Institut National Polytechnique de Grenoble
ENSIMAG

Université Joseph Fourier
U.F.R
Informatique & Mathématiques Appliquées

I.M.A.G.

DEA D'INFORMATIQUE

Projet présenté par :

Isabelle CRAMPÉ

SÉMANTIQUE DES TÂCHES DÉCOMPOSABLES ET SPÉCIALISABLES

Effectué au Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle
Projet SHERPA

Date : 21 Juin 1994

Jury : Mathias CHAILLOT
Jérôme EUZENAT
Yves LEDRU
Brigitte PLATEAU

Sommaire

Sommaire **1**
Introduction **1**
I. Résolution de problèmes **2**
A. Historique 2
B. Cadre du projet 8
C. Objectif du projet et motivations 13
II. Sémantique opérationnelle d'un système de tâches **15**
A. Présentation générale du système 15
B. Le système simplifié 17
C. Les flots de données 20
D. La vérification de l'applicabilité 23
E. L'implémentation 24
F. Terminaison et complexité 32
G. Esquisse de sémantique dénotationnelle 32
H. Conclusion 34
III. Tâche et rationalité **36**
A. La rationalité 36
B. L'utilité et les tâches 42
C. Utilisation statique de la rationalité 47
D. Utilisation dynamique de la rationalité 55
E. Conclusion 64
Conclusion et perspectives **67**
Références **69**
Annexes **73**
Le programme 74
La base de faits 79
Un exemple complet 81
Table des matières **85**

Introduction

Ce projet s'inscrit dans le cadre d'un DEA de Programmation et d'Intelligence Artificielle réalisé au Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle, un laboratoire de l'Institut Mathématiques Appliquées de Grenoble. Plus particulièrement, il est effectué sous la responsabilité de Jérôme Euzenat dans le cadre du projet Sherpa dirigé par François Rechenmann. Le projet Sherpa est un projet commun à l'IMAG et à l'Institut National de Recherche en Informatique et en Automatique. Les principaux travaux de cette équipe portent sur la représentation de connaissance et le raisonnement.

Dans le domaine de la résolution de problèmes, une certaine organisation des connaissances est de plus en plus usitée : la représentation et la résolution par tâches [Will94, Chai93, Clém90]. Dans de tels systèmes, les problèmes sont représentés par des tâches qui peuvent se résoudre soit à l'aide de méthodes algorithmiques disponibles, soit par l'assemblage de tâches plus élémentaires. D'un autre côté, certaines recherches actuelles s'orientent vers la rationalité qui consiste à résoudre au mieux un problème en fonction des ressources disponibles. Mathias Chaillot [Chai93] a exprimé un mécanisme rationnel de résolution de problèmes par tâches en se limitant à la sélection de méthodes. On se propose ici de fournir une sémantique opérationnelle à la notion de tâche en exprimant leur mécanisme de résolution et on tentera d'introduire la rationalité dans une telle caractérisation des tâches. Ce travail étendra donc en particulier celui de Mathias Chaillot, puisque la rationalité ne sera plus restreinte à la seule sélection de méthodes, tout en recouvrant les possibilités de résolution de tâches décomposables et spécialisables utilisées par Jutta Willamowski [Will94].

Dans un premier chapitre, nous donnerons un aperçu général des travaux concernant la résolution de problèmes puis nous situerons le projet dans un contexte plus précis en explicitant les notions spécifiques qui seront utilisées ainsi que les deux systèmes déjà réalisés qui serviront de support à notre développement. Après avoir défini plus précisément l'objectif du projet, nous nous pencherons dans le deuxième chapitre sur une première phase du travail effectué : la sémantique opérationnelle d'un système de tâches composé de quelques règles simples, illustré par un exemple repris tout au long du document. L'implémentation sera réalisée grâce à un méta-interprète Prolog, ce qui consiste à utiliser les fonctionnalités du langage Prolog, tout en contrôlant le déroulement de la résolution. Cette première étape effectuée, nous étudierons différentes facettes de la rationalité, suivant les domaines et les auteurs, ce qui fera l'objet du troisième chapitre; nous pourrons alors enfin voir les moyens permettant de rendre le système rationnel et l'implémentation qui en découle. Nous indiquerons entre crochets les références bibliographiques, rassemblées en fin de document.

I. Résolution de problèmes

La résolution d'un problème consiste généralement à choisir des actions élémentaires appropriées parmi un ensemble d'actions possibles et à les organiser de manière adéquate. Un environnement de résolution de problèmes (abrégé dans la suite en ERP) s'attache donc à trouver une solution pour chaque problème posé, solution sous la forme d'une suite d'actions articulées. Généralement, il existe différentes manières de trouver la (ou les) solution(s). La modélisation et la résolution du problème doivent donc tenir compte de cette diversité de moyens (et diversité de solutions très souvent). La notion de *tâche* est particulièrement bien adaptée à la représentation d'un problème mais elle souffre d'une non homogénéité de définition et d'utilisation suivant les systèmes ou les auteurs. Pour la résolution proprement dite d'un problème, il faut se tourner vers le mécanisme de planification.

Dans une première section, nous présentons le foisonnement d'idées autour de la notion de tâche et de ses applications. Pour l'élaboration du projet, nous nous limitons à deux approches, des tâches et des notions adjacentes, que nous détaillons dans la section B. Nous conclurons ce chapitre de description de l'état de l'art en exposant les objectifs du projet (section C).

A. Historique

En informatique, les tâches sont généralement associées à des processus qui traitent les événements dans des systèmes en temps-réel. Dans les systèmes à base de connaissances, les tâches permettent de modéliser les connaissances qui contrôlent la résolution de problèmes et surtout de fournir des explications ou d'acquérir des connaissances. La notion de tâche est employée avec diverses significations que nous détaillons dans le paragraphe 1 et est utilisable en particulier dans les ERP (paragraphe 2). Nous verrons alors en quoi consiste la planification (paragraphe 3) et sous quelle forme particulière elle peut être appliquée aux tâches afin de les rendre exécutables (paragraphe 4).

1 La notion de tâche

Chandrasekaran définit le concept de «tâche générique», qui correspond à un type de problème de base en organisant et représentant les connaissances nécessaires pour le résoudre. Elle est générique puisqu'elle ne présuppose pas de son domaine d'application et qu'elle peut apparaître dans la résolution de problèmes plus complexes. Cela permet, selon Chandrasekaran, d'avoir une représentation des connaissances adaptée à la tâche précise et donc de pouvoir donner des explications en termes appropriés.

D'autre part, des méthodologies ont été développées afin d'acquérir des connaissances lors de la création et du développement d'un système à base de connaissances: KADS ("Knowledge Acquisition and Documentation Structuring"), en particulier. La modélisation est basée sur la distinction de différents types génériques de connaissances selon leur rôle dans la résolution de problèmes. Les connaissances sont donc structurées en quatre niveaux :

— le niveau *domaine* qui contient les connaissances statiques du domaine (concepts, propriétés, relations).

— le niveau *inférence* qui décrit les compétences fonctionnelles permettant de gérer les pas élémentaires d'inférence.

— le niveau *tâche* qui fournit les décompositions des tâches en sous-tâches pour contrôler les inférences afin de réaliser un but en appliquant l'inférence.

— le niveau *stratégie* qui définit le contrôle associé aux tâches : il indique quelles sont les tâches à réaliser pour résoudre un problème en fonction du contexte.

En fait, seule la première couche contient des informations relatives au domaine; les autres en sont relativement indépendantes. La définition de tâche dans ce système est particulièrement intéressante vu qu'elle organise et ordonne l'exécution des inférences pour atteindre des buts.

De façon générale, on peut donc dire qu'une tâche décrit une catégorie de problèmes en donnant leur caractérisation et des moyens de résolution. Voyons comment les utiliser dans un ERP.

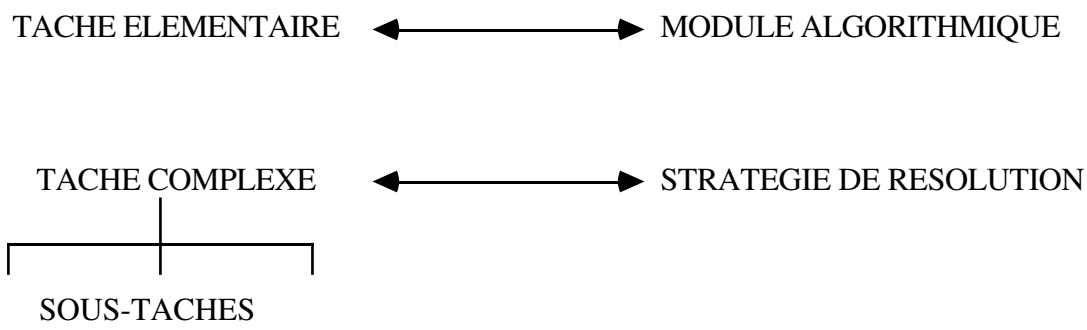
2 Les tâches et les ERP

Bertrand Rousseau [Rous88] est l'un des premiers à introduire la notion d'environnement de résolution de problèmes[1]. Il a mis en évidence les améliorations considérables qu'apporte un ERP par rapport à une bibliothèque de programmes qui n'est pas un outil adapté à la résolution de problèmes; en effet, c'est à l'utilisateur de choisir l'enchaînement des méthodes, de vérifier que les conditions d'exécution sont remplies, de tenir compte des caractéristiques du programme qu'il emploie. Il semble cependant possible de s'orienter vers un processus superviseur déchargeant l'utilisateur de ces tâches en permettant de gérer de façon autonome les exécutions complexes.

Il a proposé une modélisation d'une bibliothèque de programmes à l'aide des tâches : les modules sont partitionnés suivant le traitement qu'ils réalisent et sélectionnés à l'aide d'un mécanisme de classification; une classe de cette partition constitue alors une tâche dite élémentaire, puisque le type de problèmes qu'elle représente peut se résoudre immédiatement par l'application d'un module, sans nécessiter de décomposition. Au contraire, les tâches complexes, c'est-à-dire qui ne peuvent se résoudre directement par un module, correspondent à

[1]Sa thèse se plaçait principalement dans le domaine de la biométrie, mais ses résultats sont généraux.

une stratégie de résolution et le problème correspondant est résolu grâce à une décomposition en sous-tâches, qui elles-mêmes sont élémentaires ou non. Cette correspondance est reprise dans le schéma ci-dessous :



Signification des tâches élémentaires et complexes

La planification, détaillée dans le paragraphe suivant, définit des plans d'actions — qui sont des séquences prédéfinies de tâches élémentaires — permettant de résoudre des tâches complexes. Plusieurs méthodes de résolution sont alors possibles pour une tâche : une méthode directe ou un plan (ces deux types de méthodes pouvant exister pour une même tâche) et il faut choisir dynamiquement parmi cet ensemble celle qui va être appliquée.

3 La planification

Tout ce qui est décrit ci-dessous, sauf indication contraire, est très largement inspiré de la synthèse de Pierre Regnier [Regn90] sur le problème de la planification.

a. Origine et définition

La planification consiste à concevoir un plan d'actions pour atteindre un but à partir d'un état initial, ou, dit de manière plus générale, pour résoudre un problème. Le premier domaine dans lequel elle est apparue est la robotique, domaine dans lequel les problèmes les plus simplifiés (et adaptés à la planification) consistent à faire agir un robot dans le «monde des blocs» afin de lui faire atteindre un but (une certaine disposition des cubes) à partir d'une situation initiale fixée (une autre disposition des cubes) [Nils82]. Pour ce faire, le robot dispose d'un certain nombre d'actions primitives (`empiler, ramasser, poser, …`) et il s'agit donc de synthétiser une séquence d'actions (ou plan) qui permette de passer de l'état initial à l'état final; les états étant décrits par des prédicats tels que `mainvide, tenu(A), sur(A,B), ….` Le système STRIPS ('STanford Research Institute Problem Solver'), au début des années 70, permet de planifier les actions d'un tel robot et permet de décrire son environnement de façon moins lourde que le tout premier système, GPS ('General Problem Solver'), qui s'est attaché à la résolution de problèmes.

Précisons la définition de la planification : à partir d'une représentation initiale du monde, d'un ensemble d'opérateurs de changement d'état et de la définition du but à atteindre, elle donne les moyens de trouver un ensemble d'actions — articulées par des opérateurs (de choix, d'exécution en parallèle ou en séquence, …) — à appliquer sur le monde pour le faire passer de l'état initial à un état qui satisfait le but. Un planificateur est généralement construit à partir d'un système de règles de production (ou plus rarement à l'aide d'un démonstrateur de théorèmes) dans lequel la base de faits décrits l'état du monde et la base de règles l'ensemble des opérations possibles. Les opérations peuvent être primitives (c'est-à-dire directement exécutables) ou complexes (elles nécessitent alors une décomposition en opérations plus simples). La planification permet ainsi de trouver les moyens de résoudre un problème par une suite de méthodes; en effet, on peut assimiler l'état initial aux paramètres d'entrée du problème, l'état final aux paramètres de sortie (et aux contraintes qui leur sont imposées) et les actions élémentaires aux méthodes algorithmiques.

b. La recherche de solutions

La recherche de solutions en planification est de différents types, suivant l'espace dans lequel on effectue la recherche : un espace d'états, de sous-problèmes ou de plans partiels. Dans un espace d'états, on développe un graphe d'états afin d'obtenir une séquence d'arcs reliant l'état initial au but; cette recherche peut se faire en avant (on construit le graphe à partir de l'état initial, jusqu'à ce que l'une des feuilles s'identifie avec le but), en arrière (idem en partant du but) ou être bidirectionnelle (on développe deux graphes l'un à partir du but, l'autre à partir de l'état initial et on recherche une feuille commune); cela conduit à une explosion combinatoire du nombre d'états explorés que l'on peut cependant limiter par des heuristiques. La recherche de solutions se base donc dans ce cas sur la seule connaissance des méthodes disponibles; rien n'est connu sur la résolution possible de tâches complexes.

Dans un espace de sous-problèmes, on connait différentes règles de décompositions de but en sous-buts. Pour résoudre un problème, on applique donc ces règles afin de le décomposer en sous-buts, plus simples, qui seront résolus récursivement de la même manière, jusqu'à obtenir un plan ne contenant que des méthodes élémentaires, c'est-à-dire directement applicables.

La recherche de solution peut également s'effectuer dans un espace de plans partiels connus, c'est-à-dire qui fournissent la solution de certains problèmes; ces plans sont partiels car, par exemple, certaines variables ne sont pas valuées ou l'ordre d'exécution n'est pas complètement spécifié. La planification par recherche dans un espace de plans est de deux sortes particulièrement utilisées et intéressantes : non-linéaire ou hiérarchique. La planification non-linéaire se base sur la stratégie de l'engagement minimum, c'est-à-dire que le problème est décomposé en sous-problèmes a priori non ordonnés et sur lesquels on impose par la suite du traitement le minimum de contraintes; le plan ainsi obtenu est généralement partiellement ordonné ce qui permet l'exécution en parallèle de certaines parties. NOAH ('Nets Of Action Hierarchies') fut le premier planificateur non-linéaire (1975); il satisfait les contraintes

successivement pour permettre un développement hiérarchique du plan. La planification hiérarchique mérite une explication détaillée donnée dans le paragraphe suivant.

c. La planification hiérarchique

La planification hiérarchique, construit le plan final par affinement successif d'un plan initial suivant différents niveaux de détails, c'est-à-dire que la première étape fournira un plan grossier dont chaque partie sera affinée par la suite, et ainsi de suite jusqu'à obtenir un plan composé uniquement d'actions exécutables. Attention au fait que la planification utilisant la décomposition de buts n'est pas hiérarchique puisqu'elle ne génère pas de plans intermédiaires avec des niveaux d'abstraction différents. La hiérarchisation peut être de deux types; soit hiérarchisation des buts, comme cela a été réalisé dans ABSTRIPS ('ABstract STRIPS') où les plans successifs tiennent compte des buts d'un certain degré de difficulté, soit hiérarchisation des opérateurs, comme dans le système NOAH dans lequel les opérateurs sont définis à différents niveaux de précision, les plans successifs les affinant au fur et à mesure par des opérateurs plus concrets (seuls les opérateurs les plus précis étant directement exécutables).

d. D'autres points de vue

Il existe des méthodes efficaces autres que la planification hiérarchique. En particulier, la planification avec contraintes, implémentée dans MOLGEN ou SIPE ('System for Interactive Planning and Execution monitoring'), qui définit le plan en spécifiant de façon de plus en précise les contraintes qu'il doit satisfaire.

Des travaux plus spécifiques sur la planification s'intéressent à la planification pour des actions non déterministes [Thié93] ou à l'obtention de plans sous forme de graphes particulièrement adaptés à la réutilisation ou à l'ajout de nouvelles actions [Thié92]. De même, pour mettre en œuvre la planification, il existe des moyens moins souvent adoptés que le système à base de règles. Citons, par exemple, une utilisation de la logique linéaire [Mass92], dans laquelle toute preuve correspond à une action. Un mécanisme de preuves basé sur des règles et des axiomes permet de déterminer un plan pour passer d'un état à un autre, c'est-à-dire en particulier pour résoudre un problème.

e. Le résultat fourni

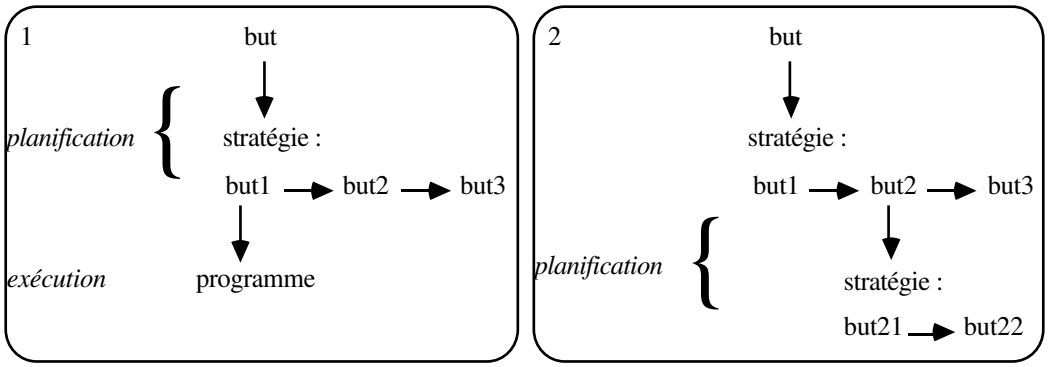
La planification fournit finalement, par application de l'une des méthodes décrites ci-dessus, un plan, c'est-à-dire un ensemble de méthodes articulées par des opérateurs d'assemblage, tels l'exécution en séquence ou en parallèle ou encore le choix. L'application d'un plan permet de résoudre une tâche donnée.

La planification hiérarchique avec niveaux d'abstraction correspond en fait à la notion de tâche; la résolution d'un problème est modélisée par la décomposition récursive d'une tâche en sous-tâches. Cependant la planification s'intéresse essentiellement à la génération de plans d'actions et pas au contrôle de la résolution, que nous désirons maîtriser.

4 Les taches exécutables

Nous présentons ici deux manières de représenter les connaissances et de gérer le processus de résolution : OCAPI [Clém90] et SCARP [Will94] (citons également le système SCAI, précurseur de SCARP qui est similaire sur le point présenté ici). OCAPI est un système développé pour gérer des algorithmes de traitement d'images — mais cependant pas spécifique à ce domaine — dans lequel les connaissances sont représentées pas des objets et des règles. Une requête spécifie le but à atteindre, ses arguments et les contraintes sur ceux-ci. A chaque but est associé un ensemble d'opérateurs qui définissent les possibilités de l'atteindre. Les règles permettent de choisir entre les différents opérateurs, d'évaluer les résultats et d'ajuster la valeur des paramètres. Il sépare donc la définition d'un problème de la manière de le résoudre. SCAI, au contraire les intègre dans une même notion : la tâche justement. Dans SCAI, les connaissances sont représentées uniquement à partir d'objets de trois types : les tâches qui décrivent un problème à résoudre et lui associent une stratégie, les méthodes qui indiquent comment gérer les programmes disponibles et les entités qui décrivent les objets manipulés dans le domaine d'application considéré.

Cependant les phases du cycle de contrôle de ces deux systèmes sont similaires : choix du but à résoudre, choix de la stratégie de résolution, puis application de cette stratégie par une décomposition en sous-buts résolus récursivement (planification hiérarchique) ou par exécution d'un programme direct. Le schéma ci-dessous permet de visualiser cette méthode de planification hiérarchique agissant par alternance de phase de planification et d'exécution :



La résolution d'un but : on choisit une stratégie pour le but courant à résoudre qui permet de le décomposer en sous-buts de plus en plus élémentaires, qui sont résolus récursivement jusqu'à obtenir un but qui se résolve par un programme que l'on exécute (1), avant de reprendre la planification au but suivant (2).

Toute cette section a été consacrée à donner une vue d'ensemble, d'une part de la notion de tâche et d'autre part de la résolution de problème grâce au mécanisme de planification. Nous allons voir plus précisément comment ces deux notions peuvent être intégrées suivant les approches de Jutta Willamowski [Will94] et Mathias Chaillot [Chai93].

B. Cadre du projet

Nous allons nous attacher à présent à décrire plus précisément les tâches qui vont être utilisées (paragraphe 1) et notamment les relations de *décomposition* et de *spécialisation* existant entre elles (paragraphe 2). Ces deux relations sont à la base d'un des deux moyens de recherche de solution à un problème donné que mous présentons dans le paragraphe 3; l'autre stratégie adopte une attitude rationnelle. Nous complèterons cette description des tâches par leur implémentation (paragraphe 4).

1 Les tâches et les méthodes

Pour modéliser un problème, nous utilisons les tâches qui peuvent se caractériser par l'association des variables d'entrée et de sortie [Will94] ou par un ensemble de modules ayant la même fonctionnalité [Chai93]. Une tâche est donc, de façon générale, la représentation d'un problème tel que l'inversion de matrice dont le paramètre d'entrée est une matrice M et le paramètre de sortie la matrice inverse M^{-1}. Pour résoudre un tel problème mathématique, il existe des moyens divers et variés plus ou moins adaptés aux paramètres d'entrée donnés.

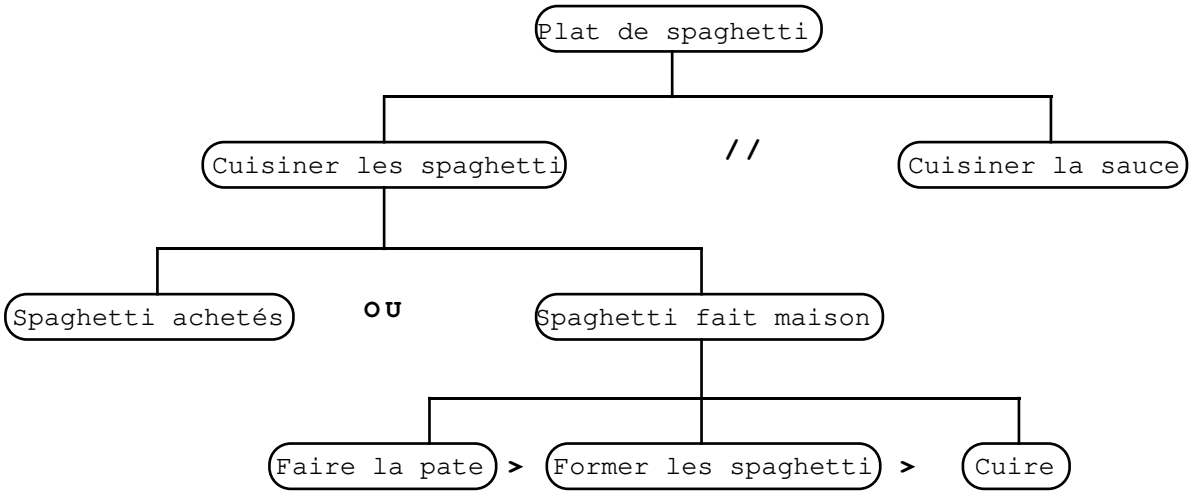
Aux tâches dites élémentaires, on peut associer un ensemble de *méthodes,* qui sont des algorithmes de résolution du problème représenté par la tâche, algorithmes applicables pour certaines valeurs des paramètres d'entrée ou de sortie. Une telle représentation des problèmes est utilisable avec des résultats très satisfaisants dans de nombreux domaines, aussi bien en analyse de données pour effectuer une Analyse en Composantes Principales [Will93] qu'en biologie moléculaire pour des problèmes d'identification de séquence d'ADN par exemple [Medi93]. Le principal atout des tâches sont les relations qui existent entre elles. Nous étudions ci-dessous la décomposition et la spécialisation.

2 Les relations entre tâches

a. La décomposition

Les tâches non élémentaires, n'ayant pas de méthode directe pour les résoudre, nécessitent une décomposition en plusieurs sous-tâches à exécuter en séquence ou en parallèle, elles-mêmes à décomposer si elles ne sont pas élémentaires. Cette décomposition en sous-tâches correspond à une décomposition d'un problème en différents sous-problèmes dont la résolution suivant un certain ordre fournit la solution du problème global.

Les recettes de cuisine permettent d'illustrer si simplement ces propos que nous vous proposons un exemple ci-dessous. Le problème consiste à préparer un plat de spaghetti, c'est-à-dire que l'on veut passer d'un état initial où l'on n'a rien à un état final qui contient le plat cuisiné. Pour cela, il faut exécuter en parallèle la préparation des spaghetti eux-mêmes et de la sauce. Pour les spaghetti, deux possibilités s'offrent au cuisinier, soit les acheter, soit les préparer. S'il souhaite les préparer, il faudra faire la pâte, former les spaghetti puis les cuire.



