INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Manuel de référence du Logiciel OCAPI Version 2.0

Sabine Moisan, Régis Vincent, Monique Thonnat,
Véronique Clément, John van den Elst

N° 0183

Novembre 1995

PROGRAMME 3

Rapport technique

ISSN 0249-6399

Manuel de référence du Logiciel OCAPI
Version 2.0

Sabine Moisan, Régis Vincent, Monique Thonnat,
Véronique Clément, John van den Elst

Programme 3 — Intelligence artificielle, systèmes cognitifs
et interaction homme-machine
Projet Orion

Rapport technique n° 0183 — Novembre 1995 — 77 pages

Résumé : Ce document est composé de quatre parties : la première intitulée **Présentation introductive** décrit les principaux concepts intervenant dans OCAPI et leurs rôles ; la seconde intitulée **Composants d'une base de connaissances** approfondit ces notions et leur utilisation pratique ; la troisième **Moteur et base de faits** présente plus particulièrement le fonctionnement du moteur d'OCAPI et la quatrième **Méthodologie d'utilisation** propose une aide méthodologique à l'utilisateur d'OCAPI pour construire une base de connaissances.

L'interface du logiciel OCAPI est décrite dans le rapport technique [MPT+95]

Mots-clé : Pilotage de programmes, Système à base de connaissances

(Abstract: pto)

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

OCAPI Reference Manual
Version 2.0

Abstract: This document is composed of four parts: the first one entitled **Présentation introductive** describes the main concepts of OCAPI and their roles. The second part: **Composants d'une base de connaissances** deepens these notions and their practical use. The third part: **Moteur et base de faits** focuses on the functionning of OCAPI's engine and the fourth one: **Méthodologie d'utilisation** provides a methodological guide for the OCAPI user to build a knowledge base.

The use of the graphical interface of OCAPI is decribed in a technical report [MPT+95]

Key-words: Program Supervision, Knowledge-Based System

Chapitre 1

Présentation introductive

Ocapi (*O*util de *C*ontrôle *A*utomatique de *P*rocédures *I*mages) est un générateur de systèmes experts, spécialisé dans le pilotage de programmes, initialement conçu pour le domaine du traitement d'images. Il est capable de piloter de manière autonome toutes les étapes nécessaires à un traitement (par exemple l'analyse des images) et de contrôler les résultats obtenus.

La première version d'Ocapi [Clé90] a été originellement conçue en visant plus particulièrement des applications en traitement d'images. Cet outil a été utilisé dans différents domaines : traitement automatique d'images de galaxies [CT91, CT93c, TCO95], ou détection d'obstacles dans un environnement routier [vdE92, TCvdE94]. Ces expériences ont permis de raffiner le modèle [CT93b], qui a été étendu [Ma90, Ala91, Vin93], pour aboutir à la version présentée dans ce rapport.

Un système expert contient la connaissance d'un expert qui développe les bases de connaissances et est utilisé par un utilisateur final qui choisit le contexte et les données d'entrée.

1.1 Description générale

Ocapi est un outil en pilotage de programmes. C'est un générateur destiné à la planification et à l'exécution de programmes d'une bibliothèque pre-existante, pour automatiser le plus possible les différentes phases qui sont nécessaires pour atteindre un certain but.

Ocapi a été conçu selon l'architecture classique à trois composants :

- un moteur indépendant de toute application,

- une base de connaissances développée pour un certain domaine d'application,
- une base de faits dépendante de l'application courante.

Le terme de "*moteur*" est utilisé pour deux composants différents d'Ocapi :

- le moteur de règles qui sert à évaluer les règles de la base de connaissances, c'est le "*moteur*" du générateur dans le sens classique ; c'est un moteur d'ordre zéro en chaînage avant ;
- le moteur de pilotage qui lance automatiquement les programmes pour résoudre un problème. Ce moteur a deux fonctionnalités principales :
  - la planification pour automatiser la sélection des opérateurs ; elle permet de trouver un plan (séquence hiérarchique d'exécution des opérateurs) approprié,
  - le contrôle d'exécution qui sert à initialiser les paramètres, à évaluer les résultats des programmes exécutés et à ajuster ces paramètres en bouclant éventuellement jusqu'à obtenir des résultats d'une qualité satisfaisante.

Une fois qu'une base de connaissances a été créée, le moteur de pilotage de programmes peut exécuter les différentes phases du traitement automatiquement. La planification consiste à choisir entre différents *opérateurs* et à décider de l'ordre d'exécution de ces opérateurs pour atteindre un certain *but* dans un *contexte* défini. Cet algorithme principal est décrit au paragraphe 3.1.1. La planification (donc le choix et l'ordre des opérateurs) dépend du contexte, du but donné et du cas particulier (par exemple une image) à traiter. Les *opérateurs*, les *buts* et le *contexte* sont décrits dans la section suivante.

1.2 Représentation des connaissances

La connaissance est représentée sous forme de *buts*, de *requêtes*, d'*opérateurs* (élémentaires et complexes) et de *règles*. Ocapi fournit deux formalismes différents pour représenter deux types de connaissance différents :

1. des objets structurés pour la connaissance descriptive,
2. des règles de production pour la connaissance déductive.

La connaissance descriptive se compose de toutes les informations disponibles sur les programmes qui seront exécutés automatiquement par Ocapi, sur les liens entre les différentes étapes de traitement, etc. Ce type de connaissance sera représentée en Ocapi par des *objets structurés.*

Les experts humains possèdent de plus une connaissance qui leur permet de choisir un opérateur approprié, d'initialiser et/ou d'ajuster la valeur de certains arguments (en fonction du contexte, du comportement des opérateurs et de résultats intermédiaires) et enfin d'évaluer les résultats obtenus. Ce type de connaissance présente plutôt un aspect déductif ou inférentiel. Elle peut être exprimée facilement dans un formalisme appelé *règle de production.*

1.2.1 Les objets

- Un *but* représente une fonctionnalité du domaine d'expertise (traitement d'images, traitement du signal, ...). Celle-ci pourra être réalisée par un algorithme ou par un traitement plus complexe.

- Un *opérateur* contient la connaissance spécifique pour résoudre un but donné. En Ocapi il y a deux types d'opérateurs : des *opérateurs élémentaires* qui décrivent les programmes de la bibliothèque et les *opérateurs complexes* , qui sont des décompositions en sous-étapes. Les opérateurs complexes permettent d'exprimer la combinaison et le séquencement d'étapes plus simples, afin d'obtenir un traitement complexe.

- Aux buts et opérateurs, sont associés des *arguments d'entrée et de sortie.* Leur description fait également partie de la base de connaissances (*par exemple une valeur numérique comprise entre 0 et 1*). Une requête pourra fixer par exemple des valeurs spécifiques à ces arguments : *image-entree = "g17", seuil = 0.3.*

- Le *contexte* d'utilisation fournit des informations symboliques concernant soit les données (pour une image, par exemple les conditions d'acquisition, le domaine d'application et même des informations sémantiques sur le contenu de la scène) soit l'environnement d'utilisation (par exemple mode automatique ou interactif).

- La demande de résolution d'un problème s'exprime sous forme de *requête.* Une requête précise quel est le but à atteindre, les objets du cas particulier à traiter (par exemple, la désignation de l'image) et la qualité des résultats qu'il faut obtenir.

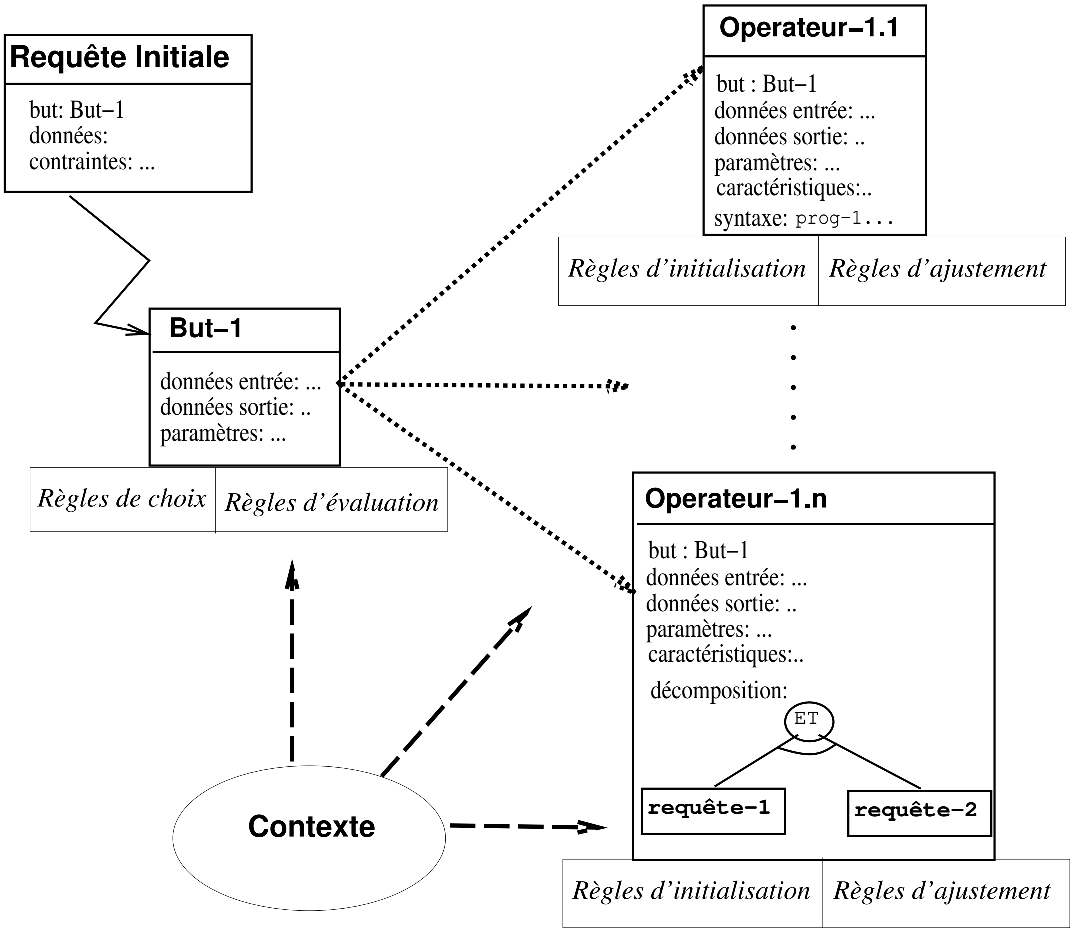

FIG. 1.1 – *Relations entre objets et règles dans Ocapi.*

1.2.2 Les règles

– Les *règles* dans Ocapi permettent un traitement dynamique, adapté à la situation courante (par exemple dépendant des résultats des opérateurs précédents). Elles sont *spécialisées* en quatre types : règles de choix, règles d'évaluation, règles d'initialisation et règles d'ajustement. Buts et opérateurs ne se voient pas attacher le même type de règles. Les deux premiers types de règles sont attachés aux buts (un but sait évaluer ses résultats et juger s'ils sont satisfaisants ou pas), les deux derniers types de règles sont attachés aux opérateurs (un opérateur sait ajuster les valeurs de ses paramètres). Les bases de règles sont donc *locales* et *spécialisées.*

Chaque objet dispose de ses règles propres : les choix entre des méthodes de seuillage ou entre des méthodes de classification, par exemple, ne s'effectuent pas selon les mêmes critères et les paramètres d'un seuillage ou d'une segmentation ne sont pas réglés de la même façon.

1. Les *règles de choix* sont attachées au but commun et elles servent à choisir, entre plusieurs opérateurs ayant tous le même but, celui qui correspond le mieux aux critères du cas particulier à traiter pour résoudre le problème.
2. Les *règles d'évaluation* contiennent la connaissance pour évaluer les résultats de l'exécution d'une requête ; elles sont attachées au but correspondant. Elles permettent de déterminer si les résultats (les valeurs de sortie) des opérateurs choisis sont satisfaisants.
3. Les *règles d'initialisation* expriment une partie de la connaissance attachée à un opérateur. Pour pouvoir utiliser l'opérateur choisi, il faut définir les valeurs d'entrée de ses paramètres, généralement en fonction du contexte et de la requête en cours de traitement.
4. Les *règles d'ajustement* permettent d'exprimer la connaissance sur la façon de modifier les valeurs des paramètres d'entrée d'un opérateur en cas de bouclage. Si les valeurs des arguments de sortie n'ont pas été jugées satisfaisantes par l'évaluation, on peut boucler sur le même opérateur, en modifiant les valeurs de certains paramètres d'entrée en fonction des résultats obtenus par ses exécutions précédentes pour la même requête. A chaque opérateur correspond un nombre maximal n de corrections et d'exécutions possibles. Si au bout de n essais les résultats ne sont toujours pas acceptables, l'opérateur est refusé.

1.3 Le moteur

Le rôle du moteur est d'utiliser la connaissance décrite par l'expert de manière à satisfaire une requête initiale formulée par l'utilisateur, correspondant à un but (cf figure 1.1). Le moteur construit un plan sous forme d'une suite d'opérateurs à appliquer, il choisit à tout moment le meilleur opérateur (en utilisant les règles de choix), il initialise les paramètres (à l'aide des règles d'initialisation), vérifie les résultats et ajuste les paramètres si besoin (à l'aide des règles d'évaluation et d'ajustement). Il peut abandonner un opérateur qui a échoué pour en essayer un autre.

1.3.1 Architecture du moteur

Le moteur fournit, en s'appuyant sur la structure de la base de connaissances, plusieurs fonctionnalités. Il permet d'une part le choix des traitements à exécuter, d'autre part le contrôle de l'exécution de ces traitements à l'aide d'un mécanisme de récupération d'erreurs. Le moteur dispose de plusieurs modules de base : un composant spécialisé pour chacun des raisonnements utilisés : le planificateur pour sélectionner et ordonnancer les modules de la bibliothèque, et le contrôleur d'exécution, pour calculer les paramètres, faire fonctionner les programmes et contrôler leurs résultats. Ces deux fonctionnalités, planification et contrôle d'exécution, sont imbriquées. Le moteur comprend de plus une interface avec les programmes de la bibliothèque et un interprète de règles. Nous donnons en figure 1.2 le graphe d'appel de ces composants.

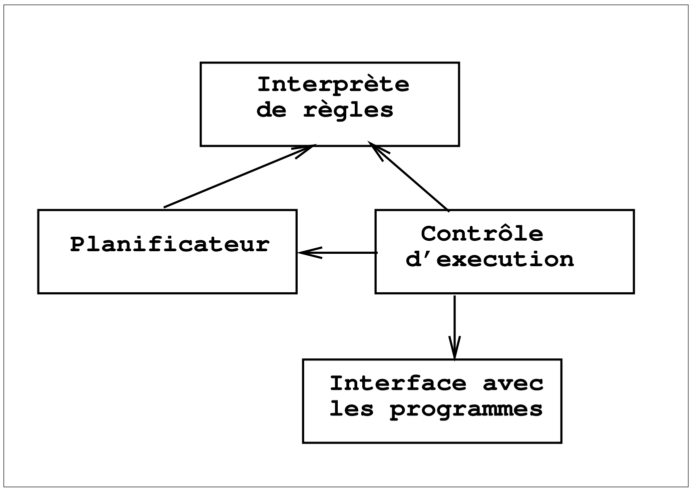

FIG. 1.2 – *Graphe d'appels du moteur*

Le planificateur effectue le choix des traitements à exécuter pouvant permettre de résoudre le problème courant. Il fonctionne en deux étapes : une simple phase de sélection qui réalise un filtrage sur les traitements dont le but est la résolution d'un problème analogue au problème posé, et une phase plus complexe, utilisant des inférences, devant choisir entre les différents traitements possibles (cf page 53).

Le contrôleur d'exécution permet l'optimisation du plan d'actions ainsi obtenu. Il comporte un sous-module : le paramétreur, spécialisé dans la gestion des valeurs de paramètres, qui détermine les valeurs des paramètres des traitements que ce soit en phase d'initialisation ou en phase d'ajustement et qui en gère l'historique.

L'interface d'exécution a pour rôle l'exécution effective des programmes de la bibliothèque et la récupération de leurs résultats dans la base de faits.

L'interprète de règles est un interprète de règles général, le même quelque soit le type des règles. Ainsi, l'expert dispose d'un seul formalisme de règles (même syntaxe, mêmes possibilités de contrôle).

