Université de Nancy 1
U.E.R. Sciences mathématiques

Centre de Recherche en
Informatique de Nancy

Résolution de problèmes dans l'espace d'états avec abstraction de concepts — le système expert OASIS

THÈSE

présentée et soutenue publiquement le **29 avril 1989**

pour l'obtention du

Doctorat de l'Université de Nancy 1

(Spécialité Informatique)

par

Yudong SUN
（孫 宇東）

Composition du jury :

Président : Jean-Paul HATON

Rapporteurs : Pierre MARCHAND
Jean SALLANTIN

Examinateurs : Paul CAUBÈRE
Jean-Claude CHARPENTIER
Marie-Christine HATON

Université de Nancy 1
U.E.R. Sciences mathématiques

Centre de Recherche en
Informatique de Nancy

Résolution de problèmes dans l'espace d'états avec abstraction de concepts — le système expert OASIS

THÈSE

présentée et soutenue publiquement le **29 avril 1989**

pour l'obtention du

Doctorat de l'Université de Nancy 1

(Spécialité Informatique)

par

Yudong SUN
(孫 宇東)

Composition du jury :

Président :	Jean-Paul HATON
Rapporteurs :	Pierre MARCHAND
	Jean SALLANTIN
Examinateurs :	Paul CAUBÈRE
	Jean-Claude CHARPENTIER
	Marie-Christine HATON

Je tiens à remercier :

Monsieur Jean-Paul Haton, Professeur à l'Université de Nancy 1 et Directeur de Recherche INRIA au CRIN, qui m'a fait confiance et a bien voulu m'accueuillir dans son laboratoire. C'est grâce à lui que j'ai pu me former en informatique. Je le remercie également pour le grand intérêt qu'il a toujours prêté à mes travaux et l'honneur qu'il me fait en présidant ce jury.

Madame Marie-Christine Haton, Maître de Conférence à l'Université de Nancy 1, qui a animé ce projet et m'a orienté tout au long de ce travail. Je tiens à lui exprimer mon extrême gratitude pour ses conseils, ses inspirations, ses encouragements et ses nombreux moments consacrés à la lecture de ce mémoire.

Mon compatriote Yifan Gong qui m'a fait confiance en me proposant auprès de Monsieur Jean-Paul Haton. Je lui suis d'autant plus reconnaissant pour l'aide généreuse qu'il m'a apportée dans les nombreux jours qui ont précédé et suivi mon arrivée au CRIN.

Monsieur Pierre Marchand, Professeur à l'Université de Nancy 1, qui a accepté d'être rapporteur de ce mémoire et de participer à ce jury. Je lui dois encore mes remerciements pour la clarté de son cours qui m'a beaucoup inspiré.

Monsieur Jean Sallantin, Chargé de Recherche CNRS au Centre de Recherche en Informatique de Montpelher, qui me fait l'honneur d'avoir accepté d'être rapporteur de ce mémoire et de siéger à ce jury. Qu'il trouve ici l'expression de ma profonde gratitude pour l'intérêt dont il fait preuve à l'égard de ce travail.

Monsieur Paul Caubère, Professeur à l'Université de Nancy 1, qui a été notre Expert dans le projet OASIS *et a été disponible dans de nombreuses occasions pour nous éclairer divers points concernant le problème de la synthèse chimique. Je le remercie également de bien vouloir siéger à ce jury.*

Monsieur Jean-Claude Charpentier, Directeur de Recherche au CNRS, qui me fait l'honneur de participer à ce jury. Qu'il trouve ici l'expression de ma profonde reconnaissance pour l'intérêt qu'il prête à ce travail.

Merci également à mes étroits collaborateurs Brigitte Devin et Laurent Pierron ainsi qu'à tous mes collègues du CRIN pour l'atmosphère de travail sympathique dont j'ai bénéficié.

Table des matières

1
Introduction

1.1 Contenu du présent travail

Le présent travail est motivé par la réalisation du projet OASIS – un système expert d'élaboration des chemins de synthèse en chimie organique – qui est un projet de collaboration entre le CRIN/INRIA et le Laboratoire de Chimie Organique I de l'Université de Nancy I (le Professeur P. Caubère).

Nous avons d'abord développé la notion d'*espace d'états avec abstraction de concepts*. Il s'agit d'un modèle général de représentation de problèmes qui décrit le domaine d'un problème donné en trois parties :

1. la représentation des états en terme de *concepts de base* ;
2. la définition, au moyen des clauses de Horn, des *concepts d'abstraction* à partir des concepts de base ;
3. la description des opérateurs de transition en terme de concepts de base et de concepts d'abstration.

Le mécanisme d'inférences correspondant comprend trois éléments :

1. un démonstrateur de théorèmes effectuant la tâche suivante :

   (un état) $\cup$ (clauses de Horn) $\stackrel{démonstration}{\Longrightarrow}$ (précondition d'un opérateur) ;

2. un évaluateur partiel destiné à spécialiser les clauses de Horn par rapport à un état donné en vue d'améliorer l'efficacité du démonstrateur de théorèmes ;
3. un interprète d'opérateurs permettant de passer d'un état à l'autre.

Le problème de la synthèse chimique se trouve aisément formulé dans le modèle de l'espace d'états avec abstraction de concepts :

1. Les molécules constituent les états, les concepts de base étant les concepts élémentaires en chimie juste suffisants pour spécifier la structure d'un graphe-molécule.

2. Les concepts d'abstraction correspondent surtout aux groupements fonctionnels (CN, $COOH$, etc.) et aux termes génériques (*libérable, électro-attracteur, polarisé*, etc.).

3. Les opérateurs correspondent aux réactions chimiques. Leur description s'appuient sur les deux types de concepts.

Reposant sur cette vision générale, nous avons développé le formalisme détaillé de représentation de connaissances et implanté les mécanismes d'inférences de base. Nous avons aussi étudié succinctement le problème du contrôle et dégagé les grandes lignes régissant l'implantation ultérieure du contrôleur.

1.2 Présentation de ce mémoire

Après ce premier chapitre d'introduction, le chapitre 2 est consacré à une réflexion générale sur l'intelligence artifielle qui tente de souligner quelques problèmes de conséquence.

Le chapitre 3 expose la nature du problème de la synthèse chimique et passe en revue un certain nombre de systèmes experts traitant ce sujet.

Le chapitre 4 a pour l'objet principal le développement de la notion d'espace d'états avec abstraction de concepts. Reposant sur ce modèle, nous décrivons l'approche générale adoptée dans le système OASIS.

Le chapitre 5 discute en détail de la représentation des connaissances : les molécules ou les états, les clauses de Horn, les réactions ou les opérateurs et les méta-règles.

Le chapitre 6 décrit la mise en œuvre des mécanismes d'inférences de base : le démonstrateur de théorèmes, l'évaluateur partiel et l'interprète d'opérateurs.

Le chapitre 7 expose l'idée générale qui présidera à l'implantation du contrôleur.

Le chapitre 8 discute du rôle de l'expert dans ce travail et aussi présente succinctement deux outils : un programme d'affichage des molécules et un programme de saisie graphique.

Le chapitre 9 conclut ce mémoire.

2
Intelligence artificielle et systèmes experts

2.1 Intelligence artificielle

2.1.1 Intelligence artificielle et logique formelle

L'intelligence artificielle a pour but de permettre à l'ordinateur d'exécuter les fonctions de l'intelligence humaine. Cette technique a connu un développement spectaculaire depuis les années soixante, bien que son objectif initial se révèle aujourd'hui plus difficile à atteindre qu'on l'avait prévu. On a abouti à des résultats concrets dans les diverses branches de cette nouvelle discipline : méthodes de description de problèmes et de représentation de la connaissance, raisonnement automatique et démonstration automatique de théorèmes, méthodes de recherche pour la résolution de problèmes, techniques d'interprétation des formes.

A part les techniques dans la reconnaissance des formes de bas niveau, le développement s'est centré sur la formulation de problèmes et le raisonnement automatique, ceux-ci étant au cœur de la faculté de la machine capable d'exécuter des tâches de l'intelligence humaine. Dans ce sens, les progrès dans le domaine de la logique et de la démonstration automatique de théorèmes [Loveland 84] doivent être les plus marquants.

L'approche logique recouvre la plupart des problèmes susceptibles de mener à une solution rigoureuse et tente de les résoudre de façon universelle. Ainsi, dans la perspective de cette approche, pour résoudre un problème, il suffira de le définir, c'est-à-dire, de le spécifier en termes logiques, la résolution étant assurée par un processus universel de démonstration de théorèmes. De ce fait, il est tentant de croire que l'approche logique s'imposerait comme le noyau de ce qu'on appelle aujourd'hui l'approche symbolique.

Cette conviction s'est vue renforcée surtout par l'avènement du langage PROLOG – un langage essentiellement basé sur le calcul des prédicats restreint aux clauses de Horn. L'origine de PROLOG date des travaux de [Colmerauer 72], avant de retrouver sa séman-

tique logique dans [Kowalski 74]. L'importance de l'approche logique a, par ailleurs, été reconnue par le projet japonais de cinquième génération d'ordinateurs [Motooka 85].

Un problème majeur à résoudre avec l'approche logique est celui de l'efficacité. Si le principe de résolution [Robinson 65] et ses raffinements ultérieurs ont rendu possible l'utilisation pratique de la démonstration de théorèmes pour la résolution de problèmes, la question de l'efficacité est loin d'être résolue. Celle-ci est liée à la stratégie selon laquelle le principe de résolution est appliqué. Actuellement, des efforts se voient consacrés à la recherche de meilleures stratégies de résolution. Citons, entre autres, les travaux de [Matwin 87,Kowalski 76] ; la solution finale impliquerait une phase de prétraitement pour structurer les clauses dans un graphe afin de dégager des plans de résolution.

Par ailleurs, dans cette perspective, le manque d'organisation dans les connaissances sous forme de clauses logiques ne serait plus un problème.

2.1.2 Heuristiques et systèmes experts

Parallèlement au développement dans le domaine de la logique formelle et de la démonstration automatique de théorèmes, diverses méthodes de résolution de problèmes ont été développées. Elles sont caractérisées par l'utilisation d'heuristiques. Elles sont capables de faire face à des problèmes auxquels l'approche purement logique ne peut apporter de réponse ; ou bien elles mènent à une solution de manière plus efficace.

Premièrement, le raisonnement humain n'est pas toujours rigoureux mais souvent heuristique. En effet, l'intelligence de l'homme réside surtout dans sa faculté de pouvoir, sous les contraintes d'un délai de temps raisonnable et des ressources de mémoire limitées, trouver une solution – pas forcément exacte – au problème auquel il doit faire face, d'une manière "pas très logique". C'est ainsi que l'homme joue aux échecs, alors que l'approche combinatoire se fondant uniquement sur une spécification logique du problème échoue.

D'autre part, dans des cas moins extrêmes, on peut parvenir plus vite à une solution si, au cours de la recherche, on a un ordre d'importance heuristique pour les branches à explorer.

En bref, les heuristiques permettent de rapprocher l'intelligence de la machine de l'intelligence humaine.

D'une manière générale, un résolveur de problème heuristique peut être vu comme comportant deux aspects :

- un aspect logique et
- un aspect heuristique de contrôle.

Les efforts dans cette voie ont conduit, entre autres, à GPS (pour *general problem solver*) [Ernst 69], pour résoudre le type de problèmes qui consistent à atteindre, en partant d'un état initial, un état cible en utilisant des opérateurs de transition d'état. C'est une approche caractérisée par une stratégie de sélection d'opérateurs qui, reposant sur l'*analyse des fins et des moyens*, vise à réduire en priorité la différence jugée la plus importante entre l'état courant du problème et l'état cible.

Si GPS était une approche pour résoudre des problèmes d'ordre général tels que la manipulation de blocs par un robot,"le singe et les bananes", d'autres travaux ont été menés pour résoudre des problèmes venant de domaines spécialisés et dont la résolution nécessitait une large quantité de connaissances spécialisées et relevait de la compétence de certains spécialistes ou experts. Les premiers systèmes experts ont ainsi vu le jour, comme DENDRAL [Feigenbaum 71] et MYCIN [Shortliffe 76].

En tant que résolveurs de problèmes dans des domaines spécialisés, les systèmes experts présentent un intérêt industriel immédiat, s'imposant comme une importante filière de l'I.A. pouvant déboucher sur des applications concrètes. Cette nouvelle technologie permet désormais, dans une certaine mesure, de commercialiser la connaissance humaine sous forme de produit et d'en faire des copies ; un nouveau mode de transmission de connaissance est ainsi né.

D'une manière générale, un système expert est toujours un résolveur de problème d'I.A. comportant un aspect logique et un aspect heuristique guidant la résolution de problème. Cependant, la complexité des connaissances et des méthodes heuristiques de résolution provenant de différents domaines spécialisés pose de nombreux problèmes, notamment en ce qui concerne les modes d'expression des connaissances, l'intégration de techniques de raisonnement de différentes natures, la coopération de multi-sources de connaissances, l'acquisition de l'expertise, etc., ce qui fait que les systèmes experts sont devenus une branche de recherche en soi, pour répondre au besoin d'application.

Pour terminer, traçons le parallèle entre l'approche heuristique de résolution de problèmes d'une part et l'approche logique – notamment le calcul des prédicats – d'autre part.

Alors que l'approche heuristique comporte deux aspects : un aspect logique traduisant des raisonnements de base, et un aspect heuristique de contrôle, l'approche logique comporte exclusivement un aspect logique, le contrôle y étant un processus universel et ne tenant pas compte des heuristiques d'un problème spécifique. Ceci est destiné à souligner le fait que l'approche heuristique a un aspect heuristique de contrôle de plus par rapport à l'approche logique, pour faire face au problème lié aux ressources matérielles et au temps de calcul, ou tout simplement pour se rapprocher de l'intelligence humaine.

Néanmoins, l'approche logique est toujours d'une importance inestimable. D'une part, elle offre un moyen efficace et universel pour résoudre les problèmes susceptibles de mener à une solution rigoureuse, et d'autre part et de ce fait, elle est même à la base de l'approche heuristique en accomplissant les raisonnements de base.

2.1.3 Approche connexionniste et heuristique

Ces dernières années ont vu un nouvel engouement des chercheurs en faveur de l'approche connexionniste ou l'approche du réseau neuronal [Kosko 87,Jones 87,Josin 87]. Contrairement à l'approche symbolique où l'information prend une forme alphanumérique et est codée et stockée dans un endroit précis de la mémoire, dans l'approche connexionniste, l'information est stockée de manière distribuée sur l'ensemble du réseau, ceci grâce à l'aptitude d'apprentissage et d'auto-adaptation qui est à la base de cette approche.

L'approche connexionniste présente le caractère heuristique propre à l'homme : apprentissage à partir des expériences, tolérer les défauts.

Cependant, on n'en est qu'à ses débuts : actuellement l'approche connexionniste ne remplit pratiquement que de simples fonctions de mémoire et il est encore peu concevable qu'elle puisse faire des raisonnements sophistiqués. Dans l'état actuel des choses, sa manière d'apprendre et de servir de mémoire se prête surtout bien à la reconnaissance des formes de bas niveau [Bourlard 89,De Saint Pierre 87] – à la reconnaissance des caractères par exemple – qui, avec l'approche symbolique (méthodes syntaxiques), exige beaucoup de raisonnements.

Une recherche approfondie dans cette voie impliquerait deux niveaux de connaissance du réseau neuronal de l'homme : le mécanisme d'apprentissage de l'homme dans la vie et le mécanisme qui est à la base de l'évolution humaine.

2.1.4 Perspectives

Alors que le raisonnement logique constitue le noyau de l'approche symbolique, la plupart des résolveurs de problèmes d'I.A. comportent aussi un aspect heuristique de contrôle, pour se rapprocher de l'intelligence humaine.

D'autre part, l'approche connexionniste est plus proche du cerveau humain. Elle est auto-adaptative, présentant un caractère autonome et évolutif.

Dans la perspective, on pourrait espérer allier les deux approches dans la construction des résolveurs de problèmes. Le raisonnement de base serait fondé sur un mécanisme général purement logique, tandis que les heuristiques seraient implantées de manière connexionniste. Dans une telle perspective, l'outil de raisonnement logique se trouverait à la disposition d'un réseau neuronal presqu'au même titre qu'un programme d'inversion de matrice à la disposition de l'homme aujourd'hui.

Les grandes voies de recherche pour l'avenir seraient :

- Le perfectionnement de l'approche logique. Les problèmes, censés avoir une solution rigoureuse et un espace de recherche contrôlable, devraient pouvoir être résolus de manière optimale, du point de vue de la quantité de recherche effectuée.
- L'approfondissement du modèle connexionniste, pour pouvoir rendre compte de la manière heuristique de l'homme dans la résolution de problèmes.

Les deux approches sont complémentaires.

Un autre problème intéressant, qui n'est sans doute pas isolé des deux points précédents, consiste à savoir :

- comment dégager des algorithmes heuristiques pour résoudre un problème à partir de sa spécification logique. Prenons le cas du jeu d'échecs. L'homme apprend à jouer en se fondant uniquement sur les règles du jeu, la spécification de celles-ci constituant la définition complète du jeu. Il est donc théoriquement possible de mécaniser le processus de recherche d'algorithmes heuristiques étant donné la spécification logique d'un problème.

Enfin, il ne faut pas oublier la poursuite des activités dans :

- la sophistication de la méthodologie classique de l'approche symbolique, notamment dans les systèmes experts.

Un exemple en est la résolution distribuée d'un problème [Haton 89]. L'idée est de bien faire coopérer plusieurs sources de connaissaces en vue d'une résolution opportuniste et incrémentale du problème.

2.2 Systèmes experts

Revenons maintenant sur le développement de systèmes experts. Comme on l'a vu précédemment, d'un point de vue structurel, un système expert est un résolveur de problèmes dans un domaine particulier, intégrant un aspect de raisonnement logique et un aspect de contrôle heuristique pour la résolution de problèmes. Toutefois, la nécessité de résoudre divers problèmes compliqués provenant de différents domaines spécialisés a porté la méthodologie des systèmes experts à un degré de sophistication impressionnant. Cette

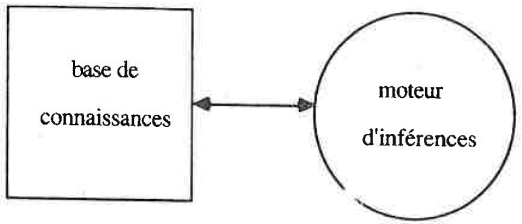


Figure 2.1. Structure générale d'un système expert

sophistication exige une conception performante de l'architecture logicielle des systèmes experts.

Dans la suite, nous discuterons les caractéristiques générales des systèmes experts d'un point de vue structurel, bien que la méthodologie des systèmes experts comporte encore d'autres aspects, comme l'acquisition de connaissances, etc.

2.2.1 Structure générale

En tant que resolveurs de problèmes dans des domaines spécialisés, les systèmes experts renferment généralement une quantité assez importante de connaissances du domaine. Leur fonctionnement repose sur un mécanisme d'exploitation de ces connaissances – appelé le moteur d'inférences – qui, suivant le problème à résoudre, entreprend les raisonnements nécessaires pour aboutir à une solution.

Ce schéma quelque peu simpliste mais commun à tout système expert est illustré sur la figure 2.1. Celle-ci met en relief les deux composantes essentielles d'un système expert : une base de connaissances et un moteur d'inférences. La séparation organique entre les connaissances et leur mécanisme d'exploitation constitue une caractéristique des systèmes experts. Cette séparation permet aux systèmes experts d'évoluer facilement, car on peut ajouter et/ou modifier les connaissances sans pour autant toucher au programme.

Base de connaissances

La base de connaissances renferme l'expertise du domaine et constitue le facteur intrinsèque conditionnant la puissance du système. Elle est généralement exprimée sous forme déclarative, ceci sans préjuger de la manière dont elle sera exploitée par la suite par le mécanisme d'inférence.

La représentation des connaissances est un sujet important dans la méthodologie des systèmes experts. On a plusieurs formalismes possibles : les règles de production, le calcul des prédicats, les objets structurés ou *frames*, les réseaux sémantiques, ainsi que des stratégies mixtes, dont le choix est fait suivant la nature de l'expertise à exprimer.

Le système MYCIN [Shortliffe 76] qui vise le problème du diagnostic médical dans le cas des maladies du sang et DENDRAL [Feigenbaum 71] qui est un système d'élucidation de structures moléculaires, s'appuient sur le formalisme des règles de production, la structure `SI-ALORS` de celui-ci étant bien adaptée à l'expertise à exprimer.

Modes de raisonnement

En ce qui concerne le mécanisme d'exploitation des connaissances, on peut généralement distinguer deux modes de raisonnement pour les moteurs d'inférences se fondant sur les règles de production : chaînage en avant et chaînage en arrière. Dans le mode de chaînage-avant, le moteur démarre avec la base de faits du problème initial et l'enrichit progressivement d'autres faits qu'il déduit en utilisant les règles de production. Dans le mode de chaînage-arrière, le moteur débute en cherchant à établir une hypothèse. Pour ce faire, il se sert des règles du système pour décomposer l'hypothèse initiale en sous-hypothèses, et puis en prenant ces sous-hypothèses comme hypothèse initiales, il cherche à les établir à leur tour. Il s'arrête avec succès sur une hypothèse si celle-ci figure dans la base de faits du système. Si une hypothèse ne peut être décomposée (pas de règle applicable) et ne figure pas non plus dans la base de faits, elle est rejetée.

Le moteur d'inférences de MYCIN [Shortliffe 76] fonctionne en chaînage-arrière ; celui de DENDRAL [Feigenbaum 71] repose essentiellement sur le chaînage-avant pour accomplir la phase *génération* dans le cycle *génération-test*.

2.2.2 Structures avancées

Le développement de la méthodologie des systèmes experts s'oriente vers une sophistication accrue, afin d'étendre son champ d'application à des problèmes plus complexes tant sur le plan de l'expertise impliquée que sur celui du raisonnement. Cela exige une représentation flexible des connaissances et un mécanisme de raisonnement efficace.

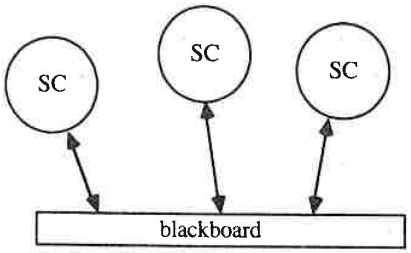


Figure 2.2. Le modèle du blackboard

En ce qui concerne les connaissances à utiliser, il faut souvent prendre en compte à la fois soit les divers aspects d'une expertise, soit plusieurs expertises. Dans les systèmes de compréhension de la parole continue [Haton 85] par exemple, il s'agit de faire coopérer au moins trois niveaux de connaissances : phonétique, lexicale et syntaxique, qui sont tous indispensables pour interpréter une phrase. Ce genre de systèmes est souvent appelé *système multi-experts*.

Modèle du blackboard

Initialement proposé par [Lesser 75] dans le système de compréhension de la parole HEARSAY-II, le modèle du blackboard a été par la suite largement repris et enrichi pour le développement d'autres systèmes multi-experts. Ce modèle propose la résolution distribuée et opportuniste d'un problème. Il est illustré sur la figure 2.2. On voit les deux composantes principales du modèle du blackboard :

- les sources de connaissances,
- le blackboard : une structure de données globale.

Au départ le blackboard contient les données du problème initial et s'enrichira par la suite d'hypothèses au cours de la résolution d'un problème. Les sources de connaissances apportent des modifications au blackboard en y inscrivant, supprimant ou modifiant des

hypothèses, lorsque leur condition de déclenchement est rempli par l'état courant du blackboard. Le processus de la résolution d'un problème se déroule ainsi de façon incrémentale jusqu'à ce qu'une solution soit obtenue.

A part les sources de connaissances et le blackboard, le modèle du blackboard contient souvent aussi une autre composante : le contrôleur, qui est destiné à activer les sources de connaissances selon certaines stratégies.

D'autres modèles

Le modèle du blackboard offre un bon schéma pour résoudre des problèmes impliquant plusieurs sources de connaissances ayant un degré d'indépendance au moins relative. Mais il n'est bien évidemment pas universel. Par exemple, il existe des problèmes qui mettent en jeu plusieurs niveaux de connaissances assez imbriqués entre eux, et pour les résoudre il faut établir des modèles appropriés.

Dans le système OASIS pour la synthèse chimique, qui fait l'objet de cette thèse, il s'agit d'un tel problème. Nous avons développé le modèle de *l'espace d'états avec abstraction de concepts*, qui est un développement de la notion de *l'espace d'états* que l'on trouve, par exemple, dans le problème du robot dans le monde des blocs [Winston 84], en y introduisant un niveau de *concepts d'abstraction* de plus, par opposition au niveau des *concepts de base*.

Le domaine d'un problème selon ce modèle comporte trois composantes :

1. la représentation d'*états* en terme de *concepts de base*,
2. la définition des *concepts d'abstraction* en clauses de Horn,
3. la représentation d'opérateurs en terme de *concepts de base* et/ou de *concepts d'abstraction*.

