est-à-dire qui exploitent une bibliothèque de programmes pour résoudre les problèmes qui se posent en fonction de l'évolution de leur environnement, d'autre part pour aider un utilisateur à exploiter au mieux une telle bibliothèque. Etant donné que nous nous intéressons en particulier à la coopération système-utilisateur, c'est le second que nous allons présenter plus en détail. A l'origine de ces approches sont toutes les difficultés que peut avoir un utilisateur non habitué à utiliser des bibliothèques de programmes pour résoudre ses problèmes. Leurs objectifs sont de conseiller à son utilisateur les programmes à appliquer, de l'aider à les exécuter, d'inhiber leur mauvaise utilisation et de gérer de façon automatique leur enchaînement pour la résolution de problèmes plus complexes.

A titre d'exemple, deux systèmes nous semblent ici particulièrement intéressants : OCAPI [Clém90], [Thon&93], et SCAI [Rech&92], [Ponc&91]. Les deux ont une architecture générale, c'est-à-dire qu'*a priori* ils ne sont pas restreints à un domaine d'application précis, bien qu'à l'origine, ils aient été développés pour des domaines spécifiques, OCAPI pour le pilotage d'algorithmes de traitement d'images et SCAI pour le calcul scientifique. Une modélisation similaire, par planification hiérarchique, a aussi été utilisée dans le cadre de la modélisation de tâches médicales [Cauh91][1]. Nous allons présenter ici d'abord les caractéristiques essentielles des formalismes de représentation des connaissances adoptés par ces systèmes et ensuite montrer comment le processus de résolution y est géré.

1.2.2.1. Comment représenter les connaissances

Différents types de connaissances interviennent dans la résolution de problèmes par pilotage automatique de programmes. Tout d'abord il faut modéliser les problèmes à

[1] Les actions élémentaires ne correspondent dans ce cas pas à l'exécution de programmes, mais à une "collecte de données médicales" par exemple. La planification hiérarchique a ici été adoptée pour modéliser des stratégies complexes de traitement médical et pour pouvoir gérer leur exécution.

résoudre. Ensuite il faut identifier les stratégies de résolution possibles et les associer aux problèmes qu'elles permettent de résoudre. Si plusieurs stratégies existent pour un même problème, il faut introduire la possibilité de modéliser ce choix. La résolution de problèmes passe finalement par l'exécution de programmes. Par conséquent, le mode d'emploi de ces programmes doit aussi être formalisé. L'enchaînement de programmes nécessite la définition d'un flot de données entre eux. Certaines valeurs de paramètres peuvent ne pas être indiquées par ce flot de données et dépendre du contexte général de résolution. Pour cela, il faut définir les moyens pour les initialiser. Finalement il faut prévoir un moyen pour vérifier si les résultats obtenus sont corrects. Ces éléments sont intégrés sous différentes formes dans les formalismes de représentation adoptés par OCAPI, SCAI et [Cauh91].

Dans OCAPI les connaissances sont représentées à l'aide de deux formalismes différents : des objets et des règles (figure 16). La base de connaissances intègre sous forme d'objets :

- un *contexte*, décrivant des informations globales sur les images à traiter,
- des *buts*, c'est-à-dire des fonctionnalités de traitement d'images,
- des *opérateurs*, qui définissent des séquences de traitement d'images,
- des *programmes* de bibliothèques,
- des *requêtes*, qui associent un but à atteindre avec les objets à traiter et la qualité des résultats requise.

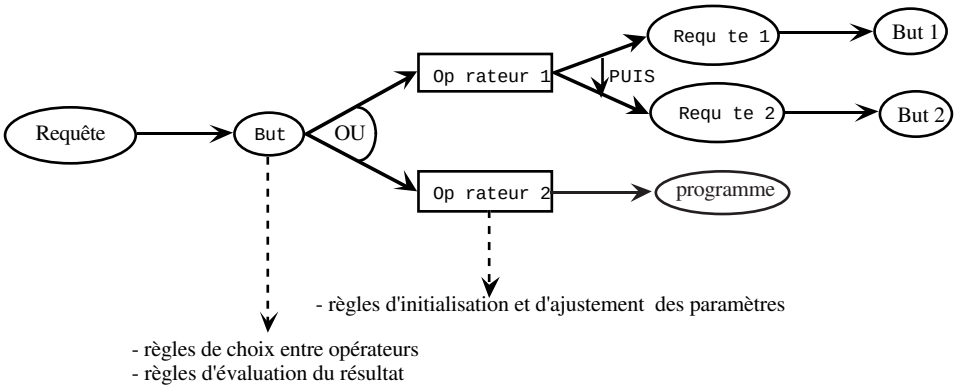


Figure 16 : Eléments d'une base de connaissances OCAPI. Une requête est une demande faite au système et fait référence à un but. Un but peut être accompli par plusieurs opérateurs. Dans une situation précise, des règles associées au but permettent de choisir l'opérateur approprié. Un opérateur peut faire référence soit à un ensemble de requêtes, soit à un programme élémentaire de la bibliothèque. Pour donner des valeurs aux paramètres, des règles d'initialisation et d'ajustement sont attachées aux opérateurs. Les résultats renvoyés par l'opérateur sont évalués au niveau but par des règles associées.

Des paquets de règles de production, associés aux buts et aux opérateurs, permettent :

- le *choix* entre les divers opérateurs,
- l'*initialisation* ou l'*ajustement* des valeurs *de paramètres*,
- l'*évaluation des résultats* obtenus après l'exécution d'un opérateur.

Dans SCAI les connaissances sont représentées de manière uniforme à l'aide d'objets [Uvie&91]. La base de connaissances sépare trois types de connaissances (figure 17) :

- les *tâches* qui décrivent un problème à résoudre et lui associent directement une stratégie de résolution,

- les *méthodes* qui définissent le mode d'emploi des programmes disponibles,
- les *entités* qui décrivent les objets manipulés dans le domaine d'application.

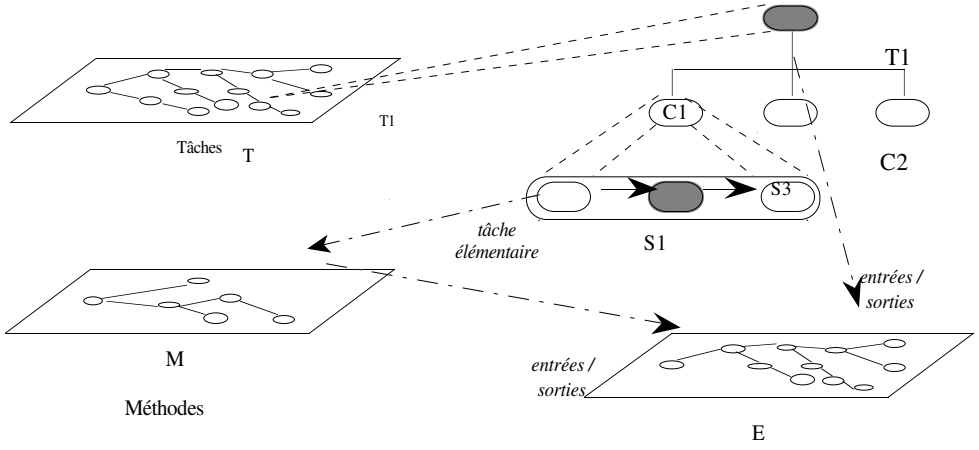


Figure 17 : Eléments d'une base de connaissances SCAI [Uvie&93]. SCAI distingue trois types de connaissances nécessaires à la résolution de problèmes : les tâches, les méthodes et les entités.

Dans SCAI, les connaissances essentielles de résolution de problèmes sont décrite par des tâches. Une tâche est définie par l'ensemble de ses entrées et de ses sorties, par des contraintes sur celles-ci (les pré-conditions et les post-conditions respectivement) et sa stratégie de résolution [Rous88]. Toutes ces connaissances sont définies par des classes et organisées dans des hiérarchies de spécialisation. L'avantage essentiel de cette organisation est la possibilité de structurer les stratégies de résolution possibles pour un problème en fonction du contexte auquel elles sont adaptées [Will91]. Ceci permet de supporter, pour un problème posé dans un contexte concret, le choix de la meilleure stratégie de résolution. Nous avons adopté la même organisation dans le modèle SCARP que nous avons développé au cours de cette thèse. Nous discuterons de ses caractéristiques et de ses avantages particuliers dans le chapitre deux.

Dans le système de [Cauh91], toutes les connaissances de résolution de problèmes sont aussi regroupées sous la notion de tâche. Ici, une tâche est formalisée par une syntaxe de type relationnel. Ce modèle de tâche (cf. figure 18) intègre les notions de :

- *objectif* : le résultat espéré d'une tâche;
- *conditions d'application* : des conditions qui permettent de décider si une tâche peut ou doit être appliquée ou non. [Cauh91] distingue ici quatre types de conditions, qui permettent soit de conclure formellement sur l'indication ou la contre-indication d'une tâche, soit de constituer des arguments en sa faveur ou défaveur;
- *type* : une tâche est soit de type "élémentaire", c'est-à-dire résolue par l'application d'une procédure, soit de type "plan", c'est-à-dire résolue en applicant un ensemble de sous-tâches;
- *corps* : le corps d'une tâche contient un ensemble de données internes. Pour les tâches de type "plan", il indique en particulier les sous-tâches qui interviennent dans sa résolution. Ceci à l'aide de deux opérateurs : ET et OU. La sémantique de ces opérateurs est particulière : ils ne s'appliquent qu'aux tâches qui ne sont pas formellement contre-indiquées. Ceci permet d'intégrer dans des plans des tâches qui ne doivent pas être exécutées dans certains cas (par exemple pour ne pas effectuer un test de grossesse pour une personne de sexe masculin);

- *pré-conditions* : un ensemble de conditions qui doivent être vérifiées pour pouvoir exécuter une tâche;
- *pré-tâches* : un ensemble de tâches qui doivent être exécutées avant de pouvoir exécuter le corps de la tâche elle-même. Les prétâches servent en particulier à rendre vraies les préconditions;
- *conditions de sortie* : [Cauh91] distingue des conditions d'abandon (l'objectif de la tâche ne pourra pas être atteint) et des conditions de terminaison normale;
- *post-tâches* : un ensemble de tâches qui doivent être exécutées après l'exécution d'une tâche et qui permettent en particulier d'évaluer ses résultats.

Pendant l'exécution, chaque tâche est liée à un contexte qui correspond au dossier du malade sous traitement et qui contient toutes les données utilisées.

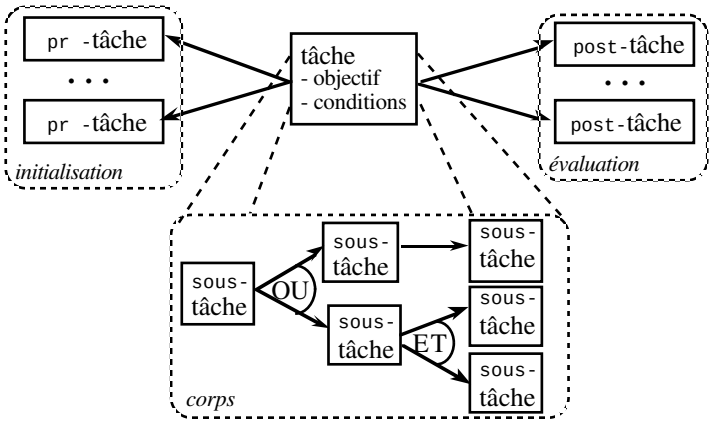


Figure 18 : Modélisation de tâches dans [Cauh91]. Une tâche peut faire référence à d'autres tâches par différents moyens, soit par son corps (dont l'exécution est sensée réaliser l'objectif associé à la tâche), soit en tant que pré-tâche (dont l'exécution préliminaire au corps doit être assurée), soit en tant que post-tâche (dont l'exécution permet de vérifier si l'objectif de la tâche est atteint).

1.2.2.2. Comment gérer le processus de résolution

Dans SCAI comme dans OCAPI ainsi que dans le système de [Cauh91], le formalisme de représentation adopté intègre les connaissances nécessaires pour pouvoir gérer la résolution de problèmes de façon automatique. Mais, même si les stratégies de résolution qui doivent être appliquées sont, en général, bien définies, de la souplesse est nécessaire pendant la résolution de problèmes. Il faut pouvoir adapter la stratégie choisie au contexte dans lequel se déroule la résolution. Dans les systèmes présentés, cette souplesse est atteinte par différents moyens.

OCAPI sépare la définition de problèmes de celle de stratégies de résolution via les notions de but et d'opérateur. Pour résoudre un même problème, plusieurs stratégies peuvent être définies. La définition du problème contient une référence directe à l'ensemble de ces stratégies. Les connaissance de choix entre stratégies sont modélisées à l'aide de règles. Celles-ci introduisent de la souplesse dans la résolution, car elles permettent d'effectuer dynamiquement le choix d'une stratégie, en fonction de la situation concrète rencontrée.

La modélisation de SCAI réunit la définition d'un problème et de sa stratégie de résolution dans seul objet, la tâche. Mais un problème et les stratégies de résolution

associées sont définis à différents niveaux d'abstraction. Ceci permet de structurer les stratégies en fonction de la spécification du problème auquel elles s'appliquent. Cette structure explicite les critères de choix entre stratégies et est exploitée pendant la résolution de problèmes pour adapter la stratégie à la situation courante (cf. chapitre deux).

Le système de [Cauh91] ne fait pas de distinction entre les notions de problème et de stratégie. La notion de tâche regroupe tout, et à un seul niveau d'abstraction. Ce modèle introduit de la souplesse essentiellement par la sémantique spécifique des opérateurs et par la modélisation d'un processus de prise de décision symbolique qui évalue les différentes conditions d'application définies pour une tâche (cf. 1.2.2.1.). Ceci permet de décider dynamiquement lesquelles parmi les tâches référencées par un plan de résolution doivent effectivement être exécutées et lesquelles peuvent être omises.

Malgré ces différences, le processus de résolution de problèmes est, dans les trois cas, globalement géré de la même façon : OCAPI, SCAI et le système de [Cauh91] adoptent une approche de planification hiérarchique et linéaire[1]. Ceci a l'avantage de permettre une alternance de phases d'exécution et de phases de planification. La phase de planification développe à chaque fois seulement le (sous-) problème qui doit être résolu en premier. Ceci jusqu'au niveau élémentaire, c'est-à-dire jusqu'à ce qu'un programme exécutable soit atteint. Celui-ci est ensuite exécuté et ses résultats peuvent immédiatement être pris en compte pour guider la spécification de la stratégie de résolution pour les (sous-)tâches suivantes (figure 19). Ainsi, le processus de résolution de problèmes est d'une part organisé globalement par le développement d'un plan complet à un niveau élevé, il n'est d'autre part que partiellement spécifié, car seul le plan de résolution pour la tâche qui doit immédiatement être résolue est détaillé jusqu'au niveau élémentaire, c'est-à-dire exécutable.

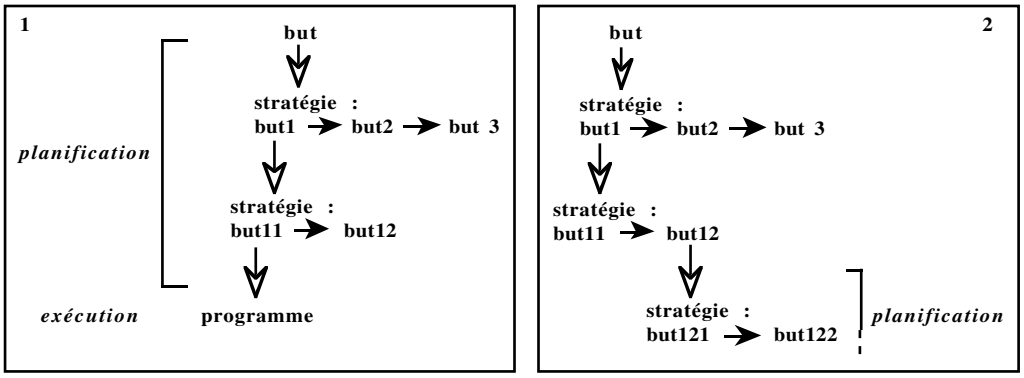


Figure 19 : Alternance des phases de planification et d'exécution. Les stratégies de résolution permettent de décomposer un *but* en sous-buts de plus en plus élémentaires. Dans des étapes successives de planification, à chaque fois le premier sous-but à exécuter est décomposé, jusqu'à ce qu'un programme permette de le résoudre. Celui-ci est ensuite exécuté (1). Ensuite la planification reprend au but suivant (2).

1.2.3. La tâche : un niveau de communication compréhensible pour l'utilisateur et pour le système

La notion de tâche dans le sens de la planification hiérarchique avec introduction de niveaux d'abstraction par hiérarchisation des opérateurs représente donc un niveau de représentation des connaissances exploitable par un système. La figure 20 montre que les

[1] dans le sens où, une fois un plan établi, l'ordre d'exécution de ses différentes étapes n'est plus modifié.

éléments de connaissances nécessaires à la résolution de problèmes ont été modélisés de manière équivalente dans les différents domaines étudiés. Ce sont les problèmes à résoudre, les stratégies de résolution par décomposition existantes qui utilisent au niveau le plus bas les actions immédiatement exécutables. Par ailleurs, ces connaissances sont complétées par la modélisation des concepts, des objets ou des données manipulés pendant la résolution. Même si, pour chacun des domaines étudiés, la tâche n'est pas le moyen unique qui réponde aux besoins particuliers, elle semble être le seul qui permette de répondre à l'ensemble de leurs besoins à la fois.

modélisation conceptuelle	*interaction homme-machine*	*planification hiérarchique*	*gestion intelligente de bibliothèques de programmes*
but	**tâche de l'utilisateur**	**but**	**problème**
tâches	**tâches au niveau intermédiaire**	**opérateurs complexes**	**stratégies bien identifiées**
inférences	**interactions élémentaires**	**opérateurs élémentaires**	**programmes**

Figure 20 : Correspondance des notions utilisées dans différents domaines. Les notions utilisées dans les domaines examinés ont les mêmes significations pour le processus de résolution.

La notion de tâche constitue donc un niveau de communication compréhensible et pour l'utilisateur et pour le système. La différence essentielle est que, pour le second, une plus grande précision des connaissances est nécessaire. Ceci concerne la formalisation (et l'implémentation) de toutes les connaissances nécessaires pour la gestion automatique de l'exécution. Une modélisation basée sur la planification hiérarchique, avec introduction de niveaux d'abstraction par hiérarchisation des opérateurs, présente par ailleurs l'avantage de permettre de gérer le processus de résolution de façon souple et efficace par alternance de phases de planification et d'exécution.

1.3. MODELISATION DE LA COOPERATION SYSTEME-UTILISATEUR

Dans les deux premières parties de ce chapitre, nous nous sommes attachés à l'identification d'un niveau adéquat de communication entre un système informatique et son utilisateur. Cela nous a conduit à la notion de tâche dans le sens d'une modélisation du processus de résolution par planification hiérarchique. Mais notre objectif global est de concevoir un système coopératif d'aide à la résolution de problèmes. Sa caractéristique principale est de coopérer avec son utilisateur pour mener à bien une résolution. Le but de cette coopération est d'exploiter la complémentarité des capacités du système informatique d'un côté, et de celles de son utilisateur de l'autre [Fisc90], [Lemk&90]. Il faut pour cela prévoir une forme de négociation dont les deux partenaires peuvent avoir l'initiative.

Nous allons par la suite exposer rapidement l'intérêt de la coopération pour les deux, pour le système d'un côté et pour l'utilisateur de l'autre. Pour chacun la coopération a un sens différent, en fonction duquel ils s'attribuent des rôles spécifiques. Aussi les fonctionnalités requises pour le système varient en fonction de ces rôles. Chacun de ces points de vue est présenté en détails, en discutant des approches existantes. Toutes les approches que nous avons identifiées adoptent, en fait, une modélisation du processus de résolution de problèmes par tâches, c'est-à-dire conforme à la planification hiérarchique. Cette modélisation est particulièrement intéressante pour la coopération, parce qu'elle peut ainsi être engagée par les deux partenaires, de manière homologue, sur chaque niveau d'abstraction et de décomposition introduit (figure 21).

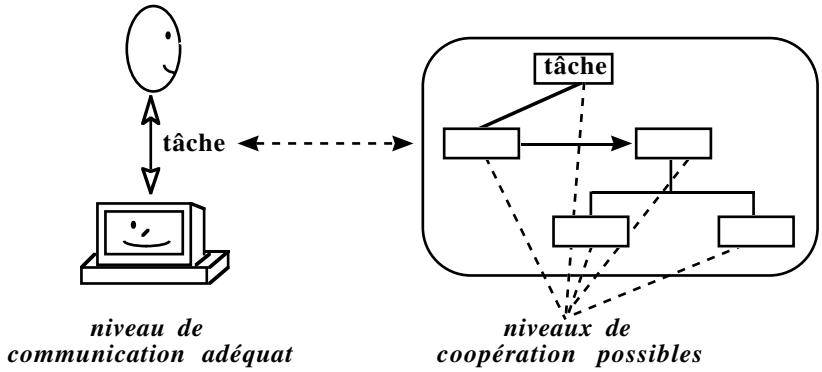


Figure 21 : Tâche : niveau de communication et de coopération adéquat. Les tâches représentent non seulement un bon niveau de communication, mais permettent aussi d'établir une coopération système-utilisateur sur tous les niveaux d'abstraction et de décomposition introduits par la modélisation.

1.3.1. Intérêt de la coopération

Une coopération entre le système et son utilisateur bénéficie, en fait, aux deux. En ce qui concerne le système, la coopération lui permet de pallier ses faiblesses, dues à deux raisons :

- il existe des problèmes qu'un système à base de connaissances ne peut et ne pourra jamais résoudre tout seul, car ils impliquent des connaissances non formalisables. Leur résolution nécessite l'intervention de l'utilisateur et un système à base de connaissances est ici incapable de fournir seul une solution [Caho&91].
- les domaines d'application d'un système à base de connaissances sont souvent trop complexes pour pouvoir modéliser de façon exhaustive toutes les connaissances nécessaires à la résolution de problèmes. Une base de connaissances est donc forcément incomplète. L'utilisateur peut avoir des connaissances complémentaires, non présentes dans celle-ci, qu'il faut savoir exploiter [Stol91] [Gutk&91].

A l'utilisateur, la coopération permet de se servir du système à base de connaissances comme outil pour organiser son propre raisonnement :

- il peut déléguer la résolution de "tâches de routine" au système, mais contrôler lui-même les décisions importantes. Le système peut lui permettre ici de créer et de gérer de façon cohérente des hypothèses concernant ces décisions. Ainsi l'utilisateur a la possibilité d'explorer et de comparer différentes versions du processus de résolution [Kant88].
- le système peut constituer une plate-forme d'essai de stratégies de résolution par rapport à de nouveaux problèmes [Delo&92]. Il peut donc être utilisé pour explorer et formaliser de nouvelles stratégies.

La coopération possible avec un système est décrite par la relation entre la quantité de travail cognitif effectué par le système et la quantité de travail cognitif effectué par l'utilisateur [Allp89], [Fisc90] (figure 22). Les deux doivent se partager le travail : chacun prend une partie des responsabilités concernant le processus de résolution de problèmes. Ensuite, chacun doit avoir la possibilité de critiquer et de remettre en cause le travail effectué par l'autre.

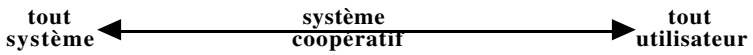


Figure 22 : Système coopératif et relation entre la quantité de travail cognitif effectué par chacun des partenaires.

1.3.2. La coopération du point de vue du système

Du point de vue du système, la coopération sert à combler ses manques de compétence en exploitant celle de son utilisateur. Aux endroits correspondants du processus de résolution, le système va faire intervenir l'utilisateur. Le système contrôle ainsi la coopération, dans le sens où il attribue une partie bien définie des responsabilités concernant le processus de résolution à l'utilisateur. Celles-ci sont :

- les responsabilités pour lesquelles l'utilisateur seul est compétent
- et celles pour lesquelles il est considéré comme plus compétent que le système.

Les premières sont toujours les mêmes, celles pour lesquelles le système n'a pas de connaissances. Elles peuvent donc être identifiées une fois pour toutes et attribuées de façon définitive à l'utilisateur. Nous en discutons dans la section 1.3.2.1. Les secondes, par contre, ne sont pas toujours identiques, car à la fois le système et l'utilisateur ont ici des compétences. Mais leur pertinence varie en fonction de la situation, de l'utilisateur, etc. Elles doivent donc être évaluées par le système, pour attribuer de façon dynamique la responsabilité correspondante au partenaire le plus compétent. Nous présentons les approches correspondantes dans la section 1.3.2.2.

Différentes responsabilités concernant le processus de résolution de problèmes peuvent, en fait, être attribuées à l'utilisateur. Elles sont situées sur deux niveaux différents et liées d'une part au contrôle de la résolution, d'autre part à la résolution elle-même. En ce qui concerne le contrôle, il s'agit par exemple d'effectuer le choix entre différentes stratégies de résolution possibles, d'évaluer les résultats obtenus par l'exécution d'une stratégie, etc. En ce qui concerne la résolution, il s'agit de mettre en œuvre la stratégie de résolution sélectionnée et de déterminer ainsi, à partir des données, les résultats du problème en question. Toutes ces responsabilités peuvent être exercées soit par le système, soit par l'utilisateur. Si le système n'a aucune connaissance pour les exercer, il peut les attribuer de manière fixe à l'utilisateur.

1.3.2.1. Attribution fixe de responsabilités à l'utilisateur

L'attribution fixe de *décisions de contrôle* à l'utilisateur a surtout été proposée et utilisée dans des systèmes destinés à la résolution automatique de problèmes, mais où certaines connaissances ne pouvaient être formalisées. Concernant le contrôle, les systèmes SCAI, OCAPI et celui de [Cauh91] peuvent servir d'exemple. Dans SCAI, il y a la possibilité d'offrir un choix entre différentes stratégies de résolution possibles à l'utilisateur [Uvie&91]. Dans OCAPI, il est possible de faire intervenir l'utilisateur, pour évaluer la qualité des résultats de traitements [Clém&93]. Dans le système de [Cauh91] toutes les tâches sont exécutées sous le contrôle de l'utilisateur, c'est-à-dire que ce dernier est consulté pour confirmer la nécessité d'exécuter chaque tâche. Dans ces cas, l'utilisateur n'est pas sollicité pour résoudre une partie du problème posé, mais pour prendre des décisions au niveau contrôle à la place du système.

Au niveau de la *résolution même d'un problème*, l'approche courante est de définir des "tâches-utilisateur". Ce sont des tâches pour la résolution desquelles l'utilisateur est compétent. Elles lui sont attribuées, ce qui signifie que le système va lui demander,

pendant le processus de résolution, d'indiquer leurs résultats. Le fait de résoudre une tâche est en fait équivalent avec le fait de contrôler ses résultats. C'est pourquoi ces approches distinguent les tâches et les valeurs (ou résultats) contrôlées par le système de celles contrôlées par l'utilisateur. Ces approches se sont généralement inspirées de KADS [Hick&88].

[Gree&92] définit des tâches-utilisateurs et des "tâches de transfert d'informations". Les premières permettent de distribuer la résolution des sous-tâches stratégiques, nécessaires pour la résolution d'une tâche complexe, entre le système et l'utilisateur. Les secondes servent à localiser les transferts d'informations entre le système et l'utilisateur pendant le processus de résolution, et à définir la direction de ces transferts. Il s'agit de quatre tâches : *recevoir*, *obtenir*, *procurer* et *présenter* une information (figure 23). Elles sont introduites dans la modélisation de la structure de décomposition de tâches en sous-tâches au même niveau que les sous-tâches stratégiques.

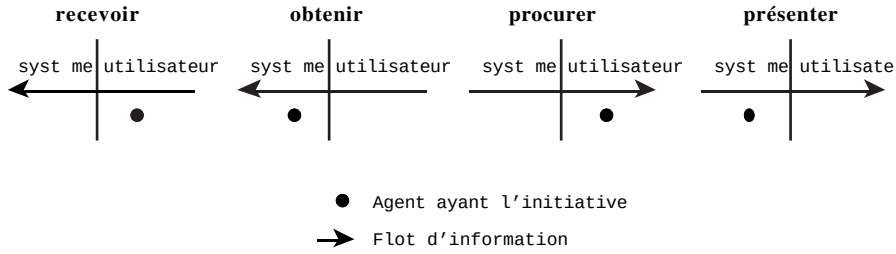


Figure 23 : Quatre types de transfert d'informations entre le système et l'utilisateur. Par chacun des deux partenaires et dans les deux sens, un transfert d'informations peut être initialisé.

L'approche de Stolze [Stol91], [Stol92] est équivalente. Il propose d'ajouter des éléments au niveau inférence de KADS. Il divise les inférences et les méta-classes en tâches-utilisateur / tâches-système et données-utilisateur / données-système respectivement (figure 24).

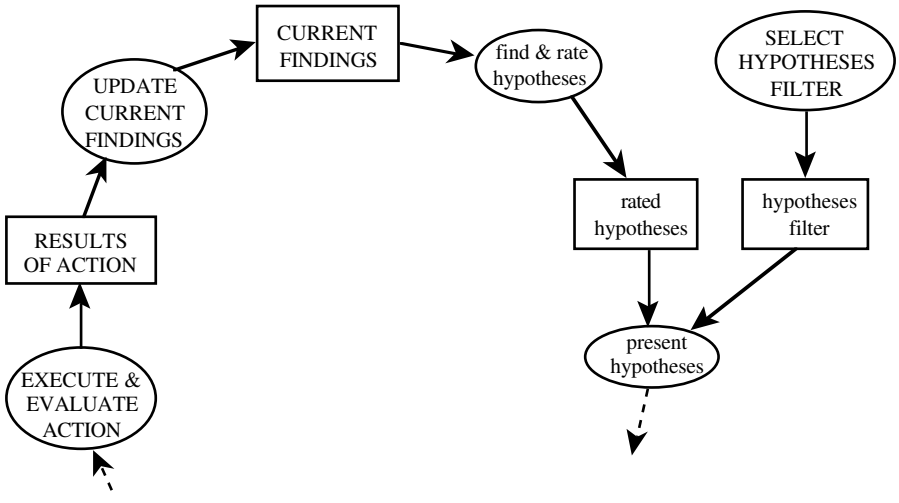


Figure 24 : Une structure coopérative d'inférence. Les tâches-utilisateur (EXECUTE-&-EVALUATE-ACTION, UPDATE-CURRENT-FINDINGS et SELECT-HYPOTHESES-FILTER) et les données-utilisateur (RESULTS-OF-ACTION, CURRENT FINDINGS) sont montrées en majuscules, les tâches effectuées et les données créées par le système en minuscules. Chaque fois que l'utilisateur modifie les CURRENT-FINDINGS le système doit ré-exécuter la tâche find-&-rate-hypotheses.

Mais, la distribution fixe de responsabilités à l'utilisateur ne semble pas satisfaisante : dans le cas idéal, une responsabilité doit à chaque fois être attribuée au partenaire qui est le plus compétent pour l'assumer. Ceci dépend d'une part des compétences de l'utilisateur par rapport à celles du système, d'autre part de la situation actuelle de résolution (des ressources disponibles etc.) (figure 25).

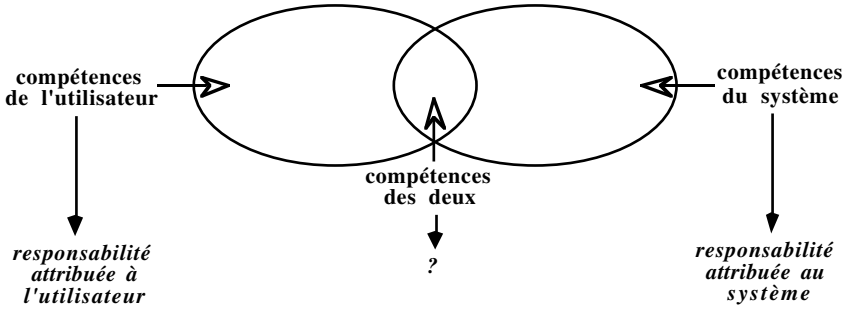


Figure 25 : Compétences du système et de l'utilisateur. Certaines compétences ne sont détenues que par l'utilisateur, d'autres uniquement par le système. Mais les deux peuvent aussi avoir des compétences en commun. Dans ce cas, il faut savoir résoudre le conflit et attribuer la responsabilité à l'un des deux.

1.3.2.2. Attribution dynamique de responsabilités à l'utilisateur

Tandis que Stolze ne prévoit qu'une attribution fixe de tâches et de données à l'utilisateur, Greef et al. veulent prendre en compte le fait que différents utilisateurs ont différentes compétences, qu'ils ne sont donc pas toujours capables d'accomplir les mêmes tâches. Pour cela, ils permettent d'introduire différents types d'utilisateurs. Pour une même tâche, décomposée en sous-tâches, différentes allocations de tâches peuvent être définies pour ces différents types d'utilisateur. Si pour un utilisateur un choix définitif *a priori* d'une allocation n'est pas possible, toutes ces allocations sont énoncées. L'allocation définitive est choisie ou négociée au moment de l'exécution, ne serait-ce qu'en proposant le choix à l'utilisateur. Ainsi le modèle de coopération permet une certaine flexibilité.

Mais un utilisateur ne rentre jamais exactement dans un type pré-défini. Dans le cas idéal, le système devrait déterminer dynamiquement l'attribution de chaque responsabilité à un des deux partenaires. Pour cela, il faut modéliser des connaissances supplémentaires, qui lui permettraient d'effectuer ce choix. Deux directions de recherche ont ici été proposées :

- la modélisation de l'utilisateur, pour évaluer ses connaissances et ses compétences effectives,
- et la modélisation du contexte de la résolution, duquel peut dépendre le choix du partenaire à solliciter.

[Fisc90] et [Caho&91] proposent de suivre la première direction. Ils estiment qu'un modèle de l'utilisateur est indispensable pour un système coopératif. Mais ce domaine ne semble pas (encore) suffisamment avancé pour être effectivement utilisé.

[Delo93] [Delo92] adopte la seconde possibilité. Elle propose de modéliser indépendamment les compétences de l'utilisateur et celles du système. A chaque compétence sont associés des *contextes favorables* qui permettent de prendre en compte des paramètres de la situation actuelle de résolution pour déterminer si cette compétence (et avec cela l'acteur qui la détient) doit être sollicitée ou non (figure 26). Ces

compétences peuvent non seulement concerner la résolution d'un problème du domaine d'application, mais aussi le contrôle de la résolution, comme par exemple l'évaluation de résultats obtenus.

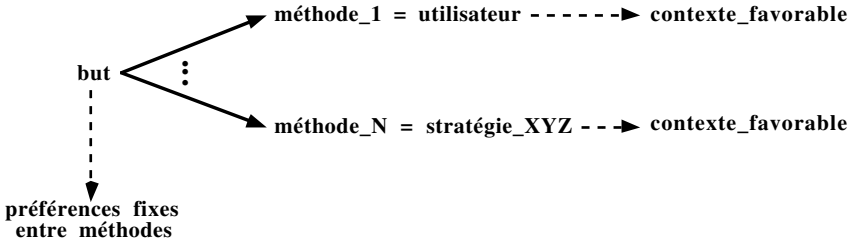


Figure 26 : Modélisation du choix entre compétences du système et de l'utilisateur. Le choix du partenaire qui va prendre la responsabilité se fait de façon indirecte : une méthode (ou compétence) est choisie en fonction des préférences entre méthodes, établies pour chaque but à résoudre, et des contextes_favorables définies pour chaque méthode de résolution. Une de ces méthodes peut être de demander à l'utilisateur de résoudre le but en question.

Mais Delouis ajoute que ceci n'est pas suffisant : aussi l'utilisateur doit avoir une influence sur la distribution des responsabilités. Il doit au moins pouvoir décider du mode général de coopération à adopter par le système. Elle propose pour cela d'identifier des *rôles abstraits* que chaque partenaire peut remplir pendant la résolution de problèmes. Ces rôles correspondent en fait aux différentes compétences de contrôle existantes : le choix-de-la-méthode-à-appliquer, l'évaluation des résultats etc. Par l'attribution de ces rôles au système ou à l'utilisateur alternativement des *modes de coopération* pourraient être introduits. Le système pourrait ensuite gérer la coopération en faisant intervenir l'utilisateur à chaque fois qu'une compétence correspondant à un des rôles qu'il détient est requise.

1.3.3. La coopération du point de vue de l'utilisateur

La possibilité de laisser l'utilisateur décider sur le mode de coopération à adopter se rapproche déjà plus du point de vue de l'utilisateur. Ce n'est plus le système qui attribue des responsabilités à l'utilisateur, mais l'utilisateur qui choisit celles qu'il veut prendre et celles qu'il laisse au système. Mais, du point de vue de l'utilisateur, la coopération a une signification plus forte : elle doit lui permettre de se servir du système comme outil pour organiser son propre raisonnement. Ceci concerne deux aspects, la possibilité pour l'utilisateur de prendre des décisions hypothétiques, et ainsi de créer et de gérer de façon cohérente différentes versions du processus de résolution, et celle d'introduire de nouveaux problèmes et d'explorer de nouvelles stratégies de résolution.

Ainsi [Kant88] propose de permettre à l'utilisateur d'une bibliothèque de programmes d'explorer de façon interactive différents enchaînements et différentes paramétrisations pour la résolution de problèmes. La résolution est ici essentiellement vue comme processus par essais et erreurs, dû aux faits que souvent les données d'un problème sont bruitées et que les connaissances nécessaires à la résolution de problèmes sont incomplètes. L'objectif est donc d'obtenir un système qui permette à l'utilisateur, lors de la résolution, de considérer et de comparer différentes hypothèses concernant la paramétrisation et le meilleur enchaînement des programmes. De nouvelles stratégies de résolution sont créées à partir des programmes en les liant par des flots de contrôle et des flots de données. Elles peuvent être mémorisées et ré-exécutées. La cohérence de l'exécution est garantie dans une certaine mesure par la vérification de contraintes concernant la valeur des différents paramètres d'entrée et de sortie des tâches.

[Delo&92] va dans le même sens. Les formes de coopération envisagées pour cela sont les suivantes :

- l'utilisateur doit pouvoir interrompre le processus de résolution, guidé par le système, pour résoudre une tâche différente et pour revenir ultérieurement à la résolution interrompue.
- l'utilisateur doit avoir la possibilité de remplacer une méthode de résolution sélectionnée par le système par une autre. Ainsi il pourrait en particulier réaliser la résolution lui-même, c'est-à-dire s'attribuer la tâche dans le sens d'une "tâche-utilisateur".
- l'utilisateur doit pouvoir modifier les résultats obtenus par le système.

Aussi dans [Cauh91] l'utilisateur doit pouvoir interrompre l'exécution d'une tâche et (ré-)activer une autre tâche. Comme support, l'utilisateur a pour cela accès à diverses listes qu'il peut modifier avant que la résolution ne reprenne (la liste des tâches en attente d'activation par l'utilisateur, la liste des tâches suspendues par l'utilisateur et la liste des tâches et des objectifs définis dans la base de connaissances). Dans la pratique, il semble que le système, après chaque cycle de résolution, demande à l'utilisateur quelle tâche doit être activée ; aucun détail est donné sur la mise en œuvre de l'interaction système-utilisateur correspondante.

Ici, nous partons de l'idée que l'utilisateur a *plus* de connaissances que le système, et qu'il doit pouvoir les appliquer en utilisant et exploitant en même temps celles modélisées dans la base de connaissances. De ce point de vue, l'utilisateur peut faire faire une partie du travail par le système, en particulier celle pour laquelle il pense que celui-ci est suffisamment compétent. Mais il doit toujours pouvoir intervenir et re-diriger le processus de résolution si ce que fait système ne lui convient pas. De la même façon il doit pouvoir modifier ces propres décisions, s'il s'avère qu'elles n'ont pas été optimales. Le système doit offrir les facilités d'interactions nécessaires pour cela.

1.4. SYNTHESE

L'étude de ces approches permet de faire trois constatations. Premièrement, la modélisation de la résolution de problèmes par planification hiérarchique est capable de répondre aux besoins de tous les domaines liés à la résolution de problèmes en coopération système-utilisateur. Ceci grâce à ses nombreux avantages, à savoir pour l'acquisition des connaissances, pour l'explication, pour la modélisation de l'interaction homme-machine et pour permettre de coopérer avec l'utilisateur. Les tâches dans le sens de la planification hiérarchique constituent donc un niveau adéquat de communication et un niveau adéquat de communication et de coopération entre le système et l'utilisateur.

Deuxièmement, toutes les approches qui ont été présentées distinguent jusqu'à cinq types de connaissances, nécessaires à la résolution de problèmes :

- les connaissances sur les objets manipulés pendant la résolution, appelées souvent "connaissances du domaine".
- les connaissances sur les opérations, méthodes ou inférences élémentaires immédiatement applicables sur ces objets.
- les connaissances sur les problèmes à résoudre.
- les connaissances sur les méthodes ou stratégies de résolution existantes, qui permettent de décomposer un problème en sous-problèmes ou de le résoudre directement en appliquant des opérations élémentaires.

- les connaissances de contrôle qui permettent de choisir et d'adapter les stratégies de résolution aux problèmes à résoudre.

Dans les différentes approches, certains types sont regroupés. Quelques-unes, comme par exemple SCAI, lient directement les connaissances sur les problèmes à résoudre avec les stratégies de résolution possibles. D'autres, comme OCAPI, ne font pas de distinction fondamentale entre méthodes élémentaires ou stratégies de résolution.

Troisièmement, l'examen des approches existantes pour modéliser la coopération système-utilisateur pendant la résolution de problèmes montre que différents aspects doivent être pris en compte. Du point de vue du système, il semble intéressant d'intégrer explicitement les interactions système-utilisateur *a priori* nécessaires pendant la résolution de problèmes dans la modélisation des connaissances. Par définition, certaines actions ou décisions de contrôle qui ne peuvent pas être effectuées par le système, peuvent ainsi être attribuées à l'utilisateur. Du point de vue de l'utilisateur, il faut lui donner la possibilité de structurer son propre raisonnement. Il doit pouvoir diriger le processus de résolution de problèmes. Pour cela, la distribution des tâches et du contrôle de leur exécution doit se faire de manière dynamique, par négociation ou interaction entre les deux partenaires. L'utilisateur doit pouvoir remettre en cause toutes les décisions prises pendant la résolution.