1.4 La base de faits

La base de faits, c'est-à-dire l'état du problème courant, peut être plus ou moins complexe. Elle contient principalement les données de départ, un contexte particulier, les différents traitements déjà exécutés sur ces données, les données acquises par les traitements.... Elle sera donc représentée de multiples façons : par des fichiers, des listes de propriétés, des objets...

La base de faits contient l'information globale sur les données à traiter dans un objet appelé contexte, dont la structure est définie par l'expert lors du développement de la base de connaissances.

Elle contient l'information locale aux opérateurs dépendante des traitements exécutés, par l'intermédiaire de leurs arguments qui sont les données de travail, les résultats obtenus, ainsi qu'une description des fichiers stockés en dehors du système expert (exemples : nombre de régions, moyenne de l'image, périmètre du plus gros objet...)

Les données stockées à l'extérieur du système expert le sont sous forme de fichiers. Elles peuvent contenir des images, histogrammes, courbes, mesures... Ce sont des données d'entrée, des résultats intermédiaires et des résultats finaux. Elles sont le prolongement direct de la base de faits interne.

Certains faits sont donnés par l'utilisateur en début (spécification du problème) ou en cours de traitement, d'autres sont déduits par le système (système expert ou programmes).

1.5 Comment obtenir Ocapi ?

Pour obtenir le logiciel Ocapi, contactez :

Projet Orion
INRIA Sophia Antipolis
2004, route des Lucioles
B.P. 93
06902 SOPHIA-ANTIPOLIS Cedex (France)
Tel : (33) 93 65 76 57
Fax : (33) 93 65 79 39

Chapitre 2

Composants d'une base de connaissance

Ce chapitre présente en détails et à l'aide d'exemples la structuration d'une base de connaissance et les différentes entités qui la composent.

2.1 Structure d'une base de connaissance

2.1.1 Bases de règles

Les règles sont rattachées aux objets «buts» et «opérateurs», formant des bases de règles locales et spécialisées. Selon le type de l'objet (but ou opérateur), il n'aura pas les même types de règles : par exemple ce sont les buts qui savent évaluer les résultats et juger s'ils sont ou non satisfaisants, par contre ce sont les opérateurs qui savent ajuster leurs paramètres. Chaque objet a ses règles propres en fonction de l'opération particulière qu'il réalise ou de la fonctionnalité qu'il implante.

2.1.2 Réseau des objets

Les objets sont interconnectés en réseaux de la manière suivante :

- A un but sont attachés un ou plusieurs opérateurs entre lesquels il pourra choisir à chacune de ses activations. Un but sera représenté comme dans la figure 2.1.
- Un opérateur élémentaire, qui correspond à un programme de la bibliothèque est un nœud terminal. Un opérateur complexe est relié à une ou plusieurs

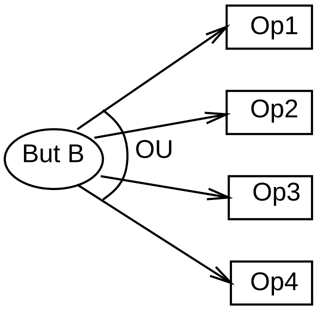

FIG. 2.1 – *Le but B peut être résolu par un des opérateurs Op1, Op2, Op3 ou Op4*

requêtes. Celles-ci seront exécutées soit de manière séquentielle (liaison de type *Puis*) soit de manière parallèle (liaison de type *Et*). Des requêtes sont reliées par un *Puis* si il y a une dépendance entre les résultats de l'une et les arguments d'entrée de l'autre. (voir figure 2.2). Sinon, elles sont reliées par un *Et*.

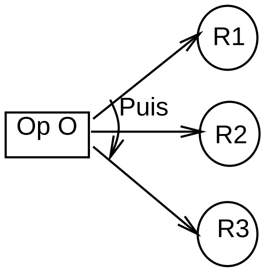

FIG. 2.2 – *L'opérateur O est une exécution séquentielle des requêtes R1, R2 et R3.*

– Une requête est un appel à un but, avec des données particulières (voir figure 2.3), et lui est donc reliée par un lien de type instanciation (*is-a*).

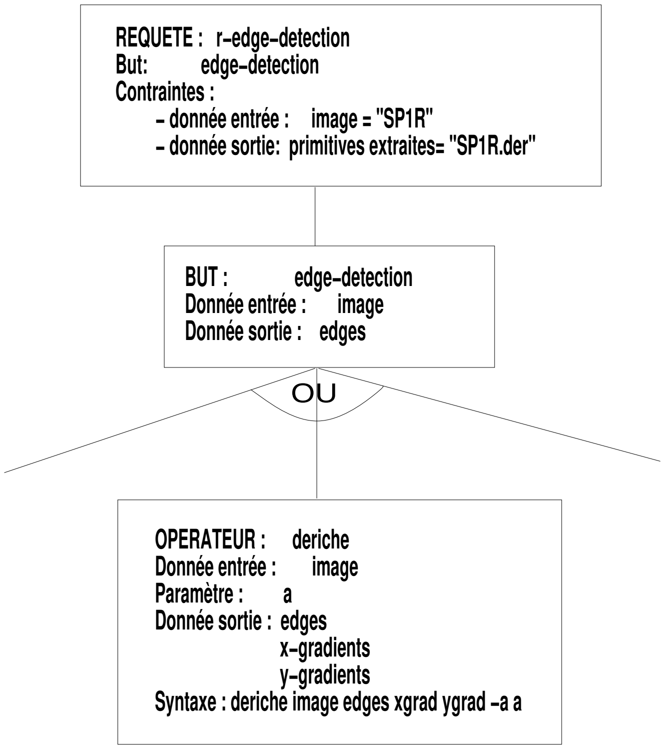

FIG. 2.3 – *Un sous-réseau d'objets*

2.2 Buts

Un but correspond à une fonctionnalité de traitement (par exemple en traitement d'images, le *filtrage* est un but). La structure d'un but est présentée figure 2.4. Un but a un nom, sa fonctionnalité est décrite sous forme de commentaire, il travaille sur des arguments (données et paramètres) d'entrée et a des arguments (données) de sortie. Chacun des opérateurs sachant résoudre ce but doit avoir au moins les mêmes arguments.

Un but contient la connaissance permettant de choisir entre les différents opérateurs qui lui sont associés : les règles de choix (par exemple : si *bruit très-haut* alors utiliser-operateur *filtre-gaussien-récursif*). Il a aussi la connaissance lui permettant de tester le résultat de son exécution : les règles d'évaluation (par exemple : si *taille* $<$ *15* alors *image-extraite trop-petite*).

But	
nom :	*symbole*
fonctionnalité :	*commentaire*
données-entrée :	*liste d'arguments*
paramètres :	*liste d'arguments*
données-sortie :	*liste d'arguments*
règles-choix :	*liste de règles*
règles-évaluation :	*liste de règles*
opérateurs :	*liste d'opérateurs*

FIG. 2.4 – *Structure d'un but avec en italique le type de chaque champ*

Un exemple de but est montré figure 2.5.

2.3 Opérateurs

C'est grâce aux opérateurs qu'une requête pourra effectivement être résolue. Cela peut se faire en utilisant un traitement complexe ou un programme.

Un opérateur (structure présentée figure 2.6) est nommé (par exemple *filtre-médian*), il est associé à un but et la fonctionnalité qu'il sait exécuter est décrite sous forme de commentaire. On lui associe des nombres maximum de bouclage : nombre maximum de convergence et d'optimisation (cf 2.3.1). Il possède aussi des

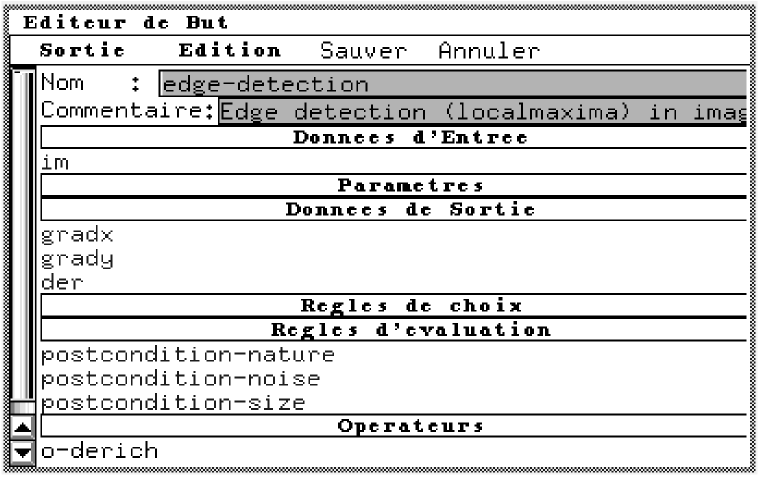

FIG. 2.5 – *Le but "edge-detection"*

caractéristiques (cf 2.3.2). Il travaille sur des arguments (données et paramètres) d'entrée et a des arguments (données) de sortie. Il a au moins les mêmes arguments que le but auquel il correspond, mais il peut en avoir plus (par exemple des paramètres spécifiques).

L'opérateur a également la connaissance lui permettant de choisir des valeurs initiales pour les paramètres d'entrée, dans les règles d'initialisation. Pour choisir une valeur, ces règles vont par exemple, pouvoir tenir compte du contexte ou poser des questions à l'utilisateur ; c'est en celà qu'elles ne sont pas redondantes avec le mécanisme de valeur par défaut associé aux arguments, qui lui permet seulement d'exprimer une valeur sous forme d'une expression (voir section 2.4.2). Des règles d'ajustement sont également associées à l'opérateur : elles permettent de préciser quand doit avoir lieu l'ajustement de tel ou tel paramètre de l'opérateur et quelle méthode utiliser pour ajuster sa valeur en cas d'échec de la précédente exécution de l'opérateur pour une requête donnée. Par exemple : si *trop-de-points à 1* alors *augmenter seuil.*

2.3.1 Bouclages

Il existe un mécanisme de bouclage sur un même opérateur, qui est réglé par deux nombres associés aux opérateurs.

Opérateur	
nom :	*symbole*
but :	*objet but*
fonctionnalité :	*commentaire*
nb-max-convergence :	*entier*
nb-max-optimisation :	*entier*
caractéristiques :	*liste de symboles*
données-entrée :	*liste d'arguments*
paramètres :	*liste d'arguments*
données-sortie :	*liste d'arguments*
règles-initialisation :	*liste de règles*
règles-ajustement :	*liste de règles*
décomposition :	*arbre de requêtes* (pour les opérateurs complexes)
appel :	*objet* (pour les opérateurs élémentaires)
	(= langage + syntaxe)

FIG. 2.6 – *Structure d'un opérateur avec en italique le type de chaque champ*

Ces deux nombres interviennent au moment de la sélection du meilleur opérateur par le moteur, qui se fait en deux passes. La première consiste à examiner tous les opérateurs, avec un nombre de bouclage faible, la deuxième permet de réexaminer un opérateur en bouclant plus longtemps. A chaque opérateur sont donc associés deux nombres. Le *nombre maximum de convergence* représente le nombre de bouclages maximum autorisé dans la première passe ; en général au bout de quelques ajustements, il est possible de déterminer s'il y a convergence vers une bonne mesure ou non. Le *nombre maximum d'optimisation* est le nombre de bouclages maximum autorisé pour optimiser l'exécution d'un opérateur. Ces nombres peuvent être consultés ou modifiés en partie conclusion des règles d'ajustement (voir page 49).

2.3.2 Caractéristiques des opérateurs

En plus de l'indication du nom du but, il est possible de donner une définition plus précise des caractéristiques d'un opérateur, pour affiner la planification. Les opérateurs possèdent pour cela un champ "caractéristiques" qui contient simplement une liste de symboles (**sans parenthèses**) représentant les caractéristiques de l'opérateur.

Exemples (pour le traitement d'images de galaxies)

Opérateur :	filtre-median
But :	filtrage
Caractéristiques :	flux-non-conservé, forme-conservée

En effet la méthode de filtrage médian ne conserve pas le flux mais préserve les formes des objets. Toutefois, ce filtre peut supprimer d'importantes caractéristiques structurelles comme les tâches chaudes ou les noyaux de galaxie.

Opérateur :	filtre-gaussien
But :	filtrage
Caractéristiques :	flux-conservé, paramètres-structurels-non-conservés

En général, avec une méthode classique de filtrage avec le profil gaussien on obtient un bon compromis. Le profil gaussien est choisi semblable à celui de l'étoile. Ce filtrage conserve le flux mais modifie certains paramètres structurels comme l'élongation centrale et les gradients.

L'accès à ces caractéristiques se fait dans les règles de choix grâce aux fonctions (utiliser-operateur-de-caractéristique 'caractéristique), (refuser-operateur-de-caractéristique 'caractéristique) et (caractéristique-op-sélectionné).

2.3.3 Opérateurs élémentaires

Les *opérateurs élémentaires* correspondent à des programmes particuliers ; ils sont décrits par l'information syntaxique nécessaire à leur exécution, qui précise le langage dans lequel ils ont été programmés et donne leur syntaxe d'appel.

Le langage peut être du Lisp, du C, ou une commande du langage d'exploitation. Si le langage est Lisp il s'agit d'un appel à une fonction Lisp (définie généralement dans un fichier Lisp associé à la base de connaissances), si le langage est C il s'agit généralement d'un appel de C depuis Lisp (voir la documentation Lisp), s'il s'agit d'un appel à une commande du système d'exploitation (shell sous Unix par exemple) cette commande doit être exécutable. Un exemple est montré pour l'opérateur élémentaire "*o-derich*", figure 2.7, dans le cas où le langage du système d'exploitation est *shell* (Unix).

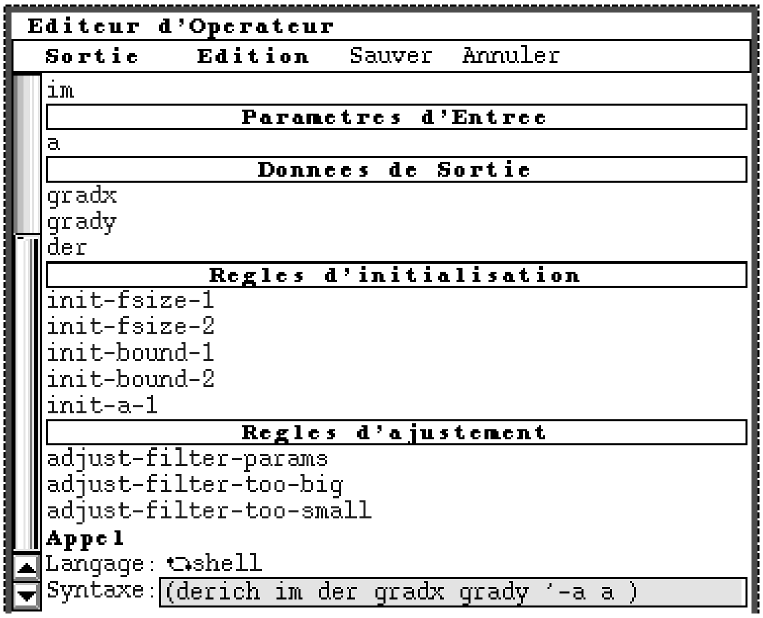

FIG. 2.7 – *L'opérateur élémentaire "o-derich"*

2.3.4 Opérateurs complexes

Un opérateur complexe représente un traitement décomposable en plusieurs étapes, par exemple l'extraction de primitives dans une image (opérateur complexe "*o-extract-prim*", figure 2.8) pourra se faire par les étapes suivantes : *détection des contours*, *calcul de l'orientation des contours*, *seuillage par hystérésis* et *chaînage.* Ces étapes s'expriment sous forme de requêtes. Elles peuvent être exécutées en séquence (*Puis*) ou en parallèle (*Et*). La séquence est utilisée quand il y a des dépendances entre les traitements : un traitement a besoin des résultats d'un traitement précédent. Les étapes sont structurées en arbres *Puis* et *Et* de requêtes internes, en précisant le *flux de données* entre les requêtes, leurs buts et l'opérateur complexe (voir section 2.5.4).
Remarque : actuellement, *Puis* et *Et* sont traités de la même façon, c'est à dire séquentiellement.

