MASTER EN SCIENCES INFORMATIQUES

Génération de programmes d'analyse statistique par transformation de stratégies de résolution de problème

Dechamps, Pascale; Parent, Fabian

Award date:
1994

Awarding institution:
Universite de Namur

FACULTÉS NOTRE-DAME DE LA PAIX DE NAMUR
INSTITUT D'INFORMATIQUE

Génération de programmes d'analyse statistique par transformation de stratégies de résolution de problème.

Pascale Dechamps
Fabian Parent

Promoteur : Monique Noirhomme

Mémoire présenté
en vue de l'obtention du diplôme de
Licencié et Maître en Informatique

Année académique : 1993-1994

Remerciements

Ce travail n'aurait pu voir le jour sans l'appui éclairé, actif et judicieux de toute une série de personnes.

En premier lieu, notre gratitude va à Madame Noirhomme qui nous a permis de réaliser le présent travail au sein de la société World Systems Europe, ainsi que pour les précieux conseils qu'elle nous a prodigués tout au long de ce mémoire.

Nous exprimons également nos plus vifs remerciements à Monsieur de Vaney pour son accueil amical et pour le temps qu'il nous a consacré. Sa collaboration et ses propositions bibliographiques nous furent d'une grande utilité.

Merci également à nos différents professeurs de licence en informatique qui, grâce à leurs cours, nous ont apporté les acquis nécessaires à la réalisation de ce travail.

Nous ne voulons pas terminer ces quelques lignes sans remercier également nos familles qui nous ont permis de poursuivre des études et nous ont soutenus au cours de ces années dont le présent mémoire est la conclusion.

Résumé

L'utilisation pratique des méta-informations statistiques, c'est-à-dire d'informations relatives aux données concrètes, devient importante pour la qualité du travail des organisations statistiques. Toutefois, avant de pouvoir utiliser ces méta-informations, il faut d'abord régler le problème de leur représentation.

La représentation et la manipulation de ces méta-informations étaient précisément l'objectif du projet EISI. Etant donné le succès du prototype développé, les fonctionalités de base ont été progressivement étendues. Une de ces extensions est notamment d'ajouter un générateur de programmes d'analyse statistique. Ce mémoire présente les différentes étapes de son élaboration.

Abstract

The practical use of statistical meta-information, i.e. information which describes data, has become very important in improving the quality of work in statistical organisations. In order to make effective use of this information, however, the representation problem must be addressed.

The representation and manipulation of meta-information was the objective of the EISI project. Because of the encouraging results of the prototype developed for EISI, the basic functionality has been progressively extended. A major extension is the requirement to develop program generation facilities for statistical analysis. This thesis elaborates the background and development procedures used.

TABLE DES MATIERES

Chapitre 1 : Introduction

Depuis plusieurs années, des recherches ont été effectuées sur le concept de méta-information statistique. Ces recherches ont amené à considérer l'utilisation pratique des méta-informations dans le travail des organisations statistiques. Cependant, avant de pouvoir les utiliser pratiquement, il est nécessaire de régler un certain nombre de problèmes, parmi lesquels celui de la définition d'une méta-information.

Le projet EISI a été mis sur pied sous l'égide du programme DOSES (Development of Statistical Expert Systems) d'Eurostat. L'objectif principal de ce projet était d'étudier les problèmes d'accès aux méta-informations. La première étape de la recherche a été de choisir le domaine d'application pour l'étude détaillée. La seconde était d'identifier et de développer un modèle pour les méta-informations statistiques. La troisième étape a consisté à définir et développer un modèle d'accès aux méta-informations statistiques. Ensuite, il s'agissait d'identifier et de documenter les méta-informations dans le domaine choisi. Enfin, la dernière étape consistait à développer un système prototype.

Comme les résultats obtenus étaient encourageants, le système EISI a été progressivement étendu, raffiné. L'une de ces extensions est d'ajouter au système un générateur de programme d'analyse, afin de pouvoir résoudre des problèmes statistiques

plus concrets. C'était le but de notre travail (au sein de la société World Systems Europe Limited) que d'étudier la faisabilité d'un tel système.

Cette société internationale, fondée en 1989, offre à une clientèle internationale composée de sociétés publiques, semi-publiques et privées une large gamme de services dans divers domaines :

- Informatique : conception d'applications informatiques, gestion et support techniques, conseil, formation, etc.;

- Recherche d'information dans les domaines Economiques et Statistiques: collecte de méta-information, prévisions, prospection, conseil;

- Recherche d'information dans le domaine des Affaires : analyse compétitive, etc.

Notre stage au sein de la société World Systems Europe Ltd[1] s'est déroulé en trois grandes étapes.

Nous avons tout d'abord pris connaissance de notre futur environnement de travail, à savoir le système EISI. Nous avons également étudié le Lisp, langage dans lequel le prototype (EISI) a été écrit. Nous avons aussi lu quelques articles traitant du *Knowledge-based software engineering*, pour nous familiariser avec certaines notions que l'on retrouve dans EISI. Nous pouvons résumer cette première étape en disant que nous avons pris connaissance de tout ce dont nous allions avoir besoin dans la suite du projet.

Lors de la seconde étape, nous avons étudié un peu plus en profondeur, les diverses structures d'EISI. Un rapport de comparaison de la qualité du logement en Wallonie et en Ecosse nous a été remis. C'est à partir de ce document que nous avons construit un exemple de problème à résoudre avec notre générateur.

[1] Notre stage s'est déroulé du 1er septembre 1993 au 31 janvier 1994.

Enfin, suite à l'étude des structures d'EISI, nous avons pu déterminer les extensions qui s'avéraient nécessaires afin de réaliser le projet, et nous les avons implémentées. Nous avons dès lors pu construire et tester (avec l'exemple construit précédemment) les différentes fonctions utiles à la génération de programme d'analyse statistique.

Structure du document

Le chapitre 2 est consacré à une présentation de l'environnement de travail dans lequel notre système va s'insérer, c'est-à-dire, le système EISI. Nous y aborderons principalement son architecture ainsi que son fonctionnement.

Le chapitre 3 donne un aperçu des idées générales du *Knowledge-based software engineering*. Ce chapitre traite en particulier de l'acquisition des spécifications. Cela nous permettra , dans le chapitre suivant, de pouvoir faire des liens entre le système EISI et un autre système.

Le chapitre 4 présente donc le fonctionnement du système IDeA (système d'aide à l'acquisition de spécifications), avant d'aborder les liens entre ce système, EISI et le *Knowledge-based software engineering*.

Le chapitre 5 est consacré au projet proprement dit. Nous y aborderons les différentes étapes de notre travail jusqu'à l'implémentation du générateur.

Enfin, dans le chapitre 6, nous présenterons un exemple de génération de programme d'analyse.

Les annexes quant à elles, regroupent principalement l'implémentation des diverses fonctions développées au cours du chapitre 5.

Chapitre 2 : Le Système E.I.S.I[1]

2.1. Introduction

Ce chapitre est consacré à une brève présentation du système EISI[2], qui permettra au lecteur de situer le cadre dans lequel notre travail a été réalisé. Il ne nous semble pas possible en effet, de comprendre notre projet sans un minimum d'informations concernant le système EISI, puisque notre travail complète ce dernier. Ainsi, dans un premier temps, nous exposerons les objectifs et le cadre du système EISI. Nous expliquerons ensuite, les différents modules qui composent son architecture. Nous nous pencherons enfin sur son fonctionnement.

---

[1] Expert Interface for Statistical Information.

[2]Pour de plus amples informations sur le système EISI, nous renvoyons le lecteur intéressé à [EPPRECHT], à [de VANEY92] et au chapitre 3 de [LUST&WEY], documents qui nous ont servi de base pour la rédaction de ce chapitre.

2.2. But et cadre du système EISI

L'utilisation des métadonnées[3], dans le domaine des statistiques officielles, est importante mais pose des problèmes (manipulation, définition, ...). C'est la raison pour laquelle le projet EISI a été mis sur pied. Son objectif est d'examiner et de résoudre les problèmes engendrés par les métadonnées.

Les métadonnées se révèlent être des informations importantes car elles spécifient les activités statistiques : elles donnent le contexte des informations, leur limitation, leur interprétation, etc. Ces informations doivent donc pouvoir être gérées et manipulées le plus simplement possible.

Cependant, le principal problème posé par les métadonnées est leur définition : il existe peu de définition formelle de ce que sont les métadonnées. Il est également difficile de les définir d'un point de vue pratique, car les utilisateurs ont des exigences différentes selon la nature et le contenu de ces métadonnées.

Diverses tentatives d'implémentation de systèmes de gestion de métadonnées ont été entreprises. Mais, elles se sont heurtées aux problèmes de la densité des structures d'information, de la représentation des métadonnées et de l'interfaçage de ces systèmes informatiques avec les systèmes statistiques existants.

Le projet EISI avait donc comme but d'étudier ces différents problèmes et de les solutionner. Le résultat de ces recherches a abouti à l'adoption d'une définition logique pour les métadonnées, qui comprend

> *"a) la méta-information factuelle, qui décrit les éléments d'information statistique en termes de structure, contenu et dérivation;*

[3] De façon très simpliste, nous pouvons définir les métadonnées comme étant des données sur des données. Par exemple, les informations attachées à l'INS (Institut National de Statistique), comme sa fonction, le pays, etc., constituent des métadonnées.

> *b) la méta-information conceptuelle, qui décrit la sémantique du domaine d'application, à partir duquel un corps de méta-information est dérivé, et qui est nécessaire pour l'interprétation des méta-informations."*[4]

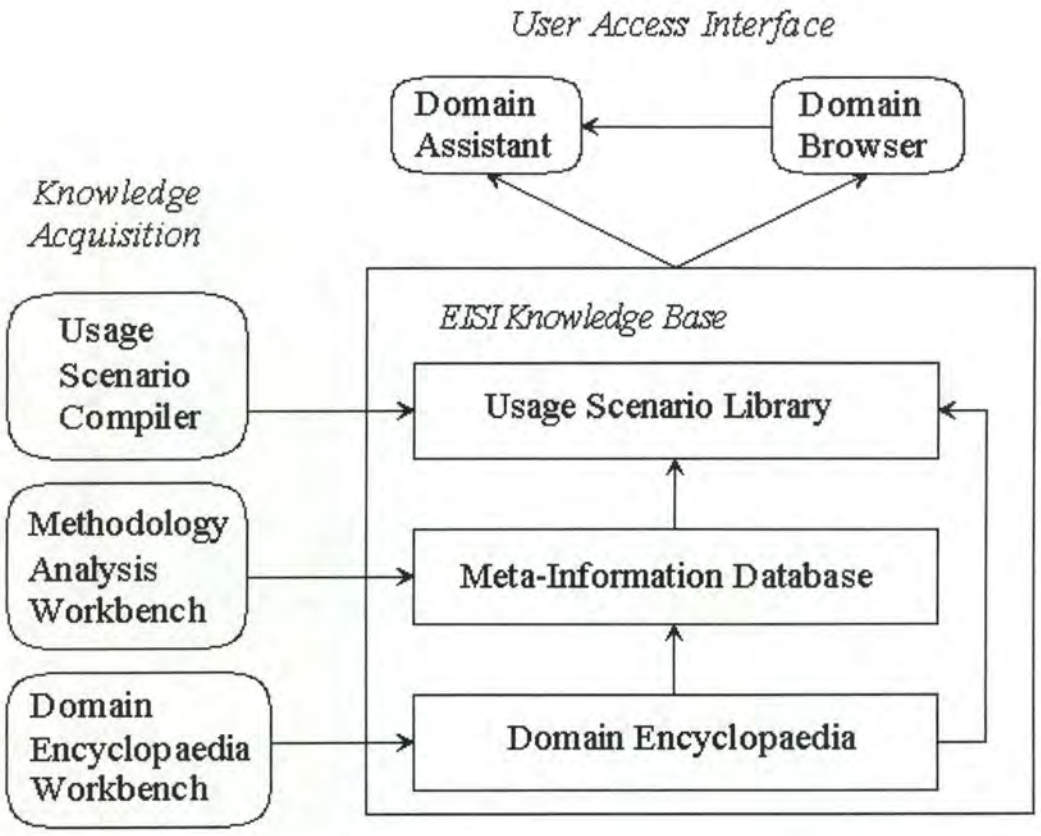
et s'est concrétisé par la mise au point d'un prototype, qui propose ainsi une solution aux problèmes pré-cités. Le système prototypique permet l'acquisition, la représentation et la manipulation de ces métadonnées. Ce prototype a été réalisé pour le domaine particulier des statistiques du Logement et de la Construction. Plusieurs raisons motivent ce choix : les caractéristiques des métadonnées de ce domaine sont communes à d'autres secteurs (économique ou social); le domaine du Logement et de la Construction est bien connu; enfin, les sources de données étaient disponibles pour les pays[5] partenaires de ce projet.

Le prototype a été construit de manière à pouvoir accéder facilement aux métadonnées et à donner une assistance à l'utilisateur. D'une part, en effet, le prototype offre la possibilité de naviguer et d'examiner l'information disponible dans le domaine. D'autre part, il fournit une assistance aux utilisateurs pour la formulation de leur problème, l'identification des métadonnées pertinentes, la sélection de techniques d'analyse et enfin, l'interprétation des données et des résultats.

2.3. Architecture du système EISI

De façon très globale, nous pouvons dire que l'architecture du système se compose de deux modules principaux : les *User Access Interfaces* et la *EISI Knowledge Base*.

[4] [de Vaney]
[5] Luxembourg, Belgique, Grande-Bretagne, Espagne, Ecosse.



2.3.1. User Access Interfaces

Les *User Access Interfaces* sont au nombre de deux : le *Domain Browser* et le *Domain Assistant*. Celles-ci permettent à l'utilisateur d'interagir avec la base de connaissance du domaine d'application.

a) Domain Browser

Le *Domain Browser* est une interface et un hypertexte qui permettent à l'utilisateur une navigation interactive et l'affichage de l'information disponible,

contenue dans la base de connaissance. Nous pouvons donc dire que c'est un module passif.

b) Domain Assistant

Le *Domain Assistant*, contrairement au *Domain Browser*, est un module "intelligent" d'assistance à l'utilisateur pour la formulation de stratégies. Il aide d'abord l'utilisateur à formuler son problème. Une fois la définition du problème assez précise et complète, le *Domain Assistant* extrait de la *Usage Scenario Library* un scénario applicable au problème. Il étend ensuite la connaissance du scénario (en fonction du contexte et des métadonnées) et donne à l'utilisateur une indication sur les données et les sources relatives au problème. Il fournit également à l'utilisateur un plan de solution pour l'analyse et l'interprétation des données et des résultats.

2.3.2. EISI Knowledge Base

Le système prototype est étayé par une base de connaissance distribuée (la *EISI Knowledge Base*), avec laquelle communiquent

- les modules d'acquisition de la connaissance : le Domain Encyclopaedia Workbench, le Methodology Analysis Workbench et le Usage Scenario Compiler

- les modules qui fournissent les deux modes d'accès au système : le *Domain Browser* et le *Domain Assistant*.

La base de connaissance est divisée en trois modules : la *Domain Encyclopaedia*, la *Meta-information Database* et la *Usage Scenario Library*, représentant respectivement la connaissance conceptuelle, factuelle et stratégique d'un domaine d'application.

a) Domain Encyclopaedia

La *Domain Encyclopaedia* contient la connaissance théorique à propos d'un domaine d'application statistique particulier. Cela inclut toute la connaissance indirectement reliée aux collections spécifiques de données : définitions de termes et de concepts, classifications et taxonomies, etc.

b) Meta-information Database

La *Meta-information Database* contient toute la méta-information concernant les collections de données disponibles dans le domaine d'application, ainsi que leurs sources et dérivations.

c) Usage Scenario Library

La *Usage Scenario Library* contient un ensemble de stratégies (scénarios) pour la résolution de problèmes (incluant l'identification des données pertinentes, de leurs sources et de leur interprétation) sous la forme de plan abstrait; chaque scénario comprend le plan de solution et le contexte du problème ainsi que les contraintes globales sur les métadonnées. Les *Usage Scenarios* constituent la connaissance fondamentale utilisée par le *Domain Assistant*.

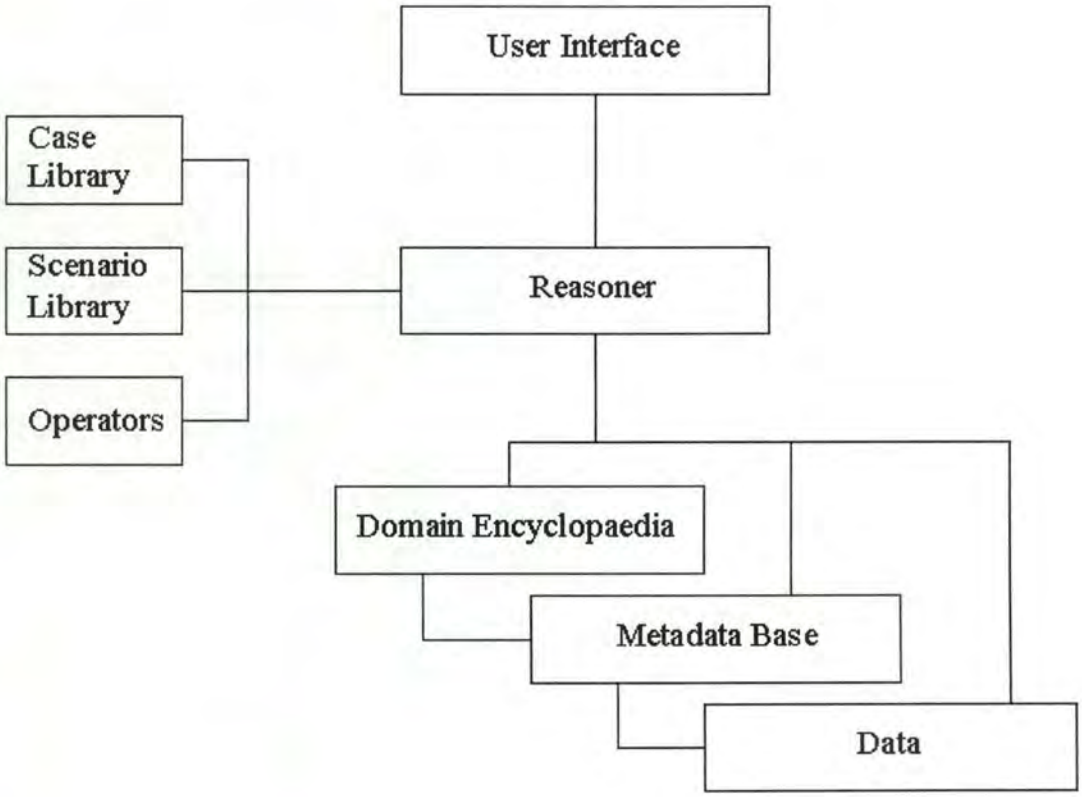
Le scénario a été défini conceptuellement par l'équipe responsable du développement du prototype, de la façon suivante :

> *"(...) un scénario décrit un problème se déroulant dans le domaine d'application, et spécifie une ou plusieurs solutions partielles dans le contexte d'utilisation de méta-information. (...) Une solution partielle décrit premièrement les caractéristiques de la méta-information requise pour le problème dans un contexte particulier, et deuxièmement, décrit*

la séquence d'opérations, ou plan, qui devrait être appliquée à la méta-information disponible, afin d'atteindre les buts ou de fournir un support de décision dans l'espace du problème."[6]

Un scénario est donc utilisé à la fois comme un moyen d'acquérir de la connaissance et comme un modèle pour représenter et organiser cette connaissance.

2.4. Description fonctionnelle



Dans le cadre de cette section, nous allons examiner le fonctionnement du mécanisme de raisonnement et en particulier le *Domain Assistant*. Ce mécanisme de raisonnement se compose des étapes suivantes :

- Problème posé par l'utilisateur

[6] [de Vaney]

- Validation
- Consultation de la *Case Library*
- Recherche d'un ensemble de candidats
- Réduction de l'ensemble des candidats
- Recherche d'information
- Evaluation
- Expansion et présentation

a) Problème posé par l'utilisateur

L'interface utilisateur présente à ce dernier un écran dont il doit remplir les champs. L'utilisateur les complète alors soit de lui-même, soit à l'aide des valeurs qui peuvent lui être proposées par l'interface (ce qui, dans ce cas, limite le risque d'erreurs). Ces différents champs constituent un *goal.* Un *goal* se compose d'un ensemble de qualificateurs qui correspondent chacun à un élément de la définition du problème (par exemple, la période de temps).

Une fois cette opération terminée, l'utilisateur le signale à l'interface (par exemple en cliquant sur un bouton OK). Le *goal* ainsi créé, est passé au *Reasoner* pour qu'il le valide.

b) Validation

Cette opération de validation consiste à vérifier que tous les champs du *goal* sont corrects. En d'autres termes, le *Reasoner* s'assure que :

- les valeurs données sont acceptables. Par exemple, il contrôle que l'année est valide;

- il y a de la méta-information disponible dans la *Meta-information Database* sur le problème posé;

- il n'y a pas de conflit sémantique d'information (ex : la période initiale donnée est postérieure à la période finale donnée), c'est-à-dire que le problème posé a un sens.

Si le *goal* n'est pas valide, le *Reasoner* le renvoie à l'interface utilisateur pour correction ou fin (s'il n'y a pas de méta-information disponible).

c) Consultation de la Case Library

Le *Reasoner* va consulter la *Case Library* - celle-ci contient tous les cas résolus précédemment par le *Reasoner* (résolutions ayant abouti à une réussite ou non) - afin de s'assurer que le problème n'a pas déjà été rencontré. Si le problème a déjà une solution, le *Reasoner* la présentera à l'utilisateur. Dans le cas contraire, il va rechercher les candidats susceptibles de résoudre le problème.

d) Recherche d'un ensemble de candidats

Cette étape consiste à construire un ensemble de candidats. Ce processus se déroule de la façon suivante : le Reasoner va comparer le *goal* avec des sélecteurs. A chacun de ceux-ci correspond un schéma (c'est-à-dire un plan de solution). Les sélecteurs contiennent des connaissances sur la relation entre la définition du problème, son contexte et la stratégie de résolution du problème. Ils comprennent aussi la spécification de la méta-information requise par leur schéma. Chaque fois qu'un *goal* et qu'un sélecteur correspondent, l'identifiant de ce sélecteur est placé dans l'ensemble des candidats possibles. Ce processus se poursuit pour tous les sélecteurs. A la fin de cette étape, trois cas peuvent se présenter :

- l'ensemble est vide. Cela signifie soit que le problème n'est pas solvable soit que la stratégie de solution n'existe pas dans le système.

- l'ensemble ne contient qu'un seul élément. Dans ce cas, il ne reste qu'à vérifier les qualificateurs du sélecteur et les préconditions (cfr. infra).

- l'ensemble contient plus d'un élément. Il est alors nécessaire de réduire cet ensemble jusqu'à n'avoir plus qu'un élément. C'est ce dont se charge l'étape suivante.

e) Réduction de l'ensemble des candidats

Quatre critères peuvent permettre de réduire l'ensemble des candidats : des qualificateurs supplémentaires, des préconditions, le niveau de généralité/spécificité du sélecteur et l'arbitrage.

- les qualificateurs supplémentaires

Le *Reasoner* prend un sélecteur de la liste constituée à l'étape d, cherche ses qualificateurs et, pour chacun d'entre eux, demande de l'information supplémentaire afin de les compléter. Par exemple, le qualificateur supplémentaire concerne les *time-series*. Le *Reasoner* demande à l'utilisateur si les *time-series* l'intéressent. S'il répond par l'affirmative, le sélecteur est toujours candidat. S'il répond par la négative, le sélecteur sera enlevé de la liste. Le *Reasoner* passera la liste en revue et éliminera tous les sélecteurs dont un des qualificateurs concerne les *time-series* (ce qui évite de reposer plusieurs fois la même question à l'utilisateur).

