Maintenance logicielle : analyse d'impact, problématique et mise en oeuvre

Samuel Adesoye Ajila

▶ **To cite this version:**

Samuel Adesoye Ajila. Maintenance logicielle : analyse d'impact, problématique et mise en oeuvre. Informatique [cs]. Université Henri Poincaré - Nancy 1, 1995. Français. NNT : 1995NAN10029 . tel-01747515

HAL Id: tel-01747515
https://hal.univ-lorraine.fr/tel-01747515

Submitted on 29 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

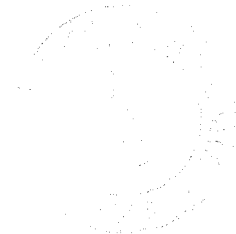
Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm



Université Henri Poincaré – Nancy I

Département de Formation Doctorale en Informatique **École Doctorale IAE + M**

Maintenance logicielle : Analyse d'impact, problématique et mise en œuvre

THÈSE

présentée et soutenue publiquement le 19 juin 1995

pour l'obtention du

Doctorat de l'Université Henri Poincaré – Nancy I
(Spécialité Informatique)

par

Samuel Adesoye AJILA

Composition du jury

Président :	Marie Christine HATON	Prof., Univeristé Henri Poincaré Nancy 1
Rapporteurs :	Claude CHRISMENT	Prof., Université Paul Sabatier, Toulouse
	Michael GRIFFITHS	Prof., Université de Nantes, Nantes
	Marie Christine HATON	Prof., Université Henri Poincaré Nancy 1
Examinateurs :	Jean Claude DERNIAME	Prof., ENSEM, INPL, Nancy
	Henri BASSON	MC., Université du Littoral, Calais

Centre de Recherche en Informatique de Nancy

Remerciements

Cette thèse est le résultat de quatre années d'activité de recherche au Centre de Recherche en Informatique de Nancy (CRIN), au sein de l'équipe Génie Logiciel de Monsieur le Professeur Jean Claude DERNIAME que je tiens ici à remercier chaleureusement. Les raisons de le remercier ne manquent pas : tout d'abord pour son accueil dans son équipe de recherche, pour l'encadrement et le suivi de mon travail, et pour ses qualités humaines.

Je tiens à remercier Monsieur Henri BASSON, Maître de Conférence à l'université du Littoral, Calais, qui a suivi cette thèse de son début jusqu'à sa soutenance et pour le temps précieux qu'il m'a accordé. Son soutien moral ne sera pas oublié.

Je tiens à exprimer ma gratitude à Madame le professeur Marie Christine HATON, qui a bien voulu être rapporteur de cette thèse et dont les nombreux conseils m'ont permis d'arriver à la version actuelle de ce rapport. Je suis fier qu'elle ait accepté de présider mon jury.

Je remercie Monsieur le professeur Claude CHRISMENT, professeur à l'université Paul Sabatier, Toulouse, d'être rapporteur de cette thèse et de m'avoir permis, grâce à ses commentaires et remarques sur mon travail, d'améliorer mon rapport et d'arriver à la version actuelle.

Je remercie Monsieur le professeur Michael GRIFFITHS, professeur à l'université de Nantes, Nantes, qui a accepté d'être rapporteur de cette thèse, pour le temps qu'il a consacré, et pour la qualité de ses remarques.

Je remercie vivement ceux qui ont accepté de lire les toutes premières lignes de ce document, Ali KABA et Amos DAVID. Merci à François CHAROY qui a accepté de relire le manuscrit.

Je voudrais aussi remercier tous les membres de l'équipe Génie Logiciel et plus particulièrement Mme MARCHAND, Nacer BOUDJLIDA, Hala SKAF, Ali KABA, Abdellatif MEZRIOUI, Malek BOUNAB, Pascal MOLLI, et Alexandre ROSSELL.

Au risque d'en oublier, je voudrais profiter de l'occasion qui m'est donnée pour remercier particulièrement : Mr. & Mme. AKINDELE, Mr. & Mme. DAVID, Mr. & Mme. SHITU, Mr. & Mme. SALAU, Mr. ANIGBOGU, ainsi que les *frères* et *sœurs* en Jésus Christ du Centre Evangélique de Pentecôte (CEP) de Nancy, tous ceux qui ont contribué directement ou non au travail que je présente aujourd'hui, et tous ceux qui m'ont formé et aidé tout au long de mes études et de mes recherches.

Pour finir, je pense tout particulièrement à mon épouse, Victoria, à son soutient, sa patience et son amour, et à Ibukunoluwa, Oluwatobi et Ayooluwa pour leurs patiences.

Samuel A. AJILA
Nancy le 19 juin 1995.

À Dieu soit la gloire pour les choses qu'Il a faites;

À Victoria, Ibukunoluwa, Oluwatobi et Ayooluwa.

Résumé

Notre travail concerne l'évaluation *a priori* de l'impact des changements réalisés sur les objets logiciels.

Il est parfois difficile de comprendre toutes les conséquences d'un changement et de garantir qu'il n'y aura pas d'effets de bord gênants au niveau de l'ensemble du système logiciel. Il est donc utile d'apporter une aide à la décision de changement permettant de comprendre et d'analyser un changement avant de le faire. En particulier on doit prendre en compte la complexité des liens qui existent entre les différents composants du logiciel.

Pour tenir compte des conséquences d'un changement donné, il faut avoir une connaissance précise sur les liens entre les objets logiciels et analyser l'impact du changement.

Pour cette raison, nous avons étudié les caractéristiques souhaitables d'une analyse d'impact qui permettront de comprendre, de comparer, et d'évaluer des approches différentes et nous proposons un cadre générique d'analyse d'impact. Ce cadre générique comporte trois parties: une approche d'analyse d'impact, la structure de cette approche, et la mesure d'efficacité de l'approche.

Nous proposons ensuite une approche d'analyse d'impact d'un changement et pour cela nous avons :

- défini un modèle générique à base de connaissances sur la nature des liens entre les objets logiciels et sur les règles de propagation du changement;
- validé ce modèle avec un prototype limité à deux étapes de cycle de vie, la conception et le code. Pour cette validation, nous avons choisi la méthode HOOD pour la conception et le langage de programmation Ada pour le code.

Les outils existants d'analyse de l'impact du changement sont basés sur les méthodes de simulation (les méthodes dites *algorithmiques*) et ils n'adressent souvent que le code et parfois même qu'une partie du code. Nous avons montré dans ce travail qu'avec le modèle proposé, on peut dépasser la limite des méthodes de simulation et donner des informations plus détaillées sur l'effet d'un changement.

Mots-clés: Maintenance logicielle, Analyse d'impact, Propagation d'un changement, Objets logiciels, Système de vues, Relations de dépendances, Base de connaissances

Table des matières

Table des figures xiii

1 Introduction 1

1.1 L'introduction et la contribution de la thèse 1

1.2 L'organisation de la thèse 5

2 La problématique générale de l'évaluation des conséquences d'un changement 7

2.1 Introduction 7

2.2 Le problème du changement 8

2.2.1 Définitions 8

2.2.2 Typologie des changements 11

2.3 Evolution de logiciel 13

2.4 Les types de systèmes logiciels 15

2.4.1 Les S-systèmes de Lehman 15

2.4.2 Les P-systèmes de Lehman 15

2.4.3 Les E-systèmes de Lehman 17

2.4.4 Les changements pendant le cycle de vie de logiciel 18

2.5 L'analyse de l'impact d'un changement 19

2.5.1 Un "cadre générique" d'analyse d'impact 20

2.5.2 Une approche d'analyse d'impact 21

2.5.3 Les parties fonctionnelles d'une approche d'analyse d'impact . . 23

2.5.4 La mesure d'efficacité d'une approche d'analyse d'impact 24

2.5.5 Les concepts de la mesure d'efficacité 27

2.6 Conclusion 28

3 État de l'art 31

3.1 Introduction 31

3.2 Les outils d'analyse statique du code 31

3.2.1 Introduction 31

3.2.2 L'analyseur de code 32

3.2.3 Les outils d'aide à la documentation 33

3.2.4 Les outils de références croisées 33

3.2.5 Les outils de restructurations 34

3.3 La technique d'analyse d'effet d'un changement 34

3.3.1 L'algorithme d'identification d'effet de bord du changement dans le code 35

3.3.2 L'applicabilité 37

3.3.3 L'utilité 37

3.3.4 Les outils de comparaison de code 38

3.4 Les méthodes d'analyse dynamique 38

3.4.1 Le test de la couverture logique 38

3.4.2 L'exécution comparative 39

3.5 Les méthodes d'évaluation symbolique 39

3.5.1 Introduction 39

3.5.2 La terminologie et la notation 40

3.5.3 L'exécution symbolique 41

3.5.4 L'évaluation symbolique globale 42

3.5.5 L'évaluation symbolique dynamique 43

3.5.6 Les moniteurs d'exécution 43

3.6 La technique d'analyse de dépendances 44

3.6.1 Le graphe de dépendances 45

3.6.2 La classification de dépendances 46

3.6.3 L'utilisation du graphe de dépendances 46

3.7 Conclusion 47

4 Le modèle WHAT-IF 49

4.1 Introduction 49

4.2 La problématique 49

4.3 L'analyse conceptuelle 50

4.4 Le système de vues d'un logiciel 52

4.5 Les types de systèmes de vues 53

4.5.1 Le type 1 et 2 : Les vues programme et structurelle 53

4.5.2 Le type 3 : Le système de vue architecturale 53

4.5.3 Le type 4 : Le système de vue basé sur un domaine spécifique . . 54

4.5.4 Le type 5 : Le système de vue orienté cycle de vie de logiciels . . 54

4.5.5 La combinaison des vues 55

4.6 WHAT-IF système de vues 57

4.7 Les relations de dépendances 60

4.7.1 La taxinomie des relations de dépendances 61

4.8 Les types de changements 65

4.9 La fonction f_{impact} et ses propriétés 66

4.9.1 L'approche "faible" 68

4.9.2 L'approche "forte" 69

4.10 Champ d'application du modèle WHAT-IF 71

4.11 Conclusion 73

5 Le prototype du modèle WHAT-IF 75

5.1 L'introduction 75

5.2 L'architecture du prototype WHAT-IF 76

5.3 L'extracteur 77

5.3.1 Les analyseurs 77

5.3.2 Le processeur 77

5.3.3 La spécification des relations de dépendances directes pour le code Ada 81

5.3.4 Les entités et la spécification des relations de dépendances directes pour les objets HOOD 82

5.3.5 La spécification de la relation de correspondance entre les objets Ada et les objets HOOD . . . 84
5.4 Le transformateur . . . 85
5.5 L'abstracteur . . . 87
5.5.1 Les requêtes et le gestionnaire de requêtes . . . 88
5.5.2 La requête WHAT-IS . . . 89
5.5.3 La requête WHAT-IF . . . 91
5.5.4 La requête WHAT-IF-Ada et le type de changements au niveau Ada . . . 92
5.5.5 La requête WHAT-IF-HOOD et les types de changement dans HOOD . . . 96
5.5.6 Les requêtes "WHAT-IF-Local" et "WHAT-IF-Global" . . . 99
5.5.7 La requête WHAT-LIST . . . 103
5.5.8 L'interface utilisateur . . . 104
5.6 La conclusion . . . 108

6 Conclusion **113**
6.1 Bilan . . . 113
6.2 Perspectives . . . 115

Bibliographie **119**

A HOOD : Une méthode de conception hiérarchisée orientée objets **131**
A.1 L'introduction . . . 131
A.2 La conception orientée objet et HOOD . . . 131
A.2.1 La conception d'objet . . . 131
A.2.2 Le modèle objet HOOD . . . 132
A.3 Caractéristiques de HOOD . . . 132
A.3.1 La formalisation du concept d'objet . . . 133
A.3.2 La description des relations inter objets . . . 134
A.4 Description formelle des objets HOOD . . . 137
A.4.1 La translation de HOOD vers Ada . . . 137

B Global-Exemple 1 : Système de base de données 139

B.1 La définition du problème 139

B.2 Le document de conception 141

B.2.1 L'introduction 141

B.2.2 L'identification des objets 141

B.2.3 L'identification des opérations 141

B.3 Formalisation de la solution : la description du squelette de l'objet (ODS) 145

B.4 Le code Ada généré et enrichi 146

B.5 Les tables générées par le processeur 150

B.5.1 La table d'identificateurs 150

B.5.2 La table de sous-programmes et la table des paramètres de sous-programmes 152

B.5.3 La table d'importation 152

B.5.4 La table des objets génériques 153

B.6 La base de faits 153

C Global-Exemple 2 : Surveillance d'environnement 157

C.1 La définition du problème 157

C.2 Le document de conception 157

C.2.1 L'introduction 157

C.2.2 L'identification des objets 158

C.2.3 L'identification des opérations 158

C.3 Formalisation de la solution : la description du squelette de l'objet (ODS) 160

C.4 Le code Ada généré et enrichi 163

C.5 Les tables générées par le processeur 166

C.5.1 La table d'identificateurs 166

C.5.2 La table d'identificateurs *entry* et la table des paramètres *entry* 167

C.5.3 La table d'importation 168

C.5.4 La table des objets génériques 168

C.6 La base de faits 168

Table des figures

1.1 La maintenance du logiciel 2
1.2 Les activités d'un système informatique 3

2.1 La nouvelle version Σ_δ de Σ 9
2.2 La distribution de l'effort de changement [Parikh 86, Laski 92] 13
2.3 Le procédé de changement 14
2.4 Les cycles de vie de S-systèmes, P-systèmes et E-systèmes 16
2.5 L'effet de changement pendant le développement de système logiciel . . 18
2.6 L'utilisation du "cadre générique" d'analyse d'impact 21
2.7 Les différente parties du "cadre générique" 21
2.8 L'approche de l'analyse d'impact : Point de vue utilisateur 22
2.9 Les parties fonctionnelles de l'approche de l'analyse d'impact 25
2.10 La mesure d'efficacité de l'ensemble des éléments d'analyse d'impact . . 26
2.11 La relation entre le début de l'impact(DI$) et l'impact estimé(IE$) . . 28
2.12 La relation entre l'impact estimé(IE$) et système$ 29

4.1 La combinaison des vues 56
4.2 Un exemple de modification 58
4.3 Le système de vues WHAT-IF 59
4.4 La traçabilité horizontale 60
4.5 La traçabilité verticale 61
4.6 La fonction f_{impact} 67
4.7 L'architecture globale du modèle WHAT-IF 70

5.1 L'architecture du prototype WHAT-IF 78
5.2 L'analyseur 79
5.3 Exemple de la table d'identificateur Ada 79
5.4 Exemple de la table d'identificateur HOOD 79
5.5 La table d'objet Ada 80
5.6 La table d'objet Ada 80

5.7 La table de correspondances entre les objets Ada et les objets HOOD . 84
5.8 L'abstracteur 87
5.9 L'interface utilisateur du prototype WHAT-IF 105
5.10 Les différents menus 106
5.11 L'information d'aide sur la requête WHAT-IS 107
5.12 Le résultat de la requête **WHAT-IF-Ada(sensor, PAT)** 109
5.13 Le résultat de la requête **WHAT-IS(monitor_temperatures, HOOD)** 110

6.1 USM Unified Model 115
6.2 L'intégration du modèle WHAT-IF dans un environnement de développement de logiciel 116
6.3 Le "reverse engineering" et le procédé de transformation entre les différents niveaux d'abstraction 116
6.4 Le "reverse engineering" de code Ada 117

A.1 La représentation d'un objet HOOD 133
A.2 La représentation d'un objet actif 134
A.3 La décomposition des objets passifs 135
A.4 La décomposition des objets actifs 136
A.5 Les flots de contrôle, de données et des exceptions 137

B.1 Le problème du système de base de données 140
B.2 La description graphique de l'objet *inquire_about_project_data* (Diagramme HOOD) 142
B.3 La description graphique de l'objet *project*(Diagramme HOOD) 143
B.4 La description graphique de l'objet *inquiry_operations* (Diagramme HOOD) 144

C.1 Le problèlem de surveillance d'environnement 158
C.2 La description graphique (Diagramme HOOD) 159

Chapitre 1

Introduction

1.1 L'introduction et la contribution de la thèse

La réalisation d'un logiciel représente un investissement important qu'il faudra amortir par une durée d'exploitation du logiciel suffisamment longue, durée pendant laquelle l'environnement et les spécifications peuvent changer. Lors d'un tel changement, une modification de l'environnement ou des spécifications entraîne, la plupart du temps, une modification des programmes.

Parmi les diverses qualités que l'on peut attendre d'un logiciel, l'efficacité et la fiabilité sont sans doute les plus connues. D'autres qualités telles que la facilité avec laquelle on passera d'une version à une autre et la capacité de modifier un logiciel pour tenir compte d'un changement d'environnement ou de spécification tout en lui conservant ses qualités initiales sont aussi très importantes. La facilité de modification est particulièrement importante économiquement et on peut démontrer cette importance en étudiant le coût de la maintenance et les différents types de maintenance.

La maintenance d'un système d'application opérationnel est souvent perçue comme une phase beaucoup moins importante que les phases de conception et de développement de ce système. Ce n'est pourtant pas la réalité sur le plan économique. Les données existantes montrent que le coût actuel de la maintenance de logiciel est entre 32% à 75% du coût total de développement [Sok 80, Parikh 86, Loyall 93]. L'un des problèmes qui se pose est le fait qu'on doit prendre en compte les conséquences d'un changement au niveau de l'ensemble du système logiciel.

Notre travail concerne l'évaluation *a priori* de l'impact de changements réalisés sur les objets logiciels.

Il y a plusieurs raisons qui peuvent motiver un changement sur les objets logiciels [Donahoo 80, Lehman 85, Thebaut 86, Cooper 89, Livadas 92, Loyall 93, Ajila 94] (cf figure 1.1) :

- la performance du logiciel doit être maintenue (la maintenance perfective) : la perfection du logiciel consiste à éliminer les faiblesses, augmenter les performances ou améliorer d'autres qualités du logiciel telles que la lisibilité, robustesse, etc..

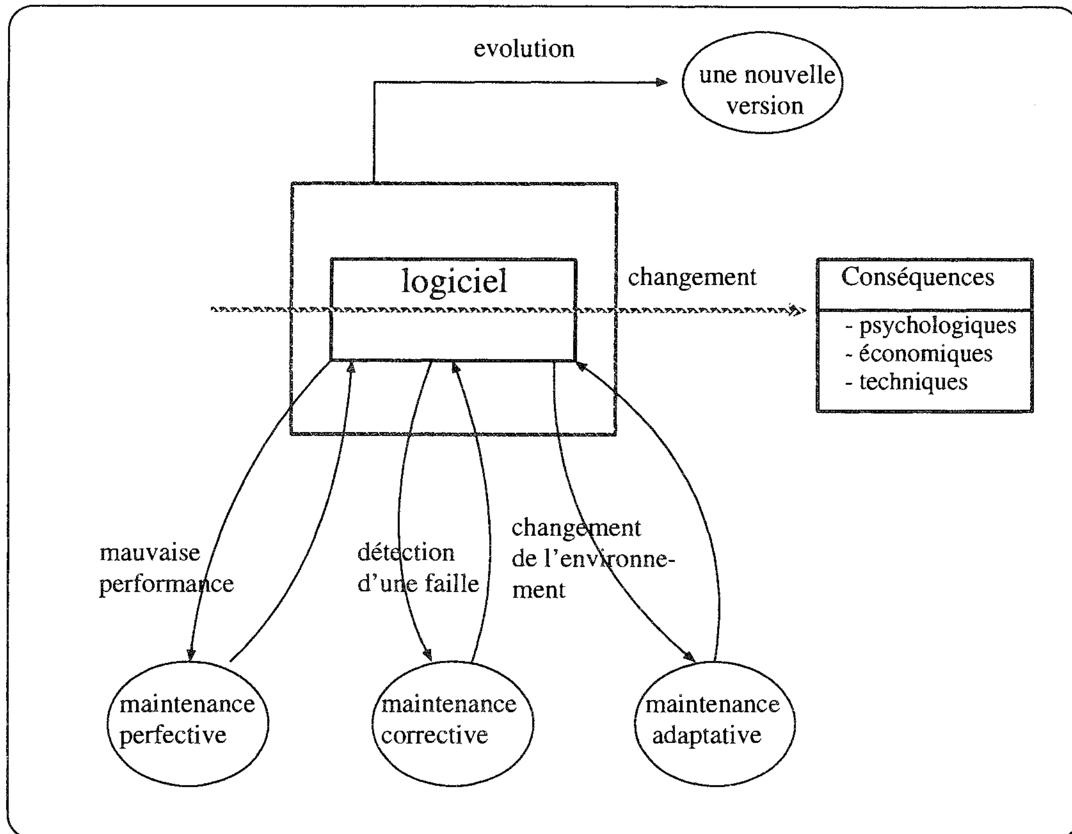


FIG. 1.1 - *La maintenance du logiciel*

- les erreurs de spécification, de conception, et de codage doivent être corrigées (la maintenance corrective) : réaction à l'apparition de failles c'est-à-dire recherche des erreurs et/ou du défaut quand la version actuelle du programme n'est pas correcte.
- le produit final doit opérer dans divers environnements (la maintenance adaptative) : réaction à des changements d'environnement : modification de matériel ou de système support par exemple.
- le logiciel doit évoluer : réaliser une nouvelle version du logiciel.

Le contrôle de la qualité est un problème clef dans la modification de logiciel. Les changements dans un logiciel peuvent introduire des erreurs qui se manifesteront plus tard avec un coût élevé. Le problème de la modification de logiciel provient du fait qu'un changement dans une partie du logiciel doit prendre en compte la complexité des liens qui existent entre les différents composants du logiciel (figure 1.2). Il est parfois difficile de comprendre toutes les conséquences d'un changement et de garantir qu'il n'y aura pas des effets de bords gênants au niveau de l'ensemble du système logiciel . Il est donc utile d'apporter une aide à la décision de changement permettant de le comprendre et de l'analyser. [Lieberherr 89, Cooper 89, Laski 92, Loyall 93].

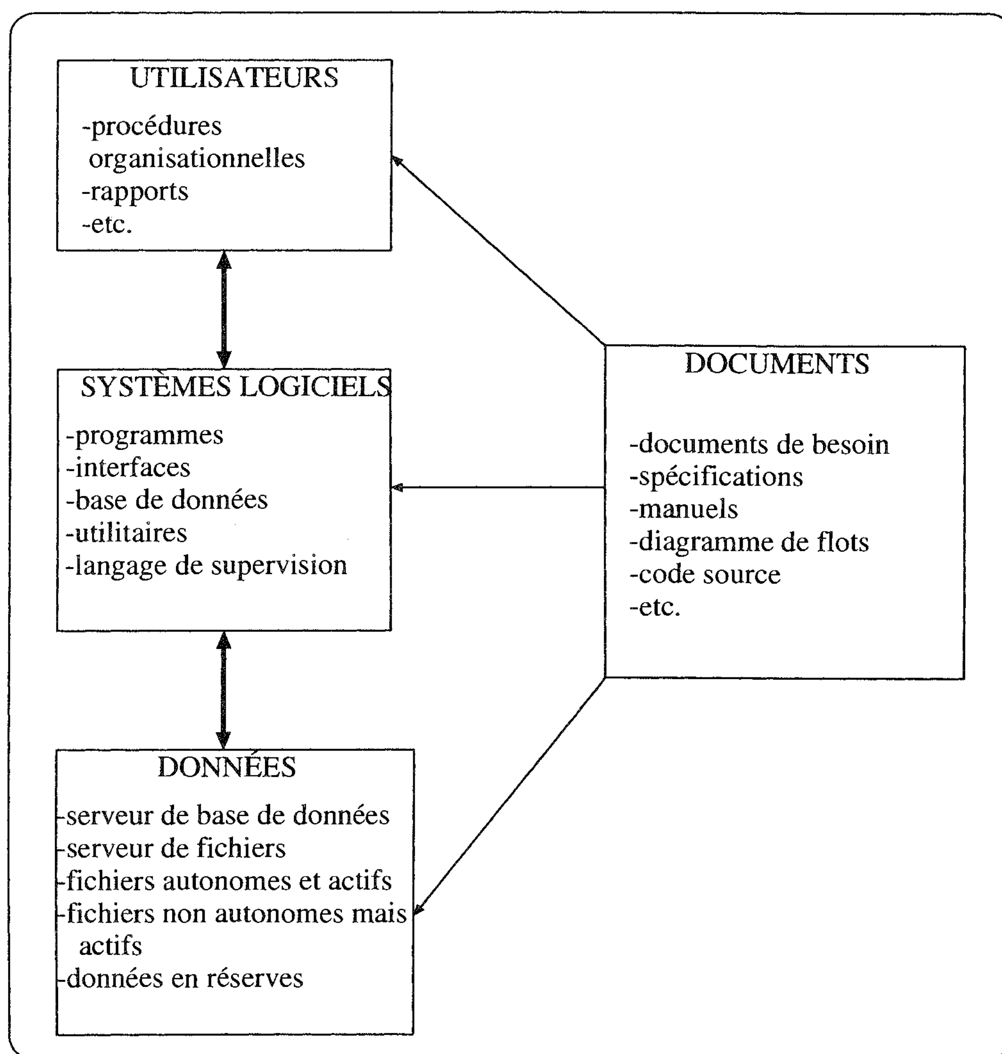


FIG. 1.2 - *Les activités d'un système informatique*

Parmi les types de conséquences possibles d'un changement (cf figure 1.1) nous n'examinerons pas ici les aspects psychologiques et économiques mais les aspects techniques.