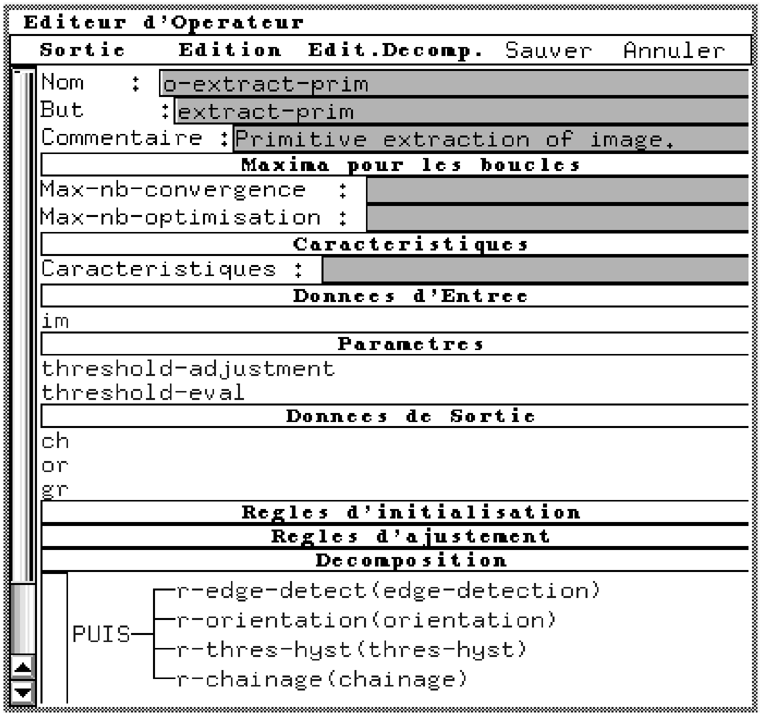

FIG. 2.8 – *L'opérateur complexe "o-extract-prim"*

2.4 Arguments

Les buts et les opérateurs travaillent sur des arguments, en entrée et en sortie : un argument d'entrée est connu ou calculé avant l'exécution ; il contient l'information sur laquelle a lieu le traitement. Les arguments d'entrée sont soit des données (issues du problème à traiter) inchangeables, soit des paramètres d'entrées, modifiables, qui peuvent être initialisés ou réglés. La valeur d'un argument (donnée) de sortie est calculée pendant l'exécution, sa localisation, (par exemple un nom de fichier) pouvant être imposée à l'avance.

Un opérateur peut avoir plus d'arguments que n'en a son but associé, ce seront en général des paramètres, tel qu'un seuil ou un type de parcours qui n'a pas de sens au niveau du but ; les données (images par exemple) sont par contre en général présentes au niveau du but.

En conséquence, il existe dans Ocapi plusieurs types d'arguments (données, paramètres) et de plus il y a une distinction entre les arguments des buts et ceux

des opérateurs, car l'information à apporter est différente selon qu'elle est associée à un but ou à un opérateur. La description des arguments au niveau d'un but est très générale. L'information exprimée au niveau de l'opérateur est plus spécifique (valeur par défaut, descriptions, ...) : la valeur par défaut peut être distincte selon chaque opérateur. L'argument de l'opérateur hérite des propriétés exprimées dans l'argument correspondant du but.
Voir figures 2.9, 2.10 et 2.13 les différentes structures d'arguments.

2.4.1 Arguments d'un but

Les arguments attachés à un but ont tous la même structure (décrite figure 2.9), qu'ils soient d'entrée ou de sortie, c'est le typage qui les distingue.

Argument-entrée-but		**Argument-sortie-but**	
nom :	*symbole*	nom :	*symbole*
fonctionnalité :	*commentaire*	fonctionnalité :	*commentaire*
type :	*symbole*	type :	*symbole*
description sémantique :	*symbole*	description sémantique :	*symbole*

FIG. 2.9 – *Structure des arguments de but*

Un argument est décrit par son nom, sa fonctionnalité, exprimée actuellement sous forme de commentaire, son type (image, texte, entier,..) et sa description sémantique (la seule qui ait un sens au niveau du but, voir section 2.4.3).

Exemple : argument de sortie d'un but.

nom :	*tc*
fonctionnalité :	*image apres seuillage par hystérésis*
type :	*image*
description sémantique :	*image-routiere*

2.4.2 Arguments d'un opérateur

Les arguments attachés à un opérateur ont des structures différentes selon qu'ils sont d'entrée (figure 2.10) ou de sortie (figure 2.13) et selon qu'il s'agit de données ou de paramètres, mais ont un certain nombre de champs communs.

Les champs communs sont : le nom (s'il est le même que celui d'un argument du but, il y a héritage des propriétés), la fonctionnalité, exprimée actuellement sous forme de commentaire, le type, la description d'implémentation (qui peut se préciser seulement au niveau de l'opérateur, voir section 2.4.3) et le champ défaut.

Le champ défaut n'a pas la même sémantique d'utilisation selon le type d'argument. Pour les arguments d'entrée il contient une valeur par défaut, qui est une valeur affectée à l'argument dans le cas où aucun autre élément de choix de valeur n'est disponible. Cette valeur peut être littérale (numérique ou symbolique) ou calculée (une expression faisant éventuellement intervenir des valeurs d'autres arguments).
Pour les données de sortie, ce champ défaut contient plutôt un nom par défaut : soit une localisation imposée (nom précis de fichier où l'on veut ranger les données en sortie), soit un nom plus générique (par exemple nom du fichier d'entrée auquel on ajoute une extension).

Les données (d'entrée ou de sortie) ont un champ "Méthode de visualisation", qui permet d'indiquer un utilitaire permettant de visualiser la donnée. La visualisation peut se faire par appel à un utilitaire de visualisation d'images, ou par appel à un utilitaire de visualisation de courbes ou par appel à un simple affichage pour les fichiers de texte ou les littéraux (valeurs numériques ou symboliques). Par exemple, pour un exécutable shell, il suffit de mettre le nom de cet exécutable à appeller dans le champ *Méthode de visualisation* de l'argument (exemples : "cat" pour un texte ascii, "xvisu" pour une image inrimage, ...).

Les données ont aussi un champ "reset" qui permet de préciser quand (avant/après l'exécution de l'opérateur) et comment (de façon facultative ou obligatoire, et par quelle commande) l'argument doit être réinitialisé. Ceci est utilisé, par exemple, dans le cas de programmes n'acceptant pas de travailler sur des images de sortie déjà existantes si elles n'ont pas le bon type de codage ; il faut d'abord détruire l'image existante (ce champ est plutôt utile pour les données de sortie et sera probablement supprimé pour les données d'entrée).

Les paramètres d'entrée, qui en Ocapi sont en principe numériques, ont un champ "intervalle" qui précise leur domaine de valeurs possibles.

Des exemples de donnée et paramètre d'entrée d'opérateurs sont montrés figures 2.11 et 2.12

Les arguments de sortie ont pour champ spécifique : l'*accès* . Il contient une méthode d'accès qui sera appelée après chaque exécution de l'opérateur. Elle exprime

Argument-entrée-opérateur	
nom :	*symbole*
fonctionnalité :	*commentaire*
défaut :	*symbole, numérique ou expression*
type :	*type prédéfini*
description-implémentation :	*type prédéfini*
méthode-visualisation :	*symbole de commande*
reset :	*objet* (pour une donnée)
valeurs-possibles :	*intervalle* (paramètre numérique)

FIG. 2.10 – *Structure d'un argument d'entrée d'opérateur*

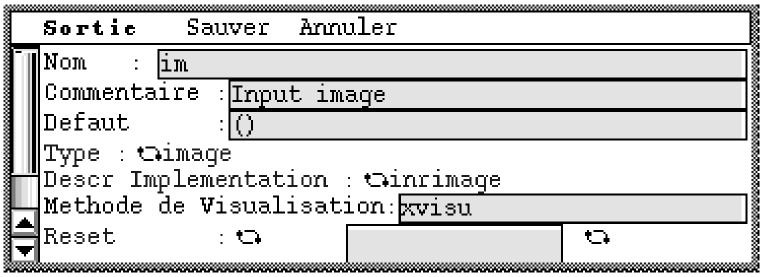

FIG. 2.11 – *Une donnée d'entrée de l'opérateur "o-derich"*

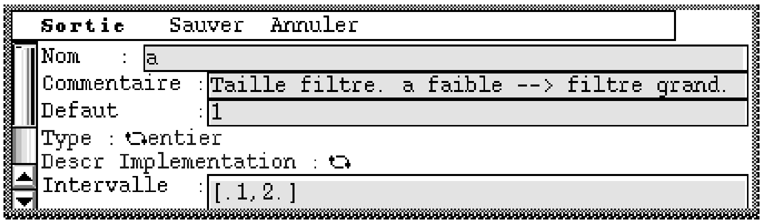

FIG. 2.12 – *Un paramétre d'entrée de l'opérateur "o-derich"*

la manière d'accéder à la valeur de l'argument parmi les informations calculées lors de l'exécution de l'opérateur. Cela peut être :

– le retour de l'exécution (utilisé dans le cas d'opérateur élémentaire codé en Le_Lisp ou en C, c'est à dire dont le code est dans l'environnement de travail), le mot clef "retour" dans le champ *accès* indique cette possibilité,

- un accès par l'intermédiaire d'un fichier ; l'accès se fait soit en donnant le numéro de l'item à rechercher dans le fichier, soit en donnant un symbole et un numéro (l'item retourné sera alors celui de rang *numéro* dans la première ligne du fichier contenant le symbole),
- le contenu d'une variable,
- ou l'évaluation d'une fonction.

Argument-sortie-opérateur	
nom :	*symbole*
fonctionnalité :	*commentaire*
type :	*symbole d'un des types prédéfinis*
description-implémentation :	*symbole de description*
méthode-visualisation :	*symbole de commande*
reset :	*objet*
accès :	*objet*

FIG. 2.13 – *Structure d'un argument de sortie d'opérateur*

Exemple : argument de sortie d'un opérateur :

nom :	*nbchain*
fonctionnalité :	*nombre de chaînes*
type :	entier
description-implémentation :	()
accès :	item *6* de la ligne *est* du fichier *chaine.r*

Ici l'exécution de l'opérateur a créé un fichier *chaine.r*, qui contient des résultats. Dans ce fichier il y a une ligne contenant le mot *est* (c'est : "nombre de chaines est : 52"), dont le 6ième item (52) correspond à la valeur qu'on veut mettre dans l'argument *nbchain*.

2.4.3 Description des données

Les descriptions servent à représenter les informations locales à une donnée particulière. Les descriptions sont structurées par types de données.

Il y a actuellement sept types différents : image, texte, liste, courbe, symbole, entier, réel (resp. en anglais *image, text, list, graph, integer, float*). A chaque type sont associés des "formats" d'implémentation (par exemple pour un entier : nombre d'octets sur lesquels il est codé, pour une image : manière dont est construite l'entête. Les formats prédéfinis sont actuellement, pour le type *image*: inrimage et visilog, pour le type *text*: ascii, pour le type *list*: dzv, pour le type *graph*: dzv et inrimage, pour le type *symbol* aucun type n'est défini, pour le type *integer*: entier, 1byte et 2bytes, et pour le type *float*: 1byte et 2bytes.

A chaque type est aussi attaché une description. Une description se décompose en deux parties : une partie implémentation et une partie sémantique.

Exemple

Description d'implémentation		
		moyen d'obtention
type	image	
format	inrimage	
contenu	taille-x	*but et argument* (calcul-taille x)
	taille-y	*but et argument* (calcul-taille y)
	nbre-pixels	(* (valeur-descr '(taille-x)) (valeur-descr '(taille-y)))
	valeur-minimale	(minmaxmoy mini) *but et argument*
	valeur-maximale	(minmaxmoy maxi)
	variance	

Description sémantique		
type	image	
description	scene-routiere	
contenu	taille-min-auto	
	taille-max-auto	
	nature :	inconnu, niveaux-gris, derich-gradients, gradients-x, gradients-y, orientations-gradient
	primitives	
	nombre :	
		suffisant : vrai, faux, inconnu

La partie implémentation est liée au format d'implémentation et l'expert définit autant de descriptions d'implémentation que de formats. Cette partie décrit des informations "objectives", mesurables, et indique les moyens d'obtention des valeurs de ses champs (ou éléments de description). Par exemple, pour une donnée de type image, l'expert peut définir sa moyenne, son minimum, son maximum,... Chacun des champs (comme *valeur-minimale*,..) a un type (symbolique/numérique) et un moyen d'*obtention* qui indique la façon dont peut être obtenue la valeur du champ. Ce moyen d'obtention peut être décrit soit par le nom d'un but de la base de connaissances, soit par une forme Lisp.

- S'il est décrit par le nom d'un but, il faut aussi indiquer le nom de l'argument de retour qu'il faut prendre en compte pour récupérer la valeur. Par exemple dans la description d'implémentation précédente pour le type image, on a fait appel au but *minmaxmoy* (qui doit faire partie de la base), sachant qu'il a comme arguments *mini* et *maxi*.

  Lors de l'accès à la valeur d'un tel champ (ex : *valeur-minimale* ou *valeur-maximale*) de la description d'une image, si le champ n'a pas de valeur, le moyen d'obtention est déclenché. Une requête initiale associée au but indiqué est alors créée automatiquement par le système et Ocapi est appelé récursivement sur cette requête. Une fois la requête exécutée, le résultat est affecté, pour le champ désiré, mais aussi pour tous les champs qui dépendent de ce même but. On évite ainsi de relancer plusieurs fois Ocapi sur la même requête initiale (même valeur des données en entrée, même but).
  Exemple : lors de l'accès au champ *valeur-minimale* d'une image particulière "image-1" le système réalise automatiquement les étapes suivantes :

  1. création d'une requête *r-req-interne*, avec le but *minmaxmoy* et la pré-contrainte sur l'argument d'entrée *ie = image-1*,
  2. appel à Ocapi *(ocapi r-req-interne '(mini maxi))*, en indiquant les arguments en sortie qui sont intéressants,
  3. exécution de la requête qui renvoie une liste formée des noms des arguments en sortie et de leurs valeurs associées $\rightarrow$ (*mini u maxi v*), puis affectation des valeurs u et v respectivement aux champs *valeur-minimale* et *valeur-maximale* de la description d'implémentation de *image-1*.

- Si le moyen d'obtention est décrit par une forme Lisp, lors de l'accès à la valeur de ce champ, le système évalue la forme Lisp ; cette forme peut faire référence à la valeur d'un autre champ de la description (ex : *taille-x*, *taille-y*, dans le calcul de *nbre-pixels*). La description est associée à la donnée et peut resservir dans d'autres parties du raisonnement travaillant sur la même donnée, les valeurs des champs sont donc calculées une seule fois pour une même donnée.

La partie sémantique n'est pas liée aux formats. L'expert peut définir autant de descriptions sémantiques qu'il le veut pour un type donné, par exemple on peut donner comme descriptions sémantiques associées au type image : image-médicale, image-galaxie, ... La structure d'une description sémantique se rapproche de celle du contexte, par exemple la description sémantique d'une image contenant une galaxie peut faire référence à la nature de la galaxie, ... Les descriptions sémantiques s'utilisent d'ailleurs dans le même esprit que le contexte, par exemple pour tester le nombre de chaines reconnues comme étant associées à des automobiles (cf exemple précédent).

La description attachée à une donnée contient l'ensemble de la description sémantique et d'implémentation. Les données des buts, des opérateurs et des requêtes ont des descriptions associées.

- Au niveau d'un but, l'expert peut préciser le nom de la description sémantique associée aux arguments.

- Au niveau d'un opérateur, l'expert choisit le format des données. Dans la description d'implémentation associée, il peut redéfinir les moyens d'obtention de certains champs de cette description. En effet l'opérateur peut calculer des valeurs auxquelles la description fait référence, après exécution de l'opérateur, ces champs pourront être affectés.

- Au niveau d'une requête initiale, l'utilisateur précise le format de ses données. De plus il peut indiquer les valeurs de champs de la description associée à une donnée (pour les champs de la description sémantique, comme pour celle d'implémentation).

- Au niveau d'une requête interne, l'expert peut fixer la valeur de certains champs de la description sémantique.

Exemple : accès à un champ de description d'implémentation à l'affectation des résultats :

Opérateur :	o-calcul-seuil-his	
But :	statistique-de-densité	
arguments-entrée	ie "*image en entrée*"	
	type : image	
arguments-sortie	sm "*seuil*"	
	type : réel	
	valeur :	(* (argument sm de requete r-hseuil)
		(- (valeur-description 'ie '(valeur-maximale))
		(valeur-description 'ie '(valeur-minimale))))
Décomposition		
	↗ r-histogramme	
Puis		
	↘ r-hseuil	

L'opérateur *o-calcul-seuil-his* calcule le seuil minimal pour une image, à partir de son histogramme. Il prend donc en entrée une image (*ie*) et renvoie le seuil (*sm*). C'est un opérateur complexe. Le calcul du seuil se déroule en deux étapes. Il faut d'abord calculer son histogramme (requête *r-histogramme* qui prend en entrée l'image *ie* et qui renvoie une courbe représentant son histogramme), puis il faut calculer le seuil minimal sur cette courbe (requête *r- hseuil* qui prend en entrée l'histogramme et qui renvoie le seuil minimal). La valeur du résultat de l'opérateur *o-calcul-seuils-his* (*sm*) est ensuite calculée en fonction du seuil minimal et des valeurs minimales et maximales de l'image traitée (champs *valeur-maximale* et *valeur-minimale* de la description de l'image).