- les préconditions

Les préconditions servent également à éliminer des éléments de la liste des candidats. En effet, comme elles permettent de s'assurer que l'information nécessaire à la résolution du problème est disponible et adéquate, les éléments pour lesquels ce ne serait pas le cas seront enlevés de la liste.

- niveau de généralité/spécificité[7]

Il s'agit ici de choisir la stratégie la plus spécifique au problème (les sélecteurs les plus généraux ne seront donc pas retenus). Cependant, il peut arriver que le Reasoner ne puisse choisir entre deux candidats :

Exemple :

L'utilisateur veut faire une comparaison entre la Grèce et l'Espagne.
Les candidats (sélecteurs) sont au nombre de trois :
A pour les comparaisons entre deux pays
B pour les comparaisons entre la Grèce et un autre pays
C pour les comparaisons entre l'Espagne et un autre pays
Lors de cette étape, le sélecteur A sera éliminé puisque c'est le sélecteur le plus général. Mais, rien ne permet de déterminer le sélecteur le plus spécifique entre B et C. Dans ce cas, c'est l'arbitrage qui va décider.

- l'arbitrage

A ce stade, plusieurs possibilités existent : soit le Reasoner choisit au hasard un des sélecteurs, soit l'utilisateur choisit lui-même.

[7] Signalons que cette étape aura lieu uniquement dans le cas où l'ensemble des candidats à la résolution du problème contient plus d'un élément.

f) Recherche d'information

Nous avons vu que le sélecteur incluait la spécification des méta-informations requises par leur schéma. Grâce à celle-ci, le *Reasoner* va rechercher dans la *Meta-information Database*, toute l'information (correspondant à ces métadonnées) nécessaire pour résoudre le problème.

g) Evaluation

Cette étape consiste à évaluer le schéma : évaluer les conditions (c'est-à-dire vérifier si elles sont respectées) et les opérations contenues dans ce schéma. L'évaluation des opérations consiste à contrôler que leurs préconditions sont respectées et à instancier les valeurs par celles données par l'utilisateur.

h) Expansion et présentation

Le processus se termine par l'expansion du schéma (il peut nécessiter l'appel à d'autres schémas partiaux) et la mise en forme dans un "cas". Le *Reasoner* stockera alors le cas dans la *Case Library*, et le présentera à l'utilisateur pour exécution (afin d'avoir la solution pratique, concrète du problème).

Chapitre 3 : Le Knowledge-based software engineering

3.1. Introduction

Nous nous proposons dans ce chapitre de donner d'une part, une vue générale du *knowledge-based software engineering*, et d'aborder d'autre part un point particulier du *knowledge-based software engineering*, en l'occurrence l'acquisition des spécifications.

La première partie de ce chapitre sera donc consacrée à une vue d'ensemble du *knowledge-based software engineering*. Nous y exposerons d'abord ses motivations. Nous tenterons ensuite d'expliciter un peu plus ce qu'est le *knowledge-based software engineering*. Nous passerons également en revue ses différents objectifs et nous brosserons enfin sa situation actuelle.

Dans la seconde partie de ce chapitre, nous nous intéresserons à un domaine particulier du *knowledge-based software engineering*, à savoir l'acquisition des spécifications. Nous y aborderons les éléments principaux : les langages de spécification, les méthodologies d'acquisition des spécifications, la validation et la maintenance des spécifications.

Ce chapitre posera ainsi les bases nécessaires à la comparaison (que nous ferons dans le chapitre 4) du système EISI avec un système d'acquisition des spécifications : IDeA. Nous y verrons, en effet, que bien que le système EISI ne relève pas du *knowledge-based software engineering*, il n'en possède pas moins des caractéristiques communes avec ce dernier.

3.2. Aperçu du Knowledge-based software engineering

Le *software engineering* est la discipline de la définition, de la conception, du développement et de la maintenance des logiciels d'exploitation. La génération actuelle des outils *CASE* (*Computer-aided software engineering*) aide à réduire le travail de "bureau" du *software engineering*. Puisque le *software engineering* est une activité mettant en oeuvre une quantité considérable d'information, des gains de productivité encore plus élevés que ceux obtenus grâce aux outils *CASE*, seront atteints avec les outils intelligents assistés par ordinateur; cette application de l'intelligence artificielle au *software engineering* est appelée le *Knowledge-based software engineering* (*KBSE*). Comme la recherche en intelligence artificielle arrive à maturité, et que les outils basés sur la connaissance sont développés, l'impact futur sera vraisemblablement significatif [Bar, Cohen&Feigenbaum].

3.2.1. Les motivations du Knowledge-based software engineering

L'impact économique potentiel de l'accroissement de la qualité et de la productivité de la conception d'un logiciel est énorme. Les logiciels constituent à l'heure actuelle 80 % du coût total moyen d'un système informatique. En dépit du large marché existant (aux USA, l'on dénombre actuellement plus d'un million de professionnels du *software engineering*), le *software engineering* en est encore à ses balbutiements, et ne satisfait pas les besoins existants, encore moins les besoins futurs prévus. Les possibilités offertes par le nouveau matériel informatique d'exécuter des applications sophistiquées ne sont pas exploitées par les technologies de *software engineering* actuelles. Les grands projets de logiciels coûtent typiquement deux fois plus que budgétisés, sont délivrés en retard ou pas du tout, et ont tellement d'erreurs que le coût

d'élimination des défauts pourrait dépasser le coût de la conception et de l'encodage combinés. Par ailleurs, la modification est si difficile que la maintenance absorbe plus de la moitié des ressources totales. Cet état de choses est souvent appelé la "crise du logiciel".

Cette crise trouve ses origines dans la différence entre la technologie *software* et la technologie *hardware*. Le *hardware* a toujours été limité par les technologies matérielles et industrielles. Les progrès réalisés dans ces technologies ont permis d'atteindre au cours de ces 25 dernières années un rapport prix/performance mille fois moins élevé. Par contraste avec le *hardware*, le *software* est un problème de conception et non pas un problème industriel. L'accroissement de productivité de la conception d'un logiciel d'un facteur 10 est dû au développement de langages de très haut niveau, aux méthodes structurées de développement de logiciels et de gestion de projets complexes, et à de meilleurs ingénieurs software.

La technologie *CASE* émergente fournit un support informatique au développement structuré de logiciels et à la gestion de projet. Cependant, cette technologie n'aborde pas les caractéristiques de base du *software engineering* : la conception d'un logiciel est une activité mettant en oeuvre une quantité considérable d'information, qui débute par une demande informelle de ce qui doit être fait, et qui résulte dans un objet formel hautement détaillé, à savoir un logiciel d'exploitation. A l'heure actuelle, ce processus est "manuel" et est prédisposé à l'erreur. En outre, le résultat final est dénué de la connaissance de conception qui conduit au logiciel d'exploitation. C'est précisément cette connaissance qui est nécessaire à la maintenance et à la mise à jour du logiciel d'exploitation, et qui pourrait être réutilisée dans le développement de nouveaux logiciels [Bar, Cohen&Feigenbaum].

3.2.2. Qu'est-ce que le Knowledge-based software engineering?

La technologie du *software engineering* s'est développée d'une manière significative depuis l'introduction de l'ordinateur numérique. Au début, les programmeurs encodaient la connaissance sous forme de suites de 0 et de 1. Ce niveau

de programmation est très loin du niveau auquel les humains imaginent les problèmes et leurs solutions. C'est pourquoi est apparue l'idée que l'ordinateur lui-même prenne en charge l'augmentation du niveau conceptuel de programmation. Les compilateurs furent l'un des premiers outils de programmation automatique développés, permettant l'encodage de la connaissance à un niveau de formules algébriques et de structures de contrôle de haut niveau.

Dans les années 1980, trois groupes d'outils de développement de logiciels, collectivement connus sous le nom de *CASE*, furent introduits sur le marché commercial : les générateurs de code, les assistants d'analyse et de conception et les outils de gestion de projet.

Le *knowledge-based software engineering* succédera vraisemblablement à la technologie *CASE*, en remplaçant les outils basés sur les données par des outils basés sur la connaissance [Bar, Cohen&Feigenbaum]. Une base de connaissance est une base de données augmentée de règles pour le raisonnement sur les données, et d'un moteur d'inférence qui applique ces règles. Grâce à cette capacité de raisonnement ajoutée, les outils basés sur la connaissance sont plus puissants que les outils basés sur les données. Des outils de synthèse de programme qui étendent grandement les capacités des générateurs de code actuels ont été développés dans des laboratoires de recherche, et quelques-uns sont disponibles dans le commerce. Des outils de conception qui raisonnent sur les sémantiques d'un domaine et aident l'utilisateur à développer une conception correcte, sont en cours de développement. Enfin, des outils de gestion de projet qui coordonnent efficacement de larges groupes de personnes, et qui comprennent l'intégration de leurs tâches séparées, remplaceront les outils existants qui aident uniquement à tracer les statistiques et les coûts d'un projet.

3.2.3. Les objectifs du Knowledge-based software engineering

Le *knowledge-based software engineering* introduit un changement fondamental dans le cycle de vie d'un logiciel : la maintenance et l'évolution sont réalisées par la modification des spécifications, suivie de la redérivation de l'implémentation, plutôt que

par la modification directe de l'implémentation. Le *knowledge-based software engineering* utilise des techniques basées sur la connaissance et d'autres techniques de l'intelligence artificielle, dans le but de dépasser les capacités des outils *CASE* actuels. En effet, les outils *CASE* actuels aident seulement à assurer la cohérence interne de la hiérarchie de modules. L'objectif d'ensemble est de fournir une assistance intelligente par ordinateur, pour l'ensemble des phases du cycle de vie d'un logiciel. Plus spécifiquement, le *knowledge-based software engineering* a cinq buts [Bar, Cohen&Feigenbaum] :

1) Formaliser les *artefacts* du développement d'un logiciel et les activités d'ingénierie qui produisent ces *artefacts*. Par exemple, les langages de spécification formels permettent de déclarer précisément et sans ambiguïtés les spécifications.

2) Utiliser des technologies de représentation de la connaissance, pour enregistrer, organiser, et extraire la connaissance associée aux décisions qu'entraîne la conception d'un logiciel d'exploitation. Cette base de connaissance devrait constituer une "mémoire de société", qui coordonnerait les efforts de l'équipe de développement et faciliterait l'évolution et la maintenance. Un enregistrement explicite des décisions de conception et de leur justification serait une amélioration considérable par comparaison aux méthodes actuelles de *debugging* et de maintenance, qui exigent la reconstitution des décisions de conception à partir du code source et de la documentation écrite.

3) Produire des assistants basés sur la connaissance pour synthétiser et valider le code source à partir de spécifications formelles. Cela permettra d'effectuer la maintenance en modifiant la spécification et en répétant les étapes de synthèse du code source, avec les modifications appropriées. Il y a également un besoin d'assistance pour la récupération des spécifications de haut niveau à partir du code source. Enfin, l'assistance par ordinateur pour la compréhension des programmes complète la synthèse du code source à partir des spécifications.

4) Produire des assistants basés sur la connaissance pour développer et valider les spécifications. L'assistance basée sur la connaissance pour le développement des spécifications aidera un utilisateur à résoudre des demandes contradictoires, à raffiner

des demandes incomplètes et informelles en spécifications précises, et à utiliser la connaissance du domaine dans le développement de conceptions de systèmes. Des méthodes de validation pourraient utiliser la spécification elle-même comme un prototype exécutable ou exécuter des types variés d'analyse des spécifications.

5) Produire des assistants basés sur la connaissance pour gérer de grands projets de logiciels. Les bases de connaissance formeront les modèles sémantiques d'un projet dans son entier, incluant l'historique, les procédures, et politiques. Dans un proche avenir, la technologie existante des systèmes experts pourra être utilisée pour intégrer l'information d'un large projet logiciel et pour automatiser les aspects non créatifs de la gestion d'un projet.

3.2.4. L'état actuel du Knowledge-based software engineering

Deux tendances majeures ont émergé dans la recherche sur la programmation automatique [Bar, Cohen&Feigenbaum].

La première tendance est l'arrivée à maturité du travail réalisé en synthèse de programme, où une spécification formelle de haut niveau est transformée en un programme qui fonctionne.

La seconde tendance est l'élargissement de la programmation automatique pour inclure l'entièreté du cycle de vie d'un logiciel : le *knowledge-based software engineering*. En particulier, la recherche en acquisition des spécifications a évolué de la spécification d'un programme à la spécification d'un système. Des assistants basés sur la connaissance pour l'acquisition, la validation, et la maintenance des spécifications ont été développés dans des laboratoires de recherche. Ce travail est encore dans la phase de base de recherche. Cependant, il y a des chances pour que le développement de techniques basées sur la connaissance transforme radicalement le cycle de vie d'un logiciel.

La technologie de l'intelligence artificielle est en cours d'introduction sur le marché du *software engineering*, par trois voies [Bar, Cohen&Feigenbaum] :

- Premièrement, les vendeurs d'outils *CASE* améliorent la performance de leurs produits en incorporant des technologies de l'intelligence artificielle;

- Deuxièmement, des compagnies qui conçoivent des systèmes experts intègrent leur système expert *shell* dans les environnements et les langages du *software engineering* conventionnels;

- Troisièmement, plusieurs grandes sociétés développent des environnements de *software engineering* qui incorporent des techniques du *knowledge-based software engineering*.

3.3. Acquisition des spécifications

3.3.1. Acquisition des spécifications basée sur les connaissances

a) Définition

L'acquisition de spécifications se définit comme le processus qui permet de produire les trois types de documents suivants [Bar,Cohen&Feigenbaum] :

- le document "d'exigences" (*requirements document*) qui décrit ce que l'utilisateur désire;

- le document de spécification (*specification document*) qui décrit le comportement externe du système ainsi que l'interface utilisateur;

- le document de conception (*design document*) qui décrit le comportement interne du système, sa décomposition en sous-modules et le format des données utilisées par ces modules.

b) Utilisation de l'intelligence artificielle

Comme nous l'avons signalé dans la section 3.2.1., le *software engineering* est supporté par des outils *CASE*. Ceux-ci ont entre autre pour objectif d'aider au développement d'un logiciel. L'idée de base est de rendre ce développement plus interactif et cohérent avec la façon dont les gens pensent et travaillent [Forte&Norman].

Au début de l'apparition des outils *CASE*, ceux-ci offraient une aide au niveau du langage utilisé, et ils ne supportaient, en général, qu'une seule étape du cycle de vie du développement d'un logiciel [Forte&Norman]. Ils ne permettaient qu'une vérification de la cohérence syntaxique des spécifications : par exemple, ils vérifient qu'il ne se produit pas de *dead ends* (blocage) entre deux modules.

Cependant, depuis quelques années, les outils *CASE* évoluent vers une couverture de tout le cycle de vie du développement d'un logiciel. De plus en plus également, ils permettent une vérification sémantique des spécifications, c'est-à-dire une preuve de complétude et de cohérence. Afin de réaliser cet objectif, l'idée est apparue d'utiliser des techniques d'acquisition de connaissance employées en intelligence artificielle pour formuler un modèle sémantique d'un domaine d'application. Cela permet d'aider l'utilisateur à développer des spécifications complètes, correctes et cohérentes, puisque ce dernier communique avec l'outil en termes orientés domaines [Bar,Cohen&Feigenbaum].

c) Composants de l'acquisition de spécifications basée sur la connaissance

L'élément central dans l'acquisition de spécifications est le développement de modèles de domaine de problèmes. Grâce à cela, les assistants intelligents vont pouvoir aider les utilisateurs à formuler leurs exigences et spécifications. Enfin, cela peut faciliter la réutilisation des spécifications de conception. Examinons un peu plus en détail ces trois composants, identifiés par [Bar, Cohen&Feigenbaum] : les modèles de domaine, les exigences et spécifications, et les spécifications de conception.

- Modèles de domaine

Grâce aux techniques d'acquisition de connaissance, il est possible de développer des modèles de domaine de problèmes. Ces modèles de domaine servent alors de base de connaissance aux systèmes experts. Ceux-ci aident l'utilisateur dans son étape d'acquisition de spécifications. Les modèles de domaine doivent représenter les propriétés statiques (objets, attributs, ...), les propriétés dynamiques (opérations sur un objet, propriétés d'un objet, ...) et les contraintes d'intégrité d'un domaine de problème.

Cette connaissance est nécessaire pour les générateurs de programmes et pour écrire de bons programmes. Cependant, créer cette base de connaissance est elle-même une activité d'acquisition de connaissance difficile puisqu'elle demande la collaboration d'ingénieurs de connaissance et d'experts du domaine traité. De plus, elle peut exiger plusieurs années avant d'être complète. C'est la raison pour laquelle des assistants intelligents ont été développés pour aider les experts à développer des modèles de domaine.

- Exigences et spécifications

Comme nous venons de le voir, le modèle d'un domaine fournit la connaissance nécessaire pour que l'assistant intelligent aide les utilisateurs à développer au mieux leurs exigences et leurs spécifications. Les assistants intelligents utilisent le

modèle de domaine comme un moyen de communication entre l'assistant automatisé et l'utilisateur. Plusieurs de ces assistants intelligents ont été mis au point pour supporter les diverses activités que l'on peut retrouver dans l'acquisition d'exigences et les spécifications : enregistrer les décisions de spécification en termes de domaine pour faciliter plus tard la maintenance, réutiliser des parties de spécifications, générer des spécifications formelles et cohérentes à partir d'exigences de départ vagues, etc. En résumé, nous pouvons dire que l'assistant intelligent utilise le modèle de domaine pour fournir un support sémantique à l'acquisition des spécifications.

- Spécifications de conception

La technologie des systèmes experts va permettre de développer des assistants intelligents interactifs. Certains de ces outils utilisent un modèle de domaine pour supporter le développement et la réutilisation d'une conception d'un système. Aussi, un premier avantage des outils de conception orientés connaissance est de pouvoir retrouver de l'information dans la base de connaissance afin de réutiliser les anciennes spécifications de conception. Un second avantage est la capacité de ces outils à expliquer, en termes du domaine, une spécification de conception qui évolue.

3.3.2. Langages de spécification

a) Objectifs des langages de spécification

Les langages de spécification vont permettre de décrire les domaines de problèmes, le comportement attendu du système que l'on développe, ainsi que toute la conception d'un tel système. Ces langages sont de plus en plus supportés par des outils automatiques. Or, plus le langage est expressif, plus il sera difficile de le supporter par ce genre d'outils automatiques. Un compromis sera donc fait entre le niveau d'expressivité du langage et le niveau d'outils de support qu'ils permettront (compilateurs, interpréteurs, etc.).

Les langages de spécification diffèrent entre eux, selon [Bar,Cohen&Feigenbaum], par ce qui peut être énoncé de façon concise, redondante ou par défaut. Ainsi, les langages qui supportent les contraintes facilitent le développement des spécifications parce que ces contraintes expriment des relations qui, autrement, auraient dû être répétées en de multiples endroits. De même, les langages de spécification avec hiérarchies de classes permettent une économie d'expression. En effet, ces langages présentent deux caractéristiques essentielles : d'une part, une hiérarchie de classes permet de regrouper des objets ayant une structure et des opérations similaires, d'autre part une sous-classe hérite, par défaut, des propriétés et des opérations de la super-classe. Grâce au langage, les similitudes sont exprimées explicitement et de façon concise, ce qui permet une économie d'expression. De plus, les changements seront hérités automatiquement par la sous-classe, ce qui assure la cohérence du système.

b) Quelques types de langages de spécification

- Langages de programmation logique

L'idée sous-jacente à ce type de langages est de faire une spécification à partir d'une description logique du domaine du problème étudié. Une caractéristique importante de ces langages est qu'ils ont une sémantique déclarative formelle. Ceci est intéressant pour pouvoir vérifier l'exactitude des programmes dérivés des spécifications.

- Langages orientés objets

Les langages orientés objets permettent d'implémenter des objets comme structures de données et des méthodes (fonctions) pour accéder à ces structures. Ces langages sont caractérisés par le fait que les objets du domaine sont classés hiérarchiquement et qu'ils peuvent hériter de propriétés de classes supérieures. La première caractéristique, la hiérarchie, permet à l'utilisateur de définir un nouvel objet comme une instance d'une classe, ce qui lui évite de devoir réécrire certaines caractéristiques communes à son objet et à la classe à laquelle il appartient. En d'autres

termes, cela donne la possibilité à l'utilisateur de construire une librairie d'objets réutilisables. La seconde caractéristique, l'héritage, permet à l'utilisateur de définir une nouvelle classe d'objets qui héritent des propriétés d'une classe existante. De plus, comme nous l'avons signalé plus haut, l'héritage réduit l'information redondante et simplifie les modifications [Ince].

Ces avantages, facilité de modification et réutilisation, sont des atouts en faveur de l'utilisation des langages orientés objets. Enfin, une spécification dans un langage orienté objet peut être utilisée pour simuler un modèle de domaine, et par là, faciliter le prototypage rapide et la validation des spécifications.

- Langages de spécification de système

Les langages de spécification de système contiennent des contraintes pour spécifier les relations entre les composants du système ainsi qu'un moyen de spécifier les transitions d'état à état. Cela implique qu'un tel langage a besoin de construction pour indiquer les états, les attributs des états et les transitions entre états. Les contraintes facilitent le raffinement des spécifications du système. Pour le prototypage et la validation du système, les spécifications devraient être exécutables [Bar, Cohen& Feigenbaum].

Un exemple d'un tel langage est "GIST". Le langage Gist, développé par [Feather], fournit une grande flexibilité ainsi qu'une facilité d'expression pour décrire tous les comportements acceptables d'un système. De plus, ce langage sert aussi bien de langage de spécification que de langage d'implémentation.

L'information est modélisée par des objets et des relations entre ceux-ci. Tous ces objets et relations correspondent à un "état". Un changement dans le domaine est modélisé par la création et la destruction d'objets et par l'insertion et la suppression de relations. Chaque changement est une transition de l'état courant vers un nouvel état. Passons brièvement en revue les caractéristiques de Gist :

1. Gist dispose d'un modèle de données relationnel et associatif, qui permet de "capturer" les structures logiques sans pour autant imposer un régime d'accès. L'uniformité de ce modèle relationnel facilite l'expression des spécifications modifiées.

2. L'information implicite permet la définition implicite des données au moyen d'autres données. Ces relations implicites sont robustes du fait de leur nature et parce qu'elles sont définies en termes d'informations dont elles dépendent.

3. Les références historiques permettent d'extraire de l'information d'anciens états du processus. La référence historique permet au spécifieur de décrire facilement et sans ambiguïté l'information provenant d'états passés, dont il a besoin. La référence historique permet ainsi la réutilisation.

4. Les contraintes expriment des restrictions sur le comportement du système, au moyen de déclarations globales et de pré et post conditions. Les contraintes, par leur nature, garantissent que tous les comportements exprimés dans une spécification qui a été modifiée ne vont pas engendrer d'autres comportements.

5. Un *demon* possède deux composants : un prédicat jouant le rôle de *trigger* et un *statement*. Lorsque le prédicat du *demon* est vrai, cela déclenche le *statement*. Cette nature descriptive du *demon* fournit une certaine robustesse : le *demon* sera toujours déclenché (à condition que le prédicat soit vrai), même si des modifications dans les spécifications changent le comportement du système.