Pour tenir compte des conséquences d'un changement donné, il faut :

1. bien comprendre sa signification. Dans ce sens, nous proposons dans cette thèse une définition d'un changement au niveau de logiciel dans la problématique générale.

2. avoir une connaissance précise sur les liens entre les objets logiciels et analyser son impact. Analyser les impacts du changement suppose que l'on ait des modèles de celui-ci adapté aux types de facteurs que l'on veut évaluer.

Pour cette raison, nous étudions dans la thèse les caractéristiques souhaitables d'une analyse d'impact qui permettront de comprendre, de comparer, et d'évaluer des approches différentes et nous proposons un cadre générique d'analyse d'impact. Ce cadre générique comporte trois parties: une approche d'analyse d'impact, la structure de cette approche, et la mesure d'efficacité de l'approche.

Nous proposons ensuite une approche d'analyse d'impact et pour cela nous avons :

1. défini un modèle générique à base de connaissances sur la nature des liens entre les objets logiciels et les règles de propagation de changement. Les outils existants d'analyse de l'impact du changement sont basés sur les méthodes de simulation (les méthodes dites *algorithmiques*) et la plupart de ces outils ne concernent que le code et parfois même, seulement une partie du code. Or, on sait qu'un logiciel n'est pas constitué uniquement du code. Il y a le document d'analyse de besoins, la spécification, et la conception. Donc, pour nous, un logiciel est un ensemble de composants incluant le code. Une contribution importante de cette thèse est la démonstration que l'analyse et la propagation de l'impact du changement ne se limite pas à une étape donnée mais couvre un ensemble d'étapes du cycle du vie de logiciel. En fait, le système de vues proposé dans le modèle est basé sur une combinaison de deux systèmes de vues : le système de vues *cascade* et la vue liée à une domaine spécifique. En effet le système de vues permet l'expression plus détaillée d'un système logiciel. En outre la vue liée à un domaine spécifique permet la spécification des relations de dépendances entre les objets d'une phase donnée et ceux qui lui correspondent dans les autres phases du cycle du vie de logiciel. Enfin la manipulation de la connaissance issue de différents composants d'un système logiciel repose sur une *représentation complète*, une *méthode d'inspection*, et une *assistance intelligente*.

2. validé ce modèle dans un prototype limité à deux étapes de cycle de vie, à savoir, la conception et le code. Dans ce prototype nous avons choisi la méthode HOOD pour la conception et le langage de programmation Ada pour le code.

1.2 L'organisation de la thèse

Après cette introduction, la thèse est structurée en cinq chapitres :

- le chapitre deux présente la problématique générale de l'étude des changements de logiciel. Il présente le problème du changement au niveau du logiciel, l'évolution d'un système logiciel, les types de systèmes logiciels, et un cadre générique d'analyse d'impact.
- le chapitre trois est dédié à l'état de l'art. Après une analyse des méthodes basées sur l'analyse d'effet de bord de changement, les méthodes d'analyses dynamiques, les méthodes d'évolutions symboliques, et les outils d'analyse de programme, nous proposons la technique d'analyse de dépendances.
- le chapitre quatre présente notre proposition du modèle générique à base de connaissances pour évaluer l'impacts des changement de logiciel.
- le chapitre cinq présente un prototype fondé sur le modèle WHAT-IF pour valider les idées proposées dans ce modèle.
- le chapitre six est consacré à la conclusion en situant notre proposition par rapport à l'état de l'art et aux objectifs initiaux.

Chapitre 2

La problématique générale de l'évaluation des conséquences d'un changement

2.1 Introduction

Beaucoup de logiciels subissent des changements plusieurs fois pendant leur durée de vie. Un logiciel peut être modifié plusieurs fois pour incorporer les changements au niveau de la conception et pour corriger les parties incorrectes. Après l'installation du logiciel, le cycle de changements dus à la maintenance commence. Ces changements sont effectués pour introduire un nouveau besoin, une nouvelle machine, un nouvel environnement, ou pour corriger les erreurs logicielles issues du programme.

Dans la réalisation de logiciels, on est souvent très préoccupé par les performances réalisées, au détriment parfois de la sûreté de fonctionnement et souvent de la facilité de modification. Ceci se traduit par de lourdes charges de maintenance, ce qui accroît le coût.

En effet, le coût d'un logiciel est en fait la somme des coûts d'étude, d'analyse, de codage, de mise au point, mais aussi de maintenance tout au long de la vie de ce logiciel.

La maintenance de logiciel est quelque chose de complexe, très chère et qui consomme beaucoup de temps. Les données existantes montrent que le coût actuel de la maintenance de logiciel est entre 32% à 75% du coût total de développement [Sok 80, Parikh 86, Loyall 93].

L'un des problèmes clefs qui se pose est le fait que chaque modification doit prendre en compte la complexité des relations de dépendances qui existent entre les différents composants du logiciel (figure 1.1). Il est clair qu'un système logiciel qu'on n'a pas construit soi même est très difficile à comprendre. La difficulté de compréhension existe même si on a participé à l'élaboration et à la construction du système, surtout quand celui-ci est un grand projet logiciel. Or, pour prendre en compte les conséquences d'un

changement donné, il faut :

- comprendre la signification du changement,
- avoir une connaissance plus ou moins précise sur les relations de dépendances,
- analyser l'impact du changement.

Il faut donc des supports qui permettent de comprendre et d'analyser chaque changement [Lieberherr 89].

Dans ce chapitre, nous présentons la problématique générale de l'analyse d'impact en commençant par le problème du changement au niveau du logiciel, l'évolution d'un système logiciel, les types de systèmes logiciels, et en fin la présentation d'un cadre générique (*framework*) d'analyse d'impact.

2.2 Le problème du changement

Tandis que le "Prism Model of Change" de Madhavji [Madhavji 92] décrit le changement dans un procédé de développement de logiciel (i.e. l'évolution de procédé de développement), la notion de changement que nous définissons ici est en rapport avec le "produit" (i.e. le logiciel). Nous allons limiter la définition suivante à des fonctions (ou procédures) dans un code. Nous avons introduit cette limitation pour réduire la complexité de la définition.

2.2.1 Définitions

Soit Σ un programme composé d'un ensemble de fonctions $F_1, ..., F_n$.

<u>Définition</u> 1 :

Un changement δ sur Σ est défini comme :

$\delta \stackrel{\text{def}}{=} \{(F_i, F_i') \mid F_i \in \Sigma,\ F_i' \text{ est la nouvelle version de } F_i\}$

La nouvelle version Σ_δ de Σ sur l'influence du changement δ est défini comme (cf. figure 2.1):

$\Sigma_\delta \stackrel{\text{def}}{=} (\Sigma - A) \cup B$ où

$\mathcal{A} = \{F_i \mid (F_i, F_i') \in \delta\}$
$\mathcal{B} = \{F_i' \mid (F_i, F_i') \in \delta\}$

<u>Définition</u> 2 :

Soit Σ et Σ_δ spécifiés par P(Σ)Q et $P'(\Sigma_\delta)Q'$ où

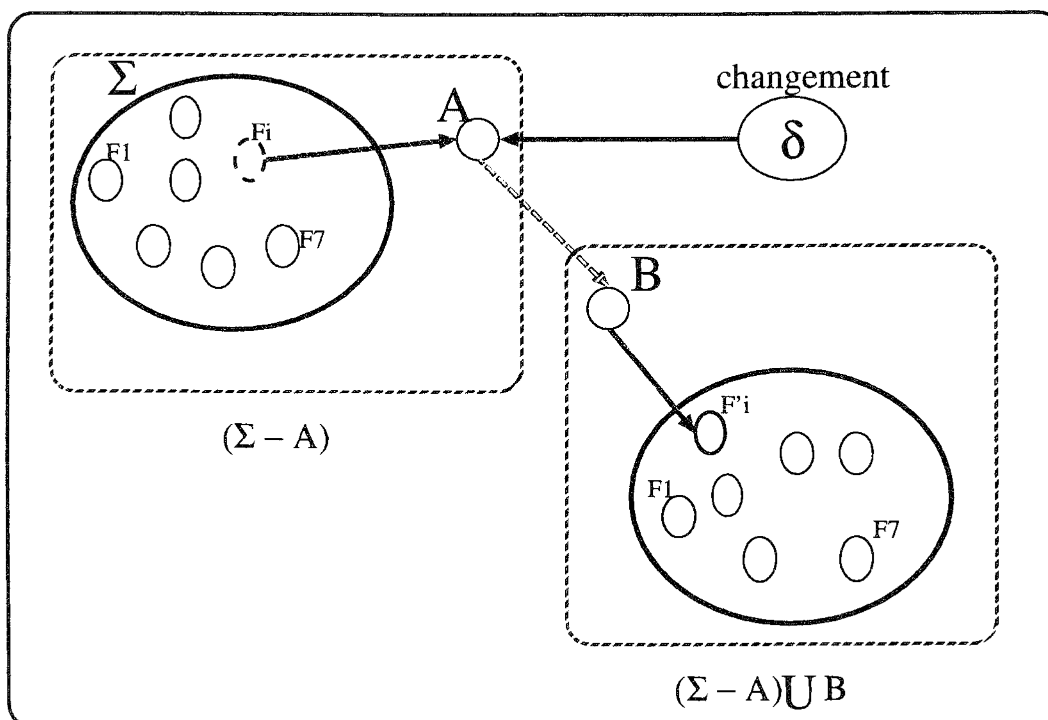


FIG. 2.1 - *La nouvelle version* Σ_δ *de* Σ

Q et P sont la post- et la pré- conditions désirées de Σ et
Q' et P' sont la post- et la pré- conditions désirées de Σ_δ,

Un changement δ sur Σ est *valide* si la post-condition Q' de Σ_δ est vraie.

Un changement dans une fonction, sans tenir compte des effets de bords dans d'autres fonctions, peut entraîner le système vers un état non satisfaisant. Donc, le changement dans une fonction exige des modifications correspondantes dans d'autres fonctions qui ont des relations avec la fonction modifiée. Ce qui signifie que le changement doit être *valide*.

Pour avoir un changement *valide*, on doit pouvoir analyser les effets du changement (dont l'impact du changement) et les propager d'une façon efficace.

Définition 3 :

Pour caractériser le changement *valide*, nous introduisons la notion de changement *complet*.
Soit Q_i la post-condition de F_i et P_i la pré-condition la plus faible requise pour que la fonction F_i satisfasse la post-condition.
Soit Q_i' la post-condition de F_i' et P_i' la pré-condition la plus faible requise pour que la fonction F_i' satisfasse la post-condition.

Soit $\delta_\Sigma(F_i)$ l'ensemble de fonctions qui peuvent appeler la fonction F_i dans Σ, i.e.
$\delta_\Sigma(F_i) = \{F_j \in \Sigma \mid F_j \text{ peut-appeler } F_i\}$.

Un changement δ sur Σ est *complet* si les deux conditions suivantes sont vraies pour tout $(F_i, F_i') \in \delta$ et pour chaque $F_j \in \delta_\Sigma(F_i)$ tel que F_j ne change pas entre deux versions, i.e $(F_j, F_j') \notin \delta$:

1. Soit S la post-condition que la fonction F_j aspire que F_i satisfasse. Alors, $Q_i' \longrightarrow S$

2. Soit R la pré-condition avec laquelle F_j appelle F_i. Alors, $R \longrightarrow P_i'$.

La première condition indique que la fonctionnalité de la nouvelle version d'une fonction doit être le sur-ensemble des conditions que la fonction d'appel attend. Si ce n'est pas le cas, alors le changement n'est pas visible par la fonction d'appel et par conséquent on doit la modifier. La deuxième condition exige que la nouvelle version doit être capable de fonctionner sur les conditions dans lesquelles la fonction d'appel invoque l'ancienne version. Sinon, la fonction d'appel doit être informée et elle doit être modifiée pour qu'elle puisse appeler la nouvelle version sous les nouvelles conditions.

Exemple :

Considérons un changement dans une fonction f_i de tri. f_i trie les entiers et on veut modifier f_i pour qu'elle puisse trier les autres type de données (qu'elle devienne polymorphe). Considérons aussi qu'il existe une autre fonction f_j qui appelle f_i dans l'ancienne version du programme. f_i qui est maintenant polymorphe possède parmi d'autres plus de paramètres qu'avant et il faut que f_j prenne en compte ce changement. Dans ce cas, il y a deux possibilités : soit on modifie f_j pour qu'elle puisse appeler f_i dans son nouvel état ou soit on écrit une autre fonction f_k qui fait la correspondance entre l'ancien format des paramètres et le nouveau format. Dans cette deuxième hypothèse on doit passer par f_k pour appeler f_i. Cette possibilité est particulièrement intéressante s'il y en a plusieurs fonctions qui utilisent f_i.

Il y a deux choses dans cet exemple : premièrement, f_i dans son nouvel état peut toujours trier les entiers (i.e $Q_i' \rightarrow S$, condition 1); deuxièmement, f_j (soit après la modification ou par l'intermédiaire de f_k) peut toujours appeler f_i (i.e. $R \rightarrow P_i'$, condition 2). Dans l'hypothèse où f_j est modifiée alors, on a plus de f_j mais f_j'. Dans ce cas on doit prendre en compte l'effet de cette modification pour en déterminer la propagation a posteriori.

Définition 4 :

Dans les définitions 1, 2, et 3 nous avons considéré le changement dit "statique". Un changement est statique lorsque l'état du système sur lequel porte le changement est passif pendant que le changement a lieu. Par exemple, un changement dans la structure de la table des symboles d'un compilateur.

Définition 5 :

A l'opposé un changement "dynamique" porte sur un système en cours d'exécution (le changement en ligne). Exemple, le changement du modèle d'un procédé en cours d'exécution.

Pour prendre en compte cette notion du changement dynamique, on doit normalement introduire la contrainte du temps. Pour donner un aperçu de ce qu'est un

changement en ligne, nous allons définir une proposition sans donner aucune preuve formelle.

Proposition

soit $\Sigma_{pe}(t)$ l'ensemble de fonctions des Σ placées dans la pile d'exécution de Σ à l'instant t. La pile d'exécution de Σ à l'instant t contient l'ensemble de fonctions qui sont prêtes mais c'est seulement une fonction parmi cet ensemble qui peut être exécutée à cet instant.

Le changement en ligne δ sur Σ à l'instant t est *valide* si

1. δ est un changement *complet* pour Σ.
2. $\forall\ (F_i, F_i') \in \delta, F_i \notin \Sigma_{pe}(t)$.

La première partie de la proposition est claire à la lumière de la définition précédemment donnée. La deuxième garantie que la fonction en cours d'exécution à l'instant t doit rester inchangée à travers les versions. C'est-à-dire, s'il y a trois fonctions f_1, f_2 et f_3 qui sont actives dans la pile d'exécution à l'instant t et qu'à cet instant même c'est f_1 qui s'exécute. Dans ce cas, on ne peut pas modifier f_1 mais on peut changer f_2 et f_3. En effet, le changement en ligne n'est pas *totalement dynamique* mais plutôt *semi-dynamique*.

En conclusion, on voit que la portée d'un changement sur un objet quelconque de logiciel, revient à :

- l'identification de changement que l'on peut opérer sur chaque objet logiciel à un niveau d'abstraction donné,
- l'identification des "victimes" du changement (c'est dire, les objets qui sont concernés par un changement, de manière locale dans le logiciel ou à tous les niveaux du logiciel), et
- l'acquisition des connaissances lorsque l'on applique un changement.

2.2.2 Typologie des changements

Il y a deux façon de caractériser les types de changements : classiques et techniques. Dans cette section nous présentons les aspects classiques et nous allons présenter les aspects techniques dans la section 2.3.

Les activités de changement peuvent être décomposées en quatre sous-catégories principales : la maintenance corrective, la maintenance adaptative, la maintenance perfective, et la maintenance préventive [Freedman 82, Lehman 85, Loyall 93].

(a) La maintenance corrective

Pour contrôler le fonctionnement quotidien d'un système logiciel, les ingénieurs de maintenance réagissent à la détection d'erreurs dans le système. L'acte de s'occuper

de ces problèmes est appelé "la maintenance corrective". Donc, le changement correctif consiste à corriger les erreurs quotidiennes. Les changements correctifs d'une durée longue peuvent être entrepris pour corriger les problèmes plus généraux dans la conception ou dans le code.

(b) La maintenance adaptative

Un logiciel est toujours une partie de l'environnement d'un système et sa relation avec cet environnement est fondamentale, par exemple, pour son aptitude à faire sa part du travail réalisé dans ce système.

La maintenance adaptative consiste à adapter le logiciel aux changements de son environnement. Par exemple, considérons un logiciel de gestion de salaire pour une entreprise. Le système logiciel dépend de la loi de taxation de l'état, les règles de taxation communes dans laquelle l'entreprise se trouve. Si la loi de taxes change ou si l'entreprise change d'emplacement alors le système doit être modifié pour prendre en compte la nouvelle modification de la loi.

Un système d'exploitation peut aussi avoir besoin de changement adaptatif si on doit effectuer des changements pour qu'il prenne en compte par exemple un nouveau système de stockage.

De la même façon, supposons qu'un compilateur soit amélioré en ajoutant un débogueur. Les fonctions d'interface doivent être aussi modifiées pour que les utilisateurs du compilateur puissent choisir le débogueur comme une option pendant la compilation. Ce type de changement est aussi un changement adaptatif.

(c) La maintenance perfective

La maintenance perfective consiste à améliorer le logiciel en répondant aux modifications définies par le client et le programmeur. Par exemple, le système fonctionne bien, mais les ingénieurs ont décidé de refaire les modules d'accès aux données pour rendre le système plus efficace, ou de refaire l'algorithme de tri pour améliorer le temps de réponse.

(d) La maintenance préventive

Un changement préventif est effectué sur un système pour prévenir (ou empêcher) un mauvais fonctionnement du système. Par exemple, l'addition d'une fonction qui permet la vérification du type de donnée d'entrée d'un système. Cette vérification évite par exemple l'entrée d'une donnée dans une base de données qui n'est pas conforme à la spécification de son type. Le mainteneur consacre 9% de son temps à ce type de changement (figure 2.2) [Laski 92].

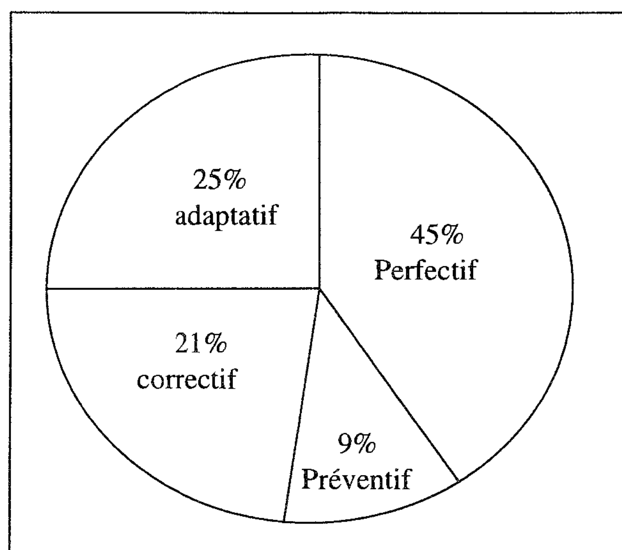


FIG. 2.2 - *La distribution de l'effort de changement [Parikh 86, Laski 92]*

2.3 Evolution de logiciel

L'évolution de logiciel est maintenant considérée comme un concept très important. Le phénomène est apparu au début des années soixante à cause de la croissance continue du logiciel [Lehman 85]. Pour un logiciel donné, cette croissance est basée sur la taille du logiciel à savoir le nombre de modules, les lignes du code ou l'espace disque occupée par le logiciel. Maintenant, on doit accepter que la notion d'évolution ne soit pas seulement due à la taille du logiciel, aux erreurs dans le logiciel, ou a quelques lacunes au niveau du procédé de développement du logiciel, mais qu'elle est intrinsèque à la nature même de l'utilisation de l'ordinateur.

L'évolution d'un système logiciel peut être provoquée par :

- Les changements qui interviennent au niveau des connaissances qui ont été utilisées pour spécifier ou implanter le logiciel;
- Les changements au niveau des spécifications fonctionnelles et de la conception du système;
- Les changements qui interviennent au niveau de la spécification de la performance du système (c.a.d. dans la spécification des ressources disponibles telles que le stockage, le temps de réponse, le débit, etc.);
- Les changements au niveau de l'environnement du système (la spécification des ressources mises à la disposition du logiciel : l'interface entre le système et d'autres logiciels, les périphériques d'entrées/sorties);
- Les changements historiques de l'implantation du système logiciel (le langage de programmation, le système d'exploitation, le support fourni par la base de donnée, ...);

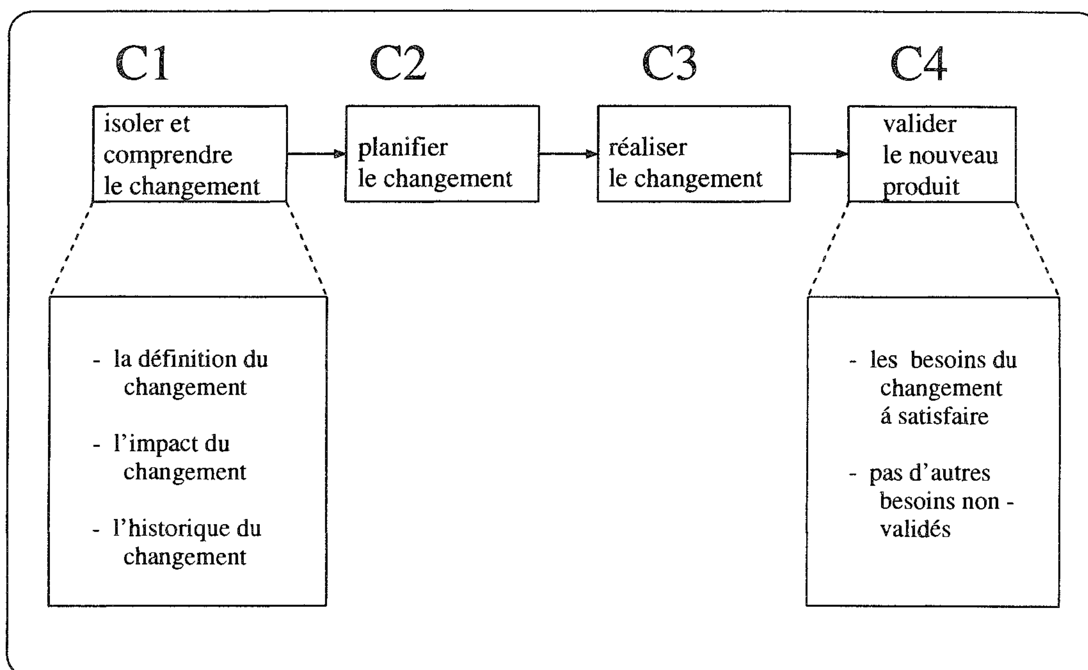


FIG. 2.3 - *Le procédé de changement*

- Les disparités entre la spécification du système logiciel et son implantation;
- Le changement au niveau des besoins stratégiques. Le désir d'améliorer l'efficacité du système.

Indépendamment du type de changement qui a provoqué l'évolution d'un système logiciel, le procédé d'évolution lui même consiste en (cf figure 2.3):

(C1) La compréhension du changement:
- Quelle est la *définition* du changement?
- Quel est *l'impact* du changement sur le système logiciel?
- La prise en compte de l'historique du changement: quelles sont les décisions qui ont été antérieurement entreprises concernant le changement et quelles sont les étapes où les décisions qui ont été prises pour des changements antérieurs?

(C2) La planification du changement:
- Dans quel *ordre* doit-on exécuter ces changements?
- De quelle façon doit-on exécuter ces changements?
- Par quels agents?

(C3) La réalisation du changement:
- Les changements doivent être réalisés par les agents conformément à leur spécification en utilisant les outils adaptés.

(C4) La validation du produit logiciel modifié:
- Assurer que tous les besoins du changement ont été satisfaits.