2.5 Requêtes

Une requête est une demande faite au système pour résoudre un problème. Elle va permettre de spécifier la nature du problème à résoudre et son environnement, c'est-à-dire les objets sur lesquels elle travaille et la qualité des résultats qu'elle souhaite obtenir. Une requête peut être exprimée par un utilisateur de manière interactive, par un programme (ce qui permet par exemple une utilisation automatique) ou par le système lui-même en cours de travail lors de la décomposition d'un opérateur complexe.

Outre un nom et la désignation du but à atteindre, une requête exprime des contraintes sur les arguments d'entrée (pre-contraintes) et sur les résultats ou arguments de sortie (post-contraintes).

Requête	
nom :	*symbole*
but :	*symbole*
pre-contraintes :	*liste de contraintes*
post-contraintes :	*liste de contraintes*

FIG. 2.14 – *Structure d'une requête avec en italique le type de chaque champ*

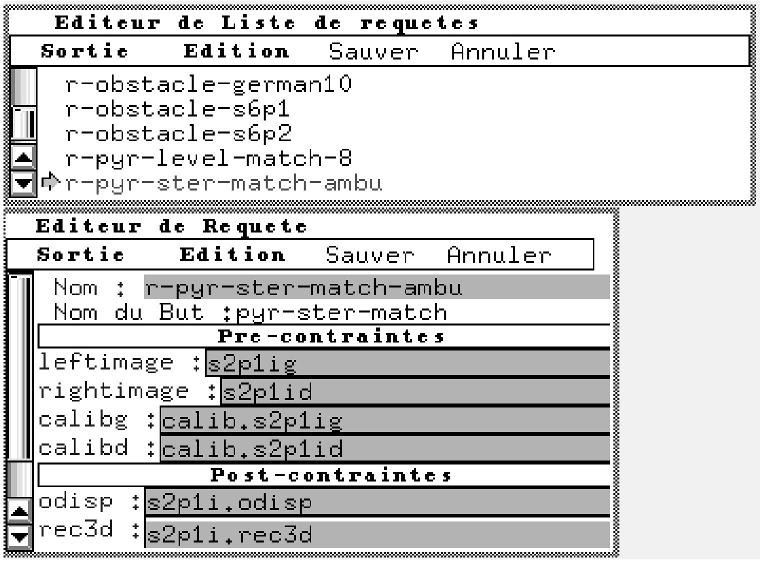

FIG. 2.15 – *Une requête initiale "r-pyr-ster-match-ambu"*

2.5.1 Contraintes sur les arguments d'entrée

Les contraintes sur les arguments d'entrée permettent de préciser des valeurs : par exemple, nommer l'image de travail ou fixer une valeur de seuil. Dans la version actuelle, **seules des affectations de valeurs peuvent avoir lieu**, il n'est pas possible de restreindre autrement les valeurs.

Exemples

seuil de requete r-thr	= .3
x de requete r-ext	= (* 2 (argument ix de operateur o-x))
image-entree de requete r-fil	= "axzd"

2.5.2 Contraintes sur les arguments de sortie

Les post-contraintes sur les résultats ou arguments de sortie permettent d'exprimer des contraintes sur les valeurs de ces arguments, telles que *le nombre de régions doit être positif* ou *le résultat de l'interprétation doit être précis.* **Dans la version actuelle, seules des affectations de valeurs (égalités) peuvent avoir lieu**, il n'est pas possible de restreindre autrement les valeurs.

Exemple

interpretation-res	'precis

2.5.3 Requêtes initiales

Les requêtes initiales sont celles qui sont définies comme points de départ possibles du raisonnement. Ces requêtes permettent d'atteindre une fonctionnalité particulière (un but) pour des données particulières. Ce sont des éléments de la base de faits.

Définir une requête initiale consiste à choisir le but correspondant et à remplir les pre- et postcontraintes. Les pre-contraintes vont indiquer l'association des arguments d'entrée du but avec des données initiales : par exemple dans la figure 2.15, la requête initiale *r-pyr-ster-match-ambu*, l'argument *leftimage* du but *pyr-ster-match* est associé à l'image contenue dans le fichier *s2p1ig* (par défaut le système va chercher ce fichier dans le réperetoire **bf** de la base de faits). Les post-contraintes peuvent indiquer par exemple dans quels fichiers l'utilisateur veut recueillir les résultats finaux.

2.5.4 Requêtes internes

A tout opérateur complexe est associé un arbre de requêtes internes, qui décrit la décomposition en étapes de l'opérateur. Cet arbre se compose de requêtes reliées par des *Et* et des *Puis*. A chacune de ces requêtes, il faut associer un but. Ensuite il faut décrire le flux de données entre les requêtes, grâce aux pre-contraintes, qui permettent d'établir des liens entre les données d'entrée et de sortie des différentes requêtes entre elles, ainsi qu'avec les arguments de l'opérateur complexe ou des buts des requêtes de l'arbre. L'établissement de tels liens est indispensable pour décrire au système comment les données vont transiter. La syntaxe des pre-contraintes dans ce cas est : (argument *nom-argument* de operateur|but|requete *nom*) L'opérateur est forcément l'opérateur complexe appelant, la requête est obligatoirement une de celles de la décomposition et le but est un de ceux associés aux requêtes de la décomposition. Un exemple de requêtes internes est montré figure 2.16.

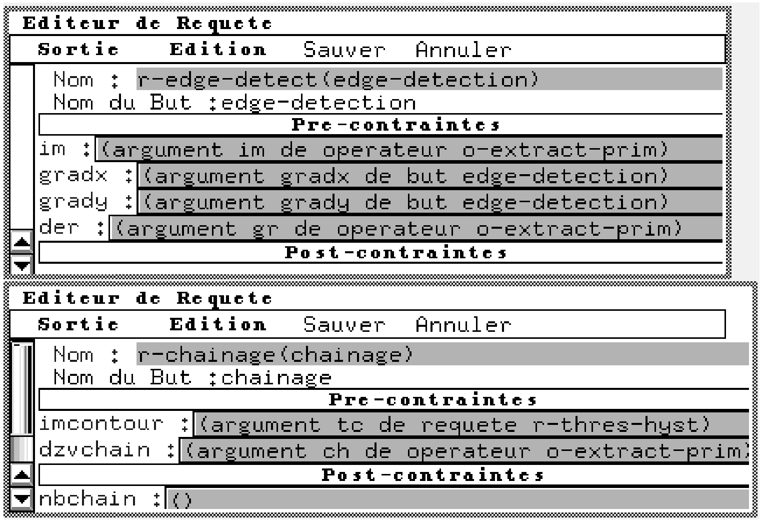

FIG. 2.16 – *Requêtes internes de l'opérateur complexe "o-extract-prim"*

2.6 Contexte

Principalement pour pouvoir choisir entre les méthodes, mais aussi pour savoir comment calculer des valeurs de paramètres ou même évaluer les résultats, il est nécessaire de disposer d'informations globales sur le contexte des données à traiter. Par exemple, dans le cas d'images, le contexte décrira le domaine d'application (robotique, biologie, scènes routières...), les conditions d'acquisition des images (type de capteurs, longueurs d'onde...), les conditions d'observation (en ville, par mauvais temps...), les caractéristiques de l'image (présence de reflets, résolution...), mais des informations plus spécifiques à une application peuvent également être exprimées, il peut même y avoir des informations sémantiques sur le contenu de la scène, comme par exemple : étoiles (présentes absentes)...

Ces informations sont décrites par l'intermédiaire d'objets : un contexte "base de connaissances", créé et décrit par l'expert et des contextes "base de faits" remplis par les utilisateurs, en suivant le schéma fourni par le contexte "base de connaissances" et en fonction de leurs problèmes particuliers.

L'expert définit une structure de contexte dans la base de connaissances qui décrit les informations liées au domaine. Mais pour une même structure de contexte dans la base de connaissances, l'utilisateur peut définir plusieurs contextes dans la base de faits correspondant aux différents cas particuliers à traiter. Il affecte des valeurs aux éléments de contexte, conformément aux domaines prévus dans les champs du contexte base de connaissances. Lors du lancement d'une exécution, il choisira un des contextes de la base de faits.

Un contexte particulier ("base de faits") pourra être décrit entièrement ou partiellement par l'utilisateur avant de lancer le système expert ; il est attaché à la requête initiale de l'utilisateur. Il est utilisé par les différents types de règles. A chaque fois qu'une règle aura besoin d'une information contextuelle, si cette information n'est pas déjà connue, elle sera demandée interactivement à l'utilisateur.

Ces contextes sont structurés en niveaux et ils permettent de définir des informations globales ou locales. Un contexte sera composé d'éléments de contexte qui peuvent être eux-même structurés (à plusieurs niveaux) ou simples. Pour décrire un élément de contexte simple l'expert indique son nom, son type (numérique ou symbolique), son domaine (les valeurs possibles pour un symbolique, les bornes de l'intervalle pour un numérique) et sa caractéristique (est-ce qu'il correspond à une information globale ou locale). Un élément de contexte structuré peut être lui-même composé d'éléments de contexte simples ou structurés. En cours de raisonnement un élément de contexte sera toujours référencé par le chemin y conduisant (exemple de chemin : *'(scène région-observée type)*).

Les modifications, en cours de raisonnement, d'une information globale sont répercutées sur tout l'arbre d'exécution tandis que celles portant sur une information locale (attachée aux requêtes lors de l'exécution) ne sont vues que du *sous-arbre fils*, car elles sont attachées aux sous-requêtes. Ce type d'information ne peut donc pas "remonter" vers un opérateur frère dans l'arbre d'exécution.

Tous les éléments non terminaux sont des éléments de contexte structurés. Par exemple, dans la figure 2.17, *formation-image*, *scene* et *calibrage* sont des éléments

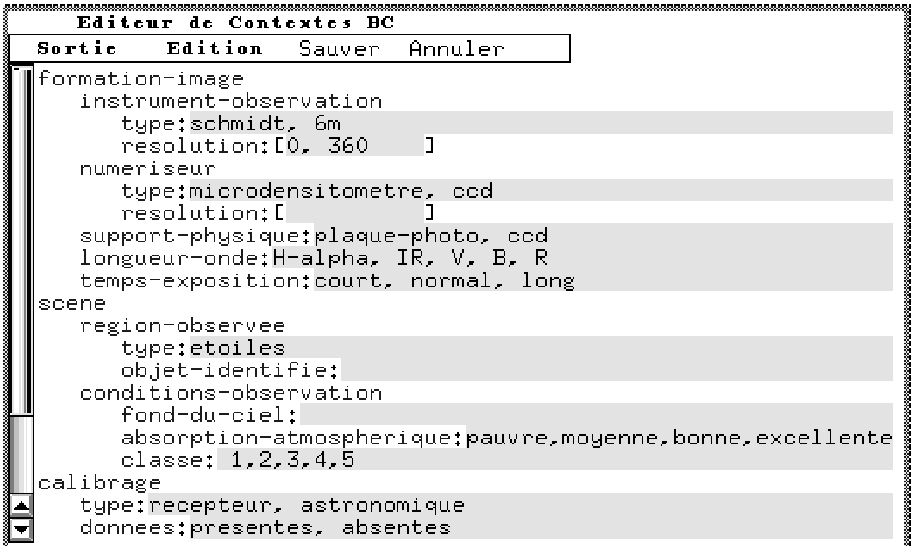

FIG. 2.17 – *Exemple de contexte "base de connaissances" structuré pour le domaine d'application "images astronomiques*

structurés, *temps-exposition* est un élément simple. Un élément de contexte simple possède un domaine, un type et une caractéristique. Par exemple, pour l'élément *type* de *instrument-observation* on a :

type :

domaine	schmidt, 6m,...
type	symbolique
caractéristique	global

Pour les éléments de contexte simple, le domaine et la caractéristique apparaissent. Les éléments de contexte locaux sont indiqués à l'interface par un *(l)* qui précède leur nom, les éléments globaux, les plus courants n'ont pas d'indication particulière (tous les éléements de la figure 2.17 sont globaux).

2.7 Différents types de règles

Ce sont les règles qui vont permettre d'exprimer la connaissance opératoire de l'expert. Cette connaissance peut être de nature inférentielle (plusieurs niveaux d'inférences sont nécessaires pour prendre une décision), mais elle peut simplement faire appel à une connaissance de type algorithmique : par exemple, calcul d'une valeur de paramètres par une formule. De plus, l'expert peut ne pas savoir exprimer sa connaissance sur un problème d'évaluation, par exemple, et souhaiter que cette évaluation soit prise en charge par l'utilisateur du système expert. Ocapi permet de prendre en compte ces différentes considérations.

La base de règles est structurée en petites bases attachées à un but ou à un opérateur. Chaque base est déclenchée indépendamment, à des moments différents, et ce selon sa fonction (choix, évaluation, initialisation, ajustement). L'interprète de règles est un moteur simple en chaînage avant.

Il y a quatre types de règles : règles de choix, d'évaluation, d'initialisation et d'ajustement, les deux premières étant attachées aux buts, les deux dernières aux opérateurs.

Nous précisons ci-dessous tout d'abord les prédicats et fonctions communs à tous les types, utilisables dans les prémisses ou les actions de toutes les règles, puis, pour chaque type de règles, ses prédicats et fonctions spécifiques, ainsi que les types de leurs arguments.

Remarques importantes :

- Si besoin, il est possible d'évaluer n'importe quelle forme Le_Lisp, que ce soit dans les prémisses ou dans les actions des règles.

- Les prémisses et les actions sont implicitement reliées par des *et* logiques entre elles.

- La syntaxe est celle de Le_Lisp (notation préfixée).

- Certaines fonctions, plus spécifiques à un type de règles, seront présentées par la suite avec le type de règle auquelles elles correspondent de façon privilégiée, mais ceci n'implique pas forcément qu'elles ne puissent pas être utilisées dans d'autres types de règles.

- La plupart des arguments de ces fonctions peuvent eux-même être obtenus par application d'autres fonctions. Par exemple si le type indiqué pour un argument est *symbole*, cela signifie qu'il est possible de mettre un symbole comme argument (ex : 'image-galaxies-1, **attention un symbole est toujours quoté en Lisp!**) ou une fonction retournant un symbole (comme (valeur 'image-entree) qui, si image-entree est un nom d'argument, retourne le symbole du nom de fichier qui contient l'image réelle correspondant à cet argument).

2.7.1 Prédicats et fonctions communs à toutes les règles

Accès aux valeurs des arguments

Une fonction d'accès aux valeurs des arguments est offerte pour toutes les règles.

Nom	Arguments
Dans les *prémisses et actions*:	
valeur	symbole

FIG. 2.18 – *Fonction commune d'accès aux valeurs des arguments*

La fonction *(valeur nom-arg)*, où *nom-arg* est un nom d'argument, permet l'accès aux résultats obtenus (c'est-à-dire aux valeurs des arguments à l'issue de l'exécution de la requête).

Prédicats sur les données et paramètres

Ces prédicats ne s'utilisent que dans les *prémisses* des règles, ils prennent tous deux arguments, et rendent tous un booléen.

Nom	Arguments
Dans les *prémisses*:	
<	numérique , numérique
<=	numérique , numérique
>	numérique , numérique
>=	numérique , numérique
=	numérique , numérique

FIG. 2.19 – *Tests d'égalité et d'inégalités de valeurs numériques*

Nom	Arguments
Dans les *prémisses*:	
etre	symbole , symbole
different-de	symbole , symbole
parmi	symbole , liste de symboles
en-dehors-de	symbole , liste de symboles

FIG. 2.20 – *Tests d'égalité/inégalité symboliques et opérations ensemblistes*

Exemples:

```
(= (valeur 'seuil) .27)
(> (+ 0.5 (valeur 'inf)))
(etre (valeur 'nom-image) 'fic1)
(parmi (valeur 'p) '(b1 b2 b3))
```

Prédicats et fonctions sur le contexte

Trois prédicats et fonctions d'utilisation d'un contexte existent. Ils portent sur le contexte "base de faits" en cours d'exécution.