Les mécanismes d'inférence de base s'appuient sur trois composantes principales (cf. figure 4.3):

1. un interprète d'opérateurs,
2. un démonstrateur de théorèmes,
3. un évaluateur partiel,

pour avoir une interprétation souple et efficace des connaissances.

3
Systèmes experts en synthèse chimique

3.1 Synthèse chimique

La synthèse chimique est de première importance dans la fabrication de nouveaux produits chimiques tels que des médicaments, des combustibles, des matières plastiques, etc. Or, la conception d'une synthèse chimique nécessite une énorme quantité de connaissances chimiques. En 1978, on mentionnait [Wipke 78] déjà plus de 4.000.000 composés chimiques et probablement 20.000 réactions chimiques. On a donc pensé à recourir aux moyens informatiques [Corey 69,Choplin 85].

Le problème de la synthèse chimique consiste fréquemment à construire une structure moléculaire complexe ou une *cible* – qui pourrait être un médicament par exemple – à partir de molécules simples qui existent directement ou indirectement dans la nature, en utilisant des réactions chimiques ou des *opérateurs*. Les molécules de départ contiennent la plupart du temps quatre ou moins d'atomes de carbone tandis que la molécule cible peut en contenir jusqu'à trente et plus. La réalisation d'une synthèse comporte deux phases : la phase théorique où l'on conçoit un *plan* c'est-à-dire des suites d'opérateurs (un arbre plus précisément) reliant les molécules de départ à la molécule cible, et la phase expérimentale qui est l'exécution du plan au laboratoire. L'aide informatique se trouve dans la première phase. Celle-ci est d'une importance extrême car, une fois le choix fait, l'exécution du plan au laboratoire demande de nombreuses personnes-années de travail. Un bon plan permet donc des économies.

Dans un cadre plus général, on peut classer les problèmes en synthèse chimique en trois catégories :

1. $? \xrightarrow{?} B$. C'est le cas que l'on vient de voir ci-dessus et qui s'appelle *rétrosynthèse*. Il est le plus fréquent et a donc la plus grande importance dans la synthèse chimique. C'est ce type de problèmes qui fait l'objet de la plupart des systèmes experts existants.

Il consiste à trouver une arborescence d'opérateurs chimiques menant à la molécule cible B en partant de molécules simples quelconques.

2. $A \xrightarrow{?} B$. C'est le cas de *synthèse*, par opposition à celui de *rétrosynthèse*. Il est caractérisé par la spécification d'une ou plusieurs molécules de départ devant participer à la fabrication de la molécule cible B. Le problème consiste à trouver une suite d'opérateurs pour relier les molécules de départ à la molécule cible.

3. $A \xrightarrow{?} ?$. C'est également un cas de *synthèse*. Il s'agit d'une situation dans laquelle on dispose d'une molécule dont on ignore l'utilité et l'on veut savoir en quoi d'utile elle peut être transformée.

Les connaissances impliquées dans la conception d'un plan de synthèse relèvent de quatre catégories :

1. Les principes structurels. Ce sont les connaissances nécessaires pour spécifier la structure d'une molécule : les atomes et leur ionicité, les liaisons.

2. Les propriétés moléculaires. Certaines sous-structures dans une molécule peuvent présenter une certaine propriété. Il s'agit des informations de haut niveau concernant la structure d'une molécule.

3. Les réactions chimiques. Ces opérateurs chimiques constituent l'étape de base dans la conception d'une synthèse, ils permettent de passer d'une molécule à une autre.

4. Stratégies de synthèse. Ce sont des méta-connaissances concernant les stratégies de recherche.

3.2 Quelques systèmes experts existants

Plusieurs systèmes experts d'aide à la synthèse chimique ont vu le jour depuis la fin des années soixante. On peut citer LHASA à l'Université de Harvard [Corey 69,Johnson 85], SECS à l'Université de Californie [Wipke 78], SYNCHEM II à l'Université de l'Etat de New York à Stony Brook [Gelernter 84], MARSEIL à Marseille [Barone 82].

Ces systèmes, tous sortis de laboratoires de chimie, ne présentent pas de différences de principe. Ils exploitent tous une importante base de réactions chimiques et profitent de stratégies heuristiques sophistiquées de planification venant de la part de leurs concepteurs. Ils traitent généralement la *rétrosynthèse*, la stratégie de base consistant à développer des chemins en arrière en partant de la molécule cible à synthétiser, en choisissant "bien" des opérateurs [Warren 78].

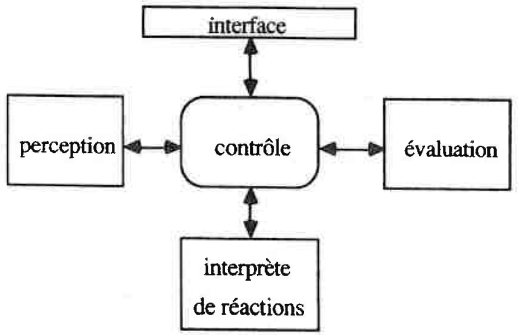


Figure 3.1. Architecture simplifiée de SECS

On peut envisager deux modes de fonctionnement : le mode interactif et le mode non-interactif. Dans le mode non-interactif, le système développe des chemins sans intervention de l'expert-utilisateur. Et pour ce faire, il nécessite de mettre en œuvre divers critères heuristiques afin d'assurer la pertinence des chemins en cours de développement. Le mode interactif, en revanche, ne se veut qu'une assistance au chimiste dans la synthèse. Le chimiste sert au système de fonction d'évaluation qui décide de continuer de développer un chemin sur un nœud ou de s'y arrêter.

La plupart des systèmes (et tous les systèmes ci-dessus) utilisent le mode interactif, et cela pour plusieurs raisons. Avant tout, le but est de trouver de "bons" chemins tels que le juge le chimiste utilisant le système. Les critères du chimiste à cet égard sont liés à de nombreux facteurs tels que le nombre d'étapes dans la synthèse, le rendement, la connaissance du chimiste des étapes impliquées, etc. Même ses expériences du passé peuvent jouer un rôle : par exemple, il peut préférer utiliser des réactions qu'il a expérimentées auparavant dans son laboratoire.

Le présent système OASIS adopte également le mode interactif.

Pour terminer cette analyse, examinons le système SECS [Wipke 78] dont l'architecture simplifiée est présentée en figure 3.1. Au commencement d'une session de travail, le chimiste dessine sur l'écran la molécule cible à synthétiser. L'interface la traduit en une *table de connectivités* qui est la représentation interne de la molécule. Ensuite, le module de perception recherche des informations de haut niveau : les propriétés de certaines sous-

structures, des groupements fonctionnels, etc. Le module de contrôle décide des types de réactions à appliquer. L'interprète de réactions effectue alors des transformations sur la molécule cible, pour obtenir des précurseurs c'est-à-dire de nouvelles cibles. Le module d'évaluation vérifie la légitimité chimique des nouvelles structures ; seules les légitimes seront prises en compte et enchaînées par la suite dans l'arbre de synthèse.

Le résultat final sera un arbre ET/OU ayant pour racine la molécule cible de départ. Un nœud ET résulte de la situation où une structure cible est scindée en deux (voire plusieurs), par exemple par une réaction de type :

$$A + B \longrightarrow C,$$

et un nœud OU provient du fait que différentes tranformations appliquées à une cible donnent lieu à des précurseurs différents.

3.3 Représentation des connaissances

En ce qui concerne la représentation des connaissances dans les systèmes experts en synthèse chimique, on doit considérer deux aspects : la représentation des structures moléculaires d'une part et l'organisation des connaissances d'autre part.

Les systèmes existants sont en général peu documentés en informatique. On y représente les structures moléculaires au moyen de tables de connectivités (cf. chapitre 5). A partir de là, les différents types de connaissances – molécules, réactions, propriétés moléculaires, etc. – sont exprimées pour être exprimées, sans que l'on donne une sémantique informatique claire à leurs liens (faute d'un modèle global). Le résultat n'est ainsi pas forcément efficace.

Dans le système OASIS, nous établissons d'abord un modèle de représentation du problème – modèle de *l'espace d'états avec abstraction de concepts* – avec les mécanismes d'inférences correspondants. Il s'agit d'ailleurs d'un modèle assez général pour représenter une catégorie de problèmes complexes. C'est en se fondant sur ce modèle que sont développés la représentation des connaissances et les mécanismes d'interprétation, afin d'assurer une meilleure efficacité.

4
Vue d'ensemble du système expert OASIS

4.1 Le projet OASIS

Le projet OASIS [Haton 87,Sun 88], démarré en 1985-86, est en cours de développement au Centre de Recherche en Informatique de Nancy (CRIN) en collaboration avec le Laboratoire de Chimie Organique (le Professeur P. Caubère), à l'Université de Nancy 1.

Dans son objectif final, OASIS vise les trois types de problèmes exposés au chapitre 3 :

1. $? \xrightarrow{?} B$, rétrosynthèse d'une molécule cible,
2. $A \xrightarrow{?} B$, synthèse d'une molécule cible à partir de molécules spécifiées,
3. $A \xrightarrow{?} ?$, transformation un peu au hasard d'une (ou plusieurs) molécule de départ.

OASIS se veut un **O**util d'**A**ide pour un **S**ystème **I**nteractif de **S**ynthèse. Dans sa conception, nous avons particulièrement insisté sur une mise en place rationnelle des dispositifs de base : l'expression et l'organisation des connaissances ainsi que leurs mécanismes d'interprétation, ceci en formulant bien le problème. Ces dispositifs de base constituent le fondement du système dont la souplesse et la robustesse conditionnent sa fonctionnalité finale et son évolution. En effet, une organisation flexible des connaissances est à la base de leur bonne exploitation de et permet au système de s'adapter à la croissance de sa base de connaissances tout en gardant sa performance. De même, la rationalisation des mécanismes d'interprétation de base conditionne l'efficacité et l'extension ultérieure du système, ainsi que la facilité d'implantation de diverses heuristiques de contrôle.

Une meilleure conception quant à la représentation des connaissaces et à la mise en œuvre des mécanismes d'inférence de base constitue l'un des points forts du système OASIS par rapport aux autres systèmes existants.

Nous avons d'abord développé le modèle de *l'espace d'états avec abstraction de concepts* pour formuler le domaine du problème en question. Il s'agit d'un développement ou d'un affinement de la notion de *l'espace d'états* classique, en y introduisant un niveau de *concepts d'abstraction*, par rapport au niveau des *concepts de base* habituels. Ce nouveau modèle présente une généralité au-delà du présent problème de la synthèse.

En s'appuyant sur ce modèle, le choix final sur la représentation des connaissances dans OASIS est une approche mixte faisant intervenir conjointement :

- des objets structurés ou *frames*,
- des clauses de Horn,
- des règles de production,

qui forment une association naturelle pour exprimer l'expertise du domaine (voir le chapitre 3 pour les connaissances impliquées). Les mécanismes d'inférences correspondants mettent en œuvre :

- la propriété d'héritage des objets structurés,
- la démonstration de théorèmes en calcul des prédicats,
- l'interprétation des règles de production.

Le tout forme le fondement du système faisant preuve de flexibilité et de puissance.

Bien que ne faisant pas l'objet de cette thèse, la mise en place d'un mécanisme d'apprentissage est une partie importante du projet OASIS et elle est poursuivie par Laurent Pierron (voir dans [Haton 87,Sun 88]). L'idée est de permettre au système d'enrichir sa base de connaissances de façon automatique à travers des inteactions avec l'expert-utilisateur du système.

Le langage d'implantation d'OASIS est LE_LISP version 15.2 [Chalioux 86].

4.2 Modèle de l'espace d'états avec abstraction de concepts

Nous développons ici la notion d'espace d'états avec abstraction de concepts, qui est à la base de notre approche dans OASIS du problème de la synthèse. Cependant, l'utilité de cette notion va au-delà du présent problème.

4.2.1 Représentation dans l'espace d'états classique

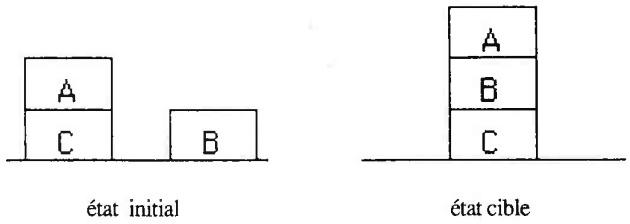


Figure 4.1. Un problème du monde des blocs

Afin d'introduire la notion d'espace d'états avec abstraction de concepts, rappelons d'abord la notion d'espace d'états classique [Fikes 71,Winston 84,Shirai 87]. A titre d'illustration, considérons un problème simple auquel est confronté un robot dans le monde des blocs. Celui-ci est illustré sur la figure 4.1. Le robot se trouve dans un état initial et il doit le transformer en un état cible, en manipulant les blocs.

Un *état* peut être représenté par une formule logique ou fbf (formule bien formée) qui est ici une conjonction de prédicats. Ainsi, pour l'état initial, on a :

```
SURTABLE(C), SURTABLE(B), SUR(A,C), DECOUVERT(A), DECOUVERT(B)
```

et pour l'état cible :

```
SURTABLE(C), SUR(B,C), SUR(A,B), DECOUVERT(A)
```

La façon de manipuler les blocs est définie par les *opérateurs*. La description d'un opérateur a trois composantes :

- une précondition,
- une liste des suppressions,
- une liste des adjonctions.

On a ici comme opérateurs :

```
depiler-poser(x,y) : prendre le bloc x de sur y
                     et le poser sur la table
  precondition : SUR(x,y), DECOUVERT(x)
  suppressions : SUR(x,y)
  adjonctions :  SURTABLE(x), DECOUVERT(y)

depiler-empiler(x,y,z) : prendre le bloc x de sur y
                         et le poser sur z
  precondition : SUR(x,y), DECOUVERT(x), DECOUVERT(z)
  suppressions : SUR(x,y), DECOUVERT(z)
  adjonctions :  SUR(x,z), DECOUVERT(y)

monter(x,y) : prendre x de sur la table et le poser sur y
  precondition : SURTABLE(x), DECOUVERT(x), DECOUVERT(y)
  suppressions : SURTABLE(x), DECOUVERT(y)
  adjonctions :  SUR(x,y)
```

Le problème consiste à trouver une suite d'opérateurs à appliquer successivement sur l'état initial pour arriver à l'état cible. La résolution du problème passe par un certain nombre d'états intermédiaires et constitue donc une exploration de *l'espace d'états*. Pour la solution du présent problème, on peut trouver la suite d'opérateurs suivante :

```
depiler-poser(A,C), monter(B,C), monter(A,B).
```

Cette description du problème en terme d'états et d'opérateurs est très générale et couvre une large gamme de problèmes.

En résumé, la formalisation d'un problème dans l'espace d'états classique a deux composantes :

1. la représentation des états,
2. la représentation des opérateurs.

4.2.2 Concepts d'abstraction

L'espace d'états simple que l'on vient de voir est caractérisé par l'existence d'un seul niveau de concepts, que l'on pourrait appeler *concepts de base*. Ce sont ces concepts de base sous forme de prédicats qui sont utilisés dans la description des états comme dans la description des opérateurs. Ce formalisme se prête bien aux problèmes impliquant des

concepts simples, tels que le monde des blocs dont la représentation ne met en jeu que des relations spatiales entre les blocs.

Cependant il se révèle insuffisamment puissant pour formaliser les problèmes dont l'expertise s'appuie sur des concepts sophistiqués, tels que le présent problème de la synthèse chimique. Dans ce dernier, les états sont des molécules et les opérateurs sont des réactions ou transformations chimiques, et un problème de synthèse consiste à rechercher une suite ou une arborescence d'opérateurs. Les *concepts de base* dans la spécification des états – molécules – sont principalement :

- les atomes et les liaisons entre les atomes.

Or ces concepts ne sont pas toujours facilement accessibles, en raison de leur granularité trop fine pour certaines situations. Dans ce cas, on préfère introduire un autre niveau de concepts – *concepts d'abstraction* – au-dessus de ceux utilisés dans la spécification des états ou *concepts de base*, de sorte que dans certaines situations, notamment dans la description des opérateurs, on peut utiliser aussi bien les concepts d'abstraction – pour représenter des informations de haut niveau – que les concepts de base.

Dans la synthèse chimique, si les

- atomes et liaisons

correspondent à notre appellation *concepts de base*, les *concepts d'abstraction* correspondent à :

- groupements fonctionnels, aromacité, électro-attraction, etc.

Dans la description d'une réaction, on utilise aussi bien les concepts de base que les concepts d'abstraction.

4.2.3 Liens entre concepts d'abstraction et concepts de base

Si l'on considère les concepts de base (CB) comme des modules de construction de base, les concepts d'abstraction (CA) seront de plus gros modules composés. Cependant, cette analogie ne permet pas d'apprécier les CA à leur juste valeur. En effet, l'utilité d'un CA est surtout justifiée lorsqu'il a plusieurs variétés, c'est-à-dire plusieurs combinaisons possibles de CB. Par exemple, en synthèse chimique, *l'aromacité* ne désigne pas une seule structure, mais sert de terme générique pour l'ensemble des structures présentant la propriété aromatique.

Etant donné qu'un CA peut être défini en terme de CB, il est naturel de représenter les relations entre eux sous forme logique, compte tenu de leur utilisation ultérieure. En traduisant chaque CA et CB par un prédicat, ces relations peuvent être formalisées au moyen des clauses de Horn (clauses PROLOG), de la manière suivante :

```
CA1 <- CB1, CB2, CA2.
CA1 <- CB3, CB4.
CA2 <- ...
```

Il s'agit d'une formalisation souple et puissante. Lorsqu'un CA a plusieurs variétés, il suffit de mettre autant de clauses, ceci dans le pire des cas. En effet, si ces variétés sont formulables de façon générale, il suffit de mettre des clauses comme si l'on programmait en PROLOG. La récursivité est, bien entendu, permise ; aussi, un CA peut figurer dans la partie droite d'une clause.

Illustrons cela sur un exemple simple. Supposons que l'on s'intéresse aux liens de parenté à l'intérieur de chacun de plusieurs groupes d'individus. Chacun des groupes se trouve complètement spécifié avec quatre concepts (prédicats) :

```
HOMME(x), FEMME(x), PARENT(x,y), MARIES(x,y),
```

ceux-ci constituent donc les CB du problème.

D'autre part, on peut avoir des tas de CA définis comme suit :

```
soeur(x,y) <- FEMME(x), PARENT(z,x), PARENT(z,y).
frere(x,y) <- HOMME(x), PARENT(z,x), PARENT(z,y).
tante(x,y) <- PARENT(z,y), soeur(x,z).
tante(x,y) <- MARIES(x,z), frere(z,u), PARENT(u,y).
...
```

L'introduction des CA permet ainsi de formuler des problèmes de manière directe et facile, sans nécessairement passer par des concepts fins. Par exemple, on peut utiliser maintenant directement le concept **tante** dans la description d'un problème sans détailler du coup les diverses variétés dans sa définition.

4.2.4 Utilisation des concepts d'abstraction

Les informations de haut niveau représentées par les concepts d'abstraction ont l'avantage d'être plus accessibles, et du point de vue de la méthodologie des systèmes experts, ceux-ci conduisent à une stratégie de représentation des connaissances qui fait preuve

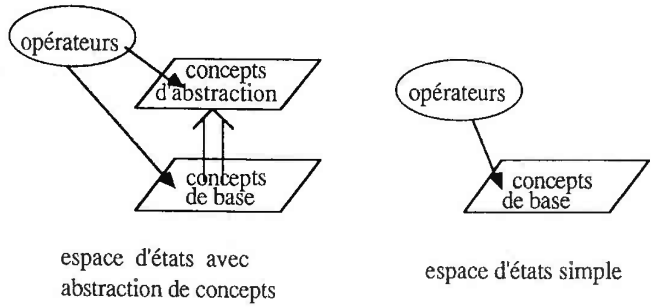


Figure 4.2. Espaces d'états avec et sans abstraction de concepts

de souplesse et de puissance. Plus encore, dans certains problèmes mettant en jeu des connaissances très complexes comme celui de la synthèse chimique, l'introduction des concepts d'abstraction se révèle indispensable.

Les opérateurs sont les premiers à profiter de l'accès facile aux informations de haut niveau rendu possible par les concepts d'abstraction. Ainsi, dans la précondition d'un opérateur, on a le droit d'utiliser deux niveaux de concepts, ce qui facilite énormément la rédaction et la gestion des connaissances. Par exemple, on peut mettre, dans la synthèse chimique, un concept tout court tel que *l'aromacité*, alors que l'on définit les diverses variétés possibles de celle-ci dans la base des clauses de Horn, On peut aisément ajouter une autre variété (une clause) sans pour autant toucher à l'opérateur.

En matière d'interprétation d'un opérateur, sa précondition constitue une conclusion logique à prouver à partir des axiomes que constituent l'état du problème et la base des clauses.

La représentation des méta-connaissances qui régissent le choix d'opérateurs à un état donné présente, bien entendu, également l'intérêt de s'appuyer sur les deux niveaux de concepts.

4.2.5 Espace d'états avec abstraction de concepts

Avec l'introduction des concepts d'abstraction, on se trouve ainsi dans un *espace d'états avec abstraction de concepts*, qui implique un niveau de concepts de plus par rapport à l'espace d'états classique, comme l'illustre la figure 4.2.

Les concepts d'abstraction sont soutenus par les concepts de base au moyen des clauses traduisant leurs définitions. La représentation des opérateurs s'appuie sur les deux niveaux de concepts.

A noter que dans [Sacerdoti 74,Shirai 87], on rencontre également la notion d'*abstraction*. Mais il ne s'agit pas là d'une abstraction de *concepts*. L'idée est d'associer un coefficient d'importance à chacun des prédicats décrivant un état. La résolution d'un problème (la planification) débute en tenant compte uniquement des prédicats ayant le niveau d'importance le plus élevé – donc dans l'espace le plus abstrait. Si cette planification réussit, on va descendre d'un niveau en tenant compte des prédicats du deuxième niveau d'importance, pour vérifier ou rejeter les plans obtenus précédemment. La planification ainsi se poursuit. Conceptuellement, il n'y a donc toujours qu'un seul niveau, et la notion d'*espace d'abstraction* n'apparaît que lorsque l'on affecte des niveaux d'importance aux prédicats.

En résumé, la formalisation d'un problème dans *l'espace d'états avec abstraction de concepts* a trois composantes :

1. la représentation d'états en terme de concepts de base,
2. la définition des concepts d'abstraction par les clauses de Horn,
3. la représentation d'opérateurs en terme de concepts de base et/ou de concepts d'abstraction.

A comparer avec celle dans *l'espace d'états classique* :

1. la représentation d'états en terme de concepts de base,
2. la représentation d'opérateurs en terme de concepts de base.

4.2.6 Mécanismes d'inférences

Etant donné un problème formulé dans l'espace d'états, sa résolution se déroule autour d'un *interprète d'opérateurs* qui assure la transition d'un état à un autre, et d'un *contrôleur* qui enchaîne les transitions.

Pour qu'un opérateur soit applicable sur un état, il faut que sa précondition soit la conséquence logique de la fbf représentant l'état. Dans l'espace d'états avec abstraction de concepts, deux niveaux de concepts (prédicats) interviennent dans la précondition d'un opérateur, ce qui fait que celle-ci est à prouver comme la conséquence logique de la fbf modulo les clauses de Horn qui donnent la définition des concepts d'abstraction. Le fonctionnement de *l'interprète d'opérateurs* fait ainsi intervenir un *démonstrateur de théorèmes* se fondant sur les clauses de Horn.

Le *contrôleur*, l'*interprète d'opérateurs* et le *démonstrateur de théorèmes* forment ainsi les mécanismes d'inférences d'un résolveur de problèmes dans l'espace d'états avec abstraction de concepts.

Cependant, le souci de gagner en efficacité conduit à l'emploi d'un autre outil supplémentaire : un *évaluateur partiel*.

Comme on vient de le voir, le démonstrateur de théorèmes est destiné à prouver des théorèmes en prenant pour axiomes l'état courant du problème et les clauses de Horn générales ; lorsqu'il y a plusieurs théorèmes à prouver en même temps sur le même état, on peut s'attendre à une médiocrité de performance, du fait de la généralité des clauses. Il est alors souhaitable de spécialiser celles-ci par rapport à l'état en question avant de les utiliser plusieurs fois dans la démonstration des théorèmes. Ce processus de *spécialisation* est en fait une *évaluation partielle* des clauses.

En résumé, les mécanismes d'inférences dans le modèle de l'espace d'états avec abstraction de concepts ont quatre composantes :

1. un contrôleur,
2. un interprète d'opérateurs,
3. un démonstrateur de théorèmes se fondant sur les clauses de Horn,
4. un évaluateur partiel pour clauses de Horn,

comme l'illustre la figure 4.3.

4.3 Représentation du problème de la synthèse chimique

Les problèmes en synthèse chimique se trouve aisément représentés dans l'espace d'états avec abstraction de concepts.