Nous considérons qu'une base de connaissances n'est pratiquement jamais exhaustive, qu'un utilisateur peut, par conséquent, avoir des connaissances plus avancées que le système à base de connaissances. Ainsi nous exigeons d'un système coopératif d'aide à la résolution de problèmes qu'il permette à l'utilisateur d'explorer de façon interactive des stratégies de résolution non pré-définies dans la base de connaissances du système. Ceci répond d'un côté au besoin de souplesse dans la résolution, et par là de coopération. De l'autre, cette capacité peut servir de moyen d'acquisition et de complétion des connaissances contenues dans la base du système.

Que la tâche soit un niveau adéquat pour la résolution de problèmes entre un système et son utilisateur est la conclusion que nous sommes tentée de tirer de l'étude de travaux existants. C'est en proposant le modèle SCARP (Système Coopératif d'Aide à la Résolution de Problèmes) et les expérimentations auxquelles il a donné lieu que nous allons montrer que tel est vraiment le cas. Dans les chapitres suivants nous présentons ce modèle qui correspond au point de vue sur la coopération que nous venons d'introduire. Nous avons tout d'abord défini un modèle des connaissances et un processus de résolution appropriés. Ceci a ensuite été complété par des mécanismes d'interaction adaptés qui permettent de mettre en œuvre la coopération (figure 27).

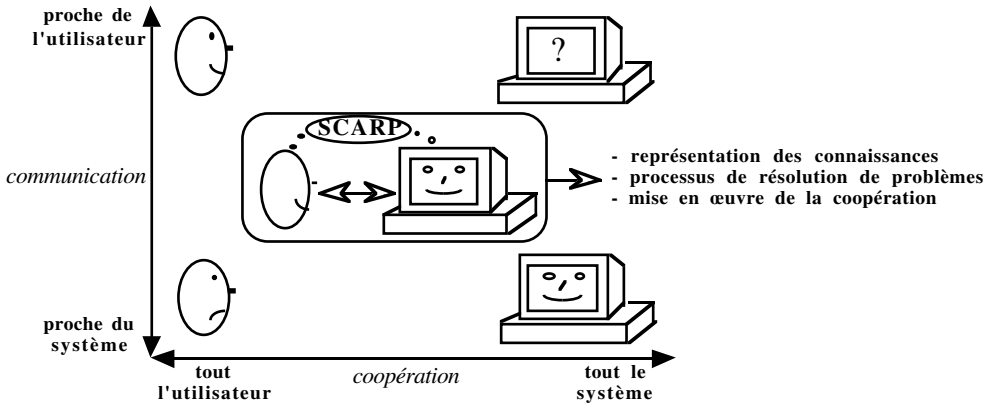


Figure 27 : Système Coopératif d'Aide à la Résolution de Problèmes (SCARP). Le système SCARP doit se situer à un bon niveau de communication et à un bon niveau de coopération entre le système et l'utilisateur.

2. SCARP - REPRESENTATION DES CONNAISSANCES

Le chapitre précédent a mis en évidence l'intérêt de modéliser la résolution de problèmes par planification hiérarchique, c'est-à-dire sur différents niveaux d'abstraction et de décomposition. Avec SCAI nous avions suivi une approche similaire [Will&93]. Mais ce système était destiné à la résolution de problèmes par enchaînement automatique de programmes disponibles et non pas à la résolution de problèmes en coopération avec l'utilisateur.

Aussi, deux aspects importants pour le développement d'un système *coopératif* d'aide à la résolution de problèmes n'étaient pas pris en compte :

- l'intégration dans le formalisme de représentation des connaissances de la modélisation des interactions système-utilisateur *a priori* nécessaires pendant la résolution.
- la définition de mécanismes dynamiques de négociation du contrôle de la résolution pendant l'exécution.

Le premier de ces aspects concerne la représentation des connaissances, le deuxième le processus de résolution de problèmes. Par conséquent, la conception d'un nouveau modèle était nécessaire. Nous allons présenter ici ce modèle, baptisé SCARP. Ce chapitre présente la prise en compte du premier aspect coopératif concernant la représentation des connaissances, le chapitre suivant concernera l'exécution et la négociation dynamique du contrôle. Nous avons choisi un modèle de représentation de connaissances par objets parce que, grâce à sa notion de hiérarchie de classes, la représentation de différents niveaux d'abstraction y est naturelle.

Les principes de base d'une telle représentation et le gestionnaire de bases de connaissances Shirka, qui a été choisi comme couche de base, sont présentés dans la première section de ce chapitre. Les trois éléments de connaissances que nous considérons nécessaires pour la résolution de problèmes dans SCARP — les tâches, les méthodes et les entités — sont identifiés dans la deuxième section.

La partie essentielle des connaissances de résolution de problèmes concerne les tâches. Leur représentation, le modèle de tâches de SCARP, est pour cela précisée dans la troisième section. Ce modèle permet de modéliser, outre les connaissances nécessaires pour la planification hiérarchique, la coopération système-utilisateur par attribution explicite de différentes tâches et décisions au système ou à l'utilisateur.

Avec notre formalisme, chaque tâche est modélisée sans préjuger de son utilisation en tant que sous-tâche dans la résolution de tâches plus complexes. L'intégration de l'exécution d'une tâche dans l'exécution d'une tâche complexe est définie par un contexte d'exécution, décrit dans la dernier section de ce chapitre. La description du modèle dans ce chapitre contient peu d'exemples ; nous en donnerons plus dans le chapitre quatre.

2.1. MODELE A OBJETS

Pour la modélisation des connaissances, nous sommes partis du modèle de représentation de connaissances par objets Shirka [Rech&90]. Shirka est un modèle de connaissances inspiré du modèle des "frames" [Fike&85], mais qui s'appuie sur la

distinction *classe - instance*. Une classe constitue une description générique d'un type d'objet, une instance un objet réel correspondant à cette description.

2.1.1. Notions de base

Dans Shirka, une classe, tout comme ses instances, est définie dans un *schéma* par un ensemble d'attributs. Un *attribut* est lui-même défini par une liste de facettes et une *facette* par une liste de valeurs. Une valeur est soit un schéma soit une référence à un schéma. A chaque attribut est associé via la facette $un ou $liste-de pour des attributs multi-valués un type simple (entier, réel, ...) ou complexe (une classe Shirka).

Dans la définition d'une classe d'objet, la description d'un attribut peut ensuite être complétée par des facettes de restriction de domaine de valeurs ($domaine, $intervalle et $a-vérifier), par des facettes d'inférences de sa valeur ($défaut, $sib-exec et $sib-filtre), ou encore en fixant sa valeur ($valeur). Les attributs avec leurs facettes associées représentent les caractéristiques d'une classe d'objet.

Les attributs d'un objet décrivent différents types d'informations concernant cet objet. Pour structurer ces connaissances, les attributs qui concernent un même type d'information sont regroupés par la définition de *classes d'attributs*. La figure 1 montre les notions de base du modèle à objets Shirka.

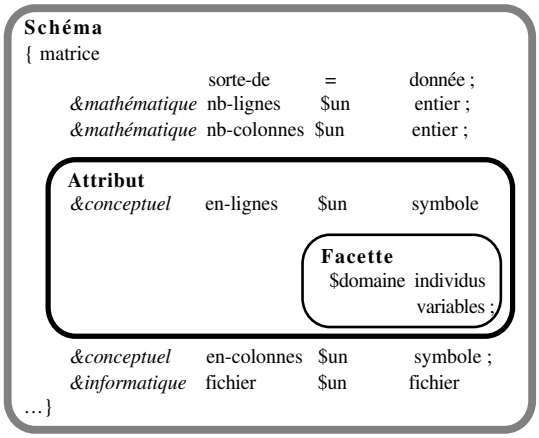


Figure 1 : Notions de base. La définition du schéma de classe *matrice* par l'ensemble de ses attributs (*nb-lignes*, *nb-colonnes* ...). Les attributs concernant un même type d'informations sont regroupés dans différentes classes d'attributs (*&mathématique*, *&conceptuel*, *&informatique*). Les caractéristiques de chaque attribut sont spécifiées par des facettes : le type de sa valeur (facette *$un*) ou l'ensemble des valeurs possibles (facette *$domaine*) par exemple. *Matrice* est une classe d'objet complexe ; car son attribut *fichier* référence une autre classe d'objet (*fichier*).

2.1.2. Organisation en hiérarchies

Les classes d'objet sont organisées en hiérarchies (figure 2). Des classes qui décrivent des concepts différents sont organisées dans différentes hiérarchies. Chaque hiérarchie de classes décrit donc les connaissances concernant un concept spécifique. A chaque niveau dans la hiérarchie de classes, la description du concept concerné est précisée. Ainsi les différents niveaux dans la hiérarchie de classes correspondent à des niveaux successifs d'abstraction dans la description du concept en question.

Une classe donnée dans la hiérarchie domine toutes ses sous-classes plus spécifiques auxquelles elle transmet, par un mécanisme d'héritage, la connaissance portant sur ses

attributs. Dans ces sous-classes, cette connaissance peut être précisée et augmentée par ajout de nouveaux attributs ou de nouvelles contraintes sur les valeurs d'attributs déjà définis dans la sur-classe, mais jamais remise en cause.

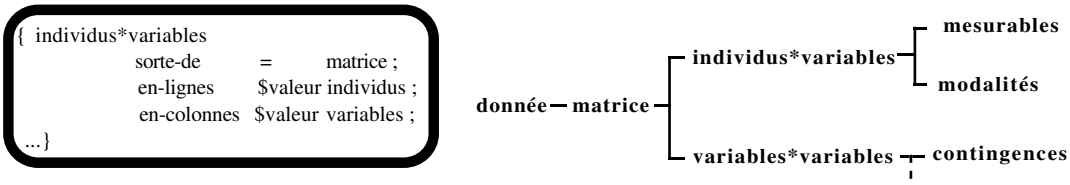


Figure 2 : Hiérarchie de classes. la partie gauche montre la définition du schéma de classe *individus*variables*, sous-classe de *matrice*. La partie droite de la figure montre la hiérarchie de classes correspondante : différentes matrices sont distinguées selon la signification des lignes, des colonnes et du type d'information représentée par ses éléments (des valeurs qualitatives ou quantitatives ou des fréquences).

2.1.3. Classement d'instances

Dans Shirka une instance est créée pour une classe précise. Ainsi elle satisfait toutes les contraintes définies pour celle-ci et récursivement celles définies pour toutes ses sur-classes. Mais l'appartenance d'une instance à une classe précise n'exclut pas son appartenance à des sous-classes plus spécifiques. La ou les sous-classes d'appartenance possibles d'une instance peuvent être identifiées à l'aide d'un mécanisme de classement qui exploite la hiérarchie de classes correspondante.

L'algorithme de classement de Shirka essaie de descendre l'instance dans cette hiérarchie. Il commence par vérifier, pour chaque sous-classe directe de la classe d'appartenance initiale, si les valeurs d'attributs de l'instance satisfont les contraintes supplémentaires définies dans cette sous-classe. Si toutes les contraintes sont satisfaites, cette classe est déclarée sûre. S'il y a des contraintes violées, la classe est déclarée impossible. Si un nouvel attribut ou une nouvelle contrainte sur la valeur d'un attribut apparaît dans une sous-classe et si la valeur de cet attribut reste indéterminée, cette classe est déclarée possible. L'algorithme de classement s'applique récursivement sur toutes les classes sûres ou possibles jusqu'aux feuilles de la hiérarchie de classes. Le résultat de ce classement est la liste des classes possibles, la liste des classes sûres et la liste des classes impossibles (figure 3).

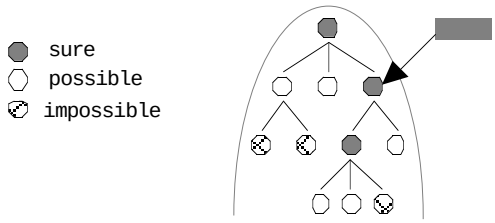


Figure 3 : **Algorithme de de classement.** Pour une instance, l'algorithme de classement détermine les classes sûres, les classes possibles et les classes impossibles dans la hiérarchie de classes correspondante.

Le classement d'une instance peut être restreint à un sous-ensemble d'attributs, appartenant à une ou à plusieurs classes d'attribut précises. Dans ce cas, la valeur des autres attributs n'est pas prise en compte pour la détermination des classes sûres, possibles et impossibles. Dans SCARP nous exploitons cette possibilité pour déterminer

la stratégie de résolution adéquate pour une tâche en fonction seulement des valeurs de ses attributs d'entrée (cf. section 3.1.2.1.). Ce sont en fait les seuls attributs renseignés avant la résolution.

2.1.4. Classement d'instances complexes

L'algorithme de classement est basé sur la vérification que les valeurs des attributs de l'instance classée sont compatibles avec la définition des autres classes dans la hiérarchie correspondante. Si un attribut de l'instance à classer fait référence à une autre instance, il peut être nécessaire de classer aussi cette instance référencée dans sa propre hiérarchie de classes. Cela s'impose, si le type de l'attribut qui fait référence à cette instance a été précisé, c'est-à-dire s'il correspond à une sous-classe du type défini initialement. Pour vérifier si l'instance référencée appartient à cette sous-classe, elle est classée à son tour. Mais ce classement n'est pas exécuté jusqu'au bout, c'est-à-dire jusqu'aux feuilles de la hiérarchie de classes. Il s'arrête dès que les contraintes concernant le type de l'attribut sont vérifiées. Etant donné que chaque instance complexe peut récursivement faire référence à d'autres instances complexes, ce classement peut provoquer de façon récursive d'autres classements partiels dans différentes hiérarchies de classes de la base de connaissances (figure 4).

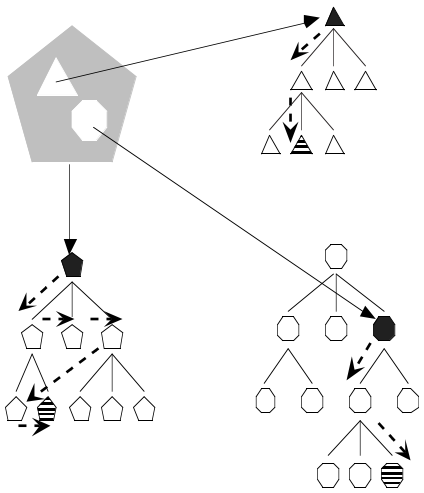


Figure 4 : **Classement d'une instance complexe.** Une instance complexe, représentée ici par un pentagone, doit être classée. Deux de ses attributs référencent des instances d'autres classes, représentées par un triangle et un hexagone. Dans les sous-classes, les types de ces attributs ont été spécifiés ; ainsi, pour classer l'instance représentée par le pentagone, les instances référencées doivent être partiellement classées dans leur propre hiérarchie pour vérifier si elles sont conformes aux types indiqués.

2.1.5. Spécialisation d'instances

Le classement d'une instance, complexe ou non, détermine ses classes d'appartenance sûres, possibles et impossibles dans la hiérarchie de classes correspondante. Suite à ce classement, une instance peut être rattachée à une des classes marquées sûres ou possibles. Cet attachement est appelé *spécialisation*. Etant donné que dans la hiérarchie de classes chaque niveau correspond à une précision du concept décrit, l'attachement d'une instance à une classe plus spécifique que sa classe initiale correspond à une augmentation des connaissances sur cette instance. Par la suite, toutes les connaissances supplémentaires dans la nouvelle classe d'appartenance, en particulier les mécanismes d'inférences, peuvent être utilisées pour cette instance.

En déterminant l'ensemble des classes sûres, possibles et impossibles, le mécanisme de classement guide et limite le choix pour la spécialisation : seules les classes sûres ou possibles et non les classes impossibles peuvent être choisies en tant que spécialisation. Nous utiliserons le mécanisme de classement à cet effet dans SCARP, pour aider au choix de la stratégie de résolution la plus adaptée au problème à résoudre (cf. section 3.1.2.1.).

2.1.6. Méta-niveau

Shirka est basé sur la distinction classe - instance. En fait, dans Shirka, une classe est elle-même une instance d'une autre classe, d'une *méta-classe*. Par défaut, toute classe Shirka est instance de la méta-classe *schéma*. La définition d'autres méta-classes, sous-classes de *schéma*, permet de regrouper des classes Shirka, même si elles font partie de différentes hiérarchies de classes (figure 5). Les classes ainsi regroupées n'ont pas forcément les mêmes attributs, mais elles se ressemblent à un "méta-niveau" plus abstrait.

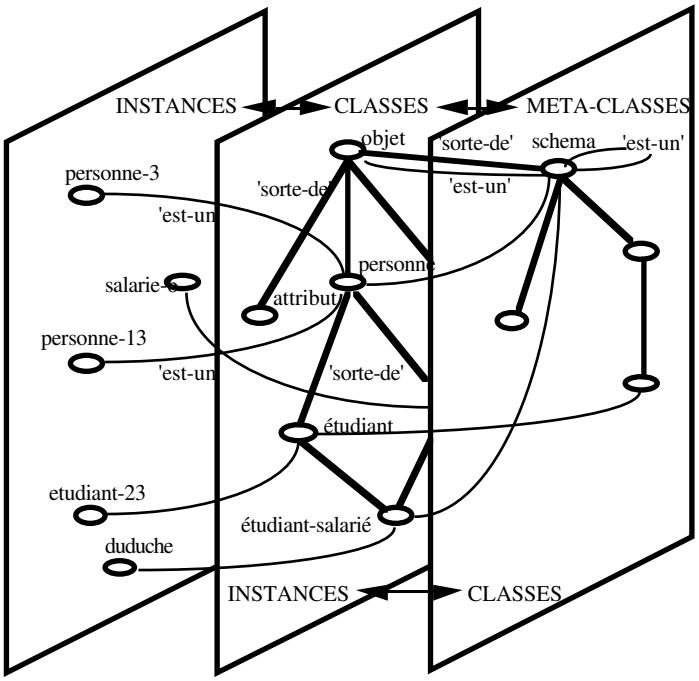


Figure 5 : Instances, classes et méta-classes. Shirka distingue trois niveaux dans la description des connaissances : les instances, les classes et les méta-classes.

Un regroupement de classes par méta-classes peut ensuite être utilisé pour accéder directement à toutes les classes appartenant à une méta-classe spécifique ou pour appliquer à une classe spécifique un type de traitement précis en fonction de sa méta-classe. Souvent, la définition de méta-classes n'est effectuée que pour pouvoir être exploitée de cette manière. Dans ces cas, la définition des méta-classes n'a pas la même signification pour la modélisation des connaissances que la définition des classes. Pour cela, il est préférable de réduire la définition de méta-classes à un minimum, de les introduire seulement si elles ont une signification pour la modélisation.

2.1.7. Intérêt du modèle à objets

Nous avons présenté les caractéristiques principales du modèle à objets et de sa réalisation avec Shirka. Avec le système SCAI nous avions déjà montré l'intérêt d'utiliser Shirka pour la représentation des connaissances de résolution de problèmes [Uvie&91],

[Will&93]. Nous avons décidé d'utiliser à nouveau Shirka pour le développement de SCARP, parce que ces caractéristiques spécifiques permettent de structurer et de représenter les connaissances nécessaires à la résolution de problèmes de façon adéquate, grâce à :

- la séparation des concepts ayant des significations différentes, comme dans notre cas les différents types de connaissances nécessaires à résolution de problèmes (par la définition de hiérarchies de classes différentes).
- la description de chaque concept à différents niveaux d'abstraction successifs (par le biais de la définition de classes - sous-classes).
- la structuration des connaissances rattachées à un concept (par l'introduction de classes d'attribut qui regroupent tous les attributs ayant un même rôle)
- la possibilité de regrouper des concepts ayant une signification a priori différente mais qui doivent être utilisée d'une même manière (par la définition de méta-classes).
- la possibilite de préciser au fur et à mesure les caractéristiques d'un objet (par l'application du mécanisme de classification)

2.2. ELEMENTS D'UNE BASE DE CONNAISSANCES

Nous venons d'introduire le modèle à objets sur lequel repose notre modélisation des connaissances de résolution de problèmes. La résolution de problèmes nécessite différents types de connaissances (cf. chapitre un). Nous rediscutons maintenant rapidement des types de connaissances identifiés et utilisés dans les différentes approches. Ensuite, nous présentons avec les éléments d'une base de connaissances SCARP les trois types de connaissances que nous considérons nécessaires.

2.2.1. Connaissances de résolution de problèmes

Les approches présentées au chapitre un séparent jusqu'à cinq types de connaissances nécessaires pour la résolution de problèmes :

- les connaissances sur les objets manipulés pendant la résolution.
- les connaissances sur les opérations, méthodes ou inférences élémentaires immédiatement applicables sur ces objets.
- les connaissances sur les problèmes à résoudre.
- les connaissances sur les stratégies de résolution existantes.
- les connaissances de contrôle qui permettent de choisir et d'adapter les stratégies de résolution aux problèmes à résoudre. Ce type de connaissances est le plus souvent seulement représenté et utilisé de façon rudimentaire.

Pour pouvoir effectivement résoudre des problèmes, il faut d'abord savoir lier les connaissances sur les problèmes à résoudre avec les stratégies de résolution existantes. Ceci doit permettre, premièrement, d'accéder à l'ensemble des stratégies de résolution possibles pour résoudre un problème donné et, deuxièmement, de choisir celle qui est la plus appropriée.

En général, la description d'un problème est directement liée à l'ensemble des stratégies de résolution possibles. Soit chaque problème contient une référence sur toutes les stratégies possibles pour sa résolution [Clém&93], [Delo&92], soit une notion d'"utilité" indique pour chaque stratégie quels problèmes elle est susceptible de résoudre

[Frie&85]. Pour aider au choix final de la stratégie la plus appropriée à un problème donné, des connaissances complémentaires sont décrites qui permettent d'évaluer l'intérêt de l'application des différentes stratégies à ce problème. Ce choix final (figure 6) est en général effectué en fonction des caractéristiques spécifiques du problème donné. En plus, des caractéristiques générales de l'environnement dans lequel la résolution se déroule ou des préférences indiquées par l'utilisateur peuvent être prises en compte.

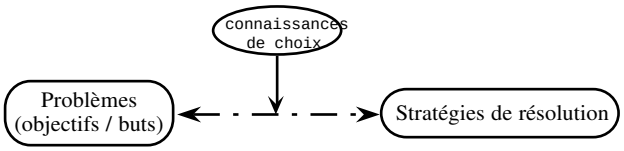


Figure 6 : Résolution de problèmes. Pour résoudre un problème il faut identifier l'ensemble des stratégies de résolution possibles et en choisir la plus appropriée au problème donné. Ce choix est en général effectué par application de connaissances spécifiques qui peuvent être décrites séparément.

Le critère primordial pour le choix est l'évaluation des caractéristiques spécifiques du problème à résoudre. Toutefois, si les différentes caractéristiques ou cas spécifiques des problèmes existants peuvent être identifiés et bien structurés, ceci permettrait de ne plus passer par des connaissances supplémentaires de contrôle ou d'évaluation, mais d'associer directement la stratégie de résolution appropriée à chaque description de problème spécifique. Ainsi il n'y aurait plus besoin de représenter séparément des connaissances de contrôle. C'est ce que nous essayons de faire pour SCARP et c'est pourquoi nous ne distinguons dans SCARP que trois types de connaissances.

2.2.2. Eléments d'une base de connaissances SCARP

Une base de connaissances SCARP contient donc trois types d'éléments. Nous les appelons *tâches*, *méthodes* et *entités*. Les tâches modélisent les problèmes existants et leur associent des stratégies de résolution qui décrivent comment la tâche peut être décomposée en sous-tâches plus élémentaires. Les méthodes définissent les inférences ou actions élémentaires possibles dans lesquelles les tâches sont finalement décomposées, et les entités les objets manipulés dans un domaine (figure 7).

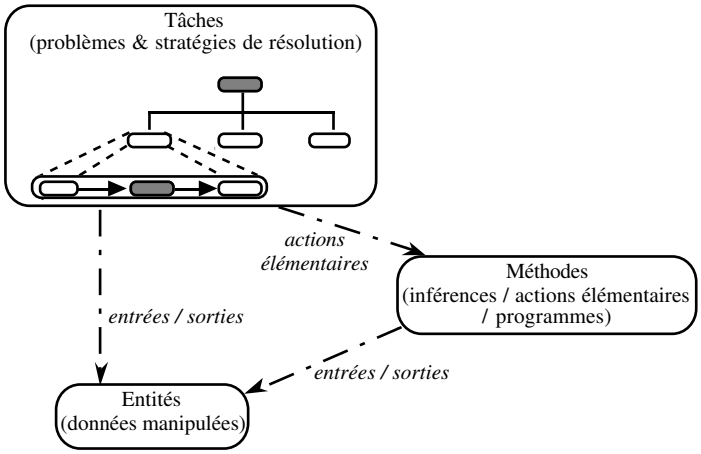


Figure 7 : Eléments d'une base de connaissances SCARP : tâches, méthodes et entités.

Les tâches, les méthodes et les entités sont représentées par des objets. Dans SCARP, pour représenter les tâches et les méthodes, une couche supplémentaire a été définie au-

dessus de Shirka. Par contre, la représentation des entités se fait directement dans Shirka. Nous allons par la suite définir plus précisément chacun de ces concepts.

2.2.2.1. Tâches

Les tâches définissent un problème à résoudre par l'association entre un ensemble d'entrées, c'est-à-dire de données à traiter, et un ensemble de sorties, c'est-à-dire de résultats produits, et par l'indication de leurs caractéristiques respectives. Si cette description est suffisamment précise, elles associent une stratégie de résolution à ce problème (figure 8). Celle-ci décrit comment la tâche concernée peut être résolue par décomposition récursive en (sous-)tâches de plus en plus élémentaires et indique l'ordre de leur exécution. Elle conduit finalement à un enchaînement de méthodes, dont l'exécution mène à la résolution du problème correspondant.

Un problème peut être décrit à différents niveaux d'abstraction. A chaque niveau, les caractéristiques de ce problème sont plus spécifiées et différents cas plus concrets sont distingués. Chaque niveau de précision dans la description du problème, chaque cas différent, correspond dans notre modèle à la définition d'une tâche. Tant que la description du problème concerné reste à un niveau trop général, aucune stratégie de résolution ne peut lui être associée. Mais dès qu'elle décrit un cas bien spécifique, on lui associe directement une stratégie de résolution appropriée.

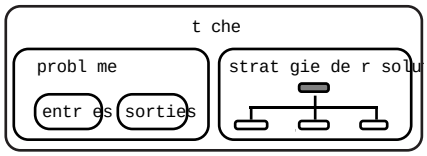


Figure 8 : **Définition d'une tâche.** Une tâche définit un problème par un ensemble d'entrées et de sorties. Si ce problème est suffisamment précis, elle lui associe directement une stratégie de résolution.

Les tâches sont modélisées par des classes d'objet. Comme toute classe d'objet, une classe modélisant une tâche fait partie d'une hiérarchie de classes. La classe racine d'une hiérarchie de tâches modélise le problème correspondant à un niveau très général. La description du problème, c'est-à-dire l'ensemble de ses entrées / sorties et leurs caractéristiques, est précisée de plus en plus dans les sous-classes successives. Si différents cas plus concrets peuvent être distingués pour un problème modélisé par une classe, ils sont modélisés par autant de sous-classes (figure 9).

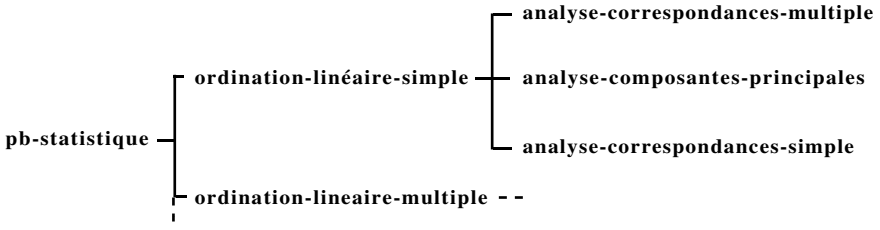


Figure 9 : **Une hiérarchie de tâches.** Les tâches modélisent des problèmes et des stratégies de résolution à différents niveaux d'abstraction. Au niveau très général un problème statistique est défini. Il peut s'agir plus précisément d'une ordination linéaire simple ou multiple. Plus on descend dans la hiérarchie, plus la définition de la tâche est précise.

2.2.2.2. Méthodes

Par sa stratégie de résolution, une tâche se décompose finalement en un ensemble de méthodes. Une méthode est définie par une association entre un ensemble d'entrées, c'est-à-dire de données initiales, un ensemble de sorties, c'est-à-dire de données produites ou déduites, et des moyens immédiats de résolution (figure 10). Des contraintes sur les valeurs d'entrée et les valeurs de sorties peuvent être définies, qui permettent de vérifier si les données fournies en entrée peuvent effectivement être traitées avec la méthode et si les sorties produites ont les caractéristiques requises pour les résultats.

Une méthode permet de résoudre un problème élémentaire, soit par une inférence, soit par l'exécution d'un programme externe. Nous distinguons ainsi (figure 10) :

- les *méthodes internes*, définies par le concepteur de la base de connaissances et complètement intégrées dans cette base, et
- les *méthodes externes* qui font appel à des programmes issus de bibliothèques de programmes, dont l'exécution doit être gérée différemment.

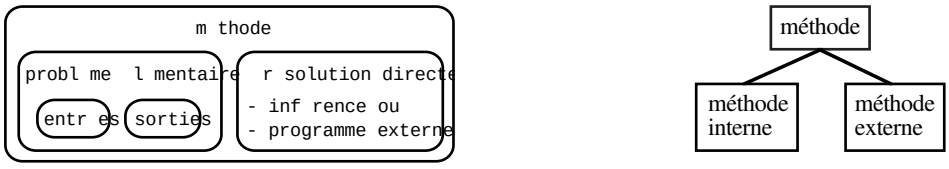


Figure 10 : **Définition de méthodes.** Les méthodes modélisent les problèmes élémentaires pour lesquels des moyens de résolution sont immédiatement disponibles. La partie gauche de la figure montre les composants principaux d'une méthode, la partie droite trois classes pré-définies dans la base de connaissances SCARP. Toutes les méthodes d'une base de connaissances SCARP sont ensuite définies en tant que sous-classes, soit de *méthode interne*, soit de *méthode externe*.

Les méthodes externes sont définies pour intégrer des programmes externes au système et leur mode d'emploi dans la base de connaissances. Par opposition aux méthodes internes, il est donc nécessaire de définir l'interfaçage entre la base de connaissances et ces programmes. Leur exécution peut ensuite être gérée en trois temps :

- la préparation des données pour l'exécution ; par exemple leur écriture dans un fichier, sous le format requis par le programme en question,
- l'exécution du programme, qui peut être gérée de façon asynchrone,
- l'intégration des résultats dans la base de connaissances ; par exemple la lecture du fichier de sortie produit par le programme et la mise à jour de la base de connaissances en fonction des résultats obtenus.

Une méthode, qu'elle soit interne ou externe, est modélisée par une classe d'objet Shirka. Les attributs de cette classe définissent ses entrées / sorties et les informations nécessaires pour pouvoir gérer l'application de la méthode. Des contraintes sur ses entrées / sorties sont exprimées à l'aide des facettes de restriction de valeurs Shirka. Les informations nécessaires pour la gestion de l'exécution sont définies par un ensemble d'attributs spécifiques qui sont différents pour les méthodes internes et les méthodes externes :

- les méthodes internes référencent via leur attribut *nom-fct* une fonction LISP. Celle-ci permet de réaliser directement l'inférence des données de sortie à partir des données d'entrée. La figure 11 montre la syntaxe et un exemple de définition pour une méthode interne.

- les méthodes externes par contre ont trois attributs spécifiques, correspondants aux trois phases d'exécution, *nom-fct-avant*, *nom-exe* et *nom-fct-après*. Elles peuvent avoir d'autres attributs supplémentaires décrivant des informations relatives à l'exécution, par exemple le chemin d'accès au programme ou le temps d'exécution. La figure 12 montre la syntaxe et un exemple de définition pour une méthode externe.

```
{ infère-rapport-signal-bruit
                    sorte-de              =          méthode-interne ;
          &entrée   signal                $un        séquence ;
          &entrée   signal-référence      $un        séquence ;
          &sortie   rapport-signal-bruit  $un        symbole
                                          $domaine   bon mauvais ;
                    nom-fct               $valeur    rapport }
(de rapport (inst)
   (lets ((signal (val? signal))
          (signal-référence (val? signal-référence))
          (rapport-signal-bruit (infère-rapport signal signal-référence)))
   (if rapport-signal-bruit (affect 'rapport-signal-bruit rapport-signal-bruit) () )))
```

Figure 11 : Exemple d'une méthode interne. Cet exemple montre l'inférence du *rapport signal-bruit* à partir d'un *signal* et d'un *signal de référence*. La fonction LISP *rapport* permet d'effectuer l'inférence. Cette fonction renvoit nil, si l'inférence n'aboutit pas. Cette caractéristique des fonctions LISP référencées par les méthodes internes permet au système, de contrôler pendant l'exécution leur échec éventuel. (Les fonction *val?* et *affect*, pré-définies dans Shirka, permettent d'obtenir ou de modifier la valeur d'un attribut. La définition des classes d'attributs *entrée* et *sortie* correspond à leur définition pour les tâches, et sera détaillée dans la section 2.3.2.).

```
{ produit-matrices
                    sorte-de              =          méthode-externe ;
                    lui-même              $a-vérifier {nb-colonnes-1=nb-lignes-2
                                                       mat-1   $var<-   matrice-1 ;
                                                       mat-2   $var<-   matrice-2 } ;
          &entrée   matrice-1             $un        matrice ;
          &entrée   matrice-2             $un        matrice ;
          &sortie   matrice-produit       $un        matrice ;
                    nom-fct-avant         $valeur    créer-fichiers-in ;
                    nom-fct-après         $valeur    créer-matrice ;
                    nom-exe               $valeur    produit-matrice }
```

Figure 12 : Exemple d'une méthode externe. Il s'agit du calcul du *produit* de deux *matrices*. Le prédicat *nb-col-1=nb-lignes-2* permet de vérifier avant l'exécution que le nombre de colonnes de la première matrice est égal au nombre de lignes de la seconde, que le produit peut donc effectivement être calculé. La fonction LISP *créer-fichiers-in* permet de créer des fichiers contenant les valeurs des matrices dans le format requis par l'exécutable *produit-matrice*. La fonction *créer-matrice* permet de créer une instance de la classe *matrice* correspondant au fichier de sortie de *produit-matrice*.

2.2.2.3. Entités

Les entités représentent les objet manipulés dans le domaine d'application, et en particulier ceux utilisés en tant qu'entrées et sorties par les tâches et les méthodes. Leur représentation se fait directement dans Shirka. Les tâches sont organisées en hiérarchies selon la précision avec laquelle elles définissent le problème concerné, c'est-à-dire ses entrées et ses sorties. Il y a ainsi souvent une correspondance dans la structure des hiérarchies de classes modélisant les tâches et des hiérarchies de classes modélisant les entités.

2.2.2.4. Nécessité d'étendre le modèle à objets

Pour représenter les entités, le modèle Shirka est directement adéquat. Pour modéliser les méthodes, il suffit de faire des ajouts mineurs, de pré-définir quelques classes (méthode, méthode-interne, méthode-externe) et d'introduire quelques attributs avec une signification précise (nom-fct etc.). Les connaissances rattachées aux tâches par contre sont plus complexes et nécessitent plus de structuration.

Tandis que les méthodes modélisent des moyens de résolution directement exécutables, les tâches peuvent modéliser des problèmes soit à un niveau abstrait soit à un niveau plus précis en leur associant une stratégie de résolution. Une tâche ne peut pas simplement être définie par ses entrées, ses sorties et sa stratégie de résolution. D'autres connaissances interviennent dans la résolution d'une tâche, comme par exemple des conditions d'exécution, la référence aux tâches à exécuter en tant que sous-tâches, la définition du flot de contrôle sur ces sous-tâches, et — pour un système coopératif — la définition de moyens d'interaction avec l'utilisateur. Nous avons donc défini, au-dessus de Shirka, un modèle de tâches adéquat pour SCARP qui est détaillé dans la section suivante.

Dans SCARP nous définissons les tâches et les méthodes d'une manière indépendante de leur utilisation au sein de tâches complexes. En fait, chaque tâche ou méthode peut être utilisée dans des tâches complexes différentes. Souvent la façon dont son exécution doit être gérée dépend de ce contexte. C'est pourquoi nous avons introduit un type d'objet supplémentaire dans SCARP, le contexte d'exécution, qui est détaillé dans la section 2.4. Il sert à définir toutes les informations nécessaires pour la gestion de l'exécution d'une tâche qui est utilisée en tant que sous-tâche au sein d'une tâche complexe.

2.3. MODELE DE TACHES

Dans la section 2.2.2.1. nous avons identifié quelques caractéristiques essentielles d'une tâche :

- une tâche modélise un problème par l'ensemble de ses entrées / sorties et permet de lui attacher une stratégie de résolution par décomposition en sous-tâches plus élémentaires.
- une tâche modélise le problème concerné à un niveau précis d'abstraction. Les hiérarchies de classes d'objets représentant les tâches dans la base de connaissances matérialisent ces niveaux d'abstraction.

Nous détaillons maintenant de façon plus précise le modèle de tâche que nous avons défini pour SCARP. Nous discutons des aspects suivants :

- la distinction de tâches modélisant des problèmes abstraits et de tâches modélisant des problèmes précis auxquels sont associés des stratégies de résolution,
- la structuration des différentes connaissances rattachées à une tâche,
- la définition de la stratégie de résolution et la prise en compte de la coopération avec l'utilisateur.

Nous allons montrer l'importance de chacun de ces aspects et indiquer les moyens qui ont permis de les réaliser à partir de Shirka. Ces ajouts consistent en la définition de méta-classes, dans la définition de classes d'attributs, et dans la pré-définition de certaines classes et de certains attributs.

2.3.1. Distinction tâches abstraites - tâches exécutables

Tandis qu'à un niveau abstrait et trop général de description, une stratégie précise de résolution ne peut être définie, à un niveau faible d'abstraction le problème correspondant est décrit de manière suffisamment précise pour lui associer une stratégie adéquate. Ainsi deux catégories de tâches s'imposent, les tâches abstraites, non directement résolubles, et les tâches précises, résolubles via une stratégie de résolution. Pour distinguer les tâches de ces deux catégories nous avons introduit deux méta-classes Shirka dans notre modèle (figure 13) : *tâche exécutable* et *tâche-à-spécialiser*.

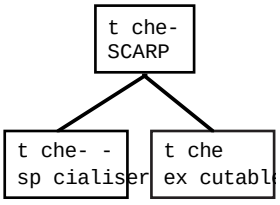


Figure 13 : La hiérarchie des méta-classes de tâches. Ces méta-classes distinguent les *tâches à spécialiser* des *tâches exécutables*.

Cette distinction est exploitée pendant la résolution : seulement une *tâche exécutable* peut être directement résolue. Pour résoudre une *tâche-à-spécialiser*, elle doit d'abord être suffisamment spécifiée, soit par le système soit en interaction avec l'utilisateur. Comme nous le montrerons dans la section 3.1.2., cette spécification se fait par l'application du mécanisme de classement et finalement par le choix d'une spécialisation.

2.3.2. Définition d'une tâche

L'ensemble des entrées et des sorties et la stratégie de résolution constituent les attributs essentiels d'une classe modélisant une tâche. Mais les connaissances rattachées à une tâche sont plus complexes : pour l'exécution d'une stratégie de résolution d'un problème, il y a souvent besoin, outre les données globales du problème, de paramètres supplémentaires. De plus, l'exécution d'une stratégie de résolution peut fournir seulement des sorties brutes qui doivent subir un traitement final pour constituer les résultats définitifs. Ainsi, en dehors des sous-tâches utilisées dans sa stratégie de résolution, d'autres traitements peuvent être attachés à une tâche, pour aider à déterminer ses paramètres supplémentaires d'entrée ou ses résultats définitifs. Par ailleurs, des pré- et des post-conditions peuvent être définies pour l'exécution d'une stratégie de résolution.

Pour modéliser et structurer toutes ces connaissances relatives à une tâche dans SCARP, nous avons introduit des classes d'attribut dans Shirka. Ces classes d'attributs regroupent les différents attributs concernant des informations qui ont un même rôle pour une tâche. Ce sont les classes d'attribut *entrée* (*globale* ou *stratégique*), *sortie* (*globale* ou *stratégique*), *condition*, *pré-*, *post-*, *sous-tâche* et *tâche de visualisation*. La figure 14

montre la hiérarchie des classes d'attributs introduits dans SCARP pour la définition de tâches. La signification des différents attributs ou groupes d'attributs est la suivante :

- L'ensemble des attributs d'*entrée globale* et de *sortie globale* d'une tâche définissent le problème à résoudre. Les *entrées globales* représentent les données initiales de ce problème, les *sorties globales* les résultats attendus.
- Les *entrées stratégiques* définissent des paramètres d'exécution pour la stratégie de résolution associée à la tâche (des taux d'erreur par exemple). Leurs valeurs dépendent en général des valeurs des *entrées globales* (le taux d'erreur dépend de la qualité des données initiales). La valeur d'une *entrée stratégique* peut être déterminée soit par des moyens d'inférence de Shirka (par la définition d'une valeur par défaut par exemple), soit par l'utilisateur. Pour aider l'utilisateur à choisir une valeur pour une entrée stratégique, une *pré-tâche* peut lui être associée. Dans ce cas, l'utilisateur peut activer cette *pré-tâche.* Celle-ci peut soit directement proposer une valeur pour l'entrée stratégique en question, soit calculer des valeurs associées qui permettront à l'utilisateur de déterminer cette valeur.
- Les *sorties stratégiques* correspondent aux sorties brutes d'une tâche. Elles nécessitent un post-traitement qui permettra de déterminer les valeurs des *sorties globales*. Ces post-traitements sont définis par des *post-tâches* qui sont attachées aux *sorties globales* dont ils déterminent la valeur. Ils peuvent servir à faire une transformation de données numériques en données symboliques, ou encore à faire pour les *sorties globales* une synthèse de l'ensemble des données représentées par les *sorties stratégiques*.
- Des *tâches de visualisation des entrées ou des sorties* (cf. figure 15) peuvent être définies pour pouvoir donner à l'utilisateur une vue synthétique de l'ensemble des entrées ou des sorties d'une tâche. Si les sorties d'une tâche correspondent aux caractéristiques d'une courbe par exemple (des coordonnées, ses maxima, ses points d'inflexion etc.), une tâche de visualisation peut permettre de visualiser graphiquement cette courbe et ses caractéristiques. Les tâches de visualisation sont définies par le concepteur de la base de connaissances pour une application précise. Elles peuvent être activées soit par le système, pendant la résolution de problèmes (cf. section 2.4), soit à tout moment par l'utilisateur.
- Des *conditions d'exécution* (cf. figure 15), plus précisément des *pré-* et des *post-conditions*, sont attachées à une tâche. Elles ont été introduites dans la définition des tâches pour expliciter clairement les conditions d'exécution[1]. Les pré-conditions définissent des contraintes sur les valeurs de l'ensemble des entrées (globales et stratégiques). Elles permettent de vérifier, une fois toutes ces valeurs fournies, si la stratégie de résolution associée à la tâche est applicable. Les post-conditions définissent des contraintes sur l'ensemble des sorties (globales et stratégiques). Les sorties satisfont ces contraintes si l'exécution a été couronnée de succès. Les *conditions d'exécution* permettent ainsi d'un côté d'empêcher une application incorrecte d'une stratégie, de l'autre de détecter des échecs d'exécution.