Nom	Arguments
Dans les *prémisses*:	
etre-contexte	liste de symboles , symbole
valeur-contexte	liste de symboles
Dans les *actions*:	
changer-contexte	liste de symboles , expression

FIG. 2.21 – *Prédicats et fonctions sur le contexte*

Les opérations possibles sont les suivantes:

– Test sur les éléments du contexte: *(etre-contexte chemin-elt-contexte valeur)*, teste l'égalité de la valeur de l'élément de contexte référencé par son chemin (ex '(acquisition capteur résolution)), avec la valeur *valeur*; si la valeur de l'élément n'est pas connue, elle est demandée interactivement à l'utilisateur.

- Lecture de la valeur d'un élément du contexte : *(valeur-contexte chemin-elt-contexte)*, retourne la valeur de l'élément référencé par son chemin.

- Affectation d'eléments du contexte : *(changer-contexte chemin-elt-contexte valeur)*, affecte la valeur "valeur" à l'élément de contexte référencé par son chemin. Les valeurs sont des valeurs littérales, des valeurs d'arguments ou des expressions des précédentes.

Les chemins des éléments de contexte sont représentés par des listes (quotées!) de symboles, comme montré en page 32.

Exemples :

Si	*(etre-contexte '(formation-image temps-exposition) 'court)*
Si	*(etre-contexte '(calibrage type) 'recepteur)*
	(etre-contexte '(calibrage donnees) 'presentes)
	(etre-contexte '(formation-image support-physique) 'plaque-photo)

Prédicats et fonctions sur les descriptions des données

Toutes les règles peuvent faire référence aux champs d'une description d'une donnée de leur opérateur ou but associé.

Nom	Arguments
Dans les *prémisses* :	
etre-description	symbole , chemin-elt-description , symbole/nombre/expression
valeur-description	symbole , symbole
Dans les *actions* :	
changer-description	symbole , chemin-elt-description , symbole/nombre/expression
valeur-description	symbole , chemin-elt-description

FIG. 2.22 – *Prédicats et fonctions sur les descriptions*

Les opérations possibles sur les descriptions des données sont :

- *(être-description nom-arg chemin-elt-description valeur)*, compare la valeur du champ de description référencé par son chemin : *chemin-elt-description* de l'argument *nom-arg* avec la valeur passée en paramètre et rend vrai si ces deux valeurs sont égales.

- *(changer-description nom-arg chemin-elt-description valeur)* affecte la valeur passée en paramètre au champ de description de chemin *chemin-elt-description* de l'argument *nom-arg* référencé.

- *(valeur-description nom-arg chemin-elt-description)* renvoie la valeur du champ de description de chemin (*chemin-elt-description*) de l'argument *nom-arg* référencé.

Exemple (règle d'évaluation) :

Si		*(juger-global? 'calibrage 'bon)*
	Alors	*(changer-description 'is '(nature) 'intensite)*

L'opérateur de calibrage prend en entrée une image en densité (champ nature de la description associée à l'image en entrée) et rend en sortie une image (is) en intensité. On affecte donc le champ nature de la description de l'image de sortie (is) à *'intensité.*

Lors d'un accès à la valeur d'un champ de description sémantique d'un argument dans une règle, si la valeur n'existe pas, on la demande interactivement à l'utilisateur. En effet, la description sémantique fait référence à des informations symboliques qui ne peuvent provenir de calculs. Par contre, si on accède à la valeur d'un champ de description d'implémentation et qu'elle n'existe pas, il faut d'abord essayer d'exécuter le moyen d'obtention s'il y en a un, sinon on demande interactivement à l'utilisateur. En effet la description d'implémentation fait référence à des informations qui proviennent la plupart du temps de mesures, elles sont donc calculables.

Fonctions sur les bases de faits locales des opérateurs :

Les opérateurs ont une base de faits locale associée. Celle-ci contient des variables locales et leurs valeurs. Ces variables servent à stocker des valeurs provisoirement au cours d'un raisonnement spécifique (principalement initialisation ou ajustement). La base de faits locale est connue seulement pendant la durée de ce raisonnement spécifique. Il n'y a aucune vérification sur le contenu de cette base de faits locale, qui est gérée comme une simple liste.

Les valeurs positionnées au cours d'un raisonnement (donc dans un type de règles) pourront être accédées dans ces mêmes règles, grâce aux prédicats et fonctions décrits dans la figure 2.23.

Nom	Arguments
Dans les *prémisses* :	
tester-local?	symbole , symbole/nombre
valeur-local	symbole
Dans les *actions* :	
affecter-local	symbole , symbole(s)/nombre(s)
affecter-local-par-utilisateur	symbole , [liste-valeurs-possibles]
raz-base-faits-locale	
valeur-local	symbole

FIG. 2.23 – *Prédicats et fonctions sur les bases de faits locales*

- *(tester-local? var-locale valeur)* teste si la variable locale *var-locale* a pour valeur *valeur*;
- *(valeur-local var-locale)* renvoie la valeur de la variable locale *var-locale*, si *var-locale* n'a pas de valeur, renvoie ().
- *(affecter-local var-locale valeur)* affecte la valeur *valeur* à la variable locale *var-locale* ; on peut aussi écrire *(affecter-local symb/val ... symb/val)* pour stocker une liste de symboles (ou valeurs) quelconques (exemple : *(affecter-local 'il 'va 'falloir 'conserver 'qqch)*).
- *(affecter-local-par-utilisateur var-locale [liste-valeurs-possibles])* demande interactivement à l'utilisateur d'affecter une valeur parmi *liste-valeurs-possibles* à la variable *var-locale*.
- *(raz-base-faits-locale)* "vide" la base de faits locale de l'opérateur courant.

Exemples : A l'initialisation d'un paramètre, on peut vouloir raisonner d'abord symboliquement, puis traduire l'information symbolique en valeurs numériques, on a donc besoin pendant ce raisonnement de variables locales pour ranger les informations intermédiaires (ici *suggestion-heuristique* et *valeur-seuil*), en fin de raisonnement l'ensemble de ces informations servira à initialiser des valeurs de paramètres.

Si		*(etre-contexte '(contraintes-utilisateur details) 'beaucoup)*
	Alors	*(affecter-local 'valeur-seuil 'moyen)*
Si		*(tester-local 'valeur-seuil 'moyen)*
	Alors	*(affecter-local 'suggestion-heuristique 25)*

2.7.2 Règles de choix

Ces règles contiennent la connaissance permettant de choisir entre différentes méthodes ayant toutes le même but. Elles sont attachées au but commun.

Leurs prémisses testent les valeurs des arguments d'entrée de la requête et les valeurs de champs du contexte.

- dans les prémisses, il est aussi possible de tester si une caracteristique fait partie de celles de l'opérateur en cours, par (*caracteristique-op-selectionne caract*), (exemple : Si (caracteristique-op-selectionne 'haute-qualite)).

Nom	Arguments
Dans les *prémisses* :	
caractéristique-op-sélectionné	symbole
Dans les *actions* :	
utiliser-operateur	symbole
refuser-operateur	symbole
utiliser-op-de-caracteristique	symbole
refuser-op-de-caracteristique	symbole

FIG. 2.24 – *Prédicats et fonctions des règles de choix*

Leurs actions choisissent certains opérateurs et/ou refusent l'utilisation d'autres. Les fonctions spécifiques de ces actions sont :

- (*utiliser-operateur nom-op*) et (*refuser-operateur nom-op*), qui prennent en argument le nom d'un opérateur. Tout opérateur auquel on aura appliqué *refuser-operateur* se verra définitivement éliminé pour la résolution de la requête en cours (voir paragraphe 3.1.4 page 55 sur le déclenchement des règles de choix) ;

- et leurs homologues, (*utiliser-operateur-de-caracteristique caract*) et (*refuser-operateur-de-caracteristique caract*), qui prennent en argument le nom d'une caractéristique souhaitée ou non.

Exemples :

Si		*(etre-contexte 'galaxie 'centree)*
	Alors	*(utiliser-operateur 'op-centre)*
Si		*(= (valeur 'parallaxe) 0)*
	Alors	*(utiliser-operateur 'parmbol2)*
Si		*(etre-contexte 'orig_indic_t 'stroemgren)*
	Alors	*(refuser-operateur 'UBV_Teff)*
		(refuser-operateur 'Detail_T)

Exemple : sélection d'un opérateur, basée sur les caractéristiques

But :	stereo-match
Operateur-1:	o-ma-stereo-match
Caracteristiques:	haute-qualite
	adaptabilite-faible
Operateur-2:	o-meygret-stereo-match
Caracteristiques:	qualite-moyenne
	forte-adaptabilite
Règles de choix :	
Si	*(etre-contexte '(contraintes-utilisateur qualite-appariement) 'haute)*
Alors	*(utiliser-operateur-de-caracteristique 'haute-qualite)*

2.7.3 Règles d'évaluation

Elles contiennent la connaissance pour évaluer les résultats de l'exécution d'une requête ; elles sont attachées au but correspondant. Ce sont elles qui sont chargées de la détection de l'échec éventuel et de son diagnostic (nature de l'échec).

Nom	Arguments
Dans les *prémisses*:	
postcontrainte	symbole
postcontrainte-min	symbole
postcontrainte-max	symbole
precontrainte	symbole
precontrainte-min	symbole
precontrainte-max	symbole
Dans les *actions*:	
juger	symbole , symbole
juger-global	symbole , symbole
juger-par-utilisateur	symbole , liste-symboles
juger-global-par-utilisateur	liste-symboles
echec	

FIG. 2.25 – *Prédicats et fonctions des règles d'évaluation*

Prémisses

Ce sont des comparaisons entre ce qui est demandé dans les contraintes de la requête sur les arguments de sortie et ce qui a été effectivement obtenu. Elles peuvent également prendre en compte des valeurs du contexte.

- Les fonctions *(postcontrainte-min nom-argument)* et *(postcontrainte-max nom-argument)* permettent l'accès aux contraintes de la requête sur les arguments de sortie ; elles rendent respectivement la valeur inférieure et la valeur supérieure exprimées dans la requête sur l'argument numérique du but considéré. Pour les paramètres symboliques, c'est le prédicat *(postcontrainte nom-argument)* qui donne la liste des symboles autorisés, '(∗) dénote une absence de contrainte.

- Les fonctions *(precontrainte-min nom-argument)*, *(precontrainte-max nom-argument)* et *(precontrainte nom-argument)* permettent l'accès aux contraintes sur les arguments d'entrée.

Actions

Elles permettent de porter des jugements symboliques après l'exécution. Les jugements sont exprimés sous forme d'une liste de symboles. Il est possible de porter des jugements spécifiques à un paramètre '(*r1 est trop grand*, *is est floue*) ou des jugements globaux sur l'exécution '(*résultats en cours d'amélioration, détection ambigüe).*

- La fonction *(juger nom-argument symbole)* permet de porter un jugement sur une donnée de sortie ; elle prend deux arguments : le premier est un nom de donnée, le second est un jugement concernant exclusivement cet argument. Celui-ci pourra être testé dans les prémisses des règles d'ajustement des opérateurs associés et aussi dans les autres règles d'évaluation du but (raisonnement à plusieurs profondeurs ; voir les derniers exemples de règles d'évaluation) en utilisant le prédicat *juger?*.

- La fonction *(juger-global symbole symbole)* prend deux arguments du type attribut et valeur, sans restrictions particulières. Ceux-ci pourront être testés dans les ajustements des opérateurs associés, par le prédicat *juger-global?*. Elle permet de porter un jugement sur le résultat global de l'opérateur (ex : pour un opérateur qui crée 3 images, on pourra qualifier chacune avec juger et l'ensemble avec juger-global), qui pourra être testé dans les règles d'ajustement par juger-global?

- De plus, il existe des fonctions (*juger-par-utilisateur nom-argument liste-valeurs-possibles*) et (*juger-global-par-utilisateur liste-valeurs-possibles*) qui permettent de demander interactivement à l'utilisateur d'évaluer certains résultats (en lui proposant une liste de réponses possibles).
  La différence entre les fonctions *juger...* et *juger-global...* est que les premières permettent d'attacher des jugements à un argument de sortie, alors que les secondes permettent d'attacher des jugements à un opérateur dans sa globalité.

- La fonction *echec* est d'une autre nature ; elle va permettre, en se basant éventuellement sur les jugements déjà portés, de demander un nouvel essai. Des jugements peuvent donc être portés sans forcément provoquer une nouvelle exécution de l'opérateur, voire l'exécution d'un nouvel opérateur. La fonction *echec* est exécutée pour signaler que le résultat est jugé insuffisant et qu'une exécution doit être relancée.

Exemples de règles d'évaluation

```
Si          (<= (valeur 'xmax) 255)
            (> (valeur 'rayleighmax) 50)
    Alors   (juger-global 'taille-filtre 'trop-grande)
            (echec)

Si          (etre-contexte '(contraintes-utilisateur mode) 'interactif)
    Alors   (juger-global-par-utilisateur 'seuillage '(correct trop-bas trop-haut))
```

```
; taille est un résultat du but en cours de traitement
Si          (< (valeur 'taille) 20)
    Alors   (juger-global 'detection 'ambigue)

Si          (juger? 'taille 'petite)
            (juger-global? 'detection 'ambigue)
    Alors   (echec)
```

```
; utilisation les bases de faits locales
Si          (> (valeur-local 'prims-difference-lr-factor)
               (valeur-local 'prims-difference-factor-max))
            (or (etre-contexte '(contraintes-utilisateur details) 'sans))
                (etre-contexte '(contraintes-utilisateur details) 'peu)))
    Alors   (valeur 'eval-gauche-seuil 'augmenter)
            (valeur 'ajustement-droit-seuil 'addition)
```

2.7.4 Règles d'initialisation

Ces règles expriment une partie de la connaissance servant à initialiser les valeurs des arguments d'entrée des opérateurs, généralement en fonction du contexte et de la requête en cours de traitement. Elles sont attachées à un opérateur. D'autres mécanismes existent tel que le mécanisme des valeurs par défaut.

Les fonctions sur les bases de faits locales des opérateurs s'utilisent de façon privilégiée dans ces règles.

Prémisses

Elles ont accès à la valeur de chacun des arguments (par la fonction commune *valeur* suivie du nom de l'argument). Les prémisses vont également tester le contexte et les valeurs des arguments d'entrée de la requête.

Nom	Arguments
Dans les *actions*:	
initialiser	symbole , symbole/nombre/valeur
initialiser-par-utilisateur	symbole

FIG. 2.26 – *Prédicats et fonctions des règles d'initialisation*

Actions

Les actions disposent de la fonction (*initialiser nom-paramètre valeur*) qui permet de proposer une valeur et de la fonction (*initialiser-par-utilisateur nom-paramètre*) qui demande une valeur interactivement à l'utilisateur. Si plusieurs initialisations ont lieu pour le même paramètre, c'est la dernière proposition qui est prise en compte.

Exemples (*seuilh, taille-fenetre, sy* et *zoom* sont des noms de paramètres) :

Si		*(etre-contexte 'details 'peu-nombreux)*
	Alors	*(initialiser 'seuilh 90)*
Si		*(etre-contexte 'bruit 'fort)*
	Alors	*(initialiser 'taille-fenetre 7)*
Si		...
	Alors	*(initialiser 'sy (valeur 'zoom))*
Si		*(etre-contexte '(contraintes-utilisateur details) 'sans)*
	Alors	*(affecter-local 'valeur-seuil 'haut)*

2.7.5 Règles d'ajustement

Elles permettent d'exprimer la connaissance sur la façon de modifier les valeurs des paramètres d'entrée d'un opérateur en fonction des résultats obtenus par ses exécutions précédentes pour la même requête. Elles sont donc attachées à cet opérateur.

Les fonctions sur les bases de faits locales des opérateurs s'utilisent de façon privilégiée dans ces règles.