- Assurer qu'il n'y a pas d'autres besoins (autres que les besoins du changement) non-validés.

2.4 Les types de systèmes logiciels

Pourquoi certains systèmes logiciels ont-ils des tendances à changer plus que d'autres? En général, on peut décrire un système logiciel en relation avec l'environnement dans lequel il fonctionne. Par rapport à des problèmes abstraits, le monde réel contient un certains nombre de concepts ou d'incertitudes qu'on ne comprend pas ou qu'on n'arrive pas à définir complètement. Donc, plus un système logiciel est dépendant du monde réel plus il est prédisposé au changement.

Lehman [Lehman 85] a déjà défini une classification des logiciels en fonction des façons dont ils peuvent changer. Dans la suite, nous allons utiliser cette classification pour montrer pourquoi certains logiciels ont des tendances à changer plus que d'autres.

Nous proposons les diagrammes de la figure 2.4 pour représenter les différents systèmes de Lehman. Ces systèmes ne sont pas récents mais ils continuent à faire autorité dans le domaine de la maintenance logicielle. Le figure 2.4 comporte trois diagrammes : le cycle de vie de S-systèmes, le cycle de vie de P-systèmes, et le cycle de vie de E-systèmes.

2.4.1 Les S-systèmes de Lehman

Il y a des systèmes qui sont définis formellement et dérivables à partir d'une "spécification". Dans ce type de systèmes, un problème donné est complètement spécifié en terme d'ensemble de circonstances sur lesquelles il est appliqué. Par exemple, on peut construire un système qui va effectuer l'addition, la multiplication, et l'inversion matricielle en donnant l'ensemble de matrices et en précisant les "contraintes exactes" de l'application. Ce problème est complètement défini et il y a une ou plusieurs solutions correctes au problème donné. Donc, "la solution" est bien connue et l'ingénieur de développement n'est plus concerné par l'exactitude de la solution mais par l'exactitude de l'implantation de la solution. Un système construit de cette façon est appelé un S-système.

Ce type de système est statique et il n'accepte pas de changement dans la spécification du problème (figure 2.4). Néanmoins, si le monde réel change, le résultat est complètement nouveau et dans ce cas on doit re-spécifier le problème.

2.4.2 Les P-systèmes de Lehman

Les informaticiens peuvent parfois définir des problèmes abstraits en utilisant les S-systèmes et développer les systèmes logiciels pour résoudre ces problèmes. Mais il n'est pas toujours facile ou possible de décrire complètement le monde réel. Dans certains cas, il peut y avoir des solutions théoriques mais impraticables voire impossibles.

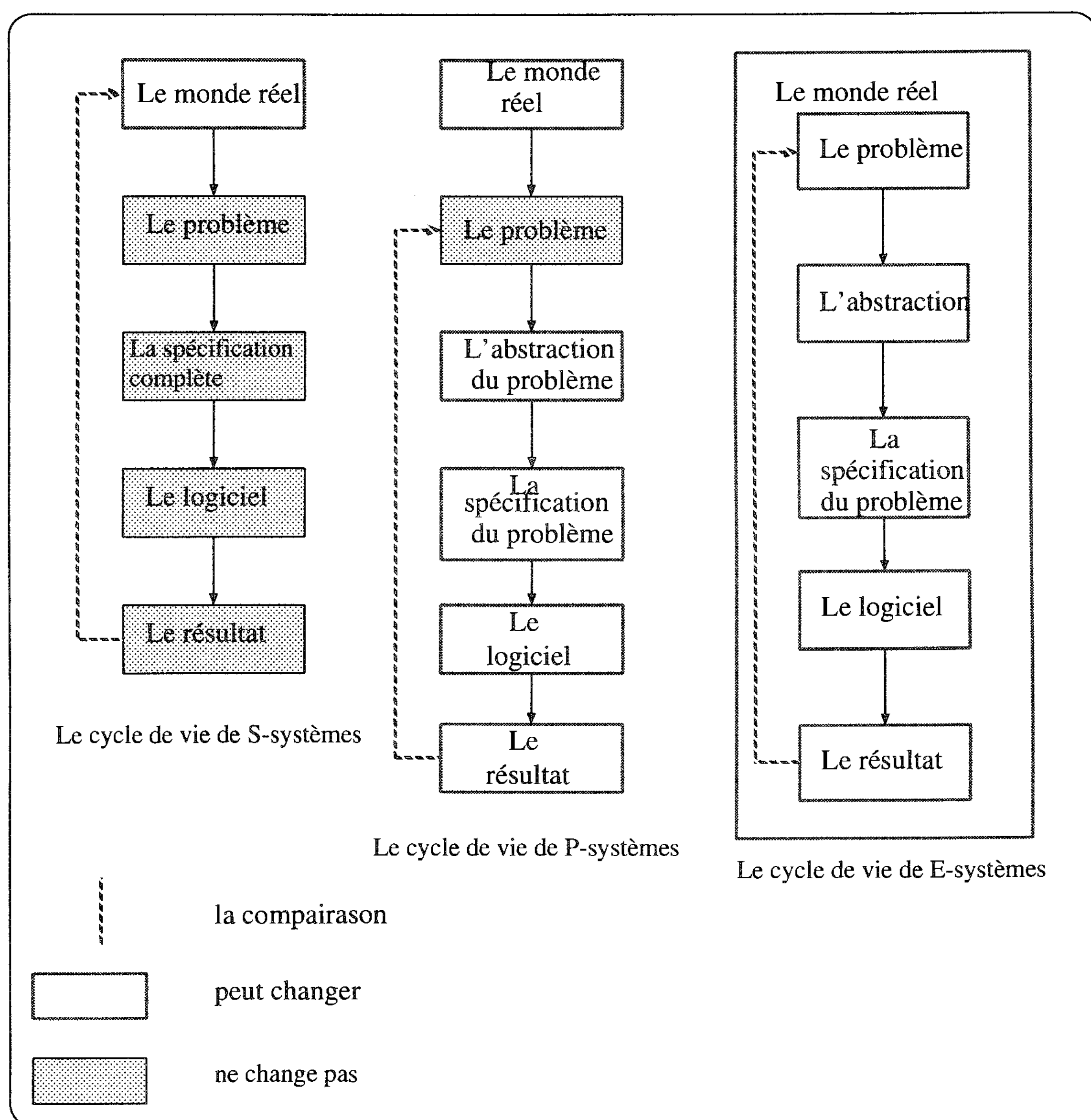


FIG. 2.4 - *Les cycles de vie de S-systèmes, P-systèmes et E-systèmes*

Par exemple, considérons un système pour le jeu d'échecs. Parce que les règles d'échecs sont complètement définies, le problème peut être spécifié complètement. A chaque étape du jeu, une solution peut entraîner le calcul de tous les mouvements possibles et leurs conséquences pour déterminer le mouvement le meilleur. Mais l'implantation d'une solution est infaisable en utilisant la technologie actuelle. Le nombre de mouvements est très élevé pour que l'évaluation pratique soit faite en temps utile. Donc, on doit développer une solution approximative qui est plus pratique à construire et à utiliser.

Pour développer "la solution" à ce problème il faut décrire des stratégies de résolution acceptables d'une façon abstraite, et à partir de cette abstraction, on peut réaliser la spécification pour le système logiciel (cf. figure 2.4). Un système logiciel qui est développé de cette façon est appelé P-système parce qu'il est basé sur une abstraction pratique du problème au lieu de la spécification complète du problème. Un P-système est plus dynamique qu'un S-système. Le résultat est comparé au problème. Si le résultat n'est pas satisfaisant, l'abstraction du problème peut être changée et le besoin modifié pour avoir un résultat plus réaliste.

Dans un P-système, les besoins sont basés sur une abstraction du problème. La solution dépend de l'interprétation de l'ingénieur qui s'occupe de l'analyse des besoins. Même si la solution exacte existe, la solution produite par un P-système dépend de l'environnement dans lequel il a été développé. Dans un S-système, la solution est acceptable si la spécification est correcte. Mais, dans un P-système, la solution est acceptable si le résultat est en accord avec le monde réel (cf. figure 2.4).

Plusieurs éléments peuvent changer dans un P-système. Quand le résultat est comparé avec le problème actuel, l'abstraction peut changer ou les besoins peuvent être modifiés, et l'implantation peut aussi être modifiée. Contrairement au S-système, le système issu d'un changement dans un P-système ne peut pas être considéré comme une nouvelle solution au nouveau problème mais, il est une modification de l'ancienne solution pour le problème actuel.

2.4.3 Les E-systèmes de Lehman

En considérant le S-système et le P-système, la situation du monde réel est supposée stable. Cependant, une troisième classe de systèmes incorporent un changement de la nature du monde réel. Un E-système est celui qui est complètement intégré dans le monde réel et qui change chaque fois que le monde réel change. La solution est basée sur un modèle des procédés abstraits. Donc, le système est une partie intégrale du modèle du monde réel.

Par exemple, un système qui prévoit la santé économique d'un pays est basé sur les fonctionnalités du modèle de l'économie. Or, notre appréciation sur l'économie est basée sur la connaissance du monde réel. Donc, chaque fois que la situation économique change, le système doit refléter le changement.

La figure 2.4 illustre la variabilité d'un E-système et sa dépendance sur le contexte du monde-réel. Tandis que le S-système ne change pratiquement pas et le P-système peut changer d'une façon coordonnée, le E-système peut changer continuellement. De

Etape de développement	L'effet des changements
L'analyse de besoins	- La spécification de besoins
La conception du système	- La spécification conceptuelle de la conception - La spécification de la conception technique
La conception du logiciel	- La spécification de la conception du logiciel
L'implantation du logiciel	- Le code du programme - La documentation du programme
Les tests unitaires et les tests d'intégration	- Le plan de test
L'installation du système	- Les documents tels que * document de utilisateur * document d'opération * quide de programmation * documents de stage.

FIG. 2.5 - *L'effet de changement pendant le développement de système logiciel*

plus, le succès de E-système dépend essentiellement de l'évaluation de la performance du système par l'utilisateur. Etant donné que le problème adressé par un E-système ne peut pas être spécifié complètement, le système doit être jugé par son comportement sur la condition d'opération actuelle.

2.4.4 Les changements pendant le cycle de vie de logiciel

En examinant un système logiciel à la lumière de sa catégorie nous voyons les différentes parties du système qui peuvent être changées pendant le développement et comment cette modification peut affecter le système. Un problème de type S-système (selon sa nature) est complètement défini et ne change pas. Un problème similaire peut être résolu en modifiant le S-système, mais, le résultat est un nouveau problème avec sa solution.

Un P-système est une solution approximative pour un problème et il peut être changé chaque fois que l'on identifie les contradictions et les omissions. En effet, chaque fois que le résultat produit par P-système est comparé et contrasté avec la situation actuelle du problème, le P-système peut être modifié pour avoir un système plus économique et efficace. Pour un P-système, un modèle est une solution approximative pour un problème donné, donc une modification peut avoir lieu dans toutes les étapes de développement.

Le E-système utilise des abstractions et des modèles approximatifs pour décrire une

situation. Donc, un E-système est soumis à des types de changements de P-système. En effet, la nature d'un E-système n'est pas aussi constante parce que le problème peut aussi changer.

L'effet des changements sur un système logiciel est résumé dans la figure 2.5. Par exemple, la modification des besoins pendant l'analyse de besoins peut aboutir à des changements dans la spécification. Une modification au niveau de la conception technique peut entraîner des changements au niveau de la conception détaillée du système et même dans les besoins originaux. Un changement à n'importe quelle étape du développement peut affecter les résultats des étapes antérieures.

2.5 L'analyse de l'impact d'un changement

L'analyse de l'impact est une activité d'identification des parties d'un système qu'on doit modifier pour accomplir un changement [Livadas 92, Loyall 93]. C'est l'identification des conséquences potentielles d'un changement. L'analyse d'impact peut précéder un changement (*analyse d'impact a priori*) ou s'utiliser en conjonction avec le changement. Elle fournit les entrées pour exécuter un changement.

Le terme "analyse d'impact" a une définition variée. Il est bien difficile parfois de situer exactement ce terme sinon à partir du contexte. Il existe aussi des termes avoisinants tels que le mot "traçabilité", le terme "effet de bord", ou le terme "stabilité logique". Or, ces termes ne signifient pas très exactement l'analyse d'impact. Par exemple, la traçabilité est définie comme la capacité de déterminer quelles parties d'un système sont en relations avec d'autres selon une spécification. De la même façon, la stabilité logique d'un système est une résistance du système aux effets de bords d'un changement [Yau 88b].

L'analyse de l'impact d'un changement peut être guidée exclusivement par la connaissance du formalisme du domaine [Williams 88]. Par exemple, si la spécification d'un objet ou un composant logiciel change, la conception des objets associés à l'objet modifié via la "relation de l'implantation" doit être révisée pour prendre en compte l'impact du changement. Il peut être guidé aussi par la connaissance des fonctionnalités d'une application spécifique. Typiquement, les règles d'analyse de l'impact d'un changement peuvent expliciter les propriétés des attributs particuliers ou être "inventées" à partir de la connaissance des valeurs des attributs ou les types de changements associés aux valeurs des attributs.

L'analyse d'impact peut avoir des portées différentes. On peut avoir une fonction d'analyse d'impact globale pour organiser le plan de l'évolution d'un logiciel ou une fonction d'analyse d'impact locale pour prendre en compte les modifications individuelles au niveau des objets du sous-système quand il s'agit d'une application particulière ou au niveau d'une phase de cycle de vie d'un système logiciel.

La fonction d'analyse d'impact peut utiliser la notion de "conductivité" dans les relations de dépendances entre les objets [Dankel II 89, Harandi 90]. Il y a des relations de dépendances qui sont de meilleurs "conducteurs" de changements que d'autres. Ces types de relations permettent à l'analyseur de déterminer les parties du système logiciel

qui "doivent" changer et les parties qui "peuvent" changer. Après quoi, l'ingénieur de maintenance peut décider et organiser les changements. Nous allons essayer d'expliciter la notion d'analyse d'impact à l'aide d'un "scénario".

Scénario : Evaluer le "coût" de changement du code départemental dans une application :

1. Une entreprise de service logiciel est en train de débattre le changement du code départemental de cinq chiffres à un chiffre plus long. L'ingénieur de maintenance est chargé d'évaluer le coût du changement.

2. L'ingénieur sait que les codes départementaux se trouvent d'une façon permanente dans deux fichiers : fournisseurs et fichier principal de clients. Il va donc utiliser un outil d'analyse d'impact pour analyser *a priori* l'impact du changement et essayer d'évaluer le "coût".

3. Résultat de l'analyse : il y a 26 variables dans 6 programmes différents affectés par le changement. Parmi ces 26 variables :

   - 5 sont triviaux (les sous-champs d'adresse, etc.),
   - 8 sont des rapports ou les formats d'écran, et
   - 13 sont issues des dépendances indirectes : le flot de données, le flot de contrôle, etc.

4. Conclusion : à partir de ce résultat,

   - l'ingénieur peut estimer le coût du changement : la main d'œuvre, les matériels, etc. : estimation qui lui servira à décider :
     (a) soit de faire le changement tel quel,
     (b) soit de refaire sa spécification, ou
     (c) soit d'en abandonner l'idée.

2.5.1 Un "cadre générique" d'analyse d'impact

Il y a bien des activités qui sont classifiées comme des activités d'analyse d'impact, pourtant il est difficile d'établir un rapport entre elles. Les approches d'analyse d'impact doivent être caractérisées pour qu'elles puissent être comprises, comparées, et évaluées. Nous allons dans cette section présenter un cadre générique d'analyse d'impact qui peut être utilisé comme un guide pour comprendre une approche d'analyse d'impact, pour comparer ou évaluer deux approches ou pour analyser la structure des approches d'analyse d'impact. Le résultat de cette démarche peut être une critique ou un positionnement ou une évaluation de l'approche (figure 2.6). Ce cadre générique nous permettra d'avoir une vue globale et claire sur le terme *"analyse d'impact"* dans le contexte de notre travail.

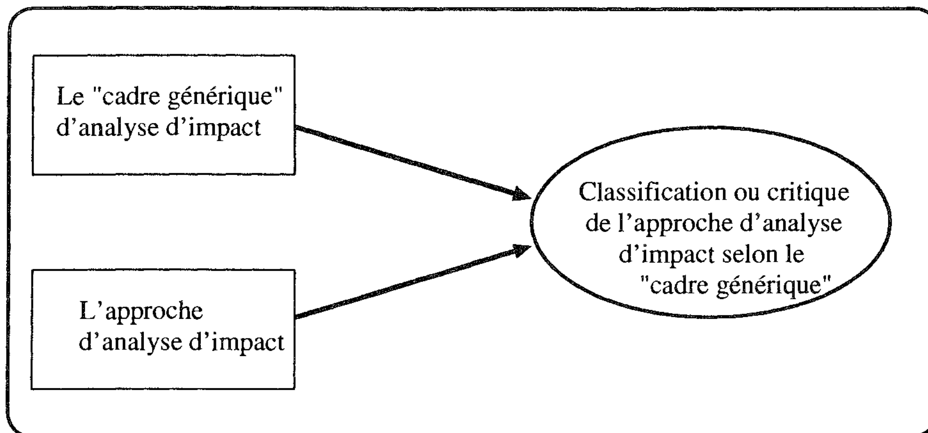


FIG. 2.6 - *L'utilisation du "cadre générique" d'analyse d'impact*

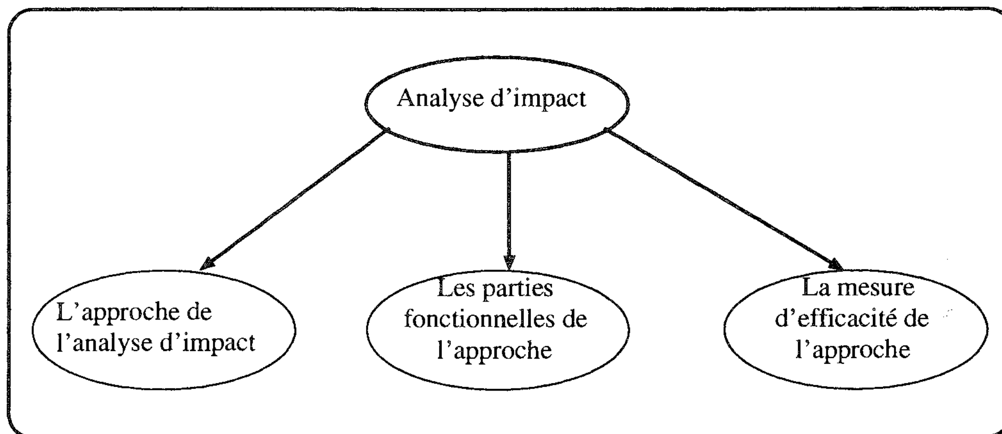


FIG. 2.7 - *Les différente parties du "cadre générique"*

Il comprend trois parties (figure 2.7), une approche d'analyse d'impact, la structure de cette approche, et la mesure de l'efficacité de l'approche. La première partie du cadre (section 2.5.2) examine comment une approche est utilisée pour accomplir une analyse d'impact. Elle examine les caractéristiques offertes par l'interface d'une approche d'analyse d'impact. La structure (les parties fonctionnelles) (section 2.5.3) examine la nature des parties internes et les méthodes utilisées pour exécuter l'analyse d'impact. La mesure de l'efficacité (section 2.5.4) examine les propriétés issues de la recherche des impacts, en particulier comment le but d'analyse d'impact a été atteint.

2.5.2 Une approche d'analyse d'impact

Cette partie du cadre générique examine comment une approche peut être utilisée pour accomplir l'analyse d'impact. Pour analyser l'impact d'un changement, il doit y avoir un changement proposé, quelque chose à modifier, et une façon d'estimer ce qu'on doit faire pour exécuter le changement.

La figure 2.8 donne une vue générique du procédé d'analyse de l'impact d'un chan-

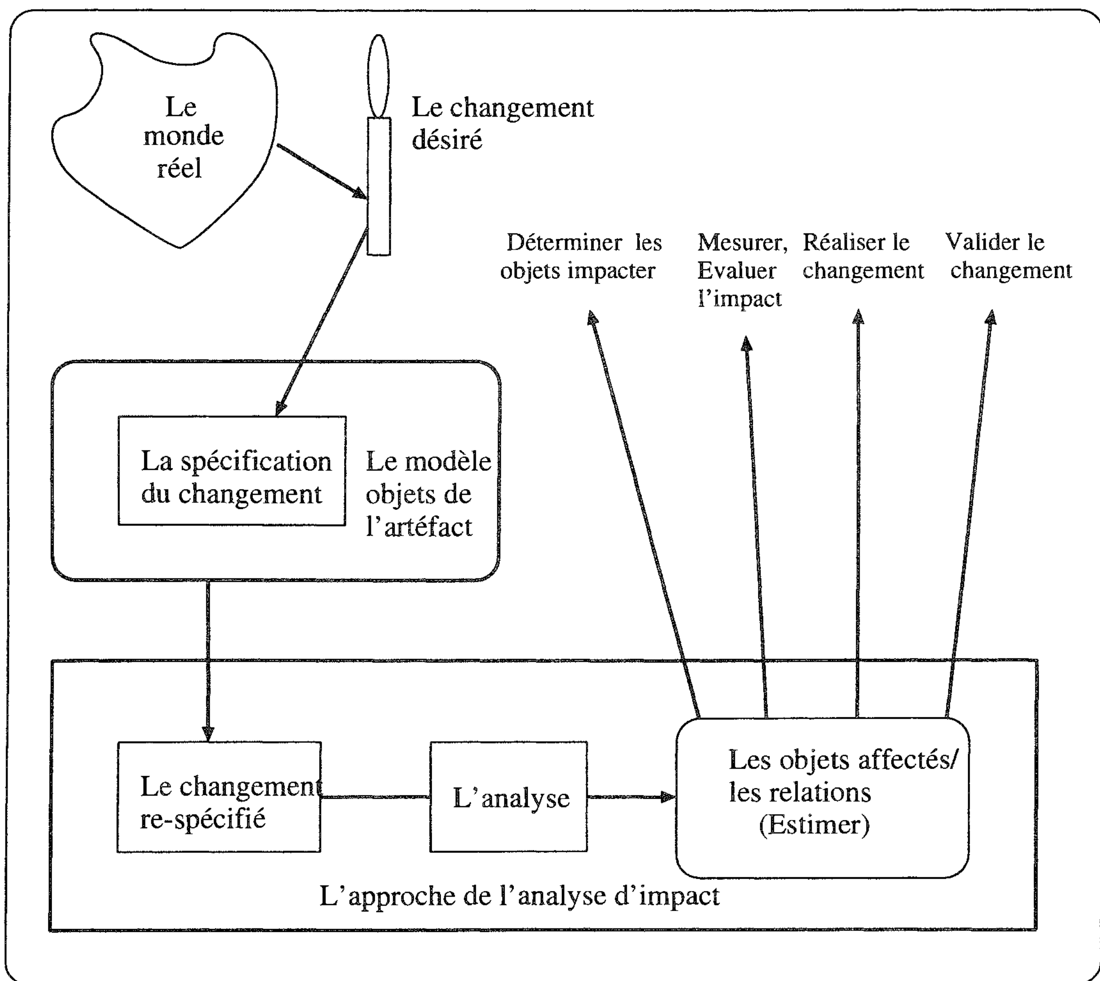


FIG. 2.8 - *L'approche de l'analyse d'impact: Point de vue utilisateur*

gement. Un changement est conçu dans le monde réel. Le changement est spécifié. La spécification du changement utilise les objets et les relations en rapport avec le changement. Les objets et les relations sont issus de l'artéfact. La spécification du changement et la connaissance d'éléments à modifier sont utilisées pour spécifier les objets qui peuvent être initialement impactés. L'approche d'analyse d'impact peut ensuite déterminer les autres objets qui peuvent être concernés par le changement.

L'approche d'analyse d'impact peut aussi donner les traits tels que :

- pourquoi estime-t-on que cet objet est impacté et comment,
- a-t-on des métriques,
- l'animation illustrant l'effet de bord d'un changement,
- l'historique du changement,
- les stratégies de changement possibles,
- des suggestions pour la validation du changement.

2.5.3 Les parties fonctionnelles d'une approche d'analyse d'impact

Nous nous intéressons à la structure de l'approche d'analyse d'impact. C'est à dire, quelle sont les fonctionnalités de l'approche, comment exécute-t-elle ces fonctionnalités, et quels sont les rôles des agents ou des outils impliqués dans ces fonctionnalités'.

La figure 2.09 illustre les différentes parties de l'approche d'analyse d'impact. La spécification d'un changement est basée sur le modèle objet et les relations au niveau d'interface de l'approche. Cette spécification est traduite au niveau du modèle objet interne.