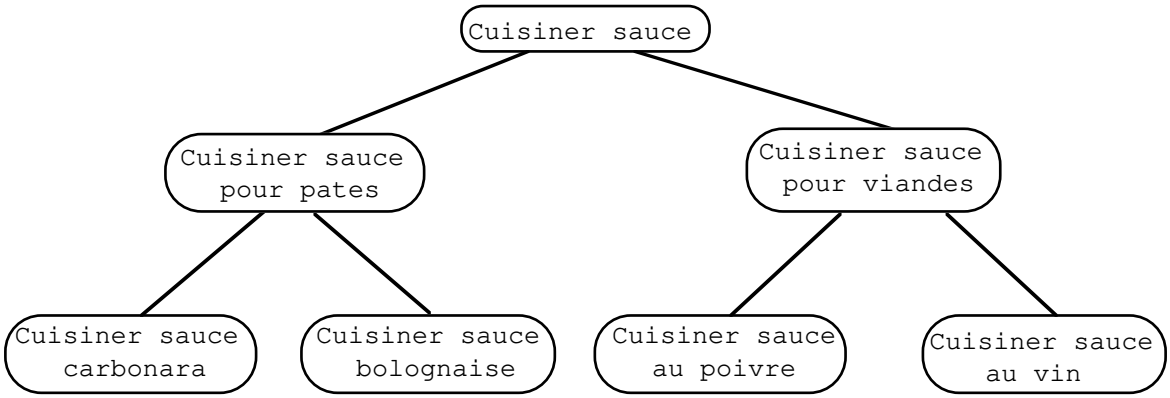
Une tâche et sa décomposition

Cet exemple permet de visualiser la façon dont une tâche peut être décomposée à l'aide de trois opérateurs : le choix (OU), l'exécution en parallèle (//) ou l'exécution en séquence (>). Les nœuds internes de l'arbre représentent des tâches qui nécessitent une décomposition (car elles sont non élémentaires) ou qui admettent une telle décomposition et une méthode; par contre, des méthodes sont obligatoirement associées aux feuilles pour que la tâche puisse être résolue. Voyons à présent la spécialisation qui est une autre relation existant entre les tâches.

b. La spécialisation

Les tâches sont organisées hiérarchiquement selon la relation de spécialisation; on dit qu'une tâche T' est plus spécifique qu'une tâche T si le domaine des valeurs des paramètres de T' est plus restreint que celui de T et si elle résout le même problème. La racine d'une hiérarchie représentera donc un problème très général, alors que les sous-tâches modélisent des cas de plus en plus concrets. Plus on est bas dans la hiérarchie, plus la définition de la tâche est précise et donc plus il est probable qu'elle ait une (ou des) méthode(s) associée(s). Ainsi, Jutta Willamowski [Will94] suppose que les tâches trop abstraites ne peuvent pas être directement résolues, seules les tâches relativement concrètes ont une méthode de résolution associée.

Prenons un exemple qui va compléter le premier. Pour préparer une sauce, il existe une hiérarchie dont la racine est le problème exprimé de la façon la plus générale; ses deux fils représentent des sous-problèmes qui ont un domaine d'application plus réduit, de même pour leurs propres fils, qui sont les feuilles de la hiérarchie.



Une hiérarchie de tâches

Dans l'exemple, l'objectif est de préparer un plat de spaghetti, il faut sélectionner une sauce qui est conforme à ce problème général; nous avons donc le choix entre «`Cuisiner une sauce carbonara`», «`Cuisiner une sauce bolognaise`» ou simplement appliquer la méthode éventuelle associée à «`Cuisiner sauce pour pâtes`».

Sur cet exemple, nous voyons qu'il est possible de trouver une méthode pour résoudre une tâche dans un contexte particulier. Ici, il fallait «Cuisiner une sauce» et vu que l'on est dans le contexte de la préparation d'un plat de spaghetti, seules les méthodes qui spécialisent la préparation d'une sauce pour pâtes sont possibles.

Ces arbres, de décomposition et de spécialisation, permettent d'obtenir les différentes possibilités de résoudre une tâche donnée.

3 Recherche rationnelle d'une solution

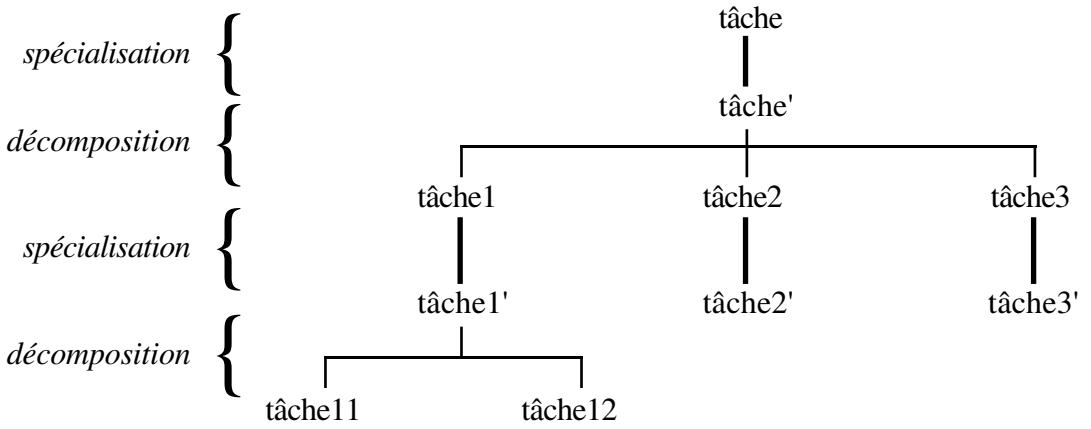
Comme il existe plusieurs moyens pour résoudre un problème, il est licite de se demander quel est celui que l'on va appliquer pour résoudre une tâche donnée. Il reste aussi surtout à déterminer la façon de rechercher une possibilité de résolution - qui est un ensemble de méthodes élémentaires articulées par des opérateurs - sans forcément construire l'ensemble des moyens possibles de manière exhaustive. Pour cela, les tâches sont particulièrement bien adaptées, et il existe différentes stratégies, dont la stratégie rationnelle à laquelle nous allons particulièrement nous attacher.

Ci-dessous sont présentées les approches de Jutta Willamowski [Will94] (en a.) et de Mathias Chaillot [Chai93] (en b.).

a. La planification hiérarchique

Les informations contenues dans les arbres de décomposition et de spécialisation permettent de trouver une solution à un problème précis posé grâce à une planification hiérarchique par niveaux d'abstraction (dont une présentation générale a été donnée en A.3.c), comme cela a été réalisé dans le système SCARP de Jutta Willamowski. La notion d'abstraction correspond,

dans le cas particulier des tâches, à la relation de spécialisation; les plans successifs sont donc construits avec des tâches de plus en plus spécialisées. Cela revient à exécuter en alternance des phases de spécialisation et de décomposition afin d'obtenir finalement un plan ne contenant que des méthodes. Les phases de spécialisation détectent la sous-tâche spécialisée la plus adaptée aux variables d'entrée du problème et les phases de décomposition décomposent la tâche en sous-tâches pour lesquelles on adoptera la même stratégie si elles ne sont pas élémentaires. Si une tâche n'est ni spécialisable, ni décomposable, elle est suffisamment précise pour admettre une méthode associée.



Résolution d'une tâche : on alterne les phases de spécialisation et de décomposition jusqu'à obtenir des tâches élémentaires qui se résolvent par des méthodes.

Ce mécanisme permet de résoudre une tâche en utilisant les hiérarchies de tâches et la relation de décomposition.

b. La solution la plus utile

Il est cependant naturel de vouloir trouver la meilleure stratégie de résolution pour un problème donné. Le terme «meilleur» est cependant ambigu et dépend des critères pris en compte. Il peut tout d'abord s'agir de trouver la solution la plus précise (ou dépassant un certain seuil de précision) ou celle dont le temps d'exécution est le plus court, c'est-à-dire que l'on considère uniquement des qualités propres de la solution, sans se préoccuper du coût de sa recherche. Mais, pour toute application dont les ressources sont limitées, il convient de chercher les solutions de manière rationnelle. Le principe de la rationalité est de tenir compte du coût de la recherche de la solution et de son utilité intrinsèque : la meilleure solution, en terme de rationalité est donc celle qui maximise une fonction d'utilité, fonction de la valeur que l'on accorde à la solution et de la perte de ressources occasionnée par sa recherche.

Lors de la résolution d'une tâche, il faut donc voir jusqu'à quel point il est «utile» de rechercher une solution plus décomposée ou plus spécialisée; la notion d'utilité permet de quantifier les ressources consommées pour rechercher une meilleure solution et le gain supplémentaire apporté par cette solution, les ressources prises en compte pouvant être de

nature diverse. La ressource primordiale est le plus souvent le temps, en particulier pour les applications en «temps réel». Il s'agit alors de trouver la meilleure solution non seulement au vu de ses qualités propres (temps d'exécution, précision, ...), mais également du temps nécessaire pour la trouver et la choisir. La fonction d'utilité dépend alors de l'utilité intrinsèque et du temps de recherche.

Dans cette optique, le système TRAM conçu par Mathias Chaillot, qui recherche une solution par spécialisations successives, utilise la notion d'utilité pour trouver la solution rationnelle parmi celles qui sont à disposition et principalement pour déterminer s'il faut continuer à spécialiser ou s'il est préférable, toujours selon l'utilité définie, de s'arrêter en utilisant la méthode directe associée à la tâche. Cette résolution se fait sous l'hypothèse que l'ensemble des sous-tâches associées à une tâche donnée est exhaustif et exclusif (c'est-à-dire qu'il existe une et une seule sous-tâche permettant de résoudre un problème, représenté par la tâche, pour des paramètres fixés) et qu'il existe une méthode associée à chaque tâche.

Pour chaque problème (dont les données sont entièrement caractérisées) il existe donc deux solutions possibles : la méthode directe rattachée à la tâche ou la spécialisation qui s'applique dans ce cas précis. Evidemment lors de la résolution d'un problème, on ne connait pas a priori la tâche spécialisable qui convient et il faut donc la trouver dans l'ensemble des spécialisations de la tâche. Pour la détecter, il s'agirait donc de tester toutes celles qui sont disponibles jusqu'à la trouver … mais cela ne semble pas très rationnel. La recherche rationnelle de la solution consiste au contraire à ne tester que les solutions qui sont susceptibles d'apporter un gain de ressources par rapport à la méthode directe — malgré le temps qui sera perdu pour voir si elle s'applique effectivement au problème entièrement caractérisé. Les solutions qui ont un temps de test trop important par rapport au gain probable ne seront pas étudiées, la valeur de la fonction d'utilité permettant de déterminer les solutions à tester ou non. Le système TRAM définit ainsi un ordre de test sur les solutions probables, toujours suivant la valeur de la fonction d'utilité; cet ordre est rationnel parce qu'il maximise l'utilité espérée de la résolution sur l'ensemble des problèmes possibles.

Toute la problématique de la rationalité et son rapport aux tâches sont exposés plus précisément dans le chapitre III.

c. Comparaison des deux stratégies

La stratégie de SCARP qui utilise la planification hiérarchique a l'avantage de résoudre une tâche en tenant compte des relations de décomposition et de spécialisation, alors que la deuxième ne se préoccupe que des sous-tâches spécialisées, mais la résolution se fait sans aucune considération de ressources. La notion de rationalité qui est introduite dans le système TRAM permet de trouver la solution d'un problème de manière rationnelle, c'est-à-dire en tenant compte du temps de recherche de la solution et de son utilité intrinsèque. Cette solution fait cependant un certain nombre d'hypothèses restrictives (exhaustivité, exclusivité, pas de

décomposition) que nous allons essayer de relâcher, ce qui conduira à concilier les avantages de ces deux systèmes. A noter que TRAM s'inscrit dans l'approche du projet SHERPA qui tente de faire de la planification en combinant classification et décomposition.

Pour clore cette partie de description des tâches et des notions adjacentes, nous traitons ci-dessous brièvement de la représentation informatique des tâches.

4 Implémentation des tâches

Une représentation à objet est utilisée pour représenter les tâches, qui sont implémentées sous forme de classes. Le traitement des hiérarchies se fait alors grâce au mécanisme de classification qui permet de déterminer les sous-classes qui correspondent aux paramètres effectifs du problème. Un environnement de résolution de problèmes utilisant les tâches peut donc être implémenté sur un système basé sur les objets, comme SHIRKA qui sépare les mécanismes d'inférences et les objets sur lesquels ils agissent [Rech85]. C'est l'option choisie par Jutta Willamowski [Will94] notamment.

Nous avons décrit ci-dessus à l'aide d'un exemple central le cadre général du projet. Nous avons présenté différentes notions relatives aux tâches et discuté des stratégies de résolution possibles, ainsi que de l'implémentation. Voyons à présent plus précisément ce qui motive la réalisation du projet.

C. Objectif du projet et motivations

Le but du projet est d'unifier deux formalismes utilisés pour décrire la résolution de problèmes par les tâches dans les systèmes SCARP et TRAM et de donner une sémantique opérationnelle d'un tel système de résolution. Plus précisément, cette unification concerne la conception même des tâches, les possibilités de résolution et le contrôle de cette résolution.

Il s'agit donc de fournir un système permettant de montrer comment est accomplie une tâche, résolution qui peut se faire par décomposition, par spécialisation ou par application d'une méthode directe. De plus, le système doit choisir sa stratégie de manière rationnelle, c'est-à-dire qu'il doit tenir compte de la notion d'utilité. Il permettra donc de prouver en particulier l'optimalité de la résolution proposée, selon les critères d'utilité choisis.

Nous proposons de concevoir dans un premier temps un système qui permette de résoudre une tâche grâce aux informations contenues dans différents arbres de spécialisation ou de décomposition puis d'introduire l'utilité dans le choix de ses solutions afin que la résolution fournisse une solution en maximisant l'utilité espérée. Par rapport aux travaux sur lesquels nous nous appuyons, le système élargit la notion d'utilité introduite par Mathias Chaillot aux possibilités de décomposition proposées par Jutta Willamowski. Evidemment, l'unification des deux systèmes que nous allons proposer se borne à leur aspect de résolution de problèmes,

chacun d'eux ayant par ailleurs des caractéristiques intéressantes. Il sera fait totalement abstraction des connaissances du domaine; nous nous bornerons à décrire le mécanisme, ce qui ne nécessite pas un environnement de représentation par objet. Les solution proposées seront implémentées en Prolog, ce qui sera remis en cause ultérieurement.

Dans le chapitre II, nous présentons la sémantique opérationnelle d'un système qui se préoccupe de nous fournir l'ensemble des moyens permettant de résoudre une tâche donnée, en envisageant toutes les possibilités de résolution proposées par le système SCARP. Le chapitre III présente plus en détails les notions de rationalité et d'utilité ainsi que deux manières principales de compléter le système afin que la résolution soit rationnelle, conformément au système TRAM. Après une évaluation et une comparaison de ces deux méthodes, il sera temps de conclure et d'envisager les perspectives dans le dernier chapitre.

II. Sémantique opérationnelle d'un système de tâches

Afin de formaliser en quoi consiste la résolution d'une tâche, nous présentons le système de tâches proposé en procédant par étapes (étapes similaires à celles de sa conception). Après en avoir donné une vision globale (section A), nous décrivons une première ébauche simplifiée (section B), dont les lacunes seront comblées grâce à l'introduction de flots de données (section C) et d'une vérification d'applicabilité des règles (section D). Nous pourrons alors voir comment un tel système peut être implémenté (section E). Ces différents points seront illustrés grâce à l'exemple précédemment cité. Afin de compléter l'aspect sémantique du système de tâche, nous traitons de sa terminaison et de sa complexité (section F) et nous proposons une ébauche de sémantique dénotationnelle (section G) avant de conclure (section H).

A. Présentation générale du système

Nous rappelons ci-dessous les fonctionnalités souhaités du système, et nous donnons la forme des données qui vont être utilisées.

1 Son rôle

Notons tout d'abord que le rôle du système est de faire de la planification dans un espace de plans partiels qui peuvent être sous la forme d'arbres de décomposition et de spécialisation ou directement sous la forme de plans; il reste alors à rechercher s'il existe effectivement une solution au problème posé parmi celles dont on dispose. Le chapitre III s'attachera à décrire un mécanisme permettant de trouver la meilleure (selon des critères donnés).

Le système doit être capable de fournir une (ou toutes les) solution(s) pour la tâche T — si de telles solutions existent dans le plan fourni par la planification — pour les paramètres donnés. Ce système va être le plus général possible, c'est-à-dire qu'il tient compte de toutes les possibilités de résolution d'une tâche (décomposition, spécialisation et méthode directe). Nous allons tout d'abord présenter les règles simples, puis nous verrons comment les enrichir par la suite. L'exemple cité dans la première partie est repris dans cette partie, afin d'illustrer ce qui aura été présenté.

2 Les données manipulées

Afin de pouvoir manipuler aisément les informations contenues dans les arbres de décomposition et de spécialisation (qui sont disponibles par hypothèse), nous les transcrivons

en un ensemble de données de la forme *T* **<=** *T'* (pour les décompositions et les méthodes directes) ou *T'* **<<** *T* (pour les spécialisations).

T **<=** *T'* signifie que *T'* est une stratégie pour résoudre *T* (le moyen de résoudre *T'* permet également de résoudre *T*); cette stratégie peut être une méthode directe ou une tâche composée, c'est-à-dire un ensemble de tâches à exécuter en séquence ou en parallèle, représentés respectivement grâce aux opérateurs **<** ou **//**. On aura ainsi par exemple :

```
Plat_de_spaghetti <= Cuisiner_les_spaghetti // Cuisiner_Sauce
Spaghetti_faits_maisons <= Faire_la_pate < Former_les_spaghetti
                         < Cuire_les_Spaghetti
```

Pour représenter le choix entre plusieurs stratégies, il suffit de le décomposer en autant de cas qu'il y a de possibilités; ainsi :

```
Cuisiner_les_spaghetti  <= Spaghetti_achetés
Cuisiner_les_spaghetti  <= Spaghetti_faits_maisons
```

T' **<<** *T* signifie que la tâche *T'* spécialise (ou est plus spécifique que) la tâche *T*. Par exemple :

```
Cuisiner_Bolognaise << Cuisiner_Sauce_pour_Pates
```

Voilà décrites les données initiales sur lesquelles se base la résolution. On pourrait regretter de ne plus avoir une vision globale du graphe de tâches, mais il est toujours possible de considérer une représentation graphique de ces données. Voyons à présent sous quelle forme le résultat sera fourni .

Le résultat final doit être un plan ne contenant que des méthodes articulées par les opérateurs **<** et **//**, que nous prenons identiques à ceux utilisés pour les tâches pour ne pas introduire deux notations pour des opérateurs qui ne diffèrent que par la nature de leurs opérandes. Afin d'associer un plan à la tâche qu'il permet de résoudre, nous utilisons la notation :

*P***:-***T* qui signifie que l'exécution du plan *P* permet de résoudre la tâche *T*.

Remarquons que le mot «plan» ici est pris au sens d'ensemble de méthodes articulées par les opérateurs d'exécution en parallèle et en séquence et que l'on ne fait pas intervenir d'opérateur de choix.

B. Le système simplifié

Rappelons qu'une tâche peut être élémentaire, décomposable en séquence de sous-tâches, décomposable en sous-tâches à exécuter en parallèle ou spécialisable.

On notera *T, T1, …, Tn* des tâches;
mT une méthode (algorithmique) associée à la tâche *T*
P, P1, …, Pn des plans.

1 Les règles simples

Nous allons présenter ci-dessous les différentes règles de notre système en expliquant brièvement leur signification. Elles représentent une partie de la sémantique opérationnelle d'un système tel que SCARP [Will94].

Considérons une tâche *T* élémentaire, à laquelle une méthode est associée. Cette méthode constitue un plan pour *T*.

(R1) $\dfrac{T <= mT}{mT :- T}$

Considérons à présent une tâche *T* se décomposant en séquence de n sous-tâches *T1, …, Tn*. L'exécution en séquence des plans de *T1, …, Tn* constitue un plan pour *T*.

(R2) $\dfrac{P1 :- T1, \ldots, Pn :- Tn,\ T <= T1 < \ldots < Tn}{P1 < \ldots < Pn :- T}$, n entier, n ≥1

Soit une tâche *T* se décomposant en n sous-tâches *T1, …, Tn* à exécuter en parallèle. L'exécution en parallèle des plans de *T1, …, Tn* permet de résoudre *T*.

(R3) $\dfrac{P1 :- T1, \ldots, Pn :- Tn,\ T <= T1 // \ldots // Tn}{P1 // \ldots // Pn :- T}$, n entier, n >1

Soit une tâche *T* qui est spécialisable par *T1*. Si on trouve un plan qui résout le problème attaché à une tâche T1 plus spécifique que T, le problème représenté par la tâche T est également résolu (par le même plan).

(R4) $\dfrac{P :- T1,\ T1 << T}{P :- T}$

2 Quelques remarques

— On pourrait introduire un «*ou*» entre tâches pour indiquer qu'il y a un choix entre différentes stratégies de résolution mais on a choisi de laisser le mécanisme de résolution traiter cela.

— Un plan est un moyen de résoudre la tâche, contrairement à la dénomination utilisée en planification, citée en I.A.3, qui fait d'un plan l'ensemble des moyens de résoudre une tâche grâce à l'utilisation de l'opérateur «*ou*». Ceci est nécessaire vu que l'utilité sera associée à une solution particulière.

— Nous insistons sur le fait que le système, à ce stade de son évolution, est *non déterministe*; en effet, pour une tâche T donnée, il peut exister plusieurs stratégies de résolution et donc y avoir différentes règles à appliquer à un moment donné. Le système appliquera l'une des règles arbitrairement.

— Une tâche peut être considérée comme *résolue* lorsque l'on a un plan qui ne contient que des méthodes. Le mécanisme ci-dessus ne nous fournit, conformément à cette contrainte, que des plans contenant exclusivement des méthodes, d'après les règles utilisées. En effet la règle R1 est la seule qui permette d'obtenir un plan - seulement à partir de faits et non de plans déjà construits - et ces plans sont des méthodes.

3 Un exemple

Reprenons l'exemple cité dans la présentation du sujet et voyons comment réagit le système. Nous décrivons en premier lieu la représentation des connaissances contenues dans les deux arbres, de décomposition et de spécialisation.

L'arbre de décomposition :

```
Plat_de_spaghetti <= Cuisiner_les_spaghetti // Cuisiner_Sauce
Cuisiner_les_spaghetti  <= Spaghetti_achetés
Cuisiner_les_spaghetti  <= Spaghetti_faits_maisons
Spaghetti_faits_maisons <= Faire_la_pate < Former_les_spaghetti
                         < Cuire_les_Spaghetti
```

L'arbre de spécialisation :

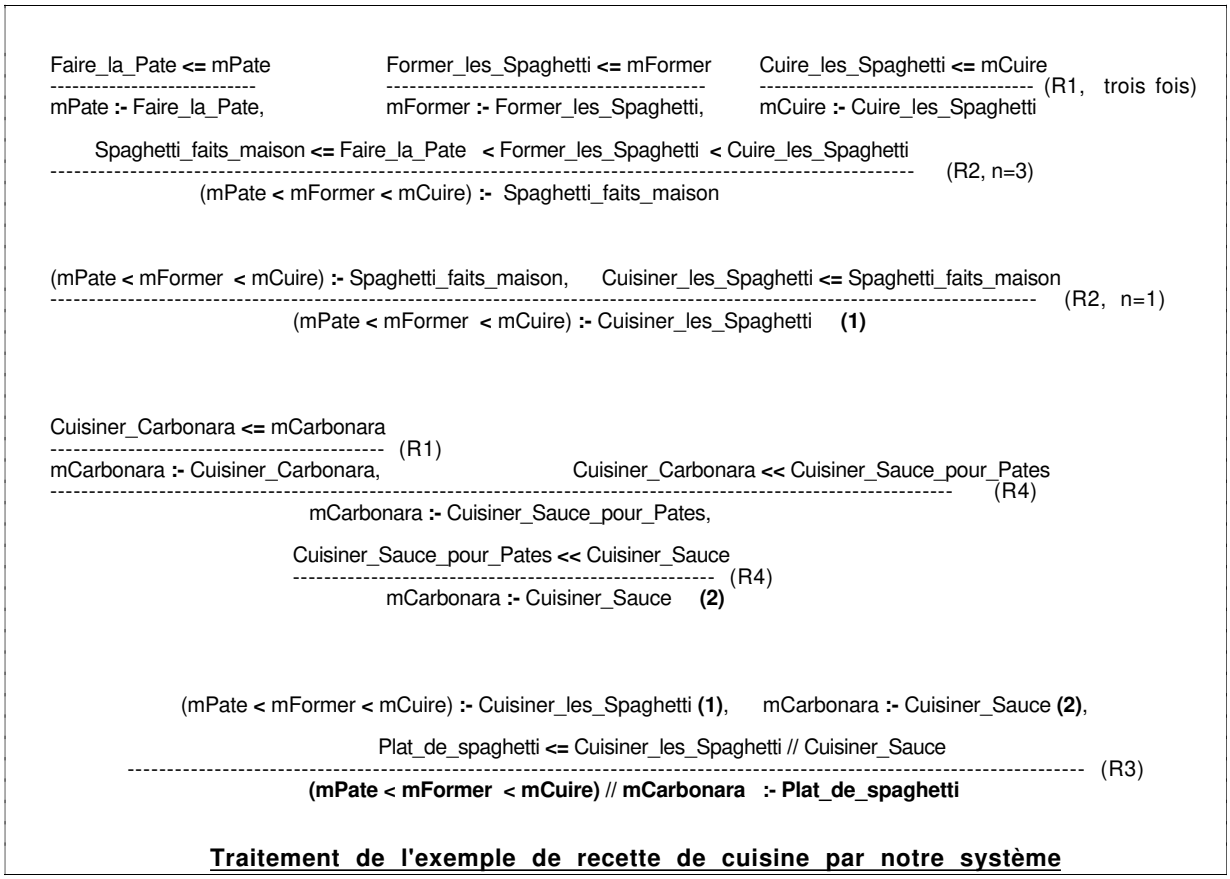
```
Cuisiner_Sauce_pour_pates << Cuisiner_Sauce
Cuisiner_Sauce_pour_viande << Cuisiner_Sauce
Cuisiner_Carbonara << Cuisiner_Sauce_pour_Pates
Cuisiner_Bolognaise << Cuisiner_Sauce_pour_Pates
Cuisiner_Sauce_au_poivre << Cuisiner_Sauce_pour_Viandes
Cuisiner_Sauce_au_vin << Cuisiner_Sauce_pour_Viandes
```

Les différentes méthodes associées aux feuilles des arbres :

```
Spaghetti_achetés <= mCuire
Cuire_les_Spaghetti <= mCuire
```

```
Faire_la_pate <= mPate
Former_les_spaghetti <= mFormer
Cuisiner_Carbonara <= mCarbonara
Cuisiner_Bolognaise <= mBolognaise
Cuisiner_Sauce_au-poivre <=  mPoivre
Cuisiner_Sauce_au-vin <= mVin
```

La figure ci-dessous montre le traitement de l'exemple pour une réponse possible du système en détaillant l'application des différentes règles.