3.3.3. Méthodologies d'acquisition de spécifications

a) Définition

Une méthodologie d'acquisition de spécification est un ensemble de méthodes et de principes pour indiquer ce qui doit être modélisé par une spécification et comment la

spécification devrait être développée. Une méthodologie se centre sur les activités du processus d'acquisition des spécifications [Bar, Cohen&Feigenbaum].

b) Approche conventionnelle

Lorsqu'on développe un logiciel, on part des exigences de l'utilisateur. Or, cette transformation des exigences en un logiciel approprié se révèle être un processus complexe. C'est pourquoi l'approche conventionnelle a pour objectif de réduire cette complexité en se concentrant d'abord sur ce que le système devrait faire, et ensuite sur la façon d'accomplir les objectifs. Cette approche se base sur l'idée suivante : les exigences sont fixées tôt (elles forment une base stable pour l'implémentation) et on permet uniquement que chaque phase puisse faire un *feed-back* vers la phase précédente. Mais, cet avantage est aussi son désavantage, car les erreurs dans les exigences et la conception ne se manifesteront qu'à l'implémentation. La correction coûtera alors cher ou sera impossible. Cette approche convient pour des domaines bien connus où les problèmes sont familiers et où les exigences peuvent être spécifiées sans tester d'abord un prototype.

c) Réutilisation

La réutilisation est un aspect très important du *software engineering*. Très vite en effet, suite à une demande croissante de logiciels, on en est arrivé à réutiliser du code. Actuellement, on se tourne vers la réutilisation à un plus haut niveau, c'est-à-dire la réutilisation des spécifications et de la conception, car c'est à ces niveaux que les bases du développement du logiciel sont établies [Williams]. On aimerait donc réutiliser les spécifications d'exigences, les spécifications de conception et les dérivations de programmes. Cependant, pour pouvoir réutiliser les spécifications de conception, par exemple, il est nécessaire d'avoir de bons composants de conception ainsi qu'une certaine connaissance qui permette de les situer et de les combiner de façon appropriée. Cela signifie qu'on a besoin d'outils plus puissants qu'un simple éditeur de texte : l'application de la technologie des systèmes experts au *software engineering* peut être un

moyen de réaliser la réutilisation de composants. En particulier, on a besoin de trois types de supports, selon [Bar,Cohen&Feigenbaum] :

1. des outils pour retrouver les conceptions pertinentes;

2. des outils pour déterminer quelles parties des conceptions précédentes peuvent être directement réutilisables et ce qu'il faut pour qu'elles soient reconçues;

3. des outils d'édition de haut niveau pour modifier la conception précédente basée sur des changements sémantiques.

d) Expérimentation et évolution

Une autre méthodologie a vu le jour grâce aux capacités de prototypage rapide des langages d'intelligence artificielle : la programmation exploratoire. Cela va permettre aux exigences et à la conception d'émerger de l'expérimentation avec un programme. Le programme devient un outil expérimental. Pour [Bar,Cohen&Feigenbaum], il est possible d'envisager la programmation exploratoire de deux façons différentes :

1. considérer le programme expérimental comme un prototype "jetable" (on développe par exemple, un prototype limité pour l'expérimenter avec l'interface utilisateur);

2. utiliser la méthodologie de programmation exploratoire pour la maintenance (car la maintenance est le coût dominant du *software engineering*).

Un but de la recherche actuelle sur le *knowledge-based software engineering* est de développer des outils pour supporter tous les niveaux de la conception d'un logiciel. Pour les outils pour l'évolution des spécifications, le premier pas est de catégoriser les types de changements qui sont le plus souvent faits aux spécifications. Le second pas

est de développer un éditeur de haut niveau qui peut faire ces changements à une spécification lorsqu'il est guidé par l'utilisateur. Il est à noter que l'évolution des spécifications est un type de réutilisation restreint : les modifications sont incrémentales. Il est donc important que les outils pour supporter cela, supportent la validation et la maintenance au niveau des spécifications.

3.3.4. Validation des spécifications

Une spécification valide décrit un logiciel d'exploitation qui satisfait aux besoins du client. La validation d'une spécification est nécessairement un processus interactif. Par contre, la vérification de la correction d'un programme peut être en principe réalisée automatiquement. Un programme est vérifié en prouvant que c'est une implémentation mathématique correcte d'une spécification formelle.

Dans bon nombre de projets de logiciel, les besoins sont négociés avec le client, et le document de spécification est alors réalisé et présenté à ce dernier pour être validé. Cependant, la validation réelle intervient seulement une fois le système implémenté et livré au client. Cette approche est fondamentalement risquée, car les révisions majeures sont beaucoup plus chères une fois le système implémenté, que pendant les premières phases du cycle de vie d'un logiciel. Or, des révisions sont inévitables car souvent, le client a seulement une vague idée du comportement désiré du système; et par ailleurs, l'ambiguïté propre au langage naturel peut amener des malentendus entre les demandes de l'analyste et le client.

Le prototypage est largement utilisé pour tester une conception avant la production. Puisque le problème posé par le *software* est un problème de conception, et non un problème industriel, le prototypage d'un logiciel a un but différent de celui du prototypage d'un *hardware*. Le prototypage d'un logiciel a pour objectif la validation d'une conception proposée, en construisant un système à bas prix qui possède assez de fonctionalités pour mettre à l'épreuve les décisions de conception majeures. Bon nombre de langages et d'environnements de l'intelligence artificielle sont bons pour le

prototypage rapide d'un logiciel, car ils ont des structures de très haut niveau et fournissent de bonnes facilités d'interaction [Bar, Cohen&Feigenbaum].

3.3.5. Maintenance des spécifications

L'étape la plus coûteuse du *software engineering* est la maintenance d'un système après sa mise en service. La maintenance d'un logiciel est le processus de modification du logiciel opérationnel existant, tout en laissant intactes ses fonctions principales [Van Vliet]. Actuellement, la maintenance constitue plus de la moitié des ressources du *software engineering* et elle est réalisée par la modification directe du code. L'effort majeur dans la maintenance est la compréhension d'un programme. Cela impose d'une part, de reconstruire l'information de conception perdue durant l'implémentation et d'autre part, de savoir que la modification d'un programme peut avoir des effets non souhaités.

Le but de la recherche en intelligence artificielle en ce qui concerne la compréhension d'un programme, est de retrouver une spécification de haut niveau à partir du code source. La maintenance inclut les activités suivantes [Bar, Cohen&Feigenbaum] et [Van Vliet] :

- la maintenance *corrective*, pour réparer les erreurs rencontrées;

- la maintenance *perfective*, pour améliorer la qualité, l'efficacité et la fiabilité d'un logiciel d'exploitation, ou pour améliorer son interface utilisateur;

- la maintenance *adaptive*, pour adapter un logiciel aux changements dans l'environnement. La maintenance adaptive ne conduit pas à des changements dans les fonctionalités du système;

- la maintenance *préventive*, qui concerne les activités visant à accroître la réalisation de la maintenance du système, telles que la mise à jour de la

documentation, l'ajout de commentaires, l'amélioration de la structure modulaire du système.

Il est à noter que les activités "réelles" de maintenance - la correction des erreurs - constituent 25 % de l'effort total de maintenance seulement. La moitié de l'effort de maintenance concerne les changements dus aux besoins changeants des utilisateurs, tandis que les 25 % restant concernent en grande partie l'adaptation du logiciel aux changements dans l'environnement externe.

Il faut également se rappeler que l'on estime que le coût total de maintenance comprend au moins 50 % des coûts du cycle de vie total.

Ces données reflètent la situation en 1970. Des études plus récentes ont montré que la situation n'avait pas changé en bien. La distribution relative des activités de maintenance est à peu près la même.

Les changements dans l'environnement du système et dans les demandes du client sont inévitables. Le logiciel modélise une partie de la réalité, et la réalité change, que nous le voulions ou pas. Le logiciel doit donc aussi changer. Il doit évoluer. Un large pourcentage de ce que nous appelons habituellement la maintenance est en fait de l'évolution [Van Vliet].

Les deux premières activités, à savoir la maintenance corrective et la maintenance perfective, sont partiellement abordées par la recherche en synthèse de programme, qui a pour objectif principal la technologie de dérivation de code correct et efficace à partir de spécifications formelles. Si cette recherche est couronnée de succès, la maintenance sera élevée au niveau des spécifications, puisqu'un code correct et efficace peut être redérivé à partir d'une spécification modifiée.

La maintenance adaptive quant à elle, est grandement facilitée si les mises à jour sont faites au niveau des spécifications plutôt qu'au niveau du code. En fin de compte, avec des outils appropriés et avec des spécifications suffisamment proches de la

compréhension conceptuelle d'un problème, les utilisateurs finaux eux-mêmes seront capables de réaliser la maintenance adaptive. De petits changements dans les fonctionnalités désirées d'un système ont pour résultat de petits changements dans la spécification. Cependant, de petits changements dans la spécification peuvent avoir pour résultat de grands changements du code. Pour cette raison, permettre de réaliser la maintenance directe des spécifications a pour résultat de larges gains dans la productivité du développement d'un logiciel, en diminuant l'effort requis par la maintenance adaptive.

Tout comme les programmes, les spécifications sont des *artefacts* conçus. La réitération successive de la conception durant la maintenance adaptive requiert la compréhension du raisonnement de conception précédent et des ramifications de chaque changement proposé. Un assistant pour la maintenance des spécifications permet de garder un enregistrement des décisions de conception des spécifications et des liens de justification pour les décisions de conception [Bar, Cohen&Feigenbaum].

Chapitre 4 : Exemple de système d'acquisition de spécification et lien avec le système EISI

4.1. Introduction

Ce chapitre est consacré à une présentation d'un système d'acquisition de spécification : IDeA. Nous verrons tout d'abord le paradigme sous-jacent à IDeA, les choix de représentation, les différents composants d'IDeA et enfin son fonctionnement. Nous serons alors à même d'établir les caractéristiques communes d'IDeA avec le système EISI.

4.2. IDeA[1]

4.2.1. Introduction

Depuis quelques années, il est apparu que les modèles du cycle de vie traditionnel n'étaient pas adéquats pour servir de base au développement d'un logiciel.

[1] Intelligent Design Aid

Rappelons brièvement qu'un certain nombre de problèmes apparaissent au cours du développement d'un logiciel [HarandiI&Lubars86] :

- un long délai s'écoule entre la définition des besoins et le moment où le logiciel est fourni;

- les difficultés de communication creusent un grand fossé entre l'utilisateur final et l'analyste;

- les exigences (besoins) ne peuvent pas toujours être très clairement exprimées à l'avance (pour certains systèmes par exemple, il y a des entrées inconnues, ce qui rend difficile l'expression fine des besoins);

- les besoins ne restent pas stables durant toute la période de développement du logiciel.

C'est pourquoi, de nouveaux paradigmes (basés, par exemple, sur des techniques d'inférence) sont apparus, pour essayer de résorber ces problèmes.

Le paradigme sous-jacent à IDeA - le paradigme du raffinement - est centré sur l'interaction entre un analyste, qui formule les exigences et les spécifications, et un développeur, qui construit le système désiré à partir de ces spécifications. Ce paradigme a deux objectifs : premièrement, faciliter la formulation, la communication et la compréhension des spécifications; deuxièmement, faciliter le développement du système grâce à un support orienté connaissance [Harandi&Lubars86], [Harandi&Lubars87].

4.2.2. Paradigme du raffinement

Le paradigme du raffinement est basé sur la philosophie suivante : le processus de développement complet d'un logiciel (et en particulier, les processus de spécification

et de conception) peut être considéré comme une succession de raffinements orientés connaissance, du modèle initial jusqu'au modèle uniforme du système désiré.

Ce paradigme vise essentiellement à réduire l'effort de communication entre l'analyste et le développeur [Harandi&Lubars86], [Harandi&Lubars87].

4.2.3. Choix de la représentation

Bien que le choix de la représentation d'un modèle ne fasse pas partie intégrante du paradigme, il est cependant important afin de mettre en oeuvre les concepts sous-jacents à IDeA (réutilisation, raffinement, etc.). La représentation choisie dans IDeA est le *dataflow*. Pour [Harandi&Lubars86], le *dataflow* répond, en effet, aux différents critères exigés pour une représentation :

- la représentation doit supporter toutes les activités du développement d'un logiciel : des spécifications au prototypage, en passant par la conception;

- la représentation doit pouvoir faciliter le travail de l'analyste dans son effort de formulation des spécifications;

- la représentation doit pouvoir faciliter la compréhension des spécifications par le développeur;

- la représentation doit être telle que l'analyste puisse examiner la conception de manière à pouvoir ainsi évaluer la qualité de ses spécifications;

- la représentation doit, enfin, pouvoir fournir un modèle uniforme à travers tout le processus de développement du logiciel.

4.2.4. Utilisation de concepts orientés domaine

L'objectif principal des concepteurs d'IDeA, [Harandi&Lubars86], était de développer un outil de support à la réutilisation de conceptions de logiciels. Cependant, afin de faciliter l'utilisation de ce système, d'autres objectifs sont venus s'ajouter :

- réduire l'effort de formulation de l'analyste, c'est-à-dire que l'on veut qu'il puisse exprimer les spécifications du système d'une manière qui ressemble le plus possible à ses modèles conceptuels;

- enrichir les informations transmises au développeur par le canal des spécifications.

Ce sont les raisons pour lesquelles les concepteurs d'IDeA ont choisi d'utiliser des concepts orientés domaine. De plus, ceux-ci sont typiquement les plus concis pour décrire un système et fournir de l'information contextuelle, qui peut être utilisée pour compléter des détails manquants.

Les spécifications données par l'utilisateur seront donc orientées domaine. Cela facilite le processus de compréhension car il concentre l'attention du développeur uniquement sur les parties pertinentes de la base de connaissance (cfr. infra).

De même, la base de connaissance inclut des informations orientées domaine, ce qui permet à l'analyste et au développeur de partager un modèle du monde commun.

4.2.5. Les composants de IDeA

IDeA possède une base de connaissance, qui contient deux éléments principaux : le *catalog of design components* et le *data dictionary*. Ces deux éléments sont organisés en une hiérarchie d'abstraction qui permet l'héritage et le partage de composants usuels.

La base de connaissance contient aussi des dictionnaires de propriétés de données et des prédicats usuels. Les propriétés et les prédicats sont organisés selon une hiérarchie d'abstraction à travers les relations de super/sous classes. [Harandi&Lubars86], [Harandi&Lubars87], [Harandi&Lubars89].

a) Le "catalog of design components"

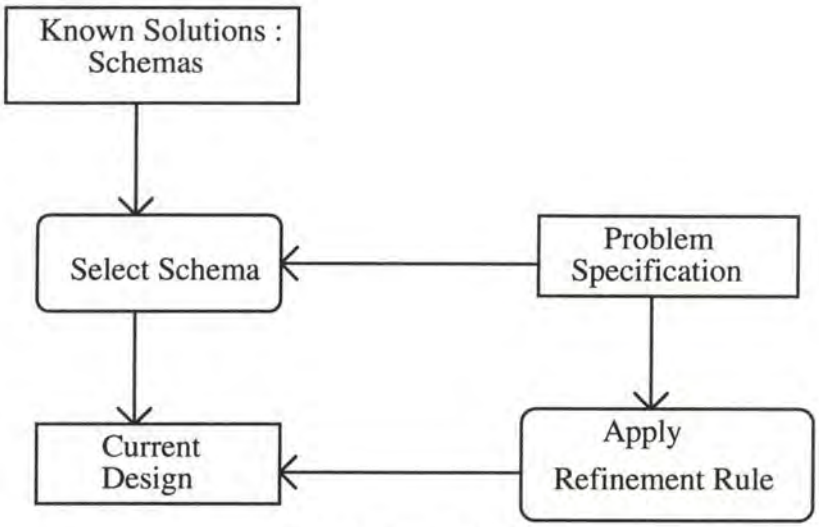
Le *catalog of design components* contient des schémas de conception (*design schemas*). Ceux-ci représentent une transformation de *dataflow* et une collection de *dataflow* d'entrée/sortie. Ces *dataflow* décrivent l'information sur les caractéristiques des données impliquées dans la transformation. Cette information se compose en réalité de deux éléments : une information "type de donnée" et les propriétés des données. L'information "type de donnée" consiste d'une part en un pointeur dans le *data dictionary* et d'autre part en information contextuelle. Cette information contextuelle permet "d'expliquer" au système IDeA comment certaines décisions de conception peuvent contraindre d'autres décisions dans le modèle du diagramme de flux. Les propriétés sur les diagrammes de flux décrivent des relations d'ordre et de "filtrage" tout en respectant certains prédicats.

Les schémas de conception contiennent aussi des règles de raffinement et de spécialisation.

b) Le "data dictionary"

Il contient les définitions de données d'objets orientés domaine à différents niveaux d'abstraction. Des valeurs initiales que l'utilisateur peut changer, des valeurs par défaut et des contraintes sont associées à ces définitions de données. Les valeurs par défaut, ainsi que les contraintes, sont des prédicats arbitraires qui spécifient des hypothèses par défaut et des contraintes absolues sur les données.

4.2.6. Fonctionnement de IDeA



Architecture d'IDeA

L'analyste formule les exigences et les spécifications de son système en termes d'entrée, de sortie ainsi qu'une description fonctionnelle à un haut niveau et dans une forme orientée domaine.

IDeA utilise ces spécifications pour trouver dans la base de connaissance (*catalog of design components*) les schémas appropriés qui décrivent la solution abstraite du problème. IDeA construit alors un modèle du système, de haut niveau, à partir des schémas localisés. IDeA le raffine ensuite successivement en utilisant des règles de raffinement de la base de connaissance et des spécifications supplémentaires fournies par l'analyste[2]. Enfin, une description détaillée du modèle est produite. Ce modèle peut alors être optimisé par des règles de transformation, et être exécuté comme prototype du système ou codé dans un langage de programmation particulier.

[2] Cela signifie que l'analyse et l'implémentation se déroulent en parallèle : en conséquence, l'implémentation peut être utilisée pour valider les spécifications fournies, et pour donner un *feed-back* immédiat à l'analyste à propos de l'exactitude et de la complétude de ses spécifications.

Examinons un peu plus en détail le processus de sélection des schémas de conception [Harandi&Lubars86], [Harandi&Lubars87]. Ce processus est composé de quatre phases :

1. Dans la première étape, il s'agit de trouver les types de donnée des objets spécifiés. Les types de donnée et les noms d'objets correspondent à des ensembles de contraintes. La description de l'utilisateur sera "mappée" avec un ensemble de contraintes. Cet ensemble est ensuite réduit à un seul ensemble qui correspond à un type de donnée particulier.

2. La seconde étape du processus de sélection consiste à trouver les schémas candidats. L'objet spécifié par l'utilisateur permet de "construire" une liste de candidats schémas, qui pourraient générer cet objet. IDeA compare alors la description de la fonction système (donnée par l'utilisateur) avec la description fonctionnelle de chaque schéma. Suite à cette comparaison, la liste sera réduite jusqu'à trouver une liste de schémas qui correspondent le mieux à la description de l'utilisateur.

3. La troisième étape consiste à vérifier la cohérence des contraintes du schéma. Cette étape est nécessaire car il est possible que les spécifications d'un objet correspondent à l'objet d'un schéma, mais que les spécifications ne satisfont pas les contraintes du schéma. Un candidat schéma sera jugé sur sa capacité à unifier les objets spécifiés par l'utilisateur avec ceux du schéma.

4. La dernière étape consiste à sélectionner le meilleur schéma, c'est-à-dire celui qui égale la plupart des objets spécifiés par l'utilisateur. Une stratégie d'évaluation est utilisée à cet effet. Dans le cas où plusieurs schémas auraient la même évaluation, ils sont montrés à l'utilisateur pour qu'il choisisse lui-même le meilleur schéma.

4.2.7. Exemple utilisant IDeA

Avant d'aborder les liens entre les systèmes IDeA et EISI, il nous a semblé utile de donner un exemple d'utilisation d'IDeA. Cet exemple est tiré de [Harandi&Lubars87].

Le problème est de mettre à jour les enregistrements d'étudiants avec leur grade. De plus, on veut que la conception puisse permettre de déterminer quels étudiants ont raté et lesquels doivent être mis à l'essai. On veut également préparer les lettres d'avis pour ces étudiants.

Afin de maximiser le potentiel de réutilisation, la conception a été abstraite pour s'appliquer à tous les types de *réussite* (abstraction de *réussite scolaire*), en créant une *famille de conception* plus grande. Le schéma de conception correspondant est représenté par la figure 1. IDeA donne de l'information sur les diagrammes de flux et les transformations de diagrammes de flux. De plus, les contraintes des schémas et les variables de contrainte partagées sont également montrées.

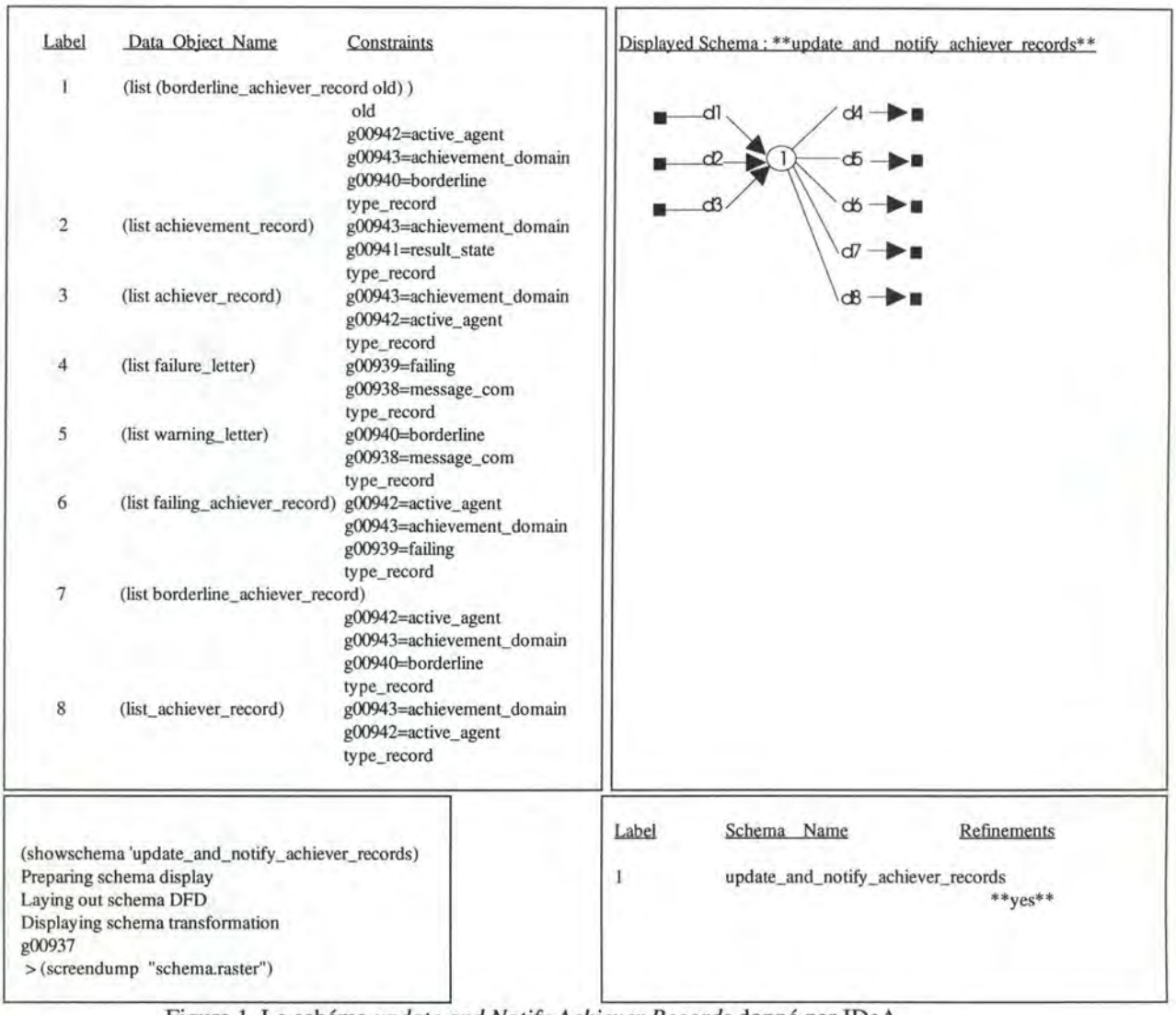


Figure 1. Le schéma *update and Notify Achiever Records* donné par IDeA

L'utilisateur commence à construire une conception en spécifiant les entrées, les sorties et les fonctions du système. Cette spécification est utilisée par le processus de sélection de schémas pour construire un modèle de haut niveau du système désiré. L'utilisateur a spécifié (figure 2)

- un diagramme de flux en entrée comme une liste d'*enregistrements d'étudiants*;

- un diagramme de flux en sortie comme des listes de *lettres d'échec* et de *lettres "frontière"*;

- une fonction qui met à jour et notifie les enregistrements d'étudiants.