Nom	Arguments
Dans les *prémisses*:	
juger?	symbole , symbole
juger-global?	symbole , symbole
valeur-min	symbole , symbole
valeur-max	symbole , symbole
valeurs	symbole , symbole
defaut	symbole
domaine-bc	symbole
domaine-bc-min	symbole
domaine-bc-max	symbole
valeur-max-nb-optimisation	
valeur-max-nb-convergence	
Dans les *actions*:	
ajuster-par	symbole , symbole
%-min	symbole , numérique
%-pas	symbole , numérique
dichotomie-min	symbole , numérique
dichotomie-max	symbole , numérique
dichotomie-pas	symbole , numérique
addition-min	symbole , numérique
addition-max	symbole , numérique
addition-pas	symbole , numérique
augmenter	symbole
diminuer	symbole
changer	symbole , symbole
changer	symbole , symbole
affecter-max-nb-optimisation	numérique
affecter-max-nb-convergence	numérique

FIG. 2.27 – *Prédicats et fonctions des règles d'ajustement*

Prémisses

Les prémisses permettent de tester les jugements émis par les règles d'évaluation, grâce aux deux prédicats *juger?* et *juger-global?*, qui permettent de connaître les jugements qui ont été portés sur l'opérateur pendant la phase d'évaluation de manière globale ou sur un argument particulier. Par exemple *(juger? 'seuil 'trop-strict)* ou *(juger-global? 'detection 'ambigue)*.

Ces jugements peuvent avoir été faits automatiquement par le moteur, par les règles d'évaluation en faisant éventuellement appel à une méthode algorithmique ou interactivement par l'utilisateur.

Les prémisses ont également accès en lecture aux informations concernant les valeurs pouvant être affectées aux différents arguments (valeur par défaut, valeurs possibles...) ainsi qu'aux valeurs des exécutions précédentes de la même requête. Des fonctions permettant de connaître les différentes contraintes sur les arguments sont également disponibles.

Nous donnons ci-dessous la liste des prédicats et fonctions correspondants.

- Accès aux informations provenant de l'évaluation : *juger-global?* et *juger?* (même syntaxe que *juger* et *juger-global*).

- Accès aux informations contenues dans la base de connaissances : la fonction *(defaut nom-arg)*, permet d'accéder à la valeur par défaut d'un argument (d'entrée ou de sortie) d'opérateur ; pour les paramètres d'entrée d'opérateur, les fonctions *(domaine-bc nom-arg)* (pour un paramètre de type symbolique, rend '(*) quand il n'existe pas de valeurs possibles), *(domaine-bc-min nom-arg)* et *(domaine-bc-max nom-arg)* (pour un paramètre de type numérique) permettent d'accéder aux valeurs autorisées de ce paramètre.

- Accès aux informations provenant de la requête : *precontrainte* (pour les symboles), *precontrainte-min* et *precontrainte-max* (pour les numériques), *postcontrainte* (pour les symboles), *postcontrainte-min* et *postcontrainte-max* (pour les numériques, qui ne prennent en compte respectivement que le prédicat $>$ ou $<$) (déja vues page 41).

- Accès aux informations sur les essais en cours : *valeur* (fonction d'accès commune à toutes les règles, donne la dernière valeur utilisée ou valeur précédente d'ordre n), *valeurs* (donne la liste des valeurs déjà utilisées), *valeur-min* et *valeur-max* (pour les numériques, donnent respectivement le minimum et le maximum des valeurs déjà utilisées).

Actions

Elles permettent de donner des consignes pour modifier la valeur des paramètres d'entrée de l'opérateur, pour sa prochaine exécution dans le cadre de la requête considérée. Elles permettent de préciser la méthode à utiliser par la suite, pour faire évoluer les paramètres. De plus, elles peuvent fixer une nouvelle valeur pour certains paramètres, ou bien uniquement préciser si le paramètre considéré doit être augmenté ou diminué. Cela est fait en fonction de la méthode d'ajustement précédemment choisie.

Ces méthodes ne sont pas définitivement fixées. Il est possible de commencer par utiliser une méthode, puis d'en changer. Il est possible également de modifier leurs coefficients. Nous donnons ci-dessous la liste des fonctions permettant d'initialiser les méthodes d'ajustement des arguments d'entrée.

- Le choix de la méthode est effectué à l'aide de la fonction (*ajuster-par nom-methode-ajustement nom-paramètre)*. Actuellement trois méthodes sont disponibles : *dichotomie, pourcentage (%)* (valeur multipliée par un coefficient) ou *addition* (valeur additionnée d'un pas). Par exemple : *(ajuster-par 'dichotomie ref)*, *(ajuster-par '% ref)* ou *(ajuster-par 'addition ref)*. Pour chaque méthode d'ajustement on a les fonctions -pas, -min et -max, qui permettent respectivement, de déterminer le coefficient de la méthode et de fixer les bornes min et max. Ces fonctions peuvent être utilisées par les règles. Dans la version actuelle **il faut donner un min et un max**.
  - Pour la dichotomie : le minimum initial est *(domaine-bc-min nom-paramètre)*; le maximum initial est *(domaine-bc-max nom-paramètre)*. Quand un paramètre est augmenté (resp. diminué), le minimum (resp. le maximum) de la méthode de dichotomie éventuellement associée est mis à jour avec sa valeur précédente.
  - Pour la méthode du pourcentage : le coefficient par défaut est de 10% ; le minimum initial est *(domaine-bc-min nom-paramètre)*, le maximum initial est *(domaine-bc-max nom-paramètre)*.
  - Pour la méthode d'addition : le coefficient par défaut est de 1 ; le minimum initial est *(domaine-bc-min nom-paramètre)*, le maximum initial est *(domaine-bc-max nom-paramètre)*.
- Le choix d'une borne minimum est traité selon la méthode préalablement choisie
  (dichotomie-min nom-paramètre exp), *(%-min nom-paramètre exp)* et
  (addition-min nom-paramètre exp).
- Le choix d'une borne maximum est traité selon la méthode préalablement choisie

(dichotomie-max nom-paramètre exp), *(%-max nom-paramètre exp)* et *(addition-max nom-paramètre exp)*.

- le choix d'un pas ou d'un coefficient de pourcentage s'effectue par appel aux fonctions
  (%-pas nom-paramètre exp), *(dichotomie-pas nom-paramètre exp)* ou *(addition-pas nom-paramètre exp)*.

Trois fonctions permettent de modifier la valeur d'un argument pour le prochain essai :

- l'affectation : changer *(changer nom-paramètre exp)*, fixe une nouvelle valeur *exp* pour l'argument considéré, par exemple (changer 'a 3) ou (changer 'z 'petit) ou
  (changer 'tutu (+ (valeur 'a) 2)) ;
- l'augmentation : (*augmenter nom-paramètre*) et la diminution : (*diminuer nom-paramètre*), les appels *(augmenter nom-paramètre)* et *(diminuer nom-paramètre)* calculent automatiquement une nouvelle valeur en prenant en compte la valeur précédente de l'argument considéré ainsi que la méthode précédemment choisie et paramétrée pour cet argument (pas, min et max).

Le détail des algorithmes utilisés est présenté au paragraphe 3.1.11 page 60.

Exemples

```
; seuil et taille-fenêtre sont des noms de paramètres)
Si          (juger? 'bruit 'trop-fort)
     Alors  (augmenter 'taille-fenetre)

Si          (juger-global? 'detection 'ambigue)
     Alors  (diminuer 'seuil)
            (changer 'option 'avec)
```

Opérateur: o-detection-zone-galaxie		
Si		...
	Alors	*(ajuster-par '% 'smuls)*
		(%-pas 'smuls .05)
		(%-min 'smuls (valeur 'saire2))))
Si		*(juger-global? 'detection 'ambigue)*
	Alors	*(diminuer 'smuls)*

C'est aussi dans ces règles que l'on peut tester ou modifier les nombres de bouclage, grâce à quatre fonctions :

- *(valeur-max-nb-optimisation)* et *(valeur-max-nb-convergence)* : renvoient la valeur du bouclage maximum d'optimisation (resp. de convergence) de l'opérateur courant et
- *(affecter-max-nb-optimisation expr)* et *(affecter-max-nb-convergence expr)* : affectent le bouclage maximum d'optimisation (resp. de convergence) de l'opérateur courant avec une expression numérique Lisp.

2.7.6 Appels récursifs

Il est possible depuis tout type de règle de faire des appels récursifs à Ocapi. Cela est principalement utile pour calculer des valeurs de paramètres en faisant donc appel à une requête secondaire, comme montré ci-dessous où *r-max* est le nom d'une requête permettant de calculer la valeur maximale de l'image d'entrée (*max* est le nom d'une donnée de sortie du but correspondant).

Exemple : règle avec appel récursif à Ocapi :

Requête :	r-max	
But associé :	calcul-maximum	
	Données en sortie :	max

Si		*(etre-contexte 'details 'nombreux)*
	Alors	*(initialiser 'seuil-max*
		(ocapi (trouver-requete 'r-max 'calcul-maximum) 'max))/1.5

La fonction *trouver-requete* qui prend en arguments le nom d'une requête et le nom du but associé, permet de retrouver l'objet requête lorsqu'on n'a que son nom (ce qui est en général le cas).

Comme on l'a vu page 25, on peut aussi avoir des appels récursifs automatiques d'Ocapi dans les moyens d'obtentions des champs de descriptions de données.

2.7.7 Appels séquentiels

Ocapi fournit un moyen de lancer des appels séquentiels, dans les cas où on a à effectuer un traitement répétitif sur toute une séquence de données, comme par exemple une suite d'images consécutives dans le temps. Si le contexte varie très peu d'une donnée à l'autre de la séquence, le plan de traitement à appliquer sur ces données va lui aussi peu varier. Un appel séquentiel permet d'éviter d'avoir à lancer autant de requêtes que de données et d'avoir à recalculer un plan pour chacune.

Dans une requête initiale (uniquement), il est donc possible de donner comme contrainte sur un argument d'entrée, non plus une donnée, mais une *séquence* de données. La planification n'a lieu que pendant le traitement de la première donnée de la séquence et les étapes 1, 2, 3 et 4 de la boucle principale du moteur (voir page 53), n'ont plus lieu à partir de la deuxième donnée. Le moteur va traiter toutes les données de la séquence en les affectant une par une au but correspondant, la requête ne sera marquée traitée que lorsque toutes les données auront été traitées. Pour exprimer une séquence de données en contraintes d'une requête, on peut soit les lister toutes, par exemple :

Pre-contraintes
img : (s8p1l.ext s8p2l.ext)

ou pour une longue séquence, avoir une numérotation des données qui permet de ne fournir que le numéro de la première et de la dernière, par exemple :

Pre-contraintes
img : s8p[1-3]l.ext

Ceci va lancer successivement la requête sur s8p1l.ext, s8p2l.ext, s8p3l.ext.

2.8 Conclusion

Plusieurs modes de représentation des connaissances sont donc proposés pour le développement d'un système expert à l'aide d'Ocapi ; l'expression de la connaissance en est facilitée, l'expert pouvant utiliser la représentation la mieux adaptée pour une connaissance donnée. La connaissance descriptive liée aux traitements et programmes se formalise bien sous forme d'objets structurés. La connaissance déductive concernant le contrôle d'exécution et la prise en compte du contexte est représentée sous forme de règles de production.

La base de connaissances est fortement structurée. La structuration par les objets de type buts et opérateurs, élementaires et complexes, autorisant le raisonnement à différents niveaux d'abstraction, permet aussi le développement de bases de connaissances par plusieurs experts dans des domaines complémentaires. D'autre part, les bases de règles étant spécialisées et réparties entre les différents objets, le temps de recherche des règles applicables est réduit car chaque base de règles est petite : le système gagne donc en efficacité.

Chapitre 3

Moteur et base de faits

Ce chapitre décrit de manière détaillée les composants et le fonctionnement du moteur ainsi que la base de faits.

3.1 Moteur

Le moteur travaille à partir des objets requêtes (en l'occurence une requête ou un arbre *Puis/Et* de requêtes). A partir d'un tel objet, un arbre d'exécution est développé : il comprend des nœuds requêtes (considérées comme des instances de but) et des nœuds instances d'opérateurs. A la requête initiale va être associé l'opérateur choisi pour la résoudre. Si cet opérateur est élémentaire (il correspond à une feuille de l'arbre) il est exécut´ ;, s'il est complexe, il se décompose en un arbre de requêtes sur lequel le traitement est appliqué récursivement.

Les règles sont utilisées pour construire cet arbre, soit dans sa structure en choisissant les opérateurs à exécuter, soit dans ses nœuds en positionnant des valeurs d'arguments. Elles peuvent aussi influer sur le contrôle en prenant par exemple la décision de boucler.

Une règle est elle-même représentée sous forme d'un objet structuré. Une caractéristique importante de ces règles est qu'elles permettent l'accès au monde Le_Lisp.

Lors d'une exécution, Ocapi fait donc appel à différents composants que nous avons présentés dans l'architecture générale : l'interprète de règles, le planificateur, le contrôleur d'exécution et l'interface avec les programmes, chacun utilisant la connaissance appropriée pour mener à bien sa tâche.

3.1.1 Boucle principale

La boucle principale du moteur d'Ocapi, présentée ci-dessous, prend en entrée une seule requête ou un arbre *Puis/Et* de requêtes.

-1- mise en correspondance globale

tant qu'il reste des requêtes non traitées

-2- choix d'une requête non traitée

-3- déclenchement des règles de choix

tant que la requête n'est pas satisfaite

-4- choix effectif de l'opérateur à exécuter

-5- paramétrisation de l'opérateur (avec déclenchement des règles d'initialisation ou d'ajustement)

-6- exécution de l'opérateur

-7- évaluation des résultats obtenus (déclenchement des règles d'évaluation)

L'étape 1 cherche dans la base de connaissances le but référencé pour chaque requête de l'arbre de décomposition, (cette étape devrait aussi vérifier la conformité de type et de format entre les arguments *donnée* fournis par une requête et ceux de chaque opérateur ayant la même fonction du but, dans le cas négatif, l'opérateur devrait être exclu). Au cas où aucun opérateur n'est disponible, le moteur annonce l'échec total du contrôle et rend la main à l'utilisateur pour modifier la base de connaissances, ou abandonner!

L'étape 2 prend une requête non traitée selon l'ordre exprimé dans la base de connaissances, la hiérarchie de l'arbre de requêtes à traiter est parcourue en profondeur d'abord.

Lorsqu'il faut déterminer un opérateur parmi plusieurs pour satisfaire une requête en fonction du contexte, le pilote déclenche les règles de choix attachées au but référencé (l'étape 3). On parle ici de *planification locale*. La *planification*, consiste donc à choisir un opérateur approprié pour résoudre une requête. Les opérateurs sont classés et le classement évolue lors des remises en cause éventuelles (replanification)

A l'étape 4 le moteur sélectionne un opérateur à exécuter, en fonction du classement établi. Ce classement fait intervenir les nombres de convergence (*max-nb-convergence*), pour examiner les opérateurs dans un premier temps. Eventuellement, si aucun opérateur n'a pu être choisi, le moteur réexamine les opérateurs en approfondissant le bouclage. Cette nouvelle phase tient compte des nombres d'optimisation : *max-nb-optimisation* (cf page 56).

Dès que l'opérateur est choisi, on cherche les valeurs initiales des arguments qui sont nécessaires pour l'exécution. Ceci est fait par le sous-module *paramétreur* du contrôleur d'exécution. Si c'est la première fois que l'opérateur est choisi, le paramétreur tient compte des valeurs de la requête appelante, celles proposées dans les règles d'initialisation, celles par défaut, ..., ou même demande interactivement à l'utilisateur lorsqu'un argument n'a toujours pas de valeur. Dans le cas de bouclage, le mécanisme de *reparamétrisation* intervient. Les règles d'ajustement sont alors déclenchées (étape 5).

Le moteur exécute l'opérateur paramétré à l'étape 6. Si l'opérateur est primitif, le programme qu'il décrit est lancé en exécution. Si l'opérateur est complexe, l'exécution est en fait un appel récursif au pilote, i.e., on revient à l'étape 1 avec un arbre de requêtes qui est la décomposition de cet opérateur.

Après l'exécution, le résultat est évalué (étape 7), automatiquement ou interactivement.

Les étapes 1, 3 et 4 sont réalisées par le planificateur : la phase de mise en correspondance globale vérifie qu'il existe au moins un opérateur susceptible de répondre à chacune des requêtes . La phase 3 permet de générer un plan en classant les opérateurs disponibles ; la phase 4 choisit effectivement le prochain opérateur à exécuter.

Avant l'exécution de l'opérateur, il est nécessaire de déterminer les valeurs de ses paramètres d'entrée : c'est le rôle du sous-module paramétreur du contrôleur, qui traite donc la phase 5 différemment selon qu'il s'agit d'une initialisation ou d'un ajustement.

Pour l'étape 6, s'il s'agit d'un opérateur élémentaire, elle est réalisée par l'interface avec les programmes ; s'il s'agit d'un opérateur complexe, le contrôleur est appelé récursivement sur la décomposition de cet opérateur.