Tout d'abord, les molécules sont des états, et les réactions sont des opérateurs. Un état peut être une ou plusieurs molécules. L'application d'un opérateur sur un état permet de

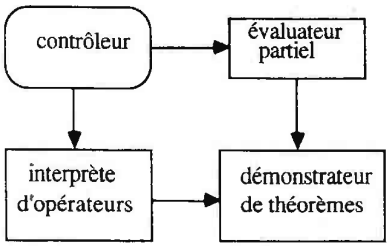


Figure 4.3. Mécanismes d'inférences dans le modèle de l'espace d'états avec abstraction de concepts

passer dans un autre état. Le problème à résoudre implique un parcours dans l'espace des états dont chaque pas est réalisé par un opérateur, le but étant de trouver un chemin ou une arborescence de chemins répondant aux conditions initiales du problème.

Réexaminons les quatre types d'expertise impliqués dans la synthèse, exposés au chapitre 3 :

1. Les principes structurels. Ce sont les connaisances nécessaires pour spécifier la structure d'une molécule : les atomes, leur ionicité, les liaisons.
2. Les propriétés moléculaires. Certaines sous-structures dans une molécule peuvent présenter une certaine propriété.
3. Les réactions chimiques. Ces opérateurs chimiques sont essentiels dans la planification d'une synthèse.
4. Les stratégies de synthèse. Ce sont des stratégies de recherche dans le processus de planification.

Le premier type de connaissances constitue les *concepts de base* pour la description des

molécules et donc des états. Le deuxième type forme des *concepts d'abstraction* et sera mis sous forme de clauses de Horn. Le troisième type comprend les opérateurs de transition d'états. Ces trois types trouvent donc bien leur place dans le modèle de l'espace d'états avec abstraction de concepts.

Le quatrième type de connaissances se trouve au méta-niveau et concerne le contrôle ou le choix d'opérateurs à appliquer à un état donné au cours de la résolution d'un problème.

Nous allons voir de plus près la formalisation de ces connaissances.

4.3.1 Les molécules ou les états

Les concepts de base étant les *atomes*, les *liaisons* et l'*ionicité*, une molécule peut être ainsi mise sous la forme d'une fbf : une conjonction de prédicats traduisant ces trois types de concepts. Cependant, le formalisme des objets structurés ou *frames* offre un choix plus approprié grâce à son caractère structuré. Toutefois, il faut considérer qu'une molécule ainsi représentée est toujours une fbf, mais sous une meilleure forme.

Dans notre approche, nous définissons d'abord la classe d'objets *atome* qui regroupe les attributs d'un atome chimique dans un même formalisme.

```
(defstruct atome
    nom
    atome
    numero
    ...
)
```

Nous définissons ensuite la classe d'objets *nœud* comme sous-classe de la classe *atome* :

```
(defstruct #:atome:noeud
    voisin
    ion
)
```

On est à présent en mesure de donner la représentation d'une molécule ou d'un état :

> Une **molécule** ou un **état** est un ensemble d'objets *nœud*. Celui-ci traduit implicitement une conjonction de prédicats.

Le fait que *nœud* soit une sous-classe de *atome* permet à un objet *nœud* d'avoir une identité atomique, par le biais de la propriété d'héritage. Les attributs appartenant en

propre à *nœud* spécifient la charge électrique d'un objet *nœud* dans une molécule – *ion* – ainsi que ses liaisons avec les autres objets *nœud* – *voisin*. Cette représentation est donc conforme au fait qu'une molécule est un ensemble d'atomes interconnectés par des liaisons et éventuellement chargés.

4.3.2 Les concepts d'abstraction sous forme de clauses

Les concepts d'abstraction sont définis à partir des concepts de base qui sont utilisés pour représenter les états. Ils représentent des informations concernant des sous-structures moléculaires.

L'apparence prédicative des concepts de base est assurée par deux types de prédicats : `(_x.attr _y)` et `(voisin _x _y _n)`. Le premier traduit le fait que l'attribut `attr` de l'objet *nœud* _x est _y, et le second indique que les deux objets *nœud* _x et _y sont reliés par une liaison de type _n.

Voici quelques exemples de définition de concepts d'abstraction à partir des concepts de base – donc en terme de `(_x.attr _y)` et `(voisin _x _y _n)` :

```
(e-attracteur _x) <- (CO _x _).
(e-attracteur _x) <- (_x.atome F).
(CO _x _y) <- (voisin _x _y 2) (_x.atome C) (_y.atome O).
```

où les symboles précédés d'un "_" sont des variables.

La dernière clause définit le groupement fonctionnel *carbonyle* ou CO qui est la sous-structure suivante :

La première clause donne une variété de ce qu'on appelle *électro-attracteur*, qui est le carbonyle. La deuxième en donne une autre, en affirmant qu'un atome fluor ($-F$) est aussi un électro-attracteur.

4.3.3 Les opérateurs

La description d'un opérateur a trois composantes principales :

```
OPERATEUR
```



```
lhs
rhs
certitude
```

`lhs` et `rhs` (*left-hand side* et *right-hand side*) ont tous deux pour valeur une conjonction de littéraux relevant des concepts de base et/ou des concepts d'abstraction. `lhs` et `rhs` définissent ainsi chacun une sous-structure moléculaire (un *pattern*). Lorsqu'on applique l'opérateur sur un état (une ou plusieurs molécules), on cherche une sous-structure de celui-ci qui corresponde à `lhs`. Le nouvel état éventuel sera obtenu en substituant la sous-structure `rhs` à celle correspondant à `lhs` dans l'état courant. A noter que ce formalisme est symétrique, permettant également l'application de l'opérateur dans le sens inverse (de droite à gauche).

Il est à remarquer que la présence des prédicats correspondant à des concepts d'abstraction dans un opérateur est destinée uniquement à participer à des préconditions à remplir (démontrer), alors que les modifications à effectuer, une fois l'opérateur révélé applicable, doivent être prescrites au moyen des prédicats de base uniquement, ce qui est logique.

La composante `certitude` permet de trouver une valeur d'évaluation de la facilité d'application de l'opérateur, en fonction du contexte d'application.

Par exemple, la réaction suivante :

a pour représentation :

```
lhs: (voisin _2 _1 2) (voisin _1 _6 1) (voisin _6 _5 2)
     (voisin _3 _4 2)
rhs: (voisin _2 _1 1) (voisin _1 _6 2) (voisin _6 _5 1)
     (voisin _5 _4 1) (voisin _4 _3 1) (voisin _3 _2 1)
certitude: ((voisin _4 _7) (e-attracteur _7) 0.8)
           (vrai                             0.5)
```

La partie `certitude` indique que, lorsque l'atome _4 se trouve relié à un électro-attracteur, la facilité de la réaction est évaluée à `0.8`, et autrement à `0.5`. La valeur `0.8` correspond

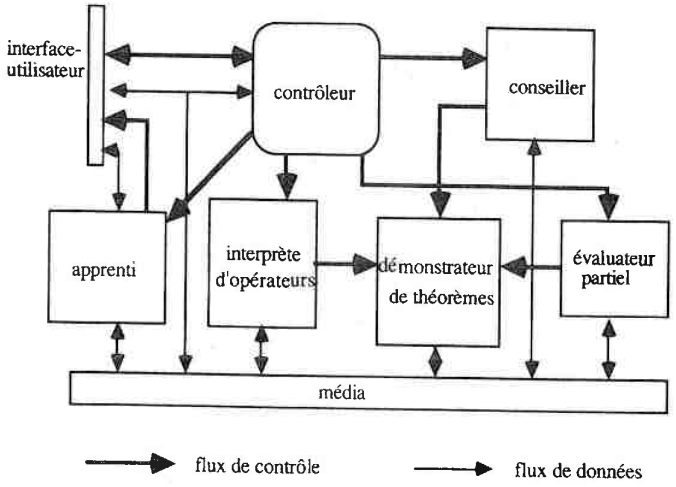


Figure 4.4. Architecture du système OASIS

en fait au schéma :

où Z désigne un électro-attracteur.

4.4 Architecture du système expert OASIS

L'architecture du système expert OASIS est présentée en figure 4.4.

Les connaissances qui n'y sont pas explicitées sont réparties entre quatre groupes :

1. les objets *atomes*,
2. les clauses de Horn,
3. les opérateurs,
4. les méta-règles.

Etant donné un état et un opérateur, l'interprète d'opérateurs donne le nouvel état à l'issue de l'application de l'opérateur. Son fonctionnement s'appuie sur le *démonstrateur de théorèmes* qui est chargé de verifier la précondition de l'opérateur en prenant pour axiome l'etat courant du problème et les clauses de Horn.

Si *l'interprète d'opérateurs* permet de convertir un état en un autre, le *contrôleur* enchaîne les pas d'avancement, dans le but de résoudre un problème précis de synthèse.

Le *démonstrateur de théorèmes* constitue un outil de base pour vérifier (prouver) quelque chose sur un état du problème. Il repose sur le calcul des prédicats, se limite aux clauses de Horn et manipule directement les objets structurés.

La présence de l'*évaluateur partiel* est facultative, en ce sens qu'il ne fait qu'améliorer l'efficacité du fonctionnement du *démonstrateur*. L'idée consiste, étant donné un état du problème, à spécialiser (partiellement évaluer) les clauses par rapport à cet état, de sorte que le processus de démonstration de théorèmes sera plus efficace après, tant que l'on ne change pas d'état.

Le *conseiller* est prévu comme étant un interprète de méta-connaissances, celles-ci sont des stratégies régissant le choix d'opérateurs au cours de la résolution d'un problème.

L'*apprenti* doit suivre le raisonnement du système, et à travers des interactions avec l'expert-utilisateur, enrichir la base de connaissances du système.

L'*interface* assure, avant tout, deux fonctions : la saisie graphique d'opérateurs chimiques et de molécules, et leur affichage.

Enfin, le *média* est une structure de données globale destinée à faciliter la communication entre les modules. Il enregistre les données initiales et intermédiaires du système, et son accès est libre pour tous les modules.

4.5 Conclusion

Nous avons d'abord développé le modèle de l'espace d'états avec abstraction de concepts qui décrit le domaine d'un problème en trois composantes :

1. la représentation des états en terme de *concepts de base* ;
2. la définition, au moyen des clauses de Horn, des *concepts d'abstraction* à partir des concepts de base ;
3. la description des opérateurs de transition en s'appuyant à la fois sur les concepts de base et sur les concepts d'abstraction.

Ce modèle est ainsi caractérisé par une abstraction de concepts et destiné à formaliser des problèmes impliquant des concepts compliqués.

Le problème de la synthèse chimique se trouve bien représenté dans ce modèle. Ainsi,

1. les molécules constituent les états et sont représentées avec les concepts de base : nature des atomes et liens entre eux ;
2. les concepts chimiques tels que les groupements fonctionnels et les termes génériques (électro-attracteur, etc.) sont définis sous forme de clauses de Horn à partir des concepts de base ;
3. les réactions constituent les opérateurs et leur description fait appel à la fois aux concepts de base et aux concepts d'abstraction.

Cette vision du problème a permis d'organiser les connaissances de façon souple. Les mécanismes d'inférences correspondants ont pu faire intervenir diverses techniques en vue d'une implantation efficace.

Par rapport aux systèmes existants, le système OASIS est ainsi conçu et organisé en se fondant sur un modèle global, ce qui permet aux différents modules qui en ressortent ainsi qu'à leur liens d'acquérir une sémantique claire. Ceci est à la base d'un système efficace.

5
Représentation des connaissances

5.1 Problème général de la représentation des connaissances

Considéré par rapport à la connaissance, un système expert se présente sous deux aspects essentiels : la représentation de la connaissance d'une part et son utilisation d'autre part. Le premier concerne le développement d'une notation suffisamment précise pour exprimer l'expertise d'un domaine ; le deuxième est la manipulation de l'expertise une fois exprimée. Un choix adéquat en matière de représentation des connaissances conditionne d'une part la possibilité de traduire de façon assez précise l'expertise en question, et d'autre part sa bonne exploitation.

5.1.1 Divers formalismes de représentation

Formalisme logique

Il s'agit généralement de la logique du premier ordre (voir [Chang 73] par exemple). Ce formalisme consiste à représenter les connaissances sous forme de formules logiques, en utilisant les notions de *constante*, *variable*, *fonction*, *prédicat*, *connecteur logique*, et *quantificateur*.

Vu le cadre général de la logique, il s'agit d'un formalisme très puissant, qui offre l'avantage d'être simple et uniforme. Cependant, le plus grand intérêt de ce formalisme est lié au fait qu'il fournit non seulement un mode de représentation, mais encore un mécanisme d'interprétation universel qui est la démonstration automatique de théorèmes.

Le langage PROLOG [Kowalski 74], reposant pour l'essentiel sur le calcul des prédicats restreint aux clauses de Horn, en témoigne la puissance ainsi que la promesse.

Toutefois, ce formalisme n'est pas universel. Dû à son incapacité d'intégrer des heuristiques et à la limite de la logique du premier ordre, il ne peut faire face de façon satisfaisante à certains problèmes. Citons, par exemple, les problèmes du type d'espace d'états.

Dans ces derniers, si chaque état est une fbf (formule bien formée), l'application d'un opérateur à un état ne peut être formalisée en logique du premier ordre, car elle implique l'invalidation de la fbf correspondant à l'état de départ. Pour pallier ce problème, Nilsson [Nilsson 71] avait proposé une approche qui consistait à associer à chaque état un numéro d'identification dès que celui-ci était engendré, de sorte que les prédicats décrivant un état prenaient tous un argument supplémentaire qui devait être le numéro d'identification de l'état en question. Ce n'était bien entendu pas une solution satisfaisante.

On reproche également à une base de connaissances sous forme de formules logiques de manquer de structuration. Mais en réalité, il ne s'agit là pas d'un aussi grave défaut qu'il paraît. C'est la simplicité et l'uniformité du formalism logique qui devrait l'emporter au final, comme on le verra par la suite.

De toute façon, il faut reconnaître le rôle central du formalisme logique dans la représentation des connaissances. En effet, dans les divers autres modes de représentation, on retrouve toujours des éléments du formalisme logique, ce qui se voit surtout directement dans les règles de production.

Règles de production

Si le formalisme logique de représentation de connaissances s'inscrit dans un cadre général de la formalisation et aussi de la résolution de problèmes, les *règles de production* (voir par exemple [Davis 75]) – ou simplement les *règles* – n'ont pas une telle prétention. Elles n'ont ainsi pas de syntaxe précise, ni de sémantique formelle, ce qui n'est d'ailleurs pas leur but. En effet, leur vocation est de pouvoir exprimer les connaissances d'un problème précis, en adoptant une syntaxe appropriée, et éventuellement en cohabitation avec d'autres formalismes de représentation de connaissances tels que les objets structurés ou les réseaux sémantiques.

Une règle de production a la structure générale suivante :

```
SI conditions ALORS actions.
```

qui peut être interprétée ainsi : si l'état courant du problème remplit la partie `conditions`, alors `actions` sera déclenchée conduisant à la modification de l'état courant.

Il s'agit donc d'un formalisme très souple, facile à adapter à des problèmes particuliers. Parallèlement, le mécanisme d'interprétation correspondant doit également s'adapter aux cas particuliers.

On choisit une syntaxe précise en fonction de ses besoins. Par exemple, la partie `conditions` peut être sous forme d'une formule logique stricte. Mais elle peut également

faire intervenir d'autres formalismes, comme l'appel d'une fonction booléenne sous forme procédurale par exemple. De même pour la partie `actions`, qui peut déclarer des faits à ajouter et/ou à supprimer par rapport à l'état du problème, mais qui peut également déclencher directement une procédure qui modifiera l'état du problème dans les coulisses.

En raison de sa structure générale, de sa souplesse et de son esprit d'ouverture vis-à-vis d'autres formalismes, les règles de production sont très en faveur dans les systèmes experts.

Le défaut des règles de production de ne pas pouvoir structurer les connaissances de manière directe peut être compensé dans une certaine mesure par sa volonté de cohabiter avec d'autres formalismes ayant le caractère structuré.

Réseaux sémantiques et objets structurés

Contrairement au formalisme logique et aux règles de production, les réseaux sémantiques et les objets structurés (*frames*) sont nés du souci de représenter les connaissances directement sous forme structurée.

Le réseau sémantique, initialement proposé par Quillian [Quillian 68], considère une base de connaissances comme étant un ensemble d'objets et d'associations entre objets. Ce formalisme a ainsi une représentation graphique évidente où les nœuds sont les objets et les arêtes les relations entre eux.

Les principes de structurer la connaissance sont au cœur de l'appproche du réseau sémantique. Cette structuration peut se faire selon quatre principes majeurs [Mylopoulos 83] : la classification, l'agrégation, la généralisation, et les partitions.

La notion de *frames* ou objets structurés a été proposée par Minsky [Minsky 75]. Un objet structuré simple est une entité qui rassemble plusieurs champs pour représenter des situations stéréotypées. Ce formalisme considère une base de connaissances comme étant un ensemble d'objets. On peut définir des objets de façon hiérarchique, et il y aura alors la notion *d'héritage de propriétés*. Ce formalisme permet d'avoir une représentation économique de certaines situations.

La notion d'objets structurés a donné lieu aux langages de programmation orientée-objets, qui offrent le mode de la programmation distribuée (voir [Ferber 86] pour un exemple).

Représentation procédurale

La représentation des connaissances sous forme procédurale voit la base de connaissances comme un ensemble d'agents actifs (procédures ou fonctions).

On choisit généralement le formalisme procédural pour représenter des connaissances figées. En raison de son caractère procédural, la représentation procédurale de connaissances a l'avantage d'être efficace à l'exécution, en évitant de nombreuses recherches que devrait effectuer un interprète dans le cas de connaissances déclaratives.

5.1.2 Conclusion

Choix d'un mode de représentation

Outre le formalisme logique qui est une approche générale, les autres formalismes constituent des outils souples, faciles à adapter à chaque cas spécifique.

Les règles de production sont très puissantes et recouvrent beaucoup de situations. Les réseaux sémantiques insistent sur les relations entre les unités de connaissances. Les objets structurés permettent d'avoir une vue hiérarchique des connaissances. Remarquons que ces formalismes sont ouverts les uns vis-à-vis des autres.

Ainsi, pour les problèmes qui ne peuvent être résolus de façon satisfaisante dans l'approche logique (heuristiques, problèmes d'efficacité, économie de représentation, etc.), on peut adapter ces formalismes moins formels à des besoins spécifiques. Une solution satisfaisante à un problème complexe peut impliquer la coopération de plusieurs d'entre eux, pour avoir une approche mixte, comme dans [OHare 85,Takenouchi 87], ainsi que le présent système OASIS qui fait intervenir le calcul des prédicats, les objets structurés et les règles de production.

Formalisme logique et structuration des connaissances

Le formalisme logique s'inscrit dans le cadre général de la représentation et de la résolution de problèmes. Dans ce contexte, son manque de structuration n'est en fait pas un problème en soi.

En effet, le fait que l'on exige une structuration d'une base de connaissances est né du souci d'avoir une exploitation plus efficace de celle-ci, c'est l'idée du formalisme des réseaux sémantiques par exemple.

Dans l'approche logique, le problème de l'efficacité est abordé d'un point de vue général. Une idée qui semble très prometteuse [Kowalski 76,Matwin 87] consiste à prétraiter les

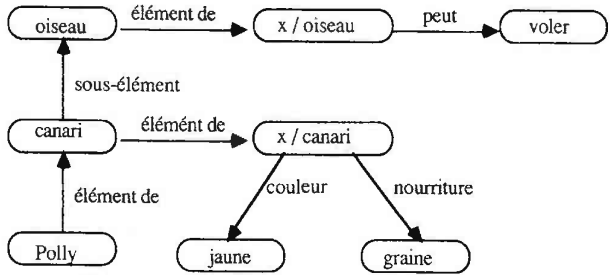


Figure 5.1. Réseau sémantique

clauses avant de se lancer dans un processus de démonstration de théorèmes. Le problème consiste à savoir comment *anticiper* sur la déduction de la clause vide.

Le problème dans le cas général étant difficile à résoudre, nous nous contentons ici de tracer le parallèle entre cette idée dans l'approche logique et la structuration dans les réseaux sémantiques, pour aboutir à la conclusion que le raisonnement sur le formalisme logique, moyennant une phase de prétraitement, peut être aussi efficace que sur un réseau sémantique.

Prenons un exemple qui était utilisé dans [Shirai 87] pour illustrer la notion du réseau sémantique :

1. Un oiseau peut voler.
2. Le canari est un oiseau.
3. La couleur d'un canari est jaune.
4. Un canari mange des graines.
5. Polly est un canari.

Le réseau sémantique correspondant à ces connaissances est présenté sur la figure 5.1. On peut s'attendre à ce que le raisonnement fondé sur ce réseau soit très efficace, grâce à cette structuration globale des connaissances.

Abordons maintenant ce même problème en calcul des prédicats. La description du problème correspond à la formalisation suivante :

1. $\forall x\ oiseau(x) \rightarrow voler(x)$.
2. $\forall x\ canari(x) \rightarrow oiseau(x)$.
3. $\forall x\ canari(x) \rightarrow couleur(x, jaune)$.
4. $\forall x\ canari(x) \rightarrow nourriture(x, graines)$.
5. $\forall x\ canari(Polly)$.

On peut en obtenir la forme *canonique* :

1. $\neg oiseau(x), voler(x)$.
2. $\neg canari(y), oiseau(y)$.
3. $\neg canari(z), couleur(z, jaune)$.
4. $\neg canari(u), nourriture(u, graines)$.
5. $canari(Polly)$.

où les virgules ont remplacé les $\vee$ (OU) et les quantificateurs universels ont été omis.

Les problèmes en calcul des prédicats peuvent souvent se ramener à la démonstration d'un théorème. Par exemple, si l'on demande : "Polly peut-il voler ?" Il suffit de prouver que $voler(Polly)$ est une conclusion logique des clauses ci-dessus. La démonstration se fait par l'absurde : on cherche à démontrer que l'ensemble des clauses, une fois $\neg voler(Polly)$ intégrée, devient contradictoire, ce qui revient à en déduire la clause vide selon le principe de résolution.

Afin de restreindre l'espace de recherche, on souhaite pouvoir anticiper sur la déduction de la clause vide. Un premier niveau d'anticipation est facile à faire et consiste simplement à prévoir *isolément* les résolutions possibles. Il suffit d'étiqueter, par leur unificateur, toute paire de littéraux qui sont potentiellement complémentaires, c'est-à-dire, deux littéraux unifiables mais ayant des signes opposés. Le résultat est un *graphe de connexion* des clauses.

Ainsi, pour les clauses de l'exemple précédent, on obtient le graphe de connexion qui est présenté sur la figure 5.2. En le comparant avec le réseau sémantique en figure 5.1, on voit qu'il offre le même degré de structuration que le réseau sémantique. En fait, la démonstration de théorèmes à l'aide du graphe de connexion peut avoir une performance comparable à celle du raisonnement utilisant le réseau sémantique.

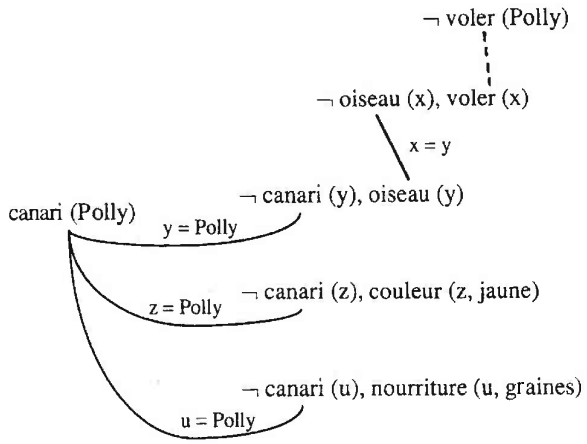


Figure 5.2. Graphe de connexion des clauses

Pour prouver : "Polly peut voler", il suffit d'intégrer $\neg voler(Polly)$ dans le graphe de connexion, en vue de la déduction de la clause vide. La solution y apparaît évidente et fait intervenir les quatre clauses se trouvant en haut de la figure 5.2.

On peut tirer la conclusion suivante : le réseau sémantique offre une structuration de connaisances directe et propre à un problème spécifique et le raisonnement correspondant implique un parcours du réseau ; dans le formalisme logique, les connaissances ne se présentent pas directement sous forme structurée, mais le démonstrateur de théorèmes pourrait se charger de la structuration indépendamment d'un problème spécifique.

5.2 Le système OASIS

Comme on l'avait vu au cours des deux chapitres précédents, le problème de la synthèse chimique peut être formulé dans le modèle de l'espace d'états. Etant donné qu'une ou plusieurs molécules forment un état, la solution du problème consiste à rechercher des chemins dans l'espace d'états satisfaisant à certaines conditions initiales. La transition d'un état à un autre est assurée par un opérateur qui est une réaction chimique.

Dans le modèle de l'espace d'états avec abstraction de concepts, la description d'un

problème s'appuie sur trois composantes :

1. la représentation d'états en termes de concepts de base,
2. la définition des concepts d'abstraction sous forme de clauses de Horn,
3. la représentation d'opérateurs en termes de concepts de base et/ou de concepts d'abstraction.

En calquant le problème de la synthèse chimique sur ce modèle, la première composante correspond à la représentation des structures moléculaires, les *concepts de base* étant

- la nature des atomes, leur ionicité, et les liaisons entre eux.