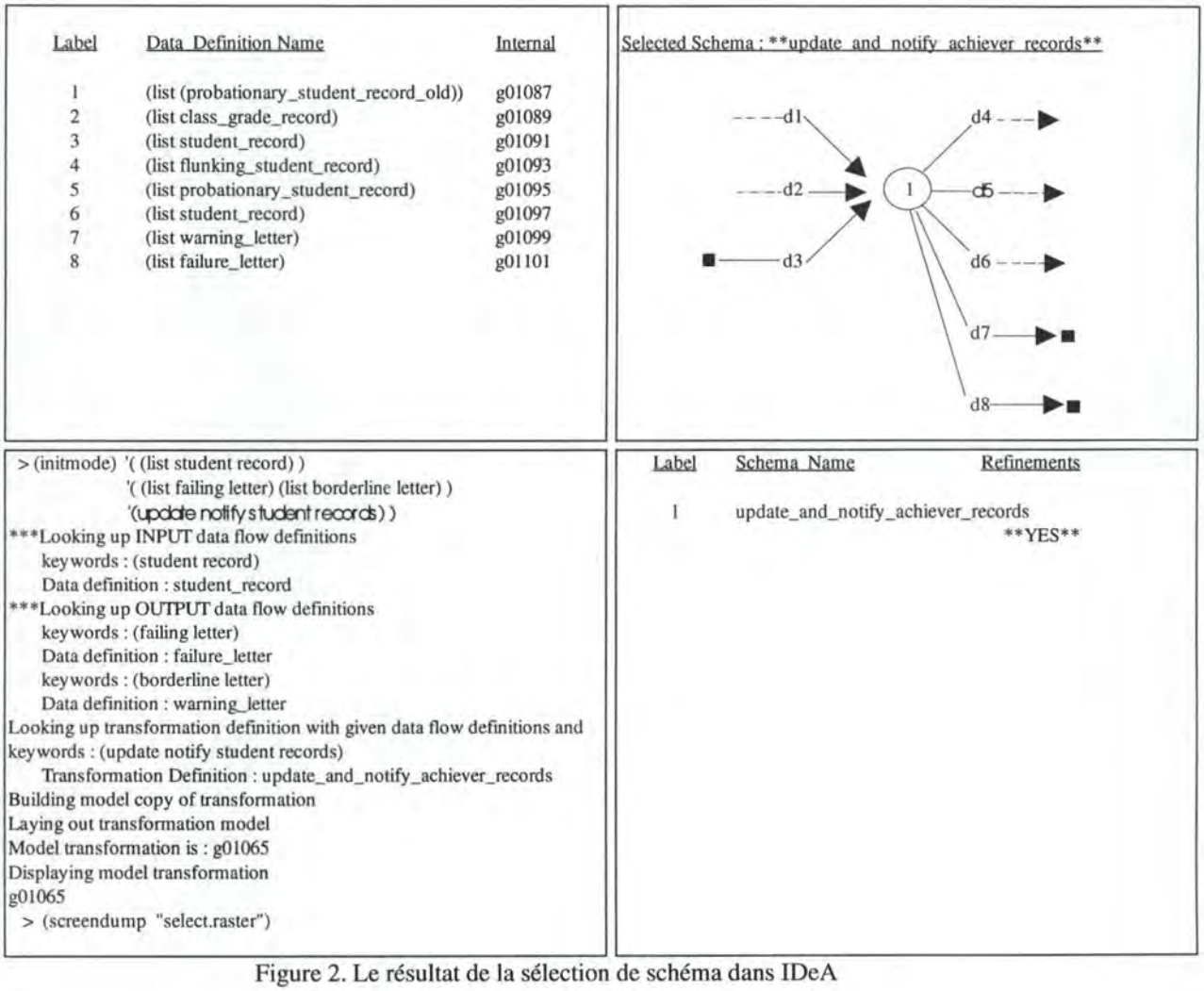


Figure 2. Le résultat de la sélection de schéma dans IDeA

IDeA interprète les diagrammes de flux comme décrivant des *enregistrements d'étudiants*, des *lettres d'échec* et des *lettres d'avertissement*. Ces types de données permettent à IDeA de sélectionner le schéma *Mise à jour et notification d'enregistrements de réussites* et de l'unifier avec les spécifications de l'utilisateur.

Le type de donnée *enregistrements d'étudiants* est suffisant pour établir la *réussite scolaire* comme *domaine de réussite*, qui est partagé par tous les diagrammes de flux dans le schéma. En conséquence, tous les types de données vont être spécialisés dans le contexte de *réussite scolaire*. Les trois diagrammes de flux spécifiés par l'utilisateur vont être totalement expliqués par le schéma. Cinq autres diagrammes de

flux vont être suggérés par le schéma. Cela peut signifier une incomplétude dans les spécifications de l'utilisateur ou de l'information non désirée dans le schéma.

Puisque le schéma *Mise à jour et notification d'enregistrements de réussites* possède une règle de raffinement applicable, IDeA l'applique pour obtenir une conception raffinée. Le schéma raffiné est montré par la figure 3. IDeA continue alors à appliquer les règles de raffinement et de spécialisation applicables pour les composants de conception plus détaillés, jusqu'à ce qu'une conception complète soit produite. Lorsque c'est nécessaire, IDeA demande à l'utilisateur de l'information supplémentaire. Cela peut lui permettre de faire une distinction entre des spécialisations concurrentes.

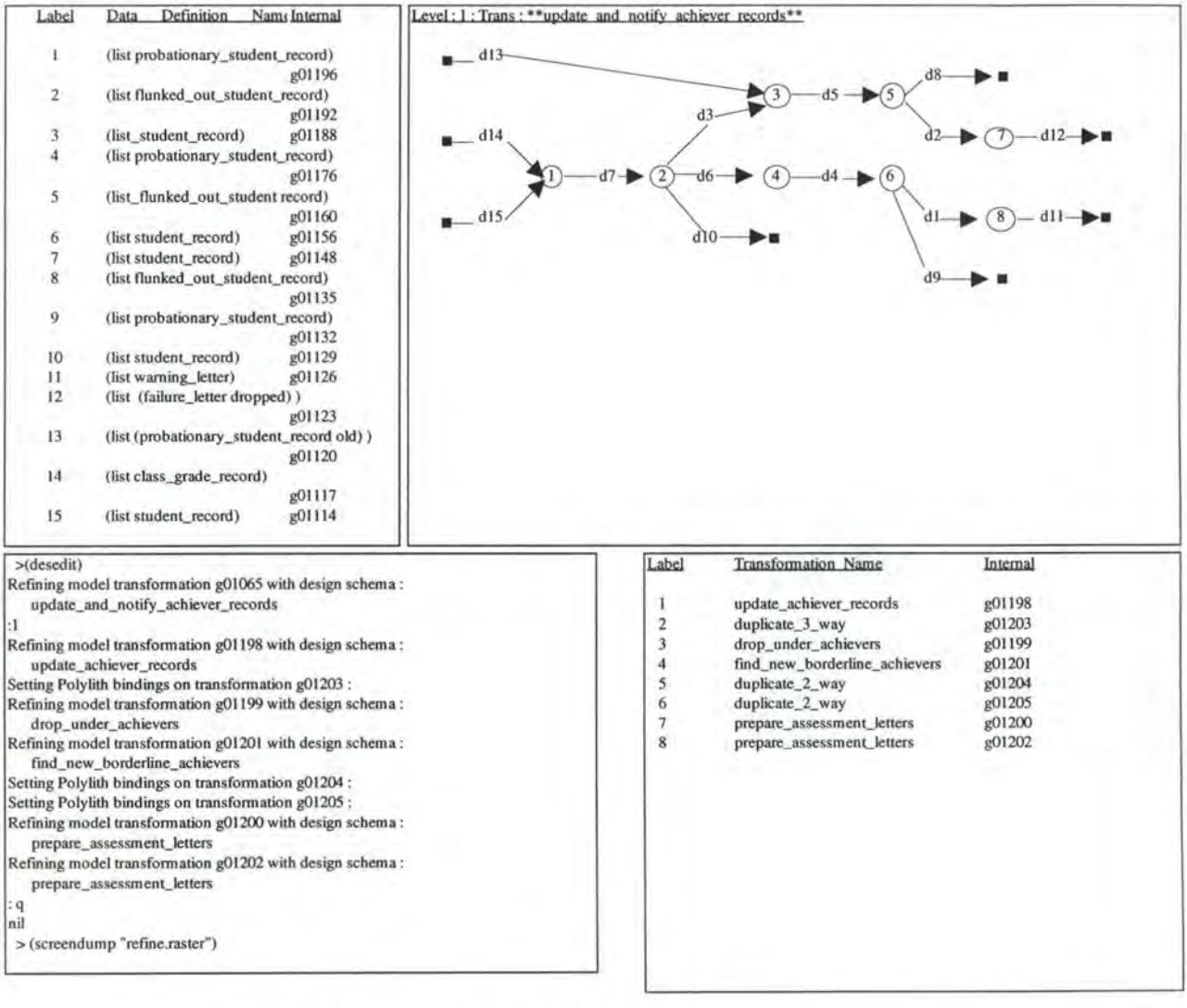


Figure 3. Application d'une règle de raffinement dans IDeA

4.3. Lien entre IDeA et le système EISI

4.3.1. Rappel

Dans le premier point de ce chapitre, nous avons présenté le système IDeA et son fonctionnement. IDeA est un système pour l'acquisition des spécifications, qui aide l'utilisateur à fournir des spécifications complètes et cohérentes.

Le système EISI, quant à lui, n'est pas un système d'aide à l'acquisition des spécifications. Comme nous l'avons vu dans le chapitre 2, ce système permet d'une part, la gestion (acquisition, représentation, manipulation) des métadonnées et d'autre part, la résolution de problèmes statistiques posés par l'utilisateur. Rappelons qu'en outre, le système aide l'utilisateur dans la formulation de son problème et l'interprétation de sa solution.

Ces deux systèmes sont donc très différents puisqu'ils n'ont pas le même cadre de travail ni les mêmes objectifs. Néanmoins, il est possible de dégager certaines idées communes à ces deux systèmes. C'est ce que nous ferons dans la section suivante.

4.3.2. Les caractéristiques communes[3]

Dans cette section, nous allons mettre en évidence les points communs que nous avons repérés entre le système EISI et IDeA.

[3] Signalons que le système EISI a été construit sans avoir connaissance de IDeA.

a) La base de connaissance

La base de connaissance d'IDeA, comme nous l'avons vu dans le point précédent, contient des schémas de conception et des définitions d'objets orientés domaine. Dans EISI, elle comprend des scénarios ainsi que des connaissances sur le domaine d'application de problèmes. L'idée sous-jacente à l'utilisation de tels types d'information est de travailler avec des concepts de haut niveau et un degré d'abstraction élevé.

b) Les schémas prédéfinis

Dans les deux systèmes, nous pouvons constater l'emploi de schémas prédéfinis: *design schemas* dans IDeA et *scenarios* dans EISI. Ceux-ci sont utilisés pour acquérir une description d'un problème à un très haut niveau. L'intérêt de l'utilisation de schémas prédéfinis se marque à deux niveaux : d'une part, cela indique une volonté d'avoir des spécifications de haut niveau; d'autre part, cela permet une plus grande réutilisation. En effet, les *design schémas* permettent d'augmenter le niveau de réutilisation des spécifications; les scénarios ont pour avantage de ne pas devoir réécrire plusieurs fois la solution de problèmes très semblables.

c) La sélection

Dans IDeA, la sélection d'un schéma se fait par la technique du raffinement successif. Comme nous l'avons expliqué dans le point précédent, le système va comparer la description d'objets de l'utilisateur avec celle des schémas prédéfinis, vérifier le respect des contraintes afin de sélectionner le meilleur schéma.

Nous trouvons dans EISI une technique de sélection sensiblement la même : la sélection du schéma approprié au type de problème se fait par comparaison avec des *goals* de manière à trouver celui qui donnera la solution. Ici également, le système

constitue une liste de schémas susceptibles de résoudre le problème et la réduit progressivement.

d) Aide à l'utilisateur

Nous pouvons qualifier ces deux systèmes d'outils actifs. En effet, IDeA permet à l'utilisateur de vérifier l'exactitude et la cohérence de ses spécifications, puisque l'implémentation peut servir à valider celles-ci. Dans EISI, le *Domain Assistant* aide l'utilisateur dans la formulation de son problème. Nous voyons là une volonté de passer des outils passifs à des outils actifs.

4.3.3. Conclusion

Comme nous l'avons déjà rappelé dans le point 4.3.1., les deux systèmes diffèrent par leurs objectifs et de ce fait, ne s'adressent pas aux mêmes personnes. IDeA est un système qui s'adresse essentiellement aux développeurs de logiciels (seuls ceux-ci d'ailleurs sont capables d'exprimer les spécifications dans un langage ou une notation qui emploient certains formalismes comme la théorie du modèle relationnel, le concept d'invariant, etc.). EISI est destiné à des personnes ayant une expérience de l'analyse de l'information, au niveau pratique, en statistiques officielles, mais qui ne s'y connaissent pas nécessairement dans le domaine du Logement et de la Construction.

Cependant (nous l'avons vu dans le point précédent), ils ont des points communs et par là, rejoignent bien des objectifs du *Knowledge-based software engineering* (cfr. chapitre 3) : utilisation de techniques - dans ces deux cas, le raffinement successif - afin d'extraire la connaissance de la base; validation des spécifications et aide à l'utilisateur dans la formulation de son problème; enfin, emploi de schémas prédéfinis pour permettre une plus grande réutilisation du travail déjà effectué.

Chapitre 5 : Conception et implémentation du générateur d'application statistique

5.1. Introduction

Ce chapitre présente la conception et l'implémentation du générateur d'application statistique. Nous exposerons tout d'abord le but du travail, les objectifs poursuivis, la méthodologie suivie ainsi que le langage utilisé afin de le réaliser. Nous étudierons alors les deux structures d'EISI qui nous intéressent particulièrement, à savoir les scénarios et les opérateurs. Nous pourrons ainsi examiner les limites des représentations et repérer les diverses extensions à réaliser afin de construire un interpréteur puis un générateur de programmes autonomes. Le dernier point de ce chapitre propose l'implémentation de ces différentes extensions.

5.2. But

Le but du travail était de déterminer la faisabilité d'une génération automatique de programmes d'analyse statistique[1] à partir de scénarios et de cas développés dans le cadre du projet EISI. Il s'agissait ensuite d'étendre le système EISI pour permettre cette

[1] Un programme d'analyse statistique est un processus de décision basé sur les données.

génération. Dans ce cas, les différentes étapes logiques du processus de raisonnement sont les suivantes :

1. choisir le scénario correspondant au problème dont on a les données et le contexte;
2. générer le cas correspondant au scénario choisi;
3. examiner les résultats du cas

   si les résultats ne sont pas bons, on retourne à l'étape 1

   sinon on va à l'étape 4;
4. générer le programme à partir du scénario sélectionné;
5. exécuter le programme

   si l'exécution se passe bien, le processus s'arrête ici

   sinon on va à l'étape suivante;
6. examiner le cas et le programme afin d'expliquer ce qui a échoué.

C'est donc l'étape 4 de ce processus qui va nous intéresser particulièrement.

5.2.1. Objectifs et méthodologie

a) Objectifs

Les objectifs du travail étaient au nombre de trois :

- Dans un premier temps, nous avons examiné la possibilité et la faisabilité d'une génération automatique de programme d'analyse statistique. Pour ce faire, il fallait étudier les changements de la structure et les fonctionalités requises dans

le système EISI afin d'étendre le mécanisme de résolution de problème pour la génération de programme et pour l'interprétation.

- Dans un second temps, nous avons étendu le système EISI avec les changements nécessaires (dégagés de l'étape précédente).

- Enfin, nous avons testé le mécanisme de génération avec des exemples simples, utilisant la représentation de problème sous-jacente du système EISI comme base.

b) Méthodologie

Afin de réaliser les objectifs pré-cités, nous avons suivi la démarche suivante:

- Etudier les structures d'information utilisées dans EISI, pour la représentation de :

  - Usage Scenarios (comme structure de solution de problème)

  - Opérateurs (comme base de fonctionalité dans les spécifications)

- Déterminer la faisabilité des Usage Scenarios comme base d'une conception pour un programme d'analyse statistique.

- Déterminer quelles extensions seraient nécessaires aux structures d'Opérateur pour permettre l'inclusion de fonctionalité.

- Déterminer les extensions nécessaires dans EISI pour permettre l'interprétation des scénarios dans l'environnement.

- Implémenter et tester dans l'environnement le mécanisme du Case-Based Reasoning.

- Considérer les extensions nécessaires pour étendre l'environnement de résolution de problème EISI pour générer des programmes autonomes.

- Tester avec les scénarios étendus précédemment.

5.3. Langage utilisé

Dans le système EISI, un grand nombre de modules sont écrits en Lisp. Notre module lui-même est écrit en Lisp et génère du Lisp. Les raisons pour lesquelles le langage Lisp a été choisi sont nombreuses :

a) Lisp est plus flexible que la plupart des langages

Cela signifie qu'il est possible d'écrire un programme Lisp pour produire virtuellement tout comportement désiré, à partir de l'ordinateur.

b) Lisp est indéfiniment extensible

Cela signifie que si le programmeur a besoin de facilités non offertes par Lisp, il peut écrire un programme Lisp qui lui fournira cette facilité.

c) Lisp offre des facilités pour construire facilement de nouvelles structures

d) Lisp travaille avec des listes qui peuvent être utilisées pour représenter efficacement de l'information très compliquée.

e) Le système des objets est très fort

Si l'on définit une structure orientée objet, chaque objet peut avoir des fonctions qui ne sont applicables qu'aux entités de cette classe. Il est donc possible d'avoir une hiérarchie de types dans laquelle un objet hérite non seulement des structures mais aussi des méthodes de sa classe.

f) Pour construire un prototype, il est intéressant d'utiliser Lisp car c'est une méthode efficace pour valider les algorithmes et les fonctionalités en avant de l'implémentation.

Il faut malgré tout, signaler un inconvénient : le type des fonctions n'est déterminé qu'à l'exécution, ce qui est également le cas pour l'assignation des variables. Ainsi, l'erreur engendrée par la multiplication d'un nombre et d'une chaîne de caractères, n'apparaîtra qu'à l'exécution. Ce ne serait pas le cas en Pascal par exemple, puisque les types sont déclarés.

5.4. Structures EISI et fonctionalités

Dans cette section, nous allons passer en revue les différentes structures existantes dans le système EISI ainsi que leurs fonctionalités. De plus, nous présenterons également des fonctions manipulant les structures entières.

5.3.1. Scenarios[2]

a) Définition

Un scénario est la spécification d'un problème existant dans le domaine (dans ce cas, les statistiques du logement et de la construction) pour en trouver la solution. Rappelons, en d'autres termes, que le scénario décrit un problème et spécifie une ou plusieurs solutions partielles, c'est-à-dire décrit la méta-information nécessaire pour le problème et la séquence d'opérations à y appliquer (cfr. chapitre 3).

b) Structure

La structure d'un scénario se présente de la façon suivante :

- Name -- le nom du scénario
- Problem-features -- les caractéristiques principales du scénario
- Surface -- une description succincte du but du scénario
- Concept-Refs -- les références aux concepts
- Metadata-Refs -- les références aux métadonnées
- Global-Constraints -- les contraintes globales
- Schema -- les opérations à exécuter sur les concepts et les métadonnées

[2] L'implémentation de la structure et des méthodes des scénarios se trouve en annexe A.

c) Fonctionalités

Les méthodes (fonctions) suivantes gèrent l'accès aux différents champs d'une structure de scénario :

* *scen/get-name* : accède au champ "name" d'une structure de scénario donnée.

* *scen/get-problem-features* : accède au champ "problem-features" d'une structure de scénario donnée.

* *scen/get-surface* : accède au champ "surface" d'une structure de scénario donnée.

* *scen/get-concept-refs* : accède au champ "concept-refs" d'une structure de scénario donnée.

* *scen/get-metadata-refs* : accède au champ "metadata-refs" d'une structure de scénario donnée.

* *scen/get-global-constraints* : accède au champ "global-constraints" d'une structure de scénario donnée.

* *scen/get-schema* : accède au champ "schema" d'une structure de scénario donnée.

Les méthodes (fonctions) suivantes gèrent la modification des valeurs des champs d'une structure de scénario :

* *scen/set-name* : modifie ou remplit le champ "name" d'une structure de scénario donnée avec le nom donné.

* *scen/set-problem-features* : modifie ou remplit le champ "problem-features" d'une structure de scénario donnée, grâce au "goal" donné.

* *scen/set-surface* : modifie ou remplit le champ "surface" d'une structure de scénario donnée avec la description donnée.

* *scen/set-concept-refs* : modifie ou remplit le champ "concept-refs" d'une structure de scénario donnée avec la liste donnée de références de concepts.

* *scen/set-metadata-refs* : modifie ou remplit le champ "metadata-refs" d'une structure de scénario donnée avec la liste donnée des références aux métadonnées.

* *scen/set-global-constraints* : modifie ou remplit le champ "global-constraints" d'une structure de scénario donnée avec la liste donnée des contraintes.

* *scen/set-schema* : modifie ou remplit le champ "schema" d'une structure de scénario donnée avec la liste donnée des opérateurs.

Fonction diverse :

* *is-scenario?* : vérifie que la structure passée en argument est bien une structure de scénario.

d) Fonctions manipulant la structure entière

Les fonctions suivantes se trouvent dans une librairie de gestion des scénarios : scenlib.lsp[3]. Ces fonctions gèrent les scénarios de façon globale.

* *store-scenario* : stocke dans une table le scénario donné.

* *fetch-scenario* : accède, dans la table des scénarios, au scénario dont on a donné le nom.

* *index-on-scenario-name* : renvoie la liste de tous les scénarios définis.

* *generate-scenario-index-entry* : crée un index pour le scénario donné.

* *generate-scenario-index* : crée un index de tous les scénarios dans la librairie.

* *clear-scenario-index* : efface l'index des scénarios.

* *scenario/stream-in* : lit un fichier donné, applique la fonction de création d'un scénario et stocke le résultat dans la table des scénarios.

5.3.2. Opérateurs[4]

Nous abordons dans cette section les opérateurs. Nous pensons en effet, que cela est utile puisque le schéma d'un scénario est composé d'une liste d'opérateurs.