Le modèle objet interne définit les objets et les relations de dépendances que l'approche utilisent pour accomplir l'analyse d'impact. Le modèle objet interne est normalement stocké dans une base; la base possède ses propres commandes pour *créer*, *naviguer*, et *modifier* les objets et les relations. La base est chargée par la décomposition des données en objets et relations en accord avec le modèle objet interne. Cette base peut être créée d'une façon semi-automatique. C'est-à-dire, à partir d'un code source par exemple, on peut extraire les informations utiles sur les objets et les relations en utilisant un analyseur de code (c'est le cas du prototype WHAT-IF, chapitre 5). Ensuite, ces informations peuvent être complétées manuellement en accord avec le modèle objet interne.

Le modèle d'impact définit les règles ou les hypothèses qui reflètent les sémantiques des effets d'un changement. Il définit aussi les classes d'objets et les relations qui sont utilisées par l'approche pour déterminer quand un changement d'un objet peut affecter un autre objet.

La fonction traçabilité/impact implante(définit) le modèle d'impact. Cette fonction définit comment les objets et les relations sont représentés, comment les règles d'impact sont programmées, et comment les algorithmes de recherche sont utilisés pour rechercher les objets et les relations qui sont impactés.

En conclusion, chaque partie de la structure de l'approche d'analyse d'impact peut varier. La table 2.1 donne un résumé de la variation des différents éléments. Dans cette table, la colonne sur la variation donne les différentes variations qui peuvent exister dans chaque partie de la structure. Par exemple, pour la base, on peut avoir un système de fichiers, une base de données relationnelles, ou une base de données objets

La variation les différentes parties de la structure de l'approche d'analyse d'impact		
Les éléments	*L'explication*	*Les variations*
L'interface du modèle objet	Quels objets et relations sont spécifiés pour l'interface du modèle objet?	- chaîne de caractère; - objets de programme; - objets issus de document; - objets utilisateurs spécifiables; etc.
Le modèle objet interne	Quels objets et relations sont utilisés par l'approche pour accomplir l'analyse d'impact?	- objets orientés document; - structure à base d'objets; - graphe; etc.
Le modèle d'impact	Comment modélise-t-il les dépendances?	- flot de données; - flot de contrôle; - chaînes de comparaisons; - graphe de dépendances.
Traçabilité/ impact	Comment l'approche exécute-t-elle l'analyse d'impact?	- décomposition; - structure de comparaison; - recherche heuristique; - recherche stochastique; - méthode non-explicite.
La base	Quel type de base	- relationnelle;- système de fichiers; - objets.

table 2.1

2.5.4 La mesure d'efficacité d'une approche d'analyse d'impact

Pour présenter effectivement la mesure d'efficacité de l'approche, nous allons introduire un certain nombre de concepts en proposant une vue synthétique de cette approche dans la figure 2.10.

Au niveau du modèle objet de l'artéfact, l'analyse d'impact est présumée prendre effet sur un ensemble d'objets appelés *système*. Ces objets sont issus du modèle entourant appelé *univers* .

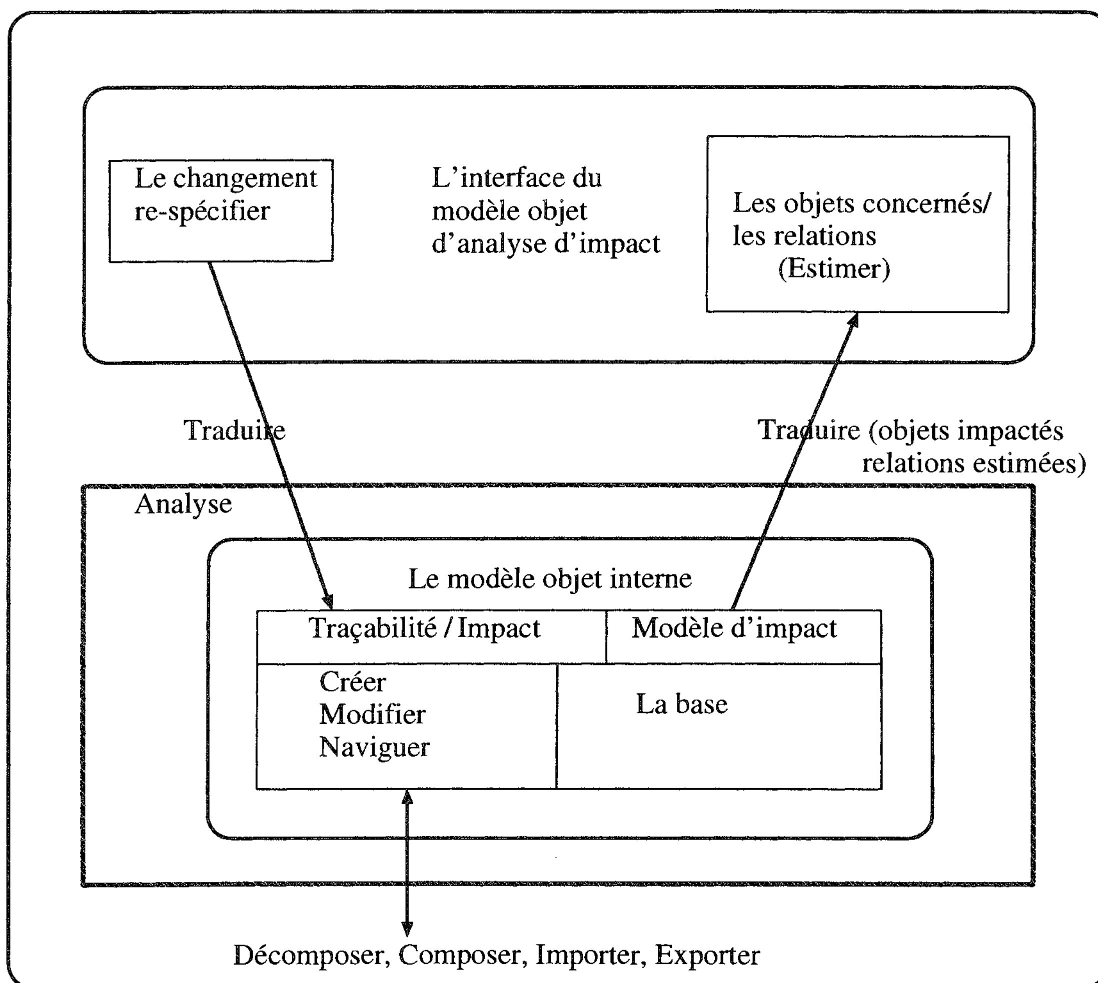


FIG. 2.9 - *Les parties fonctionnelles de l'approche de l'analyse d'impact*

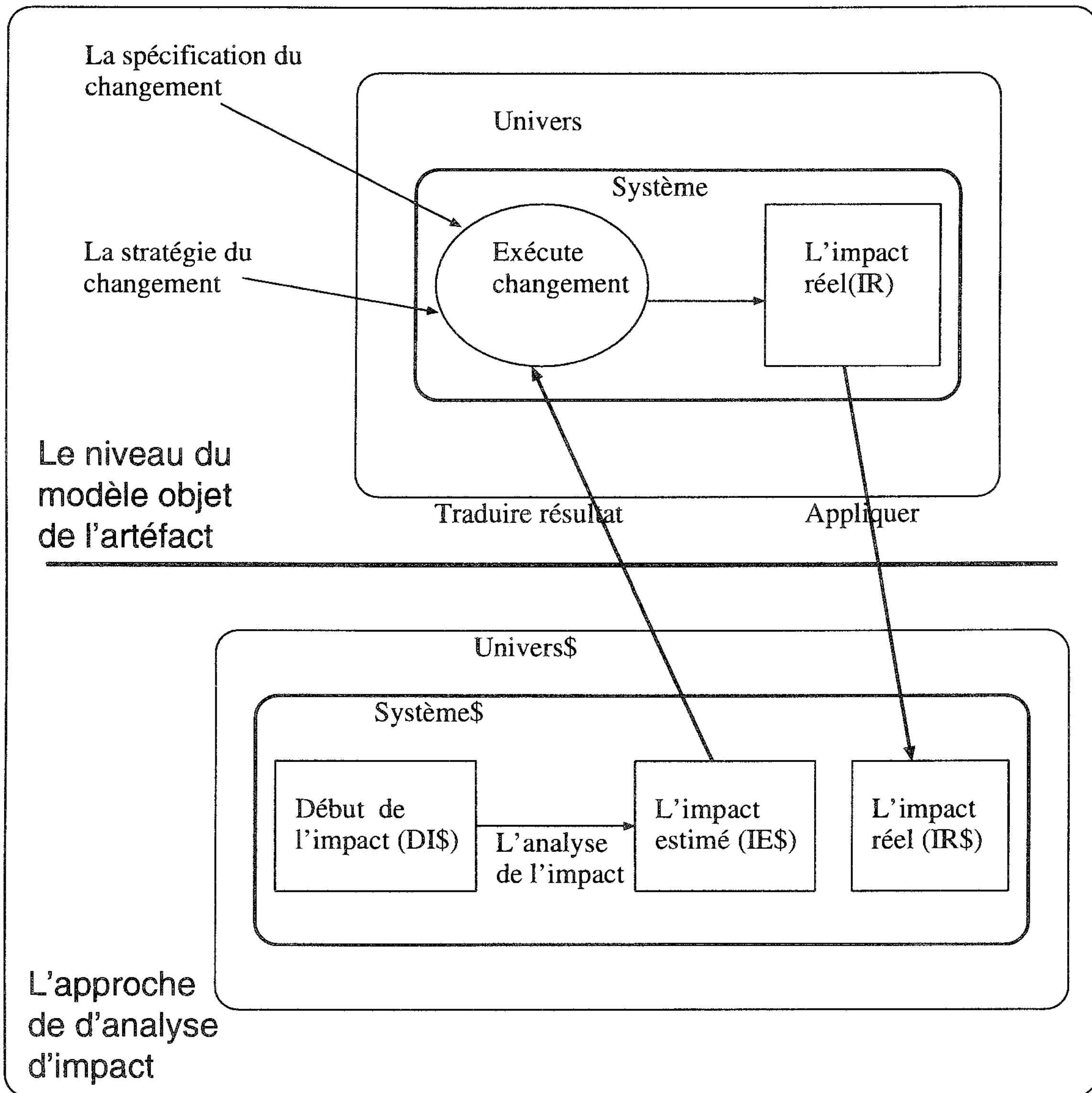


FIG. 2.10 - *La mesure d'efficacité de l'ensemble des éléments d'analyse d'impact*

Au niveau de l'interface du modèle objet, les images de *système* et *univers* sont *système$* et *univers$*. Le "début de l'impact (DI$)" est défini comme l'ensemble des objets qui sont initialement présumés affectés par un changement. "L'impact estimé (IE$)" est défini comme l'ensemble des objets estimés affectés par l'analyse d'impact. DI$ et IE$ sont supposés partager le même modèle objet. Les "chemins d'impact" sont les chemins qui lient les objets de DI$ aux objets de IE$.

"L'impact réel (IR$)" est défini comme l'ensemble d'objets (dans le modèle objet de l'artéfact) qu'on doit actuellement modifier pour accomplir un changement. IR$ est une image de IR (l'impact réel) en termes d'objets et des relations dans l'interface du modèle objet (cf. figure 2.10).

L'impact réel n'est pas en général unique, car un changement peut être implanté de plusieurs façons. Néanmoins, nous allons employer ce terme pour décrire "un" impact réel issu d'une analyse d'impact particulière.

Il est aussi possible de caractériser le début de l'impact (DI), l'impact estimé (IE), et l'impact réel (IR) en fonction du modèle objet de l'artéfact (cf. figure 2.10). Mais, pour simplifier la discussion de cette partie du cadre générique, nous allons les présenter(section 2.5.5) uniquement au niveau de l'interface du modèle objet (cf. figure 2.10).

2.5.5 Les concepts de la mesure d'efficacité

Nous présentons trois concepts pour mesurer l'efficacité d'une approche d'analyse d'impact.

(a) La relation entre le début de l'impact(DI$) et l'impact estimé(IE$)

Par définition, l'impact estimé IE$ contient le début d'impact DI$. Néanmoins, la taille relative de IE$ influence le travail effectué pour trouver les objets estimés être affectés par l'analyse d'impact.

En considérant la figure 2.11 (1, 2, et 3), on voit que pour (1) les tailles relatives de IE$ et DI$ sont égaux; pour (2) la taille de IE$ est un peu plus grande que la taille de DI$; pour (3) la taille relative de IE$ est beaucoup plus grande que la taille de DI$. Pour avoir une meilleure efficacité, il faut que la taille relative de IE$ soit aussi proche que possible de la taille de DI$. Donc, la situation idéale est (1) mais, dans la pratique on trouve normalement la situation de type (2) en grande nombre et de type (3) parfois.

(b) La relation entre l'impact estimé(IE$) et L'impact réel(IR$)

La relation entre IE$ et IR$ est aussi très importante. On veut que IR$ soit aussi (en taille) proche que possible à IE$. On préfère même que IR$ et IE$ soit exactement les mêmes. Ce qui donne un certain degré de confiance à IE$ estimé par l'approche d'analyse d'impact. On ne veut pas que IR$ soit plus grand que IE$. Le fait que IR$ soit plus grand que IE$ signifie que IE$ n'indique pas la vraie portée d'un changement.

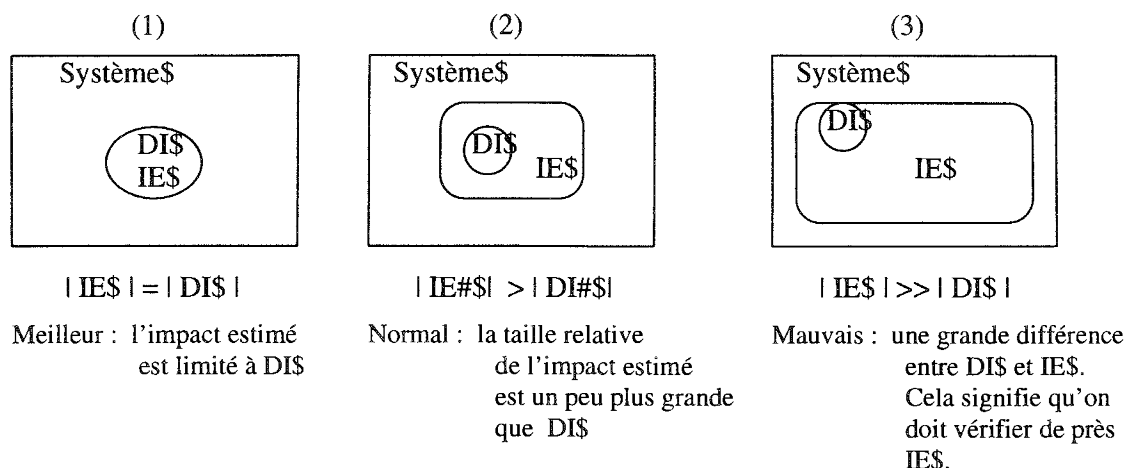


FIG. 2.11 - *La relation entre le début de l'impact(DI$) et l'impact estimé(IE$)*

Si l'impact réel (IR$) et l'impact estimé (IE$) sont exactement les mêmes alors, une telle approche d'analyse d'impact est considérée meilleur.

Si IE$ est plus grand que IR$ et que IE$ contient IR$ (i.e. IR$ $\subset$ IE$) ou si IR$ est plus grand que IE$ et que IR$ contient IE$ (i.e. IE$ $\subset$ IR$) alors, cette approche d'analyse d'impact est considérée comme juste.

Si au contraire IR$ est beaucoup plus grand que IE$ et IE$ $\subset$ IR$ ou si IR$ $\neq$ IE$ et | IR$ $\cap$ IE$ | $= 0$ alors, une telle approche est considérée comme mauvaise.

(c) La relation entre l'impact estimé(IE$) et système$

Cette hypothèse examine la relation entre IE$ et système$. En général, on ne veut pas que l'approche d'analyse d'impact estime que tout est impacté, c'est à dire, IE$ est la même que système$ (figure 2.12 (1)). En fait, il faut que IE$ respecte un critère de "minimalité" et que ce critère peut être précisé en pourcentage par rapport au système$. Dans le cas ou IE$ est égal à système$ et que l'analyse d'impact est "fiable" cela signifie que la modification demandée suggère malheureusement la reconception du système.

La "distance" entre IE$ et système$ est une façon de mesurer l'approche d'analyse d'impact. On veut que la taille de IE$ soit plus petite par rapport à la taille de système$ (figure 2.13 (2) et (3)).

2.6 Conclusion

Nous avons présenté la problématique générale de l'approche d'analyse d'impact de changement en se basant sur un cadre générique, à savoir, l'application de l'approche d'analyse d'impact, la structuration, et l'efficacité de l'approche. L'idée est de proposer d'une part, un cadre générique qui permet d'évaluer, de comparer, et de mesurer

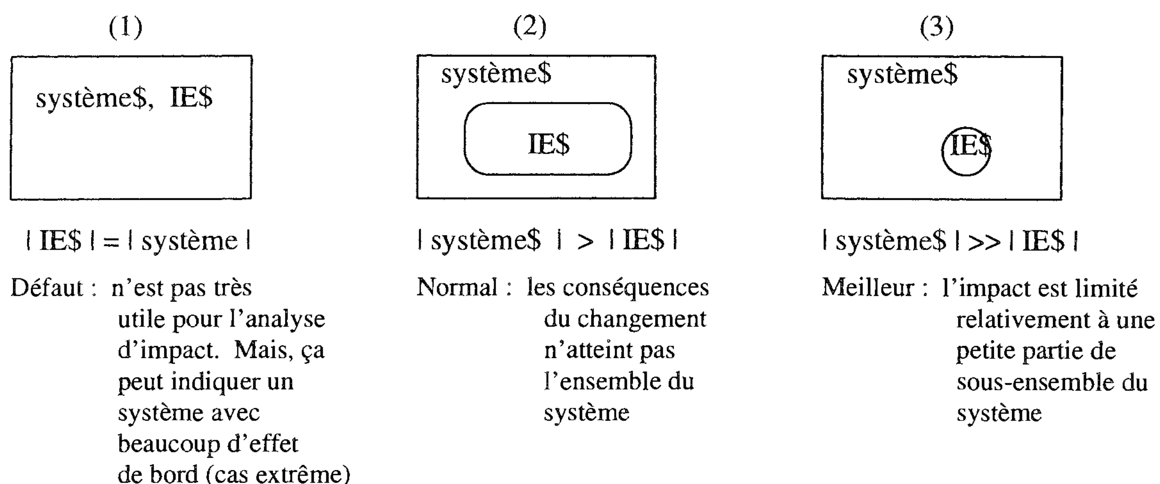


FIG. 2.12 - *La relation entre l'impact estimé(IE$) et système$*

une approche d'analyse d'impact. D'autre part, puisque le terme "analyse d'impact" est utilisé dans bien des domaines, nous avons essayé de définir d'une façon précise l'approche d'analyse d'impact par rapport à notre travail.

Nous allons dans le chapitre 4 nous inspirer d'une part de ce ce cadre générique et d'autre part de l'état de l'art (chapitre 3) pour définir la problématique du modèle WHAT-IF et proposer une solution.

Chapitre 3

État de l'art

3.1 Introduction

Il existe à l'heure actuelle peu de techniques qui permettent de comprendre, et de valider les modifications. Des méthodes de gestions de changements ont été proposées. Ces méthodes comportent les procédures pour l'autorisation d'effectuer les changements, les techniques de validation de code *walkthroughs*, et les procédures de "sign-off" [1] avant que le logiciel soit utilisé après la modification. Mais, ces méthodes n'adressent pas forcément la viabilité du logiciel qui a été modifié.

Donahoo [Donahoo 80], Holbrook [Holbrook 87], Kamkar et al. [Kamkar 93], et Chen et al [Chen 93]. ont proposés un panorama d'outils qu'on peut utiliser pour la maintenance de logiciels. Parmi les outils présentés, il y en a quelques uns qui permettent la vérification de cohérence et le croisement des références des modules du programme afin d'aider l'ingénieur de maintenance dans sa tâche et d'éviter certains types d'erreur. Il y en a aussi qui effectuent la régression en regardant les modules et les instructions de branchements qui sont déjà exécutés.

Dans ce chapitre, nous allons dans un premier temps examiner les méthodes d'analyse du changement qu'on peut utiliser pour comprendre et valider un code modifié. Parallèlement, nous allons présenter les outils existants basés sur certaines de ces méthodes. Ce sont les méthodes dit "algorithmiques" ou les "méthodes par la simulation". Nous allons aussi dans un deuxième temps présenter la technique d'analyse des relations de dépendances.

3.2 Les outils d'analyse statique du code

3.2.1 Introduction

La maintenance dans un premier lieu exige la compréhension du programme, sa fonctionnalité, sa structure interne, et son besoin opérationnel [Fay 85]. Traditionnelle-

1. C'est une technique de certification qui est effectuée systématiquement après toute modification.

ment, les ingénieurs de maintenance ont peu d'outils pour comprendre effectivement un logiciel. Les programmes sont analysés en essayant d'imiter les fonctionnalités d'ordinateur à l'aide de "listings", de "clips" sur papier, de méthodes telles que "highlighters" qui mettent en évidence certaines structures du programme, et la capacité mentale de l'ingénieur de maintenance à comprendre le traitement des données répétitifs.

Nous présentons ci-dessous un certain nombre d'outils logiciels. Ils sont présentés comme des supports pour la compréhension de programme et par conséquent des outils de maintenance [Roman 86, Software 86, Holbrook 87]. Pour chaque catégorie d'outils, nous donnons une description brève des fonctionnalités associées et une table donnant un résumé des exemples.

3.2.2 L'analyseur de code

Ce groupe d'outils analysent la structure de contrôle et le flot de données de programmes d'une façon statique. Dans ce cas, on n'examine pas le comportement dynamique du programme.

Les outils d'analyse de code varient selon les tâches effectuées. Néanmoins, on peut les classifier en deux sous-groupes : les générateurs métriques spécialisés pour le traitement par lots et les outils d'analyse par balayage logique interactive.

Les générateurs métriques spécialisés pour le traitement par lots produisent les mesures standards définies à priori pour évaluer la complexité du programme. Dans certains cas, on peut les utiliser pour déterminer l'applicabilité des techniques de restructurations [Ruven 83, Kamkar 93].

Les outils d'analyse par balayage logique interactive donnent les moyens de navigation de la logique du programme ou le flot de donnée du programme en isolant les classes d'instructions sources spécifiées (e.g. les instructions d'entrées, de conditions ou de structure de contrôle particulier).

Les outils d'analyse de code fournissent d'autres fonctions telles que l'analyse du code non utilisé dans un programme et l'identification des parties du programme mal codé.

La table 3.1 donne des exemples d'outils d'analyse de code.

Nom d'outil	Langage(s) supporté(s)
Basic Program Analyser	Basic
C-Tracer	C
Cobol Structuring Facility	VS Cobol II
F-Scan	Fortran
Fastbol	Cobol
LOGISCOPE	Pascal, C, Fortran, Cobol, Modula2

Table 3.1

3.2.3 Les outils d'aide à la documentation

Ces outils génèrent la documentation graphique qui illustre la logique du programme à différents niveaux d'abstractions. Par exemple, le diagramme de flots (e.g. Flowgen/F II, Flobol) et les hiérarchies d'appels (e.g. Tree Diagrammer).

La table 3.2 donne une liste d'outils qui génèrent la documentation du programme.

Nom d'outil	Langage(s) supporté(s)
ADF (Automatic Documentation Facility)	Cobol, Assembler
ADPL	Pascal, C, Fortran
ADS (Automatic Documentation System)	Cobol, Assembler
DFDP (Diagraphics for Data Processing)	Cobol
Softool (Programming Environment Tools)	Fortran, Cobol, C
Tree Diagrammer	C, Basic, Pascal, DBASE, Fortran, Modula 2

Table 3.2

3.2.4 Les outils de références croisées

Ces outils tracent l'utilisation des éléments de données, les paragraphes nommés, ou les procédures dans un programme. Ils peuvent être utilisés pour isoler les "plans délocalisés" dans lesquels une fonction particulière est dispersée dans les sections du code qui n'ont aucun rapport entre aux [Stan 85].

Les références d'objets sont en général identifiées par les numéros d'instructions programme sources. En outre, on trouve les informations telles que le type d'instruction

concernée (instruction de mouvement, d'affectation, de condition ...), ou une copie de l'instruction elle même.

La table 3.3 donne quelques outils qui effectuent la référence croisée.