---

[1] Pour les méthodes, ces conditions sont exprimées de façon implicite, par la définiton de prédicats Shirka. Si un seul attribut est concerné, un prédicat est directement attaché à la définition de celui-ci. Ceci correspond en fait plutôt à une précision du type de cet attribut qu'à l'explicitation d'une pré- / post-condition. Si plusieurs attributs sont concernés par un prédicat, celui-ci est défini globalement pour la classe. L'ensemble de ces pré- et des post-conditions est dans ce cas regroupés et associés à la classe par l'attribut *lui-même* (cf. figure 12) ; aucune distinction est faite entre la définition de pré- et postconditions (et d'autres contraintes définies pour la classe).

- L'attribut *stratégie* définit les étapes de résolution (*sous-tâches*) nécessaires pour résoudre la tâche (cf. section 2.3.3.). Il décrit en même temps le flot de contrôle qui ordonne ces étapes, c'est-à-dire leur enchaînement pendant le processus de résolution.
- Pour chaque étape de résolution un attribut séparé de type *sous-tâche* décrit, via un *contexte d'exécution*, quelle tâche permet de réaliser cette étape, et comment son exécution s'intègre dans la résolution de la tâche globale (cf. section 2.4.). Chaque sous-tâche peut être soit complexe à son tour, soit élémentaire, i.e. faire référence à une méthode.

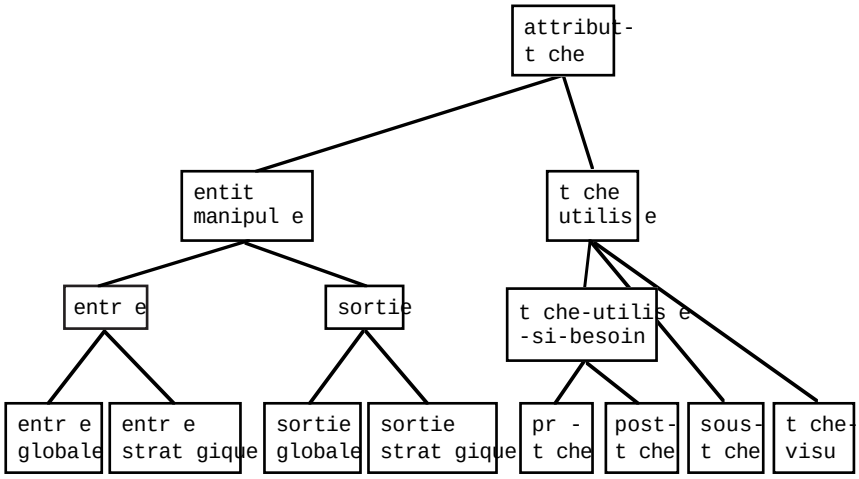


Figure 14 : Classes d'attributs introduites pour la définition de tâches. Différentes classes d'attributs permettent de regrouper tous les attributs d'une tâche avec un même rôle, par exemple toutes les entrées, toutes les sorties, etc.

Une tâche est donc définie par un ensemble d'informations différentes. La classe *tâche* regroupe les attributs de nom fixe qui sont pré-définis pour chaque tâche (figure 15). Elle constitue la racine de toutes les hiérarchies de classes modélisant des tâches dans une base de connaissances SCARP. La figure 16 en donne un exemple.

{ tâche	est-un	=	tâche-SCARP ;
	sorte-de	=	objet-SCARP ;
&condition	*pré-condition*	$un	booléen ;
&condition	*post-condition*	$un	booléen ;
	stratégie	$liste-de	objet ;
&tâche-visu	*visu-entrées*	$un	contexte-visualisation ;
&tâche-visu	*visu-sorties*	$un	contexte-visualisation }

Figure 15 : Définition de la classe tâche dans SCARP. Les attributs *pré-condition, post-condition, stratégie, visu-entrées* et *visu-sorties* représentent l'ensemble des attributs présents dans chaque tâche. Pour la définition d'une tâche précise, une sous-classe de la classe *tâche* est définie. Cette définition est complétée par des attributs des classes entrée et sortie globales et stratégiques, pré, post-et sous-tâches.

```
{ analyse-composantes-principales
                              sorte-de        =          ordination-linéaire-simple ;
    &entrée-globale           données         $un        matrice-mesurable
                                              $com       “données initiales - n lignes, p colonnes” ;
    ...
    &sortie-stratégique       vals-propres    $un        vecteur ;
    &sortie-globale           résultat        $un        évaluation-analyse
                                              $post-tâche  évaluer ;
                              stratégie       $valeur    ( séquence commencer ... terminer )
    &sous-tâche               commencer       $un        { contexte ... } ;
    ...
    &sous-tâche               terminer        $un        { contexte ... } ;
    &post-tâche               évaluer         $un        { contexte ... } ;
    &tâche-visualisation visu-sorties         $un        { contexte-visualisation ... } }
```

Figure 16 - Définition de la tâche *analyse-composantes-principales*. La figure montre en gras des nominations spécifiques de SCARP. La tâche *analyse-composantes-principales* a une *entrée globale*, une matrice qui contient les *données* à analyser, et une *sortie globale*, le *résultat* de l'analyse. Ce *résultat* est obtenu en appliquant la *post-tâche évaluer* qui effectue une synthèse des sorties stratégiques obtenus par l'exécution de la *stratégie* de résolution. Cette stratégie indique que la résolution de *analyse-composantes-principales* passe par l'exécution d'une *séquence* de sous-tâches. Les informations nécessaires pour l'exécution de chacune de ces sous-tâches, ainsi que pour la *post-tâche évaluer* et la *tâche de visualisation* des *sorties*, sont définies dans des *contextes* d'exécution correspondant.

La figure 17 montre les liens qui existent entre les attributs des différentes classes d'attribut. Les attributs de la classe *tâche* (*pré-tâche, post-tâche, sous-tâche* et *tâche-visu*) et *condition* utilisent les valeurs des attributs des classes *entrée* ou *sorties* (globale ou stratégique). A partir de ces valeurs, les *pré-tâches*, les *post-tâches* et les *sous-tâches* permettent de déterminer la valeurs d'autres attributs. Une *pré-tâche* par exemple permet de déterminer la valeur d'une *entrée stratégique* à partir de valeurs d'*entrées globales.* Une *tâches-visu* permet de visualiser l'ensemble des entrées ou des sorties d'une tâche et les *conditions* de vérifier des contraintes définies sur ces ensembles.

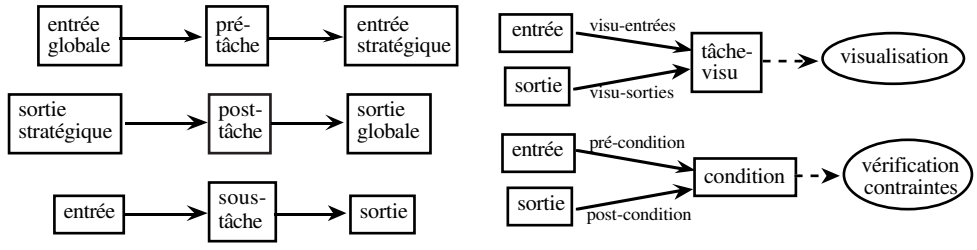


Figure 17 : Liens entre les différentes attributs d'une tâche. Les tâches auxquelles font référence les attributs des classes *pré-, post-* et *sous-tâche* prennent en entrée respectivement des *entrées globales,* des *sorties stratégiques* ou des *entrées*. Elles produisent en sortie respectivement des *entrées stratégiques*, des *sorties globales* et des *sorties*. Les tâches de visualisation*, visu-entrées et visu-sorties,* permettent de visualiser graphiquement l'ensemble des *entrées* ou des *sorties*. Les conditions, les *pré-condition* et les *post-condition*, permettent de vérifier si la stratégie de résolution associée est applicable aux *entrées* fournies ou si elle a produit des *sorties* correctes.

2.3.3. Description de la stratégie de résolution

L'attribut *stratégie* permet d'associer une stratégie de résolution à une tâche. Celle-ci définit les différentes étapes de résolution et le flot de contrôle qui ordonne ces étapes. Chaque étape de résolution est désignée par son nom. Pour indiquer le flot de contrôle, nous avons introduit différents opérateurs dans SCARP : *séquence*, *parallèle*, *choix, choix-utilisateur, spec, itération*, et *utilisateur.*

L'opérateur *séquence* indique que toutes les étapes énumérées par la suite doivent être résolues les unes après les autres. L'opérateur *parallèle* définit que toutes les étapes énoncées doivent être exécutées, mais qu'elles sont indépendantes les unes des autres et qu'elles peuvent ainsi être résolues en parallèle. Les opérateurs *choix* et *choix-utilisateur* permettent d'introduire un choix entre plusieurs étapes possibles ; la résolution d'une seule d'entre elles est nécessaire. Mais, tandis qu'un *choix* est par défaut contrôlé par le système, un *choix-utilisateur* est contrôlé par l'utilisateur (cf. chapitre trois). L'opérateur *itération* indique des étapes qui peuvent être parcourues plusieurs fois. L'opérateur *utilisateur* désigne des étapes de résolution entières qui se font sous le contrôle de l'utilisateur. La signification de l'opérateur *spec* est détaillé plus loin. La figure 18 montre la grammaire de description de la stratégie de résolution à l'aide de ces opérateurs, la figure 19 donne un exemple.

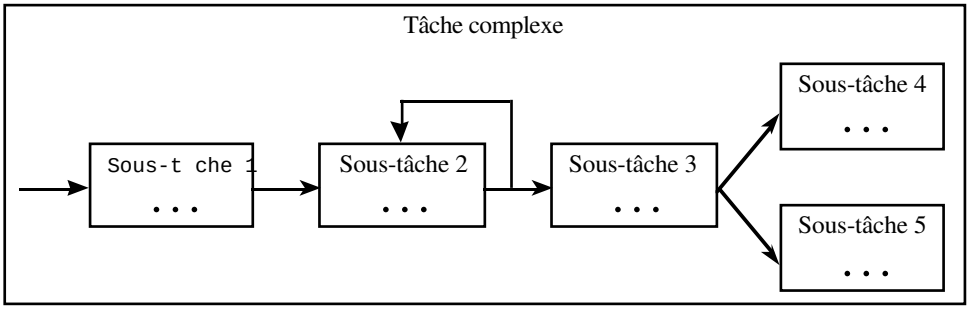
<stratégie>	:=	(séquence <liste-de-stratégies>)	||
		(parallèle <liste-de-stratégies>)	||
		(choix <liste-d'étapes>)	||
		(choix-utilisateur <liste-d'étapes>)	||
		(itération <étape>)	||
		(utilisateur <étape>)	||
		(spec)	
<liste-de-stratégies>	:=	<stratégie> <liste-de-stratégies>	||
		<stratégie>	||
		<étape>	
<liste-d'étapes>	:=	<étape> <liste-d'étapes>	||
		<étape>	
<étape>	:=	<symbole>	

Figure 18 : Grammaire de description de stratégies de résolution dans SCARP. Une stratégie de résolution est décrite en combinant les étapes de résolution nécessaires à l'aide des opérateurs pré-définis. Chaque étape de résolution est nommée par un symbole qui correspond dans la description de la classe modélisant la tâche complexe au nom d'un attribut de la classe sous-tâche qui définit le contexte d'exécution correspondant : &sous-tâche <symbole> $un contexte-d'exécution (cf section 2.4.).

Les opérateurs *séquence*, *parallèle*, *choix* et *itération* correspondent aux opérateurs classiques en algorithmique. Nous avons ajouté les opérateurs *choix-utilisateur* et *utilisateur* pour pouvoir indiquer explicitement les endroits où doit s'établir une coopération système-utilisateur pendant la résolution.

Les opérateurs *séquence*, *parallèle*, *choix* et *choix-utilisateur* s'appliquent sur un ensemble d'étapes de résolution et sont nécessaires à la gestion du flot de contrôle. Les opérateurs *itération* et *utilisateur* ne concernent qu'une seule étape. Leur définition n'est

pas indispensable pour l'indication du flot de contrôle, étant donné que le contexte d'exécution correspondant contiendra les informations complémentaires, nécessaires pour contrôler leur exécution (cf. section 2.4.). Cependant, leur définition est intéressante puisqu'elle permet de récapituler, dans le seul attribut *stratégie*, les caractéristiques essentielles d'une stratégie de résolution.



(**séquence** sous-tâche_1
(**itération** sous-tâche_2) sous-tâche_3 (**choix** sous-tâche_4 sous-tâche_5))

Figure 19 : Exemple de description d'une stratégie de résolution.

L'opérateur *spec* permet, après exécution d'une partie de la stratégie de résolution, de la préciser. Du fait que la description de la stratégie est un attribut, la stratégie est héritée dans la hiérarchie de classes décrivant les tâches. A partir du moment où un problème est défini de manière suffisamment précise dans la hiérarchie de classes, une stratégie de résolution lui est attachée. Mais cette stratégie peut être très générale et couvrir plusieurs stratégies plus spécifiques. Une stratégie plus spécifique précise en général le type des sous-tâches à exécuter pour réaliser les différentes étapes de résolution. Souvent, pendant l'exécution d'une première partie de la stratégie, des décisions importantes concernant ces sous-tâches sont prises, qui limitent et déterminent les choix possibles pour la suite de la stratégie.

Donnons un exemple : la comparaison de deux objets. Elle commence par deux étapes similaires, permettant chacune de caractériser un des objets. Puis, il faudra une étape finale pour effectuer la comparaison proprement dite. La caractérisation de ces objets peut se faire de différentes manières, chacune mettant en évidence des propriétés particulières. Dans la première étape de résolution, une manière de caractérisation va être choisie pour le premier objet. Pour pouvoir effectivement comparer les deux objets en question à partir des propriétés extraites, il faudra ensuite choisir pour le deuxième objet une manière de caractérisation compatible avec la première.

L'opérateur *spec* permet de définir à l'intérieur de stratégies générales les endroits où des décisions limitantes, prises auparavant doivent être prises en compte. Ceci revient à préciser la stratégie de la tâche à partir de la partie de stratégie déjà exécutée. Pour l'exemple donné auparavant, il est ainsi intéressant d'introduire l'opérateur *spec* après la première étape de résolution, la caractérisation du premier des deux objets à comparer. La figure 20 donne un autre exemple général pour la spécialisation de la stratégie de résolution et pour l'utilisation de l'opérateur *spec*.

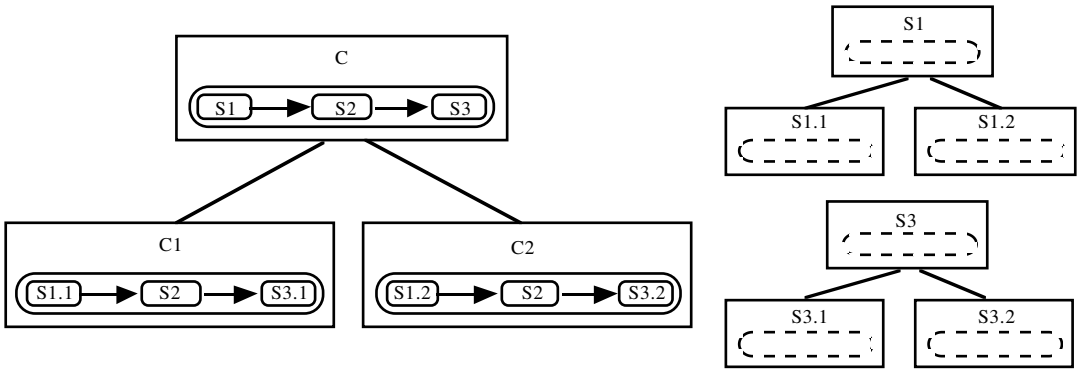


Figure 20 : Héritage de la stratégie et opérateur spec. Une stratégie de résolution peut être décrite à un niveau général. La tâche complexe C par exemple est modélisée par une séquence de trois étapes qui correspondent aux sous-tâches S1, S2 et S3. C1 et C2 contiennent des stratégies plus spécifiques : les S1 et S3 sont remplacées par S1.1 et S3.1 ou S1.2 et S3.2 respectivement. S1.1 et S1.2 (respectivement S3.1 et S3.2) sont des précisions de S1 (respectivement S3) ; elles sont ainsi modélisées par des sous-classes des classes modélisant S1 et S3. Si le choix soit de S3.1 soit de S3.2 pour la sous-tâche S3 dépend impérativement du choix de S1.1 ou de S1.2 pour la première, il est intéressant d'introduire l'opérateur spec avant la sous-tâche S3 dans la définition de la stratégie de résolution pour C.

2.3.4. Coopération avec l'utilisateur

Dans les deux sections précédentes, nous avons précisé la structuration des connaissances attachées à une tâche et la description de stratégies de résolution dans notre modèle de tâches. Pour les deux nous avons pris en compte la nécessité de pouvoir coopérer avec l'utilisateur. Parmi les connaissances attachées à une tâche nous distinguons les tâches de visualisation des entrées et des sorties. Ces tâches permettent de définir pour chaque tâche des moyens pour visualiser l'ensemble de ses entrées ou de ses sorties pour l'utilisateur. Parmi les opérateurs pour la description de la stratégie de résolution, les opérateurs *choix-utilisateur* et *utilisateur* sont particulièrement importants : ils permettent de pré-définir les interactions système-utilisateur a priori nécessaires pendant la résolution. Dans la définition des contextes d'exécution pour les sous-tâches d'autres moyens pour pré-définir des interactions sont disponibles (cf. section 2.4.).

Une tâche modélise un problème à résoudre via un ensemble d'entrées et de sorties. Mais certaines tâches, n'ont pas pour but d'obtenir des résultats précis mais de visualiser des informations pour l'utilisateur. Une telle visualisation constitue un acte coopératif envers l'utilisateur et doit pouvoir être gérée par le système. Ainsi nous avons pré-défini une classe spécifique dans SCARP, *tâche-visualisation*, racine de la hiérarchie de toutes les tâches n'effectuant que de la visualisation.

Comme nous verrons ultérieurement (cf. section 3.2.3.) la coopération consiste aussi dans l'exploration interactive de tâches nouvelles par l'utilisateur. Ceci conduit à la définition de nouvelles classes correspondant à ces tâches dans la base de connaissances. Etant donné que l'endroit exact où cette classe doit être insérée (la place dans la hiérarchie) n'est pas connu au moment de la définition, une autre classe a été pré-définie qui regroupe toutes ces nouvelles classes en tant que sous-classes : *tâche-utilisateur*. La figure 21 montre la hiérarchie de tâches pré-définie dans SCARP.

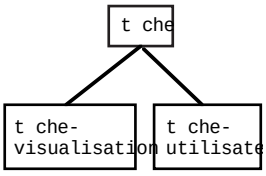


Figure 21 : Classes de tâches prédéfinies. A part la classe *tâche* qui constitue la racine de toutes les tâches, nous distinguons les tâches de *visualisation* et les tâches *utilisateur*.

2.3.5. Indépendance du contexte d'exécution

Jusqu'ici nous nous sommes intéressés à la spécification, à la structure et à l'interdépendance des différentes connaissances qui sont associées à une tâche. Nous avons distingué d'abord deux types de tâches, suivant le fait qu'elles sont exécutables ou non. Nous avons ensuite identifié différents rôles que peuvent jouer les connaissances associées à une tâche. Nous lui attachons une stratégie pour définir les différentes étapes nécessaires à sa résolution. Le modèle que nous avons présenté jusqu'ici prend aussi en compte la nécessité d'une coopération avec l'utilisateur pendant la résolution.

Mais, ce modèle décrit une tâche indépendamment de son utilisation au sein d'autres tâches. Une même tâche peut en effet être utilisée par différentes tâches englobantes[1] pour réaliser une étape spécifique de résolution. Pour pouvoir réaliser une telle étape, il ne suffit pas d'indiquer simplement la (sous-)tâche à exécuter, il faut aussi l'intégrer dans ce contexte spécifique d'exécution. Pour cela, des informations complémentaires doivent être décrites, ne serait-ce que le flot d'informations qui doit renseigner ses entrées pour pouvoir gérer l'exécution de façon automatique. Le type et le contenu de ces informations dépendent à chaque fois du contexte spécifique et requièrent donc à chaque fois une description nouvelle et indépendante.

Dans la section suivante, nous allons nous intéresser à l'utilisation d'une tâche au sein de tâches englobantes, c'est-à-dire à la manière dont elle peut être intégrée dans différents contextes d'exécution. Nous précisons les informations nécessaires à l'intégration d'une tâche dans un contexte et nous introduisons un type d'objet particulier dans notre modèle, le contexte d'exécution, qui regroupe ces informations.

2.4. CONTEXTE D'EXECUTION D'UNE TACHE

L'exécution d'une tâche peut dépendre de l'utilisation qui en est faite. Le contexte d'exécution que nous introduisons maintenant va servir à décrire de façon souple toutes les informations nécessaires pour intégrer une tâche dans le contexte spécifique d'une tâche englobante. Il lie la tâche englobante et la tâche à exécuter en tant que sous-tâche (figure 22).

Des informations diverses sont nécessaires pour intégrer une tâche dans un contexte spécifique. Elles sont identifiées dans la section 2.4.1. Pour pouvoir gérer une résolution de façon automatique, le flot d'informations entre sous-tâche et tâche englobante doit être décrit. Dans notre modèle, cette description est contenu dans le contexte d'exécution. Sa grammaire de description est détaillée dans la section 2.4.2.

Les informations à décrire pour intégrer une tâche dans un contexte spécifique dépendent aussi de la stratégie de résolution définie pour la tâche englobante. S'il faut itérer sur une (sous-)tâche par exemple, il faut spécifier des informations supplémentaires, nécessaires au contrôle de l'itération. Nous avons identifié différents

[1] Si une tâche A utilise dans sa décomposition directement la tâche B en tant que sous-tâche, A est une tâche englobante de la tâche B.

types de contrôle importants pour SCARP que nous présentons dans la section 2.4.3. Nous avons en particulier introduit des types de contrôle spécifiques pour décrire la coopération système-utilisateur pendant la résolution. Nous les détaillerons à la fin.

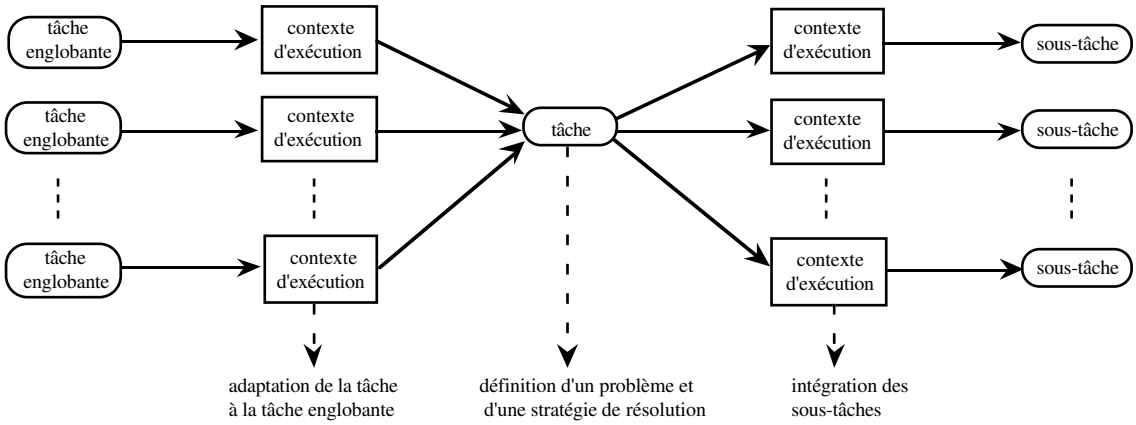


Figure 22 : Modèle de tâche SCARP : la tâche et son contexte d'exécution. Une tâche définit un problème à résoudre et, si ce problème est suffisamment spécifique, elle lui associe une stratégie de résolution. La définition d'une tâche ne préjuge pas de son utilisation. Si une tâche est utilisée en tant que sous-tâche dans une tâche englobante pour remplir un rôle précis, toutes les informations nécessaires pour adapter son exécution au contexte de la tâche englobante spécifique sont décrites dans un contexte d'exécution. Une tâche peut être utilisée dans plusieurs tâches englobantes et ainsi être référencée par différents contextes d'exécution. De même, une tâche a en général plusieurs sous-tâches et fait ainsi elle-même référence à différents contextes d'exécution.

2.4.1. Définition d'un contexte

Pour chaque étape de sa stratégie de résolution, ainsi que pour chacune de ses pré- et post-tâches et de ses tâches de visualisation, la définition d'une tâche contient un attribut qui référence un contexte d'exécution. Ce contexte d'exécution définit la tâche à exécuter pour résoudre l'étape correspondante, et comment il faut contrôler son exécution pour l'adapter à celle de la tâche englobante.

Un contexte d'exécution décrit toutes les informations nécessaires pour la réalisation d'une étape de résolution par appel à une (sous-)tâche. Il précise la tâche à exécuter, le flot de données (cf section 2.4.2.) et, via des attributs de type *condition,* des conditions importantes pour gérer l'exécution de la sous-tâche. La figure 23 montre la définition du contexte d'exécution dans SCARP, et la figure 24 la hiérarchie des classes d'attributs introduites pour la définition des contextes.

La signification des différents attributs ou groupes d'attributs d'un contexte est la suivante :

- L'attribut *exécute* indique la tâche (ou la méthode) dont l'exécution permet de réaliser l'étape de résolution en question.
- L'attribut *conditionnalité* permet d'indiquer si, pour le succès de la tâche englobante, la réalisation de la sous-tâche en question est obligatoire, optionnelle ou conditionnelle, c'est-à-dire nécessaire seulement sous certaines conditions. L'attribut *condition* permet de définir pour ces tâches conditionnelles la condition associée.
- *Pré- et post-condition* permettent, en fonction du contexte, de préciser des conditions d'exécution pour la tâche à exécuter, en particulier de renforcer les

pré-/post-conditions définies pour cette tâche. En effet, la possibilité d'appliquer une tâche pour réaliser une étape de résolution précise peut dépendre de certaines pré-conditions, la validité de ses résultats de certaines post-conditions supplémentaires, c'est-à-dire non incluses dans les pré- et post-conditions définies dans la tâche référencée elle-même. En général, l'exécution d'une tâche peut, par exemple, être considérée comme succès, si elle donne en résultat un entier. Dans un contexte spécifique il peut être nécessaire d'obtenir un entier positif.

- Les attributs *valide-entrées (valide-sorties)* permettent d'indiquer si avant (après) la résolution de la sous-tâche l'ensemble de ses entrées (sorties) doit être validé par l'utilisateur. Les attributs *visu-entrées (visu-sorties)* permettent d'indiquer que cet ensemble doit seulement être visualisé.
- L'attribut *fd* permet d'indiquer le flot de données entre la sous-tâche et la tâche englobante et ses autres sous-tâches (cf. section, 2.4.2.). S'il s'agit d'une sous-tâche itérative un flot de données itératif est séparément défini.

```
{ contexte-général     sorte-de          =                 entité-SCARP ;
                       exécute           $un / $liste-de   tâche ;
    &condition         pré-condition     $un               booléen ;
    &condition         post-condition    $un               booléen ;
                       conditionnalité   $un               symbole
                                         $domaine          obligatoire optionnelle conditionnelle
                                         $défaut           obligatoire ;
    &condition         condition         $un               booléen ;
    &condition         valide-entrées    $un               booléen ;
    &condition         valide-sorties    $un               booléen ;
    &condition         visu-entrées      $un               booléen ;
    &condition         visu-sorties      $un               booléen ;
    &flot-données      fd                $liste-de         objet
}
```

Figure 23 : Définition du *contexte général* d'exécution dans SCARP. La tâche à exécuter en tant que sous-tâche est précisée par l'attribut *exécute*. Cet attribut est mono-valué (*$un*) s'il s'agit d'une exécution simple et multi-valué (*$liste-de*) s'il s'agit d'une sous-tâche itérative. Pour une instance d'un tel contexte, c'est-à-dire pendant l'exécution de la tâche englobante, cet attribut va contenir la ou les instances des (sous-) tâche(s) exécutée(s). Les attributs du type *condition* permettent de définir des conditions importantes pour la gestion de l'exécution de la tâche référencée par *exécute*.

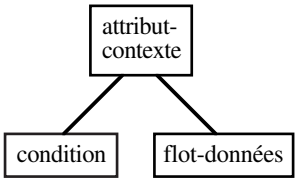


Figure 24 : Classes d'attributs définies pour les contextes. Ces classes distinguent les attributs qui définissent des conditions importantes pour l'exécution de la sous-tâche en question et des attributs définissant des flots de données.

2.4.2. Description du flot de données

Pour décrire la résolution d'une tâche, il faut définir non seulement le flot de contrôle mais aussi le flot d'informations, c'est-à-dire le flot de données entre la tâche et ses sous-tâches. Un flot de données définit comment les données transitent entre les tâches. Pour chaque sous-tâche, il indique comment les valeurs de ses attributs d'entrée sont déterminées, et quels attributs de sorties de la tâche englobante sont valuées par ses propres sorties (figure 25).

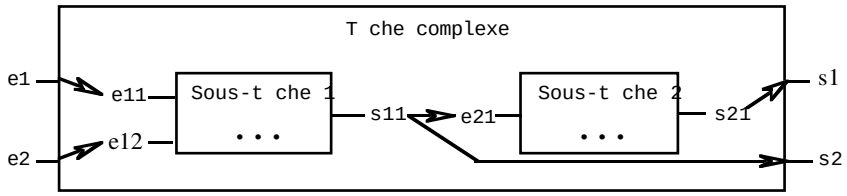


Figure 25 : Flot de données. Le flot de données décrit pour chaque sous-tâche comment ses entrées sont déterminées et si ses sorties déterminent des sorties de la tâche globale. Pour la *sous-tâche 1,* le flot de données indique que ses entrées *e11* et *e12* sont valuées par les entrées *e1* et *e2* de la *tâche complexe* et que sa sortie *s11* donne sa valeur à la sortie *s2* de la *tâche complexe*. Pour la *sous-tâche 2,* le flot de données indique que son entrée *e21* correspond à la sortie *s11* de la *sous-tâche 1* et que sa sortie *s21* détermine la sortie s1 de la *tâche complexe*.

L'attribut *fd* dans un contexte d'exécution contient la liste des flots de données qui concernent la sous-tâche en question. La figure 26 montre la grammaire que nous avons définie pour exprimer le flot de données par une notation pointée.

<flots-de-données>	:=	<flot-de-données>	||	<flot-de-données> <flots-de-données>
<flot-de-données>	:=	"(" <liste-sources> "> " <liste-destinations> ")"		
<liste-sources>	:=	<liste-chemins-accès>		
<liste-destinations>	:=	<liste-chemins-accès>		
<liste-chemins-accès>	:=	<chemin-accès>	||	<chemin-accès> <liste-chemins-accès>
<chemin-accès>	:=	<chemin-accès-ctxt>	||	<chemin>
<chemin-accès-ctxt>	:=	nom-attribut	||	nom-attribut <chemin>
<chemin>	:=	"." nom-attribut	||	"." nom-attribut <chemin>

Figure 26 : Grammaire d'expression du flot de données. Un attribut de type flot de données contient un ensemble de flots de données. Un flot de données est décrit par un ensemble de sources et de destinations qui sont indiquées par une notation pointée. Chaque élément de la notation pointée correspond au nom d'un attribut. Si le type de l'attribut ainsi référencé est complexe, c'est-à-dire fait référence à une classe Shirka, un autre nom d'attribut peut désigner un des attributs de cette classe et ainsi de suite.

Avec cette notation le flot de données de l'exemple de la figure 25 s'écrit ainsi pour sous-tâche 1 :

(.e1 > exécute.e11) (.e2 > exécute.e12) (exécute.s11 > .s2) .

Pour sous-tâche 2 il correspond à :

(.sous-tâche_1.exécute.s11 > exécute.e21) (exécute.s21 > .s1) .

Un flot peut concerner plusieurs attributs à la fois. Etant donné que dans notre modèle l'exécution de certaines sous-tâches peut ne pas être obligatoire, nous avons introduit des règles d'interprétation pour le flot de données. S'il est possible qu'une première source d'un flot de données ne contient pas de valeur, s'il s'agit d'une valeur de sortie d'une sous-tâche non obligatoire par exemple, plusieurs sources peuvent être définies. De même, plusieurs destinations peuvent être indiquées dans un flot de données. Elles doivent dans ce cas être renseignées en même temps et à partir d'une même source. Si la notation débute avec ".", le chemin d'accès à la valeur commence dans la tâche englobant le contexte, sinon il débute dans le contexte lui-même.

2.4.3. Différents types de contrôle sur une tâche

Nous avons décrit jusqu'à maintenant les informations générales qui permettent de préciser comment l'exécution d'une tâche doit s'intégrer dans une tâche englobante. Mais déjà dans la définition du flot de contrôle nous avons montré que l'exécution de sous-tâches peut s'effectuer sous le contrôle de l'utilisateur ou de façon itérative. La définition du contexte d'exécution requiert dans ces cas la définition d'informations supplémentaires.

Dans SCARP nous avons par conséquent introduit différents types de contrôle possibles sur les sous-tâches, sur une exécution "simple" d'un côté et sur une exécution itérative de l'autre. Le contrôle d'une exécution simple ne requiert pas d'informations supplémentaires. Pour pouvoir contrôler l'exécution d'une tâche itérative, il faut permettre de lui associer un traitement initial (avant la première itération), un traitement final (après la dernière itération), un pré- et un post-traitement (avant et après chaque itération), et un flot de données itératif, à exécuter lors de chaque itération. Différents types de contrôle sont possibles pour une itération. Nous avons ainsi introduit dans SCARP l'itération sur une liste, l'itération sur un intervalle et l'itération jusqu'à la satisfaction d'une condition d'arrêt. Ces types d'itération correspondent aux opérateurs *for, while* et *until*, qui sont généralement disponibles dans les langages de programmations.

Pour pouvoir pré-définir des modes d'interaction ou de coopération précis pendant la résolution, nous avons ensuite ajouté d'autres types de contrôle. Du côté de l'exécution simple nous avons précisé deux cas : une exécution simple peut ainsi plus précisément concerner un acte de visualisation de la part du système pour l'utilisateur. Cette visualisation peut, soit être sans influence sur la résolution (visualisation), soit servir à demander une interprétation ou un résultat à l'utilisateur (demande). Du côté des itérations, nous avons ajouté la possibilité d'effectuer une itération sous le contrôle explicite de l'utilisateur, c'est-à-dire jusqu'à la satisfaction de celui-ci.

Pour chacun des types de contrôle possibles sur l'exécution d'une sous-tâche, nous avons pré-défini une classe modélisant le contexte d'exécution correspondant (figure 27). Ces classes contiennent à chaque fois des attributs spécifiques qui permettent de préciser toutes les informations supplémentaires nécessaires pour le type de contrôle concerné.

Par la suite, nous n'allons pas présenter tous ces contextes d'exécution en détail. Dans la section suivante, nous décrivons seulement les contextes d'exécution qui sont particulièrement importants pour modéliser la coopération système-utilisateur pendant la résolution.

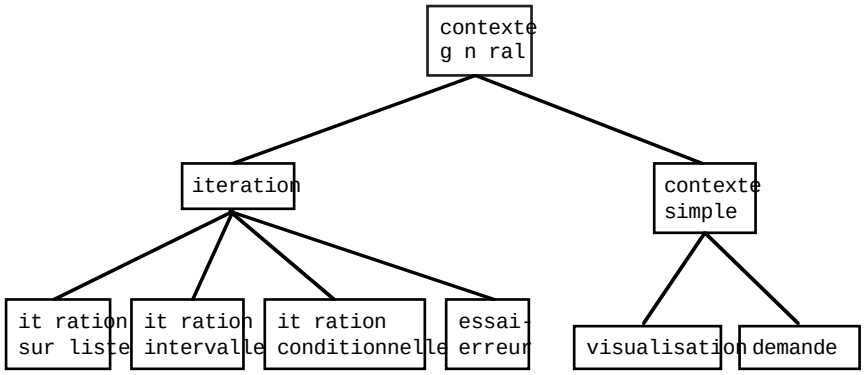


Figure 27 : Hiérarchie des différentes classes de contexte. Pour chaque type de contrôle possible dans SCARP, nous avons défini une classe correspondante, qui permet de décrire son contexte d'exécution.

2.4.4. Coopération avec l'utilisateur

Chaque contexte d'exécution intègre déjà, outre les informations nécessaires pour pouvoir automatiser la résolution, des moyens pour modéliser une certaine coopération système-utilisateur pendant la résolution de la tâche référencée. Pour cela, le contexte général définit un ensemble d'interactions possibles avec l'utilisateur : la validation ou la visualisation des entrées (sorties) avant (respectivement après) l'exécution d'une sous-tâche et la possibilité de définir une sous-tâche optionnelle, c'est-à-dire seulement à exécuter sur la demande de l'utilisateur. Des contextes d'exécution plus spécifiques permettent de pré-définir d'autres interactions système-utilisateurs :

- le contexte *visualisation* permet d'afficher des informations pour l'utilisateur,
- le contexte *demande* définit une tâche-utilisateur, c'est-à-dire une demande d'information à l'utilisateur,
- le contexte *essai-erreur* de laisse le contrôle d'une itération à l'utilisateur.

Nous présentons maintenant chacun de ces trois contextes plus en détail.

2.4.4.1. Contexte visualisation

Le contexte *visualisation* référence, via l'attribut *exécute,* une sous-tâche qui sert uniquement à visualiser des informations pour l'utilisateur. Il a été défini pour permettre d'introduire explicitement dans la stratégie de décomposition des interactions informatives pour l'utilisateur qui sont a priori sans influence directe sur le processus de résolution.

2.4.4.2. Contexte demande

A l'opposé du contexte visualisation, le contexte *demande* permet de définir des tâches de visualisation qui sont importantes pour la résolution. Ce contexte permet d'associer, à une visualisation, une demande de valeur à l'utilisateur (figure 28). Pour aider l'utilisateur à donner la valeur requise, une stratégie d'aide peut être attachée à ce contexte via l'attribut *exécute*. Soit cette stratégie d'aide propose une valeur pour répondre directement à la demande, soit elle calcule des valeurs associées ou des graphiques dont la visualisation va aider l'utilisateur à donner la valeur en question.

La description de ce contexte est basée, en plus de l'attribut *exécute,* sur deux attributs supplémentaires. L'attribut *obtenir* indique le chemin d'accès à l'attribut dont la valeur doit être demandée, l'attribut *demander* permet d'indiquer un nom d'attribut local au contexte qui contiendra la valeur déterminée par la stratégie d'aide et qui permet de définir un commentaire explicatif pour l'utilisateur.

```
...
                stratégie
                    $valeur    ( ... ( utilisateur sélection-facteurs ) ... ) ;
&sous-tâche     sélection-facteurs
                    $un   { demande
                                exécute    $un         calcul-courbe ;
                                obtenir    $valeur     .nb-facteurs-conservés ;
                                demander   $valeur     nombre-de-facteurs ;
                                nombre-de-facteurs
                                           $un         entier
                                           $com        "Sélectionnez le nombre de facteurs à conserver" ;
                                fd         $valeur
                                                ( .courbe-valeurs-propres.modèle > exécute.modèle )
                          }
...
```

Figure 28 : Exemple d'un contexte-demande. La tâche *calcul-courbe,* référencée par *exécute*, calcule en sortie une représentation graphique dont l'interprétation aide l'utilisateur à choisir le *nombre de facteurs* à conserver. Pendant l'exécution, cette application graphique est intégrée par le système dans une fenêtre d'interaction, c'est-à-dire de demande de la valeur de l'attribut *nb-facteurs-conservés* de la tâche englobante.

2.4.4.3. Contexte essai-erreur

Le *contexte essai-erreur* (figure 29) permet d'itérer sur une tâche jusqu'à ce que l'utilisateur soit satisfait du résultat. Avant chaque itération l'utilisateur a la possibilité de modifier le paramètre d'entrée qui détermine essentiellement le résultat de la tâche[1]. Un contexte essai-erreur référence via l'attribut *exécute* la tâche sur laquelle il faut itérer. Après chaque itération la validation, soit de l'ensemble des sorties, soit de la valeur de l'attribut indiqué par *à-valider,* est demandée à l'utilisateur. Si celui-ci n'est pas satisfait du résultat, il modifie la valeur de l'attribut indiqué par *à-modifier* et relance l'exécution.

[1] Pour l'instant nous avons choisi de ne considérer que le cas le plus simple où la valeur d'un seul paramètre d'entrée doit être modifiée.

```
...
                stratégie
                    $valeur     ( ... ( itération calcul-fct-covariance ) ... ) ;
&sous-tâche     calcul-fct-covariance
                    $un   { essai-erreur
                                exécute         $liste-de   calcul-covariance ;
                                à-valider       $valeur     fct-covariance ;
                                à-modifier      $valeur     nb-lags ;
                                fd              $valeur     ( exécute.fct-covariance > fct-covariance ) ;
                                fd-iter         $valeur     ( .séquence > exécute.séquence )
                            }
...
```

Figure 29 : Exemple d'un contexte essai-erreur. La tâche *calcul-covariance,* référencée par *exécute*, calcule la fonction de covariance entre plusieurs signaux contenus dans l'entité *séquence*. Le résultat de ce calcul dépend du nombre de points, choisi au début de l'exécution, qui doit être pris en compte (*nb-lags*). A la fin de chaque itération la fonction de covariance (*fct-covariance*) est visualisée pour que l'utilisateur l'évalue et décide s'il faut remodifier le paramètre *nb-lags* ou non.

2.4.5. Signification de la notion de contexte

La notion de contexte d'exécution nous permet de séparer clairement la définition d'une tâche et celle des informations nécessaires pour intégrer son exécution dans le contexte spécifique d'une tâche englobante. Le contexte d'exécution sert d'une part à définir des informations comme le flot de données, nécessaires pour gérer une exécution automatique. Il permet d'autre part de faire participer l'utilisateur au contrôle de l'exécution de chaque sous-tâche, en lui permettant de visualiser ou de valider ses entrées ou ses sorties. La structuration en différentes classes de contexte permet par ailleurs de distinguer différents types de contrôle spécifique sur une sous-tâche. Certaines d'entre elles permettent en particulier de gérer des interactions spécifiques supplémentaires avec l'utilisateur.