Traitement de l'exemple de recette de cuisine par notre système

Evidemment le système doit être capable de fournir toutes les solutions possibles qui sont pour ce cas précis :

`(mPate < mFormer < mCuire) // mCarbonara` (illustrée par la figure)

`(mPate < mFormer < mCuire) // mBolognaise`

`mCuire // mCarbonara`

`mCuire // mBolognaise`

`(mPate < mFormer < mCuire) // mPoivre`

`(mPate < mFormer < mCuire) // mVin`

`mCuire // mPoivre`

`mCuire // mVin`

Cet exemple illustre le fonctionnement du système simplifié ainsi que ses lacunes. En particulier, il ne tient pas compte des paramètres des problèmes et donc il n'est pas possible de vérifier l'applicabilité des stratégies (le système autorise dans l'état actuel de choisir une «`sauce_pour_viande`» pour accompagner des spaghetti). Pour y remédier, il faut introduire des flots de données dans les règles (section C) ainsi qu'un typage qui permettra d'appliquer une décomposition ou une spécialisation seulement si les données du problème appartiennent à leur domaine d'application (section D).

C. Les flots de données

Pour le moment, les règles agissent sur des propositions. Il est cependant nécessaire d'associer des variables aux tâches pour pouvoir tenir compte des données des problèmes. Pour cela, nous associons des variables à chaque tâche et à chaque sous-tâche de leur décomposition. Il est également nécessaire d'associer des variables aux plans puisque l'application de ceux-ci nécessiteront de connaître les paramètres des problèmes qu'ils résolvent. Remarquons que les variables du plan et de la tâche qu'il permet de résoudre ne sont pas forcément identiques; elles peuvent se calculer les unes en fonction des autres par exemple.

1 Les règles avec des flots de données

On note $x, x_1, ..., x_n, x', x_1', ..., x_n', x'', x_1'', ..., x_n''$ des vecteurs de variables; leur taille varie en fonction du nombre de données nécessaires à la résolution de la tâche. On note σ une substitution de variables classique; on la généralise aux vecteurs de variables par la définition suivante :

si x_1 et x_2 sont unifiables si et seulement si :

$longueur(x_1) = longueur(x_2)$ et

$\exists\sigma$ tel que $\forall i \in [1, .., longueur(x_1)], \sigma(x_1[i]) = \sigma(x_2[i])$

et que l'on note $\sigma(x_1) = \sigma(x_2)$.

En explicitant l'unification nécessaire entre les données des tâches pour lesquelles on connait un plan et les données des sous-tâches de décomposition ou de spécialisation, les règles s'écrivent :

(R1) $\dfrac{T(x) <= mT(x')}{mT(x') :- T(x)}$

(R2) $$\frac{P_1(x_1') :- T_1(x_1),\ldots, P_n(x_n') :- T_n(x_n),\ T(x) <= T_1(x_1'') < \ldots < T_n(x_n'')}{P_1(\sigma(x_1')) < \ldots < P_n(\sigma(x_n')) :- T(\sigma(x))}$$

pour n ≥1, si ∃σ unificateur tel que ∀i∈[1, .., n], $\sigma(x_i) = \sigma(x_i'')$.

(R3) $$\frac{P_1(x_1') :- T_1(x_1),\ \ldots,\ P_n(x_n') :- T_n(x_n),\ T(x) <= T_1(x_1'') // \ldots // T_n(x_n'')}{P_1(\sigma(x_1')) // \ldots // P_n(\sigma(x_n')) :- T(\sigma(x))}$$

pour n >1, si ∃σ unificateur tel que ∀i∈[1, .., n], $\sigma(x_i) = \sigma(x_i'')$.

(R4) $$\frac{P(x_1') :- T_1(x_1),\ T_1(x_1'') << T(x)}{P(\sigma(x_1')) :- T(\sigma(x))}$$

si ∃σ unificateur tel que $\sigma(x_1) = \sigma(x_1'')$.

Remarque : x_1'' est un vecteur de variables qui regroupe les paramètres nécessaires pour l'exécution de T_1. C'est en général une partie de x_1 ou x_1 lui-même, mais on peut imaginer qu'il se calcule à partir de différents paramètres contenus dans x_1.

2 Un dernier point à préciser

Dans l'ensemble de règles précédent, on ne s'est pas préoccupé de la forme que prennent les plans. Un plan $P_1(x_1) // P_2(x_2)$ n'est pas de la forme P(x) et cela pose problème pour l'unification qui sera nécessaire lorsqu'il sera intégré dans un plan plus compliqué. On considère alors à partir d'un langage de plan propositionnel :

P = m | P//P | P<P

son extension :

P = λx. m(x)
| λx'⊕x". P(x')//P(x")
| λx'⊕x". P(x')<P(x")

, en notant ⊕ la concaténation de vecteurs.

De cette manière, $P_1(x_1) // P_2(x_2)$ dénote $(\lambda y_1 \oplus y_2.\ P_1(y_1) // P_2(y_2))\ (x_1 \oplus x_2)$ et est donc de la forme requise P(x). De même, $P_1(x_1) < P_2(x_2)$ dénote $(\lambda y_1 \oplus y_2.\ P_1(y_1) < P_2(y_2))\ (x_1 \oplus x_2)$. Ainsi tous les plans admettent une écriture de la forme P(x) et on peut donc les utiliser et leur appliquer des substitutions, en remarquant que $\sigma(x_1 \oplus x_2) = \sigma(x_1) \oplus \sigma(x_2)$.

3 Des illustrations

On peut à présent calculer les valeurs passées aux sous-tâches à partir des paramètres de la tâche :

```
Cuisiner_les_spaghetti (nb_personnes) <= Spaghetti_achetés (nb_personnes
* 100).
```

`Spaghetti_achetés` prenant comme argument la quantité, en grammes, à cuisiner.

Mais surtout, le système est capable d'unifier les tâches déjà réalisées et celles qui sont nécessaires pour la résolution de tâches plus complexes. Supposons que l'on dispose des données suivantes :

```
cuisiner_spaghetti (nb) <=
    faire_bouillir_eau (nb*0.5) < cuire_spaghetti (nb*100).
cuisiner_sauce (nb_personnes) <=
    traiteur_cuisiner_sauce (nb_personnes * 15).
faire_plat_de_spaghetti(nb) <=
    cuisiner_spaghetti(nb) // cuisiner_sauce(nb).
```

avec `cuisiner_spaghetti`, `cuisiner_sauce` et `faire_plat_de_spaghetti` qui prennent comme arguments le nombre de personnes pour qui il faut cuisiner, `faire_bouillir_eau`, la quantité d'eau et `cuire_spaghetti` la quantité de spaghetti et `traiteur_cuisiner_sauce` le prix de revient.

A la requête `faire_plat_de_spaghetti(3)`, le système, pour appliquer la règle (R2), a besoin de connaître des plans pour les deux sous-tâches de `faire_plat_de_spaghetti` et applique donc la règle (R1), deux fois, pour trouver :

```
faire_bouillir_eau (nb1*0.5) < cuire_spaghetti (nb1*100) :-
    cuisiner_spaghetti (nb1)
```

soit encore d'après le paragraphe précédent :

```
(faire_bouillir_eau < cuire_les_spaghetti) (nb1*0.5 ⊕ nb1*100) :-
    cuisiner_spaghetti (nb1)
```

et

```
traiteur_cuisiner_sauce (nb2 * 15) :-
    cuisiner_sauce (nb2)
```

En appliquant la règle (R2), il fait une première unification entre `nb` et 3 :

```
(f_bou_eau < cuire_spaghetti) (nb1*0.5 ⊕ nb1*100) :- c_spaghetti(nb1),
                  t_c_sauce(nb2*15) :- c_sauce(nb2),
      faire_plat_de_spaghetti(3) <= c_spaghetti(3) < c_sauce(3)
```

```
(f_bou_eau < cuire_spaghetti) (σ(nb1*0.5,nb1*100))
< t_c_sauce(σ(nb2)*15):- faire_plat_de_spaghetti(3)
```

avec σ = {nb1 <- 3; nb2 <- 3} (σ(3) = 3), d'où :

σ(nb1*0.5 ⊕ nb1*100) = σ(nb1)*0.5 ⊕ σ(nb1)*100

On obtient donc comme plan final (après avoir 'redistribué' les paramètres) :

```
faire_bouillir_eau(1.5)< cuire_spaghetti(300)< traiteur_cuisiner_sauce(45)
```

Grâce à l'unification introduite sur les flots de données, le système peut maintenant supporter des résolutions plus complexes et surtout plus réalistes et donc fournir des résultats qui sont satisfaisants.

D. La vérification de l'applicabilité

Il faut que le système vérifie que la sous-tâche (de spécialisation ou de décomposition), choisie pour résoudre la tâche globale, s'applique aux paramètres effectifs de la tâche donnée.

1 La règle à modifier

Nous pouvons admettre que seules les spécialisations sont des sous-tâches qui ne s'appliquent pas pour l'ensemble du domaine d'application de la tâche et donc que les sous-tâches de décomposition s'appliquent toujours. Cette hypothèse n'est pas vraiment restrictive dans le sens où l'on peut toujours ajouter une tâche factice qui soit spécialisée et qui admettent pour seule stratégie la décomposition en question. La seule règle à modifier est donc la règle (R4), les autres pouvant s'appliquer quelles que soient les valeurs des variables des tâches.

(R4) $\dfrac{P(x_1') :- T_1(x_1),\ T_1(x_1'') \prec\prec T(x)}{P(\sigma(x_1')) :- T(\sigma(x))} \qquad V_{T1}(\sigma(x_1''))$

si $\exists\sigma$ unificateur tel que $\sigma(x_1) = \sigma(x_1'')$.

$V_{T1}(x_1)$ est un prédicat qui permet de tester si la tâche T_1 s'applique pour les données x_1. Si V_{T1} est satisfait, on sait que la tâche T_1 permet effectivement de résoudre la tâche T pour les données x_1.

2 Des illustrations

Dans l'exemple traité jusqu'à présent, nous pouvons enfin tenir compte des restrictions d'application de certaines stratégies. On peut vérifier que cette tâche spécifique s'applique dans le contexte particulier du problème:

```
Cuisiner_Sauce (q,plat) >>  Cuisiner_Sauce_pour_pates (q) si plat ∈ pâte.
Preparer_repas(type,nb_personnes) >>faire_plat_de_spaghetti(nb_personnes)
si type = ordinaire et nb_personnes >= 2.
```

D'autres exemples sont traités lors de la description de l'implémentation du système dans la section ci-dessous.

E. L'implémentation

L'implémentation d'un système, tel qu'il a été décrit ci-dessus, est une exploration exhaustive d'un arbre et/ou; en effet, son objectif est de trouver un plan pour un problème posé à partir de faits (informations issues du schéma de résolution de la tâche) et de règles (R1 à R4) et il utilise pour cela un algorithme de parcours de graphe non déterministe ainsi qu'un mécanisme d'unification. Il semble assez naturel alors d'utiliser un langage comme Prolog afin d'éviter une programmation trop fastidieuse. Nous présentons ci-dessous deux moyens pour utiliser Prolog afin de définir un mécanisme d'inférence particulier (paragraphe 1) et l'application particulière pour notre système (paragraphe 2).

1 Un méta-interprète Prolog

Deux solutions sont envisagées pour utiliser Prolog afin d'implémenter un système d'inférence. Il est possible de modifier le programme source d'un interprète Prolog pour qu'il décrive parfaitement le comportement du système voulu. Cette méthode semble cependant hasardeuse car il sera difficile et de localiser les zones à modifier et de mesurer la portée des changements que nous écririons.

La deuxième solution — plus simple de mise en œuvre mais moins performante — permettant de réutiliser certaines fonctions Prolog, particulièrement intéressantes telles que l'unification, le renommage et l'application de substitution, consiste à programmer un méta-interpréteur Prolog. Un méta-interpréteur pour un langage est un interpréteur pour le langage rédigé dans ce même langage [Ster86], ce qui s'écrit très facilement et qui permet la réutilisation des prédicats prédéfinis voulus tout en permettant de modifier facilement le comportement de la machine [Nils90]. Tous les exemples et les explications donnés au cours de ce paragraphe sont très largement inspirés de [Ster86].

Donnons un premier exemple qui simule la réduction de but pour des programmes en Prolog pur :

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```

`solve` est la réalisation du méta-interpréteur qui énonce que le but vide admet une solution et que pour résoudre une conjonction de buts, il faut les résoudre tous. La troisième clause permet de résoudre un but en choisissant une clause du programme dont la tête s'unifie avec le but et en résolvant le nouveau but. Le prédicat Prolog prédéfini `clause(tête, corps)` recherche les

clauses dont la tête s'unifie avec `tête`.L'appel est réalisé par `solve(but):-`. Le fonctionnement d'un méta-interpréteur similaire est expliqué en détail par Nilsson [Nils90].

Les méta-interpréteurs se distinguent par leur granularité, c'est-à-dire par les parties du calcul qui sont mises en évidence. Il s'agit de la choisir la plus rentable possible : rendre visible suffisamment de calculs pour être utilisable et suffisamment peu pour être efficace et facile à écrire. Le méta-interpréteur donné ci-dessus semble le plus adapté. Voyons un exemple qui va illustrer les possibilités de récupération d'informations sur le déroulement de la résolution, en l'occurrence l'arbre de démonstration du but :

```
solve(true, true).
solve((A,B), (PreuveA, PreuveB)) :- solve(A, PreuveA), Solve(B, PreuveB).
solve(A, A:-Preuve) :- clause(A,B), solve(B, Preuve).
```

La résolution de `solve(but, arbre)` renvoie dans `arbre` l'arbre de démonstration de `but`.

Il est également possible de tenir compte les prédicats `!`, `not` ainsi que tous les prédicats du système qu'il n'est pas intéressant d'expliciter pour l'application.

L'utilisation d'un méta-interpréteur a fait ses preuves aussi bien dans des systèmes complexes qui fournissent une trace ou qui raisonnent avec l'incertain, que pour le débogage (lorsque le programme ne se termine pas, qu'il donne des solutions fausses ou qu'il oublie des solutions). On peut donc espérer pouvoir implémenter notre système de cette manière.

2 L'application au système de tâches

Il semble possible en écrivant un tel méta-interpréteur de programmer un système de recherche de stratégies puisqu'il permet d'intervenir sur le mécanisme d'inférence et de le modifier. La sémantique opérationnelle décrite fait intervenir des opérateurs qui ne peuvent pas être traités directement par un programme classique Prolog. De plus, lorsque la fonction d'utilité sera introduite, elle guidera les choix en agissant sur le mécanisme d'inférence auquel il faudra donc avoir accès. Le méta-interpréteur permettra également de récolter, si besoin est, différentes informations telles que le temps d'exécution. Nous allons commencer par indiquer la forme des données, puis nous détaillerons les étapes de construction du système suivant l'ordre qui a été utilisé au début de ce chapitre pour la description du système.

a. Les données

On souhaite que les données qu'utilise le méta-interpréteur soient exprimées sous la forme simple qui a été utilisée pour décrire les décompositions et les spécialisations d'une tâche. On donne ci-dessous quelques exemples de clauses, ce qui permet de réaliser la simplicité d'écriture et surtout la ressemblance avec le formalisme utilisé pour la description.

```
/* Décomposition */
faire_plat_de_spaghetti <- cuisiner_les_spaghetti // cuisiner_sauce.
cuisiner_les_spaghetti <- cuisiner_spaghetti_maison.
cuisiner_spaghetti_maison <- faire_la_pate < former_les_spaghetti <
                             cuire_les_pates.
...
/* Spécialisation */
cuisiner_sauce >> cuisiner_sauce_pour_pates.
cuisiner_sauce_pour_pates >> cuisiner_carbonara.
...
/* Methodes */
cuire_les_pates <- cuire#15.
...
```

On rappelle que << est l'opérateur de spécialisation, que < est l'opérateur d'exécution en séquence et // celui d'exécution en parallèle. `T <- m#c` indique que `m` est la méthode associée à `T` et de coût `c`. Il n'y a pas d'opérateur de choix entre les tâches conformément à la sémantique décrite et aux compétences du système implémenté. Ces différents opérateurs sont introduits grâce aux clauses suivantes utilisant le prédicat Prolog `op(précédence, type, opérateur)` où `précédence` indique la priorité relative aux autres opérateurs (plus `précédence` est faible, plus l'opérateur est prioritaire), `type` indique s'il s'agit d'un opérateur unaire ou binaire (`xfy` indique que l'opérateur admet deux arguments) et `opérateur` est le symbole qui représente l'opérateur décrit :

```
:- op(10,xfy,<-).
:- op(10,xfy,>>).
:- op(5,xfy,<).
:- op(5,xfy,//).
:- op(5,xfy,#).
```

Pour représenter un plan de résolution de la tâche T, on utilise deux opérateurs `seq` et `parallel` dont les arguments représentent les sous-tâches à exécuter en séquence et en parallèle respectivement. Nous n'avons pas introduit d'opérateurs de choix entre méthodes pour définir un plan; on obtient ainsi un ensemble de plans solutions et cela permettra d'obtenir le plan optimal lorsque la notion d'utilité sera introduite,

Ces structures étant définies, nous pouvons à présent décrire l'implémentation du système basé sur les règles simples du paragraphe B.1.

b. Le système de règles simples

Nous appelons `solve(T,P,C)` le méta-interprète qui donne le plan `P` permettant de résoudre la tâche `T` à un coût `C`. On tient à connaître le coût qui sera nécessaire pour l'exécution du plan `P` (cela permet de vérifier en partie le comportement du méta-interprète). Dans notre cas, les clauses se résument à des faits, c'est-à-dire à des clauses dont le corps est toujours vrai (`true`).

Le prédicat `solve` :

Une tâche se résout directement par sa méthode B, s'il existe une clause de la forme `A<-B#C` (règle R1) et dans ce cas le plan associé et le coût sont respectivement la méthode et son coût :

```
solve(A,B,C) :- clause(A <- B#C,true).
```

Une tâche se résout par l'une de ses décompositions si une clause de décomposition apparait dans les données sous la forme générale `A<-B` et si la décomposition peut se résoudre (règles R2 et R3):

```
solve(A,P,C) :- clause(A <- B, true), solvedec(B,P,C).
```

Une tâche se résout par l'une de ses spécialisations si une spécialisation est présente dans les données sous la forme `A>>B` et que `B` peut se résoudre (règle R4) :

```
solve(A,P,C) :- clause(A >> B,true), solve(B,P,C).
```

Voyons plus en détail le prédicat qui résout les tâches apparaissant sous la forme $T_1//...//T_n$ ou $T_1<...<T_n$.

Le prédicat `solvedec` :

Il résout une séquence de sous-tâches ou des sous-tâches à exécuter en parallèle. Précisons que l'on n'autorise pas les expressions de la forme T1//T2<T3 puisque l'on n'a pas défini de priorité entre les opérateurs // et < et surtout que cela n'est pas indispensable vu que l'on peut introduire des tâches supplémentaires afin de traiter le cas voulu, c'est-à-dire suivant les cas :

`T1//T', T' <- T2<T3`
ou
`T'<T3 , T'<-T1//T2`

Afin de traiter la décomposition en parallèle de n sous-tâches, on introduit le prédicat `solvepar` qui est appelé lorsque l'on rencontre l'opérateur // :

```
solvedec(T1//T2,P,C) :- solvepar(T1//T2,P,C).
```

`solvepar` se résout de manière itérative sur l'ensemble des tâches en parallèle : le coût est le maximum du coût des deux sous-plans; le plan s'obtient en concaténant les deux plans par l'opérateur parallel :

```
solvepar(T1//T2,P,C) :-
    solvepar(T1,P1,C1),
    solvepar(T2,P2,C2),
    max(C1,C2,C),
    appendpar(P1,P2,P).
/* la tache ne contient plus d'opérateur //, on la résout par solve */
solvepar(T,P,C) :- solve(T,P,C).
```

On procède de manière similaire pour traiter le cas de décomposition de n sous-tâches en séquence, grâce à l'opérateur `solveseq` :

```
solvedec(T1<T2,P,C) :- solveseq(T1<T2,P,C).
solveseq(T1<T2,P,C) :-
    solveseq(T1,P1,C1),
    solveseq(T2,P2,C2),
    C is C1 + C2,
    appendseq(P1,P2,P).
/* la tache ne contient plus d'opérateur <, on la résout par solve */
solveseq(T,P,C) :- solve(T,P,C).
```

Le cas d'une clause `A<-B`, où `B` est une tâche simple (ce qui est envisageable vu qu'il n'y a pas d'opérateurs de choix entre sous-tâches) est traité par :

```
solvedec(B,P,C) :- solve(B,P,C).
```

Ceci est conforme aux règles définies : c'est en effet la règle de décomposition en séquence qui inclue ce cas.

Voyons un exemple :
Les réponses à la demande `solve(faire_plat_de_spaghetti,P,C).` sont entre autres :

```
P = parallel([cuire,carbonara])
C = 15 ;
P = parallel([seq([pate,former,cuire]),bolognaise])
C = 40 ;
```

Il est également possible de contraindre C ou P afin de limiter le nombre des réponses. Le programme complet et un exemple détaillé sont donnés en annexes.

c. Le système manipulant des variables

Afin de manipuler des variables, il suffit en fait de modifier les données en y introduisant les paramètres adéquats. Le système, sans aucune modification par rapport à la version présentée ci-dessus, va se charger de l'unification des variables entre les données et la tâche à résoudre. Par exemple, pour les données :

```
faire_plat_de_spaghetti([P,S]) <- cuisiner_les_spaghetti(P) //
                                  cuisiner_sauce(S).
cuisiner_les_spaghetti([A,M]) <- cuisiner_spaghetti_achetes(A).
cuisiner_les_spaghetti([A,M]) <- cuisiner_spaghetti_maison(M).
cuisiner_spaghetti_achetes(Q) <- cuire_les_pates(Q).
cuisiner_spaghetti_maison(Q) <- faire_la_pate(Q) <
                    former_les_spaghetti(Q) <    cuire_les_pates(Q).
cuisiner_sauce(R) >>  cuisiner_carbonara(R).
cuire_les_pates(Q) <- cuire(Q)#15.
faire_la_pate(Q) <- pate(Q)#15.
former_les_spaghetti(Q) <- former(Q)#10.
cuisiner_carbonara(Q) <- carbonara(Q)#10.
```

A la question `solve(faire_plat_de_spaghetti([[1,5],3]),P,C)` — ce qui correspond à une demande de faire une quantité 1 de spaghetti s'ils sont achetés, une quantité 5 s'ils sont faits maison avec une quantité 3 de sauce — le système fournit les solutions suivantes :

```
P = parallel([cuire(1),carbonara(3)])
C = 15 ;
P = parallel([seq([pate(5),former(5),cuire(5)]),carbonara(3)])
C = 40 ;
```

Ces résultats sont ceux escomptés.

Afin que le système permette de représenter des situations réelles, il faut que l'on puisse calculer les paramètres des sous-tâches en fonction de ceux de la tâche et pas seulement les transmettre.

d. Le système et les affectations de variables

L'exemple suivant illustre le calcul des paramètres d'une tâche :

```
cuire_les_pates(N) :- cuire(100*N,'g de pates pendant', 15, 'mn').
```

Si N est le nombre de personnes, il faut cuire 100 g de pates par personne. Or, écrit de cette manière, le calcul ne sera pas évalué lors de la recherche de solution, ce qui pose problème pour la suite de la résolution et pour la présentation de la solution. Il faut donc recourir à un artifice permettant d'effectuer les affectations qui sont nécessaires : ajouter la possibilité d'associer une condition d'affectation à chaque clause. C'est pourquoi on introduit le prédicat `§` qui permet de tenir compte des affectations et le prédicat `etl` (nommé ainsi car il réalise un `et` sur une `l`iste) pour traiter les conjonctions d'affectations; la forme générale d'une clause est alors :

```
T <- T' § etl(<liste d'affectations>).
```

où T' est un ensemble de sous-tâches en séquence ou en parallèle (éventuellement réduit à une seule tâche).

Les affectations sont de la forme `aff(Q,R)` et sont traitées par :

```
aff(Q,R) :- Q is R.
```

Le prédicat `etl` est défini par :

```
etl([A|L]) :- A, etl(L).
etl([]).
```

Afin de tenir compte des affectations, le prédicat solve est modifié par l'ajout des clauses :

```
solve(A,B,C) :- clause(A <- B#C§V,true),V.
solve(A,P,C) :- clause(A <- B§V, true), V, solvedec(B,P,C).
solve(A,P,C) :- clause(A >> B§V,true), V, solve(B,P,C).
```

On ajoute ces trois règles aux règles précédentes permettant de définir `solve`, puisque les affectations sont optionnelles. Remarquons que l'on vérifie la condition V avant de continuer la résolution pour ne pas faire de calcul inutile.

Reprenons l'exemple afin de voir que le calcul des paramètres peut se faire de façon immédiate :

```
cuire_les_pates(N) :- cuire(R,'g de pates pendant', 15, 'mn')
                      $ aff(R,100*N).
```

N est le nombre de personnes et R la quantité en grammes.

Vérifier l'applicabilité des spécialisations nécessite l'introduction d'un prédicat similaire; rappelons que méthode et décomposition s'appliquent toujours, d'après nos hypothèses.

e. Le système autorisant des conditions

Chaque clause de spécialisation doit pouvoir être conditionnelle; on ajoute donc un opérateur de condition `$` — qui doit être plus prioritaire que `>>` – et un prédicat `oul` qui permet de traiter les disjonctions de conditions et qui se rajoute au prédicat `etl` décrit ci-dessus :

```
oul([A|L]) :- A.
oul([A|L]) :- oul(L).
```

Les conditions autorisées sont les comparaisons sous la forme :

```
eq(Q,R) :- Q = R.
sup(Q,R) :- Q > R.
inf(Q,R) :- Q < R.
supe(Q,R) :- Q >= R.
infe(Q,R) :- Q =< R.
```

Afin de tenir compte des conditions qui associées aux clauses, le prédicat solve est modifié de manière similaire que pour le prédicat `$` et en tenant compte des combinaisons possibles de ces deux opérateurs. Nous verrons en fait en III.D.3 qu'afin de contrôler l'exécution, on remaniera quelque peu ce schéma général afin de déterminer explicitement la résolution choisie pour une tâche donnée.