Nom d'outil	Langage(s) supporté(s)
Byblos-Source Documentation System	Cobol
Docu/Manager	RPG
ISAS (Integrated Software Analysis)	Fortran, Assembler
MAD/3000	Cobol, Fortran, Basic
Rand Development Center SMU series	Cobol

Table 3.3

3.2.5 Les outils de restructurations

Les outils de restructurations acceptent en entrée un code non structuré et produit un code structuré. L'avantage est qu'un code structuré est plus lisible par rapport à un code non structuré. Il donne aussi un rangement qui permet un aperçu rapide de la structure locale et globale du programme [DeBalbine 75, Holbrook 87]. De plus le code résultant est dans un style cohérent.

La table 3.4 contient quelques exemples d'outils de restructurations.

Nom d'outil	Langage(s) supporté(s)
PM/SS	Cobol
Superstructure	Cobol
Cobol Structuring Facility	VS Cobol
Structured Retrofit	Cobol

Table 3.4

3.3 La technique d'analyse d'effet d'un changement

Bien que la connaissance complète de toutes les conséquences d'un changement soit complexe, il est "théoriquement" possible d'identifier au moins les effets directs d'un changement sur le code. Par exemple, la modification d'une variable X dans une instruction arithmétique peut avoir des effets sur toutes les instructions où la variable

a été utilisée. Une modification dans les instructions IF ou WHILE peut avoir des effets directs sur toutes les instructions dont l'exécution est conditionnée par le IF ou le WHILE. Ces deux exemples peuvent être identifiés comme les effets de "premiers ordres" car ce sont les effets directs issus du changement. Néanmoins, chaque effet d'un changement peut produire des effets dit secondaires ou indirects. Quand la nouvelle valeur de X est utilisée dans l'évaluation d'une variable Y, toutes les instructions dans lesquelles Y figure sont aussi affectées. Donc, on peut parler des effets de "premiers ordres", de "deuxièmes ordres" ou de "n^{ime} ordres".

3.3.1 L'algorithme d'identification d'effet de bord du changement dans le code

L'algorithme pour identifier les instructions affectées par un changement dans un programme est un processus itératif dans lequel les instructions sont identifiées et marquées. Les effets de premier ordre, de deuxième ordre, etc., sont identifiés et marqués. Le processus s'arrête quand il n'y a plus de nouvel élément dans la dernière itération.

Les étapes de l'algorithme sont les suivantes :

1. un *diff-type* outil est utilisé en premier lieu pour marquer la différence entre *l'ancien programme* et *le nouveau programme*;

2. dans chaque version du programme, chaque instruction marquée est analysée pour déterminer les effets de premier ordre. Ensuite les instructions affectées par les effets de premier ordre sont aussi marquées pour l'analyse suivante;

3. l'étape (2) est répétée jusqu'à ce qu'il n'y ait plus d'instruction à marquer ou jusqu'à un certain niveau d'analyse.

Exemple

L'exemple de la table 3.5 montre un changement simple dans un petit programme. Le programme contrôle les transactions sur un compte bancaire. La modification, lignes 7.1 et 7.2, ajoute une troisième type de transaction : un virement en espèce doit être retranché de l'avoir du compte.

```
     L'ancien programme                   Le nouveau programme
     ==================                   ====================

1.    bal := 0;                            bal := 0;
2.    read code, value;                    read code, value;
3.    do while not EOF                     do while not EOF
4.       if (code = 'dep') then               if (code = 'dep') then
5.           bal := bal + value;                  bal := bal + value;
6.       if (code = 'ck') then                if (code = 'ck') then
7.           bal := bal - value;                  bal := bal - value;
7.1                                           if (code = 'wit') then
7.2                                               bal := bal - value;
8.       write bal;                           write bal;
9.       read code, value;                    read code, value;
10.   end do;                              end do;
```

Table 3.5

```
   L'ancien programme                     Le nouveau programme
   ==================                     ====================

   bal := 0;                                 bal := 0;
   read code, value;                         read code, value;
   do while not EOF                          do while not EOF
      if (code = 'dep') then                    if (code = 'dep') then
          bal := bal + value;                       bal := bal + value;
      if (code = 'ck') then                     if (code = 'ck') then
          bal := bal - value;                       bal := bal - value;
                                          **    if (code = 'wit') then
                                          **        bal := bal - value;
      write bal;                          **    write bal;
      read code, value;                         read code, value;
  end do;                                    end do;
```

Table 3.6

Parce que le changement ici est une addition, il faut seulement analyser le nouveau programme. Le changement lui même est dans les instructions 7.1 et 7.2 qui sont

marquées (table 3.6) conformément à l'étape 1 de l'algorithme. La deuxième étape est d'analyser les instructions 7.1 et 7.2 pour trouver les effets de premiers ordres. L'expression 7.1 affecte seulement 7.2 (déjà marqué) et l'effet de 7.2 est sur toutes les instructions qui utilisent "bal" après 7.2. Dans ce cas , l'instruction 8 est marquée (table 3.6). Après le premier cycle d'analyse, il y a trois instructions marquées 7.1, 7.2, et 8.

L'analyse s'arrête là parce que l'instruction "write" n'a aucune effet sur le programme. Donc, il n'existe pas d'effets d'ordres supérieurs. Après, la fonction d'analyse d'effets d'un changement, imprime les deux listings et le programmeur peut voir que les lignes 7.1, 7.2 et 8 sont marquées. Après quoi il peut continuer, assure que les effets de son changement sont bien localisées.

3.3.2 L'applicabilité

Plusieurs petites modifications ou changements dans un programme de taille moyenne (entre 500 et 1500 lignes de code) peuvent être analysés en utilisant cette méthode. Dans certains cas extrêmes comme par exemple une modification d'une fonction qui manipule une pile, les effets de cette modification peuvent se propager à toutes les parties du programme qui utilisent la fonction et dans ce cas, il faut trouver une autre méthode d'analyse. Parce que la méthode d'analyse précédente ne peut plus s'appliquer.

Les programmes non structurés sont difficiles à étudier dans ce cadre. Par exemple, un changement dans une instruction GOTO ou IF peut avoir des effets partout, sauf si on arrive à localiser les effets.

3.3.3 L'utilité

On remarque que cet algorithme d'analyse de l'effet de bord est assez simple puisqu'il ne prend pas en compte la structure du programme ou le fait qu'il existe des effets de changement qui s'annulent. De plus, l'algorithme identifie seulement les variables et les instructions qui peuvent être affectées par un changement sans donner des informations sur la façon dont elles sont affectées. Dans ce cas, le programmeur doit décider si les effets sont désirables ou non. Le gain de cette simplicité de l'algorithme est la rapidité de l'exécution.

Ainsi, une première utilisation de cette méthode peut être de donner un aperçu rapide des effets d'un changement sur un programme. Si cette analyse montre qu'il y a peu d'effets, la suite peut être faite à la main. Sinon, si les effets sont complexes, on doit utiliser une autre méthode d'analyse. Cette méthode peut être utilisée comme un guide pour le test d'un programme modifié. L'ingénieur qui s'occupe du test doit normalement s'assurer que toutes les parties du programme qui ont été modifiées sont bien testées et, en examinant les effets du changement, il peut définir les tests qui mettent en évidence les intersections entre le changement et le reste du programme.

3.3.4 Les outils de comparaison de code

La plupart des outils de comparaison de code sont basées sur la technique d'analyse d'effets d'un changement. Ce groupe d'outils aide les programmeurs à identifier les changements entre deux versions d'un programme.

La table 3.7 donne quelques exemples des outils de comparaison de code.

Nom d'outil	Langage(s) supporté(s)
Comparex	Cobol
ISAS (Integrated Software Analysis)	Fortran, Assembler
S/Compare	C, Cobol, DDS, PL1, RPG
SAGE Maintenance Programming System	Cobol

Table 3.7

3.4 Les méthodes d'analyse dynamique

Les méthodes d'analyse dynamique sont basées sur un programme en état d'exécution à l'opposé d'analyse statique sur le code source. En général, tous les tests effectués sur un programme sont considérés comme une analyse dynamique [Myers 79, Howden 81, Rothermel 93]. Pour notre part, nous allons explorer l'utilisation de ces méthodes d'analyses dynamiques dans un contexte spécifique d'un changement au niveau d'un programme.

3.4.1 Le test de la couverture logique

Dans la littérature, les critères tels que la couverture logique d'instruction d'affectation, la couverture de branchement inconditionnel, etc. sont donnés comme des guides pour sélectionner les jeux de tests [Myers 79][Ntafos 84]. Toutes ces critères sont réellement heuristiques. Mais l'expérience montre que l'utilisation d'un standard bien défini pour les tests aide à diriger le test et augmente la probabilité de trouver les erreurs.

La littérature sur les méthodes d'analyses dynamiques décrivent plusieurs "moniteurs" de la couverture de test. Ces moniteurs aident un programmeur à déterminer si un critère de la couverture logique a été satisfait [Myers 79]. La méthode utilisée est d'introduire les instructions d'appels dans les modules pour qu'on puisse enregistrer le temps dont chaque branchement ou instruction à été exécuté. Une extension intéressante est de combiner la technique de la couverture logique avec la technique d'analyse d'effets (section 3.2). Cette combinaison permet de voir si toutes les parties affectées par un changement sont bien testées. C'est une sorte de contrôle automatique.

3.4.2 L'exécution comparative

L'une des idées intéressantes de l'analyse dynamique est le programme qui "s'auto-valide"[2] [Howden 81]. Par exemple le système PET décrit dans [Stucki 77] permet au programmeur d'introduire des assertions dynamiques sur les valeurs de variables dans le programme et ces assertions sont testées d'une façon automatique pendant l'exécution du programme. Pendant le test, le programme effectivement s'auto-valide par rapport à ces assertions. Ce qui permet de détecter les erreurs et identifier plus rapidement les fautes de bases.

Le problème avec la méthode d'assertion dynamique est que le développement des assertions elles même est une tâche très difficile. Il n'existe pas des relations claires entre plusieurs variables de base et d'autres variables qui permettent la formulation des assertions. Même si ces relations existent, elles doivent être conçues avec soins et intégrées explicitement dans le programme.

En utilisant cette méthode pour l'analyse de changement dans un programme, l'ancienne version du programme peut être utilisée pour fournir une sorte de "support" pour la technique d'auto-validation. La manière la plus simple pour le programmeur est d'introduire des points de synchronisation dans les deux versions du programme (l'ancienne et la nouvelle). Ces points de synchronisation serviront de points de comparaisons pour les deux versions. Pendant l'exécution comparative, le même type de donnée de test est introduit dans les deux programmes. A chaque point de synchronisation, les valeurs des variables qui sont différentes sont imprimées. Donc, le nouveau programme est effectivement validé par rapport à l'ancienne version.

L'exécution comparative peut être combinée avec certains caractéristiques des débogueurs disponibles dans la plupart des nouveaux systèmes temps-réels. Ces outils permettent au programmeur d'accéder à des points clefs dans le programme et sélectionner les structures de données qu'il a voulu examiner. Néanmoins, cela peut prendre beaucoup de temps pour avoir un résultat intéressant et aussi produire un grand volume de données parfois erronées.

3.5 Les méthodes d'évaluation symbolique

3.5.1 Introduction

L'évaluation symbolique est une méthode d'analyse de flot de données dans laquelle les valeurs de "temps d'exécution" de certaines variables (de sorties) du programme sont représentées symboliquement en terme de valeurs de "temps d'exécution" d'autres variables (d'entrées) du programme. Cette méthode contraste avec l'exécution normale d'un programme qui donne le résultat numérique des variables du programme via les instructions de sorties. Considérons le segment d'un programme suivant :

```
read(a,b,c);
```

2. "self-validating" program

```
t := a + b;
y := t**c;
print(y);
```

prog 3.0

La valeur imprimée pour y pendant une exécution normale peut être par exemple "β". Ce résultat numérique dépend des valeurs lues pour les variables a, b, et c. La représentation symbolique de y, en contraste donne une relation explicite entre y et (a, b, et c), c'est a dire, y $= (a+b)^c$.

Trois approches différentes de l'évaluation symbolique ont été développées, à savoir l'exécution symbolique, l'évaluation symbolique globale, et l'évaluation symbolique dynamique [Clarke 81]. Nous allons présenter chaque approche et leur application à l'analyse des effets de changement dans un programme en commençant d'abord par la terminologie et la notation.

3.5.2 La terminologie et la notation

Il est parfois commode de décrire les techniques d'analyse de flots de données en termes de graphe orienté. Les noeuds (ou les sommets) d'un graphe par définition correspondant aux instructions individuelles du programme, ou à la séquence des instructions exécutables du programme avec une seule entrée et une seule sortie. Chaque arc du graphe correspond à un transfert possible de contrôle entre deux noeuds adjacents.

Un "chemin complet de programme" (ou simplement un "chemin de programme") correspond à une séquence de noeuds et aux bords connexes. Cette séquence commence avec le noeud d'entrée du programme et termine avec le noeud de sortie, ce qui représente une séquence possible d'exécution du programme. Un "chemin partiel de programme" peut commencer et peut se terminer avec des noeuds qui ne sont pas des noeuds d'entrées ou de sorties. Considérons le programme ci-dessous et son graphe correspondant :

```
begin prog           0a
read (x, y, z)
if x > y then        0b
   t := x
   x := y                      0c
   y := t
end-if
if y > z then        0d
   t := y
   y := z                      0e
   z := t
```

```
end-if
if x > z then       Of
   t := x
   x := z                 Og
   Z := t
end-if
print (x, y, z)
end prog            Oh
```

prog 3.1

On voit ici qu'il y a huit chemins de programme distincts (Oa, Ob, ..., Oh). On peut avoir jusqu'à huit combinaisons de chemins complets à partir de ce graphe. Il y a (Oa, Ob, Od, Of, et Oh) qui représente un chemin complet dont les trois conditions b, d, et f sont fausses. Il y a aussi (Oa, Ob, Oc, Od, Oe, Of, Og, et Oh) dont les trois conditions c, e, et g sont vraies. On peut donc représenter les chemins avec leur valeurs booléennes. Dans ce cas, le chemin (Oa, Ob, Od, Of, et Oh) est représenté par "FFF" parce que les trois conditions b, c, et f sont fausses et de la même façon, le chemin (Oa, Ob, Oc, Od, Oe, Of, Og, et Oh) par "TTT".

Le "domaine" d'un programme est un ensemble possible de données d'entrées du programme. Un "chemin de domaine" est le sous-ensemble de domaine du programme qui déclenche l'exécution d'un chemin particulier du programme.

3.5.3 L'exécution symbolique

L'exécution symbolique concerne l'analyse d'un chemin d'un programme particulier. L'analyse est décrite à l'aide des expressions algébriques et celles ci nécessitent des variables d'entrées, des constantes et des conditions requises pour l'exécution d'un chemin particulier. La "condition de chemin" est une conjonction des conditions de branchement à travers un chemin donné. Cette condition est exprimée en terme de variables d'entrées du programme.

Pour illustrer cette idée de la condition de chemin, considérons le *prog 3.1*. La condition de chemin pour le chemin "FFF" du programme est simplement $x \leq y \cap y \leq z \cap x > z$. On peut noter ici que cette condition ne peut pas être satisfaite. Ce qui implique qu'il n'existe aucune donnée d'entrée qui peut déclencher l'exécution de ce chemin.

La méthode générale

Déterminer les prédicats propres pour le cas du programme *prog 3.1* est trivial car aucune variable parmi x, y, ou z n'est changé depuis le début du programme jusqu'au noeud Of à travers le chemin en question.

Mais pour six parmi les huit chemins possibles, la tâche est plus complexe, et elle implique l'utilisation de l'exécution symbolique. Considérons le chemin "TTT". La

valeur de y au noeud `Ob` en générale n'est pas la même que sa valeur au noeud `Od`. De la même façon, la valeur de x et de z au noeud `Of` n'est pas non plus même aux noeuds `Ob` et `Od`. En résumé, l'exécution symbolique concerne directement l'utilisation des indices pour indiquer les valeurs de variables utilisées dans les instructions de sorties et dans les conditions en terme de variables d'entrées ou de variables définies par le programme. Par exemple, $x_f = y_a$ pendant que $z_f = x_a$. Ici, les indices a et f indiquent les valeurs de x, y, et z au noeuds a et f.

De l'application à l'analyse d'effets de changement

Tandis que l'exécution symbolique est une technique d'analyse de chemin orienté, une application directe à l'analyse de changement la plus simple et évidente est dans le contexte de changements localisés à un chemin donné. Il y a deux composants d'analyse de changement de programme sur lesquels la technique d'exécution symbolique peuvent être appliqués : l'identification de changement symbolique dans l'expression des variables de sorties et l'identification de changement dans l'expression de domaine du chemin qui est issue des modifications à travers le chemin du programme. La capacité de l'ingénieur d'apercevoir les différences symboliques lors des modifications peut faciliter la détermination de l'impact potentiel .

Pendant que l'exécution symbolique normale produit les expressions en terme de variables d'entrées d'un programme, il est parfois dans l'intérêt de l'ingénieur de maintenance d'examiner les différentes expressions en terme de structures de données ou des variables locales à la région de la modification. La raison pour cette hypothèse est que : l'aptitude à comprendre la nature exacte d'une action est naturellement augmentée quand les effets de cette action peuvent être perçus dans un contexte lié aux causes.

3.5.4 L'évaluation symbolique globale

L'évaluation symbolique globale est une technique pour représenter symboliquement la fonctionnalité d'un programme indépendamment d'un chemin particulier. Le résultat est parfois complexe. Il ressemble à une instruction "case" du langage C dont chaque composant représente un chemin d'une fonctionnalité du programme.

La méthode générale

La méthode de la génération symbolique des instructions "case" est similaire à celle utilisée dans l'exécution symbolique. La différence entre les deux méthodes est la manière d'analyser le programme. L'exécution symbolique considère un seul chemin d'exécution pour analyser le programme tandis que l'évaluation symbolique globale utilise tous les chemins d'exécution du programme.

De l'application à l'analyse d'effet de changement

Par rapport à l'application de l'exécution symbolique, l'évaluation symbolique globale donne un moyen pour analyser les effets d'un changement quand les modifications dépassent la portée d'un seul chemin d'exécution du programme ou à la suite d'une modification, s'il n'y a plus de correspondance entre les graphes du programme original et ceux du programme modifié. Donc, l'utilisation de l'évaluation symbolique globale dans l'analyse de changement de programme peut être appliquée à un grand nombre de cas même si cette méthode est similaire à celle d'exécution symbolique.

3.5.5 L'évaluation symbolique dynamique

L'évaluation symbolique dynamique est une technique d'analyse dynamique qui fournit les représentations des variables de programme par la biais d'un chemin de programme déterminé par un jeu d'essai.

La méthode générale

L'évaluation symbolique dynamique implique le contrôle du programme pendant l'exécution par l'introduction dans le programme des instructions d'appels dans les fonctions. Cette méthode combine les techniques d'exécution symbolique et l'évaluation dynamique.

De l'application à l'analyse d'effet de changement

L'application de l'analyse d'effet de changement est analogue à celle d'exécution comparative sauf que les assertions dynamiques peuvent être énumérées et vérifiées symboliquement et numériquement. Par exemple, l'assertion que "la valeur de la variable Y dans un autre emplacement dans le programme original doit être trois fois celle de la variable Z dans un emplacement donné dans le programme modifié" peut être validé numériquement (e.g. $Z = Y = 0$). Mais cela peut être difficile symboliquement (e.g. $Z = 4X$, $Y = 6X$).

3.5.6 Les moniteurs d'exécution

Les moniteurs d'exécution sont basés dans la plupart des cas sur les méthodes d'évaluation symbolique. Ce groupe d'outils permet au programmeur de manipuler et de contrôler d'une façon interactive le procédé d'exécution d'un programme. De cette façon, l'ingénieur de maintenance peut examiner directement le comportement du programme et les effets des données d'entrées.

Dans cette catégorie d'outils il y a deux fonctions de base: la fonction de trace et la fonction de point de rupture[3]. La trace présente l'historique de l'exécution du programme en donnant une liste des instructions exécutées du programme. La trace

3. breakpointing

est utile pour identifier le chemin du programme (section 3.4.2) pour une condition particulière.

Le point de rupture permet au programmeur d'arrêter l'exécution du programme à des points spécifiés et examiner ou modifier les données. Ce qui permet d'examiner d'une façon interactive les effets des différents segments du code et d'explorer les conséquences des valeurs de données variées.

La table 3.8 donne une liste de quelques exemples des moniteurs d'exécution.

Nom d'outil	Langage supporté
C-Tracer	C
CICS Interactive Cobol Debugging System	Cobol
F BUG/1000	Fortran
IDM Interactive Debugging Monitor	RPG
XPF/Assembler	Assembler

Table 3.8

3.6 La technique d'analyse de dépendances

Le concept général de la technique d'analyse de dépendances est la définition et l'utilisation de graphe de dépendances qui résume les relations entre les différentes entités d'un système logiciel.

Le problème de la compréhension des relations de dépendances des différentes entités d'un programme est une tâche très difficile. Considérons par exemple quelques entités logicielles :

1. **Les modules de programmes** : fonctions, procédures, programme, commandes de contrôle, fichiers, etc.. Ce sont les entités qui appellent et transmettent des données des uns aux autres.
2. **Les données** : fichiers, structures de données, variables, etc.. Les données peuvent avoir des relations par calcul ou par partage de mémoire et par l'utilisation de leurs valeurs comme entrées et sorties des modules du programme.
3. **Les notations définies par un programmeur** : constantes, types, etc..

Il y a d'autres éléments du programme qui ne sont pas explicitement définits dans le code. Ils peuvent être une documentation, des concepts ou des pratiques dans l'esprit de concepteurs, programmeurs, et utilisateurs. Ils comprennent les éléments suivants :

1. **Le cahier des charges** : donne la définition du problème par rapport au monde réel.

2. **L'analyse des besoins et la spécification** : définit la fonctionnalité du système et les contraintes en relation avec le monde réel ou en rapport avec l'implantation.

3. **La conception** : décrit la décomposition fonctionnelle, flot de données, la conception de données, etc..

Chacun de ces éléments peuvent être liés les uns avec les autres et avec le code.

3.6.1 Le graphe de dépendances

Un concept qui simplifie d'une façon très forte la structuration des relations de dépendances est de considérer les relations entre les entités d'un programme comme un graphe. Chaque entité (module, fichier, variable, constant, etc.) d'un programme peut être représentée comme un noeud dans le graphe [Yau 81, Jackel 86, Gottler 90, Basson 92]. Les relations directes entre les différentes entités sont représentées par les sommets adjacents du graphe. Par exemple, les relations de dépendances dans un programme peuvent être classées comme suivantes :

- **les relations de flot de données** : ce sont les relations entre les objets quand la valeur d'un objet est utilisée pour calculer ou positionner la valeur d'un autre objet,

- **les relations de définitions** : ces relations sont définies quand une entité programme est utilisée pour définir une autre entité,

- **les relations d'appels** : elles sont définies entre deux modules de programme quand un module en appelle un autre, et

- **les relations de dépendances fonctionnelles** : elles sont définies entre les modules de programmes et les objets globaux qui sont creés ou mis à jour par les modules.

Le graphe de dépendance malgré sa simplicité, a beaucoup d'avantages :

1. Il permet d'avoir une vue cohérente sur les relations entre les entités logicielles à différents niveaux. Par exemple, une procédure de langage de supervision (JCL)[4] qui exécute un programme peut être vue comme une procédure ordinaire qui appelle une autre procédure ordinaire.

2. Chaque type de relation peut être vu d'une façon abstraite sans se soucier à priori du langage de programmation ou une considération pour le système dépendant. Cette façon de voir les choses facilite la conception automatique des outils d'analyse.

4. Job Control Language

3. Il permet d'identifier assez facilement les effets *indirects* sans une recherche fastidieuse à priori par le listing de références croisées. Par exemple, pour identifier les effets indirects d'une modification de la variable X, il suffit de rechercher tous les "fils" de X au niveau du graphe par le moyen d'une requête.

4. Il permet aussi de filtrer les fausses relations de dépendances. Par exemple, un grand programme qui contient beaucoup de variables appelées X. Un programmeur qui fait la trace de dépendances à la main doit continuellement se poser la question de savoir si c'est la même variable qu'il avait trouvée avant?

3.6.2 La classification de dépendances

Considérons un système logiciel typique qui est formé d'un ensemble de programmes qui fonctionnent d'une façon synchrone. Ce système contient en général, un grand nombre d'entités logicielles. Intuitivement le concept de "dépendances" entre ces entités est relativement clair; par exemple, si le changement d'entité A peut avoir des effets directs sur l'entité B, alors on peut dire que "B dépend-de A" ($A \rightarrow B$). Ce type de dépendance est considéré comme "directe".