Un contexte d'exécution SCARP n'est donc pas équivalent à un contexte hypothétique d'exécution comme dans ART [Clay84] ou KEE [Film88]. Dans SCARP, un contexte sert à définir les liens qui existent entre une tâche englobante et une (sous-)tâche et à décrire toutes les informations nécessaires pour permettre à SCARP de gérer son exécution. Dans ART et KEE, la notion de contexte est utilisée pour représenter des mondes hypothétiques parallèles. Un tel contexte incorpore donc à chaque fois toutes les connaissances d'un de ces mondes parallèles. Nous allons voir dans le chapitre trois, que différentes versions du processus de résolution peuvent exister et être gérées par SCARP. Mais ces versions sont toujours créées sur la demande explicite de l'utilisateur et ne constituent pas, comme dans KEE ou ART, des explorations en parallèle créées par le système lui-même.

2.5. BILAN

Nous avons détaillé le modèle complet de représentation de connaissances que nous avons défini pour SCARP. Avant de décrire (au chapitre trois) comment ce modèle est

exploité pour résoudre des problèmes en coopération avec l'utilisateur, nous allons résumer ses caractéristiques essentielles.

Tout d'abord une remarque générale : le modèle de représentation des connaissances que nous venons d'introduire n'est pas dédié à un type spécifique de problèmes, comme par exemple le diagnostic ou la conception. Nous proposons avec le modèle SCARP un outil de description de connaissances général qui permet a priori de décrire les connaissances de résolution pour tout problème décomposable en sous-problèmes. La distinction faite, par exemple dans KADS ou encore par Chandrasekaran entre différents types de problèmes et l'identification de tâches génériques correspondantes, nous semble intéressante pour guider le processus d'acquisition des connaissances. Mais ces descriptions soit restent trop conceptuelles, c'est-à-dire comme dans KADS à un niveau trop éloigné d'une implémentation opérationnelle, soit conduisent à un ensemble de systèmes dédiés et limités à un de ces types de problèmes spécifiques.

Comme nous l'avons montré, dans notre modèle, nous distinguons trois types essentiels de connaissances : les tâches, les méthodes et les entités. Les tâches définissent les problèmes existants et leurs associent des stratégies de résolution. Les méthodes représentent les moyens de résolution directs et élémentaires et les entités les objets manipulés dans un domaine. Dans notre modèle, chaque tâche est définie en dehors de toute utilisation précise. Ce modèle de tâches est complété par la notion de contexte d'exécution qui permet d'adapter une tâche à l'utilisation précise qui doit en être faite. Dans la description des tâches, nous avons directement lié les connaissances sur les problèmes existants dans un domaine avec celles sur les stratégies de résolution appropriées. Ainsi nous n'avons pas besoin de modéliser des connaissances supplémentaires de contrôle pour sélectionner la stratégie appropriée à un problème. Toutes les connaissances nécessaires pour gérer le processus de résolution sont intégrées dans le modèle SCARP ; c'est un modèle opérationnel.

Comme nous le verrons dans le chapitre trois, la structuration et l'organisation de ces connaissances en hiérarchies d'abstraction sont directement exploitées pour une résolution par planification hiérarchique. Le modèle de connaissances SCARP modélise donc le processus de résolution de problèmes par planification hiérarchique. Nous fixons, dans la description de la stratégie associée à une tâche, un flot de contrôle qui ordonne les étapes de résolution nécessaires. Nous partons ainsi de l'idée que les différentes étapes de résolution n'interagissent pas entre elles, dans le sens où une étape défait ce qui a été construit par une autre. Notre modèle n'est donc pas un modèle de planification non-linéaire.

Prenant en compte la nécessité de développer un système coopératif, nous avons identifié différents types d'interactions avec l'utilisateur qui peuvent être nécessaires pendant la résolution de problèmes. Certaines de ces interactions se situent au niveau du contrôle de l'exécution, comme par exemple concernant le contrôle d'une itération par l'utilisateur. Certaines se situent au niveau de l'exécution elle-même, comme par exemple la nécessité de demander à l'utilisateur de résoudre une tâche-utilisateur ou comme la visualisation d'informations pour l'utilisateur. Nous avons intégré ces interactions dans notre modèle de représentation des connaissances. De plus, toutes ces interactions sont suffisamment élémentaires et limitées pour être entièrement gérées par le système SCARP pendant le processus de résolution.

Avec notre modèle, nous avons structuré le plus possible les connaissances nécessaires à la résolution de problèmes. Nous avons défini différents rôles généraux que peuvent jouer ces connaissances pendant le processus de résolution. Ces rôles sont matérialisés par l'introduction de classes et de classes d'attributs pré-définies. Ainsi nous distinguons par exemple les attributs d'entrée et les attributs de sortie d'une tâche ou alors des conditions associées à son exécution. Le nom d'un élément de la base de connaissances précise toujours sa signification. Chaque classe désigne ainsi par son nom

l'objet qu'elle modélise. Les noms de ses attributs expriment la signification qu'ont les objets ou connaissances référencés par cet attribut. Les noms de classes et les noms d'attributs correspondent ainsi aux notions de fonction et de méta-classes identifiés dans KADS.

3. SCARP - RESOLUTION DE PROBLEMES

Dans le chapitre précédent, nous avons présenté notre modèle de représentation des connaissances pour la résolution de problèmes. Les connaissances représentées avec le formalisme adopté doivent ensuite être exploitées pour gérer le processus de résolution de problèmes. Celui-ci doit être géré de façon coopérative, c'est-à-dire de façon souple, contrôlée par le système ou par l'utilisateur respectivement. Nous avons adopté une approche de résolution de problèmes par planification hiérarchique, car elle permet d'établir, durant la résolution, une coopération système-utilisateur sur chacun des niveaux d'abstraction et de décomposition introduits dans la modélisation et exploités.

L'intégration des interactions système-utilisateur dans la modélisation des connaissances, présentée dans le chapitre précédent, est un premier aspect important pour la résolution de problèmes en coopération système-utilisateur. Nous montrerons dans ce chapitre comment elles sont gérées pendant le processus de résolution. Cette prédéfinition d'interactions dans la base de connaissances revient à effectuer une distribution fixe des (sous-)tâches ou du contrôle sur l'exécution de ces (sous-)tâches à l'utilisateur ou au système respectivement. Mais une résolution coopérative de problèmes ne doit pas être restreinte à une telle distribution fixe des tâches. Un deuxième aspect important pour la coopération est que cette distribution soit modifiable dynamiquement, pendant le processus de résolution. Le système doit ici garantir la cohérence de ce processus. Nous avions identifié différentes caractéristiques pour un système coopératif d'aide à la résolution de problèmes qui conduisent à une plus grande souplesse :

- indépendamment des interactions prédéfinies, un système coopératif doit permettre à l'utilisateur de décider dynamiquement quelles responsabilités il veut prendre pendant la résolution et lesquelles il confie au système ;
- une fois la résolution entamée, un système coopératif doit permettre à l'utilisateur de remettre en cause toutes les décisions prises pendant le processus de résolution, tout en garantissant la cohérence de la résolution. Par la suite l'utilisateur doit pouvoir revenir facilement sur toutes ces modifications ;
- la résolution de problèmes ne doit pas être restreinte aux tâches prédéfinies dans la base de connaissances : le système doit aussi permettre à l'utilisateur d'explorer de façon interactive de nouvelles stratégies de résolution.

Pour cela, nous proposons une architecture générale pour SCARP (figure 1). Cette architecture sépare le raisonnement nécessaire pour la résolution (plus ou moins automatique) de problèmes, réalisée par le *moteur de tâches,* de la gestion de la coopération, réalisé par le *gestionnaire de dialogues*. Ceci permet au système de gérer le processus de résolution de manière flexible, interruptible et modifiable par l'utilisateur [Jean92].

La résolution de problèmes dans SCARP est gérée par un mécanisme spécifique, le moteur de tâches. Celui-ci exploite directement la base de connaissances. Il gère de façon automatique la résolution tant que c'est possible. Au moment où une interaction avec l'utilisateur doit être effectuée, le moteur de tâches active le gestionnaire de dialogues et s'inactive lui-même.

Le gestionnaire de dialogues gère l'interaction avec l'utilisateur. S'il obtient un résultat adéquat, il le transmet au moteur de tâches qui est ainsi réactivé. Mais, dans un système coopératif, l'initiative pour une interaction peut aussi venir de l'utilisateur, par exemple

pour remettre en cause des décisions prises pendant le processus de résolution. Dans ces cas, c'est l'utilisateur qui active le gestionnaire de dialogues pour intervenir dans le fonctionnement du moteur de tâches. Le gestionnaire de dialogues contrôle l'ensemble des interactions entre le moteur de tâches et l'utilisateur. Il maintient ainsi la cohérence globale du processus de résolution.

Pour permettre à l'utilisateur de suivre et d'intervenir dans ce processus, l'interface du système par laquelle passe la coopération doit être particulièrement étudiée. Elle doit non seulement donner accès à l'état actuel du processus de résolution mais à toutes les informations reliées, et éventuellement permettre de les modifier.

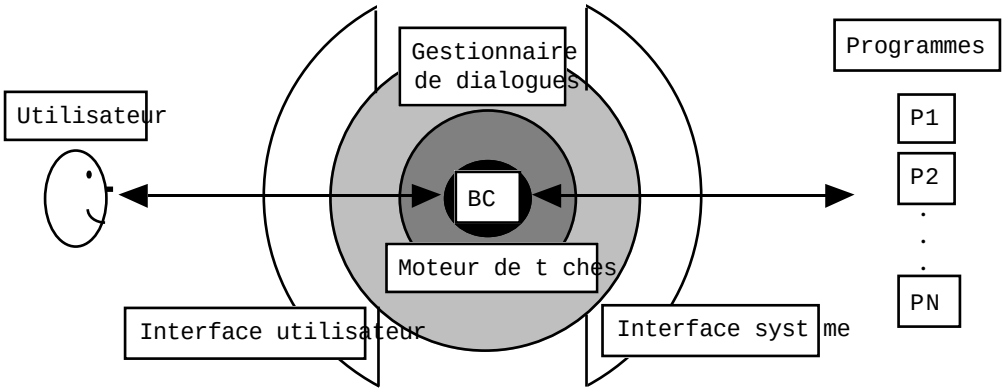


Figure 1 : L'architecture de SCARP [Chev&93]. SCARP est structuré en plusieurs couches successives : la base de connaissances (*BC*) est représentée au centre du système. Elle est exploitée par le *moteur de tâches* pour résoudre des problèmes de façon automatique. Pour la résolution de problèmes et pour pouvoir prendre en compte les interventions de l'utilisateur, le système a besoin de communiquer avec le monde extérieur, avec *l'utilisateur* d'un côté et avec le système d'exploitation de l'autre. Le *gestionnaire de dialogues* gère ces communications via *l'interface utilisateur* et *l'interface système* respectivement.

Dans ce chapitre, nous allons présenter comment la résolution de problèmes en coopération avec l'utilisateur est gérée. Tout d'abord, nous expliquons comment la résolution de problèmes est gérée par le moteur de tâches, de façon plus ou moins automatique. Nous détaillons comment l'utilisateur peut décider de prendre plus ou moins de responsabilités pendant la résolution. Ensuite, nous présentons comment le gestionnaire de dialogues permet de gérer la coopération avec l'utilisateur pendant la résolution. La conception et le fonctionnement de l'interface du système sont détaillés à la fin de ce chapitre.

3.1. MOTEUR DE TACHES

La résolution de problèmes dans SCARP est basée sur la représentation des connaissances présentée au chapitre deux. Le moteur de tâches gère la résolution de problèmes en différentes phases. L'alternance des phases les plus importantes, des phases de spécialisation et de décomposition, matérialise un processus de résolution de problèmes par planification hiérarchique. Ici, nous montrons premièrement comment la résolution peut être gérée de façon quasi automatique par le système. Nous indiquons deuxièmement les endroits où l'utilisateur peut ou doit intervenir. Nous détaillons troisièmement en quoi l'utilisateur peut modifier la part de responsabilité qu'il prend pendant le processus de résolution.

3.1.1. Gestion de la résolution de problèmes

Nous décrivons ici tout d'abord le cycle de contrôle du moteur de tâches. Ensuite nous montrons comment l'utilisateur peut prendre plus ou moins de contrôle pendant la résolution, comment il peut intervenir dans les différentes phases de résolution. Finalement, nous détaillons comment l'échec d'une tâche peut se produire au cours de ces phases et comment celui-ci est géré par le moteur.

3.1.1.1. Différentes phases de résolution de problèmes

En général, la résolution de problème passe par différentes phases [Poly65] : tout d'abord le problème général est posé ; ses caractéristiques sont de plus en plus spécifiées ; ensuite une stratégie de résolution appropriée est déterminée, éventuellement adaptée au problème spécifique, et exécutée ; au fur et à mesure de l'exécution, des résultats sont obtenus ; ceux-ci sont examinés pour vérifier s'ils sont corrects et que le raisonnement, qui a permis de les déterminer, est adéquat.

Nous retrouvons ces différentes phases dans le cycle de contrôle du moteur de tâches de SCARP. Résoudre une tâche avec SCARP correspond à instancier d'abord la classe correspondant au problème à résoudre et à obtenir les valeurs de ses attributs d'entrée, c'est-à-dire à poser le problème. Ensuite l'instance créée est spécialisée dans la hiérarchie de classes associée, ce qui permet de spécifier le problème et de choisir une stratégie de résolution adéquate. Puis, cette stratégie est exécutée, et les valeurs des attributs de sorties de l'instance sont complétées. Finalement, la validité de ces résultats est vérifiée.

Les phases de résolution du moteur de tâches de SCARP (figure 2) et leurs significations sont :

- la phase d'*instanciation* : durant cette phase la classe correspondant à la tâche à résoudre est instanciée. Ses entrées globales sont déterminées, soit via le flot de données, soit en les demandant à l'utilisateur.
- la phase de *spécialisation* : au cours de cette phase une stratégie de résolution adaptée est sélectionnée (cf. section 3.1.2.1.). En fonction des valeurs des entrées globales fournies dans la phase d'instanciation, l'algorithme de classement permet de déterminer, dans la hiérarchie de classes correspondante, les classes contenant les stratégies de résolution possibles. Une de ces classes est ensuite choisie.
- la phase de *complétion et de validation des entrées* : pendant cette phase les entrées stratégiques manquantes sont obtenues soit en demandant à l'utilisateur, soit par inférence, soit par application de pré-tâches. Puis, les pré-conditions sont vérifiées pour déterminer si la stratégie sélectionnée peut effectivement être appliquée[1]. Si cela est spécifié dans le contexte d'exécution, le système lance ensuite la tâche de visualisation des entrées et demande à l'utilisateur de valider de son côté l'ensemble des entrées.
- la phase de *décomposition ou exécution* : au cours de cette phase la stratégie de résolution, c'est-à-dire la stratégie de décomposition en sous-tâches, est exécutée. S'il y a un choix entre plusieurs sous-tâches possibles, il est effectué par le système ou par l'utilisateur. Chaque sous-tâche est ensuite à son tour

[1] Etant donné que les pré-conditions concernent, en général, aussi les entrées stratégiques, elles ne peuvent pas être prises en compte plus tôt : les valeurs des entrées stratégiques pourraient être encore inconnues.

résolue comme spécifié dans son contexte d'exécution. Si elle est complexe, ce même algorithme est appliqué récursivement. Si elle est élémentaire, la méthode associée est exécutée.

- la phase de *complétion et de validation des sorties* : durant cette phase des sorties globales non encore déterminées sont obtenues à partir des sorties stratégiques par inférence ou par application de post-tâches. Puis, les post-conditions sont vérifiées ce qui permet au système de vérifier si les résultats sont valides ou non. Si cela est spécifié dans le contexte d'exécution, le système lance ensuite la tâche de visualisation des sorties et demande aussi à l'utilisateur de valider les résultats.

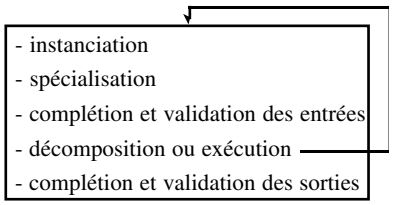


Figure 2 : Phases de résolution d'une tâche.

La figure 3 montre comment se déroule la résolution d'une tâche à partir de la phase de complétion et validation des entrées.

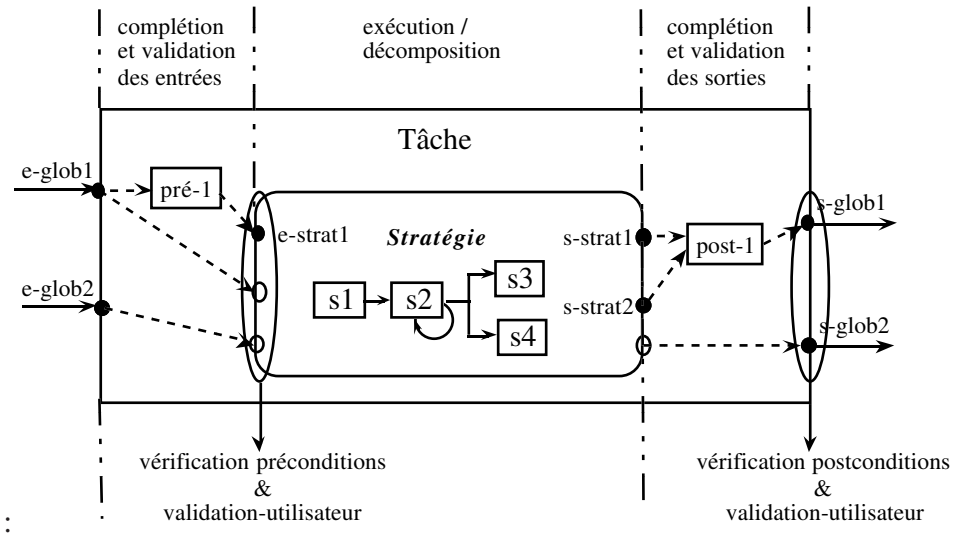


Figure 3 : Résolution d'une tâche (après la phase de *spécialisation*). Après la phase de spécialisation, les entrées globales *e-glob1* et *e-glob2* sont connues ; ensuite, pour compléter les entrées, à l'aide de la pré-tâche *pré-1* l'entrée stratégique *e-strat1* est déterminée. Les pré-conditions sont vérifiées et, si besoin, les entrées sont validées par l'utilisateur. La décomposition de la tâche complexe en sous-tâches, l'exécution de la stratégie, fournit les sorties stratégiques *s-strat1* et *s-strat2* et la sortie globale *s-glob2*. La sortie globale *s-glob1* est déterminée par l'application de la post-tâche *post-1* ; les post-conditions sont vérifiées et, si besoin, une validation des sorties est demandée à l'utilisateur.

Nous avons présenté les différentes phases de résolution d'une tâche. Les phases de spécialisation et de décomposition sont plus détaillées dans la section 3.1.2. Toutes les phases sont, par défaut, gérées et contrôlées par le moteur ; l'utilisateur n'intervient que

dans les endroits prédéfinis. Mais il est important de lui permettre de prendre plus de responsabilités concernant le processus de résolution. Dans la section suivante nous allons montrer comment ceci est possible dans SCARP.

3.1.1.2. Contrôle des différentes phases par l'utilisateur

Pour rendre le processus de résolution plus coopératif, nous avons tout d'abord introduit dans SCARP la possibilité pour l'utilisateur de prendre explicitement le contrôle des différentes phases de résolution. L'utilisateur peut prendre le contrôle d'une phase de résolution soit globalement soit seulement pendant une durée limitée et uniquement pour résoudre un problème précis. Nous indiquons ici dans quelle mesure l'utilisateur peut intervenir dans les différentes phases et ce que cela signifie pour la résolution :

- Concernant la phase d'*instanciation*, l'utilisateur peut décider si oui ou non il veut lancer la résolution de chaque sous-tâche. Chaque sous-tâche est ainsi considérée comme une tâche optionnelle. Ceci permet à l'utilisateur de lancer seulement la partie intéressante des sous-tâches et de fournir à la main les résultats des sous-tâches non exécutées.
- Concernant la phase de *spécialisation,* l'utilisateur peut décider de contrôler le choix effectif de la spécialisation. C'est alors lui qui effectue, à la place du système, le choix de la classe modélisant la tâche précise à exécuter. Le contrôle du système se réduit ainsi à la détermination des classes sûres, possibles et impossibles en fonction des entrées fournies (cf. section 3.1.2.1.) et à l'exclusion des classes impossibles. Ainsi l'utilisateur peut imposer son choix lors de la sélection de la stratégie de résolution.
- Concernant la phase de *complétion et validation des entrées,* l'utilisateur peut demander d'être systématiquement consulté pour valider l'ensemble des entrées. Ceci lui permet d'examiner, avant chaque étape, les données à traiter et en conséquence par exemple d'interrompre le processus de résolution, si le traitement à effectuer sur ces données lui semble inutile ou impossible.
- Concernant la phase d'*exécution et décomposition,* l'utilisateur peut décider s'il veut contrôler les choix définis comme automatiques et/ou les choix définis comme choix-utilisateur dans les stratégies de résolution. Ainsi, il peut d'un côté abandonner ses responsabilités concernant un choix, a priori de type *choix-utilisateur*. De l'autre il peut prendre des responsabilités supplémentaires, concernant des choix de type *choix-automatiques*.
- Concernant la phase de *complétion et validation des sorties,* l'utilisateur peut demander à être systématiquement consulté pour valider l'ensemble des sorties. Ceci lui permet d'examiner à chaque étape les résultats obtenus, et d'intervenir dès que ceux-ci ne correspondent pas à ce qu'il attend.

3.1.1.3. Gestion des échecs

Nous avons vu jusqu'ici les différentes phases de résolution et la façon dont elles sont contrôlées et réalisées. En fait, l'exécution d'une tâche peut échouer dans chacune de ces phases, et cela pour différentes raisons :

- dans la phase d'*instanciation*, s'il y a des entrées globales dont la valeur reste indéterminée ;

- dans la phase de *spécialisation,* si aucune stratégie de résolution n'est applicable ou si l'utilisateur abandonne la spécialisation ;
- dans la phase de *complétion et validation des entrées,* si la valeur d'entrées stratégiques reste indéterminée, si une pré-condition n'est pas vérifiée ou si l'utilisateur ne valide pas l'ensemble des entrées ;
- dans la phase d'*exécution et décomposition,* si une des sous-tâches considérées comme obligatoires échoue ;
- dans la phase de *complétion et validation des sorties,* si la valeur de certaines sorties globales reste indéterminée, si une post-condition n'est pas vérifiée ou si l'utilisateur ne valide pas l'ensemble des sorties.

En cas d'échec, le moteur de tâches effectue automatiquement un retour arrière chronologique au dernier point de décision modifiable de façon autonome par le moteur, c'est-à-dire au dernier point de choix entre sous-tâches (opérateurs *choix* ou *choix-utilisateur*) ou à la dernière spécialisation.

En fait, les décisions concernant les valeurs des paramètres d'entrée qui sont prises par l'utilisateur, sont aussi des points de retour arrière possibles : non seulement un mauvais choix stratégique, mais aussi un mauvais choix d'une valeur de paramètre, peut être la cause d'un échec. Mais, nous avons considéré les décisions de l'utilisateur concernant des valeurs de paramètres comme a priori plus sûres qu'un choix prédéfini parmi plusieurs (sous-)tâches possibles. Par ailleurs, la modification de valeurs de paramètres demanderait obligatoirement une interaction avec l'utilisateur. Nous avons donné la priorité au retour arrière automatique. La figure 4 montre un exemple de situation de retour arrière.

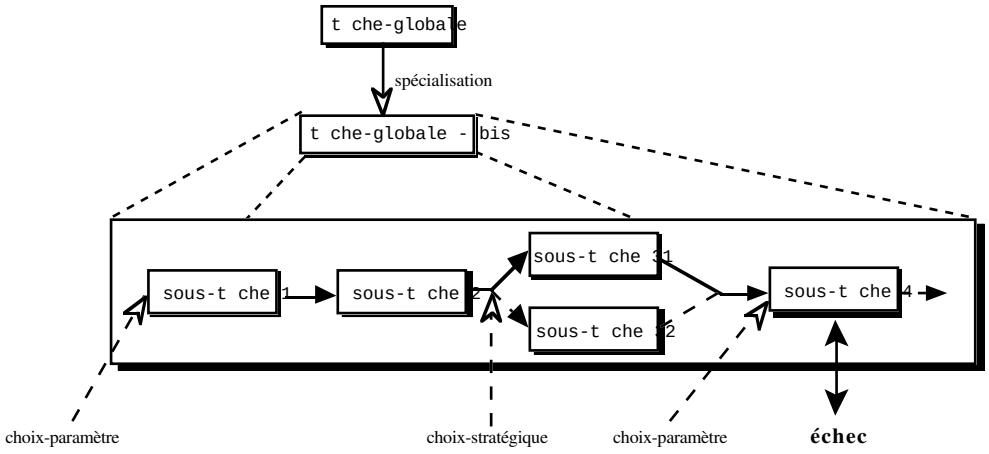


Figure 4 : Retour arrière. L'exécution d'une *tâche globale* est d'abord passée dans une phase de *spécialisation*. Ensuite *sous-tâche 1, sous-tâche 2 et sous-tâche 31* ont été résolues, mais *sous-tâche 4* échoue et il faut effectuer un retour arrière. Les points de retour arrière possibles sont la *spécialisation*, les *choix* de *paramètre*s pour *sous-tâche 1* et *sous-tâche 4* de la part de l'utilisateur et le *choix stratégique* entre *sous-tâche 31* et *sous-tâche 32*. Le moteur va ici revenir au *choix-stratégique* et essayer d'exécuter *sous-tâche 32*.

Nous avons choisi de gérer le retour arrière de façon simple et chronologique pour deux raisons. D'une part, aucune option de retour arrière intelligent, c'est-à-dire qui sélectionnerait selon certains critères un "meilleur" point de retour arrière, nous a semblée satisfaisante. D'autre part, et de toute façon, l'utilisateur peut à tout moment effectuer un retour arrière "artificiel" pour retourner à n'importe quel point de décision possible.

Un retour arrière intelligent pourrait être guidé par différents critères. Il pourrait modifier la dernière décision suffisamment importante pour re-diriger complètement le processus de résolution afin d'éviter de façon sûre l'exécution de la tâche qui a échoué. Une autre possibilité serait la remise en cause de la décision modifiable ayant le coût le plus faible, susceptible d'avoir contribué à l'échec. Aussi l'importance des données d'entrée pour le succès ou l'échec d'une tâche pourrait être prise en compte. Nous n'avons pour l'instant pas développé ces idées parce que les connaissances nécessaires pour pouvoir appliquer ces critères nous semblent difficiles à acquérir.

De plus, aussi l'utilisateur peut initier un retour arrière, et non seulement en situation d'échec, en modifiant n'importe laquelle des décisions prises pendant le processus de résolution (cf. section 3.2.2.). Une telle modification peut concerner tous les points de retour arrière possibles, outre un choix entre sous-tâches et une spécialisation aussi les valeurs de paramètres données par l'utilisateur, et même les résultats obtenues par une tâche.

3.1.1.4. Bilan

Nous avons présenté les différentes phases du processus de résolution de problèmes dans SCARP. Par défaut, ces phases sont exécutées sous le contrôle du moteur de tâches, mais nous avons montré comment l'utilisateur peut intervenir dans chacune de ces phases et ainsi prendre plus ou moins de responsabilités pendant le processus de résolution. Nous avons détaillé de même les raisons pour lesquelles la résolution d'une tâche peut échouer et comment le moteur de tâches gère de façon automatique de tels échecs.

Une planification hiérarchique est en particulier matérialisée par l'alternance des phases de spécialisation et de décomposition [Will94]. Ces deux phases sont donc les phases les plus importantes pour la résolution. C'est pourquoi nous allons les détailler par la suite.

3.1.2. Réalisation d'une planification hiérarchique

La phase de spécialisation exploite la modélisation des tâches par classes d'objets et leur organisation en hiérarchies de classes. Elle constitue une des originalités de SCARP, nous la présentons de façon très détaillée. Dans la phase de décomposition la tâche à résoudre est décomposée en sous-tâches.

3.1.2.1. Phase de spécialisation

La phase de spécialisation sert à préciser le plus possible le problème à résoudre et à choisir en fonction des données du problème une stratégie de résolution appropriée. Comme son nom l'indique, la phase de spécialisation exploite l'application du mécanisme de classement à l'organisation des tâches en hiérarchies de classes.

Rappelons que les différents niveaux d'une telle hiérarchie de classes reflètent les niveaux d'abstraction sur lesquels le problème correspondant peut être décrit. A partir du moment où la définition d'un problème est suffisamment spécifiée, une stratégie de résolution lui est associée. Si différentes stratégies de résolution existent pour résoudre un même problème général, chacune est appropriée à un contexte bien spécifique. Un contexte plus spécifique correspond à une description plus précise du problème et est ainsi modélisé par une sous-classe. La structure hiérarchique des classes modélisant les tâches reflète donc l'organisation des stratégies de résolution selon la spécificité du contexte auquel elles sont appropriées.

Au moment de la résolution d'un problème, les entités à traiter, c'est-à-dire les entrées globales, sont fournies. Les caractéristiques de ces entités sont ainsi connues dans une certaine mesure. La phase de spécialisation fonctionne de la manière suivante : le mécanisme de classement est appliqué sur la hiérarchie de classes correspondant au problème à résoudre. Il vérifie si les entités fournies en entrée sont compatibles avec les contraintes définies dans les différentes classes. La détermination des classes sûres, possibles et impossibles, revient dans ce contexte à distinguer parmi les sous-classes celles qui sont sûrement, éventuellement, et sûrement pas adaptées aux entités fournies en entrée (figure 5).

Une classe peut être possible pour deux raisons. Premièrement, la tâche modélisée par cette classe peut requérir des entrées supplémentaires, c'est-à-dire des entrées qui n'étaient pas définies dans la classe qui modélise le problème initial à résoudre. Leurs valeurs ne sont ainsi pas connues. Il est par conséquent impossible de vérifier si elles correspondent au type défini par la classe. Deuxièmement, la classe en question peut préciser le type de certaines entrées, et les connaissances sur les entrées fournies peuvent être insuffisantes pour vérifier, si elles satisfont ce type.

A la fin de la phase de spécialisation, une classe et une stratégie de résolution sont choisies. Ce choix est contraint par le fait que, pour pouvoir résoudre le problème en question, il faut sélectionner une classe sûre ou possible modélisant une tâche exécutable. L'instance est rattachée à cette classe et ensuite exécutée dans une phase de décomposition. Si, pendant la phase de spécialisation, des valeurs d'entrées manquent, elles sont fournies par l'utilisateur.

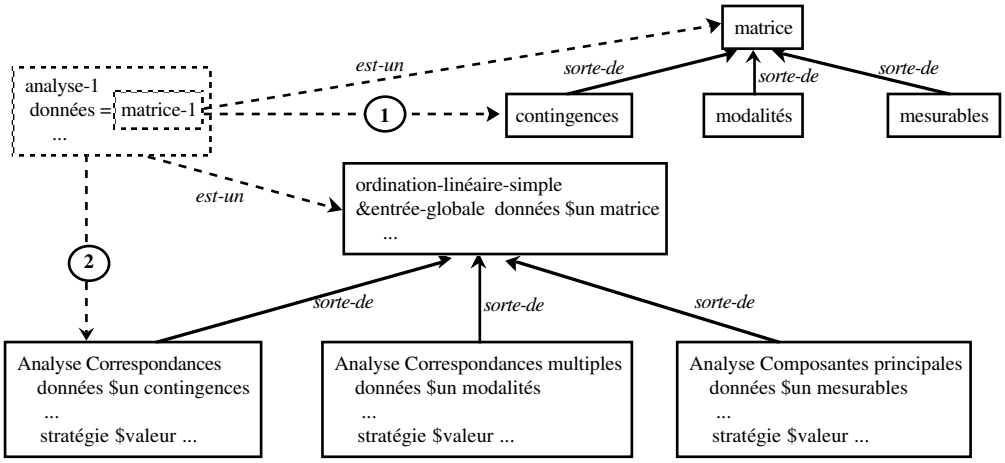


Figure 5 : La phase de spécialisation. La matrice *matrice-1* doit être analysée avec une analyse d'*ordination linéaire simple*. *Analyse-1*, une instance d'*ordination linéaire simple,* est donc créée et son entrée globale *données* reçoit comme valeur *matrice-1*. Le choix de la stratégie de résolution revient à déterminer quel type d'analyse plus précis est adapté pour traiter *matrice-1*. Pour cela il faut d'abord préciser le type de *matrice-1* : il s'agit d'un tableau de *contingences* (1) ; par conséquent, *Analyse Correspondances* avec sa stratégie associée est choisie (2).

La phase de spécialisation, plus précisément le choix de la classe précise modélisant la tâche à résoudre et celui de la stratégie de résolution à exécuter, peut être contrôlée soit par le moteur de tâches soit par l'utilisateur. L'intérêt de ces deux possibilités est évident : la première laisse le moteur gérer la résolution de problèmes de la façon la plus automatique et autonome possible, la deuxième permet à l'utilisateur d'imposer son choix au système.

Même si l'utilisateur contrôle la phase de spécialisation, il peut malgré tout demander de l'aide au système. Dans ce cas le système va proposer une spécialisation que l'utilisateur pourra accepter ou non. La seule contrainte a priori pour le choix d'une spécialisation est la suivante :

- Il faut choisir une tâche exécutable, sinon elle ne peut pas être résolue.

Si le moteur de tâches contrôle ou propose la spécialisation, il prend en compte un ensemble de critères de choix supplémentaires pour la classe de rattachement :

- Il ne faut pas choisir une tâche dont l'exécution a déjà été tentée. Ceci pour éviter un bouclage lors d'un retour arrière, de ré-appliquer une stratégie dont il est connu qu'elle ne conduit pas à une solution.
- Il faut choisir une tâche modélisée par une classe sûre. S'il n'y a pas de classes sûres mais des classes possibles, il faut choisir une classe possible et la transformer en classe sûre. Ceci est fait en demandant les valeurs d'entrée manquantes à l'utilisateur. Pour minimiser les incertitudes, il faut pour cela choisir la classe pour l'évaluation de laquelle il manque le moins d'informations, c'est-à-dire pour laquelle le nombre d'entrées globales inconnues est minimal. Les valeurs manquantes sont ensuite demandées a l'utilisateur.
- S'il y a le choix entre plusieurs tâches équivalentes selon les critères ci-dessus, il vaut mieux choisir la tâche la plus spécifique, c'est-à-dire celle qui est placée au niveau le plus bas dans la hiérarchie de classes, puisqu'elle contient une stratégie plus ciblée qu'une classe plus générale.

Une idée intéressante serait aussi d'appliquer une approche rationnelle comme le propose [Chai93], c'est-à-dire descendre dans la hiérarchie des classes si et seulement si le coût de la descente (des inférences à exécuter pour cela etc.) est inférieur au gain qui peut être obtenu par l'application d'une stratégie plus spécialisée. Mais, pour cela, il faudrait tout d'abord ajouter des mesures de coûts spécifiques dans le modèle de représentation des connaissances.

3.1.2.2. Phase de décomposition

La phase de spécialisation détermine une classe appropriée de rattachement pour l'instance représentant le problème à résoudre. Dans la phase de décomposition, la stratégie de résolution associée à cette classe est ensuite interprétée. Le moteur de tâches interprète les opérateurs qui décrivent la combinaison des étapes de résolution nécessaires. S'il s'agit d'un choix, seulement une des sous-tâches possibles est exécutée, sinon toutes les sous-tâches sont a priori obligatoires. Si le moteur contrôle le choix, il choisit au hasard une parmi toutes les sous-tâches possibles, sinon l'utilisateur effectue un choix.

Pour chaque étape de résolution à réaliser, le moteur interprète ensuite le contexte d'exécution correspondant pour déterminer si l'exécution de la sous-tâche référencée est obligatoire pour le succès de la tâche englobante. Ceci passe d'abord par l'interprétation des attributs *conditionnalité* et *condition* du contexte d'exécution. Si la sous-tâche est optionnelle, l'utilisateur décide si elle doit être exécutée. Si elle est conditionnelle, la condition associée est testée, pour savoir si elle doit être réalisée ou non. Si la sous-tâche est obligatoire, elle est exécutée.

Le contrôle de l'exécution se fait en fonction du contexte d'exécution, soit d'une itération, soit d'une tâche simple. Si le contexte référence une méthode, celle-ci est directement exécutée. S'il référence une tâche, l'exécution de cette tâche passe par les phases globales détaillées auparavant et ainsi récursivement par des phases de

spécialisation et de décomposition. L'utilisateur intervient dans la résolution de certaines sous-tâches, en particulier des tâches *utilisateurs* et des itérations *essai-erreur*.

3.1.2.2. Alternance des phases de spécialisation et de décomposition

La résolution de problèmes passe donc par une alternance de phases de spécialisation et de décomposition. Chaque (sous-)tâche n'est à son tour soumise à la phase de spécialisation qu'au début de sa propre exécution. Ainsi, la stratégie appropriée à sa résolution n'est choisie qu'au dernier moment. Le choix s'effectue donc en connaissant le contexte complet des données à traiter, c'est-à-dire aussi les résultats déjà obtenus. L'alternance des phases de spécialisation et de décomposition conduit ainsi finalement à un enchaînement très spécifique de méthodes qui mène à la solution du problème particulier (figure 6).

Comme nous avons vu dans le chapitre deux, nous proposons par ailleurs, via l'opérateur *spec,* d'utiliser la spécialisation pour préciser, pendant le processus de résolution, la stratégie de résolution adoptée. Dans ce cas, la spécialisation ne se fait pas seulement en fonction des entrées, mais aussi en fonction des spécialisations choisies pour les sous-tâches déjà exécutées. Ceci permet de limiter dans certains cas, à partir d'un moment précis, des choix ultérieurs possibles, car, le choix d'une spécialisation pour une sous-tâche peut dépendre des spécialisations déjà effectuées pour les sous-tâches précédentes. Dans ces cas, l'opérateur *spec* est introduit dans la stratégie de résolution (cf. section 2.3). Lorsque le moteur de tâche rencontre cet opérateur pendant la phase d'exécution de la stratégie de résolution, il l'interprète par une phase de spécialisation.

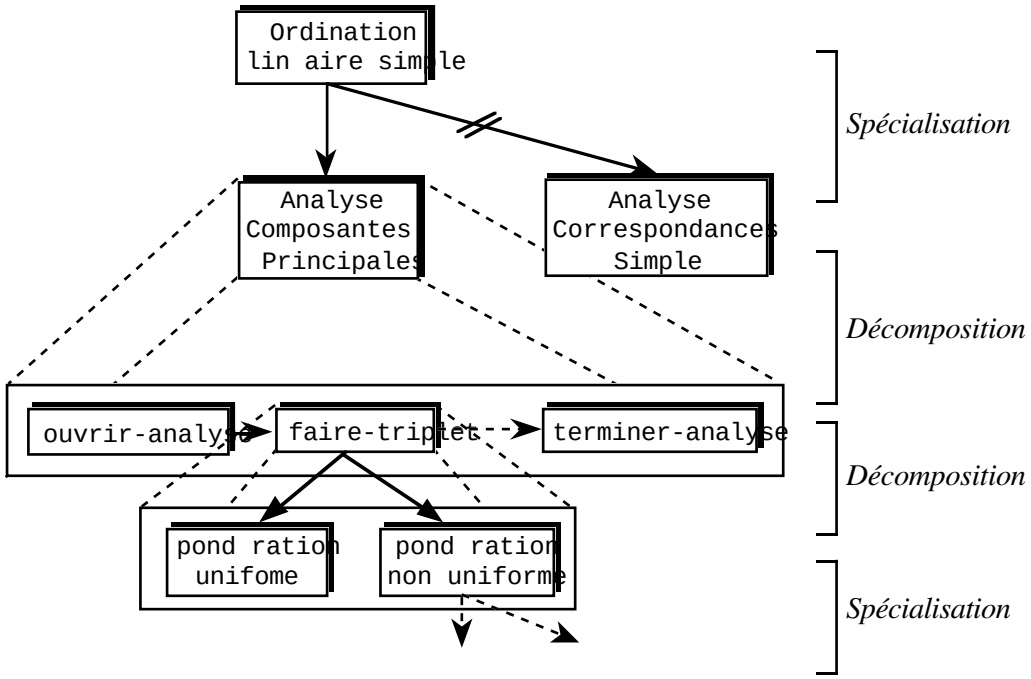


Figure 6 : Alternance des phases de spécialisation et de décomposition. Une *Ordination linéaire simple* doit être effectuée. Selon des caractéristiques des entrées globales à traiter dans la phase de *spécialisation*, une *Analyse en composantes principales* est choisie. Cette tâche est *décomposée* selon la stratégie associée. Ses sous-tâches sont à leur tour exécutées. La classe qui décrit la tâche *faire-triplet* n'a pas de sous-classes ; ainsi elle est directement *décomposée* en un choix entre *pondération uniforme* et *pondération non-uniforme*. Cette dernière est choisie et l'instance créée pour cette tâche est soumise à une phase de *spécialisation*.

3.1.3. Séparation de la gestion de la résolution et de l'interaction

Dans les sections précédentes, nous avons présenté comment la résolution de problèmes est gérée par le moteur de tâches. Celui-ci contrôle par défaut toutes les phases de résolution. S'il le souhaite, l'utilisateur peut prendre le contrôle des différentes phases et, par là, de toutes les décisions associées à celle-ci. L'utilisateur a ainsi la possibilité de contrôler non seulement les décisions qui lui sont attribuées a priori, par la définition des tâches dans la base de connaissances, mais de contrôler systématiquement toutes les décisions d'un certain type. Ceci lui permet de modifier la distribution des responsabilités prédéfinie dans la base de connaissances et constitue ainsi un premier pas vers un système coopératif.

Dans les sections précédentes, nous avons montré à quels endroits les interactions a priori nécessaires interviennent dans le processus de résolution : le contrôle d'un *choix-utilisateur* et la résolution d'une tâche *utilisateur* et d'une itération *essai-erreur* dans la phase de décomposition, la nécessité de *valider* ou de *visualiser* l'ensemble des entrées (sorties) d'une (sous-)tâche dans la phase de complétion et validation des entrées (sorties) et le contrôle de l'exécution d'une tâche *optionnelle* dans la phase d'instanciation. La demande de valeurs manquantes de paramètres à l'utilisateur et le contrôle de la phase de spécialisation par l'utilisateur s'ajoutent à cette liste.