Il s'agit donc de représenter l'objet molécule à partir de ces trois types de concepts. Notre choix consiste à adopter le formalisme des objets structurés en profitant de sa propriété d'héritage, pour avoir une représentation économique et efficace. L'accès aux objets se fait sous forme prédicative, pour avoir une uniformité et aussi une compatibilité avec les clauses de Horn et les opérateurs.

La deuxième composante correspond à la définition des concepts plus élaborés que les atomes, l'ionicité et les liaisons, tels que l'acide, l'électro-attracteur, etc. voire certaines configurations géométriques que peuvent présenter les molécules. Ces *concepts d'abstraction* sont définis sous forme de clauses de Horn, à partir des *concepts de base* ci-dessus.

La troisième composante correspond à la représentation des réactions chimiques ou des opérateurs. Celle-ci s'appuie aussi bien sur les concepts de base (les atomes, l'ionicité et les liaisons) que sur les concepts d'abstraction (l'acide, l'électro-attracteur, etc.).

Outre ces trois niveaux de connaissances, il y aura aussi un méta-niveau de connaissances correspondant aux stratégies de synthèse.

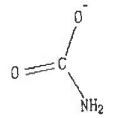
Ces quatre niveaux de connaissances constituent la base de connaissances du système OASIS.

5.3 Représentation des molécules en terme de concepts de base

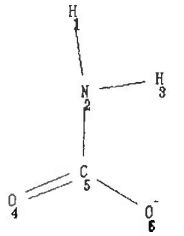
Il s'agit de trouver une représentation appropriée de la structure d'une molécule.

5.3.1 Une molécule est un graphe spécial

Examinons d'abord un exemple de molécule :



qui est en fait une représentation racourcie de la molécule suivante :



où nous avons donné un numéro d'identification à chacun des atomes pour faciliter l'explication par la suite.

Cet exemple, si simple soit-il, montre que la structure d'une molécule est un graphe et qu'il s'agit d'un graphe spécial :

1. Les nœuds du graphe ont un sens. Il s'agit des atomes chimiques : C, H, O, etc.
2. Un nœud peut aussi porter une valeur numérique : `0`, `+1`, `-1`, etc., qui représente la charge électrique (l'ionicité) de l'atome en question dans le contexte de la molécule.
3. Les arêtes ont aussi un sens. Ils s'agit des liaisons chimiques : 1 (liaison simple), 2 (liaison double), 3 (liaison triple), et peut-être aussi plus.

Toute représentation d'une molécule doit donc impliquer ces trois types de concepts ou *concepts de base* :

- la nature des nœuds, leur ionicité et les liaisons entre eux,

d'une manière ou d'une autre.

5.3.2 Représentation matricielle

Les systèmes existants utilisent généralement la représentation matricielle des molécules dite *matrice de connectivités*. Celle-ci présente l'avantage de pouvoir être implanté dans n'importe quel langage de programmation (en BASIC dans le système MARSEIL [Barone 82], et en FORTRAN et ASSEMBLEUR dans les systèmes LHASA [Corey 69, Johnson 85] et SECS [Wipke 78]).

Il faut deux matrices, une pour représenter les nœuds et l'autre pour les liaisons. Pour l'exemple de molécule précédent, la matrice pour les nœuds peut avoir la forme suivante :

nœud	1	2	3	4	5	6
atome	H	N	H	O	C	O
charge	0	0	0	0	0	-1

La matrice pour les liaisons sera :

nœud-de-départ	1	2	2	4	5
nœud-fin	2	3	5	5	6
type-de-liaison	1	1	1	2	1

Cette représentation est rigide et son utilisation est fastidieuse. Nous préférons plutôt, dans le système OASIS, une représentation symbolique.

5.3.3 Représentation sous forme d'une formule logique

Dans notre formulation du problème de la synthèse, nous considérons une molécule ou un état comme étant une fbf.

Les trois types de concepts utilisés dans la description d'une molécule peuvent se traduire par trois types de prédicats :

1. `(atome _ _)`. C'est un prédicat à deux arguments dont le premier désigne un nœud dans le graphe-molécule, et le deuxième sa nature atomique. Par exemple, `(atome 6 O)` traduit le fait que le nœud numéro 6 est un O (oxygène).

2. `(ion _ _)`. C'est un prédicat à deux arguments dont le premier désigne un nœud et le deuxième sa charge électrique (ou ionicité). Par exemple, `(ion 6 -1)` indique que le nœud 7 possède une charge électrique de `-1`.

3. `(voisin _ _ _)`. C'est un prédicat à trois arguments dont les deux premiers désignent deux nœuds et le dernier le type de la liaison (1, 2 ou 3) entre eux. Par exemple, `(voisin 4 5 2)` indique que les nœuds 4 et 5 sont reliés par une liaison double.

La représentation d'une molécule sous forme d'une fbf sera ainsi une conjonction de ces trois types de predicats. Pour la molécule précédente, on aura :

```
(atome 1 H) (ion 2 0) (atome 2 N) (ion 2 0) (atome 3 H) (ion 3 0)
(atome 4 O) (ion 4 0) (atome 5 C) (ion 5 0) (atome 6 O) (ion 6 -1)
(voisin 1 2 1) (voisin 2 3 1) (voisin 2 5 1) (voisin 4 5 2)
(voisin 5 7 1)
```

Cette représentation est plus directe que la représentation matricielle et se prête mieux à l'interprétation, grâce à sa nature symbolique.

Avant d'effectuer des déductions sur une molécule ainsi exprimée, il faut encore expliciter les connaissances concernant les atomes. En effet, un atome est caractérisé par plusieurs constantes qui lui sont propres, telles que son numéro atomique, sa masse, ses valences, son électro-négativité, etc. Par exemple, pour l'atome N (azote) (cf. un tableau periodique des éléments), on a les valeurs suivantes :

```
atome   N
numero  7
masse   14.01
...
```

On peut rajouter directement ces informations dans la représentation d'une molécule. Par exemple, pour le nœud 2 dans l'exemple précédent qui est un N, il faudra alors compléter :

```
(numero 2 7) (masse 2 14.01) ...
```

On peut également procéder de manière implicite en intégrant dans la base de connaissances les clauses suivantes :

```
(numero _x 7) <- (atome _x N).
(masse _x 14.01) <- (atome _x N).
...
```

En tout cas, le manque de structuration de cette représentation d'une molécule devient évidente. Par exemple, si l'on veut savoir la masse atomique d'un nœud, il faut parcourir les prédicats représentant la molécule, voire certaines clauses. Certes, on pourrait envisager de charger le mécanisme d'inférences d'effectuer une certaine structuration (cf. la discussion au 5.1.2), mais cela n'est pas rentable. En effet, les inférences à ce niveau se trouvant

tout en bas du raisonnement dans le cadre d'une approche heuristique, ce ne serait qu'une structuration locale.

Notre choix final consiste, tout en gardant la vision d'une molécule comme une fbf, à la représenter, non directement sous forme d'une conjonction de prédicats, mais de manière plus naturelle sous forme structurée. L'accès à une molécule se fait sous forme prédicative, comme s'il s'agissait d'une représentation en logique directe.

Il s'agit donc de structurer les connaissances en dehors du mécanisme d'inférences.

5.3.4 Choix final

La représentation finale des molécules utilise le formalisme des objets structurés (*frames*) en profitant de sa propriété d'héritage.

Nous allons d'abord définir la classe d'objets *atome* pour représenter les atomes chimiques :

```
(defstruct atome
    atome
    nom
    numero
    masse
    valence
)
```

Nous gardons dans la version actuelle du programme ces cinq attributs sans prétendre du tout qu'ils soient exhaustifs. Mais en cas d'insertion de nouveaux attributs, il suffit juste de les ajouter dans la définition de cette structure de données, sans avoir à toucher au reste du programme.

Les différents atomes chimiques sont ainsi des instances de la classe *atome* ou des *atomes*. Par exemple, on peut créer l'instance `N` (azote) avec les valeurs suivantes :

```
atome   N
nom     azote
numero  7
masse   14.01
valence (-3 +5)
```

A part l'attribut **valence** qui prend pour valeur une liste indiquant les valences possibles d'un atome, les autres attributs ont pour valeur soit un chiffre soit un symbole.

Nous définissons ensuite la classe d'objets *nœud* comme sous-classe de la classe *atome* :

```
(defstruct #:atome:noeud
    ion
    voisins
)
```

de sorte que les atomes dans le contexte d'une molécule puissent être définis comme des instances de *nœud*. L'attribut `ion` indique la charge électrique d'un *nœud* en question, et l'attribut `voisin` spécifie les liaisons le reliant à ses voisins (d'autres *nœuds*).

Ainsi `ion` aura pour valeur un nombre, et `voisins` une liste dont chacun des éléments est de type `(noeud liaison)`, où `noeud` est le numéro d'identification d'un *nœud* voisin et `liaison` est un nombre indiquant le type de la liaison correspondante.

Le fait que *nœud* soit défini comme étant une sous-classe de la classe *atome* lui permet d'avoir une identité atomique, grâce à la propriété d'héritage ; *nœud* possède ainsi tous les attributs de *atome*. Les attributs appartenant en propre à *nœud* – `ion` et `voisins` – permettent de situer un *nœud* dans le contexte d'une molécule. Nous avons ainsi notre représentation d'une molécule :

> Une *molécule* est représentée comme un ensemble d'objets *nœud*.

Pour l'exemple de molécule précédent, cet ensemble de *nœuds* est le suivant :

```
1 (noeud-create 'H
     'ion 0
     'voisins '((2 1)))

2 (noeud-create 'N
     'ion 0
     'voisins '((1 1) (3 1) (5 1)))

3 (noeud-create 'H
     'ion 0
     'voisins '((2 1)))

4 (noeud-create 'O
     'ion 0
     'voisins '((5 2)))
```

```
5 (noeud-create 'C
     'ion 0
     'voisins '((4 2) (6 1) (2 1)))

6 (noeud-create 'O
     'ion -1
     'voisins '((5 1)))
```

où la fonction **noeud-create** permet de créer une instance de *nœud*. Plus précisément,

```
(noeud-create 'N
   'ion 0
   'voisins '((1 1) (3 1) (5 1)))
```

permet de créer une instance de *nœud* ayant la liste attributs-valeurs suivante :

```
atome   N
nom     azote
numero  7
masse   14.01
valence (-3 +5)
ion     0
voisins ((1 1) (3 1) (5 1))
```

dont les cinq premiers sont partagés avec l'instance `N` de *atome*.

C'était la représentation d'une *molécule*. Un *état* aura la même représentation, c'est-à-dire, un ensemble de *nœuds*. Toutefois, du fait qu'un *état* peut être une ou plusieurs molécules, il est un graphe qui n'est pas forcément *connexe*, contrairement à une molécule.

Pour pouvoir manipuler une molécule ou un état sous forme déclarative, il faut définir des prédicats d'accès aux champs de *nœud*. Nous choisissons le type de prédicats :

```
(x.attr val)
```

pour cet usage. `x` est un *nœud*, `attr` un champs (**atome**, **masse**, **ion**, **voisins**, etc.), et `val` la valeur correspondante.

Ce type de prédicat est certes suffisant, mais il ne permet toutefois pas d'exprimer facilement le type de situations où deux *nœuds* `x` et `y` sont reliés par une liaison de type `n`. Nous préférons introduire encore un autre type de prédicats :

```
(voisin x y n)
```


pour exprimer le fait que deux *nœuds* `x` et `y` se trouvent reliés par une liaison de type `n`.

On peut maintenant considérer que, dans le modèle de l'espace d'états avec abstraction de concepts, les *concepts de base* sont traduits par les deux types de prédicats :

`(x.attr val)` et `(voisin x y n)`.

5.4 Définition des concepts d'abstraction

5.4.1 Perception d'une molécule par le chimiste

Lorsqu'un chimiste observe une molécule, il en tire des informations non seulement en terme des *concepts de base* :

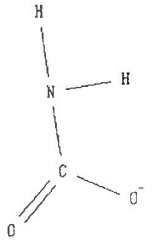
1. les atomes,
2. leur ionicité et
3. les liaisons entre eux,

qui correspondent aux prédicats : `(x.attr val)` et `(voisin x y n)`, mais aussi en terme des *concepts d'abstraction* tels que

- l'acide, l'électro-attracteur, les groupements fonctionnels, les points actifs, les configurations géométriques, etc.

Les concepts d'abstraction sont plus "gros" que les concepts de base en ce sens qu'ils sont bâtis à partir de ceux-ci.

Prenons toujours l'exemple de molécule que l'on a vu précédemment :

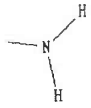


Outre les informations concernant les atomes, l'ionicité et les liaisons, le chimiste perçoit aussi d'autres *informations de haut niveau* :

- Il regarde

comme un tout en lui accordant le nom *carbonyle* ou CO dans sa tête.

- Pour lui, CO est, parmi d'autres, un *électro-attracteur*.
- Il peut aussi regarder



comme un tout : NH_2 (amine).

- Il sait que chacun des deux H attachés à N peut facilement partir et donc qu'ils sont *libérables*.

Cette perception de haut niveau d'une molécule par le chimiste est en fait soutenue par les connaissances concernant la définition de ces termes ou *concepts d'abstraction* : CO, NH_2, *électro-attracteur*, *libérable*, etc. :

- Si un C et un O sont reliés par une liaison double alors on a un CO (carbonyle).
- Le groupement fonctionnel CO est un électro-attracteur.
- Un H attaché à un N est *libérable*.

Ces *concepts d'abstraction* sont très importants en chimie, ils permettent d'exprimer, entre autres, les réactions chimiques de manière facile.

5.4.2 Formalisation de la définition des concepts d'abstraction

Il s'agit de formaliser les relations entre *concepts d'abstraction* et *concepts de base*, ces derniers étant :

`(x.attr val)` et `(voisin x y n)`.

Cela peut se faire aisément en calcul des prédicats. Pour les quelques définitions de la section précédente, on peut obtenir :

- $\forall x \forall y\ (x.atome\ C) \wedge (voisin\ x\ y\ 2) \wedge (y.atome\ O) \rightarrow (CO\ x\ y)$.
- $\forall x \forall y\ (CO\ x\ y) \rightarrow (electroattracteur\ x)$.
- $\forall x \forall y\ (x.atome\ N) \wedge (voisin\ x\ y\ 1) \wedge (y.atome\ H) \rightarrow (liberable\ y\ x)$.

Cette formalisation de la définition des concepts d'abstraction en calcul des prédicats conduit en fait à une sorte de fbf spéciales appelées *clauses de Horn*.

Une *clause* est une disjonction de littéraux ; une clause de Horn est caractérisée par la présence d'un seul littéral positif et a donc la forme suivante :

$$\forall x_1 \forall x_2 \cdots\ P_1 \vee \neg P_2 \vee \neg P_3 \vee \cdots$$

où $P_1, P_2, P_3, \cdots$ sont des prédicats. On peut également utiliser la forme équivalente :

$$\forall x_1 \forall x_2 \cdots\ P_2 \wedge P_3 \wedge \cdots \rightarrow P_1$$

ou encore,

$$\forall x_1 \forall x_2 \cdots\ P_1 \leftarrow P_2 \wedge P_3 \wedge \cdots.$$

Dans l'implantation informatique, nous adoptons la convention suivante :

- Comme toutes les variables sont universellement quantifiées, nous faisons disparaître les quantificateurs et distinguons les variables des constantes en les faisant précéder du symbole "_".
- Nous écrivons "<-" au lieu de "←".
- La fin de clause est marquée par un "&".

Les définitions précédentes ont ainsi la forme suivante dans l'implantation :

```
(CO _x _y) <- (_x.atome C) (voisin _x _y 2) (_y.atome O) &
(e-attracteur _x) <- (CO _x _y) &
(liberable _y _x) <- (_x.atome N) (voisin _x _y 1) (_y.atome H) &
```

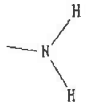
5.4.3 Possibilités ouvertes

L'utilité de la définition des *concepts d'abstraction* réside dans le fait de pouvoir disposer de plus "gros" termes – surtout des termes génériques – que

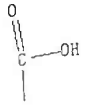
`(x.attr val)` et `(voisin x y n)` dans la description des réactions (opérateurs) ainsi que des méta-règles.

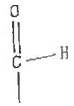
On a vu quelques exemples simples de définitions dans la section précédente. Les connaissances chimiques susceptibles d'être représentées de cette manière relèvent en fait de trois catégories :

1. Les groupements fonctionnels. Ce ne sont pas des termes génériques mais chacun correspond à une sous-structure moléculaire précise. Il s'agit de sous-structures fréquemment rencontrées et qui possèdent une certaine propriété chimique. On a déjà vu



Il y a encore d'autres tels que





2. Les termes génériques. Chacun de ces termes correspond à une variété de sous-structures moléculaires. Par exemple, le terme *électro-attracteur* – employé pour qualifier les sous-structures qui ont tendance à atirer des électrons – peut correspondre à

ainsi que beaucoup d'autres.

La définition des termes génériques se révèle surtout importante. En effet, dans la description d'un opérateur (réaction), on peut par exemple, mettre un prédicat `(e-attracteur)`, sans avoir à détailler du coup ses diverses variétés. La séparation entre une base des opérateurs et une base des clauses de Horn facilite la représentation et la gestion des connaissances.

3. Les modèles de calcul. Certaines informations concernant une molécule – la forme géométrique par exemple – ne peuvent être obtenues que moyennant des calculs suivant certains modèles théoriques, en partant des informations de base (atomes, liaisons) qui sont accessibles par `(x.attr val)` et `(voisin x y n)`. Ces modèles de calcul peuvent aussi être exprimés sous forme de clauses de Horn.

Examinons l'interprétation de ces connaissances sous forme de clauses de Horn. Etant donné qu'un état (une ou plusieurs molécules) est représenté par des concepts de base (`(x.attr val)` et `(voisin x y n)`) et que la précondition d'un opérateur (ou d'une méta-règle) implique à la fois des concepts de base et des concepts d'abstraction, la vérification de cette précondition sur l'état consiste donc à démontrer celle-ci en partant des axiomes que constituent la fbf représentant l'état d'une part et les clauses de Horn définissant les concepts d'abstraction d'autre part. Il faut ainsi un démonstrateur de théorèmes qui effectue la tâche suivante :

$$(\text{état}) \cup (\text{clauses de Horn}) \overset{\textit{démontrer}}{\Longrightarrow} (\text{précondition d'un operateur}).$$

Pour satisfaire au besoin de représenter les trois types de connaissances exposés plus haut et de les interpréter efficacement, notre implantation en LISP du démonstrateur de théorèmes s'est inspiré du langage PROLOG, de sorte que l'on peut véritablement programmer en clauses de Horn.

Ainsi, par rapport aux exemples simples de la section précédente, la syntaxe des clauses, fondée sur LISP, est étendue pour inclure les points suivants :

1. Possibilité d'utiliser les listes. Il n'y a pas de notation spéciale pour les listes. On se sert des parenthèses normales : "(" et ")". Ainsi, `(1 2 3 4)` est une liste à quatre éléments et `()` la liste vide. La notation `(_x . _y)` permet de représenter une liste dont la queue est une variable (_y).

2. Cut de contrôle. Le cut sert à court-circuiter un retour en arrière. Il est représenté par "/".
3. Négation. On peut mettre un `not` devant un prédicat lorsque celui-ci n'est pas une tête de clause. `(not p)` est faux si le prédicat p n'est pas prouvable, sinon, il est vrai.
4. Egalité. `(= _x _y)` permet d'unifier _x et _y. Si _y est ou porte sur une fonction LISP (une expression arithmétique en particulier), l'unification implique l'évaluation de celle-ci. Voir le point suivant.
5. Expressions arithmétiques. On peut utiliser les expressions arithmétiques LISP. Par exemple, le but :

   ```
   (= _y 1) (= _x (+ _y (* 2 3)))
   ```

   permettra d'affecter la valeur 7 à _x.
6. Prédicat `vrai`. Il est toujours vrai.
7. Fonctions LISP comme prédicats. Toute fonction LISP peut être utilisée comme prédicat, la valeur d'évaluation `nil` ou `()` LISP correspondant à la valeur fausse du prédicat. Les fonctions de comparaison arithmétique sont surtout utiles, telles que `(< _x _y)` et `(> _x _y)`.

A titre d'illustration de cette syntaxe, citons quelques exemples de clauses utiles.

Avec les clauses suivantes :

```
(nbH _x _nH) <- (nbHO _x () _nH) &
(nbHO _x _lH _nH) <- (voisin _x _y 1) (_y.atome H) (not (member _y _lH))
                     (nbHO _x (_y . _lH) _nH1) (= _nH (+ 1 _nH1)) / &
(nbHO _x _ 0) &
```

le prédicat `(nbH _x _nH)` permet de retrouver le nombre de H (atomes d'hydrogène) attachés au nœud _x dans une molécule. Le prédicat `(member _y _lH)` étant considéré comme une fonction LISP, on aurait pu aussi utiliser `membre` défini par les clauses suivantes :

```
(membre _x (_x . _)) &
(membre _x (_ . _y)) <- (membre _x _y) &
```

`membre` sera ainsi plus souple que la fonction LISP correspondante.

Un autre exemple consiste à définir l'*halogène* — les atomes appartenant au septième groupe du tableau périodique :

```
(halogene _x) <- (_x.numero _no) (groupe _no _gr) (= _gr 7) &
(groupe _no _gr) <- ... &
```


Le prédicat `(groupe _no _gr)` devrait calculer le numéro du groupe à partir du numéro atomique d'un atome. Bien entendu, tout cela aurait été plus facile si l'on avait mis le numéro du groupe comme un attribut de l'objet *atome*. C'est une question ouverte dont la réponse définitive dépendra de la fréquence d'utilisation de l'attribut groupe.

5.5 Opérateurs

Les opérateurs sont des réactions chimiques. Nous définissons la structure de données suivante pour les représenter :

```
(defstruct operateur
   nom
   reference
   lhs
   rhs
   certitude
)
```

`nom` a pour valeur une chaîne de caractères, `reference` une liste de chaînes de caractères. `nom` est destiné à enregistrer le nom de la réaction. `reference` peut être la bibliographie.

5.5.1 lhs et rhs

`lhs` et `rhs` constituent les deux composantes essentielles d'un opérateur. Chacun est une conjonction de prédicats et définit une sous-structure moléculaire (un *pattern*) qui peut être un graphe connexe ou non. On peut comprendre leur sémantique à travers la description de l'application d'un opérateur. Lorsqu'on applique un opérateur sur un état (une ou plusieurs molécules), on cherche une sous-structure de celui-ci qui corresponde à `lhs` (*pattern matching*). Si l'on réussit, l'opérateur devient applicable. On substitue alors la sous-structure représentée par `rhs` à celle correspondant à `lhs` dans l'état, pour aboutir au nouvel état. On peut encore schématiser cela en prenant la réaction suivante :

$$A + B \longrightarrow C,$$

qui fait intervenir trois structures moléculaires : A, B et C. En la mettant sous forme d'opérateur, `lhs` correspondra alors à $A \cup B$ et `rhs` à C.

En fait, ce n'est qu'une autre forme que la représentation des opérateurs par trois composantes :

1. une précondition,
2. une liste des suppressions et
3. une liste des adjonctions.

Mais le formalisme employé ici présente l'avantage d'être symétrique, permettant d'appliquer un opérateur dans le sens de gauche à droite comme de droite à gauche, ce qui est indispensable pour le présent problème de la synthèse chimique. Dans la description précédente, il suffit d'inverser `lhs` et `rhs` pour comprendre l'application d'un opérateur de droite à gauche.

`lhs` et `rhs` ont exactement la même syntaxe que le corps d'une clause de Horn (cf. la section précédente), c'est-à-dire la partie qui se trouve à droite de "`<-`". Il s'agit donc d'une conjonction de prédicats éventuellement précédés d'un "`not`". Pour avoir une idée précise sur la spécification de `lhs` et de `rhs` pour une réaction chimique, examinons l'interprétation d'un opérateur.

L'application d'un opérateur de gauche à droite sur un état procède ainsi :

1. Prendre `lhs` comme précondition et chercher à le remplir sur l'état.
2. Si `lhs` est rempli (instancié), supprimer dans l'état la structure correspondant à (`lhs0 - rhs0`) et y ajouter celle correspondant à (`rhs0 - lhs0`), où,

   `lhs0` = l'ensemble des prédicats de type (`_x.attr val`) ou (`voisin _x _y _n`) de `lhs`,

   et `rhs0` est la même chose mais correspond à `rhs`.

   L'état devient alors un nouvel état, dû à l'application de l'opérateur.

Pour avoir la description de l'application d'un opérateur de droite à gauche, il suffit d'inverser `lhs` et `rhs`.

Ainsi, `lhs` ou `rhs` est la précondition de l'opérateur suivant qu'il s'agit d'une application de gauche à droite ou de droite à gauche. Par ailleurs, il est à noter qu'une fois la précondition de l'opérateur remplie, les modifications à effectuer sur l'état courant ne tiendront plus compte des prédicats autres que

(`_x.attr val`) et (`voisin _x _y _n`).