[3] L'implémentation des fonctions de scenlib.lsp se trouve en annexe A.
[4] L'implémentation de la structure et des méthodes des opérateurs se trouve en annexe A.

a) Structure

La structure d'un opérateur se présente de la façon suivante :

- Name -- le nom de l'opérateur
- Parameters -- les paramètres associés à l'opérateur
- Interpretations -- interprétation du résultat possible de l'opérateur
- Synopsis -- description succincte de l'opérateur
- Explanation -- description détaillée de ce que fait l'opérateur

b) Fonctionalités

Les méthodes (fonctions) suivantes gèrent l'accès aux différents champs d'une structure d'opérateur :

* *op/name* : accède au champ "name" d'une structure d'opérateur donnée.

* *op/parameters* : accède au champ "parameters" d'une structure d'opérateur donnée.

* *op/interpretations* : accède au champ "interpretations" d'une structure d'opérateur donnée.

* *op/synopsis* : accède au champ "synopsis" d'une structure d'opérateur donnée.

* *op/explanation* : accède au champ "explanation" d'une structure d'opérateur donnée.

Les méthodes (fonctions) suivantes gèrent la modification des valeurs des champs d'une structure d'opérateur :

* *op/set-name* : modifie ou remplit le champ "name" d'une structure d'opérateur donnée, avec le nom donné.

* *op/set-parameters* : modifie ou remplit le champ "parameters" d'une structure d'opérateur donnée, avec la liste donnée des paramètres.

* *op/set-interpretations* : modifie ou remplit le champ "interpretations" d'une structure d'opérateur donnée, avec les interprétations données.

* *op/set-synopsis* : modifie ou remplit le champ "synopsis" d'une structure d'opérateur donnée, avec le texte donné de la description.

* *op/set-explanation* : modifie ou remplit le champ "explanation" d'une structure d'opérateur donnée, avec le texte donné de la description détaillée.

Les méthodes (fonctions) suivantes gèrent la création d'une structure d'opérateur:

* *op/new* : crée une nouvelle structure d'opérateur avec les arguments donnés.

* *op/create-from-list* : crée une nouvelle structure d'opérateur avec la liste donnée des arguments.

Fonction diverse :

* *is-op?* : vérifie que la structure passée est bien une structure d'opérateur.

c) Fonctions manipulant la structure entière

Les fonctions suivantes se trouvent dans une librairie de gestion d'opérateurs : opslib.lsp[5]. Ces fonctions gèrent les opérateurs de façon globale.

* *store-operator* : stocke dans une table l'opérateur donné.

* *fetch-operator* : accède, dans la table des opérateurs, à l'opérateur dont on a donné le nom.

* *index-of-operators* : renvoie la liste de tous les opérateurs définis.

* *print-all-operators* : imprime tous les opérateurs dans un fichier.

5.4. Extensions à l'environnement EISI

Avant d'aborder l'implémentation du générateur, il est nécessaire d'examiner les différentes extensions utiles. Nous verrons d'abord les limites de la représentation dans EISI. Ensuite, nous examinerons les extensions nécessaires à l'interprétation de programmes. En effet, avant de générer un programme à partir d'un scénario, il est d'abord indispensable d'interpréter le scénario. Nous verrons comment dans la suite de cette section. Nous pourrons alors nous pencher sur les extensions utiles pour la génération de programmes et celles exigées pour la génération de programmes autonomes.

[5] L'implémentation des fonctions de opslib.lsp se trouve en annexe A.

5.4.1. Limitations de la représentation dans EISI

Nous venons d'examiner la structure et les fonctionalités des scénarios et des opérateurs. A partir de là, nous pouvons constater que la représentation dans EISI est limitée aux structures conceptuelles et aux structures de métadonnées. En effet, nous pouvons tirer deux conclusions du point précédent :

- La première constatation est que les données sont uniquement représentées en termes de structure et de sémantique. En effet, seules des structures de métadonnées existent. Ceci est dû au fait que le système EISI avait pour objectif essentiel la manipulation des métadonnées.

- Au départ, le système EISI visait principalement la génération d'explications plutôt que les solutions de problèmes concrets. C'est la raison pour laquelle les seules fonctionalités des opérateurs ont pour objet leur structure. La partie fonctionnelle des opérateurs n'est pas présente.

5.4.2. Extensions exigées pour l'interprétation de programme

Durant le processus d'évaluation, il faut appliquer les formes fonctionnelles des opérateurs. Il sera donc utile d'étendre le *Case-Based Reasoner* afin de réaliser cette évaluation. Concrètement, cela implique la création d'une nouvelle structure : Operator Result (*Opres*). *Opres* est un objet qui a la même structure qu'un opérateur et qui est destiné à recevoir le résultat de l'évaluation d'un opérateur. En d'autres termes, cette structure devra stocker non seulement l'explication de chaque étape dans le plan de solution concrétisé dans le scénario, mais aussi les résultats intermédiaires de l'application de la forme fonctionnelle des opérateurs au programme. De plus, il sera indispensable de développer la fonction capable de réaliser l'évaluation d'un opérateur : opeval.lsp.

5.4.3. Extensions exigées pour la génération de programme

Nous avons vu qu'avec le système EISI tel qu'il existe, il n'est pas possible de résoudre un problème avec des données concrètes. En effet, l'implémentation des opérateurs est inexistante.

Par conséquent, afin de pouvoir inclure le code nécessaire à l'exécution des opérateurs dans l'interprétation des plans du scénario, il est nécessaire d'étendre la représentation d'un opérateur dans le système EISI. En effet, dans l'ancien système, la représentation d'un opérateur est telle qu'elle ne contient pas de contraintes et de corps. Or, les contraintes sont nécessaires afin de vérifier l'applicabilité de l'opérateur aux arguments donnés (préconditions). Le corps, quant à lui, contient la spécification exécutable de l'opérateur.

Afin de pouvoir générer des programmes, il faut pouvoir générer le code mais également l'environnement, c'est-à-dire la description des objets nécessaires pour exécuter les programmes. Il serait par conséquent intéressant d'avoir un mécanisme pour stocker les représentations intermédiaires des fonctions et des opérateurs. Cela permettrait d'avoir la possibilité d'appeler et d'exécuter les opérateurs à partir d'une structure de contrôle. Ce mécanisme est celui des scripts. Un script peut se définir comme étant la partie d'une application générée et évaluée dans un environnement indépendant du système (nous pouvons dire qu'un script est une sorte de fonction). La seconde extension consiste donc à créer la structure de script, les fonctions permettant la manipulation de cette structure et enfin, les fonctions de génération de ces scripts.

Enfin, la dernière extension dans le contexte spécifique de génération de programmes d'analyse statistique, est d'introduire la notion de représentation d'objets de données statistiques. Il sera également nécessaire de définir les opérateurs pour les opérations statistiques sur ces objets.

5.4.4. Extensions exigées pour la génération de programmes autonomes

Nous voulons pouvoir générer des programmes autonomes car il nous semble important qu'ils puissent être exécutés dans un environnement indépendant de celui dans lequel ils ont été générés. En effet, beaucoup de personnes pourraient être intéressées par ce système mais ont déjà un autre environnement de travail. Il faudrait donc pouvoir leur offrir des programmes autonomes.

Afin de réaliser cet objectif, deux approches sont possibles :

1. la génération de programmes Lisp autonomes, avec tous les objets et les fonctionalités nécessaires pour exécuter le programme extérieurement à l'environnement EISI.

Le premier avantage de cette approche est que la sémantique est connue à l'avance. En effet, avec Lisp, on a le contrôle de l'environnement, on sait comment le programme va s'exécuter. Le second avantage est que cette génération ne dépend pas de logiciel de troisième génération.

Le principal désavantage de cette approche réside dans la taille importante et la complexité des programmes résultants.

2. la génération de programmes autonomes pour un logiciel de troisième génération, comme SAS ou SPSS, transformant des descriptions d'opérateurs à partir du langage de représentation utilisé dans EISI vers le langage utilisé dans le logiciel.

Le premier avantage de cette approche se situe au niveau pratique : cette solution est en effet plus utile dans le monde réel. Le second avantage est que ce n'est pas aussi complexe que la solution Lisp.

Les désavantages se situent à deux niveaux. Tout d'abord, il est difficile de mapper la sémantique des opérateurs EISI dans la syntaxe du langage cible. Ensuite, la représentation des concepts et des métadonnées est limitée dans les logiciels externes.

Nous avons choisi d'implémenter la première solution, c'est-à-dire la génération de programmes Lisp autonomes. En effet, il nous semblait raisonnable, dans un premier temps, de compléter le prototype, c'est-à-dire de générer du Lisp, avant d'aborder une génération de programmes dans un autre langage. De cette façon, il nous était possible de tester le mécanisme de génération rapidement afin, par la suite, de pouvoir l'adapter à la génération de programmes en SAS ou SPSS, par exemple.

5.4.5. Implémentation

Dans cette section, nous abordons la description plus complète des diverses extensions (présentées dans les sections précédentes) apportées au système EISI, ainsi que les algorithmes correspondants aux différentes fonctions nécessaires à l'interpréteur et au générateur de programme[6].

a) Création d'un système de scripts

Ce système de scripts est tout à fait nouveau par rapport à l'ancien système. Il est utile afin de fournir une séparation entre les formes exécutables des scénarios et des opérateurs. Ce système de scripts inclura le stockage des résultats de l'exécution d'un script ainsi que l'histoire de son exécution. Nous devons donc définir toutes les fonctions qui vont gérer les scripts, leur exécution, etc. :

* une variable globale (**scripts**) doit être définie : il s'agit d'une *hashtable* qui contiendra les scripts créés.

[6] L'implémentation en langage Lisp de toutes ces fonctions se trouve en annexe A.

* *store-script* : stocke dans la *hashtable* le script (*lambda form*) donné sous le nom donné.

* *get-script* : accède au script stocké sous le nom donné, dans la *hashtable*.

* *perform* : applique le script stocké sous le nom donné à une liste donnée de variables.

* *run-script* : exécute le script dont on a donné le nom (avec les variables données s'il y en a).

* *run-script-with-varlist* : exécute le script dont on a donné le nom avec les variables données.

* *index-of-scripts* : renvoie la liste de tous les noms de scripts présents dans la *hashtable*.

Certaines variables globales et fonctions ont été définies afin de permettre le support de l'histoire de l'exécution des scripts :

* la variable **script-history-mode** est un booléen qui indique le mode on/off pour l'exécution de l'histoire des scripts.

* la variable **execution-history** est une liste contenant des couples (nom du script, résultat).

* *reset-execution-history* : remet la variable globale **execution-history** à *false*.

* *script-history* : "renverse" la liste **execution-history**.

* *script-history-on* : met la variable **script-history-mode** à vrai.

* *script-history-off* : met la variable **script-history-mode** à faux.

* *history-running?* : teste la variable **script-history-mode** pour connaître le mode d'exécution de l'histoire des scripts.

* *log-script-execution* : ajoute un couple (nom du script, résultat) à la liste **execution-history**.

* *display-script-history* : affiche la liste de l'histoire de l'exécution des scripts si elle n'est pas vide.

* *all-results-for-script* : renvoie une liste de tous les résultats de l'exécution d'un script dont on a donné le nom.

Deux macros ont également été spécifiées pour définir les scripts et pour leur exécution :

* *script* : stocke un script après avoir créé la forme lambda avec les arguments donnés.

* -> : permet d'exécuter le script dont on a donné le nom avec les paramètres donnés s'il y en a.

b) Implémentation des extensions aux opérateurs

Ainsi que nous l'avons dit dans le point 5.4.3. , nous avons ajouté deux champs - *body* et *constraints* - à la structure d'un opérateur afin d'en avoir une description fonctionnelle et de pouvoir l'exécuter. Par conséquent, nous avons ajouté les fonctions d'accès correspondantes :

* *op/constraints* : accède au champ "constraints" d'une structure d'opérateur donnée.

* *op/body* : accède au champ "body" d'une structure d'opérateur donnée.

ainsi que les fonctions de modification de la valeur des champs :

* *op/set-constraints* : modifie ou remplit le champ "constraints" d'une structure d'opérateur donnée, avec la liste donnée des contraintes.

* *op/set-body* : modifie ou remplit le champ "body" d'une structure d'opérateur donnée, avec la liste donnée des fonctions.

Dans le point précédent, nous avons créé un système de scripts afin d'assurer la séparation entre les formes exécutables des scénarios et des opérateurs. Nous avons donc ajouté deux fonctions nécessaires à la génération de scripts à partir des opérateurs, ainsi qu'une fonction permettant l'impression d'un opérateur :

* *op/dump-load-form* : met les champs de la structure de l'opérateur dans une forme à partir de laquelle l'opérateur pourra être re-généré.

* *operator->script* : crée un script pour la partie fonctionnelle de l'opérateur.

* *print-operator* : imprime l'opérateur donné.

c) Implémentation de la structure d'Opres

Rappelons qu'une structure *Opres* est la même qu'une structure d'opérateur, mis à part qu'un *Opres* reçoit le résultat de l'évaluation d'un opérateur.

Nous trouvons donc des fonctions d'accès aux champs d'une structure *Opres* :

* *res/name* : accède au champ "name" d'une structure *opres* donnée.

* *res/parameters* : accède au champ "parameters" d'une structure *opres* donnée.

* *res/constraints* : accède au champ "constraints" d'une structure *opres* donnée.

* *res/body* : accède au champ "body" d'une structure *opres* donnée.

* *res/interpretation* : accède au champ "interpretation" d'une structure *opres* donnée.

* *res/synopsis* : accède au champ "synopsis" d'une structure *opres* donnée.

* *res/explanation* : accède au champ "explanation" d'une structure *opres* donnée.

ainsi que les fonctions de modification de la valeur des champs :

* *res/set-name* : remplit ou modifie le champ "name" d'un *opres* donné avec le nom donné.

* *res/set-parameters* : remplit ou modifie le champ "parameters" d'un *opres* donné avec la liste donnée des paramètres.

* *res/set-constraints* : remplit ou modifie le champ "constraints" d'un *opres* donné avec le résultat donné de l'évaluation des contraintes.

* *res/set-body* : remplit ou modifie le champ "body" d'un *opres* donné avec le résultat donné de l'évaluation du corps d'un opérateur.

* *res/set-interpretation* : remplit ou modifie le champ "interpretation" d'un *opres* donné avec l'interprétation donnée.

* *res/set-synopsis* : remplit ou modifie le champ "synopsis" d'un *opres* donné avec le texte donné du synopsis.

* *res/set-explanation* : remplit ou modifie le champ "explanation" d'un *opres* donné avec le texte donné de l'explication.

Nous avons également écrit deux fonctions de création d'un *opres* :

* *res/new* : crée une nouvelle structure d'*opres* avec les arguments fournis.

* *res/create-from-list* : crée une nouvelle structure d'*opres* avec la liste des arguments fournis.

Fonctions diverses :

* *is-opres?* : vérifie que la structure donnée est bien une structure d'*opres*.

* *res/dump-load-form* : met les champs de la structure d'un *opres* dans une forme à partir de laquelle l'*opres* pourra être re-généré.

d) Génération de programmes Lisp autonomes

Maintenant que ces extensions ont été effectuées, il est possible de construire le générateur de programmes. Pour pouvoir le réaliser, nous avons travaillé en deux étapes : nous avons d'abord construit un interpréteur, puis un générateur. Ceux-ci travaillent uniquement sur les métadonnées. Dans un premier temps, les données concrètes ne seront donc pas prises en compte. En effet, nous voulons d'abord tester les mécanismes d'évaluation d'un scénario et de génération avant d'aborder le problème des données concrètes. Les sections suivantes expliquent la réalisation de ces deux étapes (interpréteur, générateur).

1° Interpréteur : sceneval

Lors de cette phase, nous avons réalisé les fonctions permettant l'évaluation du schéma du scénario donné. Cette évaluation consiste à exécuter les opérations décrites dans le schéma du scénario. Afin de mener à bien cette tâche, il est nécessaire d'accéder aux informations concernant chaque métadonnée et chaque concept référencés dans le scénario. En résumé, nous pouvons dire que l'interpréteur réalise trois opérations :

1. lecture et stockage des concepts
2. lecture et stockage des métadonnées
3. interprétation du schéma du scénario.

- **La mémoire de travail**

Afin de stocker les concepts et les métadonnées, nous avons introduit une mémoire de travail. La raison de ce choix est la suivante : la mémoire de travail va permettre de gérer de façon uniforme des variables locales, indépendamment de l'environnement général. De plus, il peut être important d'avoir la possibilité de changer de mémoire de travail (il est par exemple possible d'avoir deux mémoires de travail et de

faire ainsi des comparaisons). La mémoire de travail est en réalité un contexte pour l'exécution.

Nous avons dès lors écrit une série de fonctions utiles pour gérer une mémoire de travail[7].

* *wm/new* : crée une nouvelle mémoire de travail, sans entrées.

* *wm/clear* : efface toutes les références d'une mémoire de travail.

* *wm/put* : met, dans une mémoire de travail, une valeur indexée sous une clé donnée.

* *wm/get* : accède dans une mémoire de travail, à la valeur associée à une clé.

* *wm/drop* : efface d'une mémoire de travail la clé et la valeur associée à la clé.

* *wm/params->bindings* : génère une liste des valeurs liées à chaque label dans une liste de paramètres.

* *wm/match* : applique un prédicat sur toutes les entrées dans une mémoire de travail et renvoie une liste des clés des variables mises à *True* lorsque le prédicat est appliqué.

* *wm/do-for-all-matches* : applique une fonction à toutes les entrées qui égalent le prédicat fourni.

* *wm/list-vars-and-vals* : génère une liste des noms des variables et des valeurs stockés dans une mémoire de travail.

[7] L'implémentation de ces fonctions se trouve en annexe A.

* *wm/load-from-list* : remplit une mémoire de travail à partir d'une liste de paires clé/valeur.

Avant d'entamer la construction de l'interpréteur, il reste à écrire la fonction qui va permettre d'évaluer un opérateur : opeval.

- **Opeval**

Cette fonction sert à évaluer un opérateur, dans le contexte d'une mémoire de travail. Elle renvoie une instance *opres* contenant le résultat de l'évaluation. Cette fonction reçoit en entrée une structure d'opérateur, une liste de paramètres ainsi qu'un contexte (mémoire de travail). Cette fonction s'exécute en :

- vérifiant que

  * la structure donnée est bien une structure d'opérateur et que

  * la liste des paramètres est bien une liste et que

  * la liste des paramètres n'est pas vide et que

  * le contexte donné est bien une mémoire de travail

- mettant à jour les variables locales

- remplissant les différents champs de la structure d'un *opres* avec les valeurs données par l'opérateur

- appelant et exécutant le script correspondant à l'opérateur

- remplissant les champs *constraints*, *body*, *explanation* et *interpretation* de la structure d'*opres* en fonction du résultat de l'exécution du script

- renvoyant la structure d'*opres* ainsi obtenue.

Nous disposons maintenant de tous les éléments nécessaires au développement de l'interpréteur.

- **Interpréteur**

Afin de générer un script d'application, il est nécessaire de créer le contexte (mémoire de travail) qui permettra l'exécution du code (script). C'est en résolvant les références aux concepts et aux métadonnées que nous pourrons créer ce contexte. Les deux fonctions suivantes réalisent ce travail :

* *Determine-concept-references (aScenario aContext)*

Cette fonction trouve, à partir d'un scénario et d'une mémoire de travail donnés, tous les concepts référencés dans le scénario et les stocke dans la mémoire de travail. Cette fonction s'exécute en :

- vérifiant que

1. le scénario donné a bien la structure d'un scénario et que

2. le contexte donné correspond bien à une mémoire de travail et que

3. il y a des concepts référencés dans le scénario

- pour chaque concept référencé dans le scénario

1. cherche la structure du concept

2. place la structure dans la mémoire de travail sous une variable clé

* *Determine-metadata-references (aScenario aContext)*

Cette fonction trouve, à partir d'un scénario et d'une mémoire de travail donnés, tous les concepts référencés dans le scénario et les stocke dans la mémoire de travail. Cette fonction s'exécute en :

- vérifiant que

  1. le scénario donné a bien la structure d'un scénario et que

  2. le contexte donné correspond bien à une mémoire de travail et que

  3. il y a des métadonnées référencées dans le scénario

- pour chaque métadonnée référencée dans le scénario

  1. accède à l'objet référencé

  2. place l'objet dans la mémoire de travail sous le nom de la variable assignée

Maintenant que le contexte est créé, nous devons évaluer le schéma du scénario. Cela signifie d'une part qu'il faut faire le lien entre l'opérateur du schéma et son implémentation sous forme de script, et d'autre part, qu'il faut fournir la possibilité de "capturer" l'exécution sous forme d'une structure *Opres*. Cela est réalisé par la fonction suivante :

* *Evaluate-schema (aScenario aContext)*

Cette fonction évalue, dans une mémoire de travail donnée, les opérations spécifiées dans le schéma du scénario, en évaluant les opérateurs correspondant à ces opérations. Cette fonction s'exécute en :

- vérifiant que

  1. le scénario donné a bien la structure d'un scénario et que

  2. le contexte donné correspond bien à une mémoire de travail et que

  3. le schéma du scénario n'est pas vide

- créant le contexte grâce aux fonctions Determine-concept-references et Determine-metadata-references

- initialisant la pile qui recevra les structures de résultat

- pour chaque opération dans le schéma du scénario

  1. faire la référence à l'opérateur

  2. évaluer l'opérateur

  3. retourner la structure de résultat *Opres*

  4. mettre la structure *Opres* sur la pile des résultats

- retournant la pile des résultats inversée.

2° Générateur : scengen

Lors de cette phase, nous avons écrit les fonctions permettant de générer une fonction exécutable à partir d'un scénario. Cette fonction sera anonyme pour l'environnement, c'est-à-dire qu'elle n'aura pas de nom dans l'environnement, mais pourra être exécutée dans ce dernier. Cette fonction contiendra toutes les informations relatives aux concepts et aux métadonnées : nous avons utilisé pour ce faire les deux fonctions définies lors de la première étape. La fonction contiendra aussi l'appel aux scripts permettant d'exécuter les opérateurs du schéma. Tout cela est réalisé par les trois fonctions suivantes :

* *scenario->script (aScenario)*

Cette fonction crée une fonction lambda en créant une liste du contenu du contexte et en l'unifiant avec les appels aux scripts voulus. Cette fonction s'exécute en :

- vérifiant que le scénario est bien un scénario

- trouvant les concepts du scénario et les stockant dans le contexte (mémoire de travail) grâce à la fonction determine-concept-references de l'évaluateur

- trouvant les métadonnées du scénario et les stockant dans le contexte (mémoire de travail) grâce à la fonction determine-metadata-references de l'évaluateur

- créant la fonction lambda, c'est-à-dire :

  1. stockant dans un Let toutes les références du contexte

  2. pour chaque opération du schéma du scénario, faisant un appel au script par la fonction step-to-script-call

- stockant le script dans *scripts*.

* *step-to-script-call (aStep)*

Cette fonction transforme une opération du schéma du scénario en un appel au script correspondant. Cette fonction s'exécute comme suit :

- si aStep n'est pas vide

alors le remplacer par l'appel au script.

* *scenario->body (aScenario)*

Cette fonction génère une fonction lambda pour l'exécution du scénario, en créant une *closure* du contenu du contexte et en l'unifiant avec les appels aux scripts voulus. Cette fonction s'exécute en :

- vérifiant que le scénario est bien un scénario

- trouvant les concepts du scénario et les stockant dans le contexte (mémoire de travail) grâce à la fonction determine-concept-references de l'évaluateur

- trouvant les métadonnées du scénario et les stockant dans le contexte (mémoire de travail) grâce à la fonction determine-metadata-references de l'évaluateur

- créant la fonction lambda, c'est-à-dire :

1. stockant dans un Let toutes les références du contexte

2. pour chaque opération du schéma du scénario, faisant un appel au script par la fonction step-to-script-call.