Pendant la résolution de problèmes, diverses interactions système-utilisateur peuvent donc être obligatoires. Mais toutes les interactions dont nous avons discuté jusqu'ici sont, soit prévues dans la modélisation des connaissances, soit systématiques lors du passage dans une certaine phase de résolution. Ainsi toutes ces interactions pourraient être gérées de la même façon : dès que le moteur de tâches rencontre une interaction prédéfinie ou une décision à prendre par l'utilisateur, il communique sa demande via une fenêtre d'interaction à l'utilisateur. Celui-ci répond et ainsi le moteur peut continuer la résolution avec cette information au même endroit (figure 7).

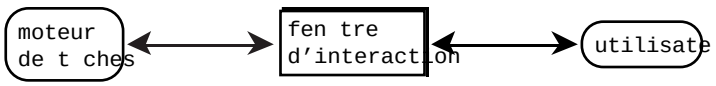


Figure 7 : Interaction - demande du moteur à l'utilisateur. Si toutes les interactions sont prédéfinies ou systématiques, le moteur est capable de les gérer de façon simple.

Mais pour plusieurs raisons, cette vue simple n'est pas satisfaisante dans le cadre d'un système coopératif. Premièrement, un utilisateur peut ne pas pouvoir répondre directement et avoir besoin d'informations supplémentaires. Deuxièmement, nous avons exigé d'un système coopératif que l'utilisateur puisse intervenir à tout moment dans le processus de résolution et modifier toutes les décisions déjà prises. Troisièmement, pour l'exploration de nouvelles tâches, l'utilisateur a besoin d'interagir différemment avec le moteur de tâches. L'initiative d'une interaction système-utilisateur peut ainsi venir de l'utilisateur et le dialogue doit dans ce cas être géré différemment.

C'est pourquoi nous avons séparé le raisonnement sur la résolution de problèmes avec le moteur de tâches du raisonnement sur le dialogue système-utilisateur avec le gestionnaire de dialogues. Celui-ci ne gère pas seulement les interactions pour lesquelles l'initiative vient du moteur, mais aussi bien toutes les interactions pour lesquelles l'initiative vient de l'utilisateur.

3.2. GESTIONNAIRE DE DIALOGUES

Nous avons identifié trois fonctionnalités principales pour le gestionnaire de dialogues :

- la gestion des interactions nécessaires pendant le processus de résolution, initialisées par le moteur de tâches.
- la gestion de la modification par l'utilisateur de décisions prises pendant le processus de résolution.
- la gestion de l'exploration de nouvelles tâches par l'utilisateur.

Nous allons discuter de chacune d'entre elles. Nous résumons d'abord les différents types d'interaction qui peuvent de toute façon s'imposer pendant le processus de résolution, soit parce qu'elles sont prédéfinies, soit parce que l'utilisateur contrôle un certain type de décisions. Ensuite, nous montrons comment toute les décisions importantes prises pendant la résolution peuvent être remises en cause. Chaque modification entraîne en fait la création d'une nouvelle version du processus de résolution. Ces versions doivent être gérées par le gestionnaire de dialogues et rester accessibles pour l'utilisateur. Finalement, nous décrivons le protocole d'interaction nécessaire pour l'exploration de nouvelles tâches.

3.2.1. Interactions nécessaires pour la résolution

Dans la première section de ce chapitre, nous avons vu que diverses interactions système-utilisateur peuvent être nécessaires pendant la résolution de problèmes. Nous pouvons distinguer différents types :

- le choix (initial) d'un problème à résoudre,
- l'obtention de valeurs manquantes pour les paramètres d'entrée,
- la spécialisation d'une tâche,
- la décision d'exécuter une tâche optionnelle,
- le *choix* d'une sous-tâche parmi plusieurs possibles,
- la visualisation d'une application,
- la visualisation ou la validation de l'ensemble des entrées ou des sorties,
- la résolution d'une tâche *utilisateur*,
- la résolution d'une tâche par *essai-erreur*.

En dehors des simples visualisations, toutes ces interactions représentent un type de demande à l'utilisateur et attendent un type de réponse bien précis. Chaque type d'interaction est matérialisé dans SCARP par une fenêtre d'interaction spécifique. Cette fenêtre permet à l'utilisateur non seulement de répondre à la demande du système, mais aussi d'accéder à des informations complémentaires (cf. section 3.3.1.).

Le protocole d'interaction est ainsi simple (figure 8) : selon le type de demande du moteur, le gestionnaire de dialogues choisit la fenêtre d'interaction correspondante et la remplit avec les informations correspondant au contexte de la demande (la tâche concernée, etc.). L'utilisateur accède éventuellement à des informations complémentaires, indique et valide la réponse. Cette validation est contrôlée par le gestionnaire de dialogues qui passe la réponse au moteur de tâches. Une annulation de l'interaction de la part de

l'utilisateur ou une réponse non conforme à la demande du moteur conduit à un échec de la résolution en cours.

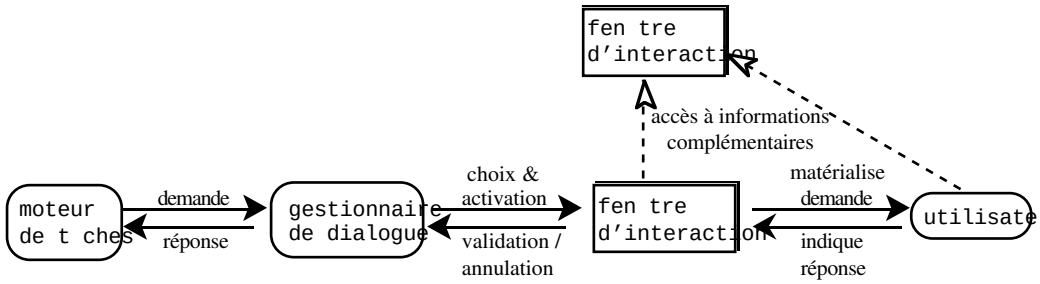


Figure 8 : Interaction initialisée par une demande du moteur.

3.2.2. Modification de décisions par l'utilisateur

Une des caractéristiques importantes que nous avons identifiées pour un système coopératif d'aide à la résolution de problèmes est de pouvoir remettre en cause, à tout moment, toutes les décisions déjà prises pendant le processus de résolution. Ceci permet de gérer le processus de résolution d'une manière souple a posteriori : une décision n'est jamais prise définitivement, elle peut toujours être remise en cause.

La remise en cause concerne tout d'abord toutes les décisions prises par l'utilisateur : il peut s'être trompé dans le choix de la valeur de paramètres par exemple. Une telle erreur doit être facile à corriger. Mais, comme l'utilisateur peut avoir des connaissances plus importantes que le système, il doit, de plus, pouvoir modifier les décisions prises par le système.

De la même façon qu'il doit être facile de modifier des décisions, il doit être facile de revenir sur les modifications. Ainsi, l'état de la résolution avant chaque modification doit être sauvegardé, afin de rester accessible à l'utilisateur. Par ailleurs, il doit être possible d'effectuer des modifications seulement localement, "juste pour voir" par exemple, et de les valider seulement en fonction de leurs résultats.

Nous allons tout d'abord identifier l'ensemble des décisions qui doivent être modifiables dans SCARP. Ensuite nous présentons les conséquences d'une modification et la façon dont ces modifications sont créées, stockées et gérées de façon cohérente.

3.2.2.1. Décisions modifiables

Toutes les décisions importantes, prises pendant le processus de résolution, doivent être modifiables. Dans SCARP, la résolution de problèmes passe par les phases détaillées à la section 3.1. Les décisions importantes prises pendant ce processus sont :

- Le choix de valeurs pour les paramètres d'entrée[1].
- Le choix d'une spécialisation.
- Le choix stratégique d'une sous-tâche.
- Le choix d'une valeur comme résultat d'une tâche utilisateur.

[1] Pour garantir la cohérence de la résolution, ceci ne concerne que les paramètres d'entrée renseignés par l'utilisateur.

Nous considérons que l'utilisateur peut avoir des connaissances complémentaires non présentes dans la base de connaissances. Par conséquent il doit aussi pouvoir transformer une tâche résolue par le système en tâche-utilisateur et modifier :

- Les résultats d'une tâche.

Si l'utilisateur modifie les résultats d'une tâche, cela signifie en fait qu'il remplace la stratégie de résolution exécutée auparavant par le moteur par sa propre stratégie de résolution. La transformation d'une tâche en tâche-utilisateur se fait sans indication d'une nouvelle stratégie. Nous verrons dans la section 3.2.3. comment l'utilisateur peut explorer de façon interactive de nouvelles stratégies de résolution.

3.2.2.2. Protocole d'interaction

Le protocole d'interaction pour effectuer une modification est, comme celui défini pour les interactions initialisées par le moteur de tâches, simple en soi. Mais, l'interaction est ici initialisée par l'utilisateur : celui-ci choisit la décision qu'il veut modifier, ce qui active la fenêtre d'interaction correspondant au type de décision. Cette fenêtre matérialise la décision actuellement prise. L'utilisateur peut ensuite indiquer son nouveau choix et le valider. La modification est prise en compte par le gestionnaire de dialogues qui contrôle toutes les interactions système-utilisateur. Il gère de façon cohérente la modification effective, c'est-à-dire la création d'une nouvelle version du processus de résolution. Il arrête le moteur de tâches si la résolution de la version précédente n'était pas encore terminée. Ensuite, il lui passe la nouvelle version et le relance pour gérer le processus de résolution modifié (figure 9).

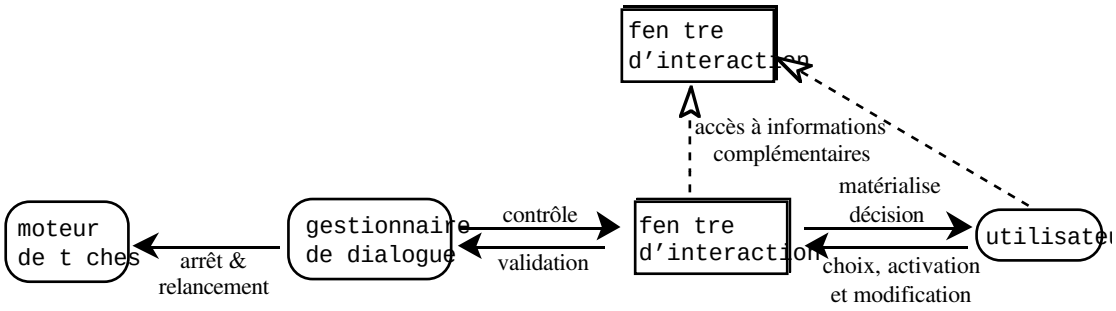


Figure 9 : Interaction initialisée par l'utilisateur.

3.2.2.3. Versions du processus de résolution

La remise en cause de décisions comme décrite ci-dessus correspond à un retour arrière dans le processus de résolution jusqu'à l'état où la décision initiale avait été prise. C'est un retour arrière artificiel, initalisé et contrôlé par l'utilisateur. Contrairement à un retour arrière dans le cas d'un échec, ici l'état de résolution n'est pas invalide. Il doit être gardé tel quel, pour pouvoir y revenir, c'est-à-dire annuler la modification. Chaque modification crée donc en quelque sorte une résolution parallèle à la résolution initiale. Nous appelons *versions* ces résolutions parallèles. Des modifications successives conduisent à la création d'une hiérarchie de versions (figure 10).

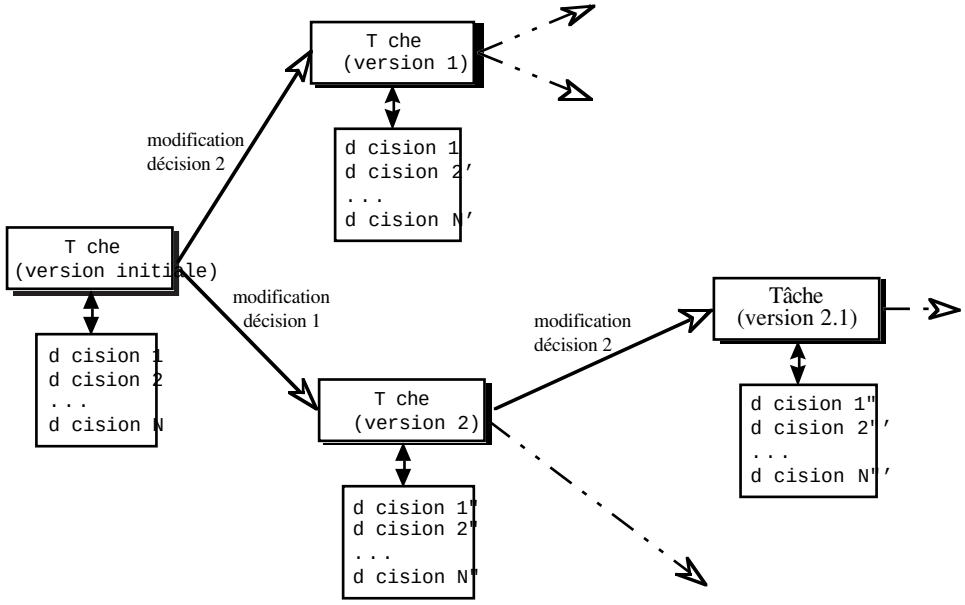


Figure 10 : Hiérarchie de versions. Chaque version de résolution d'une tâche correspond à une suite de décisions prises pendant le processus de résolution. La modification d'une de ces décisions signifie un retour arrière à l'état de résolution précédant cette décision et à la reprise du processus de résolution à partir de là avec la décision modifiée. Chaque nouvelle version peut être une version d'origine pour de nouvelles modifications. Ainsi une hiérarchie de versions est créée.

Le système, plus précisément le gestionnaire de dialogue, doit garantir la cohérence de l'état de résolution de ces différentes versions. Ceci concerne deux aspects :

- l'interruption cohérente de la résolution en cours, c'est-à-dire de la version qui doit être modifiée ;
- la création cohérente de la nouvelle version.

Le premier aspect correspond au problème suivant : l'utilisateur peut décider une modification du processus de résolution *à tout moment*, même si une résolution est en cours, si une tâche n'est que partiellement exécutée, si un calcul externe est en cours ou si une demande à l'utilisateur est activée. Dans ce cas, le processus de résolution en cours doit être interrompu et ramené dans un état cohérent, plus précisément dans l'état correspondant au début de l'action inachevée. Par conséquence, les résultats du calcul en cours ne sont plus pris en compte, ou la demande à l'utilisateur est abandonnée.

Le deuxième aspect résulte du fait que pour pouvoir effectuer une modification, il faut tout d'abord revenir dans l'état exact où la décision, qui doit être remise en cause, avait été prise. Il faut donc, pour la nouvelle version, annuler toute la partie de la résolution effectuée chronologiquement après cette décision. Ceci concerne non seulement les conséquences immédiates de cette décision mais aussi les sous-tâches qui ont été résolues et toutes les décisions qui ont été prises pendant la suite de la résolution. Souvent l'utilisateur ne veut en fait modifier qu'une décision précise ponctuellement. Les décisions prises ultérieurement peuvent indépendamment toujours être valables. Par conséquent SCARP propose à l'utilisateur de reprendre, pendant la résolution de la nouvelle version, les choix effectués dans la version précédente.

3.2.2.4. Versions globales et versions locales

Dans certains cas, l'utilisateur ne veut pas tout de suite et définitivement modifier la résolution globale, mais seulement provisoirement modifier une décision pour voir ce que cette modification donne comme résultat. Dans ce cas, la version créée reste locale à la sous-tâche concernée. Seule cette sous-tâche est ré-exécutée. Ensuite, en fonction du résultat obtenu, l'utilisateur décide, si son nouveau choix est meilleur et s'il veut l'intégrer dans une version globale nouvelle (figure 11).

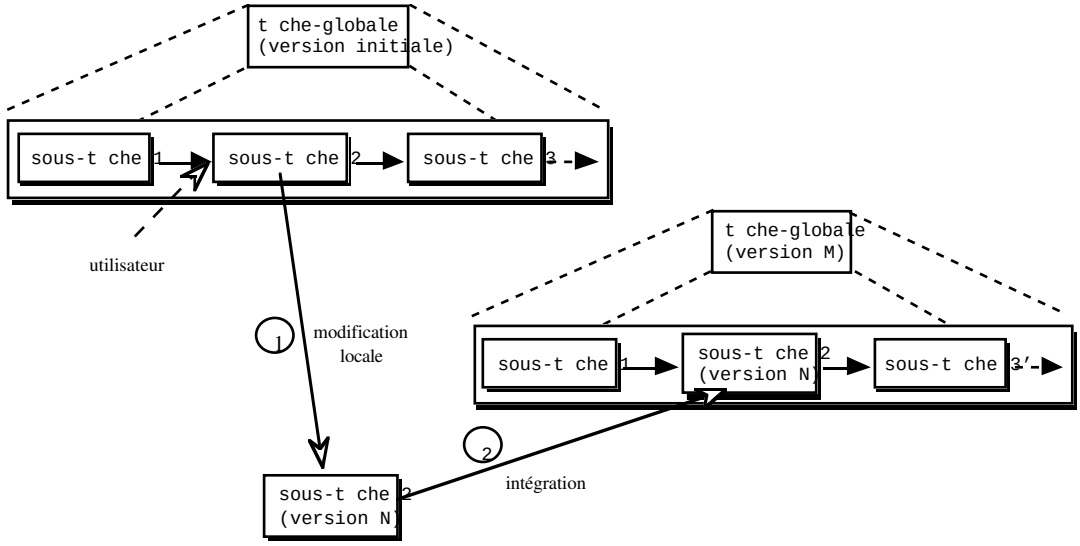


Figure 11 : Création d'une version locale et son intégration. Dans la version initiale de la tâche globale, l'utilisateur avait renseigné une valeur d'entrée de la sous-tâche 2. En cours du processus de résolution, il s'aperçoit que la valeur qu'il avait choisie n'était peut-être pas la meilleure ; il la modifie seulement localement pour voir ce que la sous-tâche 2 donne en sortie suite à cette modification ; il crée ainsi une version locale de cette sous-tâche (1). Finalement il préfère le nouveau résultat et décide d'intégrer la nouvelle version de la sous-tâche 2 dans une nouvelle version globale (2).

Même si la version n'est pas intégrée dans une version globale, mais reste une version locale à la (sous-)tâche précise qui a été modifiée, elle ne doit pas être perdue mais rester accessible pour l'utilisateur. Ainsi il pourra la comparer avec d'autres versions et éventuellement y revenir et l'intégrer ultérieurement dans une nouvelle version globale.

Chaque modification et ainsi chaque création de version concerne une (sous-)tâche précise dans la décomposition. A chaque niveau de décomposition, des versions locales peuvent être créées. Aussi à l'intérieur de chaque version locale des versions locales à celle-ci peuvent récursivement être créées. Ceci conduit à l'intérieur de la version globale initiale à la présence de hiérarchies de versions à plusieurs niveaux qui doivent être accessibles à l'utilisateur. Nous verrons dans la section 3.3.2.3. comment cet accès est géré dans SCARP.

3.2.2.5. Création de versions par le gestionnaire de dialogues

La création d'une version par l'utilisateur revient à choisir une décision préalablement prise et à la modifier. Etant donné notre modélisation des connaissances et celle du processus de résolution, ce processus est directement matérialisé par un arbre de

spécialisation et de décomposition dont les nœuds représentent les instances des (sous-) tâches résolues. Chacune des décisions modifiables concerne la résolution d'une (sous-) tâche précise. La création d'une version est ainsi matérialisée par une reprise de l'arbre d'instances à partir de l'instance concernée par la modification. Une nouvelle instance de cette tâche est créée, tenant compte de la modification effectuée sur cette instance. Puis, la tâche modifiée est résolue.

Si, une fois créée, cette version doit être intégrée dans une version plus globale, la partie de l'arbre d'instances de la version mère, antérieure à la modification, est recopiée. Une nouvelle version pour la tâche englobante est créée. Ainsi, automatiquement, toutes les décisions prises *avant* l'exécution de la tâche modifiée sont reprises telles quelles. De même, les décisions prises *après* l'exécution de la tâche modifiée sont accessibles par la version mère et peuvent par conséquent être proposées à l'utilisateur.

3.2.3. Exploration de nouvelles tâches

La possibilité de remettre en cause des décisions prises pendant le processus de résolution permet globalement de transformer la résolution de problèmes en un processus d'essai-erreur. Mais pour mener un travail de recherche, il faut aussi donner à l'utilisateur la possibilité de construire de façon interactive de nouvelles tâches. Un tel travail consiste dans la définition interactive de nouveaux problèmes et dans l'exploration interactive de nouvelles stratégies de résolution. Il doit permettre de définir une nouvelle tâche "à partir de rien" ou alors en reprenant des définitions existantes et en changeant des aspects précis.

Les caractéristiques essentielles d'une tâche sont :

- la définition de ses données d'entrée,
- la définition de ses données de sortie,
- la définition de ses sous-tâches,
- la définition de la stratégie, c'est-à-dire du flot de contrôle sur ses sous-tâches,
- la définition du flot de données entre la tâche et ses sous-tâches.

Pour définir de façon interactive des tâches nouvelles il est donc nécessaire de pouvoir indiquer et modifier toutes ces caractéristiques. L'exploration de tâches concerne donc plusieurs caractéristiques à la fois et est ainsi beaucoup plus complexe qu'une modification ponctuelle de décisions.

La création de versions consiste à modifier ponctuellement une instance de tâche précise et à créer de nouvelles instances à partir d'un arbre d'instances existantes. Mais, de telles versions restent toujours conformes à la définition de la tâche correspondante dans la base de connaissances. Toutes ces modifications, même la transformation d'une tâche en tâche-utilisateur qui n'est rien d'autre que l'annulation ou la suppression de la stratégie de résolution exécutée, peuvent être gérées en utilisant directement les structures de classes définies dans la base de connaissances. L'exploration de nouvelles tâches par contre, si celles-ci doivent être ré-exécutables et intégrées dans la base de connaissances, requiert une *création de nouvelles classes*, éventuellement à partir de classes existantes.

3.2.3.1. Protocole d'interaction de base

L'exploration de tâches requiert un protocole d'interaction plus élaboré que celui qui a été défini pour la création de versions qui ne concerne qu'une décision précise du

processus de résolution. Pour limiter la complexité du protocole d'interaction, nous avons décidé de modéliser l'exploration d'une nouvelle tâche en trois étapes séparées :

- au début l'utilisateur choisit la tâche de départ dont il va modifier les caractéristiques ;
- ensuite l'utilisateur modifie au fur et à mesure les caractéristiques de cette tâche, une caractéristique à la fois : ses entrées / sorties, ses sous-tâches et sa stratégie de résolution (figure 12). Aussi la définition de toutes les tâches référencées en tant que sous-tâches peut elle être récursivement modifiée de la même manière. Ainsi des stratégies plus complexes peuvent être définies ;
- finalement, une fois ces caractéristiques complètement définies, elle est exécutée, de manière à obtenir par un exemple de résolution le flot de données entre la tâche complexe et ses sous-tâches. En effet, ce flot est pendant cette exécution, indiqué par l'utilisateur à l'aide de la souris, puis interprété par le système pour le formalisé correctement et l'associer à la définition de classe de la nouvelle tâche.

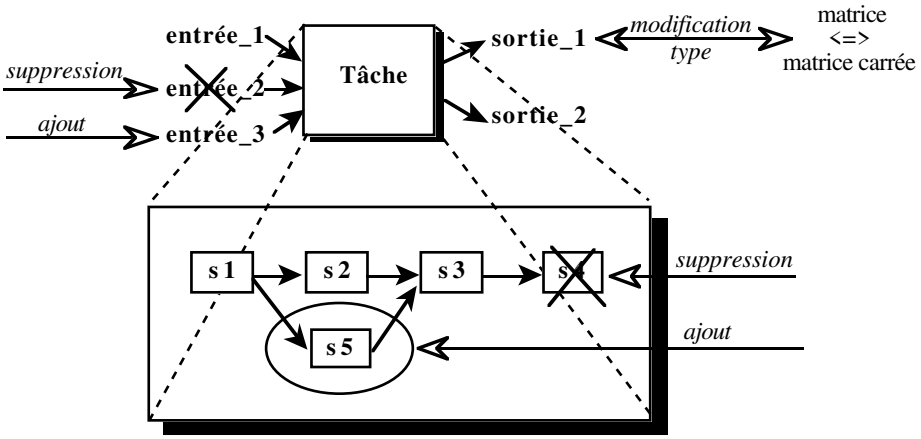


Figure 12 : Modification des caractéristiques d'une tâche. Dans la deuxième étape d'exploration d'une tâche ses entrées, ses sorties, ses sous-tâches et sa stratégie de résolution sont redéfinies. Dans cet exemple, il y a suppression de l'*entrée_2* et de la sous-tâche *s4* ; l'*entrée_3* et la sous-tâche *s5* sont ajoutées et le type de la *sortie_1* est modifié. L'ajout de *s5* introduit par ailleurs un choix entre *s2* et *s5* dans la stratégie.

Pour l'instant les phases de définition des caractéristiques de la tâche ne peuvent pas être mélangées avec son exécution et l'acquisition du flot de données associé. De même, il n'est encore pas possible d'explorer une nouvelle tâche à l'intérieur d'une résolution de problèmes en cours, pour résoudre une sous-tâche précise, c'est-à-dire pour remplacer sa stratégie prédéfinie par une stratégie définie et explorée interactivement. Par ailleurs, nous nous ne permettons pas pour l'instant de définir interactivement des itérations.

3.2.3.2. Intégration des tâches dans la base de connaissances

L'exploration de nouvelles tâches conduit à la création de nouvelles classes dans la base de connaissances. Pour réaliser l'exploration, les trois étapes décrites auparavant pour le protocole d'interaction correspondent à différentes actions de création et de modification de classes.

La première étape correspond à la création d'une classe modélisant la tâche qui va être explorée. Elle est créée en recopiant la classe modélisant la tâche d'origine et en la

rattachant en tant que sous-classe à la classe *tâche-utilisateur*, prédéfinie dans la base de connaissances SCARP. La deuxième étape correspond à l'ajout, la modification du type ou l'élimination d'attributs des classes entrée, sortie et sous-tâche. De même, la valeur de l'attribut stratégie, c'est-à-dire la stratégie de résolution de la tâche explorée, est modifiée. L'attachement de sous-tâches à une tâche nécessite de plus la définition de nouveaux contextes d'exécution qui contiennent le flot de données entre la tâche englobante et les sous-tâches.

Les tâches interactivement explorées de cette façon sont intégrées dans la base de connaissances, regroupées sous la classe *tâche-utilisateur* et accessibles et ré-exécutables par l'utilisateur. Mais cette intégration est pour l'instant incomplète. En effet, les connaissances de résolution de problèmes sont organisées de façon hiérarchique, à différents niveaux d'abstraction, afin de pouvoir les exploiter par le mécanisme de planification hiérarchique décrite au début de ce chapitre. C'est la phase de spécialisation qui exploite cette organisation hiérarchique pour choisir une stratégie de résolution appropriée au problème à résoudre. Etant donné que les tâches explorées de façon interactive ne sont pas intégrées "à leur place" dans cette organisation, elle ne sont ici pas accessibles.

Il faudrait donc placer ces classes à un endroit approprié dans la hiérarchie correspondante. Ce travail reste à faire. Nous proposons de partir pour cela d'algorithmes de classement de classes [Capp93] ou de reconnaissances de plan [Naul&91]. Ces derniers exploitent l'organisation hiérarchique des tâches, la définition du flot de contrôle [Kaut&86] [Kaut87] et du flot de données [Lin&91]. L'appariement entre la nouvelle classe et les classes déjà existantes pourrait se faire de façon optimale en exploitant d'abord les caractéristiques essentielles des tâches examinées, par exemple considérer la définition des entrées (sorties) globales avant celle des entrées (sorties) stratégiques, et celle des sous-tâches avant celles des pré- et post-tâches. La reconnaissance de plans est naturellement limitée aux plans explicitement modélisés. Pour pouvoir "reconnaître" de nouveaux plans autres que des combinaisons de plans déjà modélisés, des techniques de généralisation ou de raisonnement par analogie devraient être développées et intégrées [Beau94].

L'algorithme de placement, de mise en correspondance de tâches explorées de façon interactive et des tâches connues dans la base de connaissances, peut aussi servir pour aider un utilisateur, explorateur d'une nouvelle stratégie, si celui-ci est perdu dans la résolution, c'est-à-dire dans la définition de la stratégie : si des ressemblances entre la tâche explorée en partie et des tâches dans la base de connaissances peuvent être détectées, ces ressemblances peuvent être utilisées pour proposer des actions appropriées à l'utilisateur.

3.2.4. Gestion globale des interactions

Dans cette section, nous avons détaillé les différentes fonctionnalités du gestionnaire de dialogues de SCARP. Il contrôle tout d'abord toutes les communications entre le moteur de tâches et l'utilisateur pendant le processus de résolution. Ce sont d'une part les interactions initialisées par le moteur de tâches, prédéfinies ou nécessaires parce que l'utilisateur a pris le contrôle de certaines phases de résolution. Ce sont d'autre part les interactions initialisées par l'utilisateur, pour modifier des décisions prises pendant le processus de résolution ou pour explorer interactivement de nouvelles tâches. Suite à ce deuxième type d'interaction, le gestionnaire de dialogues gère aussi la création de versions du processus de résolution et l'introduction de nouvelles classes dans la base de connaissances.

En fait, plusieurs interactions peuvent se dérouler en parallèle, initialisées par le moteur pour demander des informations, et initialisées par l'utilisateur pour obtenir des

informations sur des décisions prises ou pour modifier ces décisions ou alors la stratégie de résolution. Le gestionnaire de dialogues contrôle l'ensemble de toutes les interactions possibles et garantit ainsi la cohérence globale du processus de résolution (figure 13).

Nous avons jusqu'ici identifié les différents types d'interactions nécessaires pour une résolution coopérative de problèmes qui sont gérées par le gestionnaire de dialogues. Mais pour rendre un système coopératif, il ne suffit pas d'identifier ces interactions et de savoir gérer leurs conséquences. Il faut aussi concevoir l'interface adéquate par laquelle ces interactions peuvent passer. Par la suite, nous allons donc détailler la conception et illustrer le fonctionnement de l'interface que nous avons définie pour SCARP.

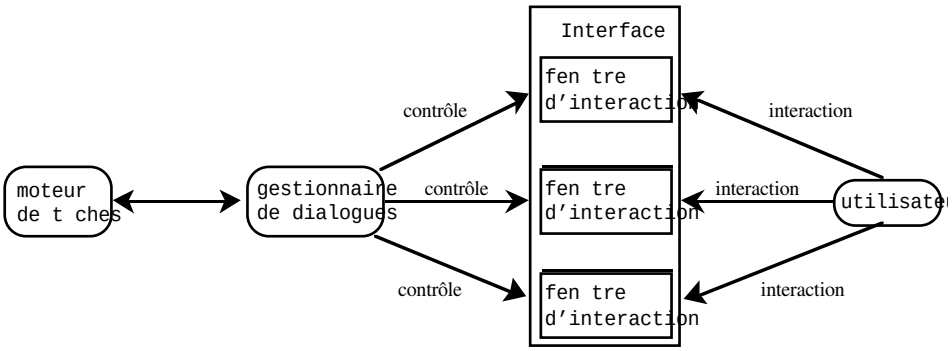


Figure 13 : Toutes les interactions sont sous le contrôle du gestionnaire de dialogues. L'ensemble des fenêtres d'interaction forment l'interface du système. Une fenêtre d'interaction n'en bloque jamais d'autres : premièrement, pour répondre à une interaction spécifique, l'utilisateur peut avoir besoin de demander des informations accessibles par ailleurs. Deuxièmement, l'utilisateur peut avoir envie de remettre en cause des décisions prises auparavant. Troisièmement, l'utilisateur doit toujours être en mesure d'abandonner une résolution en cours pour en entamer une autre. L'ensemble des interactions est contrôlé par le gestionnaire de dialogues pour éviter des incohérences, par exemple la validation simultanée de deux modifications différentes d'un même processus de résolution.

3.3. CONCEPTION ET FONCTIONNEMENT DE L'INTERFACE

L'interface-utilisateur du système SCARP doit permettre de réaliser la coopération système-utilisateur pendant le processus de résolution. Elle a différentes fonctionnalités : d'une part elle doit permettre de réaliser les différents types d'interactions ponctuelles, nécessaires pendant la résolution de problèmes (demande de valeurs de paramètres, demande de choix stratégique, etc.). D'autre part, elle doit matérialiser le processus de résolution de problèmes pour l'utilisateur et permette à l'utilisateur d'intervenir convenablement dans ce processus.

Nous distinguons ainsi deux types de fenêtres dans SCARP. Premièrement il y a un ensemble de *fenêtres spécifiques*, permettant chacune de réaliser une interaction ponctuelle, c'est-à-dire de prendre ou modifier un type de décision à prendre pendant le processus de résolution. Deuxièmement il y a une *fenêtre principale* qui permet à l'utilisateur d'évaluer le processus de résolution et, si nécessaire, d'initialiser des interactions adéquates avec le système pour le modifier.

Nous présentons d'abord l'approche générale pour la conception de fenêtres matérialisant des interactions ponctuelles. Nous en donnons deux exemples. Puis, nous détaillons la conception et le fonctionnement de la fenêtre d'interaction principale de

SCARP. Celle-ci permet en fait à l'utilisateur de gérer et de diriger globalement la résolution de problèmes.

3.3.1. Réalisation des interactions ponctuelles

Les interactions ponctuelles concernent chacune des décisions d'un type spécifique pendant le processus de résolution. Il s'agit soit de la prise initiale de cette décision, auquel cas l'interaction est initialisée par le moteur de tâches, soit de la modification d'une décision déjà prise, auquel cas l'interaction est initialisée par l'utilisateur (ou par le moteur lors d'un retour arrière). Ces décisions sont matérialisées par différentes fenêtres d'interaction.

Chaque type de décision concerne une information spécifique. La prise de la décision peut être liée à d'autres informations associées, à la situation où il faut la prendre. La fenêtre matérialisant l'interaction ponctuelle qui permet de prendre ou de modifier une décision ne doit donc pas uniquement contenir la décision proprement dite, mais aussi donner accès à toutes les informations qui lui sont liées. La figure 14 montre pour les différents types d'interaction que nous avons identifiés pour SCARP, quelles décisions il permet de prendre ou de modifier, et quelles informations sont nécessaires pour prendre cette décision.

Nous allons maintenant donner deux exemples de la conception et du fonctionnement des fenêtres d'interaction ponctuelle. Nous avons choisi de montrer les interactions ponctuelles les plus complexes : le choix d'un problème dans la base de connaissances et la spécialisation.

Fenêtres d'interaction ponctuelle :

Type d'interaction spécifique	**décision demandée ou modifiable**	**informations qui doivent être directement accessibles**
choix d'un probl me	classe modélisant le problème	définition des entrées / sorties, hiérarchie de classes, décomposition
choix de la valeur d'un param tre	valeur	type, contraintes sur la valeur, instances existantes [Griv92]
sp cialisation	spécialisation	données actuelles à traiter, classes sûres, possibles et impossibles, versions déjà existantes (succès / échec ?)
visualisation	-	(exécution de la tâche de) visualisation
choix strat gique	sous-tâche	sous-tâches possibles / déjà essayées (succès / échec ?)
validation des entr es / sorties	validation : oui ou non	(exécution de la tâche de) visualisation des entrées / sorties de la sous-tâche
r solution d'une t che utilisateur	valeur	(exécution de la tâche de) visualisation des sorties de la tâche d'aide
contr le d'une it ration essai-erreur	validation ou modification de la valeur d'entrée actuelle	(exécution de la tâche de) visualisation des sorties de la tâche itérative
modification du type d'un attribut	classe modélisant le type	hiérarchie de classes associée au type actuel
ajout d'un attribut	nom et type de l'attribut	hiérarchies de classes existantes

Figure 14 : Fonction et conception des fenêtres d'interaction ponctuelle. A part la visualisation, chaque interaction ponctuelle concerne un type de décision précis à prendre. Pour chacune une fenêtre doit être conçue qui matérialise d'un côté les informations demandées et modifiables et de l'autre les informations associées qui permettent d'évaluer et de prendre la décision en question.

3.3.1.1. Choix d'un problème

Choisir un problème dans la base de connaissances consiste à rechercher dans cette base la classe qui le modélise. Pour effectuer cette recherche, l'utilisateur a besoin d'un support adéquat qui lui permette d'identifier cette classe à partir de ses caractéristiques. Selon la précision avec laquelle l'utilisateur peut décrire son problème, il doit pouvoir le spécifier sur un niveau plus ou moins abstrait. Si pour l'utilisateur le problème qu'il veut résoudre n'est pas suffisamment précis dès le départ, il doit pouvoir spécifier ses caractéristiques (données à traiter et des résultats à obtenir) au fur et à mesure.

Dans la base de connaissances, les tâches sont structurées dans différentes hiérarchies qui correspondent aux différents problèmes décrits. Une hiérarchie de classes modélise le problème correspondant sur des niveaux successifs d'abstraction. La fonctionnalité de chaque tâche à l'intérieur de cette hiérarchie est dans une certaine mesure désignée par le nom de la classe correspondante. La visualisation et l'accès direct aux hiérarchies de classes définies dans la base de connaissances constituent donc un premier support pour la sélection du problème à résoudre.

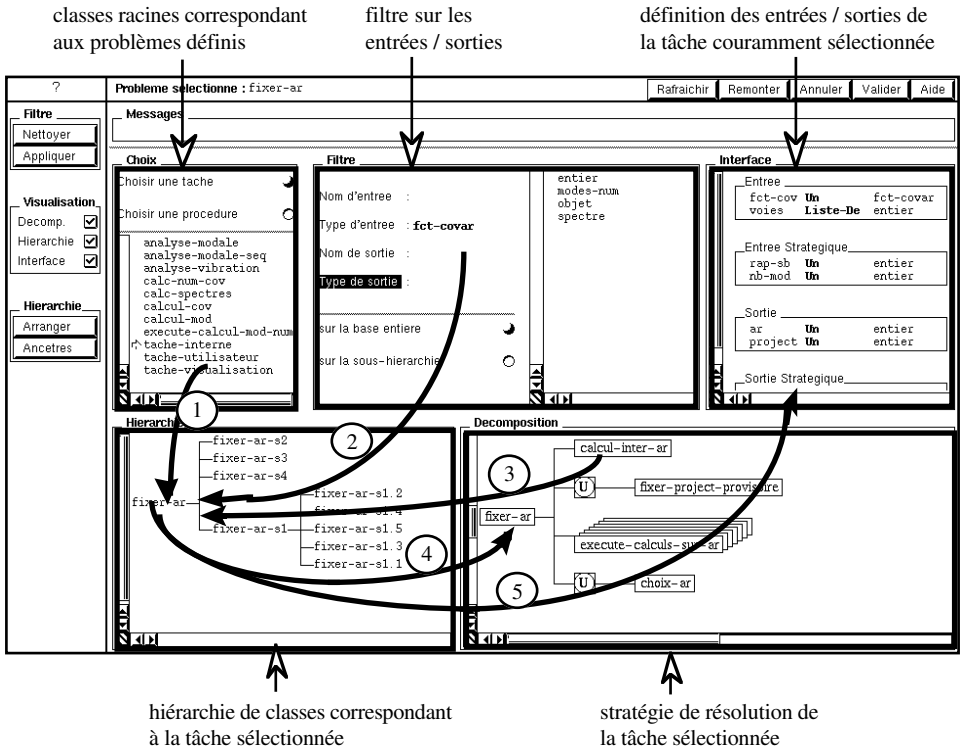


Figure 15 : Conception et fonctionnement de la fenêtre de sélection de problèmes. Un problème peut être identifié pas à pas, en commençant soit par la sélection de la classe racine correspondante (1), soit en appliquant un filtre initial sur toute la base (2). Une fois une première tâche sélectionnée, le problème peut être de plus en plus spécifié, soit par l'application récursive de filtres (2), soit en sélectionnant des sous-tâches dans la représentation de la stratégie de résolution de la tâche actuellement sélectionnée (3). Chaque fois que le choix est modifié, l'affichage de la stratégie de résolution (4) et des entrées / sorties (5) est actualisé. Les boutons à gauche de la fenêtre permettent de contrôler l'application du filtre et la visualisation des différentes caractéristiques de la tâche sélectionnée, les boutons en haut permettent de valider ou d'annuler l'interaction et de remonter pas à pas sur les actions de sélection effectuées.

A part son nom de classe, la définition d'une tâche consiste essentiellement dans la description de ses entrées, de ses sorties et de sa stratégie de résolution. Ces trois caractéristiques permettent à l'utilisateur de vérifier si sa sélection correspond effectivement à ce qu'il recherche. Leur visualisation dans la fenêtre d'interaction correspondante est dès lors indispensable.

Pour pouvoir spécifier pas à pas les caractéristiques du problème, nous avons introduit le moyen de recherche suivant : le filtrage des classes en fonction premièrement des types d'entités que les tâches prennent en entrée et produisent en sortie et deuxièmement des rôles précis que jouent ces entités dans la tâche, c'est-à-dire de leurs noms d'attribut. Pour permettre une spécification incrémentale, ce filtrage peut porter soit sur la base de connaissances entière, soit sur un sous-ensemble de cette base, obtenu par un filtrage antérieur.

La fenêtre d'interaction pour la sélection d'un problème est par conséquent structurée en plusieurs parties : la sélection d'un problème global, c'est-à-dire d'une classe racine d'une hiérarchie, la partie visualisation de la hiérarchie ou sous-hiérarchie couramment sélectionnée, les parties de visualisation de l'ensemble des entrées / sorties et de la stratégie de résolution de la tâche couramment sélectionnée et enfin le filtre permettant d'effectuer une recherche pas à pas (figure 15).

3.3.1.2. Spécialisation

La spécialisation consiste à choisir, dans la hiérarchie de classes correspondant au problème à résoudre, une classe modélisant une tâche exécutable qui soit adaptée au traitement des entités fournies en entrée. Comme toutes les autres décisions importantes, la spécialisation peut être contrôlée par l'utilisateur. Même si celui-ci en a le contrôle, il peut demander au système de lui proposer un choix.

La spécialisation est effectuée par l'intermédiaire de l'application du mécanisme de classement de Shirka sur la hiérarchie de classes modélisant le problème à résoudre : les classes de cette hiérarchie sont marquées sûres, possibles ou impossibles en fonction des contraintes contenues dans leurs définitions et des entrées fournies. Pour compléter des informations manquantes et pour vérifier qu'une classe seulement possible est réellement adaptée aux données, l'utilisateur doit pouvoir fournir les entrées non encore valuées.