En effet, la présence des prédicats correspondant à des concepts d'abstraction dans un opérateur est destinée uniquement à participer à des préconditions à remplir (démontrer).

5.5.2 Notion de VOB

Un point important concernant `lhs` et `rhs` est sur l'instanciation des variables. Introduisons d'abord la notion de VOB :

> *Une VOB est une variable figurant dans la précondition de l'opérateur, portant sur les objets et dont la première apparition se trouve dans un prédicat de type :* (`_x.attr val`) *ou* (`voisin _x _y _n`).

Prenons un exemple, supposons que l'on ait

```
(voisin _x _y 2) (_u.atome N) (liberable _z _u) (_z.atome H)
```

comme précondition d'un opérateur (`lhs` ou `rhs`), alors on a comme VOB : `_x`, `_y` et `_u`. `_z` n'est pas une VOB, car sa première apparition est dans le prédicat (`liberable _z _u`).

La notion de VOB est utilisée ainsi : lorsque l'interprète d'opérateurs cherche à remplir la précondition d'un opérateur, il assure systématiquement l'affectation de valeurs distinctes (instances *nœud* différentes) aux différentes VOB. Nous avons implanté cette manière d'instanciation de VOB par souci d'alléger l'écriture des opérateurs et d'améliorer l'efficacité du système. En effet, dans la description d'un opérateur, les différentes variables-objets demandent toujours des instances objets différentes. Par conséquent, si le mécanisme d'inférences ne s'en occupait pas, il faudrait alors mettre de nombreux (`not (= _x _y)`), ce qui non seulement alourdirait l'écriture mais aussi compromettrait l'efficacité de l'interprétation.

Plus loin, l'exemple de réaction Diels-Alder illustrera l'intérêt du traitement privilégié des VOB par le démonstrateur.

5.5.3 `certitude`

Examinons la partie `certitude` d'un opérateur. Lorsqu'un opérateur est applicable sur un état (`lhs` ou `rhs` comme précondition remplie), sa partie `certitude` permet de calculer une valeur d'évaluation de la facilité de son application, suivant le contexte dans lequel la précondition est satisfaite. Cette valeur est comprise entre 0 et 1, mais elle peut aussi être -1, indiquant que l'opérateur ne peut être appliqué. `certitude` est une liste de type :

```
( (cond1 val1)
  (cond2 val2)
  ...
  (vrai  valn)
)
```

val1, val2, ... sont des valeurs comprises entre 0 et 1, ou la valeur -1. cond1, cond2, ..., ont la même syntaxe que lhs et rhs. Il s'impose de les considérer comme une suite à la précondition de l'opérateur (lhs ou rhs). En effet, lorsque celle-ci est remplie, l'interprète d'opérateurs procède ainsi pour déterminer la valeur de certitude : lorsque cond1 est vérifiée, alors cette valeur est val1, sinon, il cherche à vérifier cond2, et ainsi de suite. La dernière ligne est obligatoire, pour donner une valeur par défaut, lorsqu'aucune des cond1, cond2, etc., n'est satisfaite.

Cette partie certitude est indispensable dans la description des réactions. En effet, lorsqu'un expert décrit une réaction, il donne d'abord un schéma général, qui correspond à nos lhs et rhs ; ensuite, il va apporter des précisions : "en présence d'un ..., la réaction se trouve favorisée, sinon OK mais moins bien", ou, dans certains contextes, elle devient impossible.

Ainsi, lorsqu'on applique une réaction sur une ou des molécules, le contexte d'application permet de calculer la valeur de certitude ou la valeur d'évaluation de la facilité de la réaction.

Les valeurs de certitude des applications successives de réactions seront utilisées pour calculer une valeur d'appréciation des différents chemins au cours de la résolution d'un problème. La façon de calculer cette valeur est à l'étude en consultation avec l'expert.

5.5.4 Exemples

Pour terminer, à titre d'illustration, prenons deux exemples d'opérateurs. Le premier est la réaction Diels-Alder et le second la réaction d'addition [Warren 78].

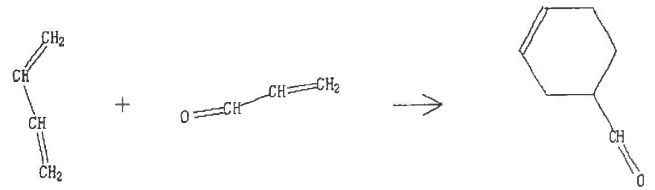
Réaction Diels-Alder

```
nom       "Diels-Alder"
reference ("Designing Organic Synthesis" "S. Warren")
lhs       ( (voisin _2 _1 2) (voisin _1 _6 1) (voisin _6 _5 2)
            (voisin _3 _4 2) )
rhs       ( (voisin _2 _1 1) (voisin _1 _6 2) (voisin _6 _5 1)
            (voisin _5 _4 1) (voisin _4 _3 1) (voisin _3 _2 1) )
certitude ( ( (voisin _4 _7 1) (e-attracteur _7) 0.8 )
            ( (voisin _3 _7 1) (e-attracteur _7) 0.8 )
            ( vrai                               0.5 ) )
```

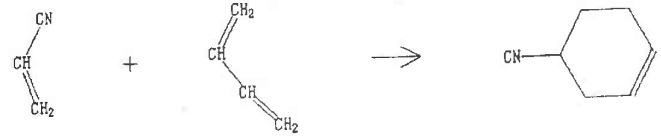

lhs et rhs traduit le schéma général :

La partie certitude apporte des précisions en affirmant que, s'il y a un électro-attracteur relié à l'atome _4 ou _3, la réaction se trouve favorisée (0.8), sinon OK (0.5).

Cet opérateur pourrait par exemple conduire au résultat suivant :



avec la valeur de certitude à 0.8. Un autre résultat également avec certitude à 0.8 serait :



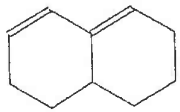
L'application de l'opérateur dans ces deux cas ont fait appel aux clauses suivantes :

```
(e-attracteur _x) <- (CO _x _) &
(e-attracteur _x) <- (CN _x _) &
```

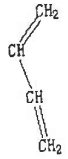
```
(CN _x _y) <- (voisin _x _y 3) (_x.atome C) (_y.atome N) &
(CO _x _y) <- (voisin _x _y 2) (_x.atome C) (_y.atome O) &
```

Sans ces clauses, la valeur de `certitude` aurait été de `0.5`.

Au cours d'une interview, l'expert a fait remarquer que la molécule suivante :



ne peut participer à la réaction Diels-Alder pour jouer le rôle de :



ci-dessus, car la configuration spatiale des deux doubles liaisons empêche une telle action. Appelons cette configuration `estsett`. Pour tenir compte de cette remarque, il faut compléter la partie `certitude` de l'opérateur de la manière suivante :

```
certitude ( ( (estsett _5 _6 _1 _2)              -1 )
            ( (voisin _4 _7 1) (e-attracteur _7) 0.8 )
            ( (voisin _3 _7 1) (e-attracteur _7) 0.8 )
            ( vrai                               0.5 ) )
```

Et l'on va définir la configuration `estsett` sous forme de clauses. Bien entendu, si la notion de cette configuration ne trouvait son application qu'ici, il suffirait de mettre sa définition directement dans la partie `certitude` à la place du prédicat `(estsett _5 _6 _1 _2)`.

Il est à noter que toutes les variables dans `lhs` de cet opérateur sont des VOB et qu'elles sont ainsi affectées de valeurs distinctes au moment du filtrage par le démonstrateur de théorèmes. Si le démonstrateur n'assurait pas cette fonction, il faudrait alors écrire, pour `lhs` par exemple,

```
lhs       ( (voisin _2 _1 2) (voisin _1 _6 1) (voisin _6 _5 2)
```



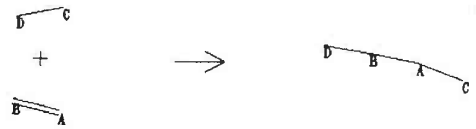
```
(voisin _3 _4 2) (not (= _3 _2)) (not (= _3 _1))
(not (= _3 _6)) (not (= _3 _5)) (not (= _4 _2))
(not (= _4 _1)) (not (= _4 _6)) (not (= _4 _5)) )
```

On voit ainsi l'utilité du traitement privilégié des VOB par le démonstrateur de théorèmes.

Réaction d'addition

```
nom       "Addition"
reference ("Designing Organic Synthesis" "S. Warren")
lhs       ( (voisin _A _B 2) (liberable _C _D) (voisin _C _D 1) )
rhs       ( (voisin _A _B 1) (voisin _A _C 1) (voisin _B _D 1) )
certitude ( ( vrai 0.7 ) )
```

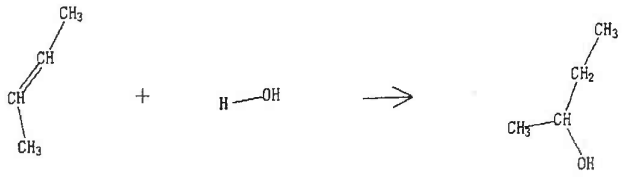
Pour que cet opérateur soit correct aussi de droite à gauche, il faudrait ajouter d'autres prédicats dans `rhs`. Nous ne citons ici comme exemple que son application de gauche à droite. Cet opérateur correspond au schéma suivant :



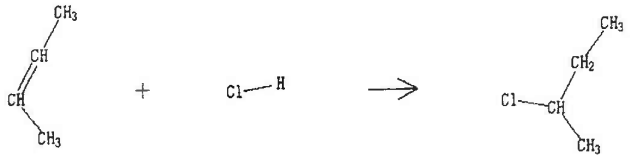
En présence des clauses suivantes :

```
(liberable _x _y) <- (_y.atome O) (voisin _y _x 1) (_x.atome H) &
(liberable _x _y) <- (_y.atome Cl) (voisin _y _x 1) (_x.atome H) &
```

cet opérateur pourrait donner lieu aux situations suivantes :



et,



5.6 Méta-règles

Les méta-règles représentent les stratégies de synthèse. Dans l'état actuel du système OASIS, nous n'avons pas encore exploré ce méta-niveau de connaissances. Néanmoins, nous pouvons affirmer qu'une méta-règle aura la forme suivante :

```
SI condition ALORS types-de-regles-a-essayer
```

La partie `condition` aura la même syntaxe que **lhs** ou **rhs** d'un opérateur. Lorsqu'elle est satisfaite sur un état, l'application de l'opérateur donne comme résultat les types de règles à tenter sur celui-ci.

L'emploi des méta-règles permet de restreindre les opérateurs à utiliser pour développer un état dans l'espace d'états au cours de la résolution d'un problème, et donc de réduire le risque combinatoire (cf. le chapitre 7).

5.7 Conclusion

En calquant le problème de la synthèse chimique sur le modèle de l'espace d'états avec abstraction de concepts, nous avons réparti les connaissances chimiques de base sur trois niveaux dans le système OASIS.

Le premier niveau concerne la représentation des molécules ou des états en terme de concepts de base. Ainsi, une molécule est définie comme un ensemble d'objets appartenant à la classe *nœud*, alors que la classe *nœud* est une sous-classe de la classe d'objets *atome*. On peut considérer que les concepts de base sont traduits par les deux types de prédicats

`(_x.attr val)` et `(voisin _x _y _n)`.

Le deuxième niveau de connaissances est la définition de certains concepts plus élaborés ou concepts d'abstraction sous forme de clauses de Horn.

Le troisième niveau de connaissances concerne les réactions chimiques ou les opérateurs, dont la description s'appuie à la fois sur les concepts de base et sur les concepts d'abstraction.

A ces trois niveaux de connaissances, s'ajouteront les méta-règles qui traduiront les stratégies de synthèse.

D'un point de vue global, notre approche à la représentation de connaissances est une approche mixte, faisant intervenir les objets structurés, le formalisme logique et les règles de production.

6
Mécanismes d'inférences de base

6.1 Introduction

Les problèmes en synthèse chimique consistent en une exploration de l'espace d'états. Cette exploration implique deux niveaux de travail :

1. le niveau des transitions d'états assurant le passage d'un état à l'autre,
2. le niveau du contrôle qui décide de l'orientation des transitions d'états.

Les mécanismes d'inférences de base sont destinés à accomplir le premier niveau de travail et font l'objet d'études du présent chapitre. Il s'agit donc surtout d'interpréter, sur un état donné, les opérateurs de façon isolée, en vue de l'obtention de nouveaux états. Les mécanismes d'inférence de base sont ainsi un interprète d'opérateurs plus quelques outils généraux. On peut schématiser la tâche de l'interpète d'opérateurs de la façon suivante :

$$\textit{état} * \textit{opérateur} \rightarrow \textit{état}.$$

Le modèle de l'espace d'états avec concepts d'abstraction a permis de décrire le domaine du problème de la synthèse chimique en trois parties :

1. Les états (molécules). Un état est représenté comme un ensemble d'objets *nœud*. Mais d'un point de vue logique, on peut le considérer comme étant une fbf ou plus exactement une conjonction de prédicats de type : (`_x.attr val`) et (`voisin _x _y _n`). C'est d'ailleurs par le biais de ce genre de prédicats que l'on accède à un état.
2. Les clauses de Horn. Les clauses de Horn définissent certains prédicats concernant les états qui sont plus élaborés que (`_x.attr val`) et (`voisin _x _y _n`), dans le but de permettre de formuler certaines informations de haut niveau directement en "gros termes", et surtout en termes génériques.
3. Les opérateurs de transition (réactions). La description d'un opérateur a deux composantes principales : une partie gauche (`lhs`) et une partie droite (`rhs`). Au moment d'appliquer un opérateur de gauche à droite sur un état, `lhs` est la précondition de

l'opérateur ; dans le sens inverse, c'est `rhs` qui est la précondition. `lhs` et `rhs` sont tous deux une conjonction de prédicats (précédés d'un `not` éventuellement). Il peut s'agir aussi bien de prédicats de type (`_x.attr val`) et (`voisin _x _y _n`) que de prédicats élaborés définis par des clauses de Horn.

En essayant d'appliquer un opérateur sur un état, l'interprète d'opérateurs doit d'abord s'assurer que la précondition de l'opérateur (`lhs` ou `rhs`) est remplie par l'état. Du fait que cette précondition fait intervenir deux niveaux de prédicats – (`_x.attr val`) et (`voisin _x _y _n`) d'une part et les prédicats élaborés d'autre part – cela revient donc à démontrer qu'elle est une conséquence logique de la fbf correspondant à l'état (sous forme d'un ensemble d'objets) modulo les clauses de Horn. On a ainsi besoin d'un démonstrateur de théorèmes se fondant sur les objets et les clauses de Horn.

Ce démonstrateur de théorèmes est un outil général et permet d'accomplir tout ce qui est d'ordre déductions sur un état. A part la satisfaction de la précondition d'un opérateur, il sert également à remplir la prémisse d'une méta-règle, et éventuellement aussi à évaluer un état. Au chapitre 8, on verra encore son utilisation dans l'affichage des molécules dans le but de regrouper les atomes qui sont à afficher comme un tout.

Pour rendre le processus de démonstration de théorèmes plus efficace, nous introduisons encore un autre outil : un évaluateur partiel. En effet, lorsqu'on a plusieurs théorèmes à prouver sur un même état, il devient alors peu judicieux que le démonstrateur utilise toujours les clauses générales, car cela peut impliquer beaucoup d'interprétations répétitives.

L'évaluateur partiel est destiné à spécialiser les clauses générales par rapport à un état, pour en obtenir une version spécifique à celui-ci, permettant une meilleure efficacité.

L'*interprète d'opérateurs*, le *démonstrateur de théorèmes* et l'*évaluateur partiel* constituent ainsi les mécanismes d'inférences de base du système OASIS (cf. l'architecture en figure 4.4). Nous allons par la suite étudier leur réalisation.

6.2 Démonstrateur de théorèmes

6.2.1 Démonstration de théorèmes en logique du premier ordre

Avant d'entrer dans les détails concernant la réalisation du *démonstrateur*, nous allons faire, de façon informelle, un petit rappel de la théorie de la logique du premier ordre.

Formalisation de problèmes en calcul des prédicats

Dans l'introduction de ce mémoire, nous avions souligné l'importance de la logique et de la démonstration de théorèmes en intelligence artificielle. En effet, un résolveur de problèmes en intelligence artificielle est en fait l'hybridation d'un aspect heuristique spécifique à certains problèmes et d'un aspect logique général.

La logique formelle offre un formalisme uniforme pour la description de problèmes et cherche à les résoudre de façon universelle par un processus de démonstration de théorèmes.

Le calcul des prédicats du premier ordre est plus réduit que d'autres formes logiques d'ordre supérieur, mais il est relativement développé aujourd'hui. La formalisation d'un problème en calcul des prédicats s'appuie sur les notions de constante, variable, fonction, prédicat, connecteur logique, et quantificateurs existentiel et universel.

Prenons un exemple pour illustrer la formalisation de problèmes en calcul des prédicats. On a les faits suivants :

1. Confucius est un homme.
2. Tout homme est mortel.

Et l'on pose la question : Confucius est-il mortel ?

On peut définir le prédicat $homme(x)$ pour représenter "x est un homme" et $mortel(x)$ "x est mortel". Le quantificateur universel $\forall x$ désigne "pour tout x". Les faits ci-dessus ont alors la formalisation suivante :

1. $homme(Confucius)$.
2. $(\forall x)\,(homme(x) \rightarrow mortel(x))$.

La première formule qui est composée d'un seul prédicat est une formule atomique, et $Confucius$ y est une constante, à la différence de x dans la deuxième formule qui est une variable.

La question consiste à démontrer que

$$mortel(Confucius)$$

est une conséquence logique des formules 1 et 2 ci-dessus, ou que

$$homme(Confucius) \wedge (\forall x\,(homme(x) \rightarrow mortel(x))) \rightarrow mortel(Confucius)$$

est un théorème, ce qui revient à démontrer que la formule ci-dessus est valide (vraie sous toute *interprétation*).

Démonstration de théorèmes

Pour démontrer qu'une formule est valide, l'équivlent est de démontrer que sa négation est contradictoire – c'est-à-dire que celle-ci soit fausse sous toute interprétation – ce que l'on appelle une démonstration par l'absurde.

On utilise la forme canonique d'une formule, qui a la forme suivante :

$$\forall x \forall y \cdots ((P_{11} \vee P_{12} \vee \cdots) \wedge (P_{21} \vee P_{22} \vee \cdots) \cdots)$$

où chacun des P_{ij} désigne un littéral (un prédicat ou la négation d'un prédicat).

Toute formule bien formée (fbf) peut se ramener à une forme canonique (à rappeler, entre autres, que les quantificateurs existentiels peuvent être éliminés par *skolémisation*). Du fait qu'il ne subsiste que des quantificateurs universels, on peut les omettre.

La forme canonique étant une conjonction de clauses, on peut donc considérer un ensemble de clauses comme étant son synonyme. Ainsi, dire qu'une formule est contradictoire, c'est dire que son ensemble de clauses est contradictoire.

Herbrand (voir dans [Chang 73]) a obtenu le théorème suivant :

> Un ensemble de clauses est contradictoire si et seulement s'il existe un ensemble fini d'instances fermées (sans variable) de ces clauses qui est contradictoire.

Robinson [Robinson 65] a rendu possible l'application pratique du théorème d'Herbrand en tant qu'une procédure de démonstration de théorèmes, en proposant une approche syntaxique. Cette approche permet d'éviter d'engendrer, en vue de trouver la contradiction, des instances de clauses de façon arbitraire. En effet, elle oriente la génération d'instances de clauses par le *principe de résolution* :

> Pour deux clauses C_1 et C_2, si les littéraux L_1 appartenant à C_1 et L_2 à C_2 sont complémentaires après unification, alors effectuer des substitutions de variables correspondant à cette unification sur les deux clauses, et faire la disjonction des deux clauses en y enlevant L_1 et L_2. La clause qui en résulte s'appelle *clause résolvante* de C_1 et C_2.
>
> Un ensemble de clauses est contradictoire si et seulement si l'on peut en déduire la clause vide $\square$ en utilisant le principe de résolution.

Prenons l'exemple précédent qui consiste à démontrer la validité de la formule :

$$homme(Confucius) \wedge (\forall x\,(homme(x) \rightarrow mortel(x))) \rightarrow mortel(Confucius).$$

On va donc démontrer que sa négation soit contradictoire. La forme canonique de celle-ci est l'ensemble de clauses suivant :

1. $homme(Confucius)$
2. $\neg homme(x) \vee mortel(x)$
3. $\neg mortel(Confucius)$.

Les clauses 1 et 2 donnent la clause résolvante :

$mortel(Confucius)$.

La résolution entre celle-ci et la clause 3 donne la clause vide $\square$.

On a ainsi démontré par l'absurde que,

$mortel(Confucius)$

est une conséquence logique des faits initiaux du problème.

Clauses de Horn et PROLOG

A la suite des travaux de Robinson, on a beaucoup travaillé sur les stratégies de résolution, le problème étant de savoir, étant donné un ensemble de clauses, comment effectuer le moins de résolution possibles pour arriver à la clause vide.

La SLD-résolution proposée par Kowalski [Kowalski 74] est une stratégie spécialement adaptée aux clauses de Horn. La syntaxe des clauses de Horn plus la SLD-résolution constitue le fondement du langage PROLOG (voir par exemple [Marchand 86] ; pour le détail d'une implantation PROLOG, voir aussi [Van Caneghem 86]).

Les clauses de Horn sont une forme restrictive de clauses caractérisée par l'existence d'un seul littéral positif, du genre :

$$P_1 \vee \neg P_2 \vee \neg P_3 \cdots \neg P_n$$

ou encore noté,

$$P_1 \leftarrow P_2, P_3, \cdots, P_n.$$

Etant donné un but :

$$\exists x \exists y \cdots B_1 \wedge B_2 \wedge \cdots B_n$$

à démontrer, la SLD-résolution part de sa négation :

$$\forall x \forall y \cdots \neg B_1 \vee \neg B_2 \vee \cdots \neg B_n$$

ou noté simplement,

$\neg B_1, \neg B_2, \cdots, \neg B_n$

et effectue la résolution de façon linéaire, en procédant de but en nouveau-but, cherchant constamment à "effacer" le but courant, jusqu'à l'obtention éventuelle de la clause vide, comme le montre la figure 6.1.

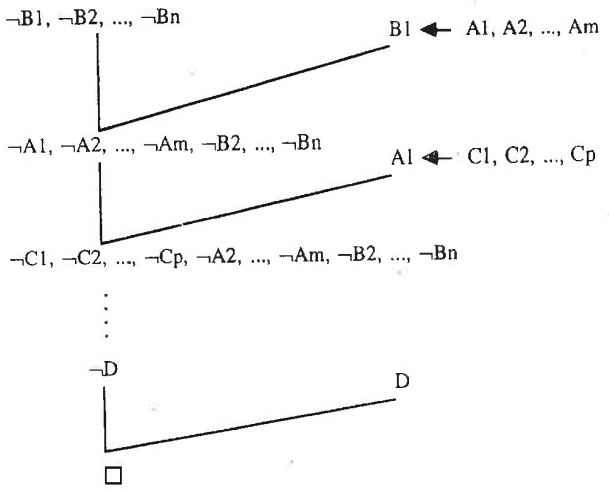


Figure 6.1. SLD-résolution

Si au cours de la résolution, on ne trouve aucune clause pour faire la résolution avec le premier littéral du but courant (pour l'effacer) – supposons que ce soit $\neg A_1$ sur la figure 6.1 par exemple – alors il faut remettre en cause la résolution immédiatement précédente, en l'occurrence, celle sur $\neg B_1$ et B_1. Dès lors, il faut chercher une autre clause pour faire la résolution avec $\neg B_1$ dans le but. Ce processus s'appelle *backtracking* ou *retour en arrière*.

C'est ainsi que fonctionne un interprète PROLOG. On voit que l'ordre des prédicats à l'intérieur de chaque clause est très important. Par ailleurs, l'ordre d'écriture des clauses ayant la même tête est tout aussi important. En effet, pour faire la résolution avec un littéral dans un but, l'interprète PROLOG choisit la première clause dans l'ordre d'écriture pour cet usage lorsqu'il y a plusieurs candidats.

6.2.2 Démonstrateur dans OASIS

Dans le système OASIS, on utilise le démonstrateur de théorèmes pour :

1. la vérification de la précondition d'un opérateur (`lhs` ou `rhs`),
2. l'évaluation de la `certitude` d'un opérateur,
3. la vérification de la prémisse d'une méta-règle,
4. l'affichage des molécules dans le cadre de l'interface,
5. éventuellement l'évaluation des états.

Caractéristiques

Le démonstrateur prend pour axiome :

1. un ensemble d'objets structurés (*nœud*) traduisant en fait une conjonction de prédicats instanciés (*prédicats fermés*),
2. un ensemble de clauses de Horn en logique des prédicats.

On peut schématiser sa fonction ainsi :

$$objets \cup clauses \overset{démonstration}{\Longrightarrow} conclusion.$$

La gestion des objets constitue une particularité du présent démonstrateur.

Les prédicats d'accès aux objets sont de type :

`(_x.attr val)` et `(voisin _x _y _n)`.