Avec le générateur tel qu'il est présenté ci-dessus, nous avons simplement créé une fonction exécutable dans l'environnement dans lequel elle a été générée. Cependant, il serait plus intéressant de mettre le résultat de la génération dans une autre structure, proche de celle d'un scénario, ce qui permettrait de pouvoir l'exécuter dans un environnement indépendant. Nous allons pour cela créer la structure d'analyse.

L'objectif est donc le suivant : étant donné un scénario, générer l'application d'analyse Lisp équivalente pour les résultats. L'analyse doit être un objet, exécutable dans le contexte de l'interpréteur Lisp. En d'autres termes, le processus de génération doit être un *mapping* entre deux objets : une structure de scénario instanciée et une structure d'analyse créée pour le processus exécutable. L'analyse résultante doit être capable d'une part, d'être exécutée dans le contexte d'un environnement de raisonnement (et utiliser les fonctionalités du système EISI, sous la forme de scripts); d'autre part

d'être exécutée dans un environnement indépendant doté de librairies appropriées pour les fonctionalités.

Dans le point suivant, nous allons donc examiner la structure choisie pour une analyse ainsi que les fonctions qui y sont associées.

analysis

L'analyse, telle qu'elle sera générée, devra au minimum contenir les informations relatives aux concepts et métadonnées du scénario, ainsi que les fonctions nécessaires pour exécuter les opérations décrites dans ce scénario. En d'autres termes, une analyse est une structure de données qui contient les variables et le processus exécutable qui permet ainsi d'exécuter l'analyse. La structure d'une analyse se présente de la façon suivante :

- Name--le nom de l'analyse
- Parent--le nom du scénario à partir duquel l'analyse est dérivée
- Constants--les valeurs des constantes référencées dans l'espace de l'analyse
- Variables--les variables et valeurs référencées dans l'espace du système
- Result-Vars--les variables à prendre comme résultat de l'exécution d'une analyse
- Body--le code exécutable de l'analyse
- Description--une description donnée par l'utilisateur de l'analyse

Nous avons ensuite écrit les diverses fonctions gérant une structure d'analyse. Nous avons d'abord écrit une fonction permettant de créer une nouvelle structure d'analyse :

* *analysis.new* : crée une nouvelle structure avec les arguments éventuellement donnés.

Nous avons également une série de fonctions qui permettent d'accéder aux divers champs de la structure d'analyse :

* *analysis.get-name* : accède au champ "name" d'une structure d'analyse donnée.

* *analysis.get-parent* : accède au champ "parent" d'une structure d'analyse donnée.

* *analysis.get-constant* : accède au champ "constant" d'une structure d'analyse donnée.

* *analysis.get-variable* : accède au champ "variable" d'une structure d'analyse donnée.

* *analysis.get-body* : accède au champ "body" d'une structure d'analyse donnée.

Il y a aussi les fonctions permettant de modifier la valeur d'un champ d'une analyse :

* *analysis.set-name* : remplit ou modifie le champ "name" d'une structure d'analyse donnée, avec le nom donné.

* *analysis.set-parent* : remplit ou modifie le champ "parent" d'une structure d'analyse donnée, avec le parent donné.

* *analysis.set-constant* : remplit ou modifie le champ "constant" d'une structure d'analyse donnée, avec la valeur donnée.

* *analysis.set-variable* : remplit ou modifie le champ "variable" d'une structure d'analyse donnée, avec la valeur donnée.

Ayant défini la structure d'une analyse et les fonctions permettant d'y accéder et de la modifier, nous avons pu définir les fonctions permettant de générer et d'exécuter une analyse :

* *analysis.generate-body* : génère le corps d'une analyse à partir d'un scénario donné, en réalisant toutes les conversions nécessaires.

* *analysis.execute* : exécute toutes les activités couvertes par le schéma du scénario.

* *analysis.generate-and-execute* : génère une analyse à partir d'un scénario et l'exécute.

Nous avons également d'autres fonctions :

* *analysis.nominate-result* : permet d'ajouter une variable résultat dans l'analyse donnée.

* *analysis.print-results* : permet d'imprimer toutes les variables résultat d'une analyse donnée.

Avant d'entamer la génération de programmes autonomes, nous allons examiner comment, en théorie, construire un résolveur de problème. Cela permettra de comprendre la logique de construction du générateur de programmes.

e) Construction d'un résolveur de problème

1° Le modèle de l'espace de problème

Selon [Forbus&De Kleer], l'espace de conception d'un résolveur de problème se compose de cinq axes :

1. le modèle de connaissance : il répond à la question "comment est représentée la connaissance du système?" Il regroupe par exemple les objets, les attributs, etc.

2. le mécanisme de référence : il répond à la question "comment les procédures vont-elles chercher les données dont elles ont besoin pour s'exécuter?" Ce mécanisme peut, par exemple, être celui du *pattern matching*.

3. le modèle de procédure : c'est-à-dire comment les procédures sont décomposées et organisées. Ce sont par exemple, les règles If...Then.

4. la stratégie d'exécution : c'est-à-dire comment les procédures sont exécutées.

5. le modèle de dépendance : c'est-à-dire quelle information est enregistrée sur les relations entre les connaissances du système. Il s'agit, par exemple, de garder la trace des exécutions des règles.

Pour construire un résolveur de problème, il s'agit donc d'effectuer des choix au niveau de ces cinq axes.

D'autre part, il faut définir un domaine ou classe de problème sur lequel on va travailler. C'est ce que l'on appelle l'espace de problème. Celui-ci se compose d'une part d'un ensemble d'états qui représentent des configurations distinctes des objets et des

relations du domaine, et d'autre part, d'un ensemble d'opérateurs qui définissent la façon de passer d'un état à un autre.

Pour définir un problème dans un espace de problème donné, il faudra définir deux éléments : un état initial (c'est-à-dire un état distinct qui représente le point de départ dans l'espace) et un but (*goal*, c'est-à-dire une spécification du sous-ensemble de l'espace de problème qui peut servir de solution au problème).

Si l'on adopte ce modèle, résoudre un problème se fait en trouvant une séquence d'opérateurs, qui, lorsqu'ils sont appliqués à l'état initial, permet d'atteindre un état satisfaisant le but. La plupart du temps, ces séquences d'opérateurs sont générées par une recherche : par exemple, on essaye différents opérateurs sur l'état initial et récursivement, sur les résultats obtenus jusqu'à ce qu'un état satisfaisant le but soit trouvé.

Ce modèle de classe de problème est attrayant. En effet, pour résoudre un problème, il suffit de définir l'espace de problème et d'ensuite déclencher un moteur de recherche dessus. L'espace de problème peut se définir en développant une représentation des états et des opérateurs; formuler un problème consiste à définir l'état initial et à spécifier le but. Le seul problème qu'il reste à résoudre lors de la conception est de choisir la stratégie de recherche à utiliser.

2° Conception de la résolution d'un problème classique

La conception proposée par [Forbus&De Kleer] se compose de deux parties : une interface pour l'espace de problème et un mécanisme de recherche.

- **L'interface**

L'interface doit principalement donner la possibilité de manipuler les états et les opérateurs.

Pour les états, nous pouvons dégager trois manipulations :

* *goal detection* : permet de s'assurer si un état donné satisfait le but. Son importance est évidente.

* *state identity* : permet de détecter quand deux descriptions d'états réfèrent au même état. Il est important d'avoir ce mécanisme car il n'est d'aucun intérêt à avoir à réexaminer des états qui l'ont déjà été.

* *state display* : permet de produire une description lisible pour l'homme, d'un état donné. Bien que les résultats soient la plupart du temps utilisés par un autre programme, ce mécanisme est souvent intéressant et nécessaire pour le *débugging*.

En ce qui concerne les opérateurs, quatre manipulations sont, conceptuellement, nécessaires[8] :

* identifier quels opérateurs sont disponibles.

* déterminer si un opérateur donné est applicable à un état particulier.

* pour un état donné et un opérateur qui y est applicable, s'assurer que l'opérateur peut être instancié à cet état.

* calculer le nouvel état suite à l'application d'un opérateur instancié à un état.

Remarquons que cette méthode est celle de l'analyse moyen-fin.

[8] Cette suite d'opérations est souvent appelée l'expansion d'un état.

Afin de définir plus précisément l'interface, nous pouvons maintenant déduire des manipulations précédentes, les procédures nécessaires à l'implémentation de cette interface :

* trois procédures sont associées aux états. La première implémente le but pour l'espace de problème. La seconde détecte quand deux états donnés sont identiques. La dernière produit une description d'un état lisible pour l'homme.

* seulement deux procédures sont nécessaires pour les opérateurs. La première fournit une liste des opérateurs de l'espace de problème et la seconde donne, étant donné un état et un opérateur, toutes les instanciations complètes de cet opérateur sur cet état et les états obtenus de l'application de cet opérateur sur l'état.

- **Le mécanisme de recherche**

Très classiquement, ce mécanisme est une liste qui, initialement, contient uniquement l'état initial. Le processus de recherche se déroule selon les étapes suivantes :

* enlever un état de la liste

* si l'état satisfait le but, on arrête et on signale la réussite en retournant le chemin trouvé

* sinon, calculer les opérateurs qui peuvent être appliqués à l'état courant et aux états résultants de chacun d'eux. Mettre à jour la liste et recommencer

* si la liste est vide alors on arrête et on signale l'échec de la recherche.

Il reste alors au concepteur à choisir la manière dont la liste sera organisée et mise à jour, ce qui déterminera la stratégie de recherche utilisée. Par

exemple, une liste non triée FIFO (First In First Out) correspond à une recherche en largeur d'abord.

e) Génération de programmes autonomes

Afin de pouvoir réaliser la génération de programmes autonomes pour un logiciel de troisième génération, il est nécessaire de modifier quelque peu l'architecture du système. Cette nouvelle architecture[9], basée sur le système Unix, comprend :

- un langage : Smile-2[10], qui est plus fonctionnel que le précédent. Il est également plus raffiné. Cela signifie d'une part que certaines structures de données ont été rationalisées. En d'autres termes, elles ont été simplifiées; d'autre part quelques opérations ont été changées. Ce langage a aussi été doté de liens plus étroits avec le *Domain Models*. Cela signifie deux choses : tout d'abord, il y a des références et des références croisées vers le *Domain Models*; ensuite, il y a un index pour faciliter la recherche et le retrait d'éléments.

- Domain Models : on crée un *framework* de structure de classe pour unifier, simplifier l'interface utilisateur. Les spécifications formelles sont basées sur VDM (avec VDM, on peut créer sa logique de spécification formelle pour son logiciel). On ajoute également un lexique de mots et une modélisation d'état/événements (ceux-ci permettent de déclencher une procédure).

- Object Database Server : il s'agit essentiellement ici d'avoir une interface utilisateur orientée objet pour pouvoir faire le lien avec la base de données SQL.

- Reasoning/Processing : ce module comprend :

[9] Cette architecture est en cours d'élaboration à l'heure où nous écrivons ces lignes et se terminera dans les mois prochains.
[10] Ce langage est utilisé pour modéliser les structures des méta-informations.

* Case Library (comme précédemment)

* Plans Library : celle-ci se compose de *strategies* (c'est-à-dire pour résoudre les problèmes) et de *narratives* (c'est-à-dire une description de haut niveau de la manière dont il faut résoudre le problème)

* Problem solving langage généré à partir du Domain Models

* Program specification/generation

* Problem elicitation/solution

Les particularités de cette nouvelle architecture sont les suivantes :

* l'architecture est totalement client/serveur

* des facilités d'entrées/sorties pour les documents, la base de données SQL et les messages EDI

* le système de cas est écrit en C++ ainsi qu'en Lisp sous X-Windows

* il y a un modèle de dissémination

* des liens sont possibles avec d'autres logiciels statistiques, par exemple SAS.

Chapitre 6 : Exemple d'application du générateur

6.1. Introduction

Dans ce chapitre, nous nous proposons de tester le mécanisme de génération.

Avant de tester le mécanisme de génération, une phase préliminaire est cependant nécessaire afin de créer le scénario sur lequel le test sera réalisé.

Nous présenterons donc dans un premier temps l'enquête sur la qualité du logement en Wallonie qui nous a servi de base pour la spécification du scénario. Nous pourrons dès lors, à partir des concepts et métadonnées identifiés dans cette enquête, créer le scénario proprement dit. Enfin, nous présenterons les résultats du test.

6.2. Enquête sur la qualité du logement en Wallonie[1]

6.2.1. Contexte et objectifs de l'enquête INL

L'enquête sur la qualité du logement en Wallonie a été réalisée en 1981-1982 à l'Institut National du Logement (INL), qui a depuis été supprimé.

L'objectif de l'enquête était double :

1. Rendre compte des informations fournies par l'enquête sur l'état du parc immobilier en Wallonie et sur les besoins en rénovations;

2. Analyser l'outil de collecte des données en vue de compléter la réflexion sur la méthode d'évaluation de la qualité du logement en Wallonie.

La politique envisagée visait à rencontrer plusieurs préoccupations :

- Le plein emploi

- L'équité sociale

- L'expression des progrès techniques

Dans ce but, trois cibles notamment étaient retenues :

a) l'abaissement des prix de revient des constructions,

b) l'équipement des logements,

c) la coordination de l'activité en matière de logement à assurer, entre autres par la révision de la législation.

[1] Le lecteur intéressé pourra trouver une présentation plus détaillée de cette enquête en annexe B.

Il s'agit donc d'une enquête par sondage réalisée pour la première fois en 1961-1962 et renouvelée en 1971-1972.

Il existait déjà des enquêtes réalisées à l'échelon des communes, des villes ou des provinces. Mais ces démarches étaient partielles et s'appuyaient sur des méthodes et des critères disparates.

Par ailleurs, les recensements généraux fournissaient également une série d'informations statistiques sur le parc logements et sur ses occupants. Ils s'appuyaient sur une définition du logement qui coïncide avec le ménage occupant. Mais les promoteurs de l'enquête INL ont toujours considéré que ces statistiques ne permettaient pas une approche qualitative du logement et de sa salubrité.

La spécificité et l'originalité, mais aussi la difficulté de l'enquête INL par rapport au recensement et aux enquêtes locales, résidaient dans la volonté d'évaluer l'état des logements en termes de salubrité et d'habitabilité de manière unifiée sur tout le territoire. Concrètement, l'un des buts de l'enquête INL était de dénombrer les logements insalubres à démolir et à reconstruire. Cette volonté impliquait pour les promoteurs une définition du logement qui soit indépendante du ménage qui l'occupe.

6.2.2. La méthode d'évaluation

Pour mener à bien sa mission, l'INL s'est défini une méthode d'évaluation basée sur le choix de caractéristiques du bâtiment et du logement comme indicateur de qualité sur lequel les enquêteurs porteront un jugement de valeur.

L'évaluation de la qualité est basée sur un ensemble "d'appréciations qualitatives" de divers aspects particuliers du bâtiment, du logement et de l'équipement, et sur une appréciation globale[2] de l'état de salubrité du logement.

6.2.3. Intérêt actuel de l'enquête

Sur le plan analytique, le sondage devrait permettre d'amorcer l'étude des facteurs qui favorisent la dégradation du patrimoine immobilier, par l'analyse des relations entre les variables qu'il est possible de construire à partir du sondage.

Pour maximiser l'exploitation des résultats du sondage, les multiples données collectées ont été classées en 6 catégories ou modules de variables :

1. Les variables d'évaluation

Ce sont les variables qui correspondent à des réponses aux questionnaires basées sur un jugement de valeur des enquêteurs. Parmi elles figurent les évaluations particulières, c'est-à-dire les évaluations des composantes des bâtiments, des caractéristiques des logements et des équipements ainsi que l'indice de la salubrité et de la structure du logement.

2. Les variables d'identification

Sur base des données disponibles dans le questionnaire, 6 variables d'identification ont été retenues : la date de construction, le type de bâtiment, l'usage du bâtiment, l'implantation provinciale, l'implantation selon le degré d'urbanité, le type de propriétaire.

[2] L'indice synthétique vise à rendre compte de la qualité des logements et des bâtiments. Il ne résulte pas d'une addition d'appréciations particulières mais d'un nouveau jugement des enquêteurs qui attribuent à chaque logement un code qui correspond aux catégories définies dans les recommandations de l'INL et qui d'après ces recommandations s'appuie sur "l'analyse des aspects physiques et structurels du logement".

3. Les variables sociales

Ce sont les principales caractéristiques des occupants collectées par le questionnaire : âge du chef de ménage, nationalité, activité socioprofessionnelle, état civil, taille du ménage, nombre d'enfants.

4. Les caractéristiques des bâtiments

C'est-à-dire les données objectives concernant les caractéristiques physiques des bâtiments : dimensions, matériaux des toitures et des murs...

5. Les caractéristiques des logements

Il s'agit essentiellement des données collectées par l'enquête au sujet du logement concernant les dimensions, la densité d'occupation, le nombre de pièces...

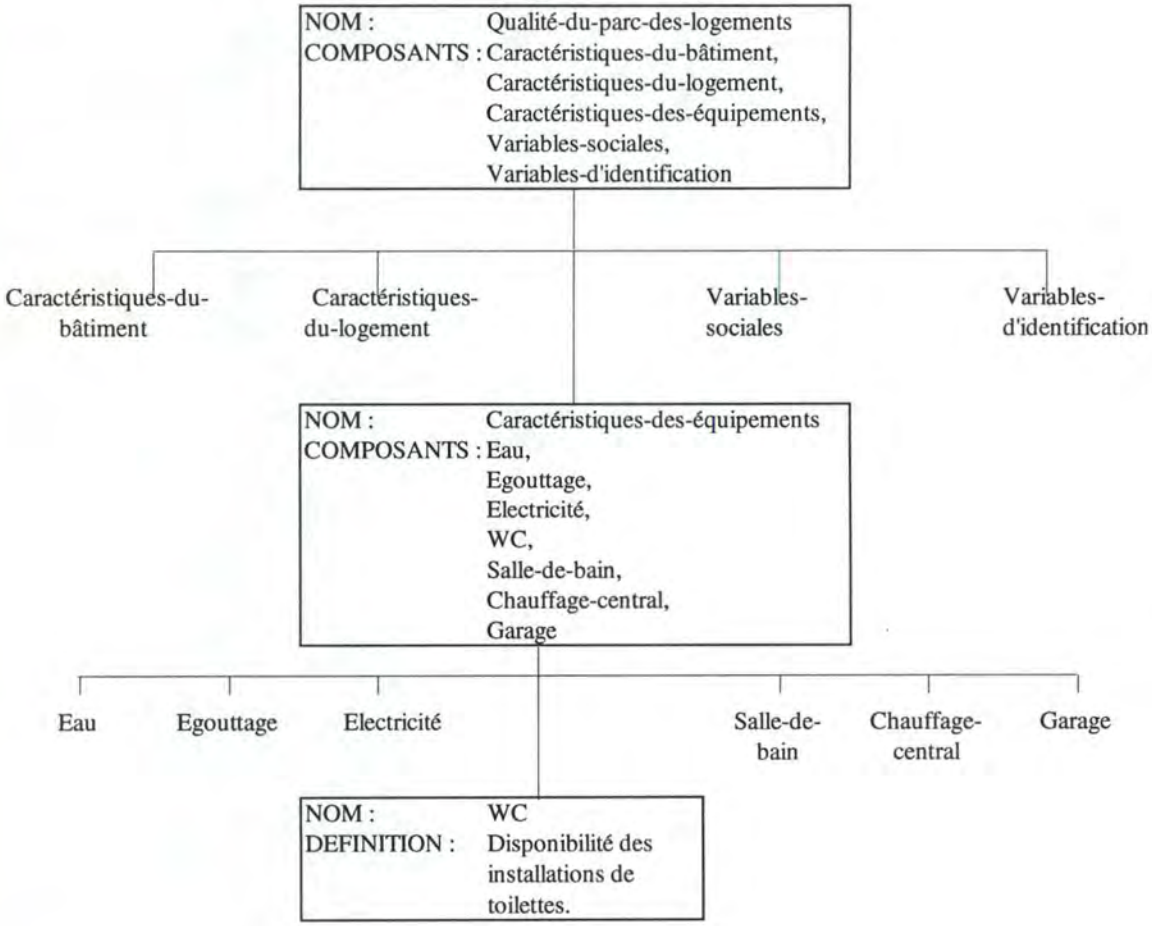
6. Les caractéristiques des équipements

C'est-à-dire l'absence ou la présence des principaux équipements primaires (eau, égouttage, électricité, wc), équipements secondaires (salle de bain, chauffage, garage,...)

6.3. Identification des concepts et des métadonnées

6.3.1. Concepts

Un ensemble de définitions de concepts organisés selon une hiérarchie faible a été identifié dans l'enquête sur la qualité du logement en Wallonie, et est représenté ci-dessous :



6.3.2. Métadonnées

Cinq classes d'information ont été identifiées dans l'enquête sur la qualité du logement en Wallonie. Les instances de ces classes sont présentées ci-dessous :

* Enquête

NOM	: **EQLW**
TITRE	: "Enquête sur la qualité du logement en Wallonie"
REGION	: Wallonie
POPULATION	: Registre de l'Administration Wallonne
UNITE STATISTIQUE	: Logement
DATE	: 1981
PERIODICITE	: Une fois
CONCEPTS	: Qualité-du-parc-des-logements, Caractéristiques-du-logement, Caractéristiques-du-bâtiment

* Organisation

NOM	: **INL**
TITRE	: "Institut National du Logement"
REGION	: Belgique
FONCTION	: *Monitoring* du parc des logements en Wallonie
COMMENTAIRES	: "Supprimé en 1983" "Aucun successeur"

* Unité statistique

NOM	: **Logement**
TITRE	: "Un logement occupé par un ou plusieurs ménages"
SOURCE	: Méthodologie EQLW

* Mesure

NOM	: **IDS**
TITRE	: "Index de salubrité"
DEFINITION	: "Salubrité du parc des logements"
ENQUETE	: EQLW
TYPE	: Qualitatif
CONCEPT	: Qualité-du-parc-des-logements
VALEURS AUTORISEES	: Salubre - Insalubre améliorable - Insalubre non améliorable

* Variables statistiques

NOM	: **VDI**
TITRE	: "Variables d'Identification"
ENQUETE	: EQLW
REGION	: Wallonie
CATEGORIES	: Date-de-construction, Type-de-bâtiment, Usage-du-bâtiment, Taille, Type-de-propriétaire, Localisation-géographique
SERIES TEMPORELLES	: Non
PERIODICITE	: Une fois

DATE : 1981

DEFINITION : "Enumération classée des caractéristiques générales"

--

NOM : **SEC**

TITRE : "Variables de Classification Socio-Economiques"

ENQUETE : EQLW

REGION : Wallonie

CATEGORIES : Age-du-chef-de-ménage, Nationalité, Activité-socio-professionnelle, Etat-civil, Taille-du-ménage, Nombre-d'enfants, Régime foncier

SERIES TEMPORELLES : Non

PERIODICITE : Une fois

DATE : 1981

DEFINITION : "Enumération classée des caractéristiques sociales"

--

NOM : **VDE**

TITRE : "Variables d'Evaluation"

ENQUETE : EQLW

REGION : Wallonie

CATEGORIES : Caractéristiques-du-bâtiment, Caractéristiques-du-logement

SERIES TEMPORELLES : Non

PERIODICITE : Une fois

DATE	: 1981
DEFINITION	: "Enumération classée des caractéristiques générales"

--

NOM	: **SDL**
TITRE	: "Salubrité du logement"
ENQUETE	: EQLW
REGION	: Wallonie
MESURE	: IDS
COMPOSANTS	: VDI, SEC, VDE
SERIES TEMPORELLES	: Non
PERIODICITE	: Une fois
DATE	: 1981
DEFINITION	: "Evaluation générale de la qualité du parc des logements"

6.4. Scénario

Le scénario sur lequel le test du mécanisme de génération sera réalisé est présenté ci-dessous :

(:**name** QualitéParcLogements.Décrire.Wallonie

:**problem-features** ((object describe)
(concept Qualité-du-parc-des-logements)
(area (Belgique Wallonie)))

:**surface** ("Décrire les concepts et les données disponibles utilisés dans l'évaluation de la qualité du parc des logements en Wallonie")

:**concept-refs** (($concept (fetch.concept[3] Qualité-du-parc-des-logements))

:**metadata-refs** (($enquête (fetch.survey[3] EQLW))
($organisation (fetch.organisation INL))
($unité-statistique (fetch.statistical-unit logement))
($mesure (fetch.measure IDS))
($variable-1 (fetch.indicator VDI))
($variable-2 (fetch.indicator SEC))
($variable-3 (fetch.indicator VDE))
($variable-4 (fetch.indicator SDL)))

:**schema** ((describe[4] $concept)
(describe $variable-4)
(describe $variable-1)
(describe $variable-2)
(describe $variable-3))))

6.5. Résultats de l'interprétation, de l'exécution du programme (script) généré, de l'exécution du programme autonome (analyse) généré

Description :

Form: <HOUSING-STOCK-QUALITY>
Class: <CONCEPT>
With Features:
DEFINITION = Quality of Housing Stock
COMPONENTS :
BUILDING-CHARACTERISTICS
DWELLING-CHARACTERISTICS

[3]Les fonctions *fetch.concept* et *fetch.[metadata_name]* permettent l'accès à un concept/une métadonnée.
[4]L'opérateur *describe* permet la description d'un concept/une donnée dans un domaine.