Les dépendances d'un système logiciel se représentent par un "graphe de dépendances directes" (GDD) : GDD = (N, S). Les noeuds dans N représentent les entités du programme et les sommets adjacents (i, j) dans S représentent la dépendance directe entre entité j et entité i. Il est parfois utile de considérer simultanément les dépendances directes et indirectes. Par exemple, si X est utilisé pour calculer Y et que Y est aussi utilisé pour calculer Z, la fermeture transitive de GDD peut être utilisée pour trouver les dépendances indirectes.

3.6.3 L'utilisation du graphe de dépendances

La technique d'analyse de dépendances est actuellement utilisée dans le système "The Maintenance Assistance" au Centre de recherche en génie logiciel[5], Université de Floride et Université de Purdue [Wilde 87, 89, 92].

Le programmeur peut utiliser le graphe de dépendances pour avoir des informations sur un programme ou pour spécifier une modification. Dans ce cas, le graphe de dépendances fonctionne de la même façon que les outils d'analyse de programmes (section 3.5). Néanmoins, le graphe de dépendances fournit une base sur laquelle on peut construire des outils de maintenance. Quelques possibilités d'utilisation du graphe de dépendances sont les suivantes :

1. **La détection d'effet de bord d'un changement** : Un outil peut être construit pour comparer deux versions d'un programme pour identifier les entités affectées par un changement. Dans ce cas, le graphe de dépendance peut être utilisé pour tracer les effets indirects. C'est le cas du projet SERC.

5. SERC : Software Engineering Research Centre

2. **La métrique de dépendance**: Les métriques basées sur la théorie de graphe peuvent être appliquées au graphe de dépendances pour identifier les composants qui contribuent d'une façon excessive à la complexité du système et au coût de la maintenance. Si les variables affectées par un changement peuvent être identifiées alors, le coût d'un changement peut être estimé.

3. **Les graphes de flot de données**: On peut générer les graphes de flot de données pour chaque module d'un programme en utilisant le graphe de dépendances [Taha 87].

4. **La gestion de contrôle**: L'analyse de dépendance peut aider dans la gestion de contrôle en donnant les informations sur un changement proposé [Freedman 82].

5. **Divers**: L'analyse de dépendances est aussi utilisée dans les outils de "reverse engineering" et "re-engineering" [Canfora 92].

3.7 Conclusion

Nous avons présenté un certains nombre de techniques d'analyse de programmes dans la première partie de ce chapitre. Nous pensons que ces techniques peuvent être utilisées pour réduire la complexité de la maintenance de programmes et peut aussi augmenter la viabilité des programmes modifiés. Nous avons donné un aperçu des techniques qui existent et qui sont utilisées dans les outils d'aide à la compréhension d'un programme et l'impact de changement.

Parmi les trois techniques d'analyse de programmes exposées (l'analyse d'effet de bord, l'analyse dynamique, et l'évaluation symbolique), à notre connaissance seule la technique d'analyse d'effet de bord d'un changement est utilisée dans des outils d'analyse de changement dans un programme. Tandis que la technique d'effet de bord est la plus utilisée, elle est aussi limitée. Elle est limitée car dès que les effets ne sont plus localisés elle échoue. De plus, c'est une technique qui est basée sur la "forme" du programme et pas sur la syntaxe ni sur la sémantique du langage de programmation.

La technique d'analyse dynamique est une technique déjà incorporée dans les outils de test, et de débogage. L'application de cette technique est très difficile voire impossible dans certains type de programmes.

La troisième technique considérée est l'évaluation symbolique. Cette technique est coûteuse, très difficile à interpréter, et dans certains cas elle est impraticable.

Par rapport à notre problématique, aucune de ces méthodes ne peut être appliquée directement. Toutes les méthodes présentées dans les sections 3.3, 3.4, et 3.5 sont applicables en générale au code et aux effets de changement bien localisés. Pour nous, un logiciel est défini comme un ensemble minimal d'étapes. Cet ensemble contient le document d'analyse de besoins, la spécification de besoins, la conception et le programme (le code et le test). Avec cette définition nous avons considéré la technique d'analyse de dépendances. Cette technique donne plus d'avantages par rapport aux autres trois

techniques déjà considérées quand il s'agit du code seulement ou les autres étapes de cycle de vie du logiciel. Elle permet de spécifier à la fois, les changements localisés et non localisés par le moyen de relations de dépendances directes et indirectes. Elle permet aussi de spécifier les relations de dépendances entre les différentes étapes d'un système logiciel en utilisant la même abstraction qui est le graphe de dépendances. Vu cet avantage, nous allons utiliser la même technique comme une technique de base pour le modèle que nous proposons dans la suite.

Chapitre 4

Le modèle WHAT-IF

4.1 Introduction

Le modèle WHAT-IF est un modèle générique à base de connaissance sur la nature des liens entre les objets logiciels et les règles de propagation de changement. L'idée principale est d'avoir un modèle qui ne dépend d'aucun langage de programmation ou de spécification ni d'une méthode de conception particulière. C'est un modèle qui peut être appliqué au cas par cas.

Nous allons aborder la définition du modèle proprement. Pour cela, nous présentons d'abord les concepts sous jacents et en particulier la notion de vue sur un ensemble d'objets logiciel. Le modèle WHAT-IF sera présenté comme une combinaison de deux systèmes de vues : le système de vue en cascade et le domaine spécifique vue.

4.2 La problématique

Le modèle WHAT-IF, a pour objectif d'évaluer *à priori* l'impact du changement d'un objet logiciel sur les autres. Cet objet peut être en relation avec d'autres objets. Par conséquent, le modèle WHAT-IF doit évaluer l'impact de l'action initiale, c'est à dire :

- quels sont les objets concernés par cette modification,
- quels sont les liens entre les différents objets, et
- quelles suites d'actions devront être entreprises pour préserver la qualité du logiciel modifié.

Pour ces objectifs, le modèle s'appuiera sur les liens inter objets.
Notre approche consiste donc en :

- l'identification des types de liens entre les objets logiciels,

- l'identification des types de changements sur les objets, et
- la définition d'un système de vues logiciel.

Nous allons aborder la définition du modèle proprement. Pour cela, nous présentons d'abord les concepts sous jacents et en particulier la notion de vue sur un ensemble d'objets logiciel. Le modèle WHAT-IF sera présenté comme une combinaison de deux systèmes de vues : le système de vue en cascade et la vue liée à un domaine spécifique.

4.3 L'analyse conceptuelle

Soit un système logiciel S, représenté par un ensemble O_S d'objets liés explicitement par des relations de dépendance,

$$O_S = \{o_1, o_2, ..., o_n\},$$

et un ensemble T_S de types de changements sur les objets O_S,

$$T_S = \{t_1, t_2, ..., t_m\}.$$

Pouvoir analyser l'impact d'un changement dans S signifie que pour un changement donné $\{t_i, o_j\}$ $o_j \in O_s$ $t_i \in T_S$, où t_i "est appliqué" à o_j (i.e. l'objet o_j est modifié selon t_i), on peut définir :

$$f_{impact}\{t_i, o_j\} \rightarrow \{o_1, ..., o_i, o_k, ...\}$$

de sorte que l'ensemble des propriétés $P = \{p_1, p_2, ...\}$ de S restent invariantes.

f_{impact} représente la fonction d'analyse d'impact. Elle définit l'ensemble des objets $\{o_1, ..., o_i, o_k, ...\}$ que l'on doit modifier pour maintenir la cohérence dans S.

Dans cette définition de f_{impact}, on voit le problème de la cohérence après une modification au niveau d'un système logiciel. La question peut être posée, de savoir quelles sont les parties du système qui doivent être cohérentes après un changement ? Nous allons essayer de donner une réponse a cette question en prenant le point de vue d'un ingénieur de maintenance.

Considérons la description informelle suivante, représentant la stratégie d'un ingénieur de maintenance :

- je veux changer une partie de système, ET
- je veux laisser toutes les autres parties telles qu'elles sont, SAUF
- les parties du système qui peuvent être en "conflit" avec le changement désiré.

Par exemple, je veux laisser la fonctionnalité intacte mais modifier la performance ou bien changer l'environnement en laissant la fonctionnalité et la performance intactes.

Pour déterminer l'impact de ce changement, il faut donc :

- identifier les parties du système qui peuvent être concernées par le changement, et
- déterminer s'il y a des conflits ou non.

Exemple: Si une partie de la spécification fonctionnelle d'un système logiciel est modifiée alors l'implantation de la partie qui a été modifiée doit prendre en compte ce changement.

On peut résumer cette dépendance par une règle de "cohérence" de l'implantation :

changement(spécification - A) $\Rightarrow$ *changement(implantation - A)*

On sait que cette règle ne suffit pas car l'implantation n'est qu'une des dépendances qui nous intéressent.

D'autres exemples de dépendances intéressantes sont les suivantes :

- *"utilise"*

  Si A *utilise* B et si B change alors A doit prendre en compte le changement dans B.

- *"entrée"*

  Si x est une *"entrée"* de A, et le type de x change alors, le *"corps"* de A doit normalement changer. Par exemple, si x est une valeur numérique, la limite de la valeur de x peut être modifiée; etc ..

- *"la méthode d'accès"*

  Si x est une structure de donnée, et y sa méthode d'accès, alors si x change, l'implantation de y doit aussi changer; etc..

Ces règles caractérisent certaines classes de dépendances importantes pour définir la consistance au niveau d'un programme.

A partir de ces règles de dépendances, on note deux points importants :

1. la granularité de ces règles dépend des parties du système logiciel qui doivent changer, c'est à dire, les types d'objets qui sont affectés par un changement. Le choix de la granularité dépend de la catégorie de dépendances qu'on peut facilement manipuler d'une façon pratique. D'un côté, on peut identifier chaque expression ou variable dans le code comme une abstraction nécessaire, ce qui donne une granularité très fine. De l'autre côté, on peut identifier le sous-système tout entier, ce qui peut présenter une très grosse granularité.

2. les "connaissances" exprimées par les règles de dépendances dépendent de la "vue" qui a été utilisée pour décrire le système logiciel. Par exemple, la vue programme décrit le code, la vue architecturale représente la structure modulaire du système, etc..

Notre conclusion est la suivante:

- si la granularité des objets référencés est très fine alors, l'analyse de la consistance que l'on peut donner sera aussi très fine, car l'analyse de la consistance dépend des relations de dépendances entre les objets logiciels et la granularité de ces objets;
- si un système peut intégrer plusieurs vues simultanément alors, l'analyse de la consistance sera aussi détaillée, car l'intégration de deux ou plusieurs vues permet une analyse très détaillée.

La capacité de pouvoir analyser l'impact d'un changement et faire la propagation de ce changement dépend des facteurs suivants:

- les vues S, et pour chacune de ces vues, l'ensemble d'objets O_S qu'on va utiliser pour représenter le système,
- le type de changements T_S qu'on peut appliquer à chaque objet,
- les relations de dépendance entre les objets de S, et
- la fonction f_{impact} et ses propriétés.

Nous allons analyser explicitement chacun de ces facteurs.

4.4 Le système de vues d'un logiciel

Une vue est caractérisée par la nature des informations qu'elle contient. Une vue sur un logiciel est une abstraction de ces informations: cela peut être un sous ensemble des informations du logiciel, ou un ensemble d'informations déduites. Un flot de données, un graphe d'appel, une liste de références croisées, l'architecture générale sont des exemples de vue.

Un logiciel est un ensemble complexe d'informations liées entre elles parmi lesquelles on trouvera: le code du programme, le cahier des charges, la conception générale, la spécification, la conception détaillée, le flot de données, le manuel utilisateur, le manuel de maintenance, mais aussi le manuel d'installation, le guide d'utilisation, relevés de performances, etc..

Un système de vues prend en compte la sémantique entre les abstractions utilisées par une vue. Par exemple, le diagramme de flots de données et le graphe d'appels représentent un "système de vues structurelles".

4.5 Les types de systèmes de vues

Le système de vues pour le modèle WHAT-IF est basé sur une combinaison de deux systèmes de vues. Mais, avant de parler de ce système de vues, nous allons d'abord présenter les types de systèmes de vues, leurs points forts, leurs points faibles, et leurs adaptabilités à un système logiciel.

Un système de vues comprend les catégories suivantes :

- type 1 : le système de vue programme,
- type 2 : le système de vue structurelle,
- type 3 : le système de vue architecturale,
- type 4 : le système de vue basé sur un domaine spécifique,
- type 5 : le système de vue orienté vers le cycle de vie du logiciel

4.5.1 Le type 1 et 2 : Les vues programme et structurelle

On peut construire une vue de type 1 et 2 à partir d'un langage de programmation en utilisant un analyseur de code. La connaissance et la structure (flot de données, flot de contrôles, la structure modulaire, le graphe d'appel, etc.) du langage sont codées dans l'analyseur. On trouve la vue de type 1 dans la plupart des outils de maintenance, et la vue de type 2 dans les outils de "reverse engineering" [Avellis 92, Chikofsky 90].

Néanmoins, pour analyser l'impact d'un changement, les connaissances basées sur les vues de type 1 ou 2 ne suffisent pas. Le problème avec ce type de vue est donc dû au fait qu'il donne seulement les dépendances entre les objets au niveau du code. Cela implique que l'ingénieur doit garder à l'esprit (ce qui est très difficile voire même impossible pour un grand projet) les correspondances entre le code, la conception, la spécification, et l'analyse des besoins.

L'avantage d'un tel modèle de changement basé sur les vues de type 1 ou 2 sur les autres types de vues est qu'il est "simple" à réaliser.

4.5.2 Le type 3 : Le système de vue architecturale

Il y a des classes d'applications qui sont implantées en utilisant les architectures standard [Caldiera 91, Devanbu 90]. Dans ce cas, la fonctionnalité du système est structurée en couches (ou machines virtuelles), paquetages ou distribuées d'une façon prédéfinie.

Un système de vue architecturale est constitué de descripteurs de classe qui décrivent les propriétés des composants de l'architecture (machine virtuelles, le sous système, les données, etc ...) et ces composants.

Quand ces descripteurs sont représentés explicitement comme des connaissances, elles donnent deux avantages :

- Elles divisent le modèle de changement en sous ensemble d'informations qui sont en relation avec chaque composant architectural.
- Les connaissances architecturales peuvent être utilisées pour améliorer le mécanisme d'indexation qui sont en relation avec la description du changement sur une partie du logiciel. Autrement dit, les abstractions architecturales donnent des connaissances basées sur l'organisation modulaire d'un modèle de changement.

Le système de vue architectural est utile si et seulement si les connaissances architecturales sont suffisamment stables. C'est-à-dire, qu'elles sont valides pour un nombre suffisamment large d'applications. Malheureusement, on constate que le champ d'application du système de vue architectural est réduit quand celui ci devient plus riche [Avellis 92].

4.5.3 Le type 4 : Le système de vue basé sur un domaine spécifique

Ce système de vue est utilisé pour un domaine particulier de logiciel. Il permet de spécifier les objets de granularités fines [Harandi 90].

Ce système de vue peut être caractérisé (parmi d'autres) par :

- une vue liée à l'application : cette vue prend en compte les informations qui relèvent de l'évolution de la fonctionnalité d'une application;
- une vue liée à l'exécution ou l'implantation d'un logiciel : cette vue prend en compte les informations sur les changements qui relèvent du langage de programmation et de la structure du produit;
- une vue liée à la conception; etc..

Avec ce système de vue, la description d'un changement fait référence à une partie particulière de la structure (la donnée, le résultat, les contraintes d'exécution, etc) d'un objet. Cela permet de spécifier les dépendances entre les objets de granularités fines et facilite l'analyse d'impact et la propagation du changement.

4.5.4 Le type 5 : Le système de vue orienté cycle de vie de logiciels

L'utilisation de ce type de vue est basé sur l'hypothèse que le système logiciel S a été développé en utilisant un modèle de procédé de développement particulier (cascade, spirale, etc.), et qu'il existe des documents intermédiaires qui décrivent les différentes abstractions (besoins, spécification, conception, etc.) de S. Dans ce cas, chaque phase

du cycle de vie est un système de vue. On a donc un système de vue pour l'analyse des besoins, la spécification, la conception, l'implantation, etc..

Les avantages de ce type de vue sont les suivants:

- les informations relatives à un système logiciel sont plus accessibles parce que les développeurs de logiciel ont l'habitude d'utiliser des documents qui relèvent d'un modèle de procédé de développement;
- dans la plupart des cas, la description des changements est basée sur des abstractions qui sont décrites à l'aide d'un modèle de procédé de développement de logiciel;
- le système de vues orienté cycle de vie de logiciels couvre l'ensemble des différentes phases de cycle de vie d'un système logiciel.

Malgré ces avantages, ce type de vue est limité par le fait que les relations entre les objets d'expression de besoins et les autre objets sont génériques et elles ne reflètent pas la sémantique de textes décrivant les besoins par rapport aux autres objets dans le système. Donc, pour utiliser cette vue pour un modèle de changement, il faut qu'il y ait une autre vue comme la vue liée à un domaine spécifique qui permet de spécifier les objets d'expression de besoins et les relations entre ces objets après avoir fait une étude détaillée sur le document de besoins. C'est aussi vrai pour les autres phases de cycle de vie (e.g spécification, conception, etc.) dans le cas ou on veut spécifier les relations entre les objets de granularités fines.

4.5.5 La combinaison des vues

Chacune de ces vues a un mérite certain en décrivant chaque fois le nombre de points d'intérêt, elles permettent de comprendre plus facilement les facettes correspondantes du logiciel. Mais aucune d'elles, ne suffit à elle seule pour déterminer l'impact d'un changement. Il faut donc les combiner.

A titre d'exemple, on peut avoir les combinaisons suivantes (figure 4.1): (a), (b), (c), et (d).

- (a) une vue structurelle et programme,
- (b) une vue architecturale, structurelle, et programme,
- (c) une vue de type cycle de vie,
- (d) une vue architecturale, domaine spécifique, structurelle,et programme.

Il y a aussi la possibilité de tout combiner.

Nous restreindrons notre champ d'études pour le modèle WHAT-IF, aux logiciels développés selon un modèle de cycle de vie en cascade [Boehm 81], et dans ce modèle nous ne considérons que les étapes d'analyse des besoins, de la spécification, de la conception, et de programme.

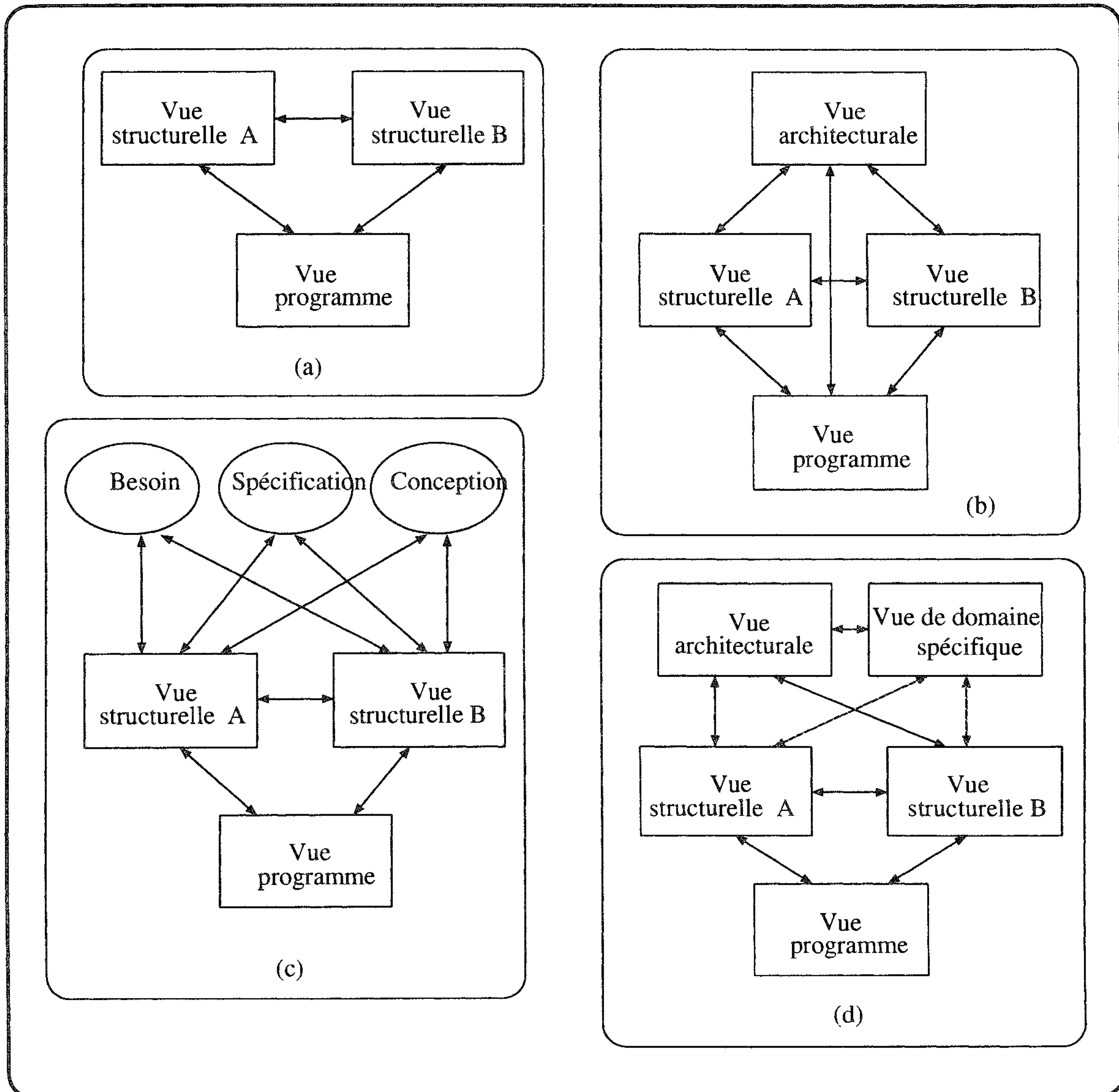


FIG. 4.1 - *La combinaison des vues*

4.6 WHAT-IF système de vues

Quels sont les systèmes de vues nécessaires pour le modèle WHAT-IF? La réponse a cette question dépend des fonctions attendues de WHAT-IF. Les fonctionnalités de base de WHAT-IF sont les suivantes :

- **"trouver"** : Soit un objet à modifier et le type du changement qu'on va y effectuer, trouver tous les objets qui peuvent être concernés par ce changement.
- **"assister"** : après avoir trouvé les objets concernés par un changement, donner les dépendances entre l'objet à modifier et les autres objets.
- **"décider"** : après avoir trouvé les objets concernés et les relations de dépendances, l'ingénieur de maintenance doit décider s'il faut continuer ou non avec le changement. Il doit évaluer le coût (en temps, en argent, et en matériel et il doit aussi décider s'il y a des conflits ou non) du changement.

Avec ces fonctionnalités, le système de vues de WHAT-IF est basé sur une combinaison de deux types de système de vues : le système de vues basé sur le modèle cascade et le système de vue basé sur un domaine spécifique.

Pourquoi cette combinaison de vues? Nous allons essayer de répondre à cette question en utilisant des exemples :

1. **Supprimer ou changer la déclaration d'une variable** : si la variable a été déclarée localement dans une procédure ou dans une fonction, l'ingénieur de maintenance n'a qu'à considérer l'impact de changement au niveau du code de la fonction. Mais, si la variable est globale, alors l'impact du changement est au niveau du programme et même au-delà parce qu'il faut aussi prendre en compte la conception et la spécification du programme. Dans cet exemple, on a besoin d'un système de vue qui permet de spécifier les relations entre les objets d'une vue de cycle de vie et les relations entre les objets de différentes vues.
2. **Modifier une fonction** : soit une modification au niveau des paramètres d'une fonction ou dans le corps d'une fonction. Cette modification peut avoir des effets sur d'autres fonctions ou procédures dans le programme. En plus, cette modification peut annuler l'hypothèse qui a été faite sur le rôle de cette fonction. Dans ce cas, l'impact de cette modification peut atteindre toutes les phases de cycle de vie du logiciel.

   Exemple:

   Procédure P() appelle procédure Q() et lui aussi appelle procédure S() (figure 4.2). S'il y a une modification dans P, Q et S sont concernées par cette modification. Les importations dans P et les exportations dans Q sont aussi concernées. La spécification et la conception de cette partie du programme peut être aussi concernée par cette modification.