Donnons un exemple de spécialisation conditionnelle :

```
cuisiner_sauce(Q,T) >> Cuisiner_sauce_pour_pates(Q)
                       $ oul([eq(T,'spaghetti'),eq(T,'nouilles')]).
```

T représente le type du plat que la sauce doit accompagner.

Lors de la recherche de solution, les différentes conditions sont prises en compte pour limiter l'usage de certaines clauses ou pour inférer des paramètres, ce qui permet d'obtenir une solution conforme au problème posé.

F. Terminaison et complexité

Le méta-interprète que nous avons proposé possède une puissance similaire à celle de Prolog sans négation. En effet, il permet de résoudre tout programme Prolog en modifiant simplement les notations (une règle écrite sous la forme `R(X) :- P(X), Q(X)` est réécrite en `R(X) <- P(X) < Q(X)`). Pour l'application que nous voulions faire, il fallait tenir compte d'un certain nombre d'opérateurs inconnus pour Prolog, ce qui a motivé l'écriture d'un méta-niveau.

Il existe donc, comme dans Prolog, une possibilité de non terminaison qui correspond, dans notre cas, à une boucle dans les relations entre tâches. Ainsi, lors de la résolution d'une tâche, il se peut que le programme boucle s'il y a un nombre infini de solutions ou s'il n'y en a aucune. Remarquons que cet aspect du méta-interprète permet d'exprimer la récursivité entre les tâches, cas pour lequel l'infinité de solutions est naturelle et attendue; cela peut refléter des situations réelles comme nous le verrons. De plus, vu que les connaissances exprimées sont finies, le programme ne peut boucler qu'au travers de la récursivité, sinon le programme termine après avoir testé l'ensemble des voies possibles, ce qui conduit à un nombre fini (éventuellement nul) de solutions.

Quant à la complexité, elle dépend des données qui sont fournies au programme vu que la résolution que nous avons adoptée correspond à une recherche exhaustive, dans la base contenant les données : les décompositions, spécialisations et méthode directe associées à chaque tâche.

Toutes ces remarques nous amènent à nous interroger sur l'aspect dénotationnel du système de tâches.

G. Esquisse de sémantique dénotationnelle

Ce chapitre a été consacré à l'étude de la sémantique opérationnelle, c'est-à-dire que nous avons étudié le fonctionnement du moteur de tâches. Il s'agit à présent de caractériser son action par rapport à l'environnement dans lequel il évolue et de voir l'interprétation qui peut en être faite. En fait, il existe une sémantique dénotationnelle pour la représentation des connaissances et son rapport avec le monde réel; après l'avoir présentée, nous allons voir comment les tâches s'intègrent dans ce modèle.

Shong Ye Tan [Tan93] propose un modèle formel de bases de connaissances basé sur les «T-algèbres». Une T-algèbre est une extension des «OSF-algèbres» données par Aït-Kaci

comme sémantique du langage LIFE [Aït-K93]. Nous donnons ici les caractéristiques de la modélisation par les T-algèbres sans les décrire plus précisément (une description complète de celles-ci est donnée dans le rapport [Tan93]).

Une T-algèbre permet de modéliser un état d'une base de connaissances à objets. Une T-algèbre admet un *domaine* et un ensemble de *sortes* qui correspondent respectivement aux objets et aux classes d'une base de connaissances. De plus, un objet contient un ensemble d'attributs qui peuvent être vus comme des liens avec les autres objets; un attribut correspond à un *trait* dans la T-algèbre. De plus, les traits sont considérés comme des fonctions partielles, c'est-à-dire qu'ils ne sont pas définis pour tous les objets; cela permet de rendre compte de l'incomplétude des bases de connaissances. Elles sont en effet incomplètes par nature :

— il est impossible d'acquérir toutes les connaissances en un temps limité;

— l'espace de stockage est limité;

— le langage impose des restrictions;

— et enfin, les requêtes doivent être performantes.

Entre deux T-algèbres $\mathfrak{A}$ et $\mathfrak{B}$, il peut exister une fonction : un *T-morphisme*. Les T-morphismes correspondent à des relations d'approximation. Si $\mathfrak{B}$ est l'image de $\mathfrak{A}$ par un T-morphisme, $\mathfrak{A}$ est une approximation de $\mathfrak{B}$, c'est-à-dire que $\mathfrak{B}$ admet un plus grand domaine d'objets ou que les traits sont définis pour plus d'objets : $\mathfrak{B}$ contient plus d'informations.

Une T-algèbre permet donc de modéliser un état de la base de connaissances et c'est une notion suffisamment «large» pour permettre également de modéliser le monde réel (postulat de [Tan93]). Dans une telle T-algèbre, le domaine contient tous les objets du monde et les traits recouvrent l'ensemble des relations imaginables entre deux objets. La relation entre le modèle du monde et les bases de connaissances peut être formalisée comme un T-morphisme. Chaque base de connaissances est donc vue comme une approximation du modèle du monde réel selon une perspective donnée. Dans notre cas, une perspective correspond à un ensemble de traits, c'est-à-dire que chaque base de connaissances tient compte uniquement de certaines relations entre les objets.

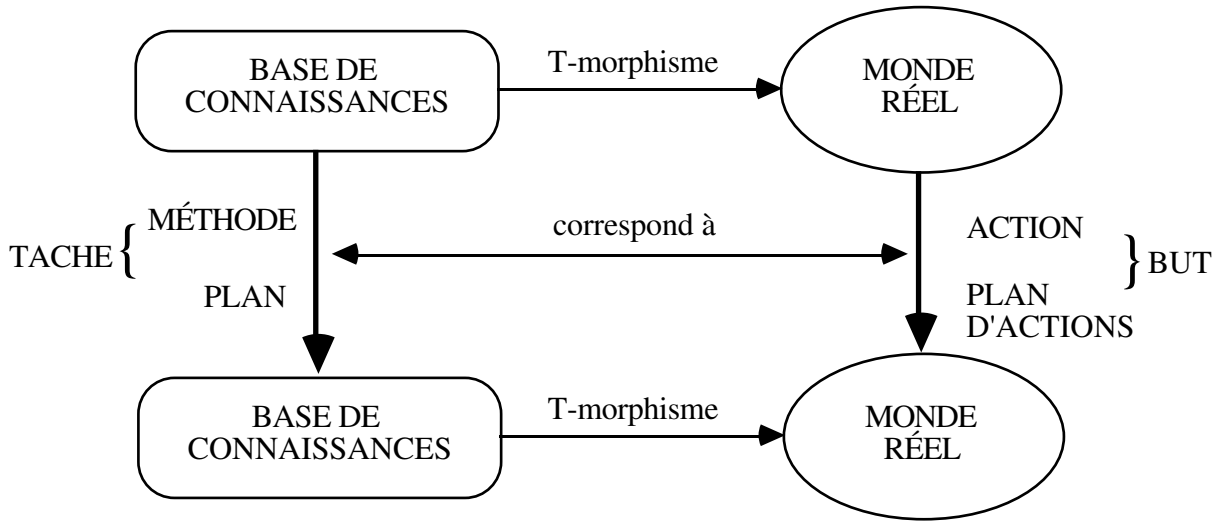
Dans un tel modèle, une tâche modifie l'état de la base de connaissances avec deux objectifs distincts : compléter la base afin qu'elle se «rapproche» du monde réel ou modéliser les actions qui interviennent dans le monde. Selon cette dernière approche, une tâche à réaliser dans une base de connaissances dénote un but que l'on veut faire atteindre au monde réel. Pour résoudre une tâche, il peut exister soit une méthode — qui dénote une action dans le monde réel —, soit un plan qui est un assemblage de méthodes et qui dénote un plan d'actions dans le monde réel. La résolution d'une tâche dans une base de connaissances permet donc de représenter la résolution d'un but dans le monde réel, ce qui permet de donner un sens à l'expression que nous avons utilisée en sémantique opérationnelle :

Plan(base de connaissances) ***:-*** *Tâche*

le plan d'action associé au plan permet de faire parvenir le monde dans un état qui satisfait le but dénoté par la tâche, soit :

Plan d'actions (monde) ***:-*** *But*

Le diagramme suivant résume le parallèle que nous avons mis en évidence :



Bases de connaissances et monde réel : les bases de connaissances permettent de modéliser le monde réel et la résolution des tâches la manière de lui faire atteindre un but.

Le système de tâches présenté peut donc être placé dans un cadre plus général, dans lequel il permet la transition entre bases de connaissances, conformément aux modifications du monde qui font suite à l'application des actions concrètes qui lui font atteindre un but. Notons que la non terminaison évoquée dans la section précédente n'est pas critique puisqu'elle peut très bien dénotée une suite infinie d'actions, par exemple dans le mouvement d'un robot. Cela correspond à un plan d'actions réel et donc doit pouvoir être pris en compte par le système de tâches.

Ces pistes pour établir une sémantique dénotationnelle des tâches pourront faire l'objet de travaux ultérieurs.

H. Conclusion

Le système, tel qu'il a été présenté, permet effectivement de rendre compte d'un mécanisme de résolution de tâche, semblable à celui proposé par Jutta Willamowski. L'implémentation immédiate qui a pu en être faite en exploitant les possibilités de Prolog, a intuitivement validé nos propos, sans que nous définissions complètement une sémantique dénotationnelle. Il aurait été possible d'envisager une modélisation différente. Prenons l'exemple de Rao [Rao94] qui propose une sémantique basée sur la logique dynamique pour modéliser la recherche et la

reconnaissance de plan. La sémantique opérationnelle dans ce cas est liée à la preuve de formules.

Cependant, il reste primitif puisqu'il est non déterministe. Nous voudrions que le traitement d'une alternative soit résolue de manière rationnelle, afin de permettre au système de traiter effectivement des problèmes sans se soucier de leur complexité dans le pire cas. Nous allons donc le doter de capacités de rationalité pour qu'il fournisse les meilleures solutions selon la fonction d'utilité définie, suivant en cela le système TRAM de Mathias Chaillot. Le prochain chapitre est consacré aux différentes approches de la rationalité, à l'expression de la fonction d'utilité et à leur application au système de tâches.

III. Tâche et rationalité

Rappelons que l'objectif du projet est d'unifier les travaux de J. Willamowski et de M. Chaillot sur la résolution de tâches. Or, ce dernier s'est intéressé à la rationalité pour guider les choix de stratégies. Nous allons présenter l'idée générale de rationalité (section A), utilisée dans différents domaines, puis voir plus explicitement comment cela peut s'appliquer aux tâches (section B). Il existe alors deux manières principales — statique (section C) ou dynamique (section D) — de concevoir l'utilisation de la rationalité dans le système de tâches. Nous conclurons ce dernier chapitre en comparant ces deux stratégies et les cas qu'elles recouvrent (section E).

A. La rationalité

La rationalité est une notion générale qui apparaît dans des domaines où des choix se présentent; voyons ce que signifie cette idée, en quoi elle est nécessaire et comment la mettre en œuvre, en particulier grâce à l'introduction d'une fonction d'utilité.

1 Qu'est-ce qu' «être rationnel» ?

La logique est considérée comme la manière rationnelle de penser. A partir des connaissances `p` et `p=>q`, on peut (et on doit) déduire `q`. Tous les systèmes basés sur les inférences logiques suivent une seule règle : «Tout ce qui est valide doit être déduit». Aucune considération de valeurs des résultats ou des inférences n'est prise en compte; la logique est incapable de savoir quelles inférences choisir, quelles sont les meilleures possibles puisque son action se limite à déduire des connaissances; elle ne fait pas aucune distinction de valeur entre deux assertions vraies [Doyl89];.

Une deuxième conception de la rationalité est économique et elle étudie les actions ou les choix. Pour faire le meilleur choix possible parmi un ensemble d'alternatives, il faut tenir compte des préférences des agents; la notion de préférence est représentée par un ordre partiel sur les différentes alternatives qui se présentent; l'agent va choisir l'alternative qu'il préfère. C'est en se référant à cette définition que l'on prouve facilement que jouer au loto est irrationnel [Doyl90]. En effet, pour le joueur de loto, l'exigence principale est de gagner de l'argent et en considérant ce critère, tout raisonnement rationnel doit lui recommander d'arrêter de jouer. En théorie de la décision, la rationalité utilise de plus, les probabilités de chaque résultat de chaque alternative, résultats qui sont quantifiés et «être rationnel» signifie que l'on choisit la meilleure action vu le but à atteindre et les préférences de l'agent. Pollock [Poll93] décrit un agent rationnel comme agissant sur le monde pour que sa situation le satisfasse le plus possible. Un

agent possède les notions de préférence (qui lui permet d'évaluer une situation) et d'intention (qui définit les actions à exécuter afin d'augmenter sa satisfaction). Il établit un parallèle avec le raisonnement humain tout en montrant que les hommes ne sont pas totalement rationnel.

Cependant, en intelligence artificielle, une telle définition de rationalité ne peut pas s'appliquer vu que l'on ne fait pas les mêmes hypothèses sur les agents. En théorie de la décision, les agents possèdent toutes les connaissances nécessaires pour effectuer le bon choix, alors que les agents qui interviennent en intelligence artificielle, et qui modélisent des systèmes réels, ont des capacités de calculs limitées, et par conséquent une capacité bornée de comparaison des choix. De plus, les préférences sont incomplètes et ils ne sont pas logiquement omniscients (terme détaillé ci-dessous). Il faut donc tenir compte de ces limites en contrôlant et optimisant le processus de décision sur les efforts de raisonnement à fournir et ne plus considérer seulement les résultats des choix. Il faut bien distinguer les notions de rationalité et de rationalité limitée (c'est-à-dire qui concerne des agents aux capacités limitées) puisqu'une stratégie rationnelle pour un agent peut se révéler désastreuse pour un agent limité : le fait de tenir compte de la limitation de ressources ou non bouleverse totalement l'optique dans laquelle les décisions sont prises. Good [Good52], le premier à prendre conscience de ce type de rationalité, en distingue deux types : celle dite de «type II» pour des agents qui raisonnent en connaissant leurs propres limites, par opposition à celle de «type I» qui ignore le coût de raisonnement.

2 Pourquoi être rationnel ?

Si aucune contrainte temporelle n'est imposée au système, la logique va permettre de trouver une solution satisfaisante, si une telle solution existe; c'est par exemple le cas de notre programme dans son état actuel. Cependant, ce cas est fort peu réaliste, on ne souhaite pas souvent perdre du temps qui pourrait être employé à la résolution d'autres problèmes. Il faut alors bien se rendre compte que les humains comme les machines ont des ressources limitées. Pour les machines, ce sont les ressources de raisonnement qui sont primordiales. Si elles sont bornées ou simplement si leur coût n'est pas négligeable, il est préférable de les employer au mieux pour trouver la solution à un problème posé.

C'est une constatation qui révèle les failles de la plupart des systèmes développés qui sont basés sur «l'omniscience logique» des agents qu'ils modélisent; omniscience logique signifiant entre autre que leur connaissance est fermée par implication. Ce type de système ne peut en aucun cas servir à modéliser des humains qui ne sont certainement pas omniscients [Fagi87]. De plus, dès que la connaissance est exprimée dans un langage un peu puissant, cette omniscience logique ne peut être atteinte qu'en un temps exponentiel. C'est pourquoi, Mamede [Mame91] notamment, s'intéresse à la construction d'un système qui raisonne avec des ressources limitées. Shugan [Shug80], quant à lui, en étudiant le comportement d'un consommateur face à un ensemble de produits, met en évidence qu'un certain coût est

nécessaire pour choisir et que la qualité du choix risque de souffrir du coût nécessaire pour la prise de décision. Il ne faut donc pas se focaliser uniquement sur la qualité propre du choix.

Il faut également avoir présent à l'esprit que certaines machines doivent réagir en fonction des ressources critiques de leur utilisateur. Il peut s'agir du temps dans les applications «temps réel» qui doivent donner un résultat le plus satisfaisant possible dans un laps de temps très court. La rationalité permet donc de prendre en compte les ressources consommées lors des choix, afin de modéliser des systèmes ayant des capacités de raisonnement limitées et de produire des résultats satisfaisant ces exigences.

Il faut donc bien voir la différence de cette stratégie de résolution avec celle qui permet de trouver le plan le plus utile en se basant sur une minimisation ou une maximisation d'une fonction d'évaluation; cette dernière est relativement facile puisqu'il est possible d'explorer le graphe en construisant la fonction d'évaluation (moyennant certaines approximations). Or, ce qui nous intéresse est de minimiser non pas le coût de la solution mais le coût de la solution et de sa recherche, ce que ne permet pas la méthode ci-dessus qui ne tient pas compte du temps passé à chercher l'optimisation.

3 Comment être rationnel ?

Etant donné que les ressources sont limitées, il faut donc utiliser le plus efficacement possible toutes celles qui sont disponibles. En anticipant la limitation des capacités, les résultats obtenus seront beaucoup plus satisfaisants que si l'on se contente d'arrêter le raisonnement lorsque les ressources sont épuisées. Deux optiques de la notion d'efficacité apparaissent. Pour Levesque & Brachman [Leve87], «être efficace» revient à répondre suffisamment rapidement (le temps étant considéré comme ressource critique) à l'application la plus critique. Ils recommandent donc des systèmes dans lesquels toutes les opérations possibles peuvent être faites très rapidement. Pour cela, l'expressivité du langage utilisé est réduite afin que tous les problèmes exprimables puissent être résolus rapidement. Cette expressivité réduite revient évidemment à limiter le nombre des opérations qui sont possibles. Cette attitude caractérise toutes les positions basées sur la complexité au pire et en particulier les approches «temps réel».

Doyle et Patil [Doyl91], quant à eux, montrent que cette limitation est si importante que le nombre des problèmes solubles est dérisoirement faible et que cette faiblesse d'expressivité interdit toute résolution de problèmes intéressants. En fait, ils adoptent, conformément à cette constatation, une définition différente de la notion d'efficacité; pour eux, «être efficace» signifie qu'il faut résoudre un problème donné de la meilleure façon possible. Ils préconisent de ne proscrire aucun type de problèmes a priori, quel que soit la complexité. Il s'agit alors seulement de voir quelles sont les ressources que l'on veut bien attribuer à la résolution d'un problème suivant l'importance accordée à son résultat, et d'utiliser toutes celles qui sont disponibles afin d'avoir un résultat encore plus satisfaisant, plutôt que de terminer en avance (dans le cas du temps). Il faut donc évaluer l'utilité de la résolution d'un problème afin de fixer les ressources

allouées : des heures ou des jours de calculs pour résoudre certains problèmes primordiaux ne sont pas considérés comme inutiles, alors que pour d'autres, l'allocation de plus de quelques secondes est complètement déraisonnable . La majeure différence entre les deux points de vue provient donc du fait que d'un côté, Levesque & Brachman veulent obtenir une solution en un temps imparti sans tenir compte des bénéfices tirés du résultat, alors que de l'autre, Doyle & Patil prennent en compte les ressources consommées mais également la valeur du résultat. De plus, afin d'évaluer l'efficacité d'une résolution (le temps nécessaire), ces derniers recommandent de considérer le cas général et non le pire cas, sauf s'il y a des conséquences dramatiques en cas de réponses trop lentes. Pour le calcul du temps moyen de recherche, ils proposent une estimation à partir d'expériences et une révision régulière au vu des nouvelles données.

De même, Shugan [Shug80] privilégie une prise de décision tenant compte du temps qui s'écoule et de la qualité du choix qui sera fait : il faut alors trouver un compromis entre une solution longue (voire infiniment longue) qui fournit le meilleur résultat et une solution rapide qui donne un résultat de qualité incertaine. Howard [Howa67] est le précurseur de ce type de raisonnement, lorsqu'il étudie la valeur de l'information; il examine les gains qu'apportent des connaissances supplémentaires sur les variables qui vont influencer le choix et il détermine alors ce que valent ces connaissances. Il s'intéressait pour sa part au coût pécuniaire, lors de la négociation d'un contrat, des informations sur les offres des concurrents ou sur le coût de réalisation du contrat. Il existe un juste milieu qui permet d'optimiser les chances de gain d'argent.

De manière très générale, la rationalité préconise de choisir l'action d'utilité espérée maximale, sachant que l'utilité d'une action est une fonction qui dépend des ressources consommées lors de la recherche de cette action et de son utilité intrinsèque. La valeur de la fonction d'utilité ne peut cependant être connue avec certitude puisque l'on ne peut pas connaître les conséquences d'une action avant de l'avoir exécutée, même si elle est déterministe [Russ91].

Le temps est la ressource consommée lors d'une résolution dont on tient compte généralement. En effet, il n'y a guère que le temps et la place mémoire qui soient consommés par un mécanisme d'inférence, cette dernière étant moins vitale car disponible plus facilement. Les systèmes sont le plus souvent des systèmes dits «réactifs» qui sont utilisés pour des applications en temps réel et qui doivent donc réagir en un temps très court en donnant une solution satisfaisante.

Remarquons que la fonction d'utilité peut dépendre des ressources (quelles qu'elles soient) de différentes manières [Horv89] :

— l'urgence : cela reflète le coût des ressources en fonction du temps qui s'écoule.

— la dernière limite ("deadline") : cela permet de modéliser la disponibilité des ressources; une fois une certaine quantité consommée, il faut arrêter les calculs.

— une combinaison des deux : les ressources sont coûteuses et bornées.

Si l'on considère que les ressources peuvent s'épuiser, il faut que le système réagisse à ce moment-là. Le système RR, pour "Reasoning with Resources" [Mame91], arrête d'inférer lorsque les ressources sont consommées et donne à ce moment-là une réponse conditionnelle (les conditions reflétant ce qu'il reste à établir afin d'être sûr de la réponse) ; il permet ainsi de modéliser des raisonneurs non omniscients en limitant ses capacités à inférer et surtout cela permet de reprendre une résolution inachevée, en terminant simplement l'examen des conditions restantes.

Horvitz [Horv88], quant à lui, s'intéresse à des systèmes pour lesquels la qualité du résultat et la quantité de ressources varient au cours du temps. Il propose une stratégie qu'il nomme "flexible computation", stratégie qui peut être arrêtée à tout moment et qui fournit alors un résultat partiel; cette méthode s'applique pour des solutions dont l'utilité est fonction continue du temps passé dans les calculs. Dean [Dean91] adopte un algorithme similaire ("anytime algorithm"). Ces techniques, qui sont continues, s'adaptent par exemple au diagnostic médical : plus le temps de calcul est long, plus l'utilité du traitement proposé est importante (vu que le diagnostic est de plus en plus précis); cependant, d'un autre côté, un délai dans l'application du traitement fait décroître son utilité; comme les algorithmes sont continus, il est possible de déterminer l'utilité totale (tenant compte de l'utilité propre au traitement et du délai) en fonction du temps et donc de trouver le temps qui la maximise. Le fait de proposer des résultats partiels permet de ne pas perdre tout le raisonnement fait si les ressources se révèlent insuffisantes pour terminer le traitement. Le résultat final sera certainement moins bon que celui trouvé par un mécanisme ne se souciant pas de continuité mais cette idée semble ingénieuse puisque qu'il faut éviter à tout prix de gaspiller des ressources; il faut cependant l'appliquer dans un domaine où des résultats intermédiaires sont exploitables.

Jusqu'à présent, nous ne nous sommes souciés que de la réalisation d'un seul but, Etzioni [Etzi91] s'intéresse au problème du contrôle suivant trois axes : le coût du calcul et de l'exécution et la valeur du but; lors de la résolution, il recherche tout d'abord parmi l'ensemble de buts celui qu'il faut résoudre puis il le résout de manière rationnelle. Les buts qui seront résolus sont ceux qui sont le plus facile à résoudre par rapport au coût nécessaire. Il donne l'exemple d'une recherche d'un trésor disséminé dans une île; il connaît des estimations du temps qu'il faut pour creuser et de la somme qui peut être enfouie; grâce à cela il détermine les endroits optimaux de recherche. Il tient compte aussi des moyens qui permettent de résoudre les buts : faut-il dynamiter un endroit pour découvrir quelques pièces au risque d'enfouir à jamais le reste du trésor? Son objectif est donc un peu différent mais il se ramène au cas d'un seul but ayant plusieurs méthodes possibles; et il montre que dans ce cas son algorithme est optimal.

Cette notion de rationalité met en place une nouvelle manière de concevoir les systèmes, en tentant de maximiser une fonction dépendant du coût en temps et de la qualité; il reste alors à

mettre au point un moyen permettant de trouver les solutions qui maximisent l'utilité et tout d'abord à calculer cette fonction d'utilité.

4 La fonction d'utilité

Comme nous l'avons dit précédemment, on ne peut que calculer l'espérance de l'utilité de chaque action. Russel décrit un moyen d'obtenir l'espérance de la fonction d'utilité qui permet de choisir rationnellement les actions à tester [Russ91] :

L'utilité espérée U d'une action A_i est l'utilité du système à être dans l'état $[A_i]$ (c'est-à-dire l'état après l'exécution de A_i) qui se calcule en fonction de l'utilité des conséquences de cette action pour chaque état initial possible W_k du système, pondérée par les probabilités de ces situations $p(W_k)$ ($[A_i,W_k]$ est l'état du système après l'exécution de A_i si son état était W_k):

$$E(U([Ai])) = \sum_k p(W_k)U([A_i,W_k])$$

Il faut pouvoir évaluer la fonction d'utilité. On distingue alors les calculs partiels et complets — un calcul complet aboutit à l'exécution d'une action, un calcul partiel non. L'utilité d'un calcul complet est l'utilité d'exécuter l'action α_{Si} dans la situation [Si] qui résulte de l'action calculatoire donc:

$$U([S_i]) = U([\alpha_{Si},[S_i]])$$

L'utilité d'un calcul partiel est un peu plus complexe car on ne peut calculer l'utilité que pour une action externe et donc il faut sommer sur l'ensemble T des séquences complètes t (qui aboutissent à une action α_t) de calculs possibles à la suite de S_i :

$$U([S_i]) = \sum_{t\in T} p(t).U([\alpha_t,[S_i.t]])$$

où $[S_i.t]$ est l'état qui résulte de l'exécution de la séquence $S_i.t$.