Le choix de la prochaine requête (étape 2), l'évaluation des résultats (étape 7) et le bouclage sont réalisés directement par le contrôleur.

Nous allons maintenant détailler chacune de ces étapes.

3.1.2 Mise en correspondance globale

C'est une fonctionnalité globale du planificateur : étant donné un arbre de requêtes, une présélection est faite sur les opérateurs pour chacune des requêtes. L'idée est d'éviter de commencer l'exécution d'un traitement alors qu'il est possible de savoir que les moyens de l'exécuter entièrement ne sont pas disponibles.

Cette phase est globale : elle a lieu sur l'arbre entier sans exécution des requêtes.

Elle est basée sur une simple correspondance entre fonctionnalités. Pour chaque requête, tous les opérateurs pouvant résoudre le but correspondant sont recherchés (c'est-à-dire les opérateurs de même fonctionnalité que le but exprimé dans la requête). Si pour au moins une requête il n'existe pas de tel opérateur dans la base de connaissances, la requête de niveau supérieur ne pourra pas être résolue avec la connaissance disponible. Cette phase a lieu sur chaque décomposition sous forme d'arbre de requêtes d'un opérateur complexe.

3.1.3 Choix de la prochaine requête à traiter

Le choix de la prochaine requête est fait selon la hiérarchie de l'arbre des requêtes à traiter, parcourue en profondeur d'abord. Les liaisons *Puis* et *Et* sont traitées de la même manière. Nous avons voulu donner à l'expert les moyens de différencier les liaisons en fonction de sa connaissance ; mais vu les types de machines actuellement à notre disposition, l'exécution a toujours lieu en séquence.

3.1.4 Déclenchement des règles de choix

Les fonctions spécifiques aux actions des règles de choix sont *utiliser-operateur[-de-carateristique]* et *refuser-operateur[-de-carateristique]*. A l'exécution de chacune de ces fonctions, il y a mise à jour du classement de l'opérateur donné, qui comptabilise le nombre de fois où l'utilisation de l'opérateur a été demandée et le nombre de fois où son utilisation a été refusée, pour la requête courante.

On distingue, parmi les règles de choix, des règles de replanification : celles-ci se caractérisent par le fait qu'elles possèdent au moins une prémisse faisant référence à un prédicat de jugement, à un prédicat portant sur la description d'un argument en sortie ou aux caractéristiques du dernier opérateur exécuté; en effet, elles sont utilisées après au moins une exécution d'opérateur.

Nous présentons d'abord la gestion des règles de choix puis les principes de replanification.

Après l'exécution des règles de choix, un classement des opérateurs disponibles a lieu. Le classement sur les opérateurs utilise quatre listes : la liste des caractéristiques voulues, la liste des caractéristiques refusées, la liste des opérateurs à utiliser (ainsi que le nombre de fois où un opérateur a été demandé) et la liste des opérateurs à refuser. Ces quatre listes sont remplies par les quatre fonctions précédemment citées.

A partir de ces listes, on peut définir pour chaque opérateur, le nombre de fois où il a été refusé ou voulu, et le nombre de caractéristiques désirées ou refusées (que nous appelons les non-caractéristiques) qu'il possède. Ces résultats permettent de classer les opérateurs. Un opérateur marqué à-refuser est définitivement éliminé pour la requête courante. Les autres marques n'ont cependant pas toutes le même poids. Le score global de chaque opérateur se calcule en pondérant les marques (la marque "à-utiliser" est plus forte que "non-caractéristiques", qui est plus forte que "caractéristiques"). On obtient ainsi un score global par opérateur. L'opérateur de plus fort score est choisi. Lorsque deux opérateurs ont le même score global on les trie par ordre décroissant selon leur nombre de marques "non-caractéristiques". Les opérateurs pour lesquels aucune fonction n'a été exécutée sont donc en queue de liste avec un score de 0 ; ils sont rangés dans leur ordre de saisie dans la base de connaissances. Les opérateurs non-éliminés sont alors attachés à la requête sous forme d'un nœud *Ou* : ils seront autant de possibilités de retour pour le pilote en cas d'échec.

La replanification intervient lorsque l'opérateur sélectionné a été exécuté un certain nombre de fois. Au cours d'une étape de mise à jour du classement des opérateurs, lorsque l'opérateur choisi a atteint son nombre maximal de convergence (pour la première phase) ou son nombre maximal d'optimisation (pour la seconde phase) il faut lancer les règles de replanification. Lors de la première phase, lorsque l'opérateur a atteint son nombre maximum de convergence, il est marqué "a-échoué" et provisoirement mis de côté. Lors de la seconde phase, lorsque l'opérateur atteint son nombre maximum d'optimisation, il est marqué "éliminé" et définitivement éliminé pour la requête courante.

3.1.5 Choix de l'opérateur à exécuter

Ce choix est effectué par le planificateur, en mode local, qui travaille sur une requête, les requêtes en amont ayant été exécutées. Le planificateur choisit le prochain opérateur à exécuter parmi ceux du nœud *Ou*, précédemment construit. Le choix de l'opérateur se base d'une part sur les résultats des règles de choix (le classement), d'autre part sur l'information de bouclage.

La sélection du meilleur opérateur se fait en deux phases. La première consiste à examiner tous les opérateurs pour évaluer leur convergence (utilisation du nombre *max-nb-convergence*), la deuxième permet de réexaminer un opérateur en approfondissant le bouclage (utilisation du nombre *max-nb-optimisation*).

- Lors du premier passage, l'opérateur de plus fort score est sélectionné, selon le tri présenté au paragraphe précédent, c'est le premier opérateur du nœud *Ou*. Si aucun n'a été sélectionné explicitement par l'utilisation d'une fonction *utiliser-operateur[-de-carateristique]*, ce sera un opérateur de score 0, en l'occurence le premier saisi dans la base de connaissances; il faut donc remarquer l'importance que peut prendre l'ordre dans lequel les opérateurs associés à un même but ont été saisis.

- Lors des autres passages, c'est-à-dire quand l'évaluation a provoqué un échec et donc a demandé un bouclage et éventuellement a entrainé l'exécution des règles de replanification (mise à jour du score global) :

  - on choisit l'opérateur "non-échoué" (parmi ceux qui n'ont pas atteint leur nombre maximum de convergence) de plus fort score, ; s'il y en a plusieurs qui ont le même score on choisit celui qui a le moins de caractéristiques refusées;
  - s'il n'y a plus d'opérateur "non-échoué" on choisit celui de plus fort score parmi les "échoués" (ceux qui ont été lancé une fois, ont atteint leur nombre maximum de convergence mais n'ont pas atteint leur nombre maximum d'optimisation); s'il y en a plusieurs qui ont le même score on choisit celui qui a le moins de caractéristiques refusées.

Une erreur est déclenchée lorsque tous les opérateurs sont marqués "éliminés" c.a.d. si tous ont échoué pour cette requête.

3.1.6 Paramétrisation

Pendant cette étape le moteur affecte une valeur à chacun des paramètres de l'opérateur. Il travaille sur le prochain opérateur à exécuter (celui qui vient d'être choisi), utilise la requête associée, et éventuellement les causes de l'échec de la précédente exécution. Cette étape retourne un opérateur avec une valeur pour chacun de ses arguments. Le choix d'une valeur de paramètre a lieu en phase d'exécution, tout l'environnement est donc connu. La paramétrisation peut être appelée soit en phase d'initialisation, soit en phase d'ajustement.

3.1.7 Initialisation

Le but de cette étape est de fournir une valeur pour chacun des arguments d'entrée de l'opérateur. L'initialisation a lieu en prenant en compte : les contraintes de la requête en cours de traitement, les valeurs possibles et les défauts des arguments d'entrée de l'opérateur et les règles d'initialisation de l'opérateur.

L'initialisation a lieu par étapes successives. Les contraintes sur les arguments d'entrée de la requête sont d'abord prises en compte, c'est-à-dire que les valeurs des arguments de la requête sont affectées aux arguments correspondants de l'opérateur. Les contraintes des requêtes sont prioritaires.

Ensuite, s'il existe des règles d'initialisation, elles sont exécutées. Elles fournissent une valeur pour certains des paramètres de l'opérateur. Seules les valeurs trouvées pour les paramètres qui n'avaient pas de valeur précédemment sont prises en compte.

Puis, pour tous les paramètres sans valeur, c'est leur valeur par défaut, si elle existe, qui est choisie.

Puis, pour tous les paramètres encore sans valeur, mais ayant un domaine fourni par l'expert, le système procède selon leur type : s'il est symbolique, la première des valeurs possibles est choisie, s'il est numérique, c'est la demi-somme des bornes des valeurs possibles.

Enfin, pour chaque paramètre qui n'a toujours pas de valeur (cas où l'expert n'a fourni aucune indication sur le paramètre), le système demande une valeur à l'utilisateur.

3.1.8 Exécution

La phase d'exécution effective est différente selon qu'il s'agit d'un opérateur complexe ou élémentaire. Pour un opérateur complexe qui a donc une décomposition sous forme de requêtes, le pilote est appelé récursivement sur l'arbre de requêtes de la décomposition.

Pour un opérateur élémentaire, l'exécution est le lancement effectif du programme correspondant, après liaison des arguments de l'opérateur, pour lequel on procède de la manière suivante.

- Si le langage est *Lisp, C* ou *F77*: la syntaxe d'appel est alors exprimée selon la syntaxe Le_Lisp (cf Manuel Le_Lisp les fonctions de chargement de modules objets *cload* et de définition de fonction externe *defextern*).

- Si le langage est *shell*: la syntaxe d'appel contient alors un appel au langage d'exploitation tel qu'il peut être exécuté par la fonction Le_Lisp *comline*. Il

y a création automatique d'une fonction Le_Lisp. Par exemple, dans le cas de l'opérateur filtre-médian dont la syntaxe d'appel est : *med '-w taille-fenêtre image-entrée image-filtrée* la fonction suivante sera créée :
(de med (x y z r) (comline (catenate "med" x " " y " " z " " r)))

Cette fonction a quatre arguments : à l'exécution, *x* prendra toujours la valeur *-w*,
y, z et *r* prenant respectivement les valeurs des arguments *taille-fenêtre, image-entrée* et *image-filtrée.*

C'est ensuite l'interface avec les données qui va permettre d'accèder aux résultats éventuellement stockés dans des fichiers, des variables ou tout simplement au retour fonctionnel de l'exécution (champ *accès* des arguments de sortie des opérateurs, voir page 21).

3.1.9 Evaluation des résultats

L'évaluation des résultats est la prise en compte des règles d'évaluation. Dans le cas où la fonction *echec* n'a pas été utilisée dans ces règles, la requête sera marquée *satisfaite.*

Il y a déclenchement des règles d'évaluation du but associé qui vont permettre de porter des jugements plus sémantiques et qui peuvent par exemple déclencher des algorithmes ou demander l'avis de l'utilisateur. Ceci a lieu au niveau du but car c'est à ce niveau que la connaissance permettant de juger si les résultats sont satisfaisants ou non est disponible. Ces règles effectuent une mise à jour du champ *jugement* de l'opérateur en utilisant les fonctions *juger* et *juger-global.*

3.1.10 Bouclage

Après évaluation des résultats, si la requête en cours n'est pas marquée *satisfaite* (c'est le cas quand le prédicat *echec* a été appelé pendant la précédente phase d'évaluation), un nouvel essai va être lancé sur cette requête. Les étapes suivantes vont donc de nouveau avoir lieu sur cette requête :

- choix effectif de l'opérateur à exécuter,
- paramétrisation de l'opérateur,
- exécution de l'opérateur,
- évaluation des résultats obtenus.

C'est le mécanisme de choix du prochain opérateur à exécuter qui déterminera si c'est le même opérateur que lors de l'essai précédent ou un autre opérateur qui va être exécuté.

Un nouveau paramétrage de l'opérateur choisi va avoir lieu : en utilisant les règles d'ajustement si il s'agit d'un opérateur déjà essayé, en utilisant les règles d'initialisation sinon.

En ce qui concerne l'exécution, comme de nouvelles informations ont été calculées ou un nouvel opérateur a été choisi, les choix qui seront faits dans le nouvel essai pourront être différents de ceux de l'essai précédent ; cela est vrai non seulement au niveau des valeurs de paramètres mais également dans la structure du traitement.
Dans le cas du choix d'un nouvel opérateur, prenons l'exemple d'un premier essai à l'aide d'un opérateur élémentaire et d'un essai suivant à l'aide d'un opérateur complexe : dans le premier cas (opérateur élémentaire) le traitement exécuté se résume au programme correspondant ; dans le second cas (opérateur complexe) un arbre de traitement va être développé.
Dans le cas d'un bouclage sur un même opérateur complexe, les règles de choix d'un sous-but pourront choisir l'utilisation d'une méthode différente de celle utilisée lors de l'essai précédent.

Après cette nouvelle exécution, une phase d'évaluation des nouveaux résultats a lieu.

3.1.11 Ajustement

Le but de cette étape est de proposer une valeur pour chacun des paramètres de l'opérateur sur lequel un bouclage est effectué. Cette valeur peut être une nouvelle valeur ; ce peut être aussi la même que lors de la précédente exécution dans le cas où plusieurs paramètres pouvant être modifiés, un seul l'est effectivement. L'ajustement prend en compte les valeurs précédentes des paramètres, les jugements de l'évaluation, c'est-à-dire les caractéristiques de l'échec, et les règles d'ajustement de l'opérateur.

Ce sont les prédicats *juger?* et *juger-global?* qui vont permettre de tester les jugements émis lors de la phase d'évaluation précédente.

Les fonctions *augmenter, diminuer* et *changer* permettent d'ajuster les paramètres en tenant compte de l'éventuelle méthode d'ajustement qui leur a été associée. Nous précisons ici leur fonctionnement. On note : *MIN* la fonction minimum et MAX la fonction maximum, min et max représentent les bornes minimum et maximum admissibles pour un paramètre, exp représente une expression Lisp.

- *(changer paramètre exp)* : impose une nouvelle valeur (s'applique aussi à des arguments symboliques)
- *(augmenter paramètre)* : type numérique

  La méthode et ses paramètres ont déjà été fixés.

  - cas de la méthode de dichotomie :
    min = valeur précédente de l'argument
    nouvelle valeur pour l'argument =
    MIN(max, min + (max − min) ∗ 1/2).
    la borne max n'étant pas modifiée.
  - cas de la méthode de pourcentage :
    nouvelle valeur pour l'argument =
    MIN(max, valeur-précédente de l'argument ∗ (1 + coeff)).
  - cas de la méthode d'addition :
    nouvelle valeur pour l'argument =
    MIN(max, valeur-précédente de l'argument + pas).
- *(diminuer paramètre)* : type numérique

  La méthode et ses paramètres ont déjà été fixés.

  - cas de la méthode de dichotomie :
    max = valeur précédente de l'argument
    nouvelle valeur pour l'argument =
    MAX(min, max − (max − min) ∗ 1/2),
    la borne min n'étant pas modifiée.
  - cas de la méthode de pourcentage :
    nouvelle valeur pour l'argument =
    MAX(min, valeur-précédente de l'argument ∗ (1 − coeff)).
  - cas de la méthode d'addition :
    nouvelle valeur pour l'argument =
    MAX(min, valeur-précédente de l'argument − pas).

L'ajustement n'a lieu qu'en une seule phase : c'est pendant l'exécution de ces fonctions que la valeur du paramètre est éventuellement modifiée. Si, pour un paramètre donné, aucune règle d'ajustement n'a demandé de modification de sa valeur, celui-ci se voit affecter la même valeur que lors de la précédente exécution. Le stockage des valeurs précédentes a également lieu pendant cette phase ; seules les valeurs modifiées sont conservées.

3.2 Base de faits

La base de faits est constituée de plusieurs composants qui peuvent être eux-mêmes structurés. La structuration de la base de faits est nécessitée par la nature extrêmement variée des informations manipulées par le système expert. Elles sont de différents niveaux d'abstraction, utilisent des représentations diverses, sont manipulées par plusieurs sous-systèmes (algorithmes spécialisés, règles). Cela facilite également la coopération de systèmes experts développés à l'aide d'Ocapi avec d'autres systèmes.

Les *faits* sont ceux *donnés par l'utilisateur*, que ce soit au lancement du système expert ou au cours de son raisonnement, et ceux *déduits par le système*. Les faits donnés en entrée par l'utilisateur sont : la requête initiale soumise au système et également la description du contexte des images à traiter.