Par ailleurs, ainsi que pour le moteur, l'existence de versions pour les spécialisations possibles peut constituer un critère de choix important pour l'utilisateur. Il faut donc lui donner accès à ces versions. La figure 16 montre la conception de la fenêtre d'interaction pour la spécialisation et illustre son fonctionnement.

Pour permettre à l'utilisateur d'effectuer lui-même la spécialisation, il a accès premièrement aux entités fournies en entrées, deuxièmement à la hiérarchie de classes, les classes étant marquées sûres, possibles ou impossibles, et troisièmement aux versions existantes. Dans la hiérarchie des classes, il peut effectuer son choix, dans la partie correspondant aux entrées il peut accéder aux valeurs fournies et donner des valeurs manquantes ; ceci peut permettre en même temps d'actualiser automatiquement le marquage des classes dans la hiérarchie.

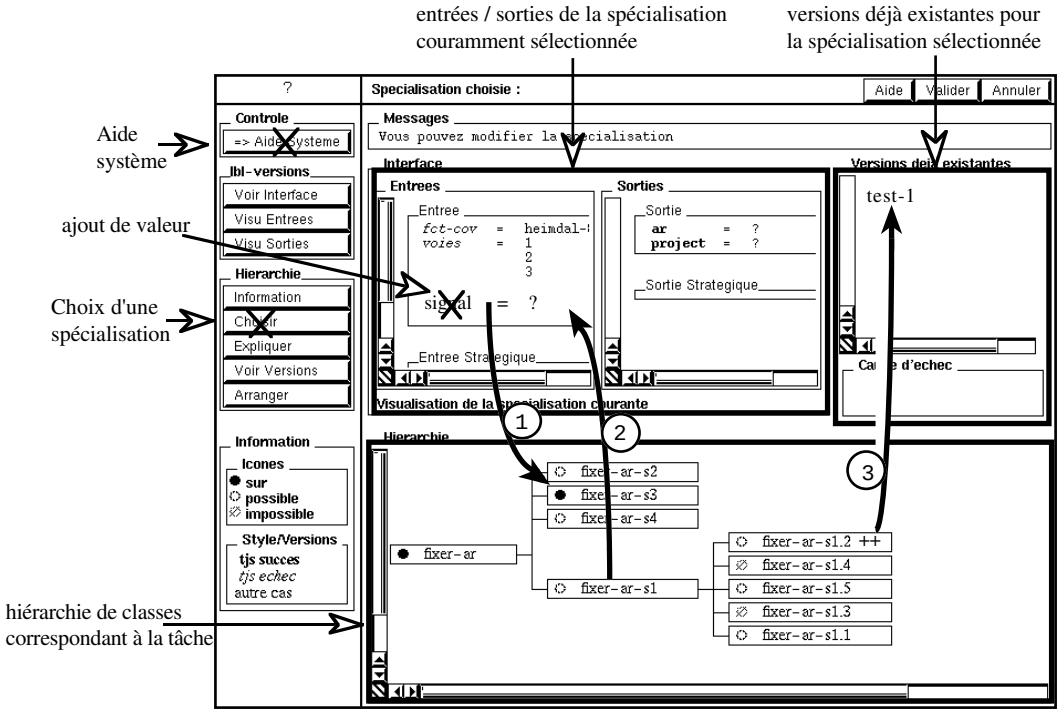


Figure 16 : Conception et fonctionnement de la fenêtre de spécialisation. Cette fenêtre donne premièrement accès à la hiérarchie de classes correspondant au problème à résoudre, les classes étant étiquetées sûres, possibles et impossibles [Cruy93]. Elle permet à l'utilisateur de choisir directement une spécialisation. La fenêtre donne deuxièmement accès à l'ensemble des entrées / sorties de la spécialisation couramment sélectionnée (2) et permet de compléter les valeurs de ces entrées, ce qui conduit à l'actualisation de l'étiquetage des classes (1). Troisièmement, l'utilisateur a accès à tous les essais de spécialisation déjà effectués, aux versions (3). Par ailleurs, il peut demander de l'aide au système qui va dans ce cas proposer une spécialisation selon ses propres critères.

3.3.2. Gestion de la résolution par l'utilisateur

Pour chaque type d'interaction ponctuelle, nous avons défini une fenêtre appropriée, qui permet de prendre ou de modifier la décision correspondante et qui donne accès à toutes les informations associées nécessaires. Mais pour permettre à l'utilisateur de gérer et de diriger lui-même la résolution de problèmes, il faut en parallèle concevoir une fenêtre d'interaction de base qui matérialise le processus de résolution de problèmes. Cette fenêtre doit permettre à l'utilisateur d'initier facilement toutes les interactions nécessaires pour évaluer et pour modifier ce processus.

Les besoins qu'a l'utilisateur pour pouvoir gérer le processus de résolution globale sont plus précisément :

- de pouvoir examiner et évaluer en permanence l'état d'avancement du processus de résolution ;
- d'avoir la possibilité d'examiner et de modifier toutes les décisions prises pendant le processus de résolution, c'est-à-dire de créer des versions de celui-ci ;
- d'accéder et de naviguer librement entre toutes les versions déjà créées ;
- d'explorer de façon interactive de nouvelles tâches, pour ne pas être limité à la résolution des tâches prédéfinies dans la base de connaissances du système.

Pour pouvoir suivre et évaluer le processus de résolution d'un problème, l'utilisateur a besoin à tout moment d'un aperçu synthétique de son état d'avancement. Deux informations sont ici particulièrement intéressantes : premièrement les résultats déjà obtenus en fonction des entrées fournies et deuxièmement l'avancement dans l'exécution de la stratégie de résolution, à savoir, quelles sont les sous-tâches déjà exécutées, les sous-tâches en cours d'exécution, et les sous-tâches qui restent encore à résoudre.

Pour pouvoir remettre en cause les décisions prises pendant le processus de résolution, tous les points de décisions doivent être visibles et accessibles à l'utilisateur. La navigation entre versions requiert donc l'accès à la hiérarchie des versions existantes.

L'exploration d'une nouvelle tâche consiste, comme nous l'avons vu, avant tout dans la (re-)définition du problème à résoudre et de la stratégie de résolution à appliquer. Elle concerne donc la définition des entrées / sorties et de la décomposition d'une tâche en sous-tâches ; ces caractéristiques doivent donc aussi être accessibles et modifiables. Les besoins pour l'exploration d'une nouvelle tâche correspondent en fait aux besoins énoncés pour permettre à l'utilisateur de suivre globalement le processus de résolution : avoir accès aux entrées, aux sorties et à la stratégie de résolution.

En somme, nous avons ainsi identifié trois composants élémentaires pour la fenêtre d'interaction de base :

- l'ensemble des entrées et des sorties du problème à résoudre, ce qui permet d'accéder soit à leurs valeurs soit à leurs définitions respectivement ;
- la représentation de la stratégie de résolution, ce qui matérialise soit le processus de résolution associé et toutes les décisions prises soit la définition de la stratégie en tant que telle ;
- la hiérarchie de versions associées à une tâche.

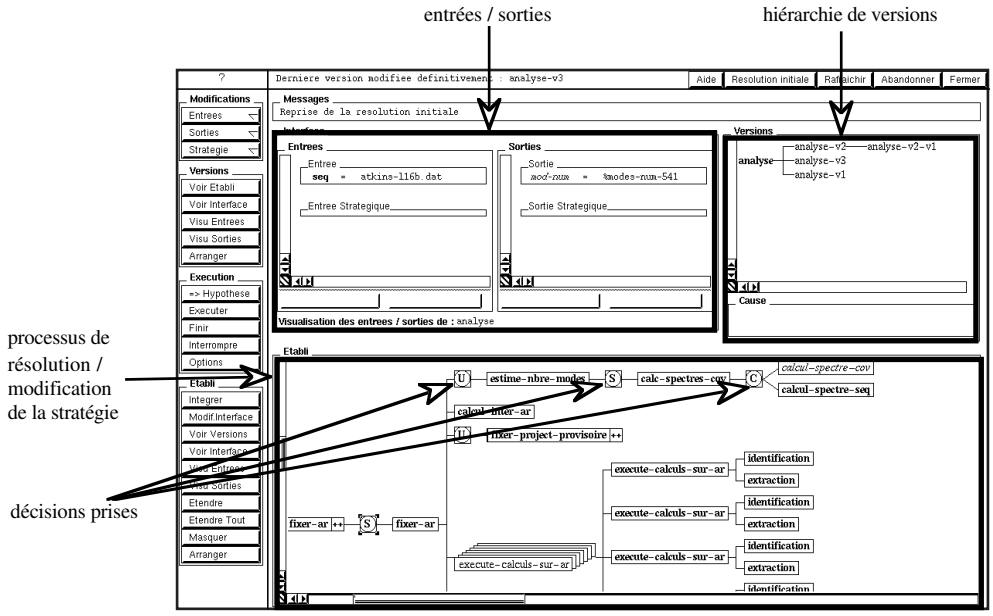


Figure 17 : Conception de la fenêtre de base. Pendant la résolution d'une tâche, cette fenêtre donne accès aux *entrées / sorties* de la tâche globale à résoudre et visualise graphiquement l'état d'avancement du *processus de résolution* ainsi que toutes les *décisions prises* pendant la résolution. Pendant l'exploration d'une tâche, ces deux parties donnent accès à la définition courante de la tâche explorée, de ses *entrées / sorties* et de sa *stratégie de résolution*. La partie montrant la *hiérarchie de versions* permet d'accéder aux versions associées à une tâche et ainsi de naviguer entre elles.

La figure 17 montre la fenêtre de base. Nous allons maintenant décrire plus en détail son fonctionnement, en particulier comment sont satisfaits les différents besoins identifiés au début de cette section.

3.3.2.1. Evaluation de l'état actuel de résolution

La fenêtre d'interaction de base permet à l'utilisateur de suivre et d'évaluer l'état actuel d'une résolution en cours. Elle donne premièrement accès aux valeurs fournies en entrée et dynamiquement aux valeurs obtenues en sortie, dès qu'elles sont déterminées au fur et à mesure de la résolution. Deuxièmement, elle visualise aussi dynamiquement l'avancement du moteur de tâches dans l'exécution de la stratégie de résolution.

Comme nous avons vu au début de ce chapitre, la résolution de problèmes s'effectue dans SCARP par planification hiérarchique, en alternant des phases de spécialisation et de décomposition. Le processus de résolution d'une tâche est ainsi directement matérialisé par l'arbre des instances créées pendant ces deux phases. La visualisation de cet arbre donne à l'utilisateur une impression immédiate de l'état d'avancement de la résolution. Ses nœuds sont étiquetés avec le nom de l'étape de résolution qui désigne le rôle de la sous-tâche correspondante dans la stratégie. Le fait que la résolution d'une tâche soit un succès, un échec ou encore non terminée est aussi directement visible : la tâche est désignée en gras, en italique ou en fonte "standard" respectivement.

3.3.2.2. Examen et remise en cause de décisions

Pour permettre à l'utilisateur d'examiner, et en conséquence de remettre en cause des décisions prises pendant le processus de résolution, ces décisions sont représentées de façon intégrée dans l'arbre des instances créées. Elles sont ainsi immédiatement visibles pour l'utilisateur. Elles concernent comme nous l'avons vu la spécialisation, le choix stratégique, la résolution de tâches-utilisateur et la modification des valeurs des paramètres d'entrées ou de sorties d'une tâche. Tandis que les deux dernières peuvent concerner n'importe quelle tâche dans l'arbre de décomposition et de spécialisation, les trois premières concernent des points de décision bien précis. Elles sont par conséquent explicitement représentées dans l'arbre de spécialisation et de décomposition par des nœuds étiquetés "S" (Spécialisation), "C" (Choix), ou "U" (tâche utilisateur) respectivement.

Tous les éléments de cet arbre, les nœuds représentants des tâches et les différents points de décision, sont directement accessibles et modifiables par l'utilisateur ; ce sont des nœuds actifs dans le sens où leur sélection à l'aide de la souris active la fenêtre d'interaction correspondante qui permet de visualiser, mais aussi de modifier l'état de la décision correspondante : la sélection d'un nœud de type tâche active un éditeur d'instances, celle d'un nœud "S" la fenêtre de contrôle de la spécialisation, celle d'un nœud "C" la fenêtre de contrôle du choix et celle d'un nœud "U" la fenêtre de contrôle d'une tâche utilisateur. Le protocole d'interaction pour une remise en cause est ainsi simple :

- Sélection du nœud correspondant à la décision à modifier.
- Indication de la modification.
- Indication si c'est une modification locale, c'est-à-dire si seulement une version locale doit être créée.
- Validation de la modification.

La modification effectuée est ensuite prise en compte par le gestionnaire de dialogues qui gère la création d'une nouvelle version et son exécution via le moteur de tâches.

3.3.2.3. Navigation entre versions

Des versions locales peuvent exister pour chaque tâche de l'arbre de spécialisation et de décomposition, et cela sur plusieurs niveaux imbriqués; l'arbre de spécialisation et de décomposition attaché à une version locale peut contenir d'autres versions locales et ainsi de suite. La possibilité de naviguer entre toutes ces versions constitue pour l'utilisateur d'une part une source d'information, à savoir ce qui a déjà été essayé pendant le processus de résolution, et d'autre part une base pour reprendre la résolution d'autres versions intéressantes, interrompues ou abandonnées antérieurement.

La navigation permet à l'utilisateur de changer progressivement de niveau d'imbrication. Pour cela il a besoin d'accéder d'une part à partir d'une tâche de l'arbre de spécialisation et de décomposition à la hiérarchie des versions associées, et d'autre part à partir d'une version dans la hiérarchie de versions à l'état de résolution et à l'arbre de spécialisation et de décomposition correspondant (figure 18).

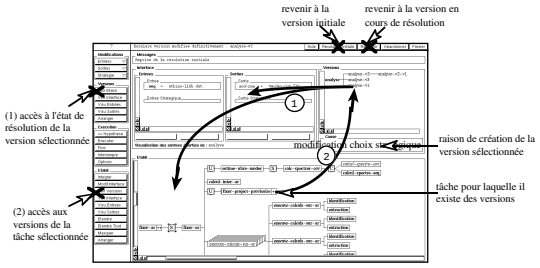


Figure 18 : Navigation entre versions. L'utilisateur peut, à partir d'un nœud représentant une tâche dans l'arbre de spécialisation et de décomposition, accéder à la hiérarchie des versions associées (2). Chaque tâche dans l'établi de résolution pour laquelle il existe des versions est pour cela marquée avec une extension du nom du nœud correspondant (++). A l'inverse, à partir de chaque version dans la hiérarchie des versions, l'utilisateur peut accéder à son état de résolution, et à l'arbre de spécialisation et de décomposition correspondant. Indépendamment, il peut à tout moment revenir directement à la version initiale ou à la dernière version créée du processus de résolution.

Pendant la navigation, l'utilisateur peut en fait alterner tant qu'il le veut ces accès. Ainsi il obtient des informations complémentaires associées et peut reprendre une version antérieure. Sinon, et pour terminer le processus de navigation, il peut revenir directement soit à la version de résolution initiale, soit à la dernière version créée pendant le processus de résolution. Tant que l'utilisateur ne décide pas de reprendre une version antérieure, la navigation n'intervient pas dans le processus de résolution courant. Celui-ci est seulement interrompu si l'utilisateur effectue une modification.

3.3.2.4. Exploration de nouvelles tâches

Ainsi que pour toutes les autres interactions de base, l'exploration de nouvelles tâches doit s'effectuer via la fenêtre d'interaction de base. Elle consiste en la (re-)définition de ses entrées, de ses sorties et de sa stratégie de résolution. Les différents éléments de cette définition sont accessibles dans la fenêtre de base via les parties qui visualisent les entrées / sorties de la tâche et via la partie matérialisant le processus de résolution.

Comme nous l'avons indiqué à la section 3.2.3.1., l'exploration d'une nouvelle tâche se fait pas à pas. Chaque pas consiste soit en un ajout, soit en une modification du type soit en une suppression d'un attribut d'entrée, de sortie ou de sous-tâche. Tandis que la modification des entrées / sorties se fait immédiatement par rapport à un attribut sélectionné, la modification de la stratégie de résolution est plus complexe. L'ajout d'une sous-tâche est réalisé par le protocole d'interaction suivant :

- Indication de l'endroit où la nouvelle sous-tâche doit s'insérer dans la stratégie actuellement définie, en tant que sous-tâche terminale, après ou avant la tâche ou le bloc stratégique sélectionné(e)[1].
- Via le sélecteur de problèmes la tâche à insérer en tant que sous-tâche est choisie. La définition de celle-ci est ensuite récursivement accessible et elle peut ainsi également être modifiée.
- Le rôle de la sous-tâche dans la stratégie de résolution est indiqué.

Souvent, dans une première phase de définition, la stratégie de résolution n'est pas indiquée de façon très structurée ; elle consiste en une simple séquence de sous-tâches. Il faut ainsi pouvoir structurer a posteriori une telle stratégie : plusieurs sous-tâches sélectionnées peuvent pour cela être regroupées en tant que séquence sous une nouvelle tâche ou sous un choix stratégique.

Après la définition statique d'une nouvelle tâche, l'utilisateur lance le moteur de tâches. Pendant l'exécution, l'utilisateur indique les flots de données à l'aide de la souris. Finalement, l'utilisateur valide cette exécution et la classe correspondant à la définition de la nouvelle tâche est mise à jour.

3.4. BILAN

Dans ce chapitre nous avons montré comment le processus coopératif de résolution de problèmes est modélisé et géré dans SCARP. Le cycle de résolution de problèmes, la résolution plus ou moins automatique de problèmes par planification hiérarchique, sont gérées par le *moteur de tâches*. La coopération avec l'utilisateur avec toutes ses conséquences est contrôlée par le *gestionnaire de dialogues*. La coopération est matérialisée par une *interface* adéquate. La nécessité de coopérer avec l'utilisateur est intervenue dans la conception de ces trois couches de SCARP.

Le moteur de tâches prend en compte les différents types d'interaction prédéfinis dans la base de connaissances (validation des entrées ou des sorties par l'utilisateur, etc.). De plus, l'utilisateur a la possibilité de modifier son mode de fonctionnement, d'abandonner ou de prendre des responsabilités supplémentaires pendant la résolution. Il peut ainsi décider de contrôler ou non les différents types de décision à prendre pendant les

---

[1] Pour l'instant, nous nous sommes limités aux types les plus élémentaires de contrôle : l'indication d'une séquence de sous-tâches ou d'un choix entre sous-tâches. La définition interactive d'une itération requière plus d'informations concernant le contrôle et n'est pas encore possible.

différentes phases de résolution (choix, spécialisation, etc.). Chaque fois qu'une interaction avec l'utilisateur s'avère nécessaire pendant la résolution, le moteur de tâches se suspend et communique cette interaction au gestionnaire de dialogues.

Toutes les interactions avec l'utilisateur sont gérées par le gestionnaire de dialogues. Celui-ci permet non seulement au moteur de tâches, mais aussi à l'utilisateur d'initialiser des interactions. L'utilisateur peut ainsi modifier des décisions prises pendant la résolution. Le gestionnaire crée dans ce cas les différentes versions du processus de résolution. L'utilisateur peut aussi explorer de nouvelles tâches. Le gestionnaire crée dans ce cas les classes correspondantes dans la base de connaissances.

L'interface permet de matérialiser et de réaliser toutes les interactions système-utilisateur, les interactions ponctuelles, qui concernent un type de décision spécifique, et les interactions de base qui permettent à l'utilisateur de contrôler globalement la résolution. L'interface visualise et donne pour cela accès à toutes les informations liées au processus de résolution et à toutes les fonctionnalités du gestionnaire de dialogues. Via des protocoles d'interactions, elle permet à l'utilisateur d'intervenir convenablement dans le processus de résolution.

Globalement, la résolution de problèmes dans SCARP se situe entre deux modes de contrôle extrêmes : la résolution peut être gérée de la manière la plus automatique possible par le moteur de tâches, ne faisant intervenir l'utilisateur que là où c'est absolument nécessaire. Inversement la résolution peut être complètement contrôlée et explorée par l'utilisateur qui définit interactivement le problème qu'il souhaite résoudre et la stratégie de résolution adéquate, en ne faisant intervenir le moteur de tâches que pour la gestion de l'exécution d'après cette stratégie. En outre, soulignons que l'utilisateur reste toujours libre pour intervenir *a posteriori* dans la résolution pour modifier toutes les décisions prises et pour ainsi explorer une autre branche de raisonnement.

4. DEVELOPPEMENT ET UTILISATION D'APPLICATIONS

Dans les chapitres précédents, nous avons défini les concepts essentiels de SCARP, concernant la formalisation et l'exploitation des connaissances. Créer une application avec SCARP revient à remplir la base de connaissances, c'est-à-dire à analyser le domaine d'application et à décrire ses tâches, ses méthodes et ses entités dans le formalisme présenté au chapitre deux. En exploitant ces connaissances, SCARP est ensuite capable de gérer le processus de résolution de problèmes en coopération avec l'utilisateur, de la manière présentée au chapitre trois. Dans ce chapitre nous allons montrer d'une part, et de façon très générale, comment développer une application avec SCARP et d'autre part comment exploiter une application pour résoudre des problèmes de manière souple et coopérative.

Dans la première partie de ce chapitre, nous discutons de l'analyse d'un domaine d'application pour SCARP. Nous proposons de suivre une analyse descendante, guidée par l'identification des tâches du domaine d'application. Nous décrivons les étapes générales d'une telle analyse et la façon dont ces étapes conduisent à la formalisation des connaissances. Cette démarche est ensuite illustrée à l'aide d'un exemple, de l'analyse effectuée pour développer l'application de l'IFREMER à Brest. Cette application est utilisée pour étudier l'état de plates-formes offshore. Elle est particulièrement intéressante parce que, tout en étant limitée en taille et en complexité, elle permet de montrer comment utiliser tous les éléments du modèle de représentation de connaissances SCARP. Elle modélise en particulier de nombreuses interactions système-utilisateur qui sont nécessaires pendant la résolution.

Dans la deuxième partie de ce chapitre, nous présentons comment une application SCARP peut être exploitée pour résoudre des problèmes de façon coopérative. Cette coopération ne se limite pas à la gestion des interactions pré-définies dans la base de connaissances, mais peut s'établir a posteriori sur tous les niveaux d'abstraction et de décomposition introduits par la modélisation des connaissances basée sur la planification hiérarchique. Nous rappelons d'abord où et comment une telle coopération peut s'établir en général. Le processus de résolution coopérative est ensuite illustré à l'aide d'un exemple. Nous avons pour cela choisi l'application SLOT, une application en analyse de données exploratoire, qui a été développée par F. Chevenet au laboratoire de Biométrie, Génétique et Biologie des Populations de l'université Claude Bernard, Lyon 1. Cette application traite d'un domaine complexe où les stratégies de résolution ne peuvent pas être décrites de manière exhaustive. Ainsi, elle permet de montrer premièrement comment l'utilisateur peut modifier le mode de contrôle de la résolution en fonction de ses propres connaissances, et deuxièmement comment il peut explorer de nouvelles stratégies de résolution de façon interactive.

4.1. ANALYSER UN DOMAINE POUR DEVELOPPER UNE APPLICATION SCARP

L'analyse d'un domaine pour développer une application SCARP doit permettre de définir les trois types d'éléments d'une base de connaissances SCARP : les tâches, les entités et les méthodes. L'essentiel des connaissances de résolution de problèmes consiste en la modélisation des tâches. Celle-ci détermine et limite en même temps les possibilités de coopération système-utilisateur pendant la résolution de problèmes.

L'analyse d'un domaine d'application peut s'effectuer selon deux manière générales, de manière descendante ou ascendante. Dans le premier cas, l'analyse part des tâches principales à résoudre et identifie récursivement les tâches de plus en plus élémentaires, les méthodes et les entités. Dans le deuxième cas, elle identifie d'abord les méthodes et les entités existantes et compose ensuite des tâches de plus en plus complexes. Nous proposons ici de suivre une démarche d'analyse descendante. Celle-ci est plus structurée et peut, à la fin, être complétée par une analyse ascendante qui compose des tâches complémentaires à partir des tâches, des méthodes et des entités déjà identifiés. Avec SCARP, cette deuxième partie, la construction de nouvelles tâches par combinaison d'éléments déjà définis dans la base de connaissances, peut se faire de manière interactive. La section 4.2.2.3. en donne un exemple.

Différentes étapes peuvent être distinguées dans l'analyse descendante d'un domaine d'application. Nous allons les décrire de façon générale pour pouvoir ensuite les illustrer à l'aide de l'analyse effectuée pour le développement de l'application de l'IFREMER avec SCARP.

4.1.1. Analyse descendante d'un domaine d'application

Le point de départ et le fil central de l'analyse d'un domaine d'application consiste dans l'identification des tâches. Etant donné que la définition des tâches, c'est-à-dire des problèmes existants dans un domaine, est constituée d'abord par l'identification de l'ensemble des entités manipulées, l'identification des tâches et des entités d'un domaine se fait de manière plus ou moins parallèle. L'identification et la spécification des entités manipulées se fait en fonction des besoins pour la définition des tâches. Les méthodes constituent les feuilles de la décomposition des tâches en sous-tâches. Leur identification est ainsi guidée par la description des stratégies de résolution associées aux tâches.

4.1.1.1. Etapes de l'analyse

L'identification des tâches d'un domaine se fait en plusieurs étapes. Dans une étape initiale, les tâches de plus haut niveau, c'est-à-dire les types de problèmes principaux à résoudre, sont définies à un niveau abstrait. Ensuite, chacun de ces problèmes principaux est analysé de manière de plus en plus précise. Ceci conduit, au fur et à mesure de l'avancement de cette analyse, à l'identification récursive des tâches (de plus en plus élémentaires), des entités et les méthodes du domaine (figure 1). L'analyse d'un domaine passe par l'analyse successive de toutes ses tâches d'abord des tâches principales et ensuite des tâches plus élémentaires.

L'analyse des tâches conduit récursivement à l'identification des entités et des méthodes d'un domaine : les entités représentent les connaissances manipulées par les tâches et les méthodes ; les méthodes (externes ou internes) décrivent des moyens de résolution directs. Pour une méthode, la classe correspondante lie le mécanisme de réalisation de la méthode aux entités qu'elle traite. Pour une méthode externe, elle définit, outre le programme à exécuter, comment il faut préparer son exécution et comment ses résultats doivent être interprétés et intégrés dans la base de connaissances.

A la fin de l'analyse, les connaissances contenues dans la base qui concernent les méthodes et les entités du domaine doivent être complétées. Car, même si toutes les méthodes et toutes les entités référencées par des tâches de la base sont déjà identifiées, d'autres peuvent exister qui pourraient être utilisées lors de l'exploration interactive de tâches nouvelles (cf. section 4.2.2.3.). L'ensemble des méthodes est complété en récapitulant les programmes et les moyens d'inférences existants dans le domaine d'application. Les méthodes non encore décrites sont ainsi identifiées et définies. De la même façon, l'ensemble des entités du domaine est complété.

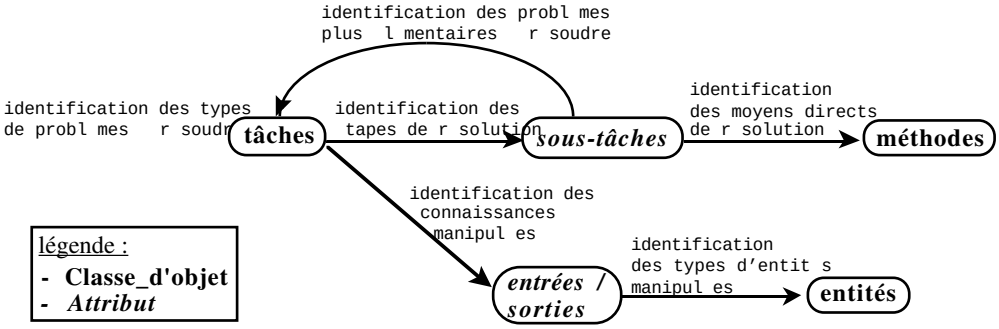


Figure 1 : L'analyse d'un domaine d'application est guidée par l'identification des problèmes à résoudre. Au début de l'analyse, les différents types de problèmes existants dans le domaine d'application sont identifiés. Les classes correspondantes dans la base de connaissances sont nommées. Chacune des tâches ainsi déterminées est ensuite à son tour analysée : les connaissances qu'elle manipule et les étapes nécessaires à sa résolution sont identifiées. La nomination des attributs correspondants indique leur signification, le rôle qu'ils jouent pour la tâche concernée. L'analyse de leur contenu conduit à la définition des tâches plus élémentaires, des méthodes et des entités du domaine d'application.

Pendant l'analyse, chaque fois qu'une nouvelle classe est introduite dans la base de connaissances, elle doit être située par rapport aux (hiérarchies de) classes déjà existantes. Dans des phases de *généralisation*, si possible, différentes définitions ayant des caractéristiques communes sont regroupées sous une définition plus générale. Ceci correspond à la définition de sur-classes communes. Dans des phases de *précision,* différents cas plus spécifiques peuvent être décrits par des sous-classes. Des phases de *précision* et de *généralisation* structurent donc la base de connaissances (figure 2).

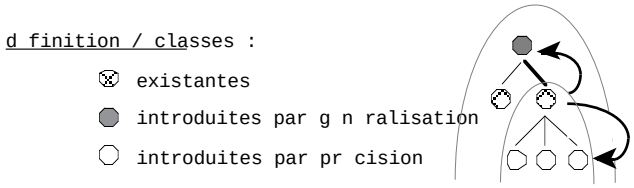


Figure 2 : Structuration des connaissances.

4.1.1.2. Analyse d'une tâche

L'étape essentielle de l'analyse d'un domaine d'application consiste dans l'analyse des tâches. Chaque tâche identifiée est séparément analysée. L'analyse d'une tâche est structurée en plusieurs phases, chacune caractérisant un aspect spécifique de la tâche en question qui est ensuite matérialisé dans sa formalisation pour SCARP (figure 3) :

- dans la phase d'*identification,* la tâche est nommée. Ce nom deviendra le nom de la classe qui la représentera dans la base de connaissances.
- pendant la phase d'*identification des connaissances manipulées,* les entrées et les sorties de la tâche sont identifiées. Premièrement, le rôle que joue chacune des entités manipulées pour la tâche est nommé. Cette nomination correspondra dans la définition de la tâche au nom de l'attribut d'entrée ou de sortie associé. Deuxièmement, le type de chaque entité est identifié et nommé, et une classe correspondante est introduite dans la base de connaissances.

- durant la phase d'*identification de la stratégie de résolution,* la stratégie de décomposition associée est déterminée. Premièrement, les différentes étapes de résolution sont nommées. Dans la définition de la tâche, la nomination de chacune de ces étapes correspondra au nom d'un attribut de type sous-tâche. Deuxièmement, la sous-tâche à résoudre dans chacune de ces étapes est identifiée. Elle doit à son tour être séparément analysée. Troisièmement, l'intégration de l'exécution de chaque sous-tâche dans l'exécution de la tâche globale elle-même est décrite par un contexte d'exécution.
- dans la phase d'*identification des entrées et des sorties stratégiques*, à partir de la définition de sa stratégie, les paramètres supplémentaires d'entrée et les sorties brutes de la tâche sont identifiés. Comme pour les entrées et les sorties stratégiques, leur rôle et leur type sont définis.
- pendant la phase d'*identification des pré- et des post-tâches,* des tâches ou des moyens d'inférence pour déterminer les entrées stratégiques (sorties globales) à partir des entrées globales (sorties stratégiques) sont introduits. S'il s'agit d'une tâche, celle-ci est identifiée. Elle doit être séparément analysée et son exécution intégrée dans la tâche englobante via un contexte d'exécution.
- dans la phase d'*identification des tâches de visualisation,* des moyens adéquats pour visualiser l'ensemble des entrées ou des sorties de la tâche sont décrits. La tâche de visualisation correspondante doit à son tour être analysée.

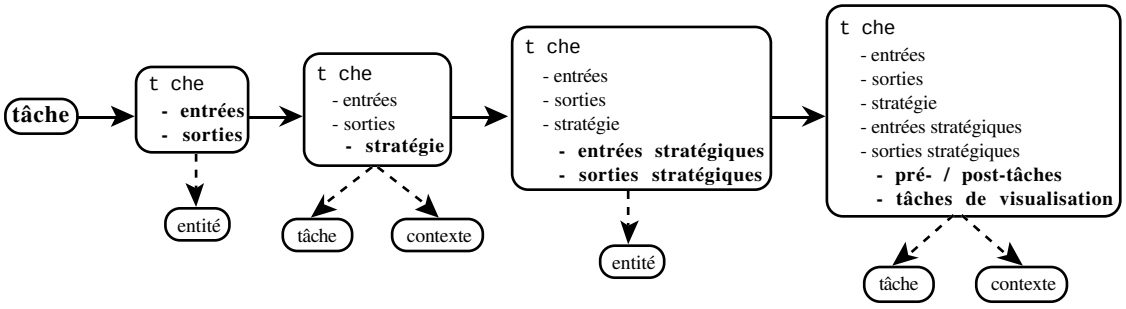


Figure 3 : Analyse d'une tâche. L'analyse est structurée en plusieurs phases qui servent à identifier les différents aspects d'une tâche. Chaque phase complète les connaissances rattachées à la tâche et identifie d'autres éléments de connaissance qui doivent être analysés ultérieurement.

La phase d'*identification de la stratégie de résolution* est la phase la plus complexe de cette analyse. Durant cette phase non seulement les étapes de résolution nécessaires et le flot de contrôle sur ces étapes sont définis, mais aussi le flot de données, le type de contrôle à effectuer sur chaque sous-tâche et l'ensemble des interventions de l'utilisateur nécessaires pendant le processus de résolution. Cette étape permet, tout d'abord, d'introduire des choix stratégiques contrôlés par l'utilisateur. De plus, la définition du contexte d'exécution permet d'introduire d'autres interventions de la part de l'utilisateur : doit-il valider ses entrées ou les résultats obtenus ou est-ce que leur visualisation peut être intéressante pendant la résolution ? Ou encore : est-ce l'utilisateur qui résout le sous-problème en question ? Si oui, est-ce qu'il y a un moyen pour l'aider dans sa tâche ?

En parallèle avec les phases d'analyse d'une tâche, les connaissances identifiées sont structurées au fur et à mesure. Si différents cas spécifiques existent, ceux-ci sont différenciés dans une phase de *précision* par la définition de sous-classes distinctes dans la base de connaissances. Concernant les tâches, il peut y avoir par exemple différentes stratégies, adaptées au traitement d'entités avec des caractéristiques différentes. De même, il peut être intéressant simplement de décrire un problème à plusieurs niveaux d'abstraction. Si des tâches identifiées sont voisines, si elles traitent d'un même problème

général, elles peuvent, dans une phase de *généralisation,* être regroupées sous une sur-classe commune.

Nous avons ici présenté une démarche très générale pour développer une application SCARP, c'est-à-dire la base de connaissances correspondante. Nous allons illustrer cette démarche à l'aide d'un exemple.

4.1.2. Exemple d'une analyse : l'application IFREMER

L'application choisie comme exemple concerne le domaine du traitement du signal : il s'agit de l'étude du comportement dynamique de plate-formes *offshore* par analyse modale. Cette application a été réalisée en coopération avec M. Prévosto et S. Coudray de l'IFREMER de Brest. Nous allons introduire rapidement le domaine d'application, pour pouvoir préciser ensuite la manière dont les connaissances ont été formalisées avec SCARP.

4.1.2.1. Description du domaine

L'objectif de l'application IFREMER est l'étude du comportement dynamique de plate-formes offshore [Prév&91], [Jean&91]. Cette étude est effectuée en utilisant des méthodes spécifiques d'analyse modale. L'analyse modale est un outil général pour l'étude du comportement dynamique de structures. Ses résultats permettent d'évaluer l'état de ces structures, une analyse régulière permet de surveiller cet état.

L'analyse modale est effectuée à partir de signaux, captés sur la structure à examiner, qui reflètent son comportement dynamique. Le plus souvent, ces signaux sont les signaux de réponse obtenus suite à une excitation *artificielle* de la structure. Des signaux d'exitation sont par exemple un échelon ou un dirac. Mais dans certains cas, une excitation artificielle n'est pas possible ou trop coûteuse. C'est par exemple le cas des plate-formes offshore qui doivent être examinées à l'IFREMER. Dans ces cas, l'analyse modale est basée sur le signal de réponse à l'excitation *naturelle,* c'est-à-dire aux signaux naturels émis par son environnement. Deux catégories de méthodes sont donc distinguées en analyse modale, les méthodes basées sur une excitation artificielle et les méthodes basées sur une excitation naturelle.

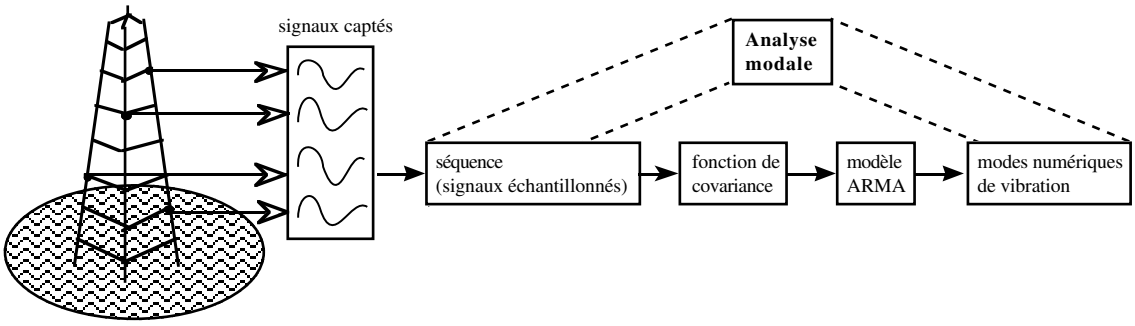


Figure 4 : L'analyse modale d'une plate-forme offshore. Cette analyse s'effectue à partir de fichiers de séquences, de séries temporelles qui représentent les signaux captés et ensuite échantillonnés sur la plate-forme, et donne en résultat des modes numériques de vibration, qui permettent d'évaluer l'état de la plate-forme.

La méthode utilisée à l'IFREMER fait partie de la deuxième catégorie. Elle exploite un ensemble de signaux, recueillis à différents endroits sur une plate-forme offshore. Cette méthode consiste d'abord, à partir de ces signaux, à effectuer le calcul d'une fonction de covariance et ensuite à identifier des paramètres d'un modèle multidimensionnel

spécifique, du modèle ARMA [Prév82]. Finalement, des modes numériques de vibration sont calculés pour la plate-forme (figure 4). Ces modes numériques permettent d'évaluer l'état global de la plate-forme en question.

Avec le modèle ARMA, le comportement dynamique de la structure examinée est supposé pouvoir être décrit par un système stationnaire, linéaire et dynamique d'ordre infini. Nous n'allons pas rentrer plus dans les détails de ce modèle, mais plutôt préciser à un niveau descriptif en quoi consiste l'analyse modale d'une plate-forme offshore qui utilise ce modèle. Ceci va nous permettre d'identifier immédiatement, de la façon présentée dans la section précédente, les connaissances à modéliser dans la base de connaissances SCARP.

4.1.2.2. Analyse du domaine et représentation des connaissances

Le point de départ de l'analyse d'un domaine d'application est l'identification de ses tâches principales. La définition des données qu'elles traitent et des résultats qu'elles produisent conduit à l'identification des entités, la définition des étapes de résolution à l'identification récursivement des tâches plus élémentaires et des méthodes. La figure 5 donne un aperçu de la décomposition récursive qui est établie avec l'analyse des tâches et qui conduit à l'identification de tâches de plus en plus élémentaires dans l'application IFREMER. Elle permet de suivre notre analyse. Nous allons maintenant décrire cette analyse et insister sur les points qui montrent l'intérêt d'utiliser SCARP et la façon dont sont intégrées les interactions nécessaires entre le système et l'utilisateur.

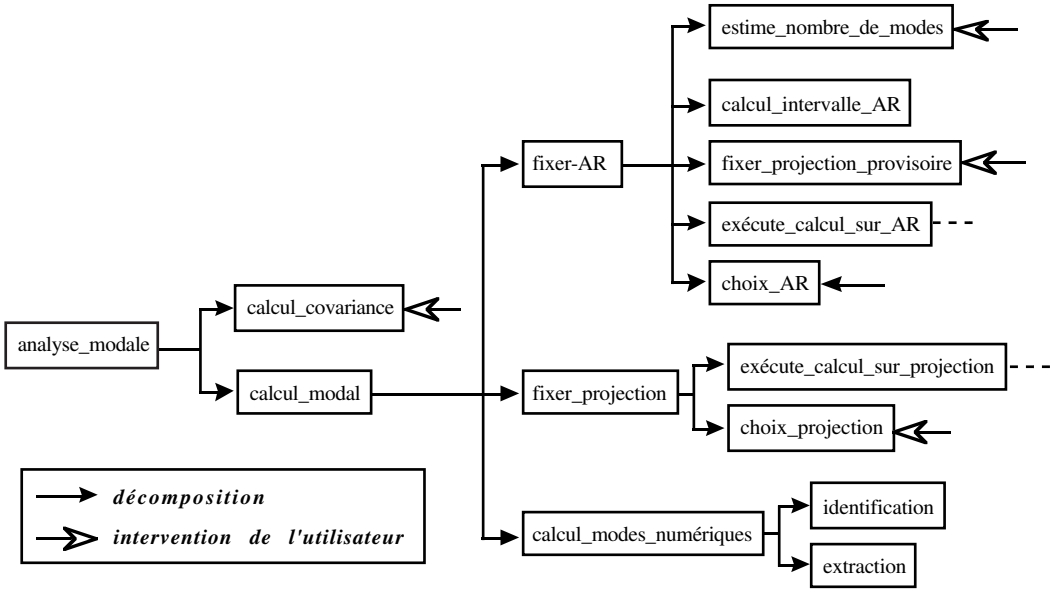


Figure 5 : Analyse récursive de tâches. Cette analyse passe par l'analyse récursive des étapes de résolution d'une tâche principale du domaine d'application, de l'*analyse_modale*. Elle identifie des tâches de plus en plus élémentaires et les interventions de la part de l'utilisateur nécessaires pendant la résolution**.** Cette figure illustre le procédé d'analyse pour l'application IFREMER. La tâche principale est l'analyse modale d'une plate-forme offshore (*analyse_modale*). Elle est résolue en deux étapes : le calcul d'une fonction de covariance (*calcul_covariance*) et le calcul des modes numériques (*calcul-modal*). Ces sous-tâches se décomposent récursivement en sous-tâches de plus en plus élémentaires. Les lignes pointillées indiquent une décomposition en sous-tâches qui correspond en fait à la décomposition de *calcul_modes_numériques*.