Le deuxième type est en fait redondant, pouvant être ainsi défini :

```
(voisin _x _y _n) <- (_x.voisins _voisins) (membre (_y _n) _voisins) &
```

(cf. la définition de la classe d'objets *nœud* au chapitre précédent). Toutefois, notre choix est de lui donner une interprétation directe, comme pour `(_x.attr val)`, à l'intérieur du démonstrateur pour avoir une meilleure efficacité.

Il est à noter qu'il ne s'agit en aucun cas de transformer systématiquement les objets en clauses atomiques (des faits) mais qu'il faut les traiter directement, afin que le démonstrateur ait une bonne performance.

Une autre particularité du démonstrateur est liée à cette gestion d'objets. Comme on l'a vu au chapitre précédent, il faut, étant donné un but à démontrer, assurer l'affectation de valeurs (instances *nœud*) distinctes aux VOB, c'est-à-dire aux variables-objets figurant dans le but et dont la première apparition est dans un prédicat de type : `(_x.attr val)` ou `(voisin _x _y _n)`. Il s'agit là également d'une gestion directe. Il n'est pas question d'insérer implicitement dans le but des prédicats du genre :

```
(not (= _x _y)),
```

l'efficacité oblige.

Algorithme 1 : cas sans objet

Considérons d'abord le cas sans objets. On est ainsi dans la situation où l'on a un but à prouver et l'on dispose uniquement d'un ensemble de clauses :

$$clauses \overset{démonstration}{\Longrightarrow} but.$$

Comme on l'a vu précédemment (cf figure 6.1), la SLD-résolution procède par effacement de buts. Supposons que l'on ait le but initial $\mathcal{B}$:

$$B_1, B_2, \cdots, B_n$$

à démontrer, on va alors chercher une clause dont la tête soit unifiable avec B_1, en vue d'effacer celui-ci. Supposons qu'on ait trouvé la clause suivante :

$$B_1 \leftarrow A_1, A_2, \cdots$$

alors en effaçant B_1, on aboutit au nouveau but suivant :

$$A_1, A_2, \cdots, B_2, \cdots, B_n.$$

On repart alors de ce nouveau but et procède de la même manière que pour le but initial $\mathcal{B}$, et ainsi de suite.

Si au cours de ce processus un prédicat ne peut être effacé, il faut alors remettre en cause l'effacement immédiatement précédent. Prenons un exemple : si A_1 dans le nouveau but ci-dessus ne peut être effacé, on va alors reconsidérer l'effacement de B_1 dans le but précédent. Il faudra trouver une autre clause que celle qui avait été utilisée pour effacer B_1.

Notre algorithme débute par la fonction `prouver(B, C)` pour démontrer un but `B` (une pile de prédicats) en prenant pour axiome un ensemble de clauses `C`. Le type de `prouver` est l'union de `VAR-VAL` et `BOOL`, de telle sorte qu'elle renvoie une liste d'associations variable-valeur en cas de succès de la démonstration, et `FAUX` en cas d'échec.

```
prouver(B, C) -> (VAR-VAL | BOOL)
  clauses := c-candidats(tete(B), C)
  var-val := VIDE
  c-effacer(B, clauses, var-val, C)
```


La fonction `c-candidats(pred, C)` permet de retrouver, parmi `C`, la liste des clauses dont la tête est le même que le prédicat `pred` et qui sont donc susceptibles de l'effacer ; elle implique le renommage des variables. Le processus d'effacement du but proprement-dit est effectué de façon récursive par la fonction `c-effacer(but, clauses, var-val, C)` qui cherche à effacer le but `but` sous la contrainte de la liste d'associations variable-valeur `var-val` et de la ressource `clauses` qui est la liste des clauses encore disponibles pour effacer le premier prédicat de `but`.

```
c-effacer(B, clauses, var-val, C) -> (VAR-VAL | BOOL)
  si B = VIDE alors var-val
  sinon si clauses = VIDE alors FAUX
  sinon
     clause := tete(clauses)
     var-val0 := unifier(var-val, tete(B), tete(clause))
     si var-val0 <> FAUX alors
        nouveau-but := concat(queue(clause), queue(B))
        clauses0 := c-candidats(tete(nouveau-but), C)
        var-val1 := c-effacer(nouveau-but, clauses0, var-val0, C)
        si var-val1 <> FAUX alors var-val1
        sinon c-effacer(B, queue(clauses), var-val, C)
     sinon c-effacer(B, queue(clauses), var-val, C)
```

Le cas où `var-val0` serait `FAUX` indiquerait que le premier prédicat de `B` ne peut être effacé par la première clause de `clauses`, qui est donc une fausse candidate. Mais il ne s'agit là pas d'un retour en arrière.

Au contraire, lorsque `var-val1` est `FAUX`, ce sera un retour en arrière, remettant en cause l'utilisation de la première clause de `clauses` pour effacer le premier prédicat de `B`.

La fonction `unifier(var-val, terme1, terme2)` permet d'unifier deux termes `terme1` et `terme2` dans le contexte d'une liste d'associations variable-valeur `var-val`. Elle renvoie une nouvelle liste de telles associations en cas de succès, et `FAUX` en cas d'échec.

```
unifier(var-val, terme1, terme2) -> (VAR-VAL | BOOL)
  cas :
    1. terme1 = VARIABLE :
       unifier(var-val, valeur(terme1), terme2)
    2. terme1 = LISTE :
       si terme2 = VARIABLE alors
          unifier(var-val, terme1, valeur(terme2))
       sinon si terme2 = LISTE alors
```

```
        var-val0 := unifier(var-val, tete(terme1), tete(terme2))
        si var-val0 alors
           unifier(var-val0, queue(terme1), queue(terme2))
        sinon FAUX
      sinon FAUX
  3. terme2 = VARIABLE :
     unifier(var-val, valeur(terme1), terme2)
  4. terme1 = terme2 : var-val
  5. sinon : FAUX
fin-cas
```

Algorithme 2 : cas avec gestion d'objets

Maintenant, on a un but à démontrer et l'on dispose comme axiome conjointement d'un ensemble de clauses et d'un ensemble d'objets ; le démonstrateur doit effectuer la tâche suivante :

$$objets \cup clauses \overset{démonstration}{\Longrightarrow} but.$$

Les clauses et le but peuvent faire intervenir cette fois des prédicats d'accès aux objets de type :

(_x.attr val) et (voisin _x _y _n).

Logiquement, un objet représente un certain nombre de clauses unitaires (qui n'ont qu'une tête sans corps) qui sont chacune un prédicat de type ci-dessus. L'important est que le démonstrateur doive les traiter directement sans passer par leur transformation systématique en clauses (unitaires).

Ainsi, étant donné un but à démontrer (effacer), on se fonde sur la même stratégie générale que dans le cas sans objet. Mais il faut distinguer trois façons d'effacement de prédicats (`c-effacer`, `o-effacer` et `v-effacer` dans l'algorithme) suivant qu'il s'agit d'un prédicat ordinaire, d'un (_x.attr val) ou d'un (voisin _x _y _n). L'algorithme débute comme suit avec la fonction `prouver(B, C, O)`, pour démontrer le but `B` en prenant pour axiome la liste de clauses `C` et la liste d'objets `O` :

```
prouver(B, C, O) -> (VAR-VAL | BOOL)
  var-val := VIDE
  prouver0(B, var-val, C, O)

---------------------
```



```
prouver0(B, var-val, C, O) -> (VAR-VAL | BOOL)
  si B = VIDE alors var-val
  sinon
    type := type-pred(tete(B))
    cas
      1. type = 'ordinaire ' :
         clauses := c-candidats(tete(B), C)
         c-effacer(B, clauses, var-val, C, O)
      2. type = 'obj-attr' :
         objets := o-candidats(var-val, tete(B), O)
         o-effacer(B, objets, var-val, C, O)
      3. type = 'voisin' :
         voisins := v-candidats(var-val, tete(B), O)
         v-effacer(B, voisins, var-val, C, O)
    fin-cas
```

Ainsi, suivant le type du premier prédicat dans le but `B`, `prouver` soumet le problème différemment.

Examinons les trois cas d'effacement en détails suivant le type du premier prédicat du but `B` :

1. Prédicat ordinaire. C'est le cas traité par l'algorithme pour le cas sans objet. On a l'algorithme suivant dans le présent contexte :

```
c-effacer(B, clauses, var-val, C, O) -> (VAR-VAL | BOOL)
  si clauses = VIDE alors FAUX
  sinon
     clause := tete(clauses)
     var-val0 := unifier(var-val, tete(B), tete(clause))
     si var-val0 <> FAUX alors
        nouveau-but := concat(queue(clause), queue(B))
        var-val1 := prouver0(nouveau-but, var-val0, C, O)
        si var-val1 <> FAUX alors var-val1
        sinon c-effacer(B, queue(clauses), var-val, C, O)
     sinon
        c-effacer(B, queue(clauses), var-val, C, O)
```

2. (_x.attr val). Les candidats pour effacer un tel prédicat ne sont plus des clauses, mais une liste d'objets dont la sélection est assurée par la fonction `o-candidats (var-val, predicat, objets)` parmi `objets`. Celle-ci distingue deux cas :

(a) _x instanciée. o-candidats renvoie la liste composée d'un seul objet correspondant à la valeur de _x.

(b) _x non-instanciée. o-candidats renvoie la liste composée de l'ensemble des objets disponibles : objets.

On a l'algorithme suivant de l'effacement :

```
o-effacer(B, objets, var-val, C, 0) -> (VAR-VAL | BOOL)
  si objets = VIDE alors FAUX
  sinon
     objet := tete(objets)
     var-val0 := o-unifier(var-val, tete(B), objet)
     si var-val0 <> FAUX alors
        nouveau-but := queue(B)
        var-val1 := prouver0(nouveau-but, var-val0, C, 0)
        si var-val1 <> FAUX alors var-val1
        sinon o-effacer(B, queue(objets), var-val, C, 0)
     sinon
        o-effacer(B, queue(objets), var-val, C, 0)
```

A noter que l'effacement d'un prédicat du présent type consiste à le réduire à néant et non le remplacer par d'autres, comme dans le cas de l'effacement d'un prédicat ordinaire par des clauses.

La fonction o-unifier(var-val, predicat, obj) permet de faire le lien entre un prédicat de type (_x.attr _y) et un objet obj. Cela revient en fait à unifier les deux listes (_x _y) et (obj val) où val désigne la valeur de l'attribut attr de obj.

3. (voisin _x _y _n). Les candidats pour effacer un tel prédicat sont une liste d'éléments du genre (voisin obj1 obj2 n) tel que (obj2 n) figure dans l'attribut voisins de l'objet obj1 (et vice versa, cf. la définition de *nœud* au chapitre précédent). La génération de cette liste est assurée par la fonction v-candidats(var-val, predicat, objets) qui distingue trois cas :

   (a) _x et _y instanciées. On a alors un seul candidat possible qui est (voisin val1 val2 n) où val1 et val2 sont respectivement la valeur de _x et celle de _y, et n est la valeur correspondante (type de liaison entre les deux *nœuds* val1 et val2).

   (b) L'une de _x et _y instanciée. Supposons que ce soit _x, qui a pour valeur obj0. v-candidats renvoie alors comme résultat la liste de tous les (voisin val1 objn n) tels que val1 est la valeur de _x et que (objn n) figure dans l'attribut *voisins* du *nœud* val1.

   (c) Aucune de _x et _y instanciée. Les candidats correspondent alors à la concaténation de l'attribut *voisins* de tous les objets (*nœud*) disponibles.

Voici l'algorithme de l'effacement :

```
v-effacer(B, voisins, var-val, C, 0) -> (VAR-VAL | BOOL)
  si voisins = VIDE alors FAUX
  sinon
     voisin := tete(voisins)
     var-val0 := unifier(var-val, tete(B), voisin)
     si var-val0 <> FAUX alors
        nouveau-but := queue(B)
        var-val1 := prouver0(nouveau-but, var-val0, C, 0)
        si var-val1 <> FAUX alors var-val1
        sinon v-effacer(B, queue(voisins), var-val, C, 0)
     sinon
        v-effacer(B, queue(voisins), var-val, C, 0)
```

Algorithme 3 : prédicats spéciaux

Le démonstrateur devra encore traiter quelques prédicats spéciaux.

Examinons d'abord le cut (/). "/" est toujours vrai. Son utilisation permet de court-circuiter un retour en arrière. Par exemple, si, lors de l'effacement du but :

$$P_1, P_2, \cdots, P_n$$

on a utilisé la clause :

$$P_1 \leftarrow A_1, /, A_2, A_3$$

pour aboutir au nouveau but :

$$A_1, /, A_2, A_3, P_2, \cdots, P_3$$

alors l'échec éventuel de l'effacement ultérieur des prédicats qui se trouvent après "/" ne va pas remettre en cause l'effacement de A_1, mais entrainera carrément un échec définitif du prédicat P_1 dans le but d'avant, de sorte qu'aucune autre clause ne sera essayée pour y effacer P_1.

Ainsi, lorsqu'un échec traverse un cut au cours de sa rétro-propagation, il va plus vite en "brûlant les étapes", jusqu'au point de décision auquel le cut était entré dans le but.

Pour implanter ce mécanisme, dans la pile traduisant un but, un cut doit pouvoir être identifié conjointement avec sa clause d'origine, pour ne pas être confondu avec les autres cuts. Ainsi, dans l'implantation, à chaque fois qu'un cut doit entrer dans le but, nous le remplaçons par un nombre entier unique.

Parallèlement, la valeur renvoyée par les fonctions d'effacement de buts, peut être :

1. une valeur de type VAR-VAL en cas de succès,
2. FAUX en cas d'un échec normal,
3. un entier correspondant au cut que l'échec a traversé en premier au cours de la rétro-propagation.

Ces fonctions seront ainsi définies comme étant de type (VAR-VAL | BOOL | ENTIER), c'est-à-dire union de VAR-VAL, BOOL et ENTIER.

Avant de donner l'algorithme, examinons les autres types de prédicats prédéfinis :

- "not". (not P) est vrai lorsque le prédicat P ne peut être démontré. Ainsi, à la rencontre d'un (not P), on va prendre P comme un but et chercher à l'effacer. En cas de succès, (not P) devient faux.
- "=". (= _x _y) consiste à unifier _x et _y. Lorsque _x et/ou _y portent sur une expression évaluable au sens LISP, elle sera évaluée.
- Le prédicat vrai. Il est interprété comme étant toujours vrai.
- Fonctions LISP. Si un prédicat est une fonction LISP, elle sera évaluée. La valeur d'évaluation nil correspond à FAUX du prédicat.

A part le problème du type, il suffit de modifier deux fonctions dans l'algorithme de la section précédente : prouver0, c-effacer, et d'ajouter la fonction p-effacer, pour rendre compte des prédicats prédéfinis. Voici la nouvelle version de prouver0 :

```
prouver0(B, var-val, C, 0) -> (VAR-VAL | BOOL | ENTIER)
  si B = VIDE alors var-val
  sinon
    type := type-pred(tete(B))
    cas
      1. type = 'ordinaire ' :
         clauses := c-candidats(tete(B), C)
         c-effacer(B, clauses, var-val, C, 0)
      2. type = 'obj-attr' :
         objets := o-candidats(var-val, tete(B), 0)
         o-effacer(B, objets, var-val, C, 0)
      3. type = 'voisin' :
         voisins := v-candidats(var-val, tete(B), 0)
         v-effacer(B, voisins, var-val, C, 0)
      4. type = 'predefini' :
```



```
         p-effacer(B, var-val, C, 0)
    fin-cas
```

Cette nouvelle version a l'intégration de p-effacer de plus par rapport à la version précédente. p-effacer est défini comme suit :

```
p-effacer(B, var-val, C, 0) -> (VAR-VAL | BOOL | ENTIER)
  P := tete(B)
  type := type-pred(P)
  cas
    1. type = ENTIER :
       var-val0 := prouver0(queue(B), var-val, C, 0)
       si var-val0 = FAUX alors P
       sinon var-val0
    2. type = '=' :
       si e-unifier(P) alors prouver0(queue(B), var-val, C, 0)
       sinon FAUX
    3. type = 'vrai' :
       prouver0(queue(B), var-val, C, 0)
    4. type = FONCTON-LISP :
       si evaluer(P) = 'nil' alors FAUX
       sinon prouver0(queue(B), var-val, C, 0)
    5. type = 'not' :
       si prouver0(queue(P), var-val, C, 0) <> FAUX alors FAUX
       sinon prouver0(queue(B), var-val, C, 0)
  fin-cas
```

Dans le deuxième cas : type = '=', la fonction e-unifier permet d'unifier deux termes, éventuellement moyennant une évaluation. Le premier cas : type = ENTIER indique qu'il s'agit d'un cut. Alors, si l'effacement du reste de B est FAUX, prouver0 renvoie l'entier du présent cut, pour signaler qu'un retour en arrière a traversé par le cut.

Ce signal sera traité par la fonction c-effacer dans sa nouvelle version :

```
c-effacer(B, clauses, var-val, C, 0) -> (VAR-VAL | BOOL | ENTIER)
  si clauses = VIDE alors FAUX
  sinon
     clause := tete(clauses)
     var-val0 := unifier(var-val, tete(B), tete(clause))
     si var-val0 alors
```

```
      clause0 := remplacer-cut-par-entier(clause)
      nouveau-but := concat(queue(clause0), queue(B))
      cut := numero-cut(clause0)
      var-val1 := prouver0(nouveau-but, var-val0, C, 0)
      si var-val1 = cut alors FAUX
      sinon si var-val1 <> FAUX alors var-val1
      sinon c-effacer(B, queue(clauses), var-val, C, 0)
    sinon
      c-effacer(B, queue(clauses), var-val, C, 0)
```

Algorithme final : affectation de valeurs différentes aux VOB différentes

On exige du démonstrateur qu'il assure l'affectation de valeurs différentes aux différentes VOB — variables-objets figurant dans le but initial et dont la première apparition se trouve dans un prédicat de type (_x.attr val) ou (voisin _x _y _n).

Ce serait inefficace si l'on procédait, à chaque fois qu'une VOB se trouve affectée d'une valeur, à une vérification pour voir si celle-ci n'est pas la même que la valeur d'une autre VOB. Notre choix consiste ainsi à enregistrer les objets disponibles sur une liste (0-disp), de sorte que les VOB sans valeur n'aient le choix que dans cette liste pour être instanciées et qu'à chaque fois qu'une VOB prend un objet de cette liste, on l'enlève de cette liste.

Au début de la démonstration d'un but, il faut déjà identifier les VOB et les enregistrer sur une liste (VOB). On a ainsi :

```
prouver(B, C, 0) -> (VAR-VAL | BOOL | ENTIER)
  var-val := VIDE
  VOB := trouver-VOB(B)
  0-disp := 0
  prouver0(B, var-val, C, 0, VOB, 0-disp)
```

Pour les fonctions prouver0, c-effacer et p-effacer, il ne faut pratiquement pas de modification, si ce n'est qu'un grossissement des listes d'arguements :

```
prouver0(B, var-val, C, 0, VOB, 0-disp) -> (VAR-VAL | BOOL | ENTIER)
  si B = VIDE alors var-val
  sinon
    type := type-pred(tete(B))
    cas
      1. type = 'ordinaire ' :
         clauses := c-candidats(tete(B), C)
```



```
         c-effacer(B, clauses, var-val, C, 0, VOB, 0-disp)
      2. type = 'obj-attr' :
         objets := o-candidats(var-val, tete(B), 0, VOB, 0-disp)
         o-effacer(B, objets, var-val, C, 0, VOB, 0-disp)
      3. type = 'voisin' :
         voisins := v-candidats(var-val, tete(B), 0, VOB, 0-disp)
         v-effacer(B, voisins, var-val, C, 0, VOB, 0-disp)
      4. type = 'predefini' :
         p-effacer(B, var-val, C, 0, VOB, 0-disp)
    fin-cas

--------------------

c-effacer(B, clauses, var-val, C, 0, VOB, 0-disp)
                                    -> (VAR-VAL | BOOL | ENTIER)
  si clauses = VIDE alors FAUX
  sinon
     clause := tete(clauses)
     var-val0 := unifier(var-val, tete(B), tete(clause))
     si var-val0 alors
        clause0 := remplacer-cut-par-entier(clause)
        nouveau-but := concat(queue(clause0), queue(B))
        cut := numero-cut(clause0)
        var-val1 := prouver0(nouveau-but, var-val0, C, 0, VOB, 0-disp)
        si var-val1 = cut alors FAUX
        sinon si var-val1 <> FAUX alors var-val1
        sinon c-effacer(B, queue(clauses), var-val, C, 0, VOB, 0-disp)
     sinon
        c-effacer(B, queue(clauses), var-val, C, 0, VOB, 0-disp)

--------------------

p-effacer(B, var-val, C, 0, VOB, 0-disp) -> (VAR-VAL | BOOL | ENTIER)
  P := tete(B)
  type := type-pred(P)
  cas
    1. type = ENTIER :
       var-val0 := prouver0(queue(B), var-val, C, 0, VOB, 0-disp)
```

```
      si var-val0 = FAUX alors P
      sinon var-val0
   2. type = '=' :
      si e-unifier(P) alors
         prouver0(queue(B), var-val, C, O, VOB, O-disp)
   3. type = 'vrai' :
      prouver0(queue(B), var-val, C, O, VOB, O-disp)
   4. type = FONCTON-LISP :
      si evaluer(P) = 'nil' alors FAUX
      sinon prouver0(queue(B), var-val, C, O, VOB, O-disp)
   5. type = 'not' :
      si prouver0(queue(P), var-val, C, O, VOB, O-disp) <> FAUX
         alors FAUX
      sinon prouver0(queue(B), var-val, C, O, VOB, O-disp)
fin-cas
```

Beaucoup de modifications doivent être apportées aux fonctions `o-candidats` et `v-candidats`, de sorte que celles-ci puissent engendrer des candidats en tenant compte de la liste des objets disponibles lorsqu'il s'agit d'instancier les VOB.

D'autre part, on doit modifier les fonctions `o-effacer` et `v-effacer` de manière à ce qu'elles puissent rendre compte de la diminution de la liste d'objets disponibles lorsqu'une VOB se trouve instanciée, ce qui se fait au moyen de deux fonctions : `o-reduire` dans le cas de `o-effacer` et `v-reduire` dans le cas de `v-effacer` :

```
o-effacer(B, objets, var-val, C, O, VOB, O-disp)
                                    -> (VAR-VAL | BOOL | ENTIER)
  si objets = VIDE alors FAUX
  sinon
     objet := tete(objets)
     var-val0 := o-unifier(var-val, tete(B), objet)
     si var-val0 <> FAUX alors
        nouveau-but := queue(B)
        O-disp0 := o-reduire(tete(B), objet, VOB, O-disp)
        var-val1 := prouver0(nouveau-but, var-val0, C, O, VOB, O-disp0)
        si var-val1 <> FAUX alors var-val1
        sinon o-effacer(B, queue(objets), var-val, C, O, VOB, O-disp)
     sinon
        o-effacer(B, queue(objets), var-val, C, O, VOB, O-disp)
```



```
--------------------

v-effacer(B, voisins, var-val, C, O, VOB, O-disp)
                                    -> (VAR-VAL | BOOL | ENTIER)
  si voisins = VIDE alors FAUX
  sinon
     voisin := tete(voisins)
     var-val0 := unifier(var-val, tete(B), voisin)
     si var-val0 <> FAUX alors
        nouveau-but := queue(B)
        O-disp0 := v-reduire(tete(B), voisin, VOB, O-disp)
        var-val1 := prouver0(nouveau-but, var-val0, C, O, VOB, O-disp0)
        si var-val1 <> FAUX alors var-val1
        sinon v-effacer(B, queue(voisins), var-val, C, O, VOB, O-disp)
     sinon
        v-effacer(B, queue(voisins), var-val, C, O, VOB, O-disp)
```

6.2.3 Gestion du démonstrateur

Pour faire fonctionner le démonstrateur, il faut lui fournir un but à démontrer, ainsi que, comme axiome, une liste de clauses et une liste d'objets ; on récupère comme résultat une liste d'associations variable-valeur permettant de rendre le but vrai.

Mais ce n'est pas tout. En effet, étant donné un but, il peut y avoir plusieurs solutions possibles, c'est-à-dire, plusieurs listes d'associations variable-valeur, toutes capables de le rendre vrai. Il faut donc fournir au démonstrateur une autre information, qui est la fonction à exécuter à chaque fois que le démonstrateur a trouvé une solution et se trouve dans la perspective d'en trouver une autre.

Pour faciliter l'utilisation du démonstrateur, nous définissons la structure de données `PTAB` qui regroupe quatre champs :

```
PTAB
   clauses
   objets
   feuille
   var-val
```

`feuille` contiendra la fonction que l'on vient de décrire. `clauses` et `objets` contiendront respectivement la liste de clauses et la liste d'objets à la disposition du démonstrateur

comme axiome. `var-val` sera un résultat de la démonstration, donc une liste d'associations variable-valeur.

Dans ce contexte, pour lancer le démonstrateur en vue de démontrer un but, il suffit de faire `prouver(but)`.