Afin de pouvoir faire un choix judicieux, il faut disposer de la valeur nette d'un calcul (V(Si)), c'est-à-dire le gain d'une action de calcul Si par rapport à l'exécution de l'action par défaut α :

$$V(Si) = U([Si]) - U([\alpha])$$

Le signe de $V(S_i)$ permet donc de voir s'il est utile d'exécuter S_i.

On peut également tenir compte du temps consommé explicitement en décomposant l'utilité en utilité intrinsèque (U_i) et temps de calcul (TC). La valeur nette pour un calcul complet devient alors :

$$V(Si) = \Delta(Si) - TC(Si) \text{ avec } \Delta(Si) = U_i([\alpha_{Si},[Si]]) - U_i([\alpha])$$

Nous avons décrit un moyen général pour estimer la valeur de l'utilité; voyons comment appliquer cela aux tâches.

B. L'utilité et les tâches

Notre problème consiste à résoudre rationnellement une tâche sachant que pour chacune il existe un certain nombre de possibilités (décompositions, spécialisations ou méthode directe). Nous présentons tout d'abord la vision du système TRAM (paragraphe 1) afin de voir par la suite comment étendre l'utilisation de la rationalité aux possibilités de décompositions. Nous allons voir de manière générale comment appliquer la notion de rationalité et d'utilité à la résolution de tâches, en regardant en particulier les ressources qui sont mises en jeu (paragraphe 2), l'expression de la fonction d'utilité (paragraphe 3) et son évaluation (paragraphe 4).

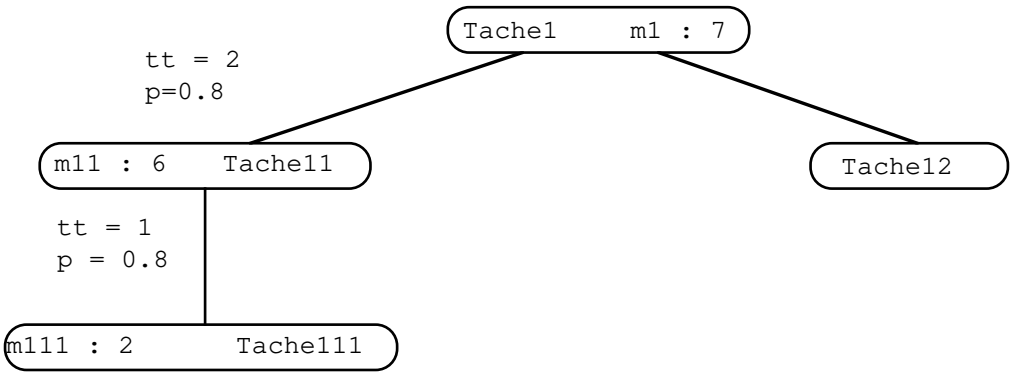
1 Rationalité et tâches spécialisables

Le système TRAM, un environnement réactif de résolution de problèmes, présente comme caractéristique une sélection de méthodes qui utilise une fonction d'utilité [Chai93]. Pour résoudre un problème donné, on dispose d'une hiérarchie de tâches (résolvant toutes le même problème mais s'appliquant pour des cas différents) organisées selon leurs conditions d'activation; plus les conditions d'activation d'une tâche sont restreintes, plus elle est «basse» dans cette hiérarchie et plus son exécution est adaptée au problème à traiter. Cette hiérarchie correspond à un arbre de spécialisation. Lors de la recherche de solution, il s'agit de voir s'il vaut mieux appliquer la méthode disponible ou persévérer afin de trouver une méthode plus rapide (une méthode s'appliquant pour des conditions d'activation plus restreintes devrait être plus rapide puisqu'elle est plus spécifique, sinon la méthode plus spécialisée n'a aucune raison d'être). La sélection se fonde donc sur les deux critères d'activabilité et de rapidité.

Elle se base sur le calcul de l'espérance d'utilité d'une action calculatoire (test d'appartenance à une sous-classe, dans notre cas) défini par Russel [Russ91] que nous avons détaillé en A.4. Il ordonne les tests suivant leur utilité espérée, ce qui permet de fixer le test qui sera effectué en premier; s'il échoue, on recalcule les espérances d'utilité sachant que le premier test a échoué, ce qui modifie les probabilités d'appartenance aux sous-classes. Tant qu'il existe des sous-tâches dont l'espérance d'utilité est positive, on choisit celle qui a la plus forte espérance d'utilité et on la teste. Si l'espérance d'utilité devient négative, le test n'est pas rentable et il est préférable d'appliquer la stratégie disponible. Cela peut paraître un peu lourd au premier abord, cependant les ressources nécessaires pour ces calculs peuvent être considérés comme négligeables par rapport à celles utilisées par les tests et celles que ces calculs peuvent faire gagner. Il faudrait un méta-méta-niveau qui contrôle ce type de décisions, mais il risquerait d'alourdir considérablement le traitement sans apporter beaucoup; on pourrait considérer un niveau encore supérieur pour nous aider dans notre décision et ainsi de suite. Cependant au premier ou

deuxième on possède déjà suffisamment d'informations pour traiter de manière satisfaisante le problème sans perdre de ressources en réfléchissant sur la manière de réfléchir sur la manière de réfléchir sur ….

Ce système permet de résoudre de manière optimale un problème à une hypothèse près; il reprend l'hypothèse "single-step" de Russel (que nous traduirons dans la suite par l'hypothèse «un pas en avant») afin de pouvoir calculer l'utilité des sous-tâches ; remarquons que cette hypothèse peut faire rater la meilleure solution. En effet, lors du calcul de l'utilité, on tient compte du temps d'exécution de la méthode alors que celui de l'une de ses sous-tâches peut être bien inférieur et cette sous-tâche serait donc la meilleure, comme l'illustre l'exemple ci-dessous.




```
en tenant compte de la méthode directe uniquement:
temps d'exécution de tache11 : 6
E(Tache11) = 0.8 * (7-6) - 2 = - 1.2 < 0
-> la tache 11 n'est pas un bon choix

en tenant compte de l'application eventuelle de m111:
temps moyen d'exécution de tache11 : 0.8*2 + 1 + 0.2*6 = 3.6
E(Tache11) = 0.8 * (7-3.6) - 2 > 0
-> la tache 11 semble meilleure que m1, on la teste
```

Exemple de résolution non optimale à cause de l'hypothèse «un pas en avant» : en relâchant l'hypothèse le résultat obtenu est meilleur.

Cependant, il n'est envisageable de calculer l'utilité d'une tâche en tenant compte de toutes ses sous-tâches que dans certains cas particuliers (lorsque l'on dispose de temps non critique comme le cas traité dans la section C), sinon il faut poser des hypothèses similaires.

Ce système ne recouvre pas toutes les résolutions que nous voulons traiter, c'est pourquoi nous allons voir de manière un peu plus générale le rapport que l'on peut établir entre les tâches et la rationalité.

2 L'utilité et les ressources

L'utilité d'une solution d'un problème dépend de l'utilité intrinsèque de la solution (sa précision, son temps d'exécution, son prix, …) mais également des ressources qui ont été

consommées pour la trouver et la choisir; il s'agit alors de ressources de calcul telles que le temps ou la place mémoire. Le plus simple est de choisir un seul critère identique pour l'utilité intrinsèque et les ressources consommées, critère additif pour que l'on puisse calculer des différences de fonction d'utilité afin d'évaluer la valeur nette d'une solution.

Cependant, une autre voie est plus communément adoptée et semble plus naturelle. Il s'agit de prendre en compte le temps comme ressource consommée par la recherche, l'utilité intrinsèque des tâches étant calculée en fonction d'un autre critère, la précision par exemple. Mais il faut alors être capable d'évaluer la fonction d'utilité associée à chaque tâche de l'arbre : quelle importance relative accordée aux différents paramètres ? On peut imaginer que l'on ait fixé un ordre sur les critères; on compare deux solutions suivant le premier critère et à valeurs égales on regarde le critère suivant. Il est peut-être également possible d'accorder un certain poids à chacun des critères et de les ramener à des valeurs numériques comparables. Dans tous les cas, la solution ne semble ni très satisfaisante, ni très facile à mettre en œuvre. C'est pourquoi, afin de poser dans un premier temps des bases solides, nous proposons de nous intéresser à une seule ressource, ressource critique de la machine ou de l'utilisateur.

3 La fonction d'utilité

Nous considérons donc que la fonction d'utilité est calculée en fonction d'un seul critère, le temps le plus souvent; c'est-à-dire que nous considérons que l'utilité intrinsèque d'une action est sa rapidité d'exécution. Ce problème, qui tente de maximiser un seul paramètre, ne se ramène cependant pas trivialement à un problème de recherche de solution optimale dans un graphe puisque l'on tient compte du temps qui va être nécessaire pour trouver une telle solution. Il est donc possible que l'on se contente d'une solution moins rapide à l'exécution mais trouvée plus rapidement.

Dans le système, à chaque tâche est associée une utilité qui n'est pas connue de façon certaine par le système. En effet, on dispose d'un catalogue de stratégies possibles pour résoudre une tâche (les arbres de décomposition et de spécialisation) mais pour un problème spécifique donné aux paramètres fixés, on ne sait pas quelles sont celles qui sont applicables. On ne peut donc pas connaître a priori les choix qu'il faut effectuer entre les différentes décompositions et spécialisations possibles.

L'utilité, via la valeur nette, va nous permettre de tester les différentes solutions possibles dans l'ordre le plus probable de gain, c'est-à-dire que l'on testera en premier celle qui est la plus susceptible d'apporter un gain de temps important tout en en consommant peu. La valeur nette de l'exécution d'une sous-tâche ST_i pour résoudre une tâche donnée est calculée par la formule donnée en A.4 que nous rappelons :

$$V(ST_i) = \Delta(ST_i) - TC(ST_i)$$

$\Delta(ST_i)$ est le gain espéré de ressources en appliquant la sous-tâche plutôt que la méthode. Afin de pouvoir calculer cette valeur, il faut faire l'hypothèse que l'on choisit les actions suivant leur bénéfice estimé immédiat, c'est-à-dire que l'on ne considère qu'un seul pas de calcul (l'hypothèse «un pas en avant»): l'utilité d'une étape de calcul est l'utilité de l'action immédiate qui lui est associée et ne dépend pas des calculs futurs potentiels; toutes les étapes calculatoires sont supposées complètes. Dans ce cas, si on note g_i le gain en ressources intrinsèques de la méthode associée à la sous-tâche par rapport à la méthode courante, alors $\Delta(ST_i) = p_i * g_i$ où pi est la probabilité que ST_i soit applicable (car le gain n'est effectif que si la sous-tâche s'applique, dans le cas contraire le gain est nul). On a donc :

$$V(ST_i) = p_i * g_i - TC(ST_i)$$

Cette valeur n'est cependant pas une valeur exacte vu qu'elle dépend de l'applicabilité de la sous-tâche et que le gain de ressources n'est qu'une valeur estimée. Remarquons ici que la valeur nette prend en fait en compte deux critères qui sont : la ressource consommée et l'applicabilité des solutions.

Voyons comment on peut calculer une telle valeur nette (estimée).

4 Le calcul de la valeur nette de gain d'utilité

Afin que l'utilité soit une notion qui puisse servir pratiquement pour rendre un système rationnel, il faut que l'on puisse calculer la valeur nette de toutes les sous-tâches d'une tâche donnée afin de choisir celle qui maximise la fonction d'utilité. Il faut donc connaître les valeurs numériques des probabilités, des ressources consommées lors de la recherche et des gains en ressources (ou la valeur des ressources consommées en exécution).

a. Les probabilités

Elles sont calculées à partir d'un jeu de tests qui doit être représentatif de l'utilisation qui va être faite du système. On pourrait envisager une remise à jour à partir des problèmes qui seront traités mais cela n'a aucune raison d'être puisque l'on suppose que les valeurs sont conformes à la réalité. Afin d'illustrer nos propos, nous traitons un exemple en parallèle de nos explications.

Le jeu de tests représentatif (de l'usage qui va être fait de la tâche) peut se résumer par un tableau indiquant quelle sous-tâche s'applique pour quel exemple (les exemples se différencient pas la valeur des paramètres du problème). Le tableau suivant présente 10 exemples pour lesquels sont indiquées (par des x) les sous-tâches qui sont applicables.

EX	1	2	3	4	5	6	7	8	9	10
ST1	X				X	X				X
ST2		X		X	X			X		X
ST3	X			X			X	X		

Les exemples pour lesquels aucune sous-tâche ne s'applique seront résolus par la méthode directe (celle-ci est supposée toujours applicable, par hypothèse). D'autres exemples que le 3 ou le 9 seront également finalement résolus par la méthode directe s'il s'avère que les sous-tâches proposées sont moins bonnes.

Les probabilités qui vont être utiles sont les probabilités qu'une sous-tâche soit applicable alors que d'autres ne le sont pas. En effet, on testera les sous-tâches les unes après les autres, dans un ordre donné, tant qu'il y aura échec, et à un moment donné (quand certaines sous-tâches auront échouées) il faudra connaître les probabilités de réussite de celles qui restent. Grâce à un tel tableau, il est possible de donner toutes les probabilités qui pourraient être nécessaires. Pour calculer la probabilité que ST_i soit applicable et que ST_{i_1}, ..., ST_{i_m} ne le soient pas (notée $p_i(i_1, ..., i_m)$, avec $p_i()$ *noté* p_i), il suffit de compter le nombre d'exemples pour lesquels il y a une croix pour ST_i et pas pour les autres.

Ainsi, $P_1(3) = 0,3$; les exemples concernés étant les 5, 6 et 10.

Donnons la liste de toutes les probabilités éventuellement nécessaires :

p1 = 0,4; p2 = 0,5; p3 = 0,4;

p1(2) = 0,2; p1(3) = 0,3; p2(1) = 0,3; p2(3) = 0,3; p3(1) = 0,3; p3(2) = 0,2;

p1(2,3) = 0,1; p2(1,3) = 0,1; p3(1,2) = 0,1;

Il y a peut-être moyen de faire ce calcul de manière incrémentale mais ce n'est pas le propos ici; cela sera approfondi ultérieurement.

Remarquons que dans le cas de sous-tâches exclusives — pour des données particulières, au plus une sous-tâche est applicable — le calcul des probabilités est grandement simplifié; une fois que l'on dispose des probabilités initiales, le calcul est trivial.

En effet, dans ce cas : $p_i(i_1, \ldots, i_n) = p_i / (1 - \sum_{k=1}^{n} p_{ik})$

Suivant les hypothèses de travail, ces probabilités seront calculées au préalable (si on dispose de temps de calcul non critique — ce qui est souvent le cas lors du développement ou l'acquisition des connaissances — et ce qui permettra de calculer ensuite plus rapidement les espérances d'utilité) ou au besoin (le temps est ressource critique en permanence ou la place mémoire est rare et donc on calcule de telles probabilités quand le calcul de l'espérance d'utilité le nécessite).

b. Les ressources consommées

La ressource consommée lors d'une recherche de solution est principalement le temps nécessaire pour tester si la sous-tâche choisie peut effectivement s'appliquer pour les paramètres effectifs de la tâche. On perd également du temps pour les différents calculs mais ceux-ci sont négligeables par rapport au temps de classification (rappelons que la classification est le

mécanisme qui permet de trouver les sous-tâches qui spécialisent une tâche donnée vu ses paramètres).

On peut supposer que les temps de tests sont identiques pour toutes les sous-tâches, afin de simplifier les calculs, mais ceci n'est pas nécessaire et on ne le supposera pas a priori.

c. La valeur des ressources intrinsèques

Les ressources consommées par l'exécution d'une méthode doivent être connues, sinon aucun choix n'est possible. Le problème est alors de calculer (ou plutôt d'estimer) les ressources consommées par une tâche complexe pour laquelle il existe plusieurs stratégies de résolution que l'on va appliquer suivant les cas. Cela dépend des hypothèses que l'on fait.

Il s'agit en effet de voir ce qui va être exécuté en ressources limitées ou critiques (ou limitées et critiques). Deux grands axes sont envisageables :

— certains calculs peuvent être faits en temps non critique et une fois que l'on a le problème et ses paramètres, on recherche rationnellement la solution (on tient compte du temps de recherche et du temps d'exécution). Il est donc possible de calculer par exemple l'utilité moyenne d'une tâche en fonction de l'utilité et des probabilités de ses sous-tâches possibles, calcul qui permet de choisir la solution la plus rationnelle. On peut relâcher dans ce cas l'hypothèse «un pas en avant» (section C).

— le temps est une ressource critique en permanence et il faut donc uniquement faire les calculs strictement nécessaires. Dans ce cas-là il est nécessaire de faire des estimations des fonctions d'utilité, ce qui va permettre de choisir la meilleure stratégie dans le cas général (section D).

C. Utilisation statique de la rationalité

1 Introduction

Cette méthode est valable lorsque l'on dispose de la fonction d'utilité bien avant de vouloir résoudre un problème; elle doit donc être modifiée exceptionnellement. Dans ce cas, on suppose que les calculs préliminaires décrits ci-dessous peuvent être faits sans consommer de ressources critiques (par exemple, la nuit sur une machine inactive) et cela permet de trouver, une fois ces calculs faits pour une fonction d'utilité donnée (ici tenant compte du temps d'exécution et de l'applicabilité), la solution d'un problème de manière rationnelle dès que l'on connait les paramètres de celui-ci.

On rappelle que chaque tâche admet une méthode permettant de la résoudre et s'appliquant quels que soient les paramètres. Comme la méthode est le premier moyen disponible pour résoudre une tâche, toutes les autres stratégies sont évaluées par rapport à cette méthode, cela permettant de décider s'il est préférable d'appliquer la méthode ou de poursuivre la recherche.

Nous présentons ci-dessous les caractéristiques de l'algorithme proposé ainsi que le plan de cette partie.

2 Idée générale

Pour résoudre de manière optimale un problème, il est nécessaire, lorsqu'un choix se présente, de connaître les stratégies qui maximisent la valeur nette d'utilité. Etant donné que la fonction d'utilité choisie est fonction de l'applicabilité et du temps d'exécution, on doit connaître le temps d'exécution moyen de chaque sous-tâche. On suppose d'autre part que l'on a établi de manière empirique, grâce à des jeux d'essais (représentatifs de l'utilisation qui va être faite du système de résolution de tâche), quelle est l'applicabilité des différentes stratégies permettant de résoudre une tâche donnée (comme expliqué au paragraphe B.4.a).

Il est possible de donner une approximation du temps d'exécution par celui de la méthode directe de la sous-tâche (selon l'hypothèse «un pas en avant»); cependant comme l'on dispose de ressources, il semble profitable de faire une approximation plus précise de ce temps en tenant compte des différentes possibilités de résolution de la tâche. Cela permettra de choisir plus justement la sous-tâche la plus utile, comme le montre un exemple pour lequel l'hypothèse conduit à un mauvais choix dans le cas général (au paragraphe B.1).

Afin de calculer le temps moyen de toutes les tâches, on dispose des temps (connus) d'exécution des méthodes qui sont les seules stratégies des feuilles de l'arbre de résolution et on présente ci-dessous un algorithme permettant de calculer le temps moyen pour exécuter une tâche à partir d'informations (probabilité d'applicabilité, temps d'exécution moyen calculé récursivement) concernant ses différentes sous-tâches possibles (paragraphe 3). Il permet également d'obtenir l'ordre dans lequel les sous-tâches devront être testées.

En appliquant récursivement l'algorithme décrit ci-dessous, on va propager la valeur de la ressource temps consommé des feuilles vers la racine de l'arbre de résolution du problème. Grâce à un arbre de résolution dont chaque nœud — représentant une tâche — possède un temps d'exécution (moyen) et l'ordre de test rationnel de ses différentes sous-tâches, il est alors facile de résoudre un problème de manière optimale (paragraphe 4). Nous verrons quelle est la sémantique que l'on peut donnée au système de tâches prenant en compte ces calculs d'utilité (paragraphe 5), puis l'idée d'implémentation de ce système amélioré (paragraphe 6). Dans ce qui suit, nous ne considérons que le temps comme critère d'une tâche mais il est possible d'imaginer d'autres voies (paragraphe 7). Nous terminerons cette section en évaluant la stratégie adoptée (paragraphe 8).

3 L'algorithme

On donne ci-dessous une vision globale de l'algorithme dont chaque étape est détaillée dans la suite.

a. Schéma général

L'algorithme permet de calculer le temps moyen pour effectuer la tâche T et l'ordre optimal pour tester les sous-tâches, connaissant les temps d'exécution de chacune de ses sous-tâches et leur probabilité d'applicabilité.

```
Temps_moyen(T : tâche; LST : liste de sous-tâches)
DEBUT
SI la tâche admet une seule stratégie, calcul simple;
SINON
    n = cardinal(LST); i = 1;
    TANT QUE (valeur_nette_positive et i≤n)
        Calcul de la valeur nette des sous-tâches restantes (i, ..., n) en
        tenant compte des sous-tâches déjà ordonnées;
        Attribution de l'indice i à la sous-tâche de valeur_nette maximale
        et élimination de celle-ci des sous-tâches restantes;
        i = i +1;
    FIN TANT QUE
    Calcul du temps moyen de T;
FIN SI
FIN
```

b. Calcul simple pour une tâche n'admettant qu'une seule sous-tâche

Si la tâche considérée admet une seule stratégie, le temps d'exécution de la tâche est facilement calculable en fonction du temps d'exécution de cette stratégie. On avait supposé qu'une tâche admettait toujours une méthode directe mais dans ce cas il suffit qu'il existe un moyen de la résoudre quels que soient les paramètres. On étend ici le travail de Mathias Chaillot en considérant les décompositions en séquence et en parallèle.

type de sous-tâche	temps d'exécution de la tâche
spécialisation	identique à celui de la sous-tâche
décomposition en séquence	somme des temps d'exécution
décomposition en parallèle	maximum des temps d'exécution
méthode	identique à celui de la méthode

c. Calcul de la valeur nette des sous-tâches

Précisons quelques notations :

Soit T la tâche à résoudre.

Soient m la méthode associée à T et t_m son temps d'exécution.

Soient ST_i, 1≤i≤n, les sous-tâches possibles de T (l'ensemble des tâches formant une décomposition constitue une seule sous-tâche).

Soient p_i, la probabilité que ST_i soit applicable, t_i son temps d'application et tt_i le temps pour tester si elle est applicable.

On se place au rang `m` de la boucle (c'est-à-dire que `m` sous-tâches sont déjà ordonnées). On calcule la valeur nette d'utilité — plus précisément dans notre cas l'espérance de gain en temps — de toutes les sous-tâches restantes en tenant compte des sous-tâches qui sont déjà ordonnées. Ces sous-tâches restantes n'auront une chance d'être appliquées que si toutes celles qui sont plus prioritaires ont échoué. La probabilité d'applicabilité à prendre en compte est donc la probabilité que *STi* s'applique et que *ST1*, .., *STm* ne s'appliquent pas ($p_i(1, ..., m)$).

$$V_m(ST_i) = p_i(1, ..., m) * (tm-t_i) - tt_i \text{ , pour } m+1 \leq i \leq n$$

$tm - t_i$: gain de temps en appliquant ST_i (temps : t_i) au lieu de m (temps : tm) si le test réussit (probabilité $p_i(1, ..., m)$).

La sous-tâche qui doit être testée en priorité est bien sûr celle de valeur nette maximale. On renumérote les sous-tâches de telle sorte qu'elle soit indicée par $m+1$.

En itérant ce calcul, on obtient un ordre de test $ST_1, ..., ST_n$ des sous-tâches.

Ajoutons que les sous-tâches sont effacées de l'ensemble des stratégies possibles pour T (pour cette fonction d'utilité donnée évidemment) dès que leur valeur nette est négative. En effet, leur valeur nette ne peut que décroître au fil du classement puisque leur domaine d'application se restreint au fur et à mesure que que d'autres sous-tâches échouent

Remarque : pourquoi recalculer les valeurs nettes à chaque pas de calcul ?

Prenons un exemple. On considère trois sous-tâches qui s'appliquent sur un réel et dont on donne les domaines (on peut supposer les temps d'exécution de ces trois sous-tâches identiques et équiprobabilité d'applicabilité, ce qui simplifie ce petit raisonnement) :

ST_1 :]- ∞, 0]

ST_2 : [-5, 2]

ST_3 : [1, 5]

A priori, il faudrait tester ST_1, ST_2 puis ST_3 (suivant la longueur de l'intervalle du domaine qui reflète l'applicabilité). Or, si le test de ST_1 échoue, il est alors préférable de tester ST_3 avant ST_2, contrairement à ce que l'on pouvait imaginer au premier abord. Cet exemple illustre le fait que les domaines d'applicabilité des sous-tâches ne sont pas indépendants et donc que toute connaissance sur la non applicabilité de l'une des sous-tâches influence les probabilités d'applicabilité des autres.

Dans le cas particulier important où les sous-tâches ont des domaines exclusifs, il suffit de faire le premier pas de ce calcul pour obtenir l'ordre de test puisque les différentes probabilités varient de manière proportionnelle au fil des échecs; les valeurs nettes peuvent donc être calculées une fois pour toute à l'aide des probabilités d'applicabilité. Ce cas est très intéressant vu qu'il nécessite beaucoup moins de données et de calculs que le cas général.

d. Calcul du temps moyen de résolution de T

Les sous-tâches vont être testées dans l'ordre $ST_1, ..., ST_n$ (n étant le nombre de tâches ayant une valeur nette d'utilité positive). On peut donc calculer le temps nécessaire pour choisir et exécuter chacune des sous-tâches (ou la méthode) en tenant compte du temps perdu pour tester les sous-tâches plus prioritaires qui ont échoué.

Si on applique la sous-tâche ST_i (probabilité $p_i(1, ..., i-1)$), on utilise un temps de :

$$t_i + tt_i + tt_1 + tt_2 + .. + tt_{i-1}$$

Si on applique la méthode (probabilité $1 - \sum_{i=1}^{n} p_i(1, ..., i-1)$, on utilise un temps de :

$$t_m + tt_1 + ... + tt_n$$

La probabilité que l'on applique la méthode est bien la somme de toutes les probabilités $p_i(1,..., i-1)$ puisque ces probabilités recouvrent des cas disjoints.