Comme indiqué auparavant la tâche principale dans l'application IFREMER est *l'analyse modale* d'une plate-forme offshore. Les données à analyser sont l'ensemble des signaux échantillonnés sur la plate-forme. Le résultat de cette analyse sont des modes numériques de vibration. Deux entités correspondantes, *séquence* (figure 6) et *modes_numériques*, sont pour cela introduites dans la base de connaissances.

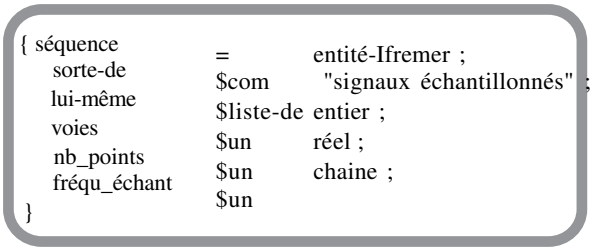


Figure 6 : Entité séquence. Cette entité modélise les données initiales d'une *analyse-modale*. Elle contient les signaux captés avec une fréquence d'échantillonnage fixe (*fréqu_échant*) via un ensemble de *voies* qui correspondent aux différents endroits de prise de mesures sur la plate-forme. Pendant le temps d'échantillonnage, un certain nombre de valeurs (*nb_points*) a été capté. Ces valeurs sont stockées dans un *fichier*.

Les deux étapes de résolution nécessaires sont premièrement le calcul d'une fonction de covariance entre les signaux recueillis aux différents endroits sur la plate-forme et deuxièmement le calcul modal des modes numériques. Le calcul modal regroupe les sous-étapes nécessaires pour déterminer les paramètres du modèle ARMA et les modes numériques.

La première (sous-)tâche à modéliser séparément est donc le *calcul* de la fonction de *covariance*. Elle prend en entrée les signaux échantillonnés, représentés par une séquence, et un nombre de *lags*, c'est-à-dire le nombre de points à prendre en compte pour le calcul, et produit en sortie une fonction de covariance. Différentes méthodes sont disponibles pour effectuer ce calcul. Chacune est appropriée d'une part à une certaine plage de valeurs pour le nombre de lags et d'autre part à des signaux avec des caractéristiques spécifiques. Les différentes stratégies de résolution pour la tâche de calcul de la fonction de covariance sont décrites par un ensemble de classes organisées en hiérarchie (figure 7).

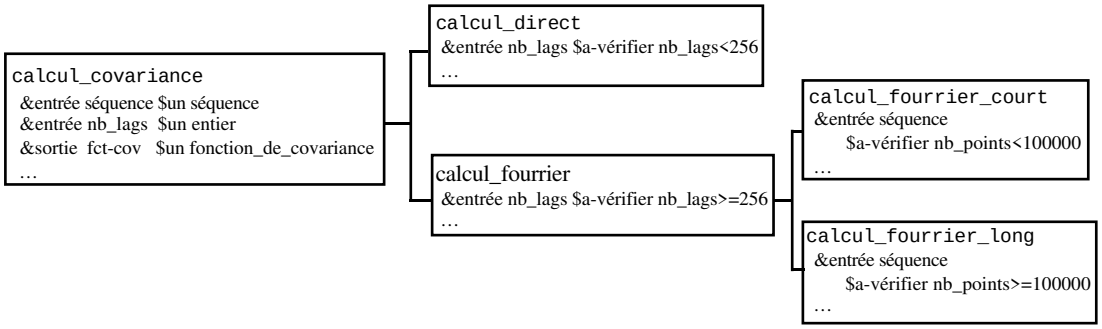


Figure 7 : Hiérarchie de classes attachée à la tâche calcul_covariance. Différentes stratégies de résolution sont adaptées à un nombre de lags plus ou moins élevé et à une séquence plus ou moins longue. Elles sont représentées par des tâches dans une hiérarchie de classes.

L'intégration de cette première sous-tâche dans la stratégie de résolution de la tâche globale (analyse modale) requiert un contrôle spécifique : il s'agit d'une itération contrôlée par l'utilisateur. La qualité de la fonction de covariance déterminée dépend en

fait du nombre de lags pris en compte pour le calcul. Ce nombre doit être optimisé via un processus d'essai-erreur. Il est une première fois choisi "par expérience". La fonction de covariance obtenue avec ce nombre doit être évaluée par l'utilisateur. Pour cela elle est visualisée, ce qui permet à l'utilisateur de vérifier si le nombre de lags qu'il avait choisi était bien adapté ou non. Si la fonction de covariance est d'une qualité suffisante, l'utilisateur l'accepte, sinon il modifie le nombre de lags et effectue un nouveau calcul.

Toutes les informations correspondantes à la résolution de cette première étape de résolution sont représentées d'une part dans le contexte d'exécution correspondant qui est de type *essai-erreur*, et d'autre part dans la définition séparée de la tâche *calcul-covariance* elle-même (figure 8).

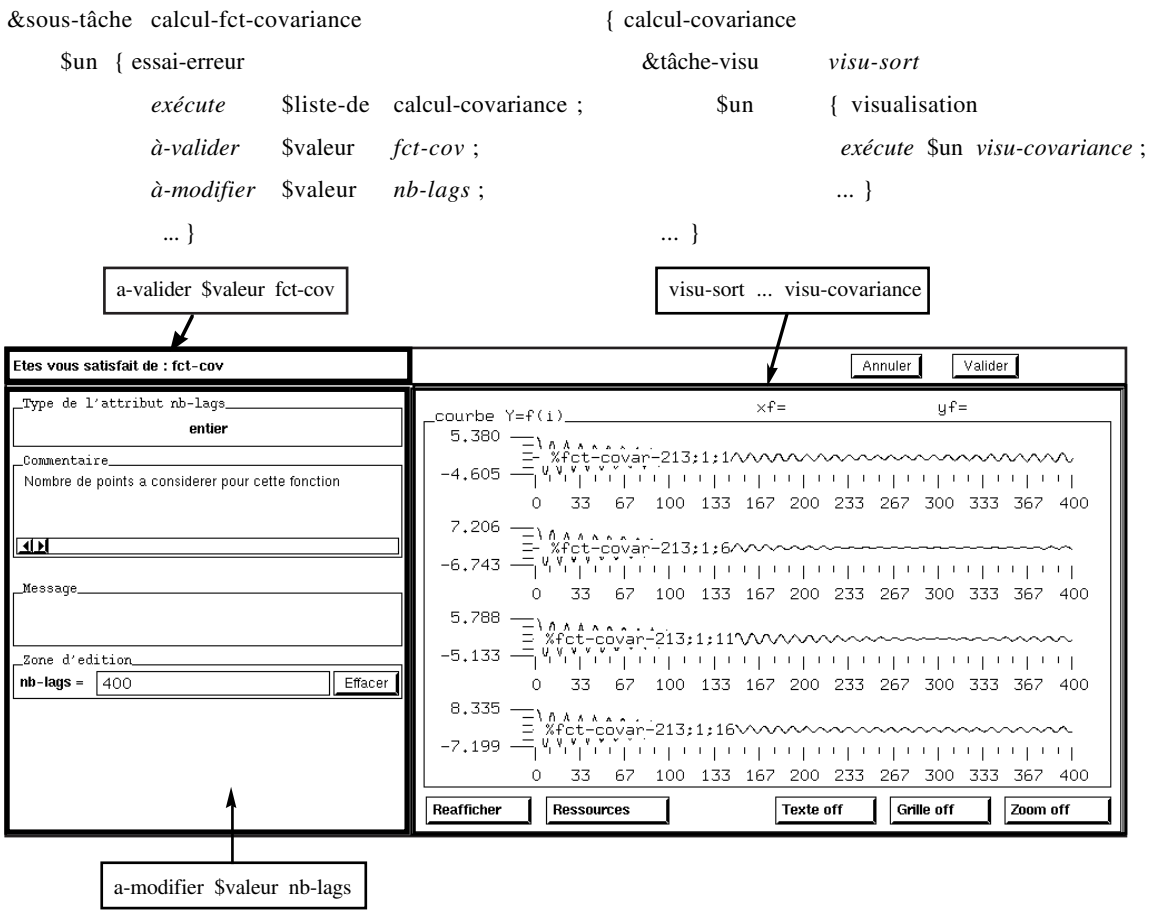


Figure 8 : Formalisation des connaissances associées à la première étape de résolution et fenêtre d'interaction SCARP résultante. Pour résoudre l'étape *calcul-fct-covariance,* la sous-tâche *calcul-covariance* est intégrée via un contexte *essai-erreur* dans la tâche englobante *analyse_modale*. Ce contexte indique que la valeur de l'attribut *fct-cov* est *à valider*, c'est-à-dire à vérifier, après chaque itération par l'utilisateur et que, si celle-ci n'est pas satisfaisante, le nombre de lags (*nb-lags*) peut être *modifié*. Pour permettre à l'utilisateur de vérifier la qualité de la fonction de covariance obtenue, SCARP utilise la tâche de visualisation de sorties (*visu-sort*) associée à la tâche calcul-covariance. Cette tâche calcule une application graphique qui est intégrée par SCARP dans la partie droite de la fenêtre d'interaction. D'autres interactions propres à cette application sont possibles pour obtenir des informations complémentaires ou plus précises (*Zoom ...*). L'éditeur d'attribut à gauche de la fenêtre permet de modifier la valeur de *nb-lags*.

La deuxième sous-tâche à modéliser séparément, le *calcul modal*, prend en entrée la fonction de covariance précédemment déterminée et donne en sortie des modes

numériques de vibration pour la séquence. Pour effectuer le calcul modal (cf. figure 5), il faut en premier lieu identifier différentes caractéristiques du modèle ARMA, d'abord son ordre *AR* et ensuite la valeur de la *projection* pour l'identification des paramètres. A partir de là, les modes numériques sont calculés. La décomposition de *calcul modal* nous conduit ainsi directement à la définition de trois autres tâches, *fixer-AR*, *fixer-projection* et *calcul-modes-numériques*.

La tâche *fixer-AR* permet de rechercher la "meilleure" valeur de l'ordre AR pour le modèle ARMA. Pour cela elle détermine d'abord un intervalle de valeurs possibles (*calcul_intervalle_ar*). Puis, une valeur provisoire pour la projection est fixée (*fixer_projection_provisoire*), car, même si la valeur de AR est fixée en premier, elle ne peut pas être déterminée indépendamment d'une valeur de projection. Ensuite, pour chaque valeur de AR possible, des modes numériques (*execute-calcul-sur-ar*) sont calculés. L'ensemble de ces modes numériques permet de choisir la valeur optimale pour AR (*choix_ar*).

La tâche *fixer-AR* a des caractéristiques spécifiques qui permettent de montrer l'intérêt de certains éléments du modèle de représentation de connaissances de SCARP. Premièrement, il faut effectuer une *itération sur* l'*intervalle* des valeurs possibles pour AR. Cette itération peut directement être modélisée par un contexte d'*itération* sur un *intervalle*. Deuxièmement, il est nécessaire d'effectuer plusieurs interactions avec l'utilisateur avant et pendant la résolution de *fixer-AR*. La figure 9 montre la définition de *fixer-AR* avec SCARP.

```
{ fixer-AR              est-un          =          tâche-exécutable ;
                        sorte-de        =          tâche-ifremer  ;
   &entrée-globale      fct-cov         $un        fonction_de_covariance ;
   …
   &entrée-stratégique  nb_modes        $un        entier
                                        $pré-tâche estime_nombre_de_modes;
   …
   &sortie-globale      AR              $un        entier ;
                        stratégie       $valeur    ( séquence calcul_intervalle_ar
                                                       ( utilisateur fixer_projection_provisoire )
                                                       ( itération exécute_calcul_sur_ar )
                                                       ( utilisateur choix_ar ) ) ;
   &pré-tâche           estime_nombre_de_modes      $un  { demande ... } ;
   &sous-tâche          calcul_intervalle_ar        $un  { contexte-simple ... } ;
   &sous-tâche          fixer_projection_provisoire $un  { demande ... } ;
   &sous-tâche          exécute_calcul_sur_ar       $un  { itération-intervalle ... } ;
   &sous-tâche          choix_ar                    $un  { demande ... } }
```

Figure 9 : Définition de la tâche fixer_AR. *Fixer_AR* est une *tâche exécutable* qui prend en entrée une fonction de covariance et détermine en sortie une valeur pour *AR*. Pour le calcul il y a besoin d'un paramètre d'entrée supplémentaire : *nb_modes*. Une pré-tâche (*estime_nombre_de_modes*) permet d'aider l'utilisateur à déterminer sa valeur. La stratégie de résolution de *fixer-AR* consiste dans une séquence de quatre sous-tâches, parmi lesquelles il y a une itération et deux tâches dont le résultat est fourni ou validé par l'utilisateur.

Les interactions nécessaires pour résoudre *fixer-AR* sont prédéfinies dans la base de connaissances et directement modélisables par des contextes *demande* SCARP :

- avant de lancer la résolution de fixer-AR, l'utilisateur doit renseigner la valeur d'une entrée stratégique de cette tâche : le nombre de modes à calculer.
- pendant la résolution l'utilisateur doit ensuite fixer la valeur de la projection provisoire.
- à la fin, l'utilisateur doit choisir la valeur définitive de AR.

Pour la première de ces interactions l'utilisateur a le choix entre, soit directement fixer le nombre de modes numériques à calculer, soit activer une pré-tâche (*estime-nombre-de-modes*). Cette pré-tâche fait référence à une stratégie d'aide qui va calculer et visualiser graphiquement les spectres des signaux initiaux. Cette visualisation facilitera pour l'utilisateur le choix de la valeur demandée. La figure 10 montre la définition de cette pré-tâche et la fenêtre d'interaction résultante.

```
&pré-tâche   estime_nombre_de_modes
  $un   { demande                                    { calcul_spectres
          exécute    $un      calcul_spectres ;        &tâche-visu      visu-sort
          obtenir    $valeur  .nb-modes ;                     $un        { visualisation
          demander   $valeur  nb-mod ;                                     exécute  $un  visu-spectres ;
          nb-mod     $un      entier                                       ... }
                     $com     "Donnez le nombre ..." ;      ... }
           ... }
```

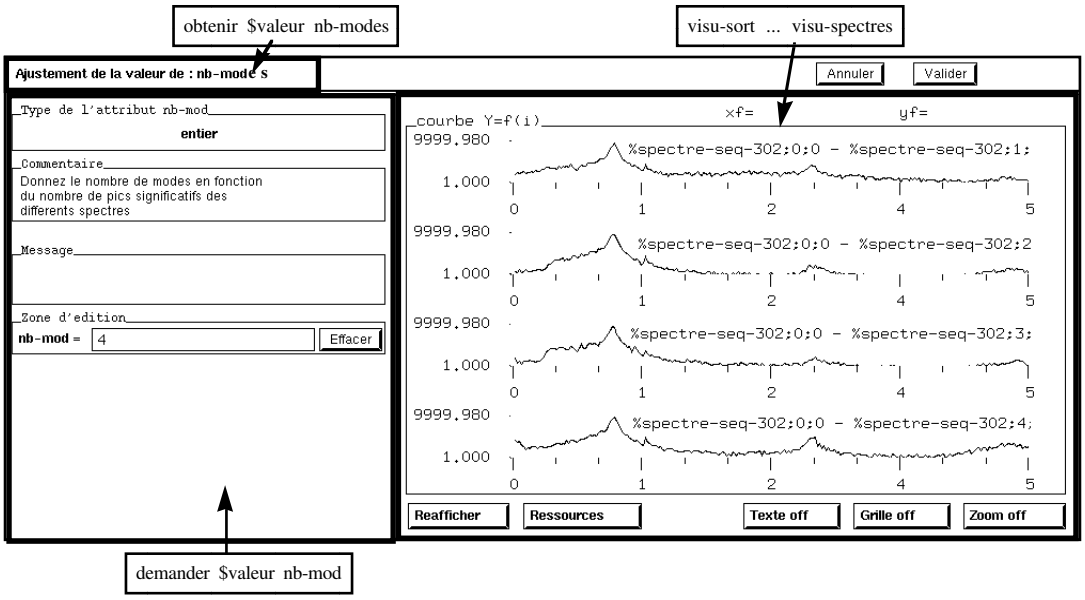


Figure 10 : Formalisation des connaissances pour aider l'utilisateur à estimer le nombre de modes à calculer et fenêtre d'interaction SCARP résultante. La pré-tâche *estime_nombre_de_modes* fait référence à la tâche *calcul-spectres* qui est intégrée via un contexte *demande* dans la tâche englobante fixer_AR. Ce contexte indique que la valeur de l'attribut *nb-modes* de la tâche englobante est à obtenir et que cette valeur correspond à l'attribut *nb-mod* du contexte. La définition de celui-ci contient un commentaire (*$com*) qui est utilisé pour expliquer à l'utilisateur la signification de la valeur demandée.

Les deux autres interactions correspondent à des tâches utilisateurs. Pour la première un programme est disponible qui permet de proposer une valeur que l'utilisateur peut accepter telle quelle, ou bien modifier. La deuxième consiste à visualiser graphiquement les pôles des modes numériques calculés et cumulés pendant l'itération. Ce graphique est interprété par l'utilisateur, et lui permet de choisir une valeur définitive pour AR.

La tâche *fixer-projection* permet ensuite de rechercher la "meilleure" valeur de *projection* pour identifier les paramètres du modèle ARMA. Son principe correspond à *fixer-AR* : des modes numériques sont calculées sur une plage de valeurs possibles pour la projection. Ensuite, la valeur définitive de la projection est déterminée. Ceci correspond à une tâche utilisateur : les pôles des modes numériques sont visualisées, ce qui permet à l'utilisateur de choisir la valeur optimale de projection.

Dans *calcul-modes-numériques,* les modes numériques finaux sont calculés, en deux étapes : d'abord les valeurs des paramètres du modèle ARMA sont déterminées (*identification*), ensuite les modes numériques de vibration sont extraits (*extraction*). Ces étapes sont directement réalisées par exécution de programmes intégrés par des méthodes *externes* dans la base de connaissances. D'autres méthodes externes permettent par exemple de calculer l'intervalle des valeurs possibles pour AR ou de proposer une valeur provisoire pour la projection. Des méthodes *internes* sont utilisées pour réaliser les tâches de visualisation.

Au fur et à mesure de l'analyse des tâches de cette application, les différentes entités manipulées sont identifiées. Elles sont regroupées en partie dans la figure 11. Cette figure montre aussi un exemple pour la phase de complétion de la base de connaissances : l'entité *spectre_séquence* a été rajoutée ultérieurement ainsi qu'une méthode permettant de calculer ce spectre.

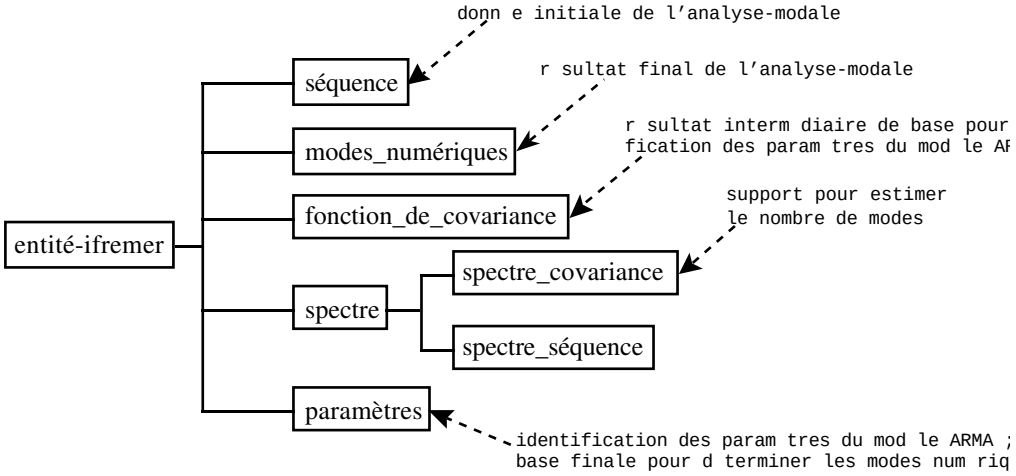


Figure 11 : Hiérarchie des classes modélisant les entités. Cette figure montre un extrait de la hiérarchie des entités identifiées pour l'application IFREMER. Elle indique pour chaque entité identifiée à partir de l'analyse de la tâche principale *analyse-modale*, pourquoi elle a été introduite. La classe *spectre_séquence* a été rajoutée séparément, puisque, avant d'effectuer une analyse modale, il peut être intéressant de déterminer (et de visualiser) le spectre de la séquence à analyser. Les deux classes *spectre_covariance* et *spectre_séquence* ont ensuite été regroupées sous *spectre*.

Nous avons exposé dans cette section les caractéristiques essentielles de l'application développée à l'IFREMER. Sa base de connaissances contient en tout une quinzaine de tâches, une quinzaine de méthodes et une quinzaine d'entités.

4.1.3. Aspects importants de l'analyse

L'analyse *descendante* comme nous l'avons proposée jusqu'ici est une démarche appropriée pour la construction d'une nouvelle base de connaissances. Toutefois, en réalité l'analyse est rarement complètement descendante : il peut toujours y avoir des allées et venues entre les différents niveaux de décomposition. L'analyse reste ainsi descendante en générale, mais contient des parties ascendantes localisées. Si, par exemple, une tâche est identifiée via l'analyse d'une première tâche complexe qui l'utilise en tant que sous-tâche, et que la même tâche est utilisée par une deuxième tâche complexe, ceci peut conduire à un raffinement de sa définition. Une telle double utilisation d'une tâche peut en particulier conduire à une séparation plus nette de sa propre définition et de la définition de ses contextes d'utilisation.

L'analyse descendante identifie tout d'abord toutes les tâches, les méthodes et les entités qui sont nécessaires pour modéliser la résolution des problèmes essentiels du domaine d'application. Nous avons proposé de compléter cette analyse par une phase de recherche et d'intégration d'autres méthodes et d'autres entités existantes, qui ne seraient pas encore définies, car non immédiatement nécessaires pour la résolution de ces problèmes essentiels. Ainsi, la connaissance sur les tâches peut ultérieurement être complétée suivant une analyse *ascendante*, basée sur une construction de nouvelles tâches à partir des briques de base présentes dans la base de connaissances (tâches, méthodes, entités). Ceci peut être fait de façon interactive, en utilisant le système SCARP

Un autre aspect important pour la souplesse de la construction de nouvelles tâches est la *granularité des méthodes* définies dans la base de connaissances. Car, plus les méthodes sont élémentaires, c'est-à-dire plus la décomposition des tâches a été poussée lors de la conception de la base de connaissances, plus les briques de base sont fines et peuvent être combinés de différentes façons. Nous montrons un exemple dans la section 4.2.2.3.

4.2. RESOLUTION EN COOPERATION

Dans l'exemple présenté dans la première partie de ce chapitre, toute la coopération était contenue dans la pré-définition des interactions a priori nécessaires pendant la résolution. Mais, dans SCARP, la coopération pendant la résolution de problèmes ne se limite pas à la gestion des interactions a priori nécessaires. Une coopération peut aussi s'établir a posteriori. Les différentes possibilités de coopération a posteriori ont été présentées au chapitre trois. Nous les rappelons rapidement. Ensuite, le processus coopératif de résolution de problèmes est illustré à l'aide d'un exemple suffisamment complexe, de l'application SLOT en analyse de données exploratoire [Chev94].

4.2.1. Possibilités de coopération

Comme détaillé au chapitre trois, en dehors des interactions prédéfinies dans la base de connaissances, l'utilisateur peut intervenir dans SCARP de trois manières dans le processus de résolution par planification hiérarchique :

- il peut décider de prendre le contrôle des différentes phases de résolution de problèmes et des décisions qui y sont prises. Ainsi l'utilisateur peut intervenir dans chaque type de décision à prendre sur les niveaux d'abstraction et de décomposition par lesquels passe la planification hiérarchique.
- il peut remettre en cause toutes les décisions prises pendant le processus de résolution et ainsi créer différentes versions de ce processus. S'il s'avère que des décisions prises au cours de la résolution n'ont pas été optimales,

l'utilisateur peut les corriger. Ceci concerne d'une part ses propres décisions, d'autre part les décisions prises par le système.

- l'utilisateur peut interactivement définir de nouveaux problèmes et explorer de nouvelles stratégies de résolution. Il a pour cela accès à toutes les tâches définies dans la base de connaissances, sur différents niveaux d'abstraction et de décomposition. Il peut modifier leur définition et les combiner pour créer de nouvelles tâches.

4.2.2. Exemple de coopération : l'application SLOT

L'application choisie comme exemple pour illustrer la coopération pendant le processus de résolution concerne l'analyse exploratoire de données multivariées. Les tâches qui sont à modéliser dans ce domaine sont beaucoup plus complexes que dans l'application de l'IFREMER. Cette application permet ainsi de mieux illustrer le processus de résolution coopératif avec SCARP. Dans cette section le domaine d'application est tout d'abord introduit rapidement. Ensuite les tâches, les méthodes et les entités essentielles de la base de connaissances sont identifiées. Finalement un exemple de session avec SLOT montre comment l'utilisateur peut intervenir dans la résolution et comment il peut explorer interactivement de nouvelles tâches.

4.2.2.1. Description du domaine

L'analyse de données consiste à faire une étude globale de données, représentant initialement un ensemble de variables mesurées sur un ensemble d'individus. Pour pouvoir effectuer cette analyse, ces données sont représentées sous forme de tableaux. L'analyse passe essentiellement par des calculs sur ces tableaux. Les résultats de ces calculs sont ensuite interprétés, généralement en passant par des représentations graphiques.

Même si l'application SLOT n'est pas limitée à l'ordination linéaire simple, la suite de la discussion sera restreinte à ce sous-domaine déjà suffisamment complexe de l'analyse de données. Il regroupe trois types d'analyse plus spécifiques : l'analyse en composantes principales, l'analyse factorielle des correspondances et l'analyse des correspondances multiples. Ce domaine est complexe pour plusieurs raisons :

- les données initiales à analyser, ou plutôt les variables mesurées, peuvent être de différentes natures : quantitatives, qualitatives ordonnées ou qualitatives nominales. Selon la nature des variables, différentes opérations leur sont applicables ou non : la multiplication par exemple n'a de sens que pour des valeurs de variables quantitatives.
- les mêmes données initiales peuvent être analysées de différentes manières, selon que l'on veut mettre en évidence les ressemblances (ou les différences) entre individus ou entre variables.
- Selon l'objectif de l'analyse et la nature des données initiales, celles-ci peuvent être codées dans un tableau de différentes façons. La plus simple consiste à croiser directement les individus examinés avec les variables mesurées. Une autre consiste à établir des tableaux de contingences contenant les fréquences d'association entre les modalités de deux caractères qualitatifs. De plus, lors du codage, les données initiales peuvent éventuellement être centrées ou normées, pour ajuster l'influence de certains individus ou variables sur l'analyse.

Une analyse de données doit répondre à un problème précis. En fonction de ce problème, des *individus*, les différents champs d'expérimentation, et des *variables* à

mesurer ont été choisis, et des jeux de test tirés. C'est ici que commence l'analyse proprement dite. Ces jeux de test, les données initiales, sont tout d'abord représentés ou codés dans un tableau. Ce tableau initial est difficile à appréhender en raison de sa nature multivariée et du volume des données recueillies. La première étape d'une analyse consiste par conséquent en la recherche d'une représentation dans un espace de (plus) faible dimension. Pour cela de nouveaux caractères sont synthétisés par combinaisons linéaires des caractères initiaux (*individus / variables*). Ceci est effectué par application de méthodes factorielles et linéaires [Sapo90] et se ramène finalement à faire du calcul matriciel. Le tableau résultant est ensuite interprété à l'aide de représentations graphiques (figure 12).

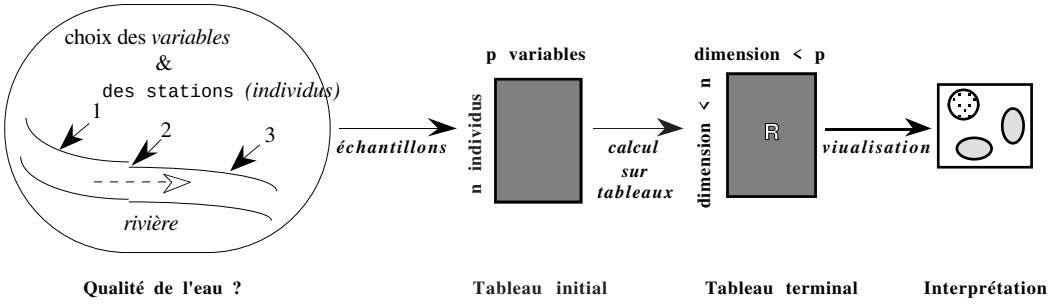


Figure 12 : Exemple simple d'analyse de données. Le but de l'analyse est d'évaluer la qualité de l'eau d'une rivière en fonction de l'endroit, car la qualité change avec les caractéristiques du milieu du rivage. Pour cela, il faut d'abord choisir les *n stations* où on va prendre des échantillons et les *p variables* significatives (la teneur en oxygène ...) à mesurer. Ainsi on obtient le *jeu de données*. Ce *jeu de données*, représenté directement dans un tableau à *n* lignes et *p* colonnes, est ensuite analysé. Dans une première phase, il est recodé. Le résultat de cette phase est un tableau de plus faible dimension qui est plus facile à interpréter. Dans une deuxième phase, l'interprétation est effectuée à l'aide de représentations graphiques.

4.2.2.2. Extraits de la base de connaissances

Avant de donner un exemple de session avec SLOT, qui montrera comment exploiter les capacités de coopération de SCARP, il est nécessaire de donner quelques précisions sur la formalisation des connaissances. L'application, qui est couramment mise à jour par F. Chevenet, se compose en tout d'à peu près 200 tâches, de 60 méthodes et d'une cinquantaine d'entités. La description que nous présentons ici se restreint à la présentation de quelques caractéristiques essentielles et des détails qui sont nécessaires pour la compréhension de l'exemple qui suit.

La base de connaissances de SLOT modélise la résolution de problèmes en analyse de données à plusieurs niveaux d'abstraction et de complexité (figure 13).

L'exemple qui sera utilisé pour illustrer la coopération est celui d'une analyse classique en ordination linéaire simple. Celle-ci se déroule en plusieurs étapes (figure 14) :

- *préparer analyse* ; ceci consiste essentiellement à gérer des fichiers.
- *faire triplet* ; le triplet avec lequel l'analyse est effectuée est construit, c'est-à-dire que les données sont codées dans un tableau initial et que les matrices de pondération des lignes et des colonnes sont déterminées.
- *préparer diagonalisation* et *diagonalisation* ; la matrice est diagonalisée, ses valeurs et ses vecteurs propres sont calculés.

- *sélection facteurs* ; en fonction des valeurs propres, l'utilisateur choisit le nombre de facteurs à conserver pour le reste de l'analyse, c'est-à-dire le nombre de caractères de synthèse qu'il souhaite calculer à partir des caractères initiaux.
- *calcul projection* ; la matrice est projetée dans le nouvel espace à plus faible dimension.
- *finir analyse* ; ceci consiste essentiellement en la gestion de fichiers.

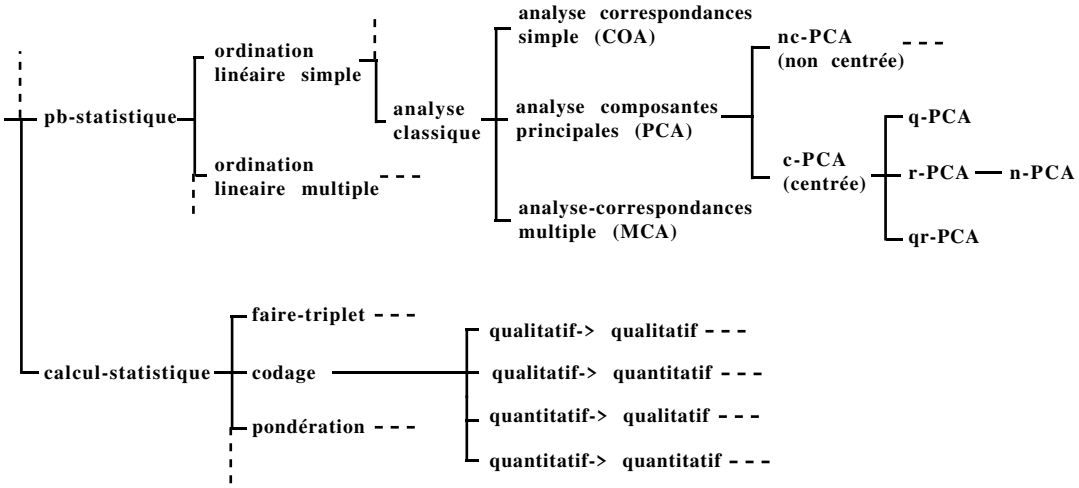


Figure 13 : Tâches de la base de connaissances SLOT. Des tâches modélisent les *problèmes statistiques* généraux à différents niveaux d'abstraction. Leur résolution passe en fait par l'enchaînement de tâches plus élémentaires, effectuant des *calculs statistiques* (*codage, pondération, etc*.).

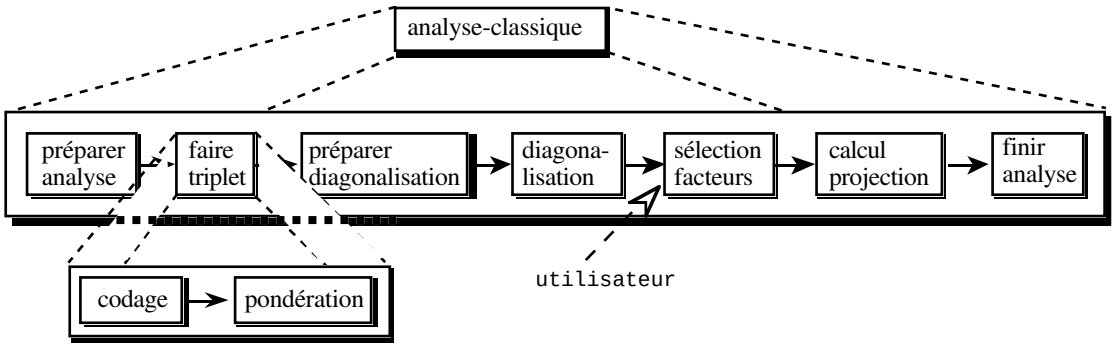


Figure 14 : Stratégie de résolution définie pour la tâche *analyse classique*. Il s'agit d'une séquence de plusieurs sous-tâches, elles-mêmes complexes, comme par exemple la sous-tâche *faire triplet*. L'utilisateur intervient dans la résolution pour *sélection*ner le nombre de *facteurs* à conserver, une décision importante pour le reste de l'analyse.

Les différents types plus spécifiques d'analyse (COA, PCA, MCA etc.) se distinguent essentiellement dans la manière dont le triplet, avec lequel l'analyse est effectuée, est calculé, et dans le codage et le calcul de matrices de pondération. Le fait de pouvoir effectuer certains calculs dépend de la nature des données. Chaque type d'analyse est ainsi approprié au traitement de certains types de données et passe par le calcul d'un type de triplet spécifique.

Les entités essentiellement manipulées dans SLOT sont des matrices, modélisées par une hiérarchie de classes décrivant différents types de tableaux existants (figure 15). En fonction du type de tableau, on utilise :

- pour les matrices de type mesurables l'analyse des composantes principales,
- pour les matrices de type modalités l'analyse des correspondances multiples,
- pour les matrices de type contingences l'analyse de correspondances simple.

SLOT manipule par ailleurs des entités représentant des graphiques et d'autres entités qui n'interviennent pas dans l'exemple présenté par la suite. Les méthodes modélisent en SLOT essentiellement différents types de calcul matriciel et graphique.

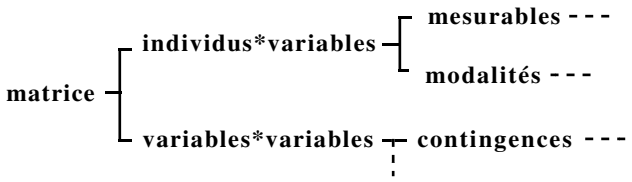


Figure 15 : Extrait de la modélisation des différents types de matrices manipulés par SLOT. Des matrices sont distinguées selon la signification des lignes, des colonnes et du type d'information représentée par ses éléments (des valeurs qualitatives ou quantitatives ou des fréquences).

4.2.2.3. Exemple d'une session

Après avoir introduit tous les aspects importants pour pouvoir comprendre l'application, nous présentons maintenant l'exemple d'une session avec SLOT. Cet exemple permet d'illustrer l'intérêt des trois volets de coopération pendant la résolution de problèmes cités au début de la section 4.2 : le contrôle d'une phase de résolution par l'utilisateur, la remise en cause de décisions et l'exploration interactive de nouvelles tâches.

Notre exemple consiste dans la résolution d'une analyse classique. Comme nous l'avons vu dans la section précédente, dans SLOT les connaissances sont toujours représentées à différents niveaux d'abstraction. Ainsi, la phase de spécialisation prend une importance majeure pour le processus de résolution. Si cette phase est entièrement contrôlée par le moteur, celui-ci va essayer de l'optimiser selon des critères fixes qui ne correspondent pas forcément aux objectifs de l'utilisateur. Par conséquent, il peut être intéressant pour celui-ci de contrôler cette phase.

Après avoir indiqué qu'il prend le contrôle de la phase de spécialisation, l'utilisateur choisit dans notre exemple (via le sélecteur de problèmes) d'effectuer une analyse classique. Il fournit en entrée une instance de la classe *mesurables* qui représente les données à analyser. Le processus de résolution passe ensuite par la phase de spécialisation qui est contrôlée par l'utilisateur. La fenêtre d'interaction est activée et permet à l'utilisateur de choisir parmi les types d'analyse possibles celui qu'il souhaite appliquer (figure 16).

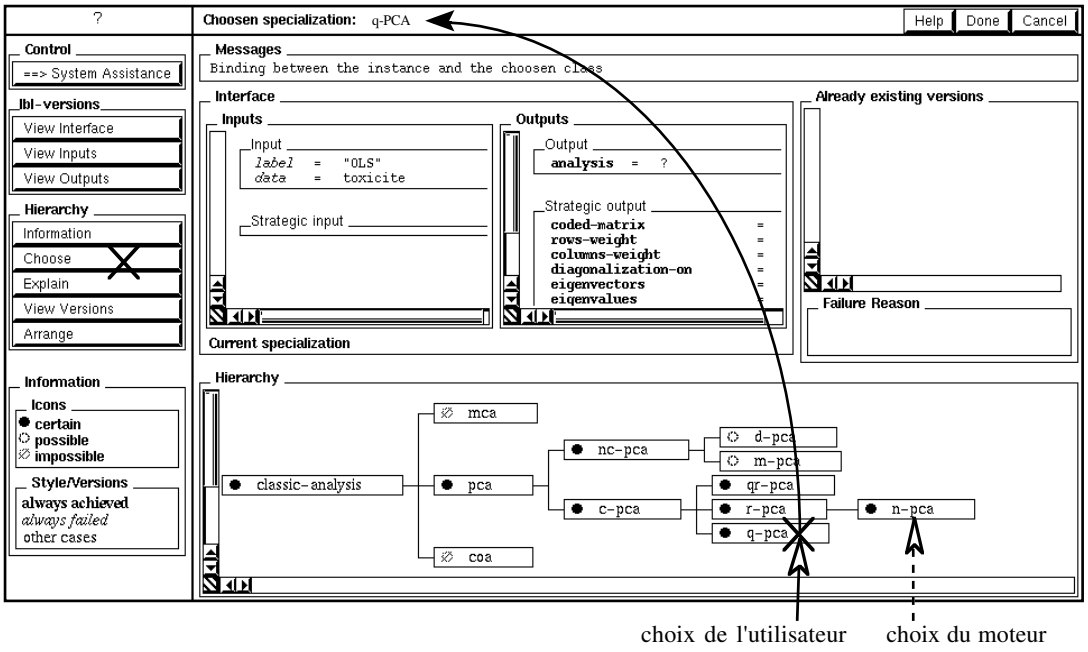


Figure 16 : Contrôle de la phase de spécialisation par l'utilisateur. En fonction des caractéristiques des entrées fournies, les classes dans la hiérarchie correspondant à l'analyse classique sont marquées sûres, possibles ou impossibles. Si le moteur contrôlait cette phase, il choisirait *n-PCA*, la classe sûre la plus spécifique, mais ici l'utilisateur choisit *q-PCA* ce qui correspond mieux à ses objectifs.

L'utilisateur choisit une *q-PCA*, c'est-à-dire une analyse en composantes principales centrées par colonnes. L'analyse se déroule comme décrit dans la stratégie de résolution associée à la q-PCA : l'analyse est ouverte, le triplet correspondant à l'analyse est construit et la diagonalisation de la matrice représentant les données est préparée et effectuée. Ensuite l'utilisateur intervient pour sélectionner dans une tâche-utilisateur le nombre de facteurs à conserver. Il interprète ici le diagramme des valeurs propres que lui montre le système.

La décision sur les facteurs à conserver est particulièrement importante, puisque le nombre choisi correspond au nombre de dimensions de l'espace dans lequel les données seront projetées. S'il est trop grand, le résultat de cette projection sera difficile à interpréter, s'il est trop petit il ne reflètera pas suffisamment bien les propriétés des données. Les valeurs propres correspondent en fait au pourcentage de l'information initiale qui est contenu dans chaque dimension.

Finalement, l'utilisateur interprète les résultats obtenus. Il n'est pas sûr d'avoir choisi un nombre de facteurs suffisamment élevé. Pour vérifier si un nombre plus élevé aurait conduit a une interprétation différente, il modifie ce nombre. Pour cela, il sélectionne, à l'aide de la souris, le noeud "U" correspondant. La fenêtre d'interaction correspondante s'active, et permet de modifier directement la valeur choisie précédemment pour le nombre de facteurs à conserver (figure 17). La validation de cette modification conduit à la création d'une nouvelle version du processus de résolution.

L'analyse de type q-PCA a été possible parce que ce type d'analyse était défini dans la base de connaissances SLOT. Considérons maintenant le cas où cette définition ne serait pas encore présente, mais où une autre, par exemple l'analyse de type n-PCA, serait déjà définie. Précisons que la différence entre ces deux types d'analyse réside dans la réalisation de l'étape du codage : tandis que pour la n-PCA, elle consiste à normer la matrice, c'est-à-dire à la centrer par colonnes et ensuite à diviser les valeurs dans chaque

ligne par leur moyenne, pour la q-PCA cette étape consiste à centrer par lignes, ce qui peut être décrit par une transposition de la matrice, un centrage par colonnes et une nouvelle transposition de la matrice résultante[1].