L'algorithme présenté précédemment correspondait en fait à la recherche d'une solution, étant donné un but à démontrer. Pour le rendre adaptable, il faut qu'il utilise le champs `PTAB.feuille`. Ainsi, il faut modifier la fonction `prouver0` de son ancienne version :

```
prouver0(B, var-val, C, O, VOB, O-disp) -> (VAR-VAL | BOOL | ENTIER)
  si B = VIDE alors var-val
  sinon
.....
```

à la nouvelle version :

```
prouver0(B, var-val, C, O, VOB, O-disp) -> (VAR-VAL | BOOL | ENTIER)
  si B = VIDE alors executer(PTAB.feuille)
.....
```

Illustrons l'utilisation de `PTAB.feuille` sur un exemple : en mettant la fonction suivante au `PTAB.feuille` :

```
afficher-var-val ()
  imprimer(PTAB.var-val)
  renvoyer FAUX
```

on verra afficher sur l'écran toutes les solutions possibles à un but.

6.3 Evaluateur partiel

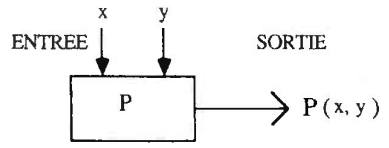
6.3.1 Evaluation partielle : une technique générale

L'évaluation partielle (voir [Futamura 82] pour une revue complète) d'un programme consiste à le spécialiser par rapport à des données incomplètes afin d'en obtenir une version spécifique mais plus efficace.

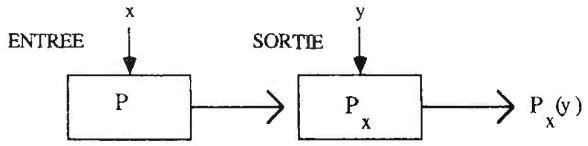
Par exemple, supposons que l'on ait le programme $\mathcal{P}$ qui prend deux données : x et y. Alors l'exécution de $\mathcal{P}$ par rapport à x et y donnera comme résultat $\mathcal{P}(x, y)$. Supposons maintenant que l'on emploie *souvent* le programme pour $x = 10$. Il s'agit donc de faire des

calculs de type $\mathcal{P}(10, y)$. Alors, au lieu d'utiliser tout le temps le programme général $\mathcal{P}$, il sera plus efficace d'engendrer, en faisant le plus de calculs possibles dans $\mathcal{P}$ pour $x = 10$, un nouveau programme, appelons $\mathcal{P}_{10}$. Ainsi, au lieu de calculer $\mathcal{P}(10, y)$ directement avec $\mathcal{P}$, on va faire $\mathcal{P}_{10}(y)$, ce qui sera plus efficace.

Le calcul du programme $\mathcal{P}$ par rapport à la donnée incomplète $x = 10$ pour avoir $\mathcal{P}_{10}$ est une évaluation partielle de $\mathcal{P}$, par opposition à l'évaluation complète où l'on fournirait également la valeur de y (cf. figure 6.2). Il est à noter qu'il s'agit de faire le plus de calculs possibles dans le programme $\mathcal{P}$ avec $x = 10$ pour avoir $\mathcal{P}_{10}$ afin que celui-ci soit efficace.



(a) évaluation complète



(b) évaluation partielle

Figure 6.2. Evaluation complète (a) et évaluation partielle (b)

Prenons encore un autre exemple pour illustrer la notion de l'évaluation partielle. L'interprète d'un langage peut être considéré comme un programme général $\mathcal{I}$ qui prend deux données : un programme P et sa donnée D. L'exécution de P avec la donnée D sous l'interprétation sera donc $\mathcal{I}(P, D)$. D'autre part, un compilateur $\mathcal{C}$ du langage est un programme qui ne prend qu'une donnée qui est un programme P, et le résultat de l'exécution est $\mathcal{C}(P)$ ou la version compilée du programme P. On a alors la relation suivante :

$$\mathcal{I}(P, D) = \mathcal{C}(P)(D),$$

traduisant le fait que l'exécution interprétée d'un programme donne le même résultat que l'exécution compilée.

Si l'on fait une évaluation partielle de $\mathcal{I}$ par rapport à P avant de prendre en compte la donnée D, on aura le programme $\mathcal{I}_P$, tel que,

$$\mathcal{I}_P(D) = \mathcal{I}(P, D) = \mathcal{C}(P)(D).$$

On a donc,

$$\mathcal{I}_P = \mathcal{C}(P),$$

qui signifie que la compilation d'un programme correspond en principe à l'évaluation partielle de l'interprète par rapport à celui-ci.

Pour terminer, on peut conclure ainsi :

1. L'évaluation partielle vise à améliorer l'efficacité d'un programme général au travers de sa spécialisation.
2. Elle est utile lorsqu'on a à utiliser plusieurs fois la version spécialisée du programme.

6.3.2 Evaluation partielle dans le système OASIS

Objectif

L'utilisation d'un évaluateur partiel dans le système OASIS vise à améliorer l'efficacité du démonstrateur de théorèmes. Comme on l'a vu précédemment, le démonstrateur sert à démontrer un but en prenant pour axiome conjointement un ensemble de clauses et un ensemble d'objets. Les clauses sont générales et figées au cours de la résolution d'un problème, traduisant la définition de certains concepts de chimie. Au contraire, les objets, qui représentent un *état* (cf. le chapitre 5), changent, ainsi que le but. Désignons par *démontrer* le tout composé du démonstrateur plus les clauses générales, il s'agit alors, à tout moment donné, d'effectuer le calcul :

$$démontrer(état, but),$$

c'est-à-dire démontrer un but sur un état.

Or, il est fréquent que l'on ait à effectuer plusieurs démonstrations (plusieurs buts à démontrer) sur un même état. En effet, sur un état courant, au cours de la résolution d'un problème, il faut, dans un premier temps, interpréter les méta-règles en vue de trouver les opérateurs à appliquer, ce qui implique la démonstration successive des prémisses de ces méta-règles ; ensuite, une fois les opérateurs sélectionnés, il faut les appliquer, ce qui implique alors la démonstration successive de leurs préconditions.

Ainsi, dans de telles situations, au lieu de faire toujours des calculs répétitifs :

$$démontrer(état, but),$$

avec *état* inchangé, il est souhaitable d'effectuer d'abord une évaluation partielle de *démontrer* par rapport à *état*, pour en avoir une version spécialisée :

$$démontrer_{état}.$$

Dès lors, la démonstration d'un but sur cet état consistera à calculer

$$démontrer_{état}(but),$$

ce qui sera plus efficace que $démontrer(état, but)$.

L'évaluateur partiel *peval* dans le cas présent accomplit ainsi la tâche :

$$peval(démontrer, état) \Longrightarrow démontrer_{état}.$$

Mise en œuvre

Notre évaluation partielle du programme de démonstration (le démonstrateur plus les clauses de Horn) consiste à spécialiser les clauses de Horn par rapport à un état donné pour en avoir une version spécifique à cet état.

De façon évidente, on peut classer les clauses en deux catégories :

1. celles qui font intervenir des prédicats d'accès aux objets, c'est-à-dire (`_x.attr val`) et/ou (`voisin _x _y _n`),
2. celles qui n'ont rien à voir avec les objets.

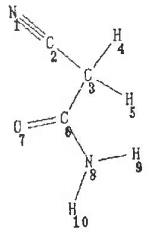
La spécialisation ne peut concerner que la première catégorie de clauses, car elle n'a pas de sens vis-à-vis de la deuxième catégorie, étant donné qu'il s'agit d'une spécialisation par rapport à un état qui est un ensemble d'objets. Par exemple, on pourrait trouver les clauses suivantes dans la deuxième catégorie :

```
(membre _x (_x . _)) &
(membre _x (_ . _y)) <- (membre _x _y) &
```

Isolément, il n'y a donc pas lieu de faire une spécialisation pour elles.

La première catégorie de clauses correspond à certaines affirmations concernant les structures moléculaires. Etant donné un état composé d'une ou plusieurs molécules, ces affirmations sont soit établies soit non-établies. Ainsi, la spécialisation de cette catégorie de clauses peut se ramener à trouver toutes les instances des têtes de clauses. Remarquons que ceci est un cas simple par rapport au cas de l'évaluation partielle de programmes PROLOG généraux [Takeuchi 86] qui nécessite un mécanisme pour détecter les boucles infinies éventuelles.

Prenons un exemple. Supposons que l'on ait un état composé d'une seule molécule qui est la suivante :



La spécialisation par rapport à cet état des clauses suivantes :

```
(CO _x _y) <- (_x.atome C) (voisin _x _y 2) (_y.atome) &
(e-attracteur _x) <- (CO _x _y) &
(e-attracteur _x) <- (CN _x _y) &
(CN _x _y) <- (_y.atome N) (voisin _y _x 3) (_x.atome C) &
(liberable _x _y) <- (_y.atome N) (voisin _x _y 1) (_x.atome H) &
(NH _x _n) <- (_x.atome N) (nbH _x _n) &
```

aura pour résultat les clauses unitaires suivantes :

```
(CO 6 7) &
(e-attracteur 6) &
(e-attracteur 2) &
(CN 2 1) &
(liberable 9 8) &
(liberable 10 8) &
(NH 8 2) &
```

Il est à noter que la spécialisation de la dernière clause a utilisé les clauses suivantes :

```
(nbH _x _nH) <- (nbHO _x () _nH) &
(nbHO _x _lH _nH) <- (voisin _x _y 1) (_y.atome H) (not (member _y _lH))
                     (nbHO _x (_y . _lH) _nH1) (= _nH (+ 1 _nH1)) / &
(nbHO _x _ 0) &
```

permettant au prédicat `(nbH _x _nH)` de calculer le nombre de H (atomes d'hydrogène) attachés à `_x`.

Nous adoptons une attitude pragmatique dans notre choix de spécialisation. En effet, nous ne spécialisons que les clauses susceptibles de conduire à des instances fréquemment utilisées ultérieurement, ce qui est tout à fait dans l'esprit de la pratique de l'évaluation partielle. Ainsi, les clauses précédentes définissant `(nbH _x _nH)` ne sont pas spécialisées.

En bref, l'évaluation partielle du programme de démonstration par rapport à un état consiste à spécialiser un sous-ensemble de l'ensemble des clauses ayant trait aux objets, en laissant intact le reste des clauses.

Pour expliciter quelles clauses spécialiser, il suffit de fournir un ensemble de prédicats **tetes** correspondant aux têtes des clauses à spécialiser. Ensuite, il suffit de faire démontrer ces prédicats l'un après l'autre en tant qu'un but par le démonstrateur de théorèmes, afin de retrouver toutes leurs instances possibles. La dernière phase consistera alors à substituer celles-ci à leurs clauses-mères à la disposition du démonstrateur. Voici l'algorithme de l'évaluateur partiel **peval** :

peval(tetes) :

1. **PTAB.clauses** := l'ensemble des clauses générales ;
2. **PTAB.objets** := l'ensemble des objets composant l'état par rapport auquel les clauses sont à spécialiser.
3. **PTAB.feuille** := la fonction suivante :

   peval-feuille ()
   (a) remplacer les variables dans le but initial B par leurs valeurs dans **PTAB.var-val**,
   (b) **resultat** := adjoindre(B, **resultat**),
   (c) renvoyer **FAUX** ;
4. **resultat** := **VIDE** ;
5. pour chaque B dans **tetes** faire tourner le demonstrateur pour demontrer B ;
6. substituer **resultat** aux clauses correspondantes dans **PTAB.clauses**.

La dernière instruction fait que le démonstrateur fonctionnera sur la version spécialisée des clauses après la phase d'évaluation partielle.

6.4 Interprète d'opérateurs

L'interprète d'opérateurs est destiné a appliquer un opérateur à un état, en vue de l'obtention d'un nouvel état :

$$état * opérateur \rightarrow état.$$

Il correspond à l'algorithme suivant :

`interp-op (etat, op)` :

1. Appeler le démonstrateur en vue de vérifier la précondition de l'opérateur `op` (`lhs` ou `rhs`) sur l'état `etat` En cas d'échec, renvoyer `echec` ; sinon, passer au point suivant.
2. Appeler le démonstrateur en vue d'évaluer la partie `certitude` de `op`. Si la valeur de `certitude` est en dessous d'un seuil prédéterminé, renvoyer `echec` ; sinon, passer au point suivant.
3. Effectuer les modifications nécessaires sur l'état `etat` en vue d'obtenir le nouvel état.
4. Le nouvel état obtenu étant toujours un ensemble d'objets *nœud* représentant un graphe, si celui-ci n'est pas connexe – il s'agit donc de plusieurs molécules – regrouper les objets en plusieurs groupes de sorte que chacun d'entre eux corresponde à un graphe connexe et séparé des autres.

Les étapes 1 et 2 peuvent être accomplies aisément par le démonstrateur. L'étape 3 correspond à la mise au monde du nouvel état. L'étape 4 correspond au passage de la vision d'un état comme un tout à celle comme molécules séparées. Nous allons voir en détails ces deux dernières étapes.

6.4.1 Génération du nouvel état

Le fait qu'un opérateur devienne applicable sur un état signifie que celui-ci possède une sous-structure (*pattern*) correspondant à `lhs` de l'opérateur (ou `rhs` s'il s'agit d'appliquer l'opérateur de droite à gauche). Alors, en remplaçant cette sous-structure `lhs` par une sous-structure correspondant à `rhs`, on obtiendra le nouvel état. On peut schématiser cette substitution comme suit :

$$\frac{\textit{état}}{\textit{rhs} \rightarrow \textit{lhs}} \longrightarrow \textit{nouvel état.}$$

Pour réaliser cette substitution, on ne tient compte que des prédicats de type (`_x.attr val`) et (`voisin _x _y _n`). Soit `lhs0` le `lhs` de l'opérateur auquel on a enlevé les prédicats des autres types, et `rhs0` la même chose mais pour `rhs`. Cette substitution consiste alors d'une part à supprimer dans l'état courant la structure correspondant à (`lhs0`−`rhs0`) — prédicats qui existent dans `lhs0` et non dans `rhs0` — et d'autre part à y ajouter la structure correspondant à (`rhs0`−`lhs0`). Cette modification de l'état se fait à travers des modifications des objets représentant l'état.

On a l'algorithme suivant traduisant ce processus de génération du nouvel état :

1. Enlever à `lhs` et `rhs` les prédicats des autres types que (`_x.attr val`) et (`voisin _x _y _n`) :


```
lhs0 := enlever-autres(lhs)
rhs0 := enlever-autres(rhs)
```

2. Faire rétrécir `lhs0` et `rhs0` en leur enlevant les prédicats qui leur sont communs :

```
lhs1 := difference(lhs0, rhs0)
rhs1 := difference(rhs0, lhs0)
```

3. Traduire la suppression de `lhs1` sur l'état courant en modifiant certains attributs des objets représentant celui-ci.
4. Traduire l'ajout de `rhs1` sur l'état courant en modifiant certains attributs des objets représentant celui-ci.

6.4.2 Restructuration de l'*état* en molécules

Le nouvel état est un ensemble d'objets représentant une ou plusieurs molécules. On s'intéresse ici à séparer ces objets *nœud* en plusieurs groupes de sorte que chacun d'entre eux corresponde exactement à une molécule (un graphe connexe). Voici l'algorithme :

```
regrouper(noeuds) -> LISTE-DE-LISTES-DE-NOEUDS
  si noeuds = VIDE alors VIDE
  sinon
     nd := tete(noeuds)
     nds := voisins(nd)
     groupe1 := un-groupe(nds, VIDE)
     reste := difference(noeuds, groupe1)
     adjoindre(groupe1, regrouper(reste))
```

où l'appel de la fonction `un-groupe` permet de trouver l'ensemble de nœuds constituant un graphe connexe et dont `nd` fait partie. La fonction `un-groupe` correspond à l'algorithme suivant :

```
un-groupe(noeuds, liste) -> LISTE-DE-NOEUDS
  si noeuds = VIDE alors liste
  sinon
     nds := voisins(tete(noeuds))
     liste := adjoindre(tete(noeuds), liste)
     noeuds := queue(noeuds)
     pour chaque x dans nds faire
        si non(membre(x, liste)) ET non(membre(x, noeuds)) alors
           noeuds := adjoindre(x, noeuds)
```

```
fin-pour
un-groupe(noeuds, liste)
```

6.5 Interprète des méta-règles ou conseiller

Les méta-règles représentent les stratégies de synthèse. Etant donné l'état courant du problème, leur interprétation permet de trouver un ensemble restreint d'opérateurs à tenter sur l'état, réduisant ainsi le risque combinatoire.

Nous avons vu au chapitre 5 qu'elles ont la forme suivante :

```
SI condition ALORS types-de-regles-a-essayer
```

La partie `condition` a la même syntaxe que `lhs` ou `rhs` d'un opérateur. Lorsqu'elle est satisfaite sur un état, l'application de l'opérateur donne comme résultat les types de règles à tenter sur celui-ci.

Nous envisageons que l'interprète des méta-règles interprète les méta-règles de façon successive et en chaînage-avant, de la manière suivante :

1. Appeler le démonstrateur en vue de vérifier la partie `condition` de la méta-règle en question sur l'état courant du problème.
2. Relever les types d'opérateurs prescrits par la méta-règle.
3. Rechercher parmi l'ensemble des opérateurs ceux qui correspondent à ces types.

6.6 Conclusion

Les mécanismes d'inférences dans le système OASIS combinent plusieurs techniques :

1. l'héritage de propriétés lié à l'usage des objets structurés (*atome*, *nœud*) ;
2. la démonstration de théorèmes en calcul des prédicats se fondant sur le formalisme des clauses de Horn et aussi celui des objets structurés ;
3. l'interpétation des règles de production ;
4. l'évaluation partielle.

Ces techniques se matérialisent en trois modules : un interprète d'opérateurs, un démonstrateur de théorèmes et un évaluateur partiel.

7
Structure de contrôle

7.1 Problème du contrôle

Si les mécanismes d'inférences de base permettent d'opérer des transitions dans l'espace d'états (ainsi que d'accomplir d'autres tâches isolées), le rôle du contrôle consiste à enchaîner les transitions suivant certaines stratégies en vue de la résolution d'un problème.

Le contrôle vise à développer un espace d'états ou espace de recherche qui soit le plus petit possible mais qui contienne la solution du problème à résoudre. Il comprend ainsi à tout moment donné au cours de la résolution d'un problème, deux points essentiels [Laurent 84] :

1. choisir un état à développer,
2. choisir les transitions à opérer sur cet état.

Dans les problèmes complexes, une exploration exhaustive de l'espace d'états n'est pas envisageable, le but n'étant pas de trouver toutes les solutions à un problème de façon rigoureuse, mais seulement quelques-unes d'entre elles, à cause des difficultés combinatoires. Les heuristiques jouent ainsi un rôle primordial. C'est le cas de la plupart des résolveurs de problèmes d'intelligence artificielle et des systèmes experts.

Il est à remarquer que le contrôle heuristique dans les systèmes experts et le contrôle dans la démonstration automatique de théorèmes ne cherchent pas le même but. Dans le premier, en supposant que la recherche de la meilleure ou de toutes les solutions implique un espace de recherche d'une taille incontrôlable, on adopte une approche heuristique et donc non-rigoureuse pour trouver une "assez bonne" solution (ou un sous-ensemble de toutes les solutions) au problème posé.

Dans le second, on se trouve dans le cadre d'une approche rigoureuse destinée à résoudre les problèmes ayant un espace de recherche théoriquement contrôlable. Le but recherché par le contrôle – notamment les stratégies régissant l'application du principe de résolution – consiste à trouver le plus rapidement possible (donc en effectuant le moins de

recherche possible) la ou les solutions au problème posé.

Dans le cadre du contrôle heuristique dans les systèmes experts, il n'y a bien entendu pas une approche universelle. Chaque problème correspond à des heuristiques qui lui sont plus ou moins propres. Dans un système spécifique, il s'agit donc d'implanter les heuristiques propres au domaine d'application.

Dans l'état actuel du système OASIS, nous n'avons pas encore exploré en profondeur les connaissances utiles au contrôle ou les méta-connaissances. Néanmoins, nous sommes en mesure d'exposer les grandes lignes concernant "l'aspect contrôle" du système.

7.2 Stratégies générales du système OASIS

Comme on l'a mentionné au chapitre 4, le système OASIS est conçu comme étant un système interactif – ou "ouvert" – de sorte que c'est l'utilisateur qui décide, au cours de la résolution d'un problème, de continuer de développer un nœud dans l'espace d'états ou de s'y arrêter. L'utilisateur sert ainsi de fonction d'évaluation. La plupart des systèmes existants adoptent cette stratégie interactive, comme le système SECS [Wipke 78] où l'on appelle cette approche la *recherche heuristique externe*.

L'adoption de cette stratégie interactive est motivée par plusieurs raisons. Avant tout, le problème de la synthèse consiste à trouver de "bons" chemins tels que le juge le chimiste utilisateur du système. Les critères du chimiste à cet égard sont liés à de nombreux facteurs tels que le nombre d'étapes dans la synthèse, le rendement, la connaissance du chimiste des étapes impliquées, etc., et aussi ses expériences du passé. D'autre part, la créativité du chimiste peut souvent s'inspirer d'une structure moléculaire particulière engendrée par le système, même si le chemin correspondant semble médiocre en apparence ; le chimiste saurait le rendre performant.

En bref, le système OASIS adopte le mode interactif où le chimiste joue le rôle de fonction d'évaluation. De son côté, le système assure le développement de l'espace d'états suivant certaines stratégies heuristiques exprimées par les méta-règles.

Reprenons les deux points du contrôle de la section précédente :

1. choisir un état à developper,
2. choisir les transitions à opérer sur cet état.

Le rôle du chimiste intervient ainsi dans le premier point où il décide de développer un état ou non, alors que le système lui propose les états susceptibles d'être développés de façon chronologique (par exemple, dans certains cas, profondeur d'abord ou largeur d'abord). Tout au long du développement de l'espace d'états, le système associe une valeur d'évaluation à chaque état engendré, en utilisant la valeur de `certitude` de chaque opérateur. Lorsque la valeur d'évaluation d'un état tombe en-dessous d'un seuil tolérable, le système écarte cet état. Dans le cas général, la valeur d'évaluation sert de référence au chimiste.

Le deuxième point du contrôle est accompli par le système qui interprète les méta-règles. Elles retrouvent les types d'opérateurs à appliquer sur l'état courant.

Nous allons ensuite examiner un peu plus en détail le cas de la rétrosynthèse.

7.3 Cas de la rétrosynthèse

7.3.1 Résolution du problème par réduction : arbre ET/OU

On peut schématiser la rétrosynthèse ainsi :

$$? \xrightarrow{?} C.$$

Il s'agit de synthétiser une molécule cible C à partir de molécules simples. C'est le cas le plus fréquent et aussi le plus important en synthèse chimique. La stratégie générale consiste à réduire la molécule cible.

En partant de la molécule cible, on applique les opérateurs dans le sens inverse, c'est-à-dire, de droite à gauche : `rhs` → `lhs`. Supposons que le premier opérateur `OP` soit,

$$C \Longrightarrow A + B,$$

permettant de réduire la cible initiale en deux nouvelles cibles. A noter que nous utilisons la double flèche pour indiquer l'application d'un opérateur dans le sens inverse ou le sens rétrosynthétique. Une autre notation équivalente de cet opérateur est :

$$A + B \longrightarrow C,$$

qui correspond à son application dans le sens normal, c'est-à-dire le sens du déroulement réel d'une réaction chimique.

Cette application du premier opérateur traduit le fait que la molécule cible peut être synthétisée par deux autres molécules plus simples : A et B.

Supposons qu'il y ait un autre opérateur `OP1` qui permet également de réduire la molécule cible C :

$$C \Longrightarrow A_1 + B_1.$$

On peut alors affirmer que C peut être synthétisée par (A et B) ou (A_1 et B_1). Autrement dit, le problème initial a été réduit en sous-problèmes A et B ou A_1 et B_1. Pour le résoudre

complètement, il faudra continuer de réduire les sous-problèmes jusqu'aux *feuilles* qui sont des molécules simples (matières premières).

Un problème en rétrosynthèse consiste ainsi à explorer un arbre ET/OU (*arbre de synthèse*). Pour l'exemple précédent, une partie de l'arbre de synthèse près de la racine est illustré sur la figure 7.1 où les traits continus correspondent aux branches ET et les traits pointillés aux branches OU.

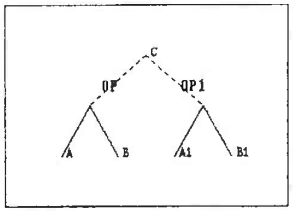


Figure 7.1. Arbre de synthèse : un arbre ET/OU

7.3.2 Exploration de l'arbre de synthèse

Notre stratégie générale d'exploration de l'arbre de synthèse consiste à parcourir les branches OU issues d'un même nœud en largeur et les branches ET en profondeur, ce qui signifie en fait une stratégie en profondeur mais avec les différentes transitions sur un nœud effectuées en un seul coup. Le parcours en largeur des branches OU permet à l'utilisateur de comparer les divers chemins possibles dès qu'ils sont engendrés. Il permet également au système de gagner en efficacité car, toutes les transitions sur un état étant opérées en même temps, il y a lieu de faire intervenir l'action de l'évaluateur partiel (cf. le chapitre 6).