Le temps moyen est la moyenne pondérée de ces différents temps d'exécution :

$$t = \sum_{i=1}^{n}[p_i(1, ..., i-1) * (t_i + \sum_{j=1}^{i} tt_j)] + (1 - \sum_{i=1}^{n} p_i(1, ..., i-1)) * (tm + \sum_{i=1}^{n} tt_i)$$

On est ainsi capable d'obtenir le temps moyen d'exécution de la tâche *T*.

Par conséquent, cet algorithme permet de calculer les temps moyens de toutes les tâches formant l'arbre de résolution de n'importe quelle tâche générale et l'ordre de test optimal des stratégies pour chaque alternative.

4 La résolution d'une tâche

Pour trouver la solution optimale d'une tâche, c'est-à-dire celle qui est maximise la fonction d'utilité dépendant de la rapidité à la trouver et de son temps d'exécution, il suffit de parcourir l'arbre de résolution de la tâche et si un choix entre plusieurs sous-tâches se présente, il faut les tester suivant l'ordre pré-établi; la première sous-tâche qui réussit est celle qui est rationnelle.

Grâce aux calculs qui viennent d'être détaillés, ce mécanisme est rendu similaire à un arbre de décision pour lequel les valeurs associées à chaque nœud tiennent compte du temps nécessaire à la recherche, ce qui en fait un algorithme rationnel.

5 La sémantique du système de tâches rationnel

Les règles du système de tâches tiennent à présent compte de l'utilité qui a été calculée; il existe donc pour chaque tâche un ordre de test de ses différentes possibilités de résolution. Vu que la valeur nette d'utilité varie au cours de la résolution suivant les échecs, nous trouvons préférable d'associer l'ordre de test aux connaissances de la base qui seront alors de la forme :

`T'<<T, ordre` et `T<=T', ordre.`

Lors de la résolution, lorsqu'il y a conflit entre plusieurs règles, il faut activer celle qui traite la solution dont la valeur de l'ordre est la plus faible (c'est-à-dire celle qui est prioritaire). Toutes les autres possibilités, qui possèdent un ordre plus faible, sont des spécialisations dont le prédicat de vérification n'est pas vrai et donc qui ont échouées.

A chacune des quatre règles il faut associer une condition supplémentaire qui reflète qu'elle est la plus prioritaire des règles applicables (du point de vue de la rationalité, c'est elle qui doit être appliquée). Pour appliquer une règle, il faut donc vérifier qu'il n'existe aucune décomposition ou méthode qui admet un ordre plus petit (ce qui signifie qu'elle serait plus prioritaire) et qu'il n'existe aucune spécialisation applicable (c'est-à-dire dont le prédicat d'applicabilité est satisfait). Vu que l'on a pris comme hypothèse qu'il existe pour chaque tâche au moins une méthode applicable quels que soient les paramètres, une fois qu'une règle est choisie elle ne peut plus échouée lors du chaînage arrière (toutes les sous-tâches pourront être résolues). On notera ordre(T'), le rang de test de T' lorsque la tâche T est implicitement connue.

(R1) $\dfrac{T\,(x) <= mT\,(x'),\ o}{mT(x') :-\ T\,(x)}$

si $\forall T'$ tel que $T<=T',\ o'$ et $T' \neq mT$, on a : $o' \geq o$,
et $\forall T'$ tel que $T'<<T,\ o'$, on a : $o' \geq o$ ou $V_{T'}$ n'est pas vérifiée.

(R2) $\dfrac{P_1(x_1') :-\ T_1(x_1),\ldots,\ P_n(x_n') :-\ T_n(x_n),\ T(x)<=\ T_1(x_1'') < \ldots < T_n(x_n''),\ o}{P_1(\sigma(x_1')) < \ldots < P_n(\sigma(x_n')) :-\ T(\sigma(x))}$

pour $n \geq 1$, si $\exists\sigma$ unificateur tel que $\forall i \in [1, .., n]$, $\sigma(x_i) = \sigma(x_i'')$,
si $\forall T'$ tel que $T<=T',\ o'$ et $T' \neq T_1 < \ldots < T_n$, on a : $o' \geq o$,
et $\forall T'$ tel que $T'<<T,\ o'$, on a : $o' \geq o$ ou $V_{T'}$ n'est pas vérifiée.

(R3) $\dfrac{P_1(x_1') :-\ T_1(x_1),\ \ldots,\ P_n(x_n') :-\ T_n(x_n),\ T(x)<=\ T_1(x_1'') \,//\, \ldots \,//\, T_n(x_n''),\ o}{P_1(\sigma(x_1')) \,//\, \ldots \,//\, P_n(\sigma(x_n')) :-\ T(\sigma(x))}$

pour $n > 1$, si $\exists\sigma$ unificateur tel que $\forall i \in [1, .., n]$, $\sigma(x_i) = \sigma(x_i'')$,
si $\forall T'$ tel que $T<=T',\ o'$ et $T' \neq T_1 \,//\, \ldots \,//\, T_n$, on a : $o' \geq o$
et $\forall T'$ tel que $T'<<T,o'$, on a : $o' \geq o$ ou $V_{T'}$ n'est pas vérifiée.

(R4) $\dfrac{P(x_1') :- T_1(x_1),\ T_1(x_1'') <<\ T(x),\ o}{P(\sigma(x_1')) :- T(\sigma(x))} \quad V_{T_1}(\sigma(x_1))$

si $\exists\sigma$ unificateur tel que $\sigma(x_1) = \sigma(x_1'')$,
si $\forall T'$ tel que $T <= T',\ o'$, on a : $o' \geq o$,
et $\forall T'$ tel que $T' << T, o'$ et $T' \neq T_1$, on a : $o' \geq o$ ou $V_{T'}$ n'est pas vérifiée.

De cette manière, la tâche sera résolue par le moyen qui est rationnel : parmi ceux qui sont applicables (dans le cas bien précis du problème entièrement connu), on choisira le premier selon l'ordre établi au préalable par l'algorithme.

Au niveau de la sémantique dénotationnelle, on peut dire que le plan d'actions choisi est celui qui a été trouvé rationnellement, c'est-à-dire qui minimise la somme du temps de recherche et du temps d'exécution (en moyenne sur l'ensemble des problèmes).

6 L'implémentation

Tous les calculs que nous avons décrits ci-dessus peuvent être faits dans un module externe à Prolog; en effet, il n'est pas vraiment approprié à ce type de calcul même si cela est faisable (ce qui est montré dans le paragraphe D.3 où des calculs analogues sont faits). Afin de limiter les interactions entre Prolog et un module externe, on suppose que les sous-tâches sont exclusives; de cette manière, il suffira de faire les calculs nécessaires avant d'exécuter le programme Prolog (et une seule fois pour une base de faits donnée) sans aucune interaction ultérieure. Il faut ensuite transmettre au module en Prolog l'ordre de test des sous-tâches pour chaque tâche. Nous pourrions introduire un *ou* entre sous-tâches, mais nous persévérons dans la voie que nous avons choisie jusqu'à présent, laissant le mécanisme d'inférence traiter les choix.

Dans la base de faits contenant toutes les clauses, les clauses sont triées dans l'ordre fourni par l'algorithme précédent. Prolog, pour résoudre une tâche donnée, va donc tester les sous-tâches selon l'ordre indiqué; lors d'un échec, il traitera automatiquement le retour-arrière et choisira la sous-tâche suivante; il applique ainsi la première solution possible vu les données du problème, cette solution étant optimale vu la fonction d'utilité choisie (qui est de minimiser le temps de recherche et d'exécution). Cette fois-ci encore, peu d'implémentation est nécessaire vu ce dont se charge Prolog et surtout vu les données qui lui sont fournies.

Si la solution optimale est la seule qui nous intéresse, il suffit donc d'ajouter un CUT (!) à la fin de chaque clause qui implique le prédicat `solve`, ainsi en cas de réussite aucun retour arrière ne sera effectué et le système ne fournira donc qu'une seule solution. Si on laisse le programme dans son état actuel, il fournit les solutions de la meilleure à la pire, rationnellement parlant. Il faut bien voir que la résolution est rationnelle grâce au pré-traitement qui a permis de classer les clauses (de résolution de chaque tâche) dans l'ordre de test rationnel; la rationalité est

«contenue» dans l'ensemble de clauses plus que dans le méta-interprète lui-même qui ne sert qu'à garantir un certain ordre d'exécution selon l'ordre des clauses.

Cette implémentation semble peut-être «trop» triviale vu l'objectif de rationalité mais il ne faut pas oublier l'effort consenti au préalable pour trier les clauses (et qui pourrait être fait dans le même module Prolog).

7 D'autres critères

On pourrait envisager un algorithme similaire pour traiter d'autres critères que le temps. Il faut évidemment que l'on connaisse, comme pour le temps, ce qui est consommé par l'exécution des méthodes, par les recherches et les tests et que l'on puisse propager la valeur de ces ressources pour obtenir celles des tâches plus complexes. Cependant, pour que ce calcul ait un sens, il faut que des ressources soient nécessaires à la recherche de la meilleure solution, sinon la rationalité n'a plus de sens et un arbre de décision classique fait l'affaire: par exemple si une précision maximale est le seul objectif.

Considérons, par exemple, la place mémoire. Dans ce cas, la ressource consommée lors de la recherche est la mémoire qui est nécessaire pour effectuer les tests ou se «souvenir» des sous-tâches déjà testées. On peut supposer que la mémoire est restituée lorsque l'exécution d'une tâche ou d'un test est terminée, ce qui semble assez raisonnable. Simplement pour calculer la mémoire nécessaire pour une tâche n'admettant qu'une seule stratégie (à comparer au paragraphe C.3.b), il ne faut plus combiner les valeurs des sous-tâches de la même manière. Pour une décomposition en séquence, on prends le maximum de la mémoire nécessaire à chaque sous-tâche, alors que pour l'exécution en parallèle, il faut cumuler la place mémoire nécessaire. Tout le reste du traitement est rigoureusement identique (calcul de la place mémoire moyenne consommée par une tâche, évaluation de la valeur nette d'utilité).

On voit, grâce à cet exemple qu'il y a peu de modifications à apporter pour considérer une autre ressource. Il suffit que cette ressource soit consommée par la recherche et que ce soit une valeur additive, sinon aucun des calculs proposés ne s'appliquent. Nous voyons donc que même si nous nous sommes limités à étudier le temps comme ressource critique, nos propos recouvrent aussi toutes les ressources qui ont un comportement similaire.

8 Evaluation et critique de cette méthode

Cet algorithme permet de trouver la solution optimale (au sens de la rationalité) pour résoudre une tâche mais il nécessite des calculs préliminaires importants. Il peut être utilisé de manière satisfaisante lorsque la fonction d'utilité est parfaitement définie et que l'on dispose de ressources avant l'exécution proprement dite en ressources limitées.Mais, si la fonction d'utilité dépend du problème précis que l'on veut résoudre (le caractère urgent peut être plus ou moins important), il semble peu réaliste ou de faire les calculs avec les fonctions d'utilité possibles et de les mémoriser ou de trouver du temps à chaque fois pour refaire tous ces calculs. Il est

cependant particulièrement utile lorsque la fonction d'utilité est fixée une fois pour toute pour une tâche donnée.

Afin de pouvoir traiter les cas où les ressources sont critiques en permanence ou surtout pour lesquels la fonction d'utilité évolue au cours du traitement, nous présentons une utilisation dynamique de la rationalité.

D. Utilisation dynamique de la rationalité

Nous verrons comment inclure la rationalité dans le système de tâches pour que la résolution soit dynamique (paragraphe 1). Nous pourrons alors voir les problèmes posés pour établir la sémantique d'un tel système de tâches (paragraphe 2) avant de détailler son implémentation (paragraphe 3). Nous évaluerons alors une telle stratégie (paragraphe 4).

1 Sélection dynamique de sous-tâches

Les résultats de cette section sont largement inspirés du travail de Mathias Chaillot que nous avons déjà présenté (paragraphe B.1) et que nous avons généralisé sur certains points que nous préciserons. Afin de pouvoir appliquer un mécanisme semblable à TRAM, au système de tâches dans lequel les tâches sont non seulement spécialisables mais aussi décomposables, il faut étendre la sélection des méthodes aux décompositions. Si on suppose que les sous-tâches de décomposition s'appliquent sur l'ensemble du domaine de la tâche, il reste à choisir entre des solutions qui s'appliquent toujours (décomposition et méthode directe) et les spécialisations.

Pour cela, on propose de choisir parmi les sous-tâches toujours applicables, celle qui est la meilleure (suivant l'utilité intrinsèque puisqu'il n'y a pas de tests nécessaires), puis de calculer les espérances d'utilité des spécialisations en fonction de l'utilité intrinsèque de cette meilleure tâche disponible. Il reste à voir comment calculer l'utilité intrinsèque des décompositions. Si on fait une hypothèse similaire à celle d' «un pas en avant», cette valeur sera estimée en fonction de l'utilité de sa méthode. Une telle restriction semble nécessaire car l'évaluation exacte requiert un traitement analogue à celui utilisé statiquement (section C) et dont le sur-coût serait beaucoup trop important (par rapport à ce que cela pourrait faire gagner); on peut cependant envisager de la calculer plus précisément sur un pas de calcul en tenant compte de ses moyens de résolution autre que la méthode.

En fait, une fois la première partie effectuée, on est ramené au cas traité par M. Chaillot, à la seule différence que la méthode «directe» peut être une décomposition et donc que son utilité intrinsèque est seulement une valeur estimée. Cette utilité estimée doit donc minorer les utilités de toutes ses résolutions possibles si l'on veut être sûr de ne pas l'appliquer à tort ou alors être une valeur moyenne si l'on se base toujours sur le cas général.

a. Sélection de la meilleure stratégie sûre

Il faut tout d'abord déterminer l'utilité (intrinsèque) des décompositions, notée Ui. Rappelons que l'utilité est une valeur additive.

Pour l'instant, on se contente de donner une approximation de l'utilité de chaque composante d'une décomposition par l'utilité de la méthode associée et on en déduit l'utilité de la tâche de décomposition suivant l'opérateur :

$$Ui(T_1 < ... < T_n) = \sum_{i=1}^{n} Ui(T_i)$$

$$Ui(T_1 \,//\, ... \,//\, T_n) = \min_{1 \le i \le n} Ui(T_i)$$

On choisit parmi toutes les décompositions et la méthode, celle qui admet la plus grande utilité et on la note m'_T (comme étant la nouvelle méthode sure applicable); on a alors :

$$Ui(m'_T) = \max \{Ui(m_T), Ui(Dec_1), ..., Ui(Dec_n)\}$$

où Dec_1, ..., Dec_n sont les décompositions associées à la tâche T et m_T sa méthode directe.

b. Ordre de tests des spécialisations

Nous relâchons l'hypothèse posée dans le système TRAM qui assure que les sous-tâches spécialisables d'une tâche donnée forment un ensemble exclusif et exhaustif. Nous proposons des équations générales; celles-ci pourraient être simplifiées, en particulier les probabilités conditionnelles pourraient s'exprimer simplement en fonction des probabilités absolues d'applicabilité si on fait l'hypothèse de l'exclusivité.

Il s'agit à présent d'évaluer les spécialisations S_i par rapport à m'_T. Pour cela, on se base sur l'estimation de la valeur nette d'utilité :

$$V[S_i] = p_i * (Ui(S_i) - Ui(m'_T)) - RC_i$$

où tt_i est le temps nécessaire à tester que le problème peut être résolu par S_i

p_i est la probabilité que S_i s'applique

RC_i représente les ressources consommées par le test et qui sont perdues que le test échoue ou réussisse.

On teste en priorité la spécialisation qui a l'espérance d'utilité maximale et que l'on appelle S_1^*. Si elle échoue, il faut prendre une nouvelle décision. On recalcule les espérances de valeur nette d'utilité en tenant compte du fait que le premier test a échoué :

$$V[S_i] = p_i^1 * (Ui(S_i) - Ui(m'_T)) - RC_i$$

où p_i^1 est la probabilité que S_i s'applique sachant que S_1^* ne s'applique pas.

On ne tient pas compte des ressources qui ont été perdues lors du test de S_1^* puisque ces ressources sont perdues, à ce moment de la recherche, que l'on applique m'_T ou une spécialisation; cela ne doit donc pas intervenir pour choisir entre les deux.

Après l'échec de k tests, l'espérance d'utilité est calculée par :

$$V[S_i] = p_i^k * (Ui(S_i) - Ui(m'_T)) - RC_i \text{ , } i>k$$

p_i^k étant la probabilité que S_i s'applique sachant que S_j, pour $j \leq k$, ne s'appliquent pas.

c. Cas particulier du temps

Si la seule caractéristique à laquelle on s'intéresse est le temps, l'utilité intrinsèque d'une tâche ne dépend que son temps d'exécution; de plus, la différence d'utilité entre deux tâches est alors le gain de temps d'exécution. La fonction d'utilité peut donc être choisie comme la fonction opposée au temps d'exécution, ce qui donne pour les tâches de décomposition (en séquence ou en parallèle) :

$$Ui(T_1 < ... < T_n) = \sum_{i=1}^{n} Ui(T_i) = \sum_{i=1}^{n} -t_i = - \sum_{i=1}^{n} t_i$$

$$Ui(T_1 \,//\, ... \,//\, T_n) = \min_{1\leq i\leq n} Ui(T_i) = \min_{1\leq i\leq n} (-t_i) = - \max_{1\leq i\leq n} t_i$$

On vérifie ainsi des formules attendues :

temps d'exécution $(T_1 < ... < T_n) = \sum_{i=1}^{n} t_i$

temps d'exécution $(T_1 \,//\, ... \,//\, T_n) = \max_{1\leq i\leq n} ti$

La meilleure méthode sure m'_T est celle qui admet l'utilité maximale, c'est-à-dire dans ce cas précis, le temps d'exécution le plus faible $t_{m'}$. La maximisation de l'utilité revient bien à appliquer la méthode la plus rapide.

La valeur nette d'utilité admet dans ce cas une forme très simple :

$$V[S_i] = p_i * (t_{m'} - t_i) - tt_i$$

$t_{m'} - t_i$ étant le gain en temps d'exécution de S_i par rapport à m'.

tt_i est le temps consommé par le test d'applicabilité de S_i.

La stratégie choisie sera celle qui maximise la valeur nette d'utilité et donc qui minimise le temps d'exécution et de recherche.

Nous allons nous pencher plus précisément sur la sémantique du système de tâche utilisant la rationalité comme présentée dans cette section.

2 La sémantique du système de tâche rationnel (bis)

Dans le cas de la sélection dynamique que nous traitons, il n'est pas possible de donner des conditions syntaxiques associées aux règles comme nous l'avons fait pour la sélection statique pour laquelle il suffisait de «coller» un ordre aux données (paragraphe C.5). En effet, la sélection d'une règle ne peut se faire qu'en connaissant le déroulement préliminaire de l'exécution. Pour activer une règle agissant sur une décomposition ou une spécialisation dont la valeur nette d'utilité n'est pas maximale a priori, il faut savoir que les stratégies ayant une valeur nette d'utilité supérieure ne peuvent pas s'appliquer (elles conduisent à un échec car la condition d'applicabilité n'est pas satisfaite; nous avons déjà mentionné que c'est le seul cas possible d'échec que nous considérons puisqu'à toute tâche est associée une méthode applicable quels que soient les paramètres).

Afin de donner une sémantique à la résolution utilisant dynamiquement la rationalité, on peut soit utiliser une fonction d'ordre remise à jour au cours de la résolution d'une tâche (au fil des échecs de stratégies qui avaient une utilité espérée plus grande), soit s'intéresser directement à la valeur nette d'utilité (puisque ce calcul sera nécessaire pour trier les stratégies possibles d'une tâche). De plus la fonction d'ordre peut se ramener à une fonction booléenne, vu que le seul critère de sélection est de savoir si la tâche est celle qui maximise l'utilité espérée ou non; dans le premier cas elle sera exécutée, dans le deuxième, il faudra attendre un échec qui entraînera le recalcul de l'ordre. Le fait d'utiliser l'ordre plutôt que l'utilité «cache» un peu ce qui guide réellement le choix, c'est pourquoi nous préférons utiliser la valeur nette d'utilité. Dans le cas statique, le traitement est fait préliminairement et donc on sait précisément comment l'ordre a été établi et surtout il n'est pas réévalué. Ci-dessous nous présentons les règles proposées qui sont modifiées par l'ajout de conditions portant sur la valeur nette d'utilité que nous abrègerons par la suite en *utilité* :

utilité(stratégie) est la valeur nette d'utilité d'exécution de la stratégie (méthode, décomposition ou spécialisation) pour résoudre la tâche courante qui est implicite.

(R1) $\dfrac{T(x) \prec= mT(x')}{mT(x') \text{ :- } T(x)}$

si $\forall T'$ tel que $T \prec= T'$ et $T' \neq mT$, on a : *utilité(T') ≤ utilité(mT)*,
et $\forall T'$ tel que $T' \prec\prec T$, on a : *utilité(T') ≤ utilité(T)* ou $V_{T'}$ n'est pas vérifiée.

(R2) $\dfrac{P_1(x_1') \text{ :- } T_1(x_1), \ldots, P_n(x_n') \text{ :- } T_n(x_n),\ T(x) \prec= T_1(x_1'') \prec \ldots \prec T_n(x_n'')}{P_1(\sigma(x_1')) \prec \ldots \prec P_n(\sigma(x_n')) \text{ :- } T(\sigma(x))}$

pour n ≥1, si $\exists\sigma$ unificateur tel que $\forall i \in [1, .., n]$, $\sigma(x_i) = \sigma(x_i'')$,
si $\forall T'$ tel que $T \prec= T'$ et $T' \neq T_1 \prec \ldots \prec T_n$, on a : *utilité(T') ≤ utilité*$(T_1 \prec \ldots \prec T_n,)$,
et $\forall T'$ tel que $T' \prec\prec T$, *o'*, on a : *utilité(T') ≤ utilité*$(T_1 \prec \ldots \prec T_n,)$ ou $V_{T'}$ n'est pas vérifiée.

(R3) $\dfrac{P_1(x_1') :- T_1(x_1),\ \ldots,\ P_n(x_n') :- T_n(x_n),\ T(x) <= T_1(x_1'') // \ldots // T_n(x_n'')}{P_1(\sigma(x_1')) // \ldots // P_n(\sigma(x_n')) :- T(\sigma(x))}$

pour n >1, si $\exists\sigma$ unificateur tel que $\forall i \in [1, .., n]$, $\sigma(x_i) = \sigma(x_i'')$,
si $\forall T'$ tel que $T <= T'$ et $T' \neq T_1 // \ldots // T_n$, on a : *utilité(T') ≤ utilité($T_1 // \ldots // T_n$)*
et $\forall T'$ tel que $T' << T$, on a : *utilité(T') ≤ utilité($T_1 // \ldots // T_n$)* ou $V_{T'}$ n'est pas vérifiée.

(R4) $\dfrac{P(x_1') :- T_1(x_1),\ T_1(x_1'') << T(x)}{P(\sigma(x_1')) :- T(\sigma(x))} \qquad V_{T_1}(\sigma(x_1))$

si $\exists\sigma$ unificateur tel que $\sigma(x_1) = \sigma(x_1'')$,
si $\forall T'$ tel que $T <= T'$, on a : *utilité(T') ≤ utilité(T_1)*,
et $\forall T'$ tel que $T' << T$ et $T' \neq T_1$, on a : *utilité(T') ≤ utilité(T_1)* ou $V_{T'}$ n'est pas vérifiée.

De même en sémantique dénotationnelle, pour justifier la rationalité de la solution proposée, il faut faire intervenir le processus de résolution dans la représentation du monde, ce qui n'est pas envisagé pour l'instant et qui pose nombre de problèmes, notamment de régression infinie.

3 L'implémentation

On suppose que les sous-tâches de spécialisation sont exclusives; cette hypothèse permet de ne pas recalculer les probabilités d'applicabilité lorsqu'une sous-tâche (de spécialisation) se révèle inapplicable. On se place dans le cas des deux critères : applicabilité et temps d'exécution; la fonction d'utilité étant définie comme au paragraphe 1.c.

a. Modification des données

Les clauses qui définissent des spécialisations doivent indiquer le temps nécessaire au test ainsi que la probabilité d'applicabilité de la sous-tâche pour la résolution de la tâche courante. Vu que toute spécialisation admet des conditions, introduites par l'opérateur `$`, on propose la forme suivante :

```
T >> T' $[[<probabilité>, <temps de test>], <conditions>]
```

Les clauses définissant les décompositions ou les méthodes ne sont pas modifiées, vu qu'elles sont toujours applicables. Le temps d'exécution d'une tâche est estimé par celui de sa méthode directe associée.

b. Stratégie générale

Comme il s'agit de trouver le moyen de résolution rationnel, on recherche tout d'abord parmi les moyens surs celui dont le coût est minimal. Ce coût est évalué en tenant compte d'un seul pas de calcul supplémentaire, c'est-à-dire que le coût d'une décomposition est la somme des coûts des méthodes directes associées à chacune des sous-tâches qui la composent. Puis on

recherche parmi les spécialisations, celle qui admet l'espérance d'utilité positive la plus élevée, si elle existe. Il ne reste plus alors qu'à résoudre la tâche rationnelle, qui a été déterminée ainsi, grâce au prédicat `solve`, un peu remanié afin qu'il résolve une spécialisation, une décomposition ou une méthode directe en réutilisant récursivement la résolution rationnelle pour résoudre les nouvelles sous-tâches. Si le moyen qui était rationnel échoue, il est enlevé des possibilités de résolution et on recommence la recherche rationnelle sur un ensemble réduit.

Il existera toujours un moyen de résoudre une tâche, vu que l'on a supposé que toute tâche admet une méthode qui s'applique quelles que soient les données de la tâche.