Le contexte initial, dont la structure (ensemble des champs, ensemble des valeurs possibles pour chacun de ses champs) est définie par l'expert, est une composante de la base de faits. Il contient l'*information globale* sur le traitement en cours ou la donnée à traiter (telle que, pour une image : le domaine d'application, les conditions de prise de vue ou même des informations sémantiques sur le contenu de l'image (présence de nuages...)). Ces informations sont dépendantes de la donnée à traiter et connues du système soit en entrée, soit en cours de raisonnement par déclenchement de programmes ou par demande interactive à l'utilisateur.

La base de faits comprend aussi l'*arbre d'exécution*, structure créée en cours de raisonnement. Il contient l'information dépendante des traitements choisis et précédemment exécutés : par exemple, les arguments d'entrée et de sortie des traitements. L'historique des différents essais y est également présent, que ce soient les valeurs ou les structures (pour deux essais différents d'un même opérateur complexe, des règles de choix à un niveau inférieur peuvent faire des choix différents, ce qui entraîne des structures de traitement différentes).

L'expert a accès à certaines de ses informations en consultation et/ou modification par les prédicats des règles : par exemple, valeurs d'arguments, historiques des valeurs de paramètres, jugements sur un résultat ou un opérateur. Par l'interface graphique, l'utilisateur a également accès à ces informations.

Les *fichiers* sont les données stockées à l'extérieur du système expert. Ils peuvent contenir les images, histogrammes, courbes, mesures... Ce sont des données d'entrée, des résultats intermédiaires et des résultats finaux. Ils sont le prolongement direct de la base de faits et permettent de gagner de la place en mémoire et de faciliter leur accès depuis n'importe quel langage (Le_Lisp ou autre) ; c'est la base de faits externe. Les mesures ou autres précisions peuvent être ramenées à l'intérieur du système expert par l'interface de données. La structure des fichiers image est directement liée aux algorithmes qui les manipulent.

Bien que le but d'Ocapi soit de construire des systèmes automatiques, il y a des cas où, au cours de son développement par exemple, un système expert incomplet peut manquer d'informations : par exemple, des valeurs par défaut ou des méthodes d'ajustement de paramètres n'ont pas encore été intégrées. Dans ce cas, si le système ne dispose pas d'autres moyens pour trouver une valeur de paramètre, il va demander interactivement à l'utilisateur de lui fournir une valeur, et ce afin de poursuivre son raisonnement. Ce mécanisme peut être utilisé en cours de développement ou de mise au point d'un système dont la version finale est destinée à fonctionner automatiquement. Il est également possible à l'expert de demander explicitement l'aide de l'utilisateur en utilisant les fonctions *juger-par-utilisateur* et *initialiser-par-utilisateur*. D'autre part, si l'utilisateur n'a pas fourni en entrée au système expert les données contextuelles nécessaires, celui-ci les lui demandera interactivement uniquement en cas de besoin. Les mécanismes de demande d'informations à l'utilisateur sont en général utilisés pour les étapes d'évaluation des résultats, pour lesquelles la connaissance est très difficile à exprimer.

Chapitre 4

Méthodologie d'utilisation

Nous proposons ici une méthodologie simple, issue de la pratique du système, pour construire une base de connaissance pour Ocapi.

4.1 Définition de sous-problèmes

Quand on construit une base de connaissances pour résoudre un problème complexe, il est recommandé d'essayer de diviser ce problème en sous-problèmes. Chaque sous-problème correspond ainsi à un sous-arbre de l'arbre de décomposition des opérateurs (qui correspond au problème complet). Ceci permet de se focaliser sur une sous-partie réduite du problème total.

Il est important ensuite de décrire le flot d'informations entre ces sous-problèmes, à l'aide des concepts fournis par Ocapi. A chaque sous-problème, on peut associer un but et définir ses entrées (quelles sont les informations et d'où viennent elles ?) et ses sorties (quelles sont les informations produites et où vont elles ?).

4.2 Construction de sous-problèmes

Les sous-problèmes individuels peuvent se construire en trois étapes. Pendant la première étape on définit la partie la plus abstraite du sous-problème (un but et ses opérateurs), dans la seconde, on construit l'arbre de décomposition et enfin on ajoute les descriptions de données et les règles à cet arbre.

4.2.1 Définition abstraite des blocs de connaissances

La première étape consiste à définir les buts et les opérateurs qui leur correspondent. On peut considérer l'ensemble constitué d'un but et de ses opérateurs comme un "bloc" de connaissances nécessaires pour résoudre un sous-problème. C'est le niveau de spécification le plus abstrait pour un sous-problème : aucune requête n'est encore spécifiée, les opérateurs eux-même ne sont pas complètement spécifiés et aucune règle n'existe encore. Les arguments et paramètres des buts et des opérateurs sont seulement spécifiés localement, mais aucun flot de données n'est encore défini.

Après cette première phase de construction on obtient un ensemble de blocs de connaissances, qui seront utilisables comme briques de base pour l'arbre de décomposition d'un opérateur. Pour définir tous les blocs de connaissances d'un sous-problème, il faut commencer par ceux qui sont aux plus bas niveaux, ou *blocs primitifs.* Ces blocs de connaissances contiennent des opérateurs élémentaires ou primitifs, qui sont les véritables opérateurs de traitement (programmes de traitement d'image par exemple). Après avoir défini les blocs primitifs, on définit les *blocs composés.* On travaille ainsi de manière ascendante. On commence par construire les blocs de connaissances composés qui sont basés sur les blocs primitifs. Puis on définit les blocs de connaissances plus généraux. Et enfin on peut définir les blocs de connaissances de plus haut niveau.

En fait, la façon de générer les blocs de connaissances n'est pas strictement ascendante. Si un sous-problème réalise une action très complexe, il est plus pratique de construire d'abord quelques blocs de connaissances de haut niveau et de raffiner ensuite. Mais il faut faire attention, si on commence par construire les blocs les plus généraux, il n'est pas sûr qu'on puisse finalement les exprimer en terme des primitives de traitement qui existent. C'est pourquoi il faut au moins commencer par la définition des blocs primitifs.

4.2.2 Arbre de décomposition d'opérateurs

Dans une seconde phase, il faut d'abord

(2a) décrire le flot de données entre les blocs de connaissances;

(2b) ensuite, on peut implémenter cela en ajoutant les différentes requêtes aux opérateurs complexes et en associant chaque requête avec un but existant. Il faut là aussi travailler de manière ascendante.

Donc, en fait, on connecte les différents blocs les uns aux autres et on construit un arbre de décomposition d'opérateurs du bas vers le haut (cet ordre est pratiquement obligatoire pour qu'Ocapi fonctionne correctement).

Pour chaque requête il faut décider quelles sont ses *précontraintes*. Dans une précontrainte d'une requête on établit la connexion entre un argument d'un bloc de connaissances et un argument d'un autre bloc. Puis on connecte les entrées/sorties des opérateurs complexes, dont on construit l'arbre de décomposition, avec les entrées/sorties correspondantes des sous-requêtes de cet arbre de décomposition. Par exemple, dans le cas montré figure 4.1, on connecte l'entrée de l'opérateur complexe avec l'entrée de la première sous-requête et la sortie de la dernière sous-requête avec la sortie de l'opérateur complexe.

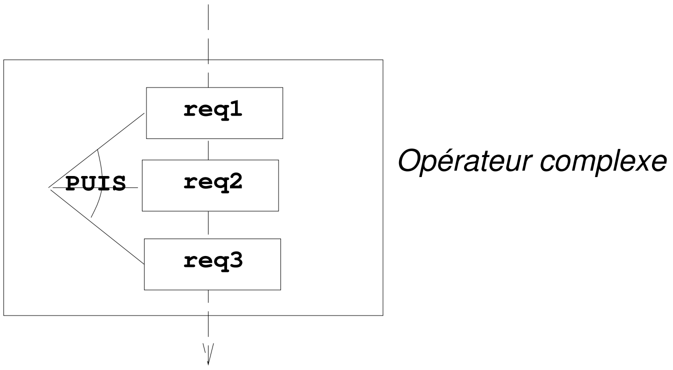

FIG. 4.1 – *Connexion des entrées/sorties dans un arbre de décomposition*

Donc dans cette phase on assemble les différents blocs de connaissances ensemble en les connectant par des requêtes. On a maintenant réellement construit un arbre de décomposition d'opérateur. Le niveau d'abstraction est plus bas. Les requêtes forcent une certaine position des buts et des opérateurs dans l'arbre et les arguments sont complètement déterminés.

4.2.3 Descriptions des données et règles

Jusqu'à présent on a simplement construit un plan sous forme de script, pour l'exécution de plusieurs opérateurs primitifs. Dans la troisième phase on ajoute les différentes règles de production aux buts et aux opérateurs. Cette phase est la plus complexe. Il s'agit de définir les bonnes règles de choix, d'initialisation, d'ajustement et d'évaluation. Beaucoup de conditions et de conclusions de ces règles dépendent des valeurs du contexte et des descriptions de données. Donc, tout en définissant les règles on peut aussi définir le contexte et les descriptions de données.

- D'abord les *règles de choix* doivent permettre de choisir un opérateur parmi plusieurs, en fonction des valeurs de certains champs du contexte et des descriptions de données d'entrées. Ces règles peuvent aussi être utilisées pour exprimer des preconditions sur les données d'entrée.

- Les *règles d'initialisation*, quant à elles sont parfois très délicates à définir. Le meilleur moyen est de commencer par définir les règles qui raisonnent avec des *valeurs symboliques*. Il est par exemple plus facile de raisonner en termes d'une "grande" ou "petite" taille de filtre, plutôt que d'utiliser des nombres réels. Ensuite lorsque les valeurs symboliques sont définies, on les traduit sous forme de nombres réels.Pour cette traduction, il est souvent nécessaire de faire des tests sur le bloc en cours de définition en utilisant différents arguments d'entrée.

  Exemple d'initialisation des paramètres d'un opérateur, en deux étapes de raisonnement :

  1) Raisonnement avec une information symbolique

     Operateur : o-thres-hyst
     Regles d'initialisation :
     si (etre-contexte '(contraintes-utilisateur details) 'peu)
     alors (initialiser 'seuil 'haut)

  2) Traduction de l'information symbolique en nombres

     ◦ Recueil d'expertise
       Interrogation de spécialistes dans l'utilisation de cet opérateur.
       si (etre (valeur 'seuil) 'moyen)
       alors (initialiser 'valeur-heuristique 25)

     ◦ Tests
       Obtention d'une valeur à partir des résultats sur un ensemble de données représentatives.

     ◦ Méthode de calcul → nombre.
       Détermination d'une méthode de calcul (par exemple à partir de la forme de l'histogramme).

- Après les règles d'initialisation, on peut définir les *règles d'ajustement.* D'abord il faut définir les paramètres d'adaptation. Par exemple on peut choisir les valeurs minimales et maximales des arguments et une méthode d'adaptation. Ensuite on définit les règles qui indiquent les actions à entreprendre selon la situation.

- Enfin on définit les *règles d'évaluation.* Pour permettre à ces règles d'évaluer les résultats d'un bloc de connaissances, Ocapi fournit des moyens, comme l'accès automatique à des fichiers pour mettre certains résultats d'un opérateur dans un paramètre de sortie de bloc.

- Pour le moment il n'existe pas de type de règle spécifique pour les pre/post-conditions, il faut utiliser les règles existantes. Les règles d'évaluation peuvent par exemple servir à décrire des postconditions, qui expriment les effets d'un bloc de connaissances (but et opérateurs).

  - Les préconditions sont les conditions sur les descriptions qui autorisent l'application du but/opérateur.
  - Les postconditions stockent les résultats d'un but/opérateur dans les descriptions des arguments en sortie. Si un opérateur, par exemple, réduit la taille d'une image et que la taille de l'image est un champ des descriptions, une règle de "postcondition" va ranger la nouvelle taille de l'image dans la description.
    Remarque : même si la description des arguments en sortie est la même que la description des arguments en entrée, il faut le dire explicitement, sinon on perd de l'information.

On essayera de définir les règles de façon ascendante, en commençant par les règles des blocs de connaissances primitifs. La raison pour cela est que les blocs primitifs sont basés sur des opérateurs existants. L'information qu'ils donnent est la seule que l'on ait initialement. Avant de définir des règles à un niveau plus haut dans l'arbre, il faut d'abord savoir quelle est l'information disponible, sinon on pourrait définir des règles à des niveaux hauts dans l'arbre, qui ne pourraient finalement pas être implémentées.

Dans le but de rendre les règles générales, extensibles et claires, il faut autant que possible utiliser des règles "à grain fin". Des règles "à grain fin" sont des règles qui ne peuvent pas être représentées par plusieurs autres règles, plus élémentaires. Les règles à granularité plus grande, sont des raccourcis de raisonnement très généraux, qui ne prennent pas toujours bien en compte les exceptions.

En construisant les règles on a parfois besoin d'ajouter des opérateurs additionnels (pour calculer par exemple une valeur initiale) et de changer la structure des arbres (par exemple pour ajouter l'opérateur dans l'arbre statique, ajouter une requête dans un arbre de décomposition,...). On peut aussi être amené à modifier le contexte, au vu des informations nécessaires pour le raisonnement.

4.2.4 Validation

Une fois que l'arbre entier de décomposition d'un opérateur est construit, il reste à vérifier qu'il est utilisable avec différents contextes et sur différentes données (par exemples images d'un certain domaine). Le système développé doit être capable d'adapter dynamiquement les séquences d'opérateurs et leurs paramètres, afin d'obtenir le résultat désiré dans différents contextes et pour différentes données. Si ce n'est pas le cas, il faut revenir à la première phase et chercher les erreurs.

Table des matières

Index

Bibliographie

[Ala91] C. Alardo. Extensions des facilités de planification du générateur de système expert Ocapi. Rapport de DESS IA, Université de Paris VI, septembre 1991.

[Clé90] V. Clément. *Raisonnement cognitifs appliques au pilotage d'algorithme de traitement d'images.* PhD Thesis, Université de Nice-Sophia Antipolis, 1990.

[CT91] C. Clément and M. Thonnat. Progal : un système expert en traitement d'images de galaxies. In *8^{me} congrès RFIA*, Lyon, Novembre 1991.

[CT93a] C. Clément and M. Thonnat. Pilotage de procédures de traitement d'images pour la description morphologiques de galaxies. *TS*, 9(5), January 1993.

[CT93b] V. Clément and M. Thonnat. Integration of Image Processing Procedures, OCAPI: a Knowledge-Based Approach. *CVGIP*, 57(2), March 1993.

[CT93c] V. Clément and M. Thonnat. Pilotage de procédures de traitement d'images pour la description morphologique de galaxies. *TS*, 9(5), January 1993. numéro spécial Intelligence Artificielle.

[Ma90] R. Ma. Extensions du générateur de système expert Ocapi. Technical Report, I.N.R.I.A., Novembre 1990.

[MPT+95] S. Moisan, F. Ployette, M. Thonnat, V. Clément, and R. Vincent. Documentation de l'interface du logiciel OCAPI version *2.0.* Technical Report n. 0186, I.N.R.I.A. Sophia Antipolis, 1995.

[MVT+95] S. Moisan, R. Vincent, M. Thonnat, V. Clément, and J. van den Elst. Manuel de Référence du logiciel OCAPI version *2.0*. Technical Report n. 0183, I.N.R.I.A. Sophia Antipolis, 1995.

[TCO95] M. Thonnat, V. Clément, and J. C. Ossola. Automatic galaxy description. *Astrophysical Letters and Communication*, 31(1-6), 1995.

[TCvdE93] M. Thonnat, V. Clément, and J. van den Elst. Supervision of Perception Tasks for Autonomous Systems : the OCAPI Approach. Rapport de Recherche n^o 2000, Institut National de Recherche en Informatique et en Automatique, June 1993.

[TCvdE94] M. Thonnat, V. Clément, and J. van den Elst. Supervision of Perception Tasks for Autonomous Systems: the OCAPI Approach. *Journal of Information Science and Technology*, 3(2):140–163, Jan 1994. Also in Rapport de Recherche 2000, 1993, INRIA Sophia Antipolis.

[vdE92] J. van den Elst. Semantical Integration of Image Processing Operators for Object Detection in Road Scenes. Technical Report, I.N.R.I.A and University of Amsterdam, march 1992.

[Vin93] R. Vincent. Conception et réalisation d'un module d'apprentissage pour un générateur de systèmes experts. Rapport de D.E.A., Université de Nice Sophia-Antipolis, septembre 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399