On peut comprendre la chronologie liée à cette stratégie d'exploration de l'arbre de synthèse à l'aide de la figure 7.2 où un nœud portant un plus grand numéro est exploré plus tard et, comme sur la figure 7.1, les branches ET sont en traits continus et les branches OU en pointillés.

L'utilisateur intervient à un nœud pour lui associer l'une des trois valeurs d'évaluation suivantes :

1. réussit,
2. échoue,
3. à enchaîner.

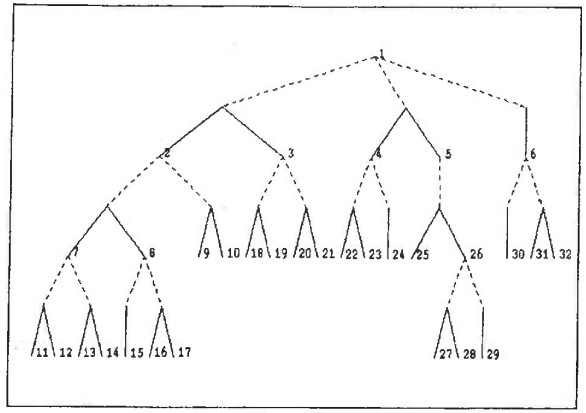


Figure 7.2. Exploration d'un arbre de synthèse

Le système calcule une valeur d'appréciation pour chaque nœud à partir des valeurs de `certitude` des opérateurs. Cette valeur d'appréciation sert de référence à l'utilisateur pour estimer l'éloignement du nœud courant par rapport au nœud de départ (la molécule cible). La manière de calculer la valeur d'appréciation est à l'étude.

Voici la manière dont on procède à l'exploration d'un nœud (état) :

1. déclencher l'évaluateur partiel par rapport à cet état ;
2. interpréter les méta-règles pour trouver les opérateurs à appliquer ;
3. interpréter successivement chacun de ces opérateurs et faire évaluer à l'utilisateur chaque nouvel état dès qu'il est engendré ;
4. prendre pour continuer le nouvel état correspondant à l'application du premier opérateur réussit.

7.4 Structure de données *média*

La structure de données *média* est destinée à faciliter la communication entre les différents modules du système. Elle enregistre les données initiales et intermédiaires relatives à la résolution d'un problème. Chaque module y lit des données nécessaires au cours de son exécution et écrit les résultats obtenus.

De façon non-exhaustive, le *média* regroupe les champs suivants :

```
MEDIA
   etat-courant
   operateur-courant
   nouvel-etat
   operateurs-disponibles
   mode-de-fonctionnement
   ......
```

`nouvel-etat` est l'état obtenu à l'issue de l'application de `operateur-courant` sur `etat-courant`. `operateurs-disponibles` sont les opérateurs à appliquer sur `etat-courant`. `mode-de-fonctionnement` indique le sens d'interprétation des opérateurs (de gauche à droite ou de droite à gauche).

7.5 Conclusion

Le contrôle dans le système OASIS a pour rôle d'orienter intelligemment le développement de l'espace d'états d'un problème. Pour ce faire, il exploite des heuristiques – sous forme de méta-règles – pour trouver un ensemble restreint d'opérateurs à appliquer sur l'état courant du problème d'une part ; il s'appuie sur une interactivité avec l'utilisateur qui sert de fonction d'évaluation des états d'autre part.

8
Divers

8.1 Rôle de l'expert dans ce travail

L'expert (le professeur P. Caubère) a joué un rôle important tout au long de ce travail. On peut résumer ses interventions en trois points essentiels :

1. Définition du problème de la synthèse chimique. Il a ainsi fixé les objectifs finals du projet OASIS du point de vue d'un expert-utilisateur.
2. Transmission de ses connaissances sur la synthèse chimique au travers des interviews. Il nous a fait part de sa vision globale du problème et de sa manière d'organiser les connaissances : atomes, molécules, perception d'une molécule, réactions, stratégies de synthèse.
3. Validation de nos résultats ou de nos réflexions.

Les deux derniers points se répètent au cours de la réalisation du projet, formant le cycle *explication-validation*. Notre objectif est de simuler, d'un point de vue structurel, le plus fidèlement possible la pensée de l'expert, ce qui implique :

1. la bonne compréhension de la pensée de l'expert,
2. la recherche des moyens informatiques appropriés.

Dans la suite du projet OASIS, l'expert interviendrait plus souvent sur des points précis pour :

1. apporter des précisions sur, par exemple, une réaction spécifique,
2. expliciter ses méta-connaissances, c'est-à-dire, ses stratégies régissant la résolution d'un problème de synthèse,

toujours suivant le cycle *explication-validation*.

8.2 Etat actuel du système OASIS

Avec les outils développés dans ce travail – démonstrateur de théorèmes, interprète d'opérateurs, évaluateur partiel, outil d'affichage de molécules – le système OASIS est actuellement à l'état de démonstration sur l'interprétation des réactions de façon isolée. Ainsi, les exemples de réactions (opérateurs) présentés dans ce mémoire proviennent du fonctionnement du système. Visuellement, le résultat est satisfaisant.

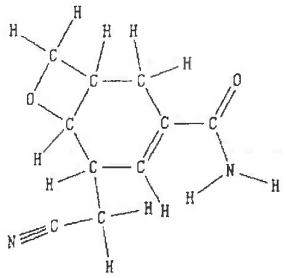
Faute d'avoir en main les informations nécessaires concernant les systèmes existants, une comparaison directe d'efficacité ne peut être effectuée. D'un autre point de vue, une telle comparaison serait surtout utile en interprétant les réactions de façon enchaînée, ce que le système OASIS n'est pas en mesure de faire à l'état actuel.

8.3 Affichage des molécules

8.3.1 Définition du problème

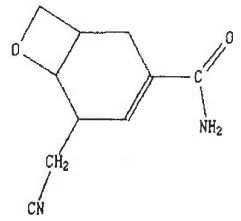
Le problème de l'affichage consiste à visualiser sur écran d'ordinateur les molécules à partir de leur représentation interne LISP.

Prenons un exemple de molécule :



Sa représentation interne est un ensemble d'objets *nœud* (cf. le chapitre 5) qui spécifie deux types d'informations : la nature atomique des nœuds (éventuellement aussi leurs charges électriques) et les liaisons entre les eux. La visualisation ci-dessus traduit bien cette représentation interne.

Cependant, une autre présentation graphique est plus habituelle :



On préfère en effet d'afficher certains groupements d'atomes comme un tout (un seul nœud) et l'on ne fait pas figurer les atomes d'hydrogène et de carbone non ambigus.

De manière générale, la visualisation d'une molécule procède par quatre étapes :

1. regroupement des nœuds (atomes) en nouveaux nœuds,
2. recherche des cycles et des cycles contigus,
3. calcul des coordonnées pour les nœuds,
4. affichage proprement-dit, moyennant éventuellement une homothétie et une rotation.

8.3.2 Regroupement des nœuds en nouveaux nœuds

Le regroupement des nœuds en nouveaux nœuds peut être accompli avec le démonstrateur de théorèmes, moyennant la définition des clauses de Horn traduisant les groupements de nœuds devant être affichés comme un tout. Par exemple, pour le groupement NH_n, on peut définir la clause suivante :

```
(NH _x _n _l) <- (_x.atome N) (vnn H _x _l ()) (= _n (length _l))
                 (> _n 0) &
```

où _x est l'atome du centre (un N en l'occurrence), _n un nombre à mettre en indice de NH dans l'affichage et _l la liste des atomes à supprimer (les _n H en l'occurrence). Le prédicat (vnn H _x _l ()) permet de retrouver les atomes H attachés à l'atome _x, grâce aux clauses suivantes :

```
(vnn _A _x (_y . _lA) _B) <- (voisin _x _y 1) (_y.atome _A)
                             (not (member _y _B))
                             (vnn _A _x _lA (_y . _B)) / &
(vnn _ _ () _) &
```

En respectant la même syntaxe vis-à-vis des têtes de clauses, on peut définir pour les groupements d'atomes : CH_n, CN et OH les clauses suivantes :

```
(CH _x _n _l) <- (_x.atome C) (vnn H _x _l ()) (= _n (length _l))
                 (> _n 0) &
(CN _x 1 (_y)) <- (_x.atome C) (voisin _x _y 3) (_y.atome N) &
(OH _x 1 (_y)) <- (_x.atome O) (voisin _x _y 1) (_y.atome H) &
```

Ainsi, les groupements d'atomes destinés à être affichés comme un tout sont spécifiés de façon déclarative sous forme de clauses de Horn.

8.3.3 Recherche des cycles et des cycles contigus

Recherche des cycles

Il s'agit de retrouver les cycles les plus petits présents dans un graphe molécule. Nous procédons en deux étapes :

1. rechercher tous les cycles et
2. dégager les cycles les plus petits.

La première étape correspond à l'algorithme suivant :

1. Prendre n'importe quel nœud pour nœud de départ.
2. Développer progressivement un arbre.
3. A tout moment, vérifier si un nœud feuille de l'arbre figure déjà dans la branche correspondante. Si oui, on a trouvé un cycle et arrête le développement de cette branche. Si ce cycle ne figure pas dans la liste des cycles retrouvés, le mettre dedans.

L'algorithme suivant permet de retrouver parmi l'ensemble (une liste) de cycles `cycles` les cycles les plus petits :

```
petits (cycles) -> CYCLES
  petits0 (VIDE, tete(cycles), queue(cycles))

--------------------

petits0 (cycles1, cycle, cycles2) -> CYCLES
  cas
    1. cycle = NEANT : cycles1
    2. cycle composable de concat(cycles1, cycles2) :
```



```
       petits0 (cycles1, tete(cycles2), queue(cycles2))
    3. sinon :
       cycles1 := adjoindre(cycle, cycles1)
       petits0 (cycles1, tete(cycles2), queue(cycles2))
  fin-cas
```

Recherche des cycles contigus

Il s'agit de regrouper les cycles (les plus petits) en plusieurs groupes dont chacun est un ensemble de cycles qui sont contigus entre eux. On procède de la manière suivante :

1. Prendre un cycle pour commencer.
2. Associer à ce cycle les cycles ayant au moins un nœud en commun avec lui pour former le groupe `G1`, le reste des cycles étant `R`.
3. Enlever à `R` les cycles ayant au moins un nœud en commun avec un (ou plus) des cycles dans `G1` et les intégrer dans `G1`. Répéter ce processus autant de fois que possible.
4. `G1` devient alors le premier groupe de cycles contigus. Reprendre le point 1 avec `R` pour trouver `G2`, `G3`, etc.

8.3.4 Calcul des coordonnées

Il s'agit de calculer une position relative pour chacun des nœuds d'un graphe molécule. Nous procédons de la manière suivante :

1. Commencer par un groupe de cycles contigus.
2. Développer les branches de ces cycles.
3. A la rencontre d'autres cycles, reprendre le point 1.

Ces calculs sont très compliqués. Nous ne les détaillons pas ici.

8.3.5 Affichage

L'affichage proprement-dit consiste à mettre les nœuds sur écran ainsi que les liaisons correspondantes, moyennant éventuellement une homothétie et une rotation.

L'ensemble des présentations graphiques de molécules dans ce mémoire provient du présent programme d'affichage. Il semble que le calcul des coordonnées des nœuds reste à améliorer pour que la présentation soit plus conforme aux configurations spatiales des molécules.

8.4 Saisie des molécules et des opérateurs

8.4.1 Saisie

C'est l'inverse de ce qu'accomplit le programme d'affichage. Il s'agit de transformer en représentation interne LISP ce que l'utilisateur dessine sur écran d'ordinateur comme opérateur ou molécule. Notre matériel est un poste de travail SUN-UNIX. La saisie se fait au moyen de la souris. Nous n'allons pas entrer dans le détail ici.

A remarquer que l'on a la possibilité d'utiliser directement les groupements fonctionnels à condition que leur définition respecte une syntaxe appropriée que nous avons prédéfinie. Par exemple, pour $COOH$, il faut écrire :

```
COOH = (C 0) (O 1) (O 2) (H 3) (O 1 2) (O 2 1) (2 3 1);
```

8.4.2 Vérification de la validité des molécules

Il s'agit de vérifier la validité chimique d'une molécule saisie. Par exemple, un atome carbone ne doit pas avoir plus de quatre liaisons.

De façon générale, les connaissances concernant la validité chimique des molécules peuvent être exprimées sous forme de clauses de Horn. Le critère le plus élémentaire porte, étant donné un atome (un *nœud*), sur la relation entre sa valence, le nombre d'électrons se trouvant sur la dernière couche, sa charge électrique et le nombre de ses liaisons, de la manière suivante :

```
(noeud-valid _x) <- (nb-liaisons _x _n) (_x.ion _i) (_x.valence _v)
                    (d-couche _x _nbE) (contrainte _n _nbE _i _v) &
(nb-liaisons _x _n) <- (nb-liais0 _x () _n) &
(nb-liais0 _x _l _n) <- (voisin _x _y _b) (not (member _y _l))
                        (nb-liais0 _x (_y . _l) _n0)
                        (= _n (+ _b _n0)) / &
(nb-liais0 _ _ 0) &
(contrainte _n _nbE _i _v) <- ... &
```

`(contrainte _n _nbE _i _v)` devrait exprimer cette relation.

A remarquer que ce même principe pourrait aussi être appliqué aux molécules intermédiaires dans la résolution d'un problème pour, par exemple, vérifier leur stabilité et donc leur droit d'existence.

9
Conclusion

Ce travail s'inscrit dans le cadre du projet OASIS : un système expert d'aide à l'élaboration des plans de synthèse chimique.

Une contribution à caractère général est le développement du modèle de *l'espace d'états avec abstraction de concepts* qui est un affinement de la notion d'espace d'états classique. Dans le cadre de ce modèle, le domaine d'un problème donné est décrit en trois composantes :

1. les états qui sont représentés en terme de concepts de base,
2. les concepts d'abstraction qui sont définis sous forme de clauses de Horn,
3. les opérateurs de transition dont la description repose à la fois sur les concepts de base et sur les concepts d'abstraction.

En nous fondant sur ce modèle, nous avons réparti les connaissances dans le système OASIS sur quatre niveaux :

1. Les molécules, qui sont des états, sont représentées chacune comme un ensemble d'objets. Sur le plan logique, on peut les assimiler à des conjonctions de prédicats du type `(_x.attr val)` ou `(voisin _x _y _n)`, c'est-à-dire, prédicats élémentaires.
2. Les clauses de Horn assurent le lien entre prédicats élémentaires et prédicats élaborés.
3. Les opérateurs (réactions) sont décrits en terme de prédicats élémentaires et de prédicats élaborés.
4. Les méta-règles (stratégies de recherche) seront représentées également en terme de ces deux types de prédicats.

D'un point de vue global, notre approche à la représentation de connaissances est une approche mixte, faisant intervenir les objets structurés, les clauses de Horn et les règles de production.

Quant aux mécanismes d'interprétation des connaissances, nous avons fait coopérer plusieurs techniques : l'héritage de propriétés lié à l'usage des objets structurés, la dé-

monstration de théorèmes se fondant sur les clauses de Horn et les objets, l'évaluation partielle des clauses de Horn, l'interprétation des opérateurs.

Une suite à ce travail devrait se dérouler autour de deux axes. Le premier est l'acquisition des connaissances et surtout celle des opérateurs (réactions). Un travail en cours (voir L. Pierron dans [Haton 87,Sun 88]) concerne l'utilisation des techniques de l'apprentissage symbolique automatique pour l'enrichissement, à partir de la trace d'une consultation interactive du système, des bases de connaissances chimiques.

Le deuxième axe est la réalisation d'un contrôleur performant s'appuyant sur des méta-connaissances. La mise au point de ce contrôleur permettra une orientation intelligente du développement de l'espace d'états du problème ainsi qu'un arbitrage efficace des différents agents intervenant dans la résolution du problème.

Bibliographie

[Barone 82] R. Barone, M. Chanon, P. Cadiot, et J.-M. Cense. Microcomputers and organic synthesis. *Bull. Soc. Chim. Belg.*, 91(4):333–336, 1982.

[Bourlard 89] H. Bourlard et C. J. Wellekens. Les applications des modèles connexionnistes à la reconnaissance de la parole. *Architectures avancées pour l'intelligence artificielle*, pages 181–192, EC2, 1989.

[Challioux 86] J. Challioux et al. *Le_Lisp version 15.2 : manuel de référence.* INRIA, mai 1986.

[Chang 73] C.-L. Chang et R. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[Choplin 85] F. Choplin. L'ordinateur en chimie. *Pour la Science*, 50–60, novembre 1985.

[Colmerauer 72] A. Colmerauer, H. Kanoui, R. Pasero, et P. Roussel. *Un système de communication homme-machine en français.* Rapport, GIA, Université d'Aix-Marseille II, octobre 1972.

[Corey 69] E. J. Corey et W. T. Wipke. Computer-assisted design of complex organic synthesis. *Science*, 166:178, 1969.

[Davis 75] R. Davis et J. King. An overview of production systems. *Machine Intelligence*, 10, 75.

[De Saint Pierre 87] T. De Saint Pierre. Codification et apprentissage connexionniste de caractères multipolices. *Proc. Cognitiva*, pages 284–289, 1987.

[Ernst 69] G. W. Ernst et W. Newell. *A Case Study in Generality and Problem Solving.* Academic Press, 1969.

[Feigenbaum 71] E. A. Feigenbaum, B. G. Buchanan, et J. Leiderberg. On generality and problem solving: a case study using the DENDRAL program. *Machine Intelligence*, 6, 71.

[Ferber 86] J. Ferber. La programmation par acteur (1). *Micro-Systèmes*, 140–171, mars 1986.

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

[Futamura 82] Y. Futamura. Partial computation of programs. *Journal of IECE of Japan*, 66(2):157–165, 1982.

[Gelernter 84] H. Gelernter, G. A. Miller, D. L. Larsen, et D. J. Berndt. Realisation of a large expert problem-solving system. *Proc. First Conference of Artificial Intelligence Applications*, pages 92–106, 1984.

[Haton 85] J.-P. Haton. Intelligence artificielle en compréhension automatique de la parole : état des recherches et comparaison avec la vision par ordinateur. *T.S.I.*, 3:285–266, 1985.

[Haton 87] M.-C. Haton, B. Devin, L. Pierron, et P. Caubère. Representation des connaissances et apprentissage en synthèse organique : une démarche experte. *Proc. Cognitiva*, pages 303–307, 1987.

[Haton 89] J.-P. Haton. Panorama des systèmes multi-agents. *Architectures avancées pour l'intelligence artificielle*, pages 247–261, EC2, 1989.

[Johnson 85] A. P. Johnson. Computer aids to synthesis planning. *Chemistry in Britain*, 59–67, janvier 1985.

[Jones 87] W. P. Jones et J. Hoskins. Back-propagation. *Byte*, 155–162, octobre 1987.

[Josin 87] G. Josin. Neural-network heuristics. *Byte*, 183–192, octobre 1987.

[Kosko 87] B. Kosko. Constructing an associative memory. *Byte*, 137–144, septembre 1987.

[Kowalski 74] R. Kowalski. Predicate logic as programming language. *Proc. IFIP Congress*, pages 569–574, 1974.

[Kowalski 76] R. Kowalski. A proof procedure using connection graphs. *J. ACM*, 22(4):572–595, 1976.

[Laurent 84] J.-P. Laurent. Structure de contrôle dans les systèmes experts. *T.S.I.*, 3(3):161–176, 1984.

[Lesser 75] V. R. Lesser, R. D. Fennell, L. D. Erman, et D. R. Reddy. Organization of the HEARSAY-II speech understanding system. *IEEE Trans. ASSP*, 23:11–23, 1975.

[Loveland 84] D. W. Loveland. Automated theorem-proving: a quarter centuary review. *Contemporary Mathematics vol. 29*, pages 1–45, 1984.

[Marchand 86] P. Marchand. Langages Formels. Polycopié du cours de DEA, Université de Nancy I, 1986.

[Matwin 87] S. Matwin et T. Pietrzykowski. Intelligent backtracking in plan-based deduction. *IEEE Trans. PAMI*, 7(6):682–691, 1987.

[Minsky 75] M. Minsky. A framework for representing knowledge. P. H. Winston, éditeur, *The Psychology of Computer Vision*, McGraw-Hill, 1975.

[Motooka 85] T. Moto-oka et M. Kitsuregawa. *The Fifth Generation Computer*. John Wiley & Sons, 1985.

[Mylopoulos 83] J. Mylopoulos et H. Levesque. An overview of knowledge representation. M. L. Broodie, J. Mylopoulos, et J. V. Schmidt, éditeurs, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer-Verlag, 1983.

[Nilsson 71] N. Nilsson. *Problem Solving Methods in Artificial Intelligence*. 1971.

[OHare 85] G. M. O'Hare et D. A. Bell. The coexistence approach to knowledge representation. *Expert Systems*, 2(4):230–237, 1985.

[Quillian 68] R. Quillian. Semantic memory. M. Minsky, éditeur, *Semantic Information Processing*, MIT Press, 1968.

[Robinson 65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[Sacerdoti 74] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *J. Artificial Intelligence*, 5:115–135, 1974.

[Shirai 87] Y. Shirai et J. Tsujii. *Intelligence Artificielle*. Editions Eyrolles, 1987.

[Shortliffe 76] E. Shortliffe. *Computer-Based Medical Consultations:* MYCIN. American Elsevier, 1976.

[Sun 88] Y. Sun, L. Pierron, et M.-C. Haton. Résolution de problème et raisonnement expert en synthèse de molécules organiques : le système OASIS. *Proc. Journées Internat. Systèmes Experts*, pages 303–307, Avignon, 1988.

[Takenouchi 87] H. Takenouchi et Y. Iwashita. An integrated knowledge representation scheme for expert systems. *Expert Systems*, 4(1):38–43, 1987.

[Takeuchi 86] A. Takeuchi et K. Furukawa. Partial evaluation of PROLOG programs and its application to meta-programming. H.-J. Kugler, éditeur, *Information Processing 86*, pages 415–420, Elsevier, 1986.

[Van Caneghem 86] M. Van Caneghem. *L'Anatomie de Prolog*. InterEditions, 1986.

[Warren 78] S. Warren. *Designing Organic Syntheses*. John Wiley & Sons, 1978.

[Winston 84] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, 1984.

[Wipke 78] W. T. Wipke, G. I. Ouchi, et S. Krishnan. Simulation and evaluation of chemical synthesis – SECS: an application of artificial intelligence techniques. *J. Artificial Intelligence*, 11:173–193, 1978.

Résumé

Ce travail porte sur la représentation des connaissances et les mécanismes d'inférences de base dans le système expert OASIS : un système d'élaboration des chemins de synthèse en chimie organique. Nous avons d'abord développé un modèle général de représentation de problèmes – le modèle de l'*espace d'états avec abstraction de concepts* – qui est un affinement de la notion d'espace d'états classique en y introduisant un niveau de *concepts d'abstraction* de plus par rapport au niveau des *concepts de base*. Ce modèle décrit le domaine d'un problème donné en trois parties : 1. la représentation des états en terme de concepts de base ; 2. la définition, au moyen des clauses de Horn, des concepts d'abstraction à partir des concepts de base ; 3. la description des opérateurs de transition en terme de concepts de base et de concepts d'abstraction. Le mécanisme d'inférences correspondant repose sur trois composantes de base : 1. un démonstrateur de théorèmes servant à démontrer la précondition d'un opérateur sur un état donné modulo les clauses de Horn, 2. un évaluateur partiel destiné à spécialiser les clauses de Horn par rapport à un état en vue d'améliorer l'efficacité du démonstrateur de théorèmes, 3. un interprète d'opérateurs permetant de passer d'un état à l'autre. Le problème de la synthèse chimique se trouve aisément formulé dans le modèle de l'espace d'états avec abstraction de concepts : les molécules constituent les états, les concepts de base étant les concepts élémentaires en chimie juste suffisants pour spécifier la structure d'un graphe-molécule ; les concepts d'abstraction correspondent surtout à certains termes génériques (*libérable, électro-attracteur, polarisé*, etc.) et aux groupements fonctionnels (CN, $COOH$, etc.) ; les opérateurs correspondent aux réactions chimiques dont la description s'appuie sur les deux types de concepts. Reposant sur cette vision générale, nous avons développé le formalisme détaillé de représentation des connaissances qui intègre les objets structurés, le calcul des prédicats et les règles de production. Nous avons implanté les mécanismes d'inférences de base qui combinent plusieurs techniques : l'héritage de propriétés lié à l'usage des objets structurés, la démonstration de théorèmes intégrant la syntaxe des clauses de Horn et celle des objets, l'interprétation des règles de production et l'évaluation partielle des clauses de Horn.

Mots-clés :

- résolution de problèmes - espace d'états -
- abstraction de concepts - démonstration de théorèmes -
- évaluation partielle - système expert -
- synthèse chimique -