Détaillons un peu plus chaque phase de cette résolution.

c. Recherche du moyen rationnel

Il faut tout d'abord récupérer la liste des moyens surs (S) et des moyens hypothétiques (H). Pour cela, on utilise le prédicat `bagof`, comme suit :

```
partition(T,S,H) :-
/* met dans S l'ensemble des X qui sont décompositions ou méthode de T */
    bagof_bis(X, clause(T <- X,true), S),
/* met dans H l'ensemble des spécialisations */
    bagof_bis(X,clause(T >> X,true), H).

/* bagof rend faux quand il ne trouve rien; bagof_bis rend la liste vide
*/
bagof_bis(X,C,S) :- bagof(X,C,S),!.
bagof_bis(X,C,[]).
```

Pour rechercher l'élément de S qui admet le coût le plus faible, utilisons le prédicat `meilleure_sure(liste, tache résultat, coût)` (le prédicat `cout_tache(T,C)` renvoie dans `C` le coût de la méthode directe associée à la tâche `T`):

```
meilleure_sure([T1|L], T1, C) :-
  cout_tache(T1,C),
  meilleure_sure(L,T2,D), C=<D.  /* T1 est de coût le plus faible */
meilleure_sure([T1|L], T2, D) :-
  cout_tache(T1,C),
  meilleure_sure(L,T2,D), C>D.   /* T2 est de coût le plus faible */
meilleure_sure([T1],T1,C) :-
  cout_tache(T1,C).  /* T1 est la seule de la liste */
```

Une fois que l'on a cette tâche et son coût, il est possible de calculer les espérances d'utilité de toutes les spécialisations et de rechercher celle qui admet la plus forte espérance positive. Si toutes les spécialisations ont une espérance négative ou s'il la tâche n'admet pas de spécialisation, le moyen rationnel de résolution est le moyen sûr.

```
  /* meilleure_cond(liste à tester, meilleure tache courante, son espérance
(0 au début), le coût de la méthode sure (pour calculer les espérances
successives), le résultat) */
  meilleure_cond([T|L],M,E,C,R) :-
    esp_utilite(T,C,E1), E1 =< E,  /* T est moins bonne que M */
    meilleure_cond(L,M,E,C,R),!.
  meilleure_cond([T|L],M,E,C,R) :-
    esp_utilite(T,C,E1), E1 > E,   /* T est meilleure que M */
    meilleure_cond(L,T,E1,C,R),!.
  meilleure_cond([],M,E,C,M).      /* il n y a plus de taches a tester, la
meilleure est la tache courante */
```

d. Résolution du moyen rationnel

A l'aide des prédicats définis ci-dessus, trouver le moyen rationnel est trivial; précisons la définition du prédicat `solve_rat(A,P,C)` qui résout rationnellement la tâche (le plan étant `P` et le coût `C`) :

```
  solve_rat(A,P,C) :-
      partition(A,S,H),
      solve_rat_bis(S,H,P,C).

  solve_rat_bis(S,H,P,C) :-
      meilleure_sure(S,M,D),
      meilleure_cond(H,M,0,D,T),
      solve(S,H,T,P,C).
```

On introduit le prédicat `solve_rat_bis` afin d'avoir en paramètre les listes des moyens surs et conditionnels puisque lors de l'échec du moyen rationnel courant, il faut itérer la recherche sur les moyens qui n'ont pas échoué; on aurait pu également éliminer le fait correspondant de la base mais on a jugé préférable de la laisser intacte au cours du traitement.

Le prédicat `solve` n'est plus chargé de choisir le moyen de résoudre une tâche, il se contente de traiter celui qui lui est fourni. Il doit également tenir compte des conditions et des affectations; il faut en effet vérifier ou affecter ce qui est indiqué. Il s'écrit :

```
  /* résolution d'une méthode */
  solve(S,H,A#C§B,A,C) :- B,!.
  solve(S,H,A#C,A,C) :- !.
  /* résolution d'une décomposition ou spécialisation. Une tache peut être
résolue ou alors doit être enlevée des taches possibles; cette distinction
est à faire pour chaque type de taches (avec ou sans conditions ou
affectations) */
  solve(S,H,A§B$[T,V],P,C) :- B,V, solvedec(A,P,C),!.
  solve(S,H,A§B$[T,V],P,C) :- solve_echec(S,H,A§B$[T,V],P,C),!.
  solve(S,H,A§B,P,C) :- B, solvedec(A,P,C),!.
  solve(S,H,A§B,P,C) :- solve_echec(S,H,A§B,P,C),!.
  solve(S,H,A$[T,V],P,C) :- V, solvedec(A,P,C),!.
  solve(S,H,A$[T,V],P,C) :- solve_echec(S,H,A$[T,V],P,C),!.
  solve(S,H,A,P,C) :- solvedec(A,P,C),!.
  solve(S,H,A,P,C) :- solve_echec(A,P,C),!.
  /* si une tache échoue :il faut prendre une solution moins bonne a priori
et donc commencer par enlever des possibilités la tache qui vient d échouer
et recommencer la recherche rationnelle sur les ensembles diminués */
  solve_echec(S,H,A,P,C) :-
      write('echec'), nl,
      enlever(A,S,H,S1,H1),
      solve_rat_bis(S1,H1,P,C).
```

Ni le prédicat `solvedec` ni les prédicats de conditions n'ont été modifiés .

e. Un exemple

Il serait trop long de traiter un exemple entier de résolution. Nous proposons de voir un pas d'un tel mécanisme de résolution; nous ne donnons que les clauses qui servent effectivement :

Les clauses de la base de faits :

```
  cuisiner_sauce(Q) <- m_cui_sauce(Q)#150.
  cuisiner_sauce([Q,T]) >> cuisiner_sauce_pour_pates(R) § aff(R,Q*100)
                           $ [[2,1],eq(T,'pates')] .
  cuisiner_sauce([Q,T]) >> cuisiner_sauce_pour_viandes(R) § aff(R,Q*20)
                           $ [[3,2],eq(T,'viande')].
  cuisiner_sauce_pour_pates(Q) <- m_cui_sau_pates(Q)#140.
  cuisiner_sauce_pour_viandes(Q) <- m_cui_sau_viande(Q)#135.
```

(les probabilités sont indiquées en dixième)

Cela équivaut à l'arbre de résolution :

```
cuisiner_sauce([q,t])<-
___m_cui_sauce([q,t])#150<-
++
___$eq(t,pates)
___cuisiner_sauce_pour_pates(q*100)<-
_______m_cui_sau_pates(q*100)#140<-
++
___$eq(t,viande)
___cuisiner_sauce_pour_viandes(q*20)<-
_______m_cui_sau_viande(q*20)#135<-
```

Voyons ce qui va se passer à l'appel : `solve_rat(cuisiner_sauce([3,'pates]),P,C).`
En premier lieu, les moyens de résolution sont partitionnés:

```
S = m_cui_sauce(Q)#150.
H = [cuisiner_sauce_pour_pates(R), cuisiner_sauce_pour_viandes(R)].
```

Le meilleur moyen sûr (et le seul dans ce cas, mais on pourrait envisager des décompositions) est `m_cui_sauce(Q)#150.`

On calcule les espérances d'utilité des deux moyens hypothétiques :

E(h1) = 0,2 * (150-140) - 1 = 1.

E(h2) = 0,3 * (150-135) - 2 = 2,5.

Le moyen rationnel est donc `cuisiner_sauce_pour_viandes(R)`. Lorsque l'on tente de le résoudre, il faut vérifier la condition : `eq('pates','viande')`, test qui échoue et donc la tentative courante de résolution est retirée de la liste H et on recommence la recherche. A ce moment-là, seule `cuisiner_sauce_pour_pates(Q)` est meilleure que la méthode sure; elle sera donc choisie et elle réussit, ce qui conduit finalement à la solution `m_cui_sau_pates(Q)#140.`

Cet exemple simple illustre le principe de résolution et montre que la résolution suit le principe de rationalité qui était fixé.

4 Evaluation d'une telle stratégie

La stratégie que nous avons présentée ci-dessus permet d'obtenir de façon rationnelle la résolution d'une tâche donnée. La fonction d'utilité peut également être modifiée en cours de traitement en modifiant simplement les prédicats `cout_tache` et `esp_utilite` en fonction des nouvelles contraintes.

Elle est cependant basée sur des approximations : on ne peut calculer que l'espérance d'utilité et l'utilité intrinsèque d'une sous-tâche est approximée par celle de sa méthode associée.

On pourrait également envisager un méta-méta-niveau afin de contrôler les décisions prises au méta-niveau, telles que l'hypothèse d'un seul pas en avant. Et cela pourrait continuer à l'infini, chaque niveau étant surveillé par le niveau supérieur. Cependant cela semble vite sans intérêt vu que les informations disponibles à un niveau et non au précédent sont peu nombreuses et donc qu'il est possible de faire des choix par défaut au premier ou deuxième niveau [Doyl90], plutôt que de réfléchir sur la façon de réfléchir.

E. Conclusion

Nous nous proposons de conclure cette partie en comparant les deux stratégies présentées et en donnant quelques idées permettant de tenir compte des ressources de manière plus générale.

1 Comparaison des méthodes statique et dynamique

Faire une sélection dynamique est nécessaire si l'état courant de la résolution influe sur la performance des tâches. Cela permet de moduler la résolution au vu des changements qui peuvent intervenir au cours du traitement : ainsi une tâche ne doit pas être résolue de la même manière selon la criticité du temps qui peut être connue plus ou moins précisément pendant la résolution. Comme nous l'avions déjà signalé, la méthode statique ne peut s'appliquer efficacement que si la fonction d'utilité ne varie pas ni au cours de la résolution et si possible pas d'une résolution à l'autre; dit plus généralement : si l'utilité ne dépend pas de la situation (pris au sens large) c'est-à-dire, par exemple, des ressources consommées ou des événements extérieurs.

Un autre avantage indéniable de la méthode dynamique est que l'on n'exécute que les calculs (d'utilité) nécessaires et que l'on ne s'encombre pas avec de nombreuses valeurs de probabilités ou d'utilité. Cependant tous ces calculs doivent être effectués pendant un temps qui est critique. L'avantage de la méthode statique réside principalement dans le fait que la résolution est plus sûre, en particulier à cause de l'évaluation du temps moyen d'exécution qui n'est pas approximé en utilisant l'hypothèse «un pas en avant».

Il s'agit donc de voir de quel type va être la résolution du problème, afin de choisir en toute connaissance de cause la stratégie la plus adaptée. Ce méta-méta-raisonnement est laissé au bon sens de l'utilisateur.

2 La limitation des ressources

Jusqu'à présent, nous n'avons pas détaillé le type de la fonction d'utilité. Nous avions évoqué que les ressources pouvaient être limitées ou critiques. Dans les exemples qui font intervenir le temps, il est considéré comme critique, c'est-à-dire que l'utilité décroît au fur et à

mesure que le temps passe. Cependant, on peut imaginer que l'exécution doive être limitée dans le temps; pour cela, il suffit de poser que l'utilité de toute action dont le temps d'application et de recherche dépasse ce temps critique est nul — pour celles dont le temps ne dépasse pas ce temps critique, l'utilité peut être dépendante du temps écoulé ou non.

En ce qui concerne le traitement statique de la rationalité, il suffit de poser que le temps d'exécution ou de recherche est infini dès qu'il dépasse le seuil critique, ce qui interdira de diriger la résolution vers des sous-tâches dont l'exécution et la recherche seront trop longues. Il faut alors, en fonction des recherches antérieures, connaître le temps restant; la méthode directe peut s'exécuter pendant ce temps-là (sinon ce n'est pas cette sous-tâche qui aurait été choisie) et il serait donc judicieux de rechercher une possible méthode plus rapide tout en se gardant le temps d'exécuter la méthode directe si les recherches se révèlent infructueuses trop longtemps.

Lors de l'utilisation dynamique de la rationalité, la fonction d'utilité n'est pas explicitée; il suffit d'ajouter une contrainte sur celle-ci : être nulle pour toutes les tâches qui nécessitent des ressources trop importantes (qui dépassent le seuil toléré). La valeur nette d'utilité ne pourra donc être positive que pour des tâches qui pourront se résoudre avec les ressources imparties et la résolution se fera donc conformément aux contraintes posées sur les ressources. Dans cette approche également il est possible de combiner ressources critiques et limitées.

3 Et si l'on considérait plusieurs ressources …

Il faut tout d'abord remarquer que l'on tient compte en fait de deux critères dans tout ce qui a été décrit : l'applicabilité et une ressource (le temps dans les exemples). Afin de considérer plusieurs ressources (le temps et la place mémoire par exemple), on peut envisager d'établir une relation d'ordre entre elles, de telle sorte que l'on ne tienne compte de la deuxième ressource qu'en cas d'égalité sur la première, et de la troisième seulement si …. On peut également imaginer une fonction d'utilité qui combine les différentes ressources, en leur accordant à chacune une importance relative : un gain de 10 octets est équivalent à un gain d'une seconde, …. Il faut bien voir que cette fonction d'utilité dépend non seulement de plusieurs ressources, mais également de la situation dans laquelle le système se trouve, situation qui peut modifier en cours de traitement la manière de tenir compte des paramètres dans l'évaluation de l'utilité. On peut imaginer

Le point qui reste le plus intéressant à notre avis est la prise en compte d'une ressource et d'un critère de qualité comme la précision ou le prix. On peut alors pour simplifier l'expression de la fonction d'utilité (mais peut-être pas son estimation effective), tout ramener au coût, c'est-à-dire que l'on estime la valeur monétaire des ressources (coût du temps de calcul, coût de la place mémoire, …) et la valeur des critères de qualité (ce que fera gagner l'obtention de la dixième décimale par rapport à la neuvième, …). Il suffit alors de soustraire le coût des ressources de la valeur que l'on accorde à la qualité pour obtenir une fonction d'utilité qui permettra le compromis le plus intéressant entre les ressources et la qualité.

Toutes ces remarques montrent l'importance et la diversité des cas qu'un système rationnel permet de traiter en modifiant uniquement l'expression de l'utilité (qui n'est pas forcément trivial à déterminer dans les cas complexes).

Conclusion et perspectives

Le projet a permis de fusionner deux conceptions des tâches, celles de Jutta Willamowski et de Mathias Chaillot. Nous avons tout d'abord établi une sémantique opérationnelle d'un système de tâches dont l'expressivité est similaire au système SCARP, puis nous avons introduit la rationalité dans le système afin que l'exécution tienne compte de la notion d'utilité. Nous avons donc établi la sémantique d'un moteur de tâches qui est une extension des deux systèmes SCARP et TRAM; le premier a été complété par l'introduction de la rationalité et le second par l'introduction de tâches décomposables et le relâchement de certaines hypothèses. Nous avons validé notre approche en réalisant une implémentation immédiate de la sémantique proposée. Cette fusion a nécessité des modifications importantes du travail de Mathias Chaillot afin d'introduire la décomposition.

Cependant ce qui a été effectué n'est qu'une première étape, vu les limites qui ont été posées. Un premier point gênant subsiste : l'hypothèse «un pas en avant». Il faudrait trouver un moyen de relâcher cette hypothèse afin d'accorder moins d'importance à la méthode directe et pour que le choix pris soit un peu plus conforme aux possibilités, en tenant compte de possibles étapes ultérieures de calcul. Le traitement que cela impose doit cependant être relativement restreint afin de consommer peu de ressources critiques. Des questions de méta-méta-niveau surgissent : qu'apporteront des calculs supplémentaires? Quelle est la valeur d'une plus grande précision sur les ressources consommées? Quelles sont les ressources qu'il faut allouer à ces calculs afin de maximiser le rapport «gain / ressources consommées»? Cependant, il semble illusoire de ne vouloir faire aucune approximation, vu que toutes les techniques qui sont confrontées à des choix multiples font des hypothèses restrictives afin d'éviter une explosion combinatoire : en programmation des jeux, par exemple, la profondeur de recherche est limitée malgré des algorithmes qui permettent de couper certaines branches de recherche.

Par ailleurs, le travail actuel est encore relativement limité du point de vue des connaissances qu'il permet de modéliser; en effet il s'est focalisé sur les mécanismes de tâches, sans se soucier des connaissances du domaine. Cette approche a permis d'isoler les grands principes de résolution et donc de clarifier notablement les concepts et l'implémentation (le mécanisme d'inférence s'implémente directement en Prolog). Cependant, il est nécessaire de se concentrer maintenant sur des problèmes plus proches de la réalité en réintroduisant les connaissances du domaine dans notre modélisation. C'est pourquoi, on propose de s'orienter dans un premier temps vers des représentations de connaissances à objets et notamment vers une implémentation en LIFE [Aït-K91, Aït-K94], langage qui offre les fonctionnalités de Prolog ainsi que les notions d'objets (par le biais des ψ-termes) et de subsomption entre termes (qui permet un

mécanisme de classification sur les objets). Il sera alors possible d'introduire des objets dans l'implémentation proposée, sachant que l'implémentation actuelle de notre mécanisme peut être transféré directement en LIFE[1].

La perspective actuelle est double. Il faudrait établir précisément la sémantique dénotationnelle d'un système rationnel; il s'agit principalement d'ajouter l'information que le système trouve la résolution la plus utile pour atteindre le but escompté et pour cela de quantifier l'utilité de tous ses éléments, c'est-à-dire de voir les ressources consommées et les produits qui peuvent également être considérés comme des ressources gagnées (temps économisé à l'exécution ou informations acquéries). Il reste donc un effort de travail à fournir pour parvenir à une base de connaissances rationnelle.

D'un autre côté, par rapport à une problématique plus globale des systèmes dits «temps réel», celui que nous avons décrit se place dans le cadre de la résolution basée sur des méthodes multiples. Ce genre de stratégie présente l'inconvénient d'être non interruptible, c'est-à-dire que le traitement ne fournira aucune solution s'il est interrompu trop tôt (à cause d'une mauvaise évaluation du temps disponible par exemple). C'est pourquoi, certains auteurs privilégient le raisonnement progressif qui fournit rapidement une première solution et qui enchaîne ensuite d'autres méthodes qui fourniront des résultats de qualité croissante. Dans le même ordre d'idée, les algorithmes «anytime» peuvent être interrompus à tout moment (d'où leur dénomination) puisqu'ils procèdent par itérations successives qui affinent les résultats. Ne citons comme exemple que les méthodes numériques d'approximation qui évaluent leur résultat par le calcul d'une suite x_n convergente. Nous pensons qu'il est possible de résoudre une tâche en procédant par étapes successives, ce processus étant interruptible, c'est-à-dire que si le temps imparti est écoulé, on peut délivré à l'utilisateur un plan intermédiaire qui fournit une première ébauche de résolution mais qui contient encore des tâches non élémentaires. De cette manière la résolution pourra être reprise ultérieurement pour être affinée.

---

[1] à l'exception de deux points :

— LIFE proposant un grand nombre d'opérateurs, il faut modifier le nom actuel de nos opérateurs.

— Les prédicats bagof et append de Prolog existent en LIFE comme fonction et cela entraîne quelques modifications lors de leur appel.

Références

[Aït-K91] Hassan Aït-Kaci. An overview of LIFE. *LNCS. Vol. 504*. pp 42-58. 1991.

[Aït-K93] Hassan Aït-Kaci & Andreas Podelski. Towards a meanings of LIFE. Journal of Logic Programming. Vol. 16 n°3-4. pp195-234. 1993.

[Aït-K94] Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Poleski & Peter Van Roy. The Wild LIFE Handbook. *PRL Research Report. DEC, Paris Research Laboratory*. Rueil-Malmaison. 1994.

[Chai93] Mathias Chaillot. Une architecture de contrôle réactif pour la résolution coopérative de problèmes. *Thèse, INPG*. Grenoble. 1993.

[Char94] François Charpillet & Philippe Théret. IA et temps réel. *Bulletin de l'AFIA. n°17*. 1994.

[Clém90] Véronique Clément & Monique Thonnat. Integration of image processing procedures : OCAPI, a knowledge-based approach. *Rapport de recherche INRIA n°1307*. Sophia-Antipolis. 1990.

[Dean91] Thomas Dean. Decision-theoretic control of inference for time-critical applications. *International journal of intelligent systems. Vol. 6 n°4*. pp 417-441. 1991.

[Doyl89] Jon Doyle. Rational control of reasoning in Artificial Intelligence. *LNCS. Vol. 465*. pp19-48. 1989.

[Doyl90] Jon Doyle. Rationality and its role in reasoning. *Proceedings of the 8th AAAI conference*. pp1093-1100. Boston. 1990.

[Doyl91] Jon Doyle & Ramesh S. Patil. Two theses of knowledge representation : language restrictions, taxonomic classification and the utility of representation services. *Artificial Intelligence. Vol. 48 n°3*. pp 261-297. 1991.

[Etzi91] Oren Etzioni. Embedding decision-analytic control in a learning arcitecture. *Artificial Intelligence. Vol. 49*. pp 129-159. 1991.

[Fagi87] Ronald Fagin & Joseph Y. Halpern. Belief, awareness and limited reasoning. *Artificial Intelligence. Vol. 34 n°1*. pp40-41. 1987.

[Good52] I.J. Good. Rational decisions. *Journal of the Royal Statistical Society B (14)*. pp 107-114. 1952.

[Horv88] Eric J. Horvitz. Reasoning under varying and uncertain resource constraints. *Proceedings of the 7th AAAI conference*. pp 111-116. Saint Paul. 1988.

[Horv89] Eric J. Horvitz. Rational metareasoning and compilation for optimizing decisions under bounded resources. *Computational Intelligence. Vol. 2* (actes of the 2nd international conference on computational intelligence). pp205-216. 1989.

[Howa67] Ronald A. Howard. Value of information lotteries. *IEEE Transactions on Systems Science and Cybernetics. Vol. 3 n° 1*. 1967.

[Leve87] Hector J. Levesque & Ronald J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence. Vol. 3*. pp78-93. 1987.

[Mame91] Nuno J. Mamede & Joao P.Martins. RR- An Intelligent Resource-Bounded reasoner. *Artificial Intelligence and Simulation of Behaviour 91*. Edited by Luc Steels and Barbara Smith. Springer-Verlag. pp 36-47. 1991.

[Masi89] Gérard Masini, A. Napoli, D. Colnet, D. Léonard & K. Tombre. *Les langages à objets*. Interéditions. Paris. 1989.

[Mass92] Marcel Masseron, ChristopheTollu & Jacqueline Vauzeilles. Planification et logique linéaire. *Revue d'intelligence artificielle. Vol. 6 n°3.* pp 285-311. 1992.

[Medi93] Claudine Médigue, J Willamowski, O. Schmeltzer, P. Uvietta, F. Rechenmann, F. Chevenet, G. Perrière & C. Gautier. Modeling tasks for problem solving in molecular biology. *Actes of the Workshop 'Artificial Intelligence and Genome' IJCAI.* pp 67-76. Chambéry. 1993.

[Mutch86] David Mutchler. Optimal allocation of very limited search resources. *Proceedings of the 5th AAAI conference*. pp 467-471. Philadelphia. 1986.

[Nils82] Ulf J. Nilsson. *Principles of Artificial Intelligence*. Chapter 7-8, pp 275-360. Springer-Verlag. Londres. 1982.

[Nils90] Ulf J. Nilsson & Jan Matuszyuski. *Programming and Prolog*. Wiley, Chichester. Chapter 8 pp151-165. 1990.

[Poll93] John L. Pollock. The phylogeny of rationality. *Cognitive Science. Vol. 17 n°4.* pp563-588. 1993.

[Rao94] Anand S. Rao. Means-end plan recognition - Towards a theory of reactive recognition. *Actes of the 4th Knowledge Representation*. pp 497-508. Bonn. 1994.

[Rech85] François Rechenmann. SHIRKA: Mécanismes d'inférence sur une base de connaissances centrée-objet. *5ième congrès Reconnaissance de Formes et Intelligence Artificielle*. pp 1243-1254.1985.

[Regn90] Pierre Regnier. Planification : historique, principes, problèmes et méthodes (de GPS à ABTWEAK). *Rapport IRIT N°90/22/R*. IRIT Toulouse.1990.

[Rous88] Bertrand Rousseau. Vers un environement de résolution de problèmes en biométrie. *Thèse université Claude Bernard*. Lyon. 1988.

[Russ89] Stuart Russel & Eric Wefald. On optimal game-tree search using traditionnal meta-reasoning. *Proceedings IJCAI*. pp334-340. Detroit. 1989.

[Russ91] Stuart Russel & Eric Wefald. Principles of metareasoning. *Artificial Intelligence. Vol. 49*. pp361-395. 1991.

[Shug80] Steven M. Shugan. The cost of thinking. *Journal of consumer Research. Vol. 7*. pp 99-111. 1980.

[Ster86] Leon Sterling & Shapiro. *The Art of Prolog* (traduit par Michel Eytan, Masson, 1990). The MIT press. Chapter 19. pp301-328. 1986.

[Tan93] Shong Ye Tan. Types et classes dans une base de connaissances à objets. *DEA INPG/UJF*. Grenoble. 1993.

[Thie92] Sylvie Thiébaux & Joachim Hertzberg. A semi-reactive planner based on a possible models action formalisation. *Tasso-Report n°37, GMD*. Sankt Augustin. 1992.

[Thie93] Sylvie Thiébaux, Joachim Hertzberg, William Shoaff & Moti Schneider. A stochastic model of actions and plans for anytime planning under uncertainty. *TR-93-027.International Computer Science Institute*. Berkeley. 1993.

[Will93] Jutta Willamowski & François Chevenet. Modeling methodological knowledge in data analysis. *Artificial Intelligence, Expert Systems, Natural Language.* pp211-220. Avignon. 1993.

[Will94] Jutta Willamowski. Modélisation de tâches pour la résolution coopérative de problèmes. *Thèse UJF*. Grenoble. 1994.

Annexes

Nous donnons en annexe le programme complet du système de tâches écrit en Prolog, ainsi que des exemples de résolution à partir d'une certaine base de faits. Les exemples traiteront toujours de recettes de cuisine, ce domaine étant celui dont nous nous sommes servis jusqu'à présent; de plus, le projet n'avait pas l'ambition de fournir un sytème opérationnel pour des applications réelles mais seulement de poser les bases de formalisation de l'utilisation des tâches.