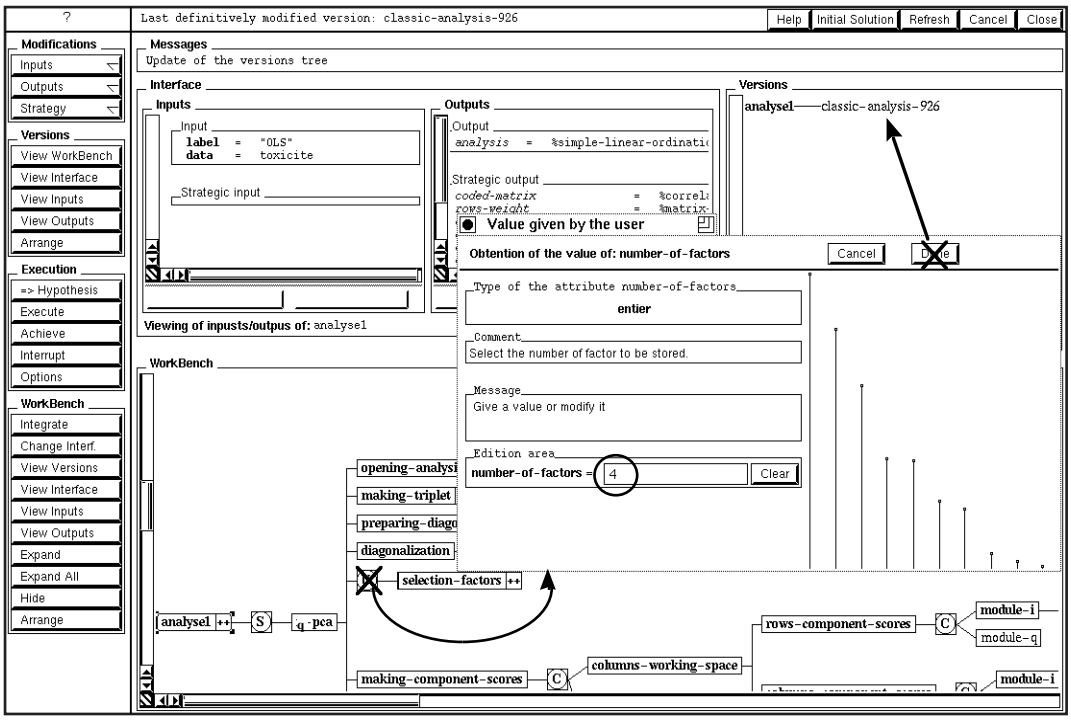


Figure 17 : Remise en cause du nombre de facteurs à conserver. Le nombre de facteurs initialement choisi n'était peut-être pas optimal. Ainsi l'utilisateur décide de modifier son choix et de garder finalement *4* facteurs. Il valide sa modification, ce qui conduit à la création d'une nouvelle version du processus de résolution.

L'analyse de type q-PCA peut ainsi être définie interactivement pendant une session SCARP. Pour cela l'utilisateur reprend une tâche dont la définition ressemble à la tâche qu'il veut définir (ou une tâche vide s'il n'y en a aucune) et modifie sa définition. Ici il utilise la n-PCA. Les étapes générales pour une telle modification sont la modification de la définition des entrées et des sorties, et de la stratégie de résolution et ensuite l'exécution de cette nouvelle tâche pour indiquer le flot de données associé. La figure 18 montre comment se déroule notre exemple : la définition des entrées ou des sorties est en fait la même pour n-PCA et q-PCA, mais la stratégie de résolution doit être modifiée. Cette modification ne concerne que l'étape de codage qui se trouve au troisième niveau de décomposition de la stratégie de résolution. La stratégie définie pour n-PCA est donc repris et "éclatée" jusqu'au trosième niveau. Puis la stratégie à appliquer pour résoudre l'étape de codage est redéfinie. Ceci conduit récursivement à la création de nouvelles tâches à tous les niveaux intermédiaires de décomposition, définies par des sous-classes de la classe *tâche-utilisateur*. Ensuite, la nouvelle tâche globale, q-PCA, est exécutée et le flot de données est indiqué par l'utilisateur. Puis elle est validée et sa définition est intégrée dans la base de connaissances. Cette tâche peut ainsi par la suite être resélectionnée par l'utilisateur et sa résolution gérée par le système. La figure 19 illustre comment les interactions système-utilisateur sont gérées, qui sont nécessaires pour effectuer la définition et l'exécution.

---

[1] Etant donné son utilisation fréquente pendant le calcul matriciel, la méthode nécessaire correspondant à la transposition d'une matrice est définie dans la base de connaissances.

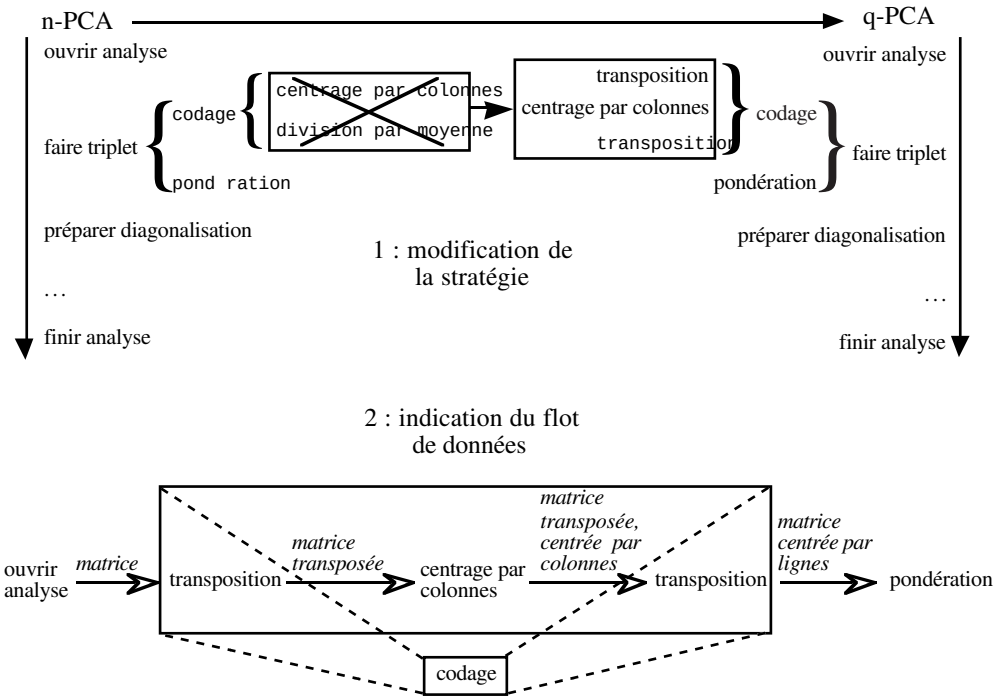


Figure 18 : Création d'une tâche (q-PCA) à partir d'une tâche existante (n-PCA). La stratégie de résolution de ces tâches se distinguent dans l'étape du codage (1). Après avoir modifié la stratégie de résolution, l'utilisateur lance l'exécution de la nouvelle tâche et indique le flot de données (2).

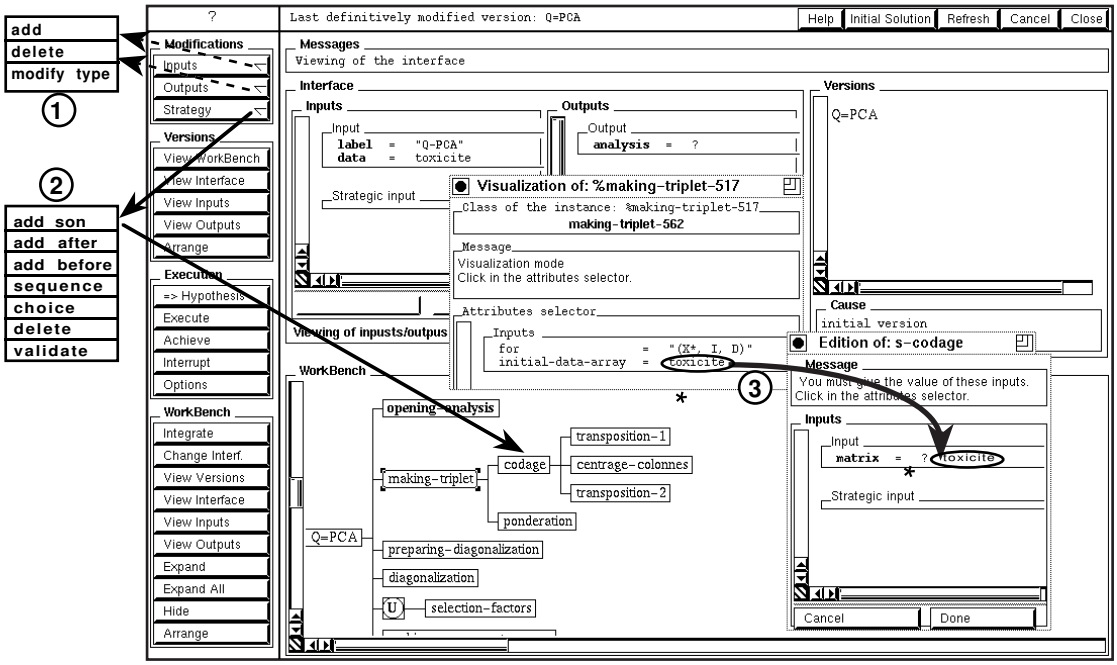


Figure 19 : Interactions pour la définition de la tâche q-PCA à partir de n-PCA. Cet exemple ne nécessite pas de modification de la définition des entrées ou des sorties qui passeraient par le menu 1. La re-définition de la stratégie consiste dans la re-définition des sous-tâches de l'étape du codage via le menu 2. Ensuite, pendant l'exécution exemplaire, le flot de données est indiqué avec la souris. Ici l'entrée de la (sous-)tâche *making-triplet* (la matrice *toxicité*) est transférée dans l'entrée de sa sous-tâche *codage* (3).

4.3. BILAN

Nous avons montré dans ce chapitre comment un domaine est analysé, dans l'objectif de développer une application SCARP, et comment cette application est ensuite exploitée, de façon coopérative, pour la résolution de problèmes. L'analyse d'un domaine d'application est guidée par l'identification des tâches principales à résoudre. Chaque tâche est séparément analysée, ce qui conduit au fur et à mesure que l'analyse progresse, à l'identification des tâches plus élémentaires, des méthodes et des entités du domaine. L'analyse identifie en même temps les interactions a priori nécessaires avec l'utilisateur pendant le processus de résolution.

L'analyse et la formalisation initiale des connaissances d'un domaine permettent de créer une base de connaissances qui correspond à la résolution de problèmes typiques. Leur résolution peut ensuite être gérée par SCARP, faisant intervenir l'utilisateur plus ou moins — selon son propre choix — dans le processus de résolution. L'utilisateur peut contrôler de façon globale différentes phases de résolution de problèmes, remettre en cause toutes les décisions prises pendant le processus de résolution et explorer de nouvelles tâches.

En effet, si la base de connaissances n'est pas complète, c'est-à-dire si le problème que veut résoudre l'utilisateur ou la stratégie qu'il veut appliquer ne sont pas présents dans cette base, l'utilisateur peut les définir de façon interactive, pendant une session. Comme briques de base il peut utiliser toutes les tâches, toutes les méthodes et toutes les entités, définies à différents niveaux d'abstraction dans la base de connaissances. Ainsi la connaissance contenue initialement dans la base peut être augmentée au cours de sessions successives.

Mais la création de nouvelles tâches doit encore être complétée par un mécanisme qui placerait la nouvelle classe résultante à l'endroit approprié dans la hiérarchie correspondante. Ce mécanisme devra aussi fonctionner de manière coopérative, c'est-à-dire en interaction avec l'utilisateur, car, ayant défini la tâche, l'utilisateur est le seul à connaître exactement sa signification. Ainsi, il doit au moins valider les décisions du système. Par ailleurs, la définition effectuée de manière interactive doit peut-être être modifiée dans une certaine mesure, pour pouvoir être insérée dans une hiérarchie de classes. Elle doit être adaptée au moins afin que ses (noms d')attributs correspondent à la définition de celles de ses futures sur-classes. Aussi cette modification doit se faire en accord avec l'utilisateur qui a créé la tâche.

CONCLUSION

L'objectif de cette thèse était de concevoir un système coopératif d'aide à la résolution de problèmes, capable aussi bien de résoudre des problèmes de la manière la plus autonome et automatique possible, que de permettre à un utilisateur d'intervenir ponctuellement dans le processus de résolution ou même de le diriger complètement. Pour cela, nous avons proposé avec le système SCARP une modélisation des connaissances et du processus de résolution de problèmes par planification hiérarchique. Ceci permet non seulement de gérer efficacement le processus de résolution par le système mais rend possible en même temps une coopération entre le système et son utilisateur, et cela sur tous les niveaux d'abstraction et de décomposition introduits.

Le modèle SCARP n'est pas dédié à un type spécifique de problèmes, comme par exemple le diagnostic ou la conception. Nous proposons avec ce modèle un moyen de description de connaissances général qui permet a priori de décrire les connaissances de résolution pour tout problème décomposable en sous-problèmes. Il a été expérimenté avec succès dans des domaines d'application comme le traitement du signal [Jean&91], l'analyse de données [Chev&93], l'automatique [Bass&94] ou la biologie moléculaire [Médi&93].

La modélisation proposée avec SCARP est basée sur l'identification de trois types de connaissances que nous considérons nécessaires pour la résolution de problèmes par planification hiérarchique :

- les *tâches* qui représentent les problèmes existants dans un domaine et leur associent des stratégies de résolution par décomposition en sous-tâches plus élémentaires,
- les *méthodes* modélisant les opérations ou inférences possibles qui permettent de résoudre directement des tâches élémentaires,
- les *entités* représentant les objets manipulés par les tâches et les méthodes.

Pour la représentation des connaissances, nous avons choisi le modèle à objets comme base de départ. Celui-ci est particulièrement intéressant pour modéliser les connaissances de résolution de problèmes qui repose sur une planification hiérarchique puisqu'il permet de représenter naturellement, via des hiérarchies, différents niveaux d'abstraction. De plus, le modèle à objets prévoit, grâce au mécanisme de classification, un moyen puissant pour exploiter cette organisation hiérarchique des connaissances.

Le modèle à objets permet de représenter directement les entités manipulées pendant la résolution de problèmes. Nous l'avons étendu pour pouvoir représenter les tâches et les méthodes. Une première originalité de notre approche résulte justement du fait que les tâches soient aussi modélisées par des classes d'objets : ceci permet de représenter des problèmes à différents niveaux d'abstraction successifs et de structurer en même temps les stratégies de résolution associées. Cette organisation permet de spécifier un problème de façon de plus en plus précise et d'associer finalement à chacune des définitions la stratégie de résolution appropriée. Ceci permet d'utiliser, pendant la résolution, le mécanisme de classification pour guider le choix de la stratégie la plus appropriée au problème courant à résoudre.

Le modèle de tâches permet d'exprimer toutes les connaissances nécessaires pour faire contrôler la résolution de façon automatique par un système à base de connaissances.

Mais la résolution d'une tâche peut nécessiter des interactions avec l'utilisateur et un système coopératif doit être capable de les gérer convenablement. Pour cela, il a d'abord fallu identifier les différents types d'interactions qui peuvent a priori être nécessaires pendant la résolution et les intégrer dans la modélisation des connaissances. Ces interactions peuvent en fait concerner la résolution complète de (sous-)tâches par l'utilisateur ou alors certaines décisions ponctuelles, comme par exemple la validation des résultats d'une tâche. En conséquence, une deuxième spécificité du modèle SCARP réside dans l'intégration des interactions a priori nécessaires dans la représentation des connaissances.

Pour gérer la résolution par planification hiérarchique nous avons ensuite défini un mécanisme approprié : le moteur de tâches. Sa caractéristique essentielle est le fonctionnement par alternance de phases de planification et d'exécution, ce qui permet d'entamer une résolution avec une stratégie très générale et de spécifier cette stratégie seulement au fur et à mesure de l'exécution, en fonction des résultats intermédiaires déjà obtenus.

Pendant le processus de résolution, différents types de décisions sont à prendre. Elles déterminent le cours de la résolution et sont par défaut contrôlées par le système. Mais un système coopératif doit permettre à son utilisateur de négocier dynamiquement le contrôle de ces décisions. Ainsi nous avons introduit dans SCARP la possibilité pour l'utilisateur de confier ou retirer au système la responsabilité de prendre ces décisions.

La résolution de problèmes consiste souvent en un processus essai-erreur : plusieurs hypothèses sont développées les unes après les autres, comparées entre elles, puis abandonnées et reprises au fur et à mesure de l'avancement dans la résolution pour finalement trouver la meilleure solution. Par conséquent, aucune décision initiale ni modification ne doit être définitive. En ce qui concerne les décisions prises, l'utilisateur doit pouvoir les remettre en cause pour imposer son propre choix et pour essayer différentes possibilités. Chacune de ces possibilités correspond en fait à une nouvelle version du processus de résolution. Cette facilité pour l'utilisateur de remettre en cause toutes les décisions prises pendant le processus de résolution de problèmes, de créer et de gérer ainsi différentes versions de ce processus est une autre originalité de SCARP.

Etant donné la complexité, le grand volume et l'évolutivité des connaissances à acquérir et à représenter, toute base de connaissances est incomplète. Pour un système d'aide à la résolution de problèmes, ceci signifie que les connaissances sur les problèmes et les stratégies de résolution possibles ne sont jamais modélisées de façon exhaustive, qu'il existe toujours des problèmes et des stratégies qui ne sont pas représentés. Surtout dans un système qui se veut coopératif et où l'utilisateur est considéré "expert" au même titre (si ce n'est plus) que le système lui-même, l'utilisateur doit avoir la possibilité de définir interactivement de nouveaux problèmes et de nouvelles stratégies de résolution et de les intégrer dans la base de connaissances. Nous avons donc introduit cette possibilité dans SCARP.

Mais cette fonctionnalité doit encore être complétée : pour l'instant ces nouvelles tâches sont activables manuellement par l'utilisateur, mais non pas automatiquement par le système, car elles ne sont pas encore intégrées à leur place adéquate dans la structure hiérarchique de la base de connaissances. Des algorithmes de reconnaissance de plans et de classification de classes devraient permettre d'effectuer ce placement. Ils sont basées sur un appariement de la définition d'une nouvelle tâche avec la définition des tâches qui forment la structure hiérarchique de la base de connaissances. En parallèle avec cet appariement et pour pouvoir définitivement insérer la définition de la nouvelle tâche dans cette hiérarchie, il peut être nécessaire de l'adapter aux normes établies auparavant dans la base de connaissances.

A l'issue de cette thèse, deux perspectives plus générales semblent particulièrement intéressantes :

- l'élargissement des capacités coopératives du système et
- l'intégration complète du modèle de tâches dans le modèle à objets.

Avec SCARP nous nous sommes concentrés sur un aspect spécifique de ce qui peut être considéré comme une coopération entre un système d'aide à la résolution de problèmes et son utilisateur : coopération dans le sens où elle doit permettre à l'utilisateur d'intervenir de différentes manières et à des degrés variables dans le processus de résolution de problèmes jusqu'au point de le diriger complètement. Nous n'avons pas tenté de donner au système les connaissances nécessaires pour décider de son côté dynamiquement quand il faut faire intervenir l'utilisateur. Ceci dépend des connaissances de celui-ci. Il faudrait donc pour cela modéliser l'utilisateur, ce qui pourrait permettre au système de s'adapter à ses besoins particuliers.

La modélisation de l'utilisateur doit essentiellement permettre au système de comprendre les intentions et les actions qu'entreprend l'utilisateur pour pouvoir y réagir de façon coopérative. Dans un premier temps cette fonctionnalité pourrait être remplie par des algorithmes de reconnaissance de plans et de classement de classes comme nous les avons introduits plus haut : ils peuvent par exemple servir à aider un utilisateur qui n'arrive pas à finaliser la spécification interactive d'une nouvelle tâche. La partie déjà spécifiée de cette tâche pourrait être appariée avec la définition des tâches connues ; en fonction des tâches ressemblantes identifiées l'utilisateur pourrait ensuite être conseillé sur des actions à entreprendre. Pour remplir la même fonction, le raisonnement par analogie constitue une autre approche intéressante.

Enfin, l'intégration complète du modèle de tâches dans le modèle à objets, ceci permettrait de formaliser par des tâches les mécanismes d'inférence propre du modèle à objets, comme par exemple la classification [Gens&92] [Mari93]. Le fonctionnement de ces mécanismes serait ainsi facilement compréhensible pour l'utilisateur et celui-ci pourrait même intervenir dans leur déroulement. Aussi la modélisation avec le modèle SCARP de tâches de "plus haut niveau", comme l'explication du raisonnement effectué, semble intéressante. Non seulement les tâches du domaine d'application, mais aussi les tâches du système lui-même, pourraient ainsi être décrites avec ce même modèle. Le système SCARP pourrait donc non seulement être utilisé pour gérer la résolution de problèmes propres à un domaine d'application mais aussi pour gérer des tâches plus générales. Un exemple de ce type de problème est la conception de hiérarchies de classes, donc la structuration de la base de connaissances, à partir de connaissances non structurées [Euze93].

BIBLIOGRAPHIE

[Allp89] Allport D., *A computational architecture for co-operative systems,* Proceedings of the International Conference on Knowledge Based Computer Systems, Bombay, India, 1989, Springer, Lecture Notes in Artificial Intelligence 4440, pp 3-18.

[Bass&94] Bassot C., Michau F., *Aide à l'interprétation de courbes de simulation pour l'analyse des systèmes dynamiques,* 9ème Congrès RFIA, Paris, janvier 1994, Vol. 2, pp. 721-726.

[Beau94] Beaubouche r N., *ANAIS : raisonnement à partir de cas et résolution de problèmes,* thèse de doctorat, Université Paris 6, 1994, à paraître.

[Breu&88] Breuker J., Wielinga B., *Models of expertise in knowledge acquisition,* dans "Topics in expert system design", Guida G., Tasso C. (eds), North Holland Publishing Company, Amsterdam, 1988.

[Caho&91] Cahour B., Falzon P., *Assistance à l'opérateur et modélisation de sa compétence,* Intellectica,1991/2, 12, pp 159-186.

[Capp93] Capponi C., *Classification des classes par les types,* actes du congrès Représentation par Objets (Journée Classification), La Grande Motte, France, juin 1993, pp 215-224.

[Cauh91] Cauhapé D., *Modélisation et traitement des connaissances sur le temps et les tâches médicales pour les Systèmes Experts en Cancerologie,* thèse de doctorat, Université de Bordeaux I, Bordeau, juillet 1991.

[Chai93] Chaillot M., *Une Architecture de contrôle réactif pour la résolution coopérative de problèmes,* thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, septembre 1993.

[Chan86] Chandrasekaran B., *Generic tasks in knowledge-based reasoning : high-level building blocks for expert system design,* IEEE Expert, automne 86, pp 23-30.

[Chan87] Chandrasekaran B., *Towards a functional achitecture for intelligence based on generic information processing tasks,* IJCAI, Milan; Vol 2, 1987, pp 1183-1192.

[Chan&87] Chandrasekaran B., Smith J., Sticklen J. *Generic tasks as building blocks for knowledge-based systems: the diagnosis and routine design examples,* The Knowledge Engineering Review, No. 3, Vol. 3, 1988, pp. 183 - 210.

[Clan&84] Clancey W. J., *Heuristic classification*, Artificial Intelligence No. 27, Vol. 3, 1984, pp. 289-350.

[Chev&93] Chevenet F., Willamowski J., Jean-Marie F., *A development shell for cooperative problem solving environments,* Third International Conference on Expert Systems For Numerical Computing, Purdue (IN, USA), mai 1993, Computer Sciences Department, Purdue University, research report CSD-TR-93-028, CAPO Report CER-93-14, pp. 1-7; à paraître en version longue dans *Mathematics and Computers in Simulation*, Elsevier Science Publishers, Amsterdam (Pays-Bas).

[Chev94] Chevenet F., *Environnements coopératifs de résolution de problèmes pour l'analyse statistique. Représentations objets et analyse des données multivariées en écologie*, Thèse de doctorat, Université Claude Bernhard Lyon I, 1994, à paraître.

[Clay84] Clay B. D., *Inference ART programming tutorial Vol. 2, a first look at viewpoints,* Inference Corporation, Los Angeles (CA, USA), 1984.

[Clém90] Clément V., *Raisonnements cognitifs appliqués au pilotage d'algorithmes de traitement d'images,* Thèse de doctorat, Université de Nice Sofia-Antipolis, 1990.

[Clém&93] Clément V., Thonnat M. *A knowledge based approach to integration of image processing procedures,* CVGIP: image understanding, Vol. 57, No. 2, mars 1993, pp. 166-184.

[Cruy92] Cruypenninck F., *Interface de visualistaion et d'explication du raisonnement par classification d'objets complexes,* Mémoire d'ingénieur CNAM, CUEFA, Grenoble, 1992.

[Dekk92] Dekker L., *Les tâches génériques selon Chandrasekaran,* Laboratoire d'Informatique Fondamentale de Lille, Publication ERA-92-116, 01.06.92.

[Delo92] Delouis I., *Une architecture pour la représentation formelle et la simulation des modèles conceptuels : application au développement d'un système d'aide,* 1ères rencontres jeunes chercheurs en IA - septembre 92, Rennes, pp. 143 - 156.

[Delo&92] Delouis I., Krivine J.-P., *Opérationnalisation du modèle conceptuel : vers une architecture permettant une meilleure coopération système-utilisateur,* actes de la 12ème conférence internationale sur intelligence artificielle, systèmes experts et langage naturelle, Avignon, juin 1992, pp. 165 - 176.

[Delo93] Delouis I., *LISA : un langage réflexif pour la modélisation du contrôle dans les systèmes à bases de connaissances. Application à la planification des réseaux électriques,* thèse de doctorat, Université de Paris Sud, Orsay, juin 1993.

[Erns&69] Ernst G. W., Newell A., *GPS : a case study in generality and problem solving*, Academic Press, New York, 1969.

[Euze&93] Euzenat J., *Brief overview of T-TREE: the TROPES taxonomy building tool,* Proceedings of the 4th ASIS SIG/CR classification research workshop, Columbus (OH USA), octobre 1993, pp. 69 - 87.

[Ferb89] Ferber J., *Objets et agents : une étude des structures de représentation et de communications en Intelligence Artificielle,* thèse d'état, Université de Paris VI, Bordeau, juin 1989.

[Fike&71] Fikes R. E., Nilsson N. J., *STRIPS : a new approach to the application of theorem proving to problem solving*, Artificial Intelligence, Vol.. 2, 1971, pp 189-208.

[Fike&85] Fikes R., Kehler T., *The role of frame-based representation in reasoning,* Communications of the ACM, septembre 1985, Vol. 28, No. 9.

[Film88] Filman R. E., *Reasoning with worlds and truth maintenance in a knowledge-based programming environment,* Communications of the ACM, avril 88, Vol. 31 pp 382-400.

[Fisc90] Fischer G., *Communication requirements for cooperative problem solving systems,* Information Systems, Vol. 15, No. 1, 1990 pp 21-36.

[Frie&85] Friedland P. E., Iwasaki Y., *The concept and Iimplementation of skeletal plans,* Journal of Automated Reasoning, Vol. 1, No. 2, 1985, pp 161-208.

[Gens&92] Gensel J., Girard P., *Expression d'un modèle de tâches à l'aide d'une représentation par objets,* actes des 1ères journées représentations par objets (RPO), La Grande Motte, 1992, pp 225-236.

[Gree&92] Greef H. P., Breuker J. A., *Analysing system-user cooperation in KADS,* Knowledge Acquisition (1992), 4, pp. 89-108.

[Griv92] Grivaud S., *Navigation dans une base de connaissances à objets,* Mémoire d'ingénieur CNAM, CUEFA, Grenoble, 1992.

[Gutk&91] Gutknecht M., Pfeifer R., Stolze M., *Cooperative hybrid systems,* Proceedings of the 12th IJCAI, Sydney, Australie, août 91, 1991, pp 824-829.

[Hato&91] Haton J.-P. et al., *Le raisonnement en intelligence artificielle,* Collection iia, InterEditions, Paris, 1991.

[Hick&88] Hickman F.R., Killin J.L., Land L., Mulhall T., Porter D., Taylor R.M., *Analysis for knowledge-based systems - a practical guide to the KADS methodology,* Ellis Horwood Books in Information Technology, 1989.

[Horn91] Horn W., *The challenge of deep models, inference structures and abstract tasks,* Applied Artificial Intelligence, Vol. 5, pp 87-96, 1991.

[Jean&91] Jean-Marie F., Rousseau B., Rechenmann F., Prévosto M., *Embedding knowledge whithin scientific computing codes. An application to offshore structures vibration analysis,* EUROCAIPEP, Rueil-Malmaison, septembre 1991.

[Jean92] Jean-Marie F., *Une interface coopérative pour l'aide à la résolution de problèmes,* Ergo'IA, Biarritz, octobre 1992.

[John&91] Johnson P., Johnson H., *Knowledge analysis of tasks : task analysis and specification for human-computer systems,* dans A. Downton (éd.), Engineering the Human-Computer Interface, Mc Graw Hill, 1980, pp 119-144.

[Kant88] Kant E., *Interactive problem solving,* IEEE Expert, 1988, Vol. 3, No. 4, pp 36-49.

[Kaut&86] Kautz H., Allen J. F., *Generalized plan recognition,* AAAI : 5th national conference on AI, Philadelphia , Pennsylvania, 1986,pp 32-37.

[Kaut87] Kautz H., *A formal theory of plan recognition,* Technical Report 215, Department of Computer Science, University of Rochester, Mai 1987.

[Lâas&89] Lâasri H., Maître B., *Organisation du contrôle dans les architectures de blackboard,* Revue d'Intelligence Artificielle, Vol. 3, janvier 1989, pp. 105--146.

[Lemk&90] Lemke A.C., Fischer G., *Planning and execution of tasks in cooperative work environments,* 5th Conference on AI Applications CAIA, Miami (FL) 1990, pp 256-261.

[Lin&91] Lin D., Goebel R., *A message passing algorithm for plan recognition,* IJCAI, Sydney, Australie, 1991, pp 280-285.

[Mari93] Mariño O., *Classification dans les systèmes de représentation par objets,* thèse de doctorat, Université Joseph Fourrier, Grenoble, octobre 1993.

[Masi&89] Masini G., Napoli A., Colnet D., Léonard D., Tombre K., *Les langages à objets,* InterEditions, Paris, 1989.

[Médi&93] Médigue C., Willamowski J., Schmeltzer O., Uvietta P., Rechenmann F., Chevenet F., Perrière G., Gautier C., *Modeling tasks for problem solving in molecular biology,* Workshop "Artificial Intelligence and Genome", IJCAI, Chambéry, 1993, pp. 67-76

[Mora81] Moran T.P., *The command language grammar: a representation for the user interface of interactive computer systems,* International Journal of Man-Machine Studies, 1981, Vol.15, pp 3-50.

[Naul&91] Nault J.-P., Schmeltzer O., *La reconnaissance d'intentions par appariement d'objets,* Rapport de Projet (Année Spéciale Intelligence Artificielle), Grenoble, 1991.

[Newe&63] Newell A., Simon H. A., *GPS : a program that simulates human thought*, Computers and Thought, New York, McGraw-Hill, 1963.

[Newe82] Newell A., *The knowledge level*, Artificial Intelligence No. 18, Vol. 1, 1982, pp. 87-127.

[Payn&86] Payne S. J., Green T. R., *Task-action grammars: a model of the mental representation of task languages,* Human Computer Interaction Vol. 2, 1986, pp 93-133.

[Poly65] Polya G., *Comment poser et résoudre un problème,* Dunod, Paris, 1965.

[Ponc&91] Poncabaré T., Rechenmann F., *SCAI : un environnement de développement de systèmes à bases de connaissaances en calcul scientifique et technique,* actes du congrès Convention IA 91, Hermes, Paris, pp. 491-509.

[Prév82] Prévosto M., *Algorithmes d'identification des caractéristiques vibratoires de structures mécaniques complexes*, Thèse de docteur-ingénieur, Université de Rennes 1, mars 1982.

[Prév&91] Prévosto M., Rechenmann F., *SAID: a knowledge-based system in signal processing*, actes de la conférence européenne d'automatique, Grenoble, juillet 1991.

[Rech&90] Rechenmann F., Fontanille P., Uvietta P., *Shirka, système de gestion de bases de connaissances centrées-objet*, manuel d'utilisation, INRIA Rhône-Alpes, 1990

[Rech&92] Rechenmann F., Rousseau B., *A development shell for knowledge-based systems in scientific computing*, dans Houstis E.N., Rice J.R. (éds), Expert Systems for Numerical Computing, Elsevier science Publishers, New-York (NY, USA), 1992, pp 157-173.

[Regn90] Regnier P., *Planification : historique; principles, problèmes et méthodes (de GPS à ABTWEAK),* Rapport IRIT No. 90/22/R, 1990

[Rous88] Rousseau B.,*Vers un environnement de résolution de problèmes en biométrie,* thèse de doctorat, Université Claude Bernard Lyon I, 1988.

[Sace74] Sacerdoti.E. D., *Planning in a hierarchy of abstraction spaces*, Artificial Intelligence, Vol. 5, No. 2 , 1974, pp. 115-135.

[Sace75] Sacerdoti.E. D., *A structure for plans and behavior*, Elsevier science Publishers, New-York (NY, USA), 1977 (publication originale 1975).

[Sapo90] Saporta G., *Probalités, analyse des données et statistiques,* Editions Technid, 1990.

[Scap&89] Scapin D. L., Pierret-Goldbreich C., *Towards a method for task description : MAD,* Deuxième Conférence scientifiaque internationale sur le travail à l'écran de Visualisation, Montréal (Canada), septembre 89.

[Schr&88] Schreiber G., Breuker J., Bredeweg B., Wielinga B., *Modelling in KBS development,* Second European Knowledge Acquisition Workshop EKAW'88, juin 88, Bonn, R.F.A.

[Sebi88] Sebillotte S., *Hierachical planning as method for task analysis : the example of office task analysis,* Behaviour and information technology, Vol. 7, No. 3, 1988, pp 275-293.

[Stee90] Steels L., *Components of expertise*, AI Magazine, Vol. 11, No. 2, pp. 29-49.

[Stef81a] Stefik M., *Planning with constraints (Molgen : part 1),* Artificial Intelligence, Vol. 16, No. 2, 1981, pp 111-139.

[Stef81b] Stefik M., *Planning and meta-planning (Molgen : part 2),* Artificial Intelligence, Vol. 16, No. 2, 1981, pp 141-169

[Stol91] Stolze M., *Task level frameworks for cooperative expert system design,* Artificial Intelligence Communications, Vol. 4, No. 2/3, 1991, pp 98-106.

[Stol92] Stolze M., *From knowledge engineering to work-oriented development of knowledge systems,* Thèse de doctorat, Université de Zürich, Suisse, 1992.

[Taub91] Tauber M. J., *ETAG : extended task action grammar - a language for the description of the user's task language,* Human-Computer-Interaction - INTERACT '90, pp 163-168.

[Thon&93] Thonnat M., Clément V., Van den Elst J. *Supervision of perception tasks for autonomous systems : the OCAPI apporach,* rapport de recherche INRIA N° 2000, INRIA, Sophia-Antipolis, juin 1993.

[Uvie&91] Uvietta, P., Willamowski J., *Modélisation de connaissances en biologie moléculaire,* 8éme Congrès RFIA, Lyon, 27-29 novembre 1991, Vol. 3, pp 1067-1078.

[Uvie&93] Uvietta P., Willamowski J., Ziébelin D., *Task based modelling for problem solving strategies*, Proceedings of the 5th IEEE international conference on Tools with Artificial Intelligence, Boston (USA), novembre 93, pp. 123-126.

[Wile&86] Wilenga B. J., Breuker J. A. *Models of expertise*, ECAI, 1986, pp306-318.

[Will91] Willamowski J., *Modélisation de la démarche du bio-généticien*, Les cahiers IMABIO, n° 2, "Informatique et Génomes - Traitement de l'information des séquences biologiques", CNRS, octobre 1991.

[Will&93] Willamowski J., Chevenet F., *Modeling methodological knowledge in data analysis*, actes de la 13ème conférence internationale sur intelligence artificielle, systèmes experts et langage naturelle, Avignon, mai 1993, Vol. 1, pp. 211-220.

[Will94] Willamowski J., *Modélisation de tâches pour la résolution de problèmes en coopération système-utilisateur,* 9ème Congrès RFIA, Paris, janvier 1994, Vol. 2, pp. 305-316.

[Wils&88] Wilson M. D., Barnard P. J., Green T. R. G., MacLean A., *Knowledge-based task analysis for human-computer systems,* in G. C. van der Veer, T. R. G. Green, J.-M. Hoc, D. M. Murray, "Working with computers : Theory versus Outcome", pp 47-87.

Table des matières

Introduction 1

Résolution de problèmes 1

Résolution coopérative de problèmes 1

Structure du document 3

1. Modélisation de la résolution coopérative de problèmes 4

1.1. Niveau de communication compréhensible et manipulable par l'utilisateur 6

1.1.1. Communication dans les systèmes à base de connaissances 6

1.1.1.1. Modélisation fonctionnelle 8

1.1.1.2. Modélisation conceptuelle 10

1.1.2. Traduction des intentions d'un utilisateur en interaction avec la machine 12

1.1.3. Notion de tâche 14

1.2. La tâche : un niveau de description exploitable de façon automatique 15

1.2.1. Tâche et planification 15

1.2.1.1. Différents types de planification 16

1.2.1.2. Tâches et planification hiérarchique 18

1.2.2. Tâche exécutable 19

1.2.2.1. Comment représenter les connaissances 20

1.2.2.2. Comment gérer le processus de résolution 23

1.2.3. La tâche : un niveau de communication compréhensible pour l'utilisateur et pour le système 24

1.3. Modélisation de la coopération système-utilisateur 25

1.3.1. Intérêt de la coopération 26

1.3.2. La coopération du point de vue du système 27

1.3.2.1. Attribution fixe de responsabilités à l'utilisateur 27

1.3.2.2. Attribution dynamique de responsabilités à l'utilisateur 29

1.3.3. La coopération du point de vue de l'utilisateur 30

1.4. Synthèse 31

2. SCARP - représentation des connaissances ……33
2.1. Modèle à objets ……33
2.1.1. Notions de base……34
2.1.2. Organisation en hiérarchies……34
2.1.3. Classement d'instances ……35
2.1.4. Classement d'instances complexes ……36
2.1.5. Spécialisation d'instances……36
2.1.6. Méta-niveau ……37
2.1.7. Intérêt du modèle à objets……37
2.2. Eléments d'une base de connaissances ……38
2.2.1. Connaissances de résolution de problèmes……38
2.2.2. Eléments d'une base de connaissances SCARP ……39
2.2.2.1. Tâches……40
2.2.2.2. Méthodes……41
2.2.2.3. Entités ……43
2.2.2.4. Nécessité d'étendre le modèle à objets ……43
2.3. Modèle de tâches ……43
2.3.1. Distinction tâches abstraites - tâches exécutables ……44
2.3.2. Définition d'une tâche……44
2.3.3. Description de la stratégie de résolution……48
2.3.4. Coopération avec l'utilisateur ……50
2.3.5. Indépendance du contexte d'exécution……51
2.4. Contexte d'exécution d'une tâche ……51
2.4.1. Définition d'un contexte ……52
2.4.2. Description du flot de données……54
2.4.3. Différents types de contrôle sur une tâche ……55
2.4.4. Coopération avec l'utilisateur ……56
2.4.4.1. Contexte visualisation ……56
2.4.4.2. Contexte demande ……56

2.4.4.3. Contexte essai-erreur 57
2.4.5. Signification de la notion de contexte 58
2.5. Bilan 58
3. SCARP - résolution de problèmes 61
3.1. Moteur de tâches 62
3.1.1. Gestion de la résolution de problèmes 63
3.1.1.1. Différentes phases de résolution de problèmes 63
3.1.1.2. Contrôle des différentes phases par l'utilisateur 65
3.1.1.3. Gestion des échecs 65
3.1.1.4. Bilan 67
3.1.2. Réalisation d'une planification hiérarchique 67
3.1.2.1. Phase de spécialisation 67
3.1.2.2. Phase de décomposition 69
3.1.2.2. Alternance des phases de spécialisation et de décomposition 70
3.1.3. Séparation de la gestion de la résolution et de l'interaction 71
3.2. Gestionnaire de dialogues 72
3.2.1. Interactions nécessaires pour la résolution 72
3.2.2. Modification de décisions par l'utilisateur 73
3.2.2.1. Décisions modifiables 73
3.2.2.2. Protocole d'interaction 74
3.2.2.3. Versions du processus de résolution 74
3.2.2.4. Versions globales et versions locales 76
3.2.2.5. Création de versions par le gestionnaire de dialogues 76
3.2.3. Exploration de nouvelles tâches 77
3.2.3.1. Protocole d'interaction de base 77
3.2.3.2. Intégration des tâches dans la base de connaissances 78
3.2.4. Gestion globale des interactions 79
3.3. Conception et fonctionnement de l'interface 80
3.3.1. Réalisation des interactions ponctuelles 81
3.3.1.1. Choix d'un problème 82
3.3.1.2. Spécialisation 83
3.3.2. Gestion de la résolution par l'utilisateur 84

3.3.2.1. Evaluation de l'état actuel de résolution 86
3.3.2.2. Examen et remise en cause de décisions 86
3.3.2.3. Navigation entre versions....87
3.3.2.4. Exploration de nouvelles tâches....88
3.4. Bilan 88
4. Développement et utilisation d'applications....90
4.1. Analyser un domaine pour développer une application SCARP....90
4.1.1. Analyse descendante d'un domaine d'application....91
4.1.1.1. Etapes de l'analyse....91
4.1.1.2. Analyse d'une tâche 92
4.1.2. Exemple d'une analyse : l'application IFREMER 94
4.1.2.1. Description du domaine....94
4.1.2.2. Analyse du domaine et représentation des connaissances....95
4.1.3. Aspects importants de l'analyse....101
4.2. Résolution en coopération....101
4.2.1. Possibilités de coopération....101
4.2.2. Exemple de coopération : l'application SLOT....102
4.2.2.1. Description du domaine....102
4.2.2.2. Extraits de la base de connaissances....103
4.2.2.3. Exemple d'une session....105
4.3. Bilan 109
Conclusion....110
Bibliographie 113