AMENITIES-AVAILABLE
SOCIO-ECONOMIC-FACTORS
IDENTIFICATION-FACTORS

Description :

Form: <SDL>
Class: <INDICATOR>
With Features:
TITLE = Salubrité du Logement
SURVEY = EQLW
AREA = WALLONIE
MEASURE = IDS
COMPONENTS :
VDI
SEC
VDE
TIME-SERIES = NO
PERIODICITY = ONCE
DATA-SINCE = 1981
DEFINITION = General evaluation of the Housing Stock quality

Description :

Form: <VDI>
Class: <INDICATOR>
With Features:
TITLE = Variables d'Identification
SURVEY = EQLW
AREA = WALLONIE
CATEGORIES :
PERIOD-OF-CONSTRUCTION
TYPE-OF-BUILDING
USE-OF-BUILDING
SETTLEMENT-SIZE
TYPE-OF-OWNER
GEOGRAPHIC-LOCATION
TIME-SERIES = NO

PERIODICITY = ONCE
DATA-SINCE = 1981
DEFINITION = Classified enumeration of general characteristics

Description :

Form: <SEC>
Class: <INDICATOR>
With Features:
TITLE = Variables de Classification Socio-Economique
SURVEY = EQLW
AREA = WALLONIE
CATEGORIES :
AGE-OF-HEAD-OF-HOUSEHOLD
NATIONALITY
PROFESSIONAL-STATUS-OF-HEAD-OF-HOUSEHOLD
CIVIL-STATUS
SIZE-OF-HOUSEHOLD
NUMBER-OF-CHILDREN
TENURE
TIME-SERIES = NO
PERIODICITY = ONCE
DATA-SINCE = 1981
DEFINITION = Classified enumeration of Social characteristics

Description :

Form: <VDE>
Class: <INDICATOR>
With Features:
TITLE = Variables d'Evaluation
SURVEY = EQLW
AREA = WALLONIE
CATEGORIES :
BUILDING-CHARACTERISTICS
DWELLING-CHARACTERISTICS
TIME-SERIES = NO
PERIODICITY = ONCE

DATA-SINCE = 1981
DEFINITION = Classified enumeration of general characteristics

Conclusion

Ce mémoire est consacré à l'étude de la faisabilité d'un générateur de programmes statistiques et à la conception de celui-ci. La réalisation de ce générateur avait pour objectif d'étendre les fonctionalités du système EISI, système développé pour permettre aux statisticiens, d'une part de gérer la méta-information statistique dans un domaine particulier et d'autre part, de résoudre des problèmes statistiques (ce système rencontre, comme nous l'avons vu, des objectifs du *Knowledge-based Software Engineering*, notamment la réutilisation, l'aide à l'utilisateur, etc.).

Afin de pouvoir développer le prototype, nous avons examiné et étendu les diverses structures existantes dans l'environnement de travail. Pour construire ce prototype, nous avons d'abord travaillé de manière restrictive (travail sur les métadonnées uniquement et génération de programmes Lisp) puis, nous sommes passés à une vision des choses plus générale (travail sur des données concrètes et génération de programmes autonomes pour des logiciels de troisième génération). Ce dernier point est en cours de d'élaboration.

L'objectif futur de l'équipe de développement du système EISI, est d'en faire un système commercial, implémenté sous Unix et PC. Lors d'une réunion prochaine à Genève, les résultats seront présentés et le système pourrait alors être adopté par des pays de l'Est (Slovaquie, Pologne, etc.) vivement intéressés notamment par la génération de programmes directs.

Bibliographie

[Bahrami] A. Bahrami, Designing Artificial Intelligence Based Software, Sigma Press-Wilmslow (Hasted Press), 1988.

[Barr,Cohen&Feigenbaum] A. Barr, P.R. Cohen & E.A. Feigenbaum, The handbook of Artificial Intelligence, Volume IV, chap.XX, Addison-Wesley Publishing Company, Inc., 1989.

[Charniak&Dermott] E. Charniak & D.Mc Dermott, Introduction to Artificial Intelligence, Addison-Wesley Publishing Company, USA, 1986.

[de Vaney] Christopher J.de Vaney, Integrating Statistical Meta-Information and Methodologies, 1993.

[de Vaney93] Christopher, J.de Vaney, DOSES, Expert interface to statistical information (EISI), final report, 1993.

[de Vaney92] Christopher, J.de Vaney, The expert Interface to Statistical Information, Rationale, Technique and Experiences, 1992.

[Epprecht] Eugenio K. Epprecht, An approach for Knowledge Representation and Reasoning with contextual Strategies in Knowledge-based Statistical consultancy Systems, 1992.

[Feather] M.S. Feather, Reuse in the context of a transformation-based methodology, in Software reusability, concepts and models, Volume I, ACM Press, 1989.

[Forbus&De Kleer] Kenneth D. Forbus & Johan De Kleer, Building Problem Solvers, MIT Press, 1993.

[Forte&Norman] G. Forte & R.J. Norman, A Self-Assessment by the Software Engineering Community, in Communications of the ACM, April 1992, Volume 35, Number 4.

[Harandi&Lubars86] Mehdi T. Harandi & Mitchell D. Lubars, Knowledge-based software development : a paradigm and a tool, in AFIPS Conference Proceedings, Volume 55, 1986.

[Harandi&Lubars87] Mehdi T. Harandi & Mitchell D. Lubars, Knowledge-based software design using design schemas, in Proceedings 9th international conference on Software Engineering, March 30 - April 2, 1987, Monterey, California, USA.

[Harandi&Lubars89] Mehdi T. Harandi & Mitchell D. Lubars, Addressing software reuse through knowledge-based design, in Software Reusability, Vol. II, Applications and Experiences, ACM Press, 1989.

[Hasemer&Domingue] T. Hasemer & J. Domingue, Common Lisp programming for Artificial Intelligence, Addison-Wesley Publishing Company, Great-Britain, 1989.

[Ince] D. Ince, Object-oriented software engineering with C++, Mc Graw-Hill book company, UK, 1991.

[Kvanli] A.H. Kvanli, Statistics, a computer integreted approach, West Publishing Company, USA, 1988.

[London&Feather] P.E. London & M.S. Feather, Implementing specification freedoms, in Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann Publishers, Inc., California, 1986.

[Luger&Stubblefield] G.F. Luger, W.A. Stubblefield, Artificial Intelligence and the Design of Expert Systems, The Benjamin/Cummings Publishing Company, California 1989.

[Lust&Wey] Frédéric Lust & Alain Wey, Elaboration d'une architecture générale pour un système de résolution de problèmes, application au domaine des statistiques officielles, 1993.

[Moser&Kalton] C.A. Moser & G. Kalton, Survey Methods in Social Investigation (2nd edition), ed. Gower, England, 1989.

[Norman&Chen] Ronald J. Norman & Gene Chen, Working together to integrate CASE, in IEEE Software, March 1992.

[Pyster] A.B. Pyster, Compiler design and construction tools and techniques with C and Pascal, 2nd edition, Van Nostrand Reinhold Company, New York, 1988.

[Sato] H. Sato, A Data Model, Knowledge Base and Natural Language, Processing for sharing a large statistical database, in Statistical and Scientific Data Base Management, Rafanelli & al., Springen Berlin, 1989.

[Scherlis] W.L. Scherlis, Abstract data types, Specialization and program reuse, in Lecture Notes in computer science, Advanced programming environments, Proceedings of an International Workshop, Trondheim, Norway, June 1986, Springer-Verlag.

[Tierney] L. Tierney, Lisp-Stat, An object-oriented Environment for Statistical computing and Dynamic Graphics, a Wiley-Interscience Publication, John Wiley & Sons, 1990.

[Van Vliet] H. Van Vliet, Software engineering : principles and practice, Chichester, Wiley, 1993.

[Williams] Williams, Software reuse (abstract) in Proceedings 10th international conference on software engineering, April 11-15 Singapore 1988.

[Winston&Horn] P.H. Winston & B.K.P. Horn, Lisp, second edition, Addison-Wesley Publishing Company, USA, 1984.

[Winston] P.H. Winston, Artificial Intelligence, second edition, Addison-Wesley Publishing Company, USA, 1984.

[Yuasa&Hagiya] T. Yuasa & M. Hagiya, Introduction to Common Lisp, Academic Press, inc., USA,1987.

[Zeroual&Robillard] Kacem Zeroual & Pierre-N. Robillard, KBMS : A Knowledge-Based System for Modeling Software System Specifications, in IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 3, June 1992.

ANNEXES

ANNEXE A : Listing des fonctions implémentées

Cette annexe présente les listings des fonctions présentées au cours de ce mémoire. Celles-ci sont groupées selon la structure à laquelle elles sont destinées. Cette annexe comprend :

```
#|

Scenario.lsp

Domain Assistant Usage Scenario object and methods

|#

;; 1. The structure definition

( defstruct scenario
  ( name nil )
  ( problem-features nil )
  ( surface nil )
  ( concept-refs nil )
  ( metadata-refs nil )
  ( global-constraints nil )
  ( schema nil ))

;; 2. The accessor/setter methods

( predicate is-scenario? ( aThing )
  "generic predicate for scenarios"
  ( scenario-p aThing ))

( to scen/get-name ( aScenario )
  "Get the name of a scenario"
  ( if-true ( is-scenario? aScenario )
    ( scenario-name aScenario )))

( to scen/get-problem-features ( aScenario )
  "Get the problem features of a scenario"
   ( if-true ( is-scenario? aScenario )
     ( scenario-problem-features aScenario )))

( to scen/get-surface ( aScenario )
  "Get the surface form explanation for a scenario"
  ( if-true ( is-scenario? aScenario )
    ( scenario-surface aScenario )))

( to scen/get-concept-refs ( aScenario )
  "Get the references to concepts required by the scenario"
   ( if-true ( is-scenario? aScenario )
     ( scenario-concept-refs aScenario )))
```

```
( to scen/get-metadata-refs ( aScenario )
  "Get the references to metadata required by the scenario"
  ( if-true ( is-scenario? aScenario )
    ( scenario-metadata-refs aScenario )))

( to scen/get-global-constraints ( aScenario )
  "Get the lambda form of the global constraints applicable against a
  scenario"
  ( if-true ( is-scenario? aScenario )
    ( scenario-global-constraints aScenario )))

( to scen/get-schema ( aScenario )
  "Get the list of operator references (=Schema) from the scenario"
  ( if-true ( is-scenario? aScenario )
    ( scenario-schema aScenario )))
```

```
#|

Scenlib.lsp

Library Management for Usage Scenarios in the Domain Assistant

|#

;; Global operators management functions

( defvar *scenarios* ( make-hash-table :size 1011 ))

( defun store-scenario ( aScenario )
 ( if-true  ( and ( is-scenario? aScenario )
                 ( scen/get-name aScenario ))
      ( setf ( gethash (scen/get-name aScenario) *scenarios* )
       aScenario )))

( defun fetch-scenario ( aName )
 ( if-true aName
  ( gethash aName *scenarios* )))

( defun index-on-scenario-name ()
 "Return a list of all defined scenarios"
 ( let (( result nil ))
  ( maphash #'( lambda ( aVar aValue ) ( push aVar result ))
          *scenarios* )
  result ))

( defun generate-scenario-index-entry ( aScenario )
 "Create an index entry for a case object"
 ( if-true ( is-scenario aScenario )
  ( list ( scen/get-name aScenario )
       ( scen/get-problem-features aScenario ))))

( defvar *scenario-index* nil )

( defun generate-scenario-index ()
 "Generate an index of all scenarios in the library, of the form
  (Scenario-name problem-features )"
 ( let ((the-result nil))
  ( maphash #'( lambda ( aVar aValue )
              ( push ( generate-scenario-index-entry aValue) the-result))
    *scenarios* )
  ( setf *scenario-index* the-result )))
```

```
( defun clear-scenario-index ()
  "Clear the scenario index"
  ( setf *scenario-index* nil ))

;; read in the scenarios from an import file, and store them in the library

( defun scenario/stream-in ( aFileName )
  "Read in a set of scenarios as listed keyword specs, and apply the
   make-scenario function to them, storing them using the name as the
   key"
  ( with-open-file ( infile aFileName
                     :direction :input
                     :if-does-not-exist :error )
    ( let (( all-specs (read infile )))
      ( foreach x in all-specs
        ( store-scenario ( apply #'make-scenario x ))))))
```

```
#|

Operator.lsp

The operator object and methods for use in the planning system

|#

( defstruct operator
  ( name nil )
  ( parameters nil )
  ( constraints nil )
  ( body nil )
  ( interpretations nil )
  ( synopsis nil )
  ( explanation nil ))

;; Accessors

( predicate is-op? ( anOp )
  "Test to see if the thing is an operator"
  ( operator-p anOp ))

( to op/name (anOp)
  "Get the name of an operator"
  ( if-true (is-op? anOp)
    ( operator-name anOp )))

( to op/set-name (anOp aName)
  "Set the name of an operator from a supplied name"
  ( if-true ( and ( is-op? anOp )
                  ( not-null aName )
                  ( symbolp aName ))
    ( setf ( operator-name anOp ) aName )))

( to op/parameters (anOp)
  "Get the list of parameters for an operator"
  ( if-true ( is-op? anOp )
    ( operator-parameters anOp )))

( to op/set-parameters (anOp paramList)
  "Set the parameter markings for an operator from a supplied parameter
   list.  Each element of the list MUST be a symbol"
  ( if-true ( and ( is-op? anOp )
                  ( not-null paramList )
```

```
            ( forall x in paramList ( symbolp x )))
    ( setf ( operator-parameters anOp) paramList )))

( to op/constraints ( anOp )
  "Get the constraints block for an operator"
  ( if-true ( is-op? anOp )
    ( operator-constraints anOp )))

( to op/set-constraints ( anOp aConstraintsList )
  "Set the constraints of an operator.  The constraints list is only
  verified to ensure that it's not null, and that it is a list"
  ( if-true ( and ( is-op? anOp )
            ( not-null aConstraintsList )
            ( listp aConstraintsList ))
    ( setf ( operator-constraints anOp ) aConstraintsList )))

( to op/body ( anOp )
  "Get the function body block for an operator"
  ( if-true ( is-op? anOp )
    ( operator-body anOp )))

( to op/set-body ( anOp aFunctionList )
  "Set the body of an operator.  The function list is only
  verified to ensure that it's not null, and that it is a list"
  ( if-true ( and ( is-op? anOp )
            ( not-null aFunctionList )
            ( listp aFunctionList ))
    ( setf ( operator-body anOp ) aFunctionList )))

( to op/interpretations ( anOp )
  "Get the association list of interpretations permissible for the
  results of applying the operator"
  ( if-true ( is-op? anOp )
    ( operator-interpretations anOp )))

( to op/set-interpretations ( anOp anInterpretationsBlock )
  "Set the interpretations form of an operator. The interpretations form
  is as a list of (key* interpretation) forms, where the first element
  of each form is a list of one or more keys, and the second element is
  a symbolic or textual tag for the result"
  ( if-true ( and ( is-op? anOp )
            ( listp anInterpretationsBlock )
            ( not-null anInterpretationsBlock )
            ( forall x in anInterpretationsBlock
              ( and ( listp ( car x ))
```

```
                    ( not-null ( car x ))
                    ( or ( symbolp (rest x ))
                        ( stringp (rest x ))))))
    ( setf (operator-interpretations anOp) anInterpretationsBlock )))

( to op/synopsis ( anOp )
  "Get the synopsis text for the operator"
  ( if-true ( is-op? anOp )
    ( operator-synopsis anOp )))

( to op/set-synopsis ( anOp aText )
  "Set the synopsis of the operator from the supplied text"
  ( if-true ( and ( is-op? anOp )
                  ( not-null aText ))
    ( setf ( operator-synopsis anOp ) aText )))

( to op/explanation ( anOp )
  "Get the explanation form of an operator"
  ( if-true ( is-op? anOp )
    ( operator-explanation anOp )))

( to op/set-explanation ( anOp anExplanationForm )
  "Set the explanation form of the operator to the supplied form.
   This is a list of the texts of the explanation."
  ( if-true ( and ( is-op? anOp )
                  ( not-null anExplanationForm ))
    ( setf ( operator-explanation  anOp ) anExplanationForm )))

;; creator function for operator

( to op/new ( &rest args )
  "Create a new operator structure with the supplied arguments"
  ( apply #'make-operator args ))

( to op/create-from-list ( anArgList )
  "Create an operator from a supplied list of arguments"
  ( apply #'make-operator anArgList ))

;; Create a loadable "dump" list for the operator

( to op/dump-load-form ( anOp )
  "Dump the fields of an operator into a form from which the operator
   can be re-generated by applying op/create-from-list"
 ( if-true ( is-op? anOp )
   ( list
```

```
    ':name (op/name anOp)
    ':parameters (op/parameters anOp)
    ':constraints (op/constraints anOp)
    ':body (op/body anOp )
    ':interpretations (op/interpretations anOp)
    ':synopsis (op/synopsis anOp)
    ':explanation (op/explanation anOp))))

;; create a script function form for the operator

( to operator->script ( anOp )
  "Create a script for the functional part of the operator"
  ( if-true ( is-op? anOp )
    ( store-script ( op/name anOp )
      `( lambda ,(op/parameters anOp)
         ( if ( and ,@(op/constraints anOp) )
            ( begin ,@(op/body anOp ))
            'fail )))))

;; Print an operator structure to a stream ( generic for both Operator
;; and Opres structures

( to print-operator ( anOp &optional ( aStream t ))
  "Print an operator to the supplied stream, in a pretty form"
  ( if-true ( or ( is-op? anOp )
                 ( is-opres? anOp ))
    ( if ( is-op? anOp )
      ( begin
       ( format aStream "~%Operator : ~a" ( operator-name anOp ))
       ( format aStream "~%  Parameters : ~a" ( operator-parameters anOp ))
       ( format aStream "~%  Synopsis : ~a " (operator-synopsis anOp ))
       ( format aStream "~%  Constraints : ~%")
        ( pprint ( operator-constraints anOp ) aStream )
       ( format aStream "~%  Operation : ~%" )
        ( pprint ( operator-body anOp ) aStream )
       ( format aStream "~%  Interpretations ~%" )
        ( pprint ( operator-interpretations anOp ) aStream ))
      ( begin
       ( format aStream "~%Op-result : ~a" ( opres-name anOp ))
       ( format aStream "~%  Parameters : ~a" ( opres-parameters anOp ))
       ( format aStream "~%  Synopsis : ~a " (opres-synopsis anOp ))
       ( format aStream "~%  Constraints : ~a" ( opres-constraints anOp ))
       ( format aStream "~%  Operation : ~a"( opres-body anOp ) )
       ( format aStream "~%  Interpretations ~a"
          ( opres-interpretations anOp ))))))
```

```
#|

Opslib.lsp

Library Management for operators in the Domain Assistant

|#

;; Global operators management functions

( defvar *operators* ( make-hash-table :size 1011 ))

( defun store-operator ( anOpForm )
  ( if-true  ( and ( is-op? anOpForm )
                   ( op/name anOpForm ))
      ( setf ( gethash (op/name anOpForm) *operators* ) anOpForm )))

( defun fetch-operator ( aName )
  ( if-true aName
    ( gethash aName *operators* )))

( defun index-of-operators ()
  "Return a list of all defined operators"
   ( let (( result nil ))
     ( maphash #'( lambda ( aVar aValue ) ( push aVar result ))
              *operators* )
     result ))

;; create scripts for each operator

( defun generate-scripts-for-operators ()
  ( maphash #'( lambda ( aVar aValue )
               ( operator->script aValue )) *operators* ))

;; stream in a set of operators defined in list form, write them
;; into the Operators library and generate scripts for them.

( defun stream-in-operators ( aFileName )
  "Open a file, and stream in the operator specifications. For each
   specification, convert it into an operator object, generate a script
   for it and store it in the operations library"
  ( let (( the-specs nil )
         ( the-operator nil ))
    ( with-open-file ( import aFileName
                      :direction :input
```

```
                    :if-does-not-exist :error )
    ( setf the-specs ( read import )))
    ( if-true the-specs
     ( foreach x in the-specs
      ( begin
       ( setf the-operator ( op/create-from-list x ))
       ( if-true ( is-op? the-operator )
        ( operator->script the-operator )
        ( store-operator the-operator )))))))

;; The loader function

( stream-in-operators "h:\\xlisp\\lsp\\operator.imp" )

;; print all the operators to an output file (pretty-print)

( to print-all-operators ( &optional ( aFileName "Print.out" ))
  "Pretty-print all the operators to an output file"
  ( with-open-file ( prnout aFileName
                    :direction :output
                    :if-does-not-exist :create
                    :if-exists :overwrite )
   ( maphash
    #'( lambda ( aVar aValue )
       ( print-operator aValue prnout ))
    *operators* )))
```

```
#|

Scripts.lsp

Application scripts manager

Requires: krlmacs.lsp

|#

( defvar *scripts* ( make-hash-table :size 2011 ))

( defun store-script ( aName aLambdaForm )
  ( if-True ( and aName aLambdaForm )
        ( setf ( gethash aName *scripts* ) aLambdaForm )))


( defun get-script ( aName )
  ( if-True aName
    ( gethash aName *scripts* )))

( defun perform ( aName aVarList )
  ( let (( script ( get-script aName )))
    ( if script
      ( apply script aVarList )
      '*No-Script-Found* )))

( defun run-script ( aName &rest vars )
  ( let ((result ( perform aName vars )))
   ( if (history-running?)
     ( log-script-execution aName result ))
   result ))

( defun run-script-with-varlist ( aName Vars )
  ( let (( result ( perform aName Vars )))
    ( if (history-running?)
      ( log-script-execution aName result ))
    result ))

( defun index-of-scripts ()
 "Map over the scripts table, returning a list of all script names"
 ( let (( result nil ))
 ( maphash
   #'( lambda (aScript aForm) (push aScript result )) *scripts*)
  result ))
```

```
;; History support for script execution -- saves the name of the script and
;; the result

;; The ALIST of (script-name results)

( defvar *execution-history* $false )

;; Boolean for the on/off mode for script history execution

( defvar *script-history-mode* $false )

( defun reset-execution-history ()
  ( setf *execution-history* $false ))

( defun script-history ()
  ( reverse *execution-history* ))

( defun script-history-on ( &optional ( clear $false ))
  ( if clear
    ( reset-execution-history ))
  ( setf *script-history-mode* $true ))

( defun script-history-off ()
  ( if-true *script-history-mode*
    ( setf *script-history-mode* $false )
    ( scripts-history )))

( defun history-running? ()
   *script-history-mode* )

( defun log-script-execution ( aName aResult )
  ( push ( list aName aResult ) *execution-history* ))

;; macro for defining scripts

( defmacro script ( name varforms &rest body )
  `( store-script ',name '( lambda ,varforms ,@body )))

;; macro for executing scripts -- be careful!

( defmacro -> ( aName &rest Parameters )
  `( run-script ',aName ,@Parameters ))
```

```
;; Script history utility functions

;; display-script-history

( defun display-script-history ()
  ( if-true (script-history)
    ( foreach x in (script-history)
      ( format t "~%Script: ~a    Result: ~a" (first x) (second x)))))

( defun all-results-for-script ( aName )
  "Return a list of all results from the execution of a script in
   the current history"
 ( let ((result nil))
  ( if-true (script-history)
    ( foreach x in (script-history)
      ( if ( eql (first x) aName )
           ( push x result ) ))
   result )))
```

```
#|

Opres.lsp

The opres result object and methods for use in the planning system

|#

( defstruct ( opres ( :include operator )))

;; Accessors

( predicate is-opres? ( anOp )
  "Test to see if the thing is an opres"
  ( opres-p anOp ))

( to res/name (anOp)
  "Get the name of an opres"
  ( if-true (is-opres? anOp)
    ( opres-name anOp )))

( to res/set-name (anOp aName)
  "Set the name of an opres from a supplied operator"
  ( if-true ( and ( is-opres? anOp )
                  ( not-null aName )
                  ( symbolp aName ))
    ( setf ( opres-name anOp ) aName )))

( to res/parameters (anOp)
  "Get the list of parameters for an opres"
  ( if-true ( is-opres? anOp )
    ( opres-parameters anOp )))

( to res/set-parameters (anOp paramList)
  "Set the parameter markings for an opres from a supplied parameter
   list."
  ( if-true ( and ( is-opres? anOp )
                  ( not-null paramList ))
    ( setf ( opres-parameters anOp) paramList )))

( to res/constraints ( anOp )
  "Get the constraints result for an opres"
  ( if-true ( is-opres? anOp )
    ( opres-constraints anOp )))
```

```
( to res/set-constraints ( anOp aConstraintsResult )
  "Set the constraints result of an opres."
  ( if-true ( is-opres? anOp )
    ( setf ( opres-constraints anOp ) aConstraintsResult )))

( to res/body ( anOp )
  "Get the function result for an opres"
  ( if-true ( is-opres? anOp )
    ( opres-body anOp )))

( to res/set-body ( anOp aFunctionResult )
  "Set the body (=Function) result of an operator"
  ( if-true ( is-opres? anOp )
    ( setf ( opres-body anOp ) aFunctionResult )))

( to res/interpretation ( anOp )
  "Get the interpretation of the operator result"
  ( if-true ( is-opres? anOp )
    ( opres-interpretations anOp )))

( to res/set-interpretation ( anOp anInterpretation )
  "Set the interpretations form of an opres. This is the interpretation
   on the result of an operator (See the operator class)"
   ( if-true ( and ( is-opres? anOp )
                   ( not-null anInterpretation))
     ( setf (opres-interpretations anOp) anInterpretation )))

( to res/synopsis ( anOp )
  "Get the synopsis text for the opres"
   ( if-true ( is-opres? anOp )
     ( opres-synopsis anOp )))

( to res/set-synopsis ( anOp aText )
  "Set the synopsis of the opres from the supplied text"
  ( if-true ( and ( is-opres? anOp )
                  ( not-null aText ))
    ( setf ( opres-synopsis anOp ) aText )))

( to res/explanation ( anOp )
  "Get the explanation form of an opres"
  ( if-true ( is-opres? anOp )
    ( opres-explanation anOp )))

( to res/set-explanation ( anOp anExplanationForm )
  "Set the explanation form of the opres to the supplied form.
```

```
  This is a list of the texts of the explanation."
  ( if-true ( and ( is-opres? anOp )
                  ( not-null anExplanationForm ))
    ( setf ( opres-explanation anOp) anExplanationForm )))

;; creator function for opres

( to res/new ( &rest args )
  "Create a new opres structure with the supplied arguments"
  ( apply #'make-opres args ))

( to res/create-from-list ( anArgList )
  "Create an opres from a supplied list of arguments"
  ( apply #'make-opres anArgList ))

;; Create a loadable "dump" list for the opres

( to res/dump-load-form ( anOp )
  "Dump the fields of an opres into a form from which the opres
   can be re-generated by applying res/create-from-list"
 ( if-true ( is-opres? anOp )
   ( list
     ':name (res/name anOp)
     ':parameters (res/parameters anOp)
     ':constraints (res/constraints anOp)
     ':body (res/body anOp )
     ':interpretations (res/interpretations anOp)
     ':synopsis (res/synopsis anOp)
     ':explanation (res/explanation anOp))))
```

```
#|

Wm.lsp

Working Memory structure and methods

|#

( defstruct wm
  ( vars ( make-hash-table :size 1023 :test #'equal )))

;; wm/new  -- Create a new working memory

( defun wm/new ()
 "Create a new working-memory structure, with no entries"
 ( make-wm ))

;; wm/clear -- Clear method for a Working Memory

( defun wm/clear ( aWM )
 "Clear all references from a Working Memory structure"
 ( if ( wm-p aWM )
    ( clrhash ( wm-vars aWM ))))

;; wm/put -- Storage method for a value in a Working Memory

( defun wm/put ( aWM aKey aValue )
 "Place a value in a Working Memory, indexed under a supplied key"
 ( if ( wm-p aWM )
    ( setf ( gethash aKey ( wm-vars aWM )) aValue )))


;; wm/get -- Retrieval method for a value in a Working Memory

( defun wm/get ( aWM aKey )
 "Get the value associated with a key in a working memory"
 ( if ( wm-p aWM )
    ( gethash aKey ( wm-vars aWM ))))

;; wm/params->bindings -- Method to generate a list of values associated
;; with an parameter list

( defun wm/params->bindings ( aWM aParameterList )
 "Generate a list of the values bound to each label in ParameterList,
  including NIL if there. The values are in the generated list in the
```

```
  order of the parameter list."
 ( if-true ( and ( wm-p aWM )
                 ( listp aParameterList )
                 ( forall x in aParameterList
                   ( and ( symbolp x )
                        ( not-null x ))))
   ( foreach x in aParameterList
     ( wm/get aWM x ))))

;; wm/list-vars-and-vals -- Create a list of all variables and values
;;                          stored in the Working Memory

( defun wm/list-vars-and-vals ( aWM )
  "Generate a list of variable names and values stored in the working
   memory. The result is a list of pairs, where the head of the pair
   is the key in the working memory, and the tail of the pair is the
   value."
 ( if ( wm-p aWM )
   ( let (( results nil ))
     ( maphash
       #'( lambda ( aVar aValue )
            ( push ( list aVar aValue ) results ))
       ( wm-vars aWM ))
     results )))
```

```
#|

OpEval.lsp

Evaluate an Operator in the context of a working memory, returning
an opres instance containing the results.

|#

( defun evaluate-operator ( anOp aParamList aContext )
  "Evaluate an operator structure in the context of a working
  memory, containing all the values to be addressed in the
  course of the evaluation"
  ( if-true ( and
            ( is-op? anOp )
            ( listp aParamList )
            ( not-null aParamList )
            ( wm-p aContext ))
   ;; the local variable setup
   ( let ( ( the-result ( res/new ))
           ( the-dummies ( op/parameters anOp ))
           ( script-result nil )
           ( the-values ( wm/params->bindings aContext aParamList )))

     ;; 1. Assign the name of the result form to that of the operator

     ( res/set-name the-result (op/name anOp ))

     ;; 2. Assign the synopsis of the result form to that of the operator

     ( res/set-synopsis the-result ( op/synopsis anOp ))

     ;; 2.5 Assign the parameters used in the call to the operator result

     ( res/set-parameters the-result aParamList )

     ;; 3. Expand the explanation of the operator in the bindings context

     ( res/set-explanation the-result (op/explanation anOp))

     ;; 4. Call the script that implements the operator and its
     ;; constraints

     ( setf script-result
       ( run-script-with-varlist ( op/name anOp ) the-values ))
```

```
( if ( equal script-result 'fail )
   ( res/set-constraints 'Constraints-failure )
   ( begin
     ( res/set-constraints the-result 'constraints-held )
     ( res/set-body the-result script-result )))

;; 6. If the constraints predicate held, and there are explanations
;;    associated with the operator, attempt to match the result
;;    with the corresponding explanation, otherwise assign the
;;    'no-explanation form

( if ( not ( equal (res/constraints the-result ) 'fail ))
   ( begin
     ( res/set-explanation the-result ( op/explanation anOp ))
     ( res/set-interpretation the-result
       ( or ( if ( op/interpretations anOp )
              ( assoc (res/body the-result)
                ( op/interpretations anOp )))
           'no-specific-interpretation ))))

;; now return the result

the-result )))
```

```
#|

Sceneval.lsp

Evaluation and expansion of a scenario.

|#

;; read and store the concept references into the working memory

( defun determine-concept-references ( aScenario aContext )
  "Given a scenario and a working-memory (=Context), find all the
   referenced concepts and store them in the working memory under
   the supplied keys"
  ( if-true ( and ( is-scenario? aScenario )
                  ( wm-p aContext )
                  ( not-null ( scen/get-concept-refs aScenario )))
    ( foreach x in ( scen/get-concept-refs aScenario )
      ( wm/put aContext ( first x )
        ( run-script-with-varlist ( first ( second x ))
                                  ( rest  ( second x )))))))

( defun determine-metadata-references ( aScenario aContext )
  "Given a scenario and a working-memory (=Context), find all the
   referenced metadata objects and store them in the working memory under
   the supplied keys"
  ( if-true ( and ( is-scenario? aScenario )
                  ( wm-p aContext )
                  ( not-null ( scen/get-metadata-refs aScenario )))
    ( foreach x in ( scen/get-metadata-refs aScenario )
      ( wm/put aContext ( first x )
        ( run-script-with-varlist ( first ( second x ))
                                  ( rest  ( second x )))))))

( defun evaluate-schema ( aScenario aContext )
  "Evaluate the operations indicated in a scenario schema by evaluating
   the operators specifying the operations, in the context of a
   supplied Working Memory object."
  ( if-true ( and ( is-scenario? aScenario )
                  ( wm-p aContext )
                  ( not-null ( scen/get-schema aScenario ))
                  ( determine-concept-references aScenario *context*)
                  ( determine-metadata-references aScenario *context*))
    ( let (( the-results nil ))
      ( foreach x in ( scen/get-schema aScenario )
```

```
( push ( evaluate-operator ( fetch-operator (first x))
                    (rest x) aContext)
   the-results ))
(reverse the-results ))))
```

```
#|

Scengen.lsp

Resolve all the data references in a scenario, and generate an executable
function.

|#

;; -- generate a labmda form with all the data as local variables
;; in a LET statement

( defun scenario->script ( aScenario )
  "Generate a straightforward(?) LAMBDA function for the application,
   by creating a list of the contents of the context, and splicing this
   with the script calls required (not at function level yet)"
   ( if-true ( and ( is-scenario? aScenario )
                   ( determine-concept-references aScenario *context* )
                   ( determine-metadata-references aScenario *context* ))
         ( store-script ( scen/get-name aScenario )
           `( lambda ()
              ( let ,(wm/list-vars-and-vals *context*)
                  ,@( foreach x in ( scen/get-schema aScenario )
                     ( step-to-script-call x )))))))

( defun step-to-script-call ( aStep )
  "Turn a scenario schema step into a straightforward script call"
  ( let ((tmp aStep))
    ( if tmp
      ( push '-> tmp ))
    tmp ))

( defun scenario->body ( aScenario )
  "Generate a LAMBDA function for the execution of the scenario,
   by creating a closure of the contents of the context, and splicing this
   with the script calls required"
   ( if-true ( and ( is-scenario? aScenario )
                   ( determine-concept-references aScenario *context* )
                   ( determine-metadata-references aScenario *context* ))
           `( lambda ()
              ( let ,(wm/list-vars-and-vals *context*)
                  ,@( foreach x in ( scen/get-schema aScenario )
                     ( step-to-script-call x ))))))
```