Le programme

```
/* SYSTEMES DE RESOLUTION DE TACHES */

/* Cet ensemble de règles est un méta-interprète Prolog permettant de
résoudre des taches de manière rationnelle, c'est-à-dire en tenant compte
d'une fonction d'utilité pour choisir entre les différentes possibilités */

/* Voici la signification des différentes notations utilisées pour
représenter les tâches :

A << B           A spécialise B.
A <- m#cost      A admet une méthode de cout cost
A <- B // C.     A se décompose en B et C en parallèle
A <- B < C.      A se décompose en B et C en séquence

Une tache T conditionnelle (i.e. de spécialisation) est sous la forme :
<nom>$[[<proba>,<temps test>],<condition d'application]
proba est la probabilite que la condition d'application soit remplie
(évaluée à partir d'exemples significatifs)
et le temps de test est celui nécessaire pour voir si la condition est
satisfaite (on pourrait envisager une valeur par défaut s'il ne
dépend pas des sous-tâches).

Pour certaines sous-taches il est necessaire de calculer les paramètres
en fonction de ceux de la tache; pour cela on introduit l'opérateur §
qui regroupe l'ensemble des affectations nécessaires :
T$[<liste d'affectations>]

Les decompositions et la methode directe n'ont pas de condition
d'application */

:- op(10,xfy,<-).
:- op(10,xfy,>>).
:- op(8,xfy,$).
:- op(7,xfy,§).
:- op(5,xfy,<).
:- op(5,xfy,//).
:- op(5,xfy ,#).


 /* PRE TRAITEMENT POUR LE CALCUL DES MEILLEURS STRATEGIES */
/*********************************************************/

 /* PARTITION des strategies qui permettent de résoudre T en sures S
(méthode directe et décompositions) et hypothetiques H (celles qui
proviennent de spécialisations) */

partition(T,S,H) :-
/* met dans S l'ensemble des X qui sont décompositions ou methode de T */
      bagof_bis(X, clause(T <- X,true), S),
/* met dans H l'ensemble des spécialisations */
      bagof_bis(X,clause(T >> X,true), H).

/* bagof rend faux quand il ne trouve rien et bagof_bis rend la liste vide
*/
bagof_bis(X,C,S) :- bagof(X,C,S),!.
bagof_bis(X,C,[]).
```

```
/* calcul du COUT d'une tache par approximation single step :
toutes les sous-taches ont leur cout approxime par le cout de leur methode
*/
cout_tache(T$A$V,C) :- cout(T,C),!.
cout_tache(T$A,C) :- cout(T,C),!.
cout_tache(T$V,C) :- cout(T,C),!.
cout_tache(T,C) :- cout(T,C),!.

cout(T#C,C) :- !.
cout(T1 // T2, V) :- cout(T1,V1), cout(T2,V2), max(V1,V2,V),!.
cout(T1 < T2, V) :- cout(T1,V1), cout(T2,V2), V is V1 + V2,!.
cout(T,C) :- clause(T <- M#C,true),!.
cout(T,C) :- clause(T <- M#C$V,true).


/* meilleure_sure(L,T,C) recherche la STRATÉGIE SURE T
(L est la liste des stratégies sures) de cout C le plus faible
On est sur qu il en existe au moins une (la methode) */

meilleure_sure([T1|L], T1, C) :- cout_tache(T1,C),
      meilleure_sure(L,T2,D),
      C=<D.

meilleure_sure([T1|L], T2, D) :- cout_tache(T1,C),
      meilleure_sure(L,T2,D),
      C>D.

meilleure_sure([T1],T1,C) :- cout_tache(T1,C).

/* quelques fonctions utiles */

/* extrait la proba et le temps de test d'une tache T conditionnelle */
proba(T$[[P,TT],V],P).

temps_test(T$[[P,TT],V],TT).

/* fournit l esperance d utilite de l'application de T au lieu de
l'application de la methode disponible de cout Tm */
esp_utilite(T,Tm,R) :-
      proba(T,P1),
      cout_tache(T,C1),
      temps_test(T,TT),
      R is (P1*(Tm-C1)/10 - TT).


/* recherche de la STRATÉGIE CONDITIONNELLE R meilleure que la stratégie
sure de moindre cout (M,C). Toutes les stratégies sont évaluées en fonction
du cout de la methode sure C.
meilleure_cond(liste à tester,
                  meilleure tache courante,
                  son esperance (0 initialement),
                  le cout de la methode directe(pour calculer les espérances
                   successives),
                  le resultat) */

meilleure_cond([T|L],M,E,C,R) :-
      esp_utilite(T,C,E1),
      E1 =< E,    /* T est moins bonne que M */
      /* on continue la recherche dans le reste de la liste */
      meilleure_cond(L,M,E,C,R),!.

meilleure_cond([T|L],M,E,C,R) :-
```

```
      esp_utilite(T,C,E1),
      E1 > E,     /* T est meilleure que M */
      /* on recherche dans le reste de la liste une meilleure que T*/
      meilleure_cond(L,T,E1,C,R),!.

/* il n y a plus de taches a tester, la meilleure est la courante */
meilleure_cond([],M,E,C,M).


 /* LA RESOLUTION RATIONNELLE */
/*****************************/

/* on choisit la sous-tache de maniere rationnelle et on la resout
recursivement */

/* si la tache est une methode, c'est fini */
solve_rat(M#C, M#C,C) :-
      write('Résolution par application de la méthode : '),
      write(M),nl,nl,!.

/* sinon, on partitionne et on resoud sur cette partition …*/
solve_rat(A,P,C) :-
      partition(A,S,H),
      solve_rat_bis(S,H,P,C).

/* … et on recherche la tache courante rationnelle */
solve_rat_bis(S,H,P,C) :-
      write('Les moyens de la résoudre sont :'), nl,
      write('S=  '), write(S),nl,
      write('H=  '),write(H),nl,
      /* en trouvant la stratégie sure de moindre cout */
      meilleure_sure(S,M,D),
      /* et en recherchant ensuite parmi les conditionnelles s'il en existe
            une meilleure */
      meilleure_cond(H,M,0,D,T),
      write('La tâche rationnelle à résoudre est : '),write(T),nl,
      /* on resoud la tache rationnelle */
      /* si cela echoue, solve va enlever cette tache de celles qui peuvent
etre choisies et ré-appelle solve_rat_bis; on passe S et H en paramètres,
afin de pouvoir les mettre à jour lorsq'une tache a echoue */
      solve(S,H,T,P,C).


 /* LA RESOLUTION */
/*****************/

/* la tache est methode elementaire, elle peut admettre une affectation */
solve(S,H,A#C§B,A,C) :-
      B,
      write('Résolution par application de la méthode : '),
      write(A),nl,nl,!.

solve(S,H,A#C,A,C) :-
      write('Résolution par application de la méthode : '),
      write(A),nl,nl,!.

/* la tache est une decomposition ( < n≥1; // n>1)
les spécialisations sont traitees comme des decompositions à un element
avec une verification. Pour chaque type de taches, soit on peut la
resoudre, soit il faut l'enlever des taches possibles */
solve(S,H,A§B$[T,V],P,C) :- B,V, solvedec(A,P,C),!.
solve(S,H,A§B$[T,V],P,C) :- solve_echec(S,H,A§B$[T,V],P,C),!.
solve(S,H,A§B,P,C) :- B, solvedec(A,P,C),!.
```

```
solve(S,H,A§B,P,C) :- solve_echec(S,H,A§B,P,C),!.
solve(S,H,A$[T,V],P,C) :- V, solvedec(A,P,C),!.
solve(S,H,A$[T,V],P,C) :- solve_echec(S,H,A$[T,V],P,C),!.
solve(S,H,A,P,C) :- solvedec(A,P,C),!.
solve(S,H,A,P,C) :- solve_echec(A,P,C),!.

/* si une tache echoue : il faut prendre une solution moins bonne a priori;
on enleve des possibilites la tache qui vient d echouer */
solve_echec(S,H,A,P,C) :-
      write('......... ECHEC .........'),
      write(' on reprend la résolution de la tâche courante'), nl,nl,
      enlever(A,S,H,S1,H1),
      solve_rat_bis(S1,H1,P,C).


 /* VERIFICATION DES CONDITIONS */
/*******************************/

/* le prédicat etl autorise les listes de conjonctions et le prédicat et
est binaire */
etl([A|L]):- A, etl(L).
etl([]).
et(A,B) :- A,B.
ou(A,B) :- A.
ou(A,B) :- B.

aff(Q,R) :- Q is R.
eq(Q,R) :- Q = R.
sup(Q,R) :- Q > R.
inf(Q,R) :- Q < R.
supe(Q,R) :- Q >= R.
infe(Q,R) :- Q =< R.


 /* DECOMPOSITION EN PARALLELE */
/*******************************/
/* resolution par decomposition en parallele : B peut se décomposer en n
sous-taches en parallele on boucle donc sur l'ensemble des opérateurs // */

solvedec(T1//T2,P,C) :-
      solvepar(T1//T2,P,C).

solvepar(T1//T2,P,C) :-
      solvepar(T1,P1,C1),
      solvepar(T2,P2,C2),
      max(C1,C2,C),
      appendpar(P1,P2,P).

/* la tache ne contient plus d'operateur //, on la resoud récursivement par
solve_rat */
solvepar(T,P,C) :-
      write('La sous-tâche à résoudre est :'), write(T), nl,
      solve_rat(T,P,C).

 /* DECOMPOSITION EN SEQUENCE */
/*******************************/
/* resolution par decomposition en sequence : B peut se décomposer en n
sous-taches en sequence, on boucle donc tant qu'il y a des opérateurs <
dans la tache */

solvedec(T1<T2,P,C) :-
      solveseq(T1<T2,P,C).
```

```
solveseq(T1<T2,P,C) :-
      solveseq(T1,P1,C1),
      solveseq(T2,P2,C2),
      C is C1 + C2,
      appendseq(P1,P2,P).

/* la tache ne contient plus d'operateur <, on la resoud par solve_rat */

solveseq(T,P,C) :-
      write('La sous-tâche à résoudre est :'), write(T), nl,
      solve_rat(T,P,C).

/* B est simplement B <- T */
solvedec(T,P,C) :-
      solve_rat(T,P,C).


 /* FONCTIONS AUXILIAIRES */
/************************/
append([],X,X) :- !.
append([X|Y],Z,[X|W]) :- append(Y,Z,W).


/* Concatène deux liste d'événements en séquence */
appendseq(seq(A),seq(B),seq(C)) :- !, append(A,B,C).
appendseq(seq(A),B,seq(C)) :- !, append(A,[B],C).
appendseq(A,seq(B),seq(C)) :- !, append([A],B,C).
appendseq(A,B,seq([A,B])) :- !.


/* Concatène deux liste d'événements en parallèle */
appendpar(parallel(A),parallel(B),parallel(C)) :- !, append(A,B,C).
appendpar(parallel(A),B,parallel(C)) :- !, append(A,[B],C).
appendpar(A,parallel(B),parallel(C)) :- !, append([A],B,C).
appendpar(A,B,parallel([A,B])) :- !.


/* enleve T de S ou de H (il n appartient qu'à une liste) et met les listes
transformees dans S1 et H1 */
enlever(T,S,H,S1,H) :- moins(T,S,S1),!.
enlever(T,S,H,S,H1) :- moins(T,H,H1).

/* moins(T,L,L1) : L1 est la liste L a laquelle on a enleve T */
moins(T,[T|S],S) :- !.
moins(T,[T1|S],[T1|S1]) :- moins(T,S,S1).

/* MAX */
max(X,Y,X) :- X>Y, !.
max(X,Y,Y).
```

La base de faits

Rappelons les notations employées pour définir une tâche et ses décompositions ou spécialisations :

une tâche est un nom suivi d'un ensemble d'arguments (ou liste d'arguments).

`T >> T'` signifie que la tâche `T'` est plus spécifique que la tâche `T`.

`T <- T'` signifie que `T'` permet de résoudre `T`; `T'` peut être uen tâche simple ou une décomposition.

`T1 // T2` signifie que `T1` et `T2` doivent être exécutées en parallèle.

`T1 < T2` signifie que `T1` et `T2` doivent être exécuter en séquence.

De plus, on peut associer à une sous-tâche une liste d'affectations (pour déterminer ses paramètres en fonction de ceux de la tâche) grâce à l'opérateur `§` :

```
<nom de la tâche> § <affectations>
```

Pour les tâches de spécialisation, on indique la probabilité qu'elle soit applicable, le temps de test et les conditions d'applicabilité selon le formalisme suivant :

```
<nom de la tâche> $ [[<proba>,<temps de test>], conditions]
```

Les conditions et les affectations étant exprimées à l'aide des prédicats `et`, `etl` et `ou` qui sont présentés en II.E.2.

```
/* Decomposition */

faire_plat_de_spaghetti(Q) <- cuisiner_les_spaghetti([Q,Q]) //
      cuisiner_sauce([Q,'pates']) // prechauffer_assiettes('pates').
cuisiner_les_spaghetti([P,S]) <- cuisiner_spaghetti_achetes(P).
cuisiner_les_spaghetti([P,S]) <- cuisiner_spaghetti_maison(S).
cuisiner_spaghetti_achetes(Q) <- cuire_les_pates(Q,'achetes').
cuisiner_spaghetti_maison(Q) <- faire_la_pate(Q) < former_les_spaghetti(Q)
      < cuire_les_pates(Q,'maison').
cuisiner_carbonara(Q) <- ingredients(Q) < melanger(Q).
cuisiner_bolognaise(Q) <- viande_hachee(V) //
      sauce(S) § etl([aff(V,Q/2),aff(S,Q/2)]).
faire_viande(Q) <- cuisiner_viande(Q) // cuisiner_sauce([Q,'viande']) //
      prechauffer_assiettes('viande').

/* Spécialisation (les probabilités sont indiquées en dixièmes) */

/* Q nb de personnes ; R quantite en g */
cuisiner_sauce([Q,T]) >> cuisiner_sauce_pour_pates(R) § aff(R,Q*100)
      $ [[2,10],et(eq(T,'pates'), supe(Q,3))] .
cuisiner_sauce([Q,T]) >> cuisiner_sauce_pour_viandes(R) § aff(R,Q*20)
      $ [[3,10],eq(T,'viande')].
cuisiner_sauce_pour_pates(Q) >> cuisiner_carbonara(Q)
      $ [[8,10],inf(Q,500)].
cuisiner_sauce_pour_pates(Q) >> cuisiner_bolognaise(Q)
      $ [[5,10],supe(Q,500)].
cuisiner_sauce_pour_viandes(Q) >> cuisiner_au_poivre(Q)
      $ [[2,10],infe(Q,100)].
cuisiner_sauce_pour_viandes(Q) >> cuisiner_au_vin(Q) $ [[5,10],sup(Q,100)].
cuire_les_pates(Q,T) >> cuire_les_pates_ach(Q) $ [[5,10],eq(T,'achetes')].
cuire_les_pates(Q,T) >> cuire_les_pates_mais(Q) $ [[5,10],eq(T,'maison')].
```

```
/* Methodes
/* tr_plat : indique un plat préparé par un traiteur */

faire_plat_de_spaghetti(Q) <- tr_f_plat_spagh(Q)#150.
cuisiner_les_spaghetti(Q) <- tr_cui_spagh(Q)#145.
cuisiner_spaghetti_achetes(Q) <- tr_cuis_spa_ach(Q)#140.
cuisiner_spaghetti_maison(Q) <- tr_cuis_spa_mais(Q)#130.
faire_viande(Q) <- tr_fa_viande(Q)#150.
cuisiner_sauce(Q) <- tr_cui_sauce(Q)#100.
cuisiner_sauce_pour_pates(Q) <- tr_cui_sau_pates(Q)#35.
cuisiner_sauce_pour_viandes(Q) <- tr_cui_sau_viande(Q)#40.
cuire_les_pates(Q,T) <- tr_cuire_pates(Q)#80.
cuire_les_pates_ach(Q) <- cuire(R,15)#15 § aff(R,100*Q).
cuire_les_pates_mais(Q) <- cuire(R,5)#5 § aff(R,100*Q).
faire_la_pate(Q) <- pate(Q)#15.
former_les_spaghetti(Q) <- former(Q)#10.
cuisiner_carbonara(Q) <- carbonara(Q)#15.
cuisiner_bolognaise(Q) <- bolognaise(Q)#10.
cuisiner_au_poivre(Q) <- poivre(Q)#10.
cuisiner_au_vin(Q) <- vin(Q)#15.
prechauffer_assiettes('pates') <- chauffer(5)#5.
prechauffer_assiettes('viande') <- chauffer(10)#10.
ingredients(Q) <- ingr(Q)#5.
melanger(Q) <- melang(Q)#10.
viande_hachee(Q) <- viande_hac(Q)#10.
sauce(Q) <- sauc(Q)#10.
cuisiner_viande(Q) <- viande(Q)#15.
```

Un exemple complet

Afin de suivre la résolution de la tâche, nous proposons les traces suivantes de l'exécution du programme :

— les moyens de résolution de la tâche courante (`S` est la liste contenant les tâches qui sont sures, c'est-à-dire toujours applicables, et `H` est la liste des tâches qui, au contraire, sont conditionnelles) ,

— la tâche rationnelle parmi cet ensemble,

— éventuellement les sous-tâches successives,

— la méthode directe choisie.

— lors d'un échec, (c'est-à-dire d'une décision rationnelle qui finalement ne permet pas la résolution de la tâche courante), on affiche 'ECHEC'.

L'appel au système se fait sous la forme :

`solve_rat(tâche(paramètres), P,C)` (`_rat` pour rationnel) où `P` représente le plan final qui est proposé par le système et `C` son coût d'exécution.

```
solve_rat(faire_plat_de_spaghetti(6),P,C).
```

Les moyens de la résoudre sont :

```
S=
[cuisiner_les_spaghetti([6,6])//cuisiner_sauce([6,pates])//prechauffer_assi
ettes(pates),tr_f_plat_spagh(6)#150]
H=  []
```

La tâche rationnelle à résoudre est :

```
cuisiner_les_spaghetti([6,6])//cuisiner_sauce([6,pates])//prechauffer_assie
ttes(pates)
```

La sous-tâche à résoudre est :

```
cuisiner_les_spaghetti([6,6])
```

Les moyens de la résoudre sont :

```
S=
[cuisiner_spaghetti_achetes(6),cuisiner_spaghetti_maison(6),tr_cui_spagh([6
,6])#145]
H=  []
```

La tâche rationnelle à résoudre est :

```
cuisiner_spaghetti_maison(6)
```

La sous-tâche à résoudre est :

```
cuisiner_spaghetti_maison(6)
```

Les moyens de la résoudre sont :

```
S=
[faire_la_pate(6)<former_les_spaghetti(6)<cuire_les_pates(6,maison),tr_cuis
_spa_mais(6)#130]
H=  []
```

La tâche rationnelle à résoudre est :

```
faire_la_pate(6)<former_les_spaghetti(6)<cuire_les_pates(6,maison)
```

La sous-tâche à résoudre est :
```
faire_la_pate(6)
```
Les moyens de la résoudre sont :
```
S=  [pate(6)#15]
H=  []
```

La tâche rationnelle à résoudre est :
```
pate(6)#15
```
Résolution par application de la méthode :
```
pate(6)
```

La sous-tâche à résoudre est :
```
former_les_spaghetti(6)
```
Les moyens de la résoudre sont :
```
S=  [former(6)#10]
H=  []
```

La tâche rationnelle à résoudre est :
```
former(6)#10
```
Résolution par application de la méthode :
```
former(6)
```

La sous-tâche à résoudre est :
```
cuire_les_pates(6,maison)
```
Les moyens de la résoudre sont :
```
S=  [tr_cuire_pates(6)#80]
H=
[cuire_les_pates_ach(6)$[[5,10],eq(maison,achetes)],cuire_les_pates_mais(6)
$[[5,10],eq(maison,maison)]]
```

La tâche rationnelle à résoudre est :
```
cuire_les_pates_mais(6)$[[5,10],eq(maison,maison)]
```
Les moyens de la résoudre sont :
```
S=  [cuire(_10085,5)#5§aff(_10085,100*6)]
H=  []
```

La tâche rationnelle à résoudre est :
```
cuire(_10085,5)#5§aff(_10085,100*6)
```
Résolution par application de la méthode :
```
cuire(600,5)
```

La sous-tâche à résoudre est :
```
cuisiner_sauce([6,pates])
```
Les moyens de la résoudre sont :
```
S=  [tr_cui_sauce([6,pates])#100]
H=
[cuisiner_sauce_pour_pates(_10724)§aff(_10724,6*100)$[[2,10],et(eq(pates,pa
tes),supe(6,3))],cuisiner_sauce_pour_viandes(_10708)§aff(_10708,6*20)$[[3,1
0],eq(pates,viande)]]
```

La tâche rationnelle à résoudre est :

```
cuisiner_sauce_pour_viandes(_10708)§aff(_10708,6*20)$[[3,10],eq(pates,viand
e)]
......... ECHEC ......... on reprend la résolution de la tâche courante
```

Les moyens de la résoudre sont :

```
S=  [tr_cui_sauce([6,pates])#100]
H=
[cuisiner_sauce_pour_pates(_10724)§aff(_10724,6*100)$[[2,10],et(eq(pates,pa
tes),supe(6,3))]]
```

La tâche rationnelle à résoudre est :

```
cuisiner_sauce_pour_pates(_10724)§aff(_10724,6*100)$[[2,10],et(eq(pates,pat
es),supe(6,3))]
```

Les moyens de la résoudre sont :

```
S=  [tr_cui_sau_pates(600)#35]
H=
[cuisiner_carbonara(600)$[[8,10],inf(600,500)],cuisiner_bolognaise(6*100)$[
[5,10],supe(600,500)]]
```

La tâche rationnelle à résoudre est :

```
cuisiner_carbonara(600)$[[8,10],inf(600,500)]
......... ECHEC ......... on reprend la résolution de la tâche courante
```

Les moyens de la résoudre sont :

```
S=  [tr_cui_sau_pates(600)#35]
H=  [cuisiner_bolognaise(6*100)$[[5,10],supe(600,500)]]
```

La tâche rationnelle à résoudre est :

```
cuisiner_bolognaise(600)$[[5,10],supe(600,500)]
```

Les moyens de la résoudre sont :

```
S=
[viande_hachee(_13935)//sauce(_13936)§etl([aff(_13935,600/2),aff(_13936,600
/2)]),bolognaise(600)#10]
H=  []
```

La tâche rationnelle à résoudre est :

```
viande_hachee(_13935)//sauce(_13936)§etl([aff(_13935,300),aff(_13936,300)])
```

La sous-tâche à résoudre est :

```
viande_hachee(300)
```

Les moyens de la résoudre sont :

```
S=  [viande_hac(300)#10]
H=  []
```

La tâche rationnelle à résoudre est :

```
viande_hac(300)#10
```

Résolution par application de la méthode :

```
viande_hac(300)
```

La sous-tâche à résoudre est :

```
sauce(300)
```

Les moyens de la résoudre sont :

```
S=  [sauc(300)#10]
H=  []
```

La tâche rationnelle à résoudre est :
```
sauc(300)#10
```
Résolution par application de la méthode :
```
sauc(300)
```

La sous-tâche à résoudre est :
```
prechauffer_assiettes(pates)
```
Les moyens de la résoudre sont :
```
S=  [chauffer(5)#5]
H=  []
```

La tâche rationnelle à résoudre est :
```
chauffer(5)#5
```
Résolution par application de la méthode :
```
chauffer(5)
```

```
P =
parallel([seq([pate(6),former(6),cuire(600,5)]),viande_hac(300),sauc(300),c
hauffer(5)])
C = 30 ;

no
```

Table des matières

Sommaire **1**
Introduction **1**
I. Résolution de problèmes **2**
A. Historique 2
1 La notion de tâche 2
2 Les tâches et les ERP 3
3 La planification 4
a. Origine et définition 4
b. La recherche de solutions 5
c. La planification hiérarchique 6
d. D'autres points de vue 6
e. Le résultat fourni 6
4 Les taches exécutables 7
B. Cadre du projet 8
1 Les tâches et les méthodes 8
2 Les relations entre tâches 8
a. La décomposition 8
b. La spécialisation 9
3 Recherche rationnelle d'une solution 10
a. La planification hiérarchique 10
b. La solution la plus utile 11
c. Comparaison des deux stratégies 12
4 Implémentation des tâches 13
C. Objectif du projet et motivations 13
II. Sémantique opérationnelle d'un système de tâches **15**
A. Présentation générale du système 15
1 Son rôle 15
2 Les données manipulées 15
B. Le système simplifié 17
1 Les règles simples 17
2 Quelques remarques 17
3 Un exemple 18
C. Les flots de données 20
1 Les règles avec des flots de données 20

2 Un dernier point à préciser 21
3 Des illustrations 21
D. La vérification de l'applicabilité 23
1 La règle à modifier 23
2 Des illustrations 23
E. L'implémentation 24
1 Un méta-interprète Prolog 24
2 L'application au système de tâches 25
a. Les données 25
b. Le système de règles simples 27
c. Le système manipulant des variables 29
d. Le système et les affectations de variables 30
e. Le système autorisant des conditions 31
F. Terminaison et complexité 32
G. Esquisse de sémantique dénotationnelle 32
H. Conclusion 34
III. Tâche et rationalité 36
A. La rationalité 36
1 Qu'est-ce qu' «être rationnel» ? 36
2 Pourquoi être rationnel ? 37
3 Comment être rationnel ? 38
4 La fonction d'utilité 41
B. L'utilité et les tâches 42
1 Rationalité et tâches spécialisables 42
2 L'utilité et les ressources 43
3 La fonction d'utilité 44
4 Le calcul de la valeur nette de gain d'utilité 45
a. Les probabilités 45
b. Les ressources consommées 46
c. La valeur des ressources intrinsèques 47
C. Utilisation statique de la rationalité 47
1 Introduction 47
2 Idée générale 48
3 L'algorithme 48
a. Schéma général 49
b..... Calcul simple pour une tâche n'admettant qu'une seule sous-tâche 49
c. Calcul de la valeur nette des sous-tâches 49

d. Calcul du temps moyen de résolution de T 51
4 La résolution d'une tâche 51
5 La sémantique du système de tâches rationnel 52
6 L'implémentation 53
7 D'autres critères..... 54
8 Evaluation et critique de cette méthode..... 54
D. Utilisation dynamique de la rationalité..... 55
1 Sélection dynamique de sous-tâches 55
a. Sélection de la meilleure stratégie sûre 56
b. Ordre de tests des spécialisations 56
c. Cas particulier du temps..... 57
2 La sémantique du système de tâche rationnel (bis) 58
3 L'implémentation 59
a. Modification des données..... 59
b. Stratégie générale 59
c. Recherche du moyen rationnel 60
d. Résolution du moyen rationnel..... 61
e. Un exemple..... 62
4 Evaluation d'une telle stratégie..... 63
E. Conclusion..... 64
1 Comparaison des méthodes statique et dynamique 64
2 La limitation des ressources 64
3 Et si l'on considérait plusieurs ressources 65
Conclusion et perspectives 67
Références 69
Annexes 73
Le programme..... 74
La base de faits..... 79
Un exemple complet..... 81
Table des matières.....85