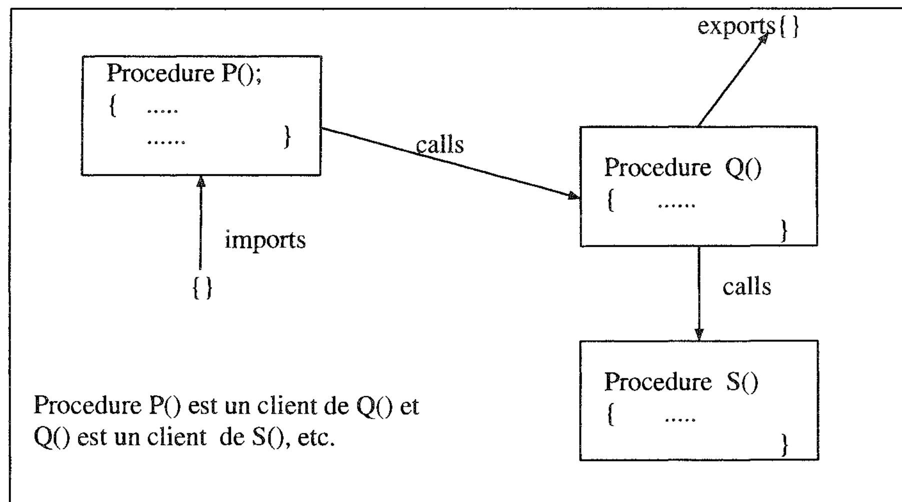


FIG. 4.2 - *Un exemple de modification*

3. **Un changement au niveau de l'analyse de besoins** : une addition ou une modification de besoin peut entraîner des impacts partout dans le système. Dans ce cas, on doit normalement "re-faire" toutes les parties du logiciel concernées par cette modification.

Le système de vues WHAT-IF comprend deux parties : le système de vues en cascade et le domaine spécifique vue (figure 4.3).

Le système de vues en cascade est composé de quatre sous systèmes de vues : l'analyse de besoins, la spécification, la conception, et le programme (figure 4.3).

Premièrement, le système de vues WHAT-IF permet l'expression plus détaillée d'un système logiciel. Le système logiciel est défini comme un ensemble de quatre grands composants : la définition de besoins, la spécification, la conception, et le code. Deuxièmement, la vue liée à un domaine spécifique permet la spécification des relations de dépendances entre les objets de granularité fine d'une phase donnée et celles des autres phases. Troisièmement, ce système de vues permet la combinaison de sous système de vues. En général, avec ce système de vues, on n'est pas obligé de tout concevoir en même temps. Même lorsque le système est entièrement conçu, on n'est pas obligé de rechercher l'impact d'un changement dans tous les sous systèmes de vues si l'impact ne concerne qu'un sous système ou deux. On peut donc avoir les combinaisons suivantes (figure 4.3) :

- r1 : conception et programme sous systèmes,
- r2 : spécification et conception sous systèmes,
- r3 : besoin et spécification sous systèmes,
- r4 : besoin et conception sous systèmes,
- r5 : spécification et programme sous systèmes, et

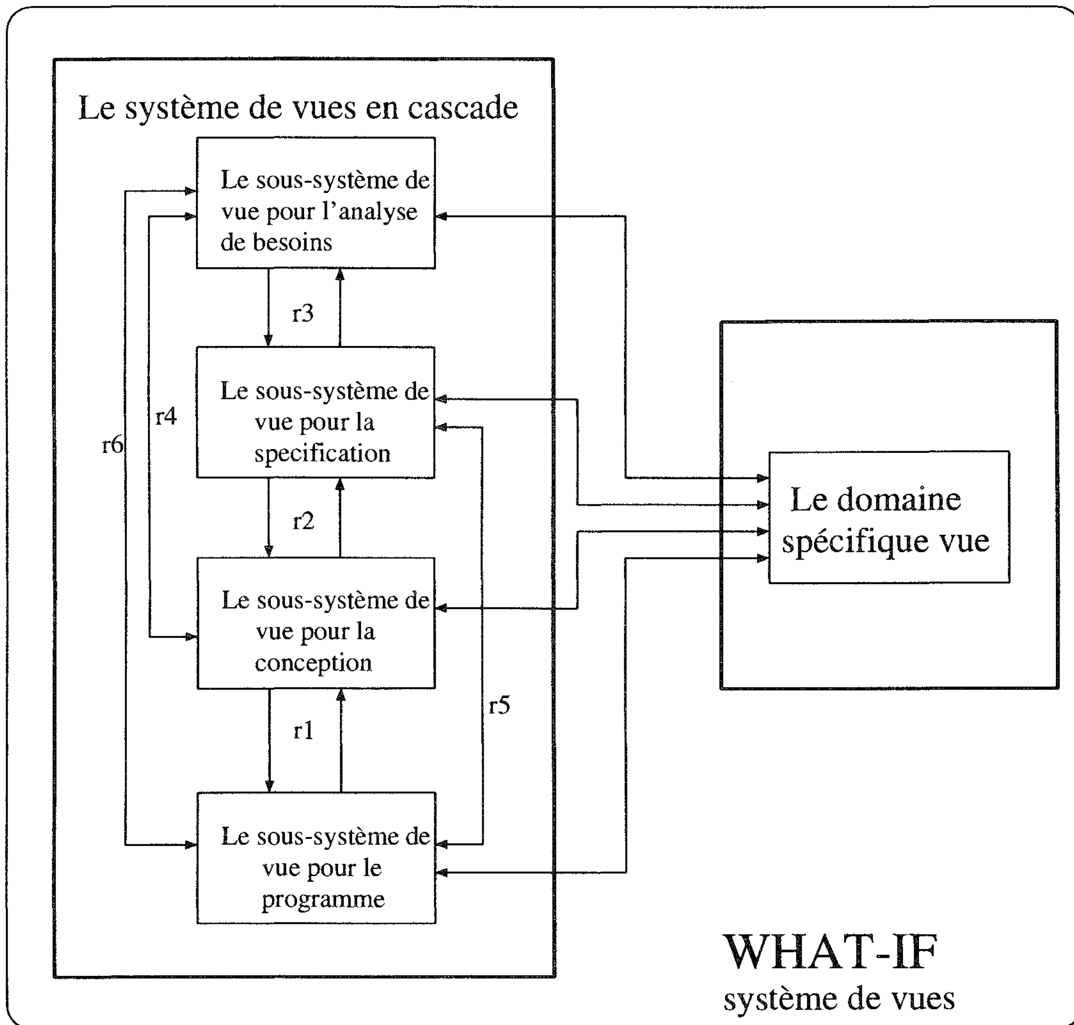


FIG. 4.3 - *Le système de vues WHAT-IF*

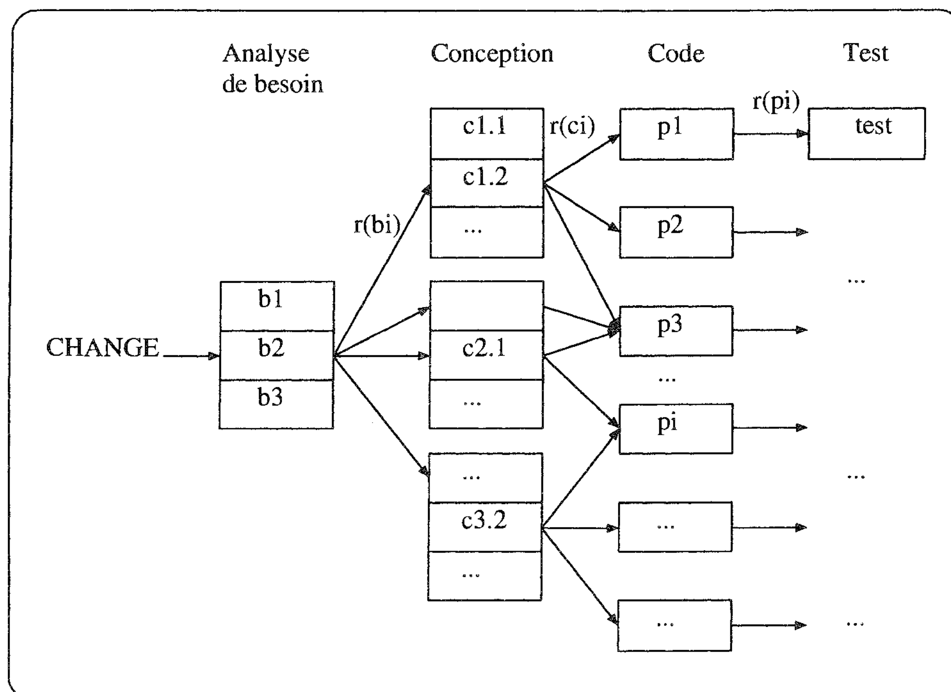


FIG. 4.4 - *La traçabilité horizontale*

- r6 : besoin et programme sous systèmes.

4.7 Les relations de dépendances

Les objets logiciel sont en rapports les uns avec les autres par des relations de dépendances complexes. La connaissance de ces dépendances est fondamentale pour l'analyse de l'impact et la propagation de changement [Wilde 89, Harandi 90, Ajila 91, 92].

Nous avons défini deux catégories de relations de dépendances [Basson 90, Ajila 91]. Ceci est en accord avec le système de vues WHAT-IF.

• **Les relations de dépendances** *inter-vues* : ce sont les relations entre les objets d'une sous-vue et une autre sous-vue. Ces relations permettent la traçabilité horizontale entre les différentes représentations du logiciel.

La figure 4.4 illustre la façon dont les relations de dépendances inter-vues sont déterminées à partir du document d'analyse de besoins. Chaque besoin est examiné et un rapport est établi entre le besoin et sa spécification. Ensuite, chaque spécification est examinée pour établir des rapports avec la conception. La même chose est faite pour la conception et le code. Le résultat est un graphe qui vérifie les dépendances inter-vues.

• **Les relations de dépendances** *intra-vue* : ce sont les relations entre les objets à l'intérieur d'une vue. Ces relations permettent la traçabilité verticale entre les objets d'une même représentation de logiciel, par exemple, la définition des besoins, la

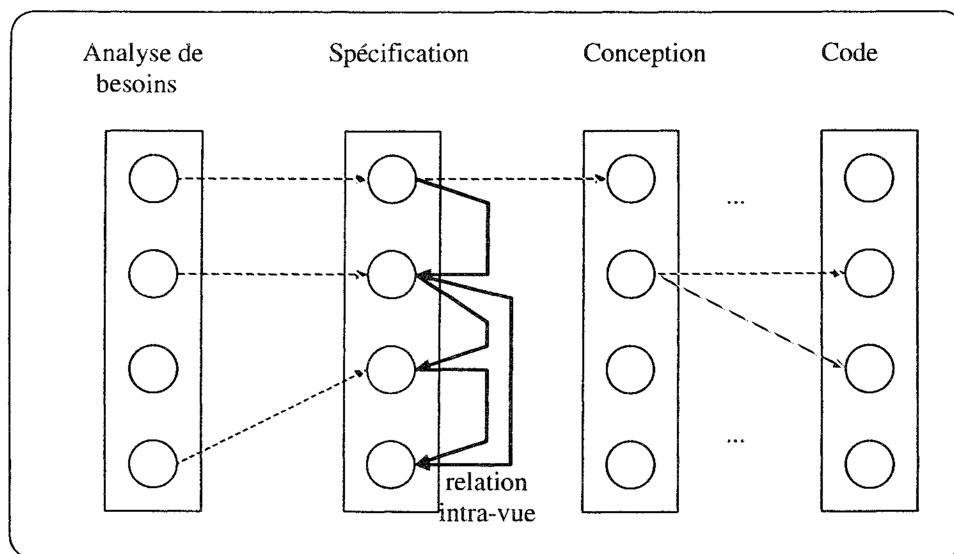


FIG. 4.5 - *La traçabilité verticale*

spécification, la conception, et le code (figure 4.5).

Par exemple, pour le code, on identifie globalement les types de relations de dépendances suivantes :

- les relations d'appel : ce sont les relations entre deux modules (fonctions, procédures, etc.) quand l'un appelle l'autre.
- les relations de dépendances fonctionnelles : ce sont les relations entre les modules et les données qui sont crées ou mises à jour par ces modules.
- les relations de définitions : ces relations existent entre les entités du programme utilisé pour définir une autre entité.
  Exemple : la définition de types, de tableaux, etc..
- les relations de flot de données : ces relations existent entre les données quand la valeur d'un objet peut être utilisée pour calculer la valeur des autres objets.

4.7.1 La taxinomie des relations de dépendances

Définitions et Notations

Nous utilisons les lettres majuscules en caractère gras pour représenter l'ensemble des objets logiciels (e. g. **A**, **B**, ...), les lettres minuscules en caractère gras pour les objets (e.g **a**, **b**, ...), et la lettre **r** ou **s** en caractère gras pour représenter une relation quelconque.

Définition

Soit $\mathbf{A}$ et $\mathbf{B}$ deux ensembles d'objets logiciels. L'ensemble des relations entre $\mathbf{A}$ et $\mathbf{B}$, notée $\mathbf{A} \leftrightarrow \mathbf{B}$, est définie comme l'ensemble des sous-ensembles de produit cartésien $\mathbf{A}$ X $\mathbf{B}$:

$$\mathbf{A} \leftrightarrow \mathbf{B} = \mathcal{P}(\mathbf{A} \text{ X } \mathbf{B}).$$

Ce qui signifie qu'une relation $\mathbf{r}$ entre $\mathbf{A}$ et $\mathbf{B}$ est un ensemble de couples $\{(\mathbf{a_1}, \mathbf{b_1}), (\mathbf{a_2}, \mathbf{b_2}), ...\}$ avec $\mathbf{a_i} \in \mathbf{A}$ et $\mathbf{b_i} \in \mathbf{B}$ pour $\forall i$.

Pour exprimer qu'un certain couple d'objets logiciels $\mathbf{a} \in \mathbf{B}$, $\mathbf{b} \in \mathbf{B}$ appartiennent à une relation $\mathbf{r}$, on va utiliser la notation suivante : **a r b**, par exemple, **a contient b** où **contient** est une relation.

Etant donné qu'une relation quelconque dans $\mathbf{A} \leftrightarrow \mathbf{B}$ est un sous-ensemble de $\mathbf{A}$ X $\mathbf{B}$, on peut parler de l'intersection de deux relations notée $\mathbf{r} \cap \mathbf{s}$, et de leur union notée $\mathbf{r} \cup \mathbf{s}$. On peut aussi exprimer le fait qu'une relation est un sous-ensemble d'une autre relation par $\mathbf{r} \subseteq \mathbf{s}$.

Définition

L'inverse d'une relation $\mathbf{r} \in \mathbf{A} \leftrightarrow \mathbf{B}$ est la relation $\mathbf{r^{-1}}$ dans $\mathbf{B} \leftrightarrow \mathbf{A}$ tel que $\mathbf{br^{-1}a} \Leftrightarrow \mathbf{a\ r\ b}$, $\mathbf{a} \in \mathbf{A}$ et $\mathbf{b} \in \mathbf{B}$.

Définition

La composition de deux relations $\mathbf{r} \in \mathbf{A} \leftrightarrow \mathbf{B}$ et $\mathbf{s} \in \mathbf{B} \leftrightarrow \mathbf{C}$ notée **r;s** est une relation dans $\mathbf{A} \leftrightarrow \mathbf{C}$ qui est valide entre les objets logiciels $\mathbf{a}$ et $\mathbf{c}$ si et seulement si **a r b** et **b s c** pour certain $\mathbf{c} \in \mathbf{C}$.

Définition

Pour un ensemble $\mathbf{A}$, la relation d'identité sur $\mathbf{A}$ notée $id(\mathbf{A})$ (ou tout simplement *id* en absence d'ambiguïté) est une relation "diagonale" qui est valide seulement entre chaque objet et l'objet lui même.

On appel *null* la relation vide.

Définition

Soit $\mathbf{r} \in \mathbf{A} \leftrightarrow \mathbf{A}$ (l'ensemble source et l'ensemble cible sont identiques). La *"puissance successive"* de r est définie comme :

$$\mathbf{r^0} = id$$
$$\mathbf{r^i} = \mathbf{r}; \mathbf{r^{i-1}} \ (i > 0).$$

Définition

Une relation $\mathbf{r} \in \mathbf{A} \leftrightarrow \mathbf{A}$ a une fermeture transitive notée $\mathbf{r}^+$, et une fermeture transitive réflexive notée $\mathbf{r}*$. où

$\mathbf{r}^+ = \mathbf{r} \cup \mathbf{r}^2 \cup \mathbf{r}^3 \cup \ldots$
$\mathbf{r}^* = id \cup \mathbf{r}^+$

Définition

Une relation $\mathbf{r} \in \mathbf{A} \leftrightarrow \mathbf{A}$ est définie comme :

- **transitive** si et seulement si $\mathbf{r}^2 \subseteq \mathbf{r}$ (ou $\mathbf{r}^+ = \mathbf{r}$),
- **réflexive** si et seulement si $id \subseteq \mathbf{r}$,
- **symétrique** si et seulement si $\mathbf{r}^{-1} = \mathbf{r}$,
- **anti-symétrique** si et seulement si $\mathbf{r} \cap \mathbf{r}^{-1} \subseteq id$,
- **fonctionnelle** si et seulement si $\mathbf{r}^{-1}; \mathbf{r} \subseteq id$ (une fonction partielle), et
- **totale** si et seulement si $id \subseteq \mathbf{r}; \mathbf{r}^{-1}$.

Nous utilisons ensuite ces définitions et notations pour spécifier un certain nombre de relations de dépendances. Il s'agit des relations génériques.

- **a contient b** : cette relation est valide si est seulement si l'objet représenté par **b** est un composant de **a**. Typiquement, **a** peut être une partie d'un système logiciel décrit à un niveau d'abstraction (e.g. spécification, conception, code, etc.) et **b** un composant (un module, une section dans un document, un objet de la conception, etc.) de cette description. On appelle **"b est-une-partie-de a"** la relation d'inverse ($contient^{-1}$).
- **a modèle b** : cette relation est valide si et seulement si **a** inclut la description de ce que **b** fait i.e. **b** est une façon de faire ce que **a** prescrit. **"b est-une-instance-de a"** est la relation d'inverse. Par exemple, le mode d'emploi d'une machine **modèle** l'utilisation de cette machine; une description abstraite d'un type de donné **modèle** l'implantation de ce type.
- **a est-un-complément-de b** : cette relation est valide si et seulement si **a** et **b** coopèrent pour atteindre un but. Par exemple, dans UNIX[6], les fonctions telle que *refer, tbl, eqn, troff* co-opèrent pour faire un traitement du texte.

propriétés :

"est-complément-de" est une relation symétrique.

est-une-partie-de ; contient $\subseteq$ **est-complément-de**

Dans cette notation le point virgule signifie "la composition de relations" et $\subseteq$ signifie "sous-ensemble de".

6. UNIX est une marque déposée de AT & T

La relation composite "**est-une-partie-de ; contient**" est une relation de composition. Cette relation est vrai entre deux objets **a** et **c** si et seulement si, pour un objet **b**, "**a est-une-partie-de b**" et "**b contient c**". De la même façon, si **r** et **s** sont deux relations alors $\mathbf{r} \subseteq \mathbf{s}$ implique qu'un couple d'objets logiciels reliés par **r** sont aussi **s**. Dans ce cas on peut conclure que "**a est-un-complément-de c**" si **a** n'est pas un sous-ensemble de **c**. Même si **a** est un sous-ensemble de **c** alors la relation de composition "**est-une-partie-de ; contient**" est une relation fonctionnelle ($\mathbf{r}^{-1}; \mathbf{r} \subseteq id$).

- **a est-déclaré-dans b** : cette relation est valide si **a** est déclaré dans le corps de **b**. Par exemple, dans un langage à structure de bloc, **b** peut être une structure de données et **a** déclaré dans **b** e.g. *PL/1, Algol 60 , Ada.*

- **a partage-information-avec b** : cette relation est possible si **a** et **b** ont accès à des informations communes. Cette relation est valide dans les langages de programmation structurés comme *Algol 60* et *Pascal.*

propriétés :

est-déclaré-dans $\subseteq$ **est-une-partie-de**

"**partage-information-avec**" est une relation symétrique.

partage-information-avec $\subseteq$ **est-déclaré-dans ; est-une-déclaration-de** où **est-une-déclaration-de** est la relation d'inverse de **est-déclaré-dans**.

- **a appelle b** : cette relation est valide entre les procédures si et seulement si **a** peut appeler **b**.

- **a crée b** : cette relation est possible si et seulement si **a** peut créer **b**. Elle est valide dans les langages ou systèmes tel que *Ada tache, PL/1 tache et Simula classes.*

- **a active b** : cette relation est possible dans les systèmes qui supportent coroutines ou processus parallèles (e.g. Ada). Elle est valide si **a** peut ré-activer l'exécution de **b** qui est suspendu.

- **a utilise b** : cette relation est valide si et seulement si **a** peut utiliser **b**. Par exemple, si **a** est un module et **b** est une opérande ou une fonction ou un bloc etc.

propriétés :

appelle $\bigcup$ **crée** $\bigcup$ **active** $\subseteq$ **utilise**

C'est-à-dire, **a** ne peut pas utiliser **b**, si **b** n'est pas visible dans **a**.

- **a est-visible-dans b** : cette relation est valide si et seulement si soit **b** est déclaré dans **a**, ou **b** est définie dans **a** (e.g. externe de C) ou **a** appelle **b**.

- **a définie b** : cette relation est valide si **a** peut définir **b**. Elle est valide dans C ou on peut différencier la déclaration d'une variable et sa définition (e.g. externe de C).

- **a modifie b** : cette relation est valide si et seulement si **a** est une instruction et **b** est une variable et la valeur de **b** peut être modifiée pendant l'exécution de **a**. Par exemple, si **a** est une instruction d'affectation alors la valeur de la variable de la branche gauche de **a** peut être modifiée.

propriétés:
définie $\bigcup$ **modifie** $\subseteq$ **est-visible-dans**

- **a importe b** : cette relation est valide si et seulement si **a** peut importer **b**. Par exemple, en Ada, on peut importer des objets pour rendre ceux-ci visibles dans le module d'importation.

- **a exporte b** : cette relation est vrai si et seulement si **a** peut exporter **b**.

Nous allons voir au chapitre cinq la définition précise des relations de dépendances pour le code Ada et les objets HOOD.

4.8 Les types de changements

Pour le WHAT-IF système de vue, on doit définir :

1. les types d'objets utilisés dans chaque sous-système de vue,
2. les types de changement qu'on peut opérer sur chaque objet.

Les objets manipulés par WHAT-IF sont identifiés et décrits de la façon suivante :

- les objets de document d'analyse de besoins (chapitres, sections, sous-sections, etc.). Dans la plupart des cas, les outils de production de documents utilisent un format *ASCII* pour sauvegarder les documents. Ce format sert à représenter le document sous forme physique. Ce format possède souvent une grammaire ce qui permet de l'analyser "facilement" [Queille 94, Escudié 94];

- les objets de la spécification (entités, actions, et assertions);

- les objets de la conception (les décisions, les plans, et les assertions : e.g en HOOD : les objets actifs ou passifs, les opérations, les classes, les environnements.); et

- les objets de l'implantation (ou de la réalisation) (fichiers, modules, procédures, fonctions, variables, etc. e.g. en Ada : packages, tasks, procedure, function, type, etc.).

Pour chaque objet dans chaque vue (ou sous vue) de S, on doit définir un catalogue de changements.

Selon la vue et les objets dans cette vue, les changements peuvent être triviaux tel que :

- l'élimination,
- l'insertion,
- la fusion,
- la répartition,
- le changement de type (variable, fonction, etc.)

Ils peuvent être complexes tel que l'identification de nouveaux besoins, la révision de la conception, etc..

4.9 La fonction f_{impact} et ses propriétés

On reprend ici la définition de f_{impact} donnée dans l'analyse conceptuelle (section 4.3). Pour un changement donné $\{t_i, o_j\}$ $o_j \in O_s$ $t_i \in T_S$, où t_i "est appliqué" à o_j (i.e. l'objet o_j est modifié selon t_i), la fonction f_{impact} est définie comme :

$$f_{impact}\{t_i, o_j\} \rightarrow \{o_1, ..., o_i, o_k, ...\}$$

où O_s est l'ensemble d'objets de S (un système logiciel) liés explicitement par des relations de dépendance, T_s est l'ensemble des types de changements sur les objets de O_s, et $\{o_1, ..., o_i, o_k, ...\}$ sont les objets directement concernés par le changement.

L'intérêt global de WHAT-IF est d'évaluer l'impact d'un changement et de donner les relations de dépendances entre l'objet à modifier et les autres objets dans S. Dans ce cas, il y a une observation :

Le fait qu'on peut trouver l'ensemble $\{o_1, ..., o_i, o_k, ...\}$ et les relations de dépendances entre o_j et l'ensemble $\{o_1, ..., o_i, o_k, ...\}$ prouve qu'on sait a priori comment appliquer t_i à o_j, et en particulier les effets du changement sur $\{o_1, ..., o_i, o_k, ...\}$,

En plus de cette observation on peut poser la question suivante :

Est-ce-qu'on peut estimer la fonction f_{impact} ? Si la réponse est oui, alors comment ?