ANNEXE B : Présentation de l'enquête sur la qualité du logement en Wallonie

1. L'enquête INL sur la qualité du logement

1.1. Contexte et objectifs de l'enquête INL

L'enquête sur la qualité du logement en Wallonie a été réalisée en 1981-1982 à l'Institut National du Logement (INL), qui a depuis été supprimé.

L'objectif de l'enquête était double :

1. Rendre compte des informations fournies par l'enquête sur l'état du parc immobilier en Wallonie et sur les besoins en rénovations.

2. Analyser l'outil de collecte des données en vue de compléter la réflexion sur la méthode d'évaluation de la qualité du logement en Wallonie.

La politique envisagée visait à rencontrer plusieurs préoccupations :

- Le plein emploi

  "Le plein emploi des constructions doit être réalisé"

- L'équité sociale

  "Chaque ménage doit avoir à sa disposition un logement sain, confortable et avenant"

- L'expression des progrès techniques

  "Le logement doit être pourvu, dans toute la mesure du possible, des perfectionnements techniques, le progrès technique étant ainsi mis au service de la société"

Dans ce but, trois cibles notamment étaient retenues :

a) l'abaissement des prix de revient des constructions,

b) l'équipement des logements,

c) la coordination de l'activité en matière de logement à assurer, entre autres par la révision de la législation.

Les principes de cette politique sont nés dans la perspective de l'essor industriel et tout particulièrement dans celle de l'industrialisation de la construction, avec pour corollaire l'espoir du plein emploi et de l'extension des marchés liée à l'accroissement généralisé de la consommation grâce à la diminution des prix de revient.

Comme l'indique le projet de loi[1] : "*les modalités d'application de ces principes sont à trouver dans les réponses à faire aux questions suivantes : Combien de logements faudra-t-il construire? Où et comment construire? Quelle sera l'aide de l'Etat? Quelles sont les modifications à apporter à la législation?*".

Lors de l'installation du Conseil Supérieur du Logement et du Comité de gestion de l'INL le 7 novembre 1956, le ministre de la Santé publique et de la Famille, M. Leburton, affirmait qu'"*il ne serait pas possible au gouvernement de remplir le rôle qui lui est assigné, sans le concours d'un organisme d'étude des problèmes du logement, d'information et de documentation*".

Tel est, dans ses toutes grandes lignes, le contexte dans lequel a été préparée l'enquête sur la qualité du logement en Belgique. Il s'agit donc d'une enquête par sondage réalisée pour la première fois en 1961-1962 et renouvelée en 1971-1972. En 1981-1982, l'opération s'inscrit dans la voie tracée par les précédentes enquêtes, avec cependant une première et fondamentale différence : elle devait être menée de manière distincte en Flandre et en Wallonie, sur base de questionnaires différents.

Il existait déjà des enquêtes réalisées à l'échelon des communes, des villes ou des provinces. Mais ces démarches étaient partielles et s'appuyaient sur des méthodes et des critères disparates.

Par ailleurs, les recensements généraux fournissaient également une série d'informations statistiques sur le parc logements et sur ses occupants. Ils s'appuyaient sur une définition du logement qui coïncide avec le ménage occupant. Mais les

---

[1]Voir Projet de loi modifiant et complétant la législation relative au logement. Chambre des représentants, 13 juin 1955.

promoteurs de l'enquête INL ont toujours considéré que ces statistiques ne permettaient pas une approche qualitative du logement et de sa salubrité.

La spécificité et l'originalité, mais aussi la difficulté de l'enquête INL par rapport au recensement et aux enquêtes locales, résidaient dans la volonté d'évaluer l'état des logements en termes de salubrité et d'habitabilité de manière unifiée sur tout le territoire. Concrètement, l'un des buts de l'enquête INL était de dénombrer les logements insalubres à démolir et à reconstruire. Cette volonté impliquait pour les promoteurs une définition du logement qui soit indépendante du ménage qui l'occupe.

Les précisions apportées par Jacques Bouillon en 1973 dans son rapport aux Nations Unies permettent de circonscrire les premières limites de l'approche de la qualité du logement par l'enquête INL. Le logement est défini dans ses seules dimensions matérielles et physiques, qui constituent des éléments importants dans le fait de loger ou de se loger. L'évaluation de sa qualité se fera donc par rapport à l'état du patrimoine bâti destiné au logement. La démarche adoptée répond à une conception de la politique du logement orientée vers la gestion du parc immobilier : remplacement ou rénovation. Cette approche qualitative exige des définitions précises des notions de salubrité et d'habitabilité, notions qui sont extrêmement complexes et qui soulèvent le problème du choix des indications qui permettent de les objectiver.

1.2. La méthode d'évaluation

Pour mener à bien sa mission, l'INL s'est défini une méthode d'évaluation basée sur le choix de caractéristiques du bâtiment et du logement comme indicateur de qualité sur lequel les enquêteurs porteront un jugement de valeur.

Appréciations qualitatives et subjectives

L'évaluation de la qualité est basée sur un ensemble "d'appréciations qualitatives" de divers aspects particuliers du bâtiment, du logement et de l'équipement, et sur une appréciation globale de l'état de salubrité du logement (voir 1.3. Les critères d'évaluation de l'INL).

L'INL était pleinement conscient des risques de cette méthode subjective. Pour parvenir à limiter les risques, l'INL a travaillé sur deux plans : la formation des enquêteurs et l'élaboration des formulaires d'enquêtes.

1.3. Les critères d'évaluation de l'INL

Les appréciations ou jugements de valeur des enquêteurs portent sur diverses caractéristiques particulières des bâtiments et des logements. L'INL a, en outre, construit un indice de salubrité basé sur une appréciation globale de l'état des bâtiments et des logements qui s'appuie explicitement sur certaines appréciations particulières et constitue en quelque sorte un jugement de synthèse.

Les évaluations particulières

Les aspects particuliers du bâtiment, du logement et de l'équipement ont été évalués selon les items suivants :

Etat des toitures	bon - améliorable - mauvais
Etat des murs extérieurs	bon - améliorable - mauvais
Etat des fondations	bon - améliorable - mauvais
Etat des menuiseries extérieures	bon - améliorable - mauvais
Exiguïté des pièces	aucune - 1 - 2 ou +
Hauteurs des plafonds	bon - améliorable - mauvais
Murs extérieurs	bon - améliorable - mauvais
Planchers	bon - améliorable - mauvais
Circulation intérieure	aisée - incommode - difficile ou mauvaise
Eclairage - aération	bon - améliorable - mauvais
Humidité + origine	néant - accidentel - perm. loc. - perm. gén.
Nombre de pièces inhabitables	aucune - 1 - 2 - 3 - ...
Etat de l'installation électrique	bon - améliorable - mauvais
Nombre de points d'utilisation	suffisante - insuffisante
Isolation acoustique	bonne - faible - mauvaise

Isolation thermique	très bonne - bonne - faible - mauvaise
Ventilation cuisine	bonne - améliorable - mauvaise
Ventilation salle de bain	bonne - améliorable - mauvaise
Accès au logement	aisé - incommode - mauvaise
Densité de population	normale - trop faible - exagérée

"Etat de salubrité" et "structure du logement"

L'indice synthétique vise à rendre compte de la qualité des logements et des bâtiments. Il ne résulte pas d'une addition d'appréciations particulières mais d'un nouveau jugement des enquêteurs qui attribuent à chaque logement un code qui correspond aux catégories définies dans les recommandations de l'INL et qui, d'après ces recommandations, s'appuie sur "l'analyse des aspects physiques et structurels du logement".

L'indice INL de la qualité du logement est subdivisé en 3 catégories :

- logements salubres,
- logements insalubres améliorables,
- logements insalubres non améliorables.

Les logements salubres et les logements insalubres sont eux-mêmes subdivisés selon l'appréciation de la structure des logements en 3 catégories.

Les logements salubres peuvent être :

1. de bonne structure,
2. fonctionnellement inadaptés et de structure améliorable,
3. fonctionnellement inadaptés et de mauvaise structure.

Les logements insalubres améliorables peuvent être :

1. de bonne structure,
2. de structure améliorable,
3. de mauvaise structure.

La notion de salubrité renvoie surtout aux qualités physiques du bâtiment.

La notion de structure renvoie à la structure du logement.

2. Intérêt actuel de l'enquête

L'enquête INL constitue un objet d'étude à trois niveaux :

1. Sur le plan méthodologique, le risque lié à l'évaluation subjective des enquêteurs implique de vérifier la fiabilité de la démarche.

2. D'un point de vue pratique, le sondage INL peut être considéré et utilisé comme un outil à perfectionner sans doute, permettant de définir les marchés potentiels dans le secteur de l'assainissement et de la rénovation des bâtiments.

3. Sur le plan analytique, le sondage devrait permettre d'amorcer une pré-étude des facteurs qui favorisent la dégradation du patrimoine immobilier.

Il permet d'amorcer l'étude de ces facteurs par l'analyse des relations entre les variables qu'il est possible de construire à partir du sondage.

Choix des variables et schéma d'analyse

Pour maximiser l'exploitation des résultats du sondage, les multiples données collectées ont été classées en 6 catégories ou modules de variables :

1. Les variables d'évaluation

Ce sont les variables qui correspondent à des réponses aux questionnaires basées sur un jugement de valeur des enquêteurs. Parmi elles figurent les évaluations particulières, c'est-à-dire les évaluations des composantes des bâtiments, des caractéristiques des logements et des équipements ainsi que l'indice de la salubrité et de la structure du logement.

2. Les variables d'identification

Sur base des données disponibles dans le questionnaire, 6 variables d'identification ont été retenues : la date de construction, le type de bâtiment, l'usage du bâtiment, l'implantation provinciale, l'implantation selon le degré d'urbanité, le type de propriétaire.

3. Les variables sociales

Ce sont les principales caractéristiques des occupants collectées par le questionnaire : âge du chef de ménage, nationalité, activité socioprofessionnelle, état civil, taille du ménage, nombre d'enfants.

4. Les caractéristiques des bâtiments

C'est-à-dire les données objectives concernant les caractéristiques physiques des bâtiments : dimensions, matériaux des toitures et des murs...

5. Les caractéristiques des logements

Il s'agit essentiellement des données collectées par l'enquête au sujet du logement concernant les dimensions, la densité d'occupation, le nombre de pièces...

6. Les caractéristiques des équipements

C'est-à-dire l'absence ou la présence des principaux équipements primaires (eau, égouttage, électricité, wc), équipements secondaires (salle de bain, chauffage, garage,...)

En ce qui concerne l'indice de salubrité et de structure du logement, pour la clarté de l'étude, il est paru nécessaire de distinguer l'évaluation de la salubrité du bâtiment de l'évaluation de la structure du logement.

En conséquence, l'indice d'évaluation de l'état du logement sera divisé en 2 variables qui seront traitées séparément :

- l'évaluation du bâtiment dont rend compte "l'état de salubrité"
- l'évaluation du logement dont rend compte "l'état de la structure"