Nous allons d'abord analyser l'observation et ensuite répondre à la question.

Pour l'observation, on définit la fonction f_{impact} de la manière suivante :

$$\begin{aligned} f_{impact}\{t_i, o_j\} \longrightarrow \; & \{o_1\{o_{11}(o_{111}(...), ...), o_{12}(o_{121}(...), ...), ..., o_{1n}(o_{1n1}(...), ...)\}, ..., \\ & o_i\{o_{i1}(o_{i11}(...), ...), o_{i2}(o_{i21}(...), ...), ..., o_{in}(o_{in1}(...), ...)\}, \\ & o_k\{o_{k1}(o_{k11}(...), ...), o_{k2}(o_{k21}(...), ...), ..., o_{kn}(o_{kn1}(...), ...)\}, ...\} \end{aligned}$$

où

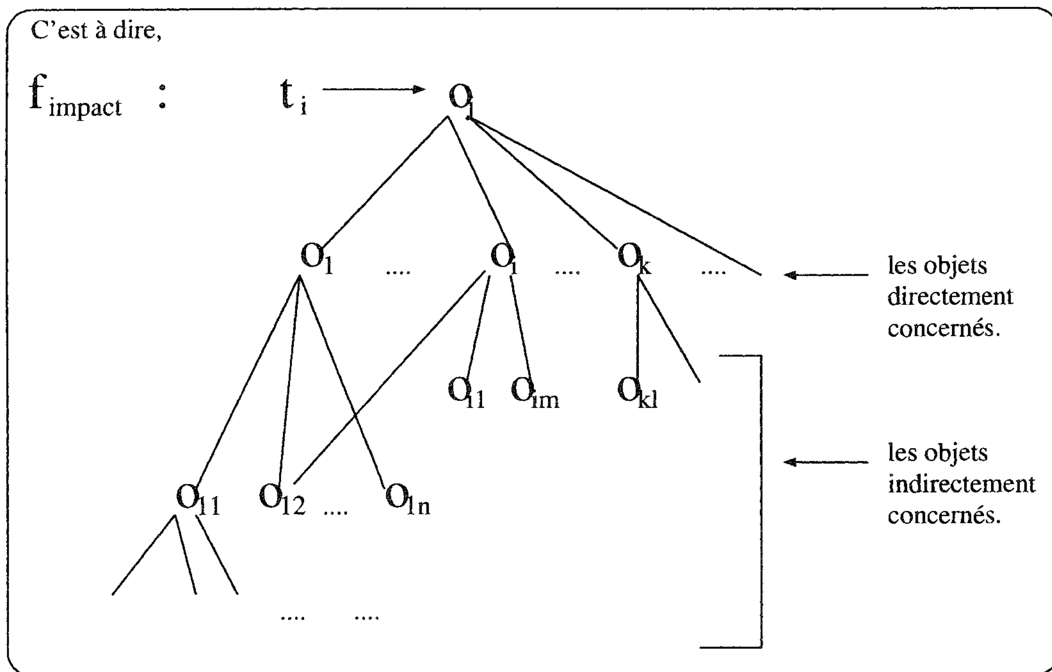


FIG. 4.6 - *La fonction f_{impact}*

$\{o_1, ..., o_i, o_k, ...\}$ sont les objets directement concernés par le changement et

$\{\{o_{11}(o_{111}(...), ...), o_{12}(o_{121}(...), ...), ..., o_{1n}(o_{1n1}(...), ...)\}, ...,$
$\{o_{k1}(o_{k11}(...), ...), o_{k2}(o_{k21}(...), ...), ..., o_{kn}(o_{kn1}(...), ...)\}, ...\}$

sont les objets indirectement concernés (figure 4.6).

Avec cette définition de f_{impact}, nous pouvons analyser les effets d'un changement en termes des objets concernés et la propagation de ce changement. Il peut y avoir au niveau de cette définition les chemins qui ne sont pas utiles. Mais, d'une façon pratique, on peut trouver des chemins à partir des objets directement impactés et les relations de dépendances entre ces objets et les objets indirectement impactés.

Pour la question, estimer la fonction f_{impact} revient à trouver les objets:

$\{o_1\{o_{11}(o_{111}(...), ...), o_{12}(o_{121}(...), ...), ..., o_{1n}(o_{1n1}(...), ...)\}, ...,$
$o_i\{o_{i1}(o_{i11}(...), ...), o_{i2}(o_{i21}(...), ...), ..., o_{in}(o_{in1}(...), ...)\},$
$o_k\{o_{k1}(o_{k11}(...), ...), o_{k2}(o_{k21}(...), ...), ..., o_{kn}(o_{kn1}(...), ...)\}, ...\}$

qui sont concernés par un changement. Pour cela, nous avons besoin des connaissances approfondies sur la signification de l'application de t_i à o_j (i.e $f_{impact}\{t_i, o_j\}$) et il faut que ces connaissances aussi soient représentées d'une façon explicite afin de mieux les manipuler. Dans ce cas, nous disposons de choix entre deux approches de

représentations [Rich 92] : l'approche faible[7], et l'approche forte[8].

4.9.1 L'approche "faible"

Pour qu'un outil quelconque de génie logiciel manipule les objets qui décrivent un système logiciel, ces objets doivent être représentés d'une manière explicite. Le choix d'une représentation est une balance entre la capacité de la représentation choisie et le coût de l'implantation de cette représentation.

Généralement, l'approche dite "faible" permet une représentation pas assez détaillées. C'est-à-dire, que l'on représente moins d'information explicitement, ce qui permet une implantation rapide. Or tant que les informations ne sont pas représentées explicitement, les outils manipuleront mal les objets relatifs aux informations. Par exemple, il est peu probable pour un outil qui représente l'analyse des besoins sous forme textuelle de trouver les inconsistances logiques dans le texte. Bien des outils logiciels fonctionnent au niveau du texte, de l'arbre abstrait, ou des diagrammes complétés avec du texte. De tels types de représentation laissent beaucoup d'informations implicites ou totalement absentes. Par exemple, l'arbre abstrait d'un programme donne explicitement la structure syntaxique du programme mais implicitement la structure de flot de donnée et de contrôle.

En utilisant cette approche, la fonction f_{impact} peut être codée comme une grande "table" avec deux clefs : l'objet o_j à modifier et le type de changement t_i.

Avec cette méthode de représentation, le "moteur d'inférence" qui dirige l'analyse de l'impact d'un changement utilise les clefs pour accéder à la table. On peut la résumer de la façon suivante : "donne moi la *victime* du changement (o_j) et le type du changement (t_i) et je donne la *valeur* de l'ensemble d'objets impactés".

Les problèmes liées à cette méthode de la représentation de f_{impact} sont les suivants :

- Les relations de dépendances ne sont pas représentées explicitement dans la table et à cause de cela, on doit "re-calculer" les relations de dépendances chaque fois. Cela signifie que l'ingénieur de maintenance doit avoir une idée précise sur les relations de dépendances entre les objets.

- Le fait que les types de changements soient codés explicitement dans la table signifie qu'on doit connaître pour chaque objet le type de changement (ou les types de changements dans certains cas).

- Avec cette approche, il est difficile d'introduire de nouveaux faits dans la table sauf si on connaît la technique qui a été utilisée pour construire la table et le "gestionnaire" de la table.

7. the "shallow" approach
8. the "deep" approach

4.9.2 L'approche "forte"

Cette approche permet une expression plus détaillée et plus complète des composants d'un système logiciel et ses tâches. Cette approche est connue sous le nom "système expert de la deuxième génération" [Avellis 91, Rich 92]. En général, cette approche présente trois caractéristiques : une représentation plus complète des composants logiciel, une méthode d'inspection détaillée sur ces composants, et une assistance intelligente.

Une représentation complète

Un outil ne peut pas raisonner sur les informations qui ne sont pas représentées explicitement. Donc, pour avoir un support compréhensible pour les tâches complexes d'un logiciel, on a besoin d'une méthode de représentation qui permette à la fois l'expression de la syntaxe et la sémantique des objets manipulés.

Un outil de support pour le changement complexe et algorithmique dans un logiciel doit normalement avoir toutes les détails de l'algorithme. Par exemple, dans le travail de Rich et al [Rich 92], le "Plan Calculus" a été utilisé pour représenter le programme et sa conception. Par rapport à la forme textuelle ou l'arbre abstrait, le "Plan Calculus" rend le flot de données et le flot de contrôle plus explicite. Ce qui facilite la manipulation directe du programme et sa conception.

Pour représenter les aspects non-algorithmiques de logiciel, on peut utiliser la représentation standard (e. g. Frames) à base de connaissances.

Une méthode d'inspection

Les ingénieurs de logiciels comme les ingénieurs dans d'autres domaines scientifiques, raisonnent rarement à partir des principes initiaux. En fait, ils raisonnent souvent dans les cas identifiés à partir de "faits" standards. Avec une base de faits d'une application particulière, il est possible d'effectuer certaines tâches de génie logiciel par inspection. Par exemple, dans une analyse par inspection, les propriétés d'un programme peuvent être déduites à partir des occurrences de faits qui font référence aux propriétés connues du programme. De la même façon, dans la conception, la reconnaissance des propriétés clefs de la spécification peut entraîner la construction de la conception par la combinaison des composants standards de la conception.

L'un des intérêts de cette méthode repose sur le fait que les "faits" peuvent être représentés d'une manière déclarative. Il y a deux points forts dans ce cas. Premièrement, l'on peut utiliser la même base de faits comme un support pour plusieurs tâches. Deuxièmement, les connaissances déclaratives sont plus faciles à enrichir.

Une assistance intelligente

Même avec la représentation forte et la méthode d'inspection basée sur les faits, on ne peut pas remplacer entièrement les ingénieurs de maintenance. Le mieux que

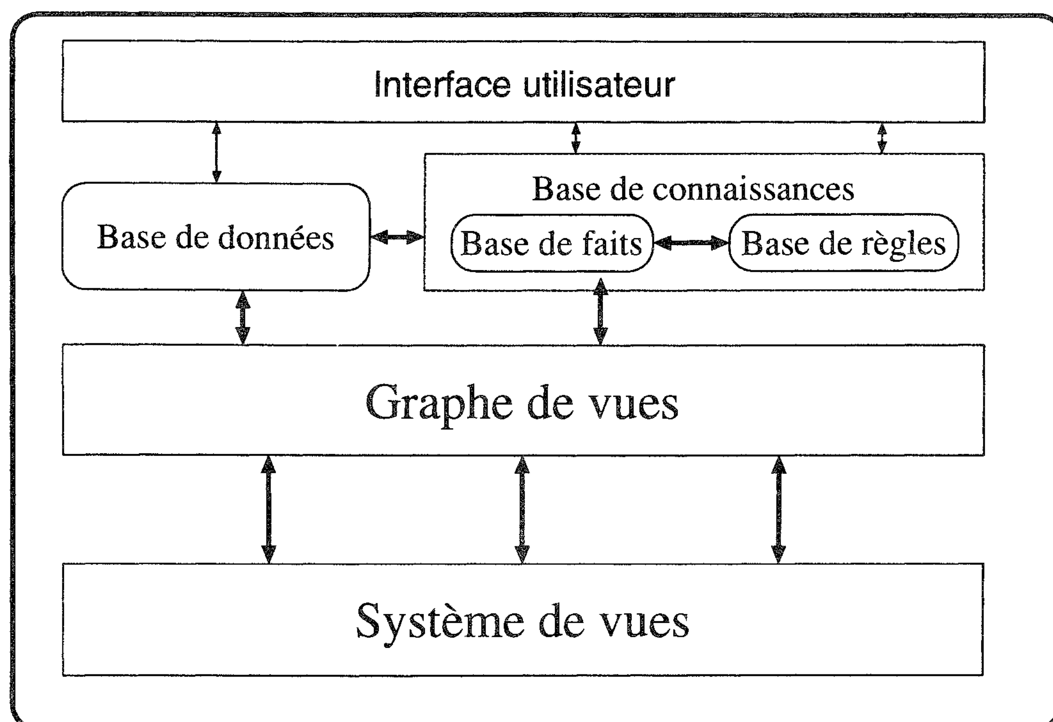


FIG. 4.7 - *L'architecture globale du modèle WHAT-IF*

l'on puisse faire est d'introduire la notion d'assistance intelligente pour que l'ensemble de système (logiciel, l'environnement et les ingénieurs) fonctionne d'une façon équilibrée et intelligente. Au lieu d'accepter et d'exécuter les commandes, une assistance intelligente peut effectuer des contrôles sur la nature des décisions, ajouter les détails qui manquent (si possible) et demander des conseils sur l'activation des opérations complexes. Ces capacités peuvent contribuer à la productivité de l'ingénieur et à la fiabilité du logiciel. Le second avantage d'une assistance intelligente est la possibilité d'expliquer les actions et les décisions qui ont été prises d'une façon claire pour que l'ingénieur de maintenance puise comprendre les contrôles effectués sur le système. En utilisant la méthode d'inspection basée sur les faits, il est plus facile pour l'ingénieur de comprendre le système et pour le système de comprendre les conseils de l'ingénieur.

Nous avons choisi l'approche forte pour représenter et manipuler la fonction f_{impact} et ses propriétés. Ce choix est basé sur le fait que l'approche forte permet à priori une expression plus détaillée et plus complète des composants d'un système logiciel et de ses tâches. La figure 4.7 montre l'architecture globale basée sur cette approche.

La représentation et la manipulation de f_{impact} se fait de la façon suivante :

1. Chaque sous système de vue est représenté sous la forme de graphes. Les nœuds correspondent aux objets de la vue et les arcs correspondent aux relations de dépendances entre ces objets. Donc, pour le système de vue entier, on a un "multi-graphe" [Yau 81, Basson 92, Ajila 94]. Dans WHAT-IF, ce multi-graphe est composé d'un graphe de besoin, d'un graphe de spécification, d'un graphe de conception, et d'un graphe de programme.

2. Les relations de dépendances entre les nœuds sont organisées comme un ensemble de faits et stockées dans une base de faits (figure 4.7). Chaque domaine spécifique a son ensemble de faits particuliers. Les faits significatifs dans WHAT-IF seront vus au chapitre 5.

3. Les relations composites, les contraintes, et les connaissances relatives à l'application d'un changement particulier sont aussi organisées en règles éventuellement stockées dans une base de règles (figure 4.7).

4. Les informations spécifiques à chaque composant logiciel sont stockées dans une base de données (figure 4.7)

4.10 Champ d'application du modèle WHAT-IF

Nous avons présenté un modèle à base de connaissances pour l'analyse d'impact et la propagation du changement. Ce modèle peut être appliqué avec l'hypothèse que le logiciel d'application a été développé en utilisant le modèle de procédé de développement de logiciel en cascade [Boehm 81] et qu'il existe des éléments intermédiaires qui décrivent les différentes abstractions (au moins : analyse des besoins, spécification, conception et programmes) du logiciel et que ces abstractions existent sous forme électronique ou papier. Le code source doit être sous forme électronique. Cette application peut être faite de la façon suivante :

1. Partant de l'hypothèse ci-dessus, on fait une analyse détaillée sur l'ensemble des composants du logiciel. Pour cette analyse, on doit aussi définir un certaine nombre d'entités et de relations qui décrivent l'ensemble de composants. Cette analyse peut être faite de deux manières :

   (a) **Entièrement automatique** : Dans ce cas on doit implanter pour chaque système de vue un analyseur statique pour construire le graphe de vues. Cette façon de faire suppose que la spécification et la conception du logiciel aient été réalisées en utilisant une méthode formelle et que cette méthode puisse être analysée d'une manière automatique. Par exemple, on peut utiliser la méthode HOOD pour la conception et la méthode VDM[9] (ou le langage Z) pour la spécification. On suppose aussi que l'analyse des besoins a été réalisée en utilisant un outil construction de document à partir duquel on peut extraire des informations utiles pour le graphe de vues.

   (b) **Semi automatique** : Si les conditions dans (a) ne sont pas réunies à savoir, l'utilisation des langages ou méthodes formelles pour la spécification et la conception, alors le graphe de vues peut être construit d'une façon semi automatique. Dans ce cas le code au moins doit être analysé automatiquement. Le graphe de vues peut être complété manuellement ou des informations peuvent être ajoutées directement dans la base de données ou/et la base de

9. Vienna Development Method

faits et la base de règles tout en respectant la syntaxe et la sémantique des relations de dépendances.

2. A partir du graphe de vues on peut extraire les "faits" pour constituer la base de faits.

3. La base de données est constituée à partir des tables de symboles et l'arbre abstrait créent par l'analyseur statique.

4. La base de règles est formée de deux façons :

   (a) par les relations composites et les contraintes.

   Exemple: Cet exemple s'inscrit dans la vue programme et le langage de programmation Ada comme une application.

   Soit deux composants modulaires S_i et S_j d'une unité d'élaboration de Ada, la relation de dépendance "S_i **a-besoin-de** S_j" est vraie si et seulement si l'une des conditions suivantes est vraie.
   Pour S_i quelconque :
   - si S_j est une tâche Ada et S_i **-active-** S_j, **ou**
   - si S_j est un composant de la bibliothèque ou une spécification Ada et S_i **-définie-** S_j, **ou**
   - si S_j est un composant générique de Ada et S_i **-instancie-**S_j, **ou**
   - si S_j un sous-programme et S_i **-appelle-** S_j, **ou**
   - si S_j spécification Ada et S_i **-est-le-corps-de-**S_j.

   A partir de cette définition on établit la règle suivante :

   si S_i **-active-** S_j
   ou S_i **-définie-** S_j
   ...
   ou S_i **-est-le-corps-de-** S_j
   alors S_i **-a-besoin-de-** S_j

   (b) par les connaissances relatives à l'application d'un changement.

   Exemple: Cet exemple s'applique sur les objets de la vue programme.

   Soit un objet $< X >$ avec un type de donné $< T >$ et un type de changement <*delete-type-of*>. Pour ce type de changement on peut écrire une règle suivante :

   Si le type de changement est <*delete-type-of*>
   et Si l'objet $< X >$ et de type $< T >$
   et Si il n'existe qu'un seul objet de type $< T >$
   alors <*delete-type-of*> $< T >$.

5. Il faut ensuite établir des requêtes sur la nature des relations entre les composants logiciel et les types de changements (cf. section 5.5) Ces requêtes peuvent être pré-définies ou laissées à la charge de l'utilisateur. Pour le prototype WHAT-IF, les requêtes sont pré-définies (section 5.5.1). Chaque fois qu'il y a un changement, selon l'objet qui va changer et l'aspect du système logiciel concerné, les règles sont sélectionnées et intégrées dans la procédure qui est définie pour cette catégorie de requête. Ensuite, le "moteur d'inférence" en acceptant la requête peut accéder à la base de données par le biais du gestionnaire de la base de données ou exécuter les règles ou tous les deux à la fois pour répondre à la question. Dans le cadre du prototype WHAT-IF, le gestionnaire de requêtes est implanté en PROLOG. Le moteur d'inférence est le moteur d'inférence de PROLOG. Il est complété par un système qui permet d'interroger le moteur à partir de l'interface utilisateur et de déclencher le moteur selon le type de requête.

4.11 Conclusion

Par rapport à l'état de l'art et à la problématique générale, nous avons présenté un modèle générique d'analyse d'impact d'un changement.

Le système de vues basé sur une combinaison de deux systèmes de vues permet l'expression plus détaillée d'un système logiciel. Il permet aussi la spécification des relations de dépendances entre les objets de granularités fines. Avec ce système de vues, le changement et par conséquent, l'analyse du changement peut être faite à n'importe quel niveau du système logiciel. Cela permet une analyse en amont et en aval.

La manipulation des connaissances issues de différents composants d'un système logiciel repose sur une représentation complète, une méthode d'inspection, et une assistance intelligente par le moyen de la base de faits, base de règles, et le moteur d'inférence.

Nous avons appliqué ce modèle à deux sous-systèmes de vues pour valider nos idées. Cette application est présentée dans le chapitre suivant.

Chapitre 5

Le prototype du modèle WHAT-IF

5.1 L'introduction

Le prototype du modèle WHAT-IF est limité à deux sous-systèmes de vues (cf. figure 4.3) :

- le sous-système de vue pour la conception et
- le sous-système de vue pour le programme.

Nous avons choisi la méthode de conception HOOD pour le sous-système de vue pour la conception et le langage de programmation **Ada** pour le programme. Ces deux choix sont basés sur les points suivants :

1. Le langage de programmation **Ada** comporte des unités distinctes (paquetages, tâches, sous-programmes, unités génériques, etc.), avec des relations sémantiquement riches. Le langage Ada permet aussi la définition d'un composant de programme à deux niveaux, un niveau de *spécification*[10] et un niveau de *corps*[11].
2. La méthode de conception HOOD par opposition aux autres méthodes de conception classiques (qui privilégient la décomposition d'un système soit par les fonctions, soit par les données), permet la conception orientée objets. Cette méthode est une approche de synthèse basée sur le concept objet, qui regroupe à la fois les données et les opérations sur ces données [Lai 91, Heitz 92]. Ce concept permet de satisfaire les principes fondamentaux du génie logiciel, visant à l'obtention d'une *forte cohésion logique* et d'un *faible couplage*. L'annexe A donne une description plus détaillée de la méthode HOOD.
3. L'historique de la méthode HOOD est liée au langage Ada depuis le début de sa conception. La transition vers le code Ada consiste en la génération de squelettes

10. la partie spécification de Ada
11. la partie corps de Ada

d'unités de code à partir des définitions d'objets ODS[12] [Lai 91]. Cette génération pourra être automatisée ou semi automatisée. Il y a donc des relations importants entre composants de la conception et composants de programme.

4. Il existe actuellement, des outils et des environnements de construction de logiciels qui permettent la transition automatique du diagramme HOOD à travers le squelette d'objet (ODS) vers la partie de la spécification Ada et aussi dans certains cas une partie du corps Ada. Un exemple de ce type d'environnement est CONCERTO [Legout 92, Heitz 92].

Nous allons utiliser dans ce chapitre deux exemples tirés du livre, *Software Engineering with Ada* [Booch 88], pour illustrer les différents points forts de notre prototype. Le premier exemple, Global-Exemple 1 est basé sur le problème de système de base de données, le deuxième, Global-Exemple 2 est basé sur le problème de surveillance d'environnement. La spécification complète des deux exemples est présentée dans les annexes B et C.

5.2 L'architecture du prototype WHAT-IF

L'architecture du prototype comporte quatre sous-systèmes (cf. figure 5.1) :

- l'environnement CONCERTO,
- L'extracteur,
- la base de connaissances,
- l'abstracteur.

L'outil HOOD de l'environnement CONCERTO nous permet de générer automatiquement à partir d'un diagramme HOOD (réalisé aussi à l'aide du même outil), le squelette de description d'objet et la partie spécification de Ada. La spécification de Ada est ensuite enrichie pour avoir le code complet en Ada.

L'extracteur prend en entrée le squelette de description d'objet de HOOD et le code Ada. Il en produit les relations de dépendances directes pour les objets HOOD et pour les objets Ada. Ces relations sont transformées en des faits et des règles.

La base de connaissances comporte trois base; une base de données pour stocker les tables, une base de faits, et une base de règles (cf. figure 5.1). Nous avons séparé la base de faits d'avec la base de données d'une part pour diminuer l'explosion combinatoire de données. D'autre part, pour éviter de re-calculer chaque fois les faits à partir de la base de données. Nous avons aussi introduit la notion d'agenda qui nous permet de réduire d'avantage l'explosion combinatoire de données.

L'abstracteur reçoit les requêtes (cf. figure 5.1), exécute le procédé d'abstraction (moteur d'inférence) sur les relations et produit la réponse.

12. Object Description Skeleton

Dans les sections suivantes, nous allons expliciter les différentes parties du prototype.

5.3 L'extracteur

L'extracteur est constitué de quatre parties :

- l'analyseur d'ODS,
- l'analyseur du code Ada,
- le processeur,
- le transformateur.

Les analyseurs acceptent comme entrées, l'ODS et le code Ada, et produisent les arbres abstraits décorés (cf. figure 5.1). Le processeur accepte les arbres abstraits décorés et produit les tables et les relations de dépendances directes. Le transformateur prend les relations et les transforme en faits et dans certaines cas en règles.

5.3.1 Les analyseurs

Les deux analyseurs sont basés sur une même architecture (figure 5.2). Ils sont implantés respectivement avec LEX et YACC (deux outils standards de UNIX).

La syntaxe utilisée pour l'analyseur du code Ada est basée sur la syntaxe de Ada 83. Pour HOOD ODS, la syntaxe utilisée est proche de la syntaxe de ODS décrite dans [Lai 91].

5.3.2 Le processeur

Le processeur transforme les arbres abstraits décorés en structures de données qui représentent les relations de dépendances directes. Les tables (figures 5.3 et 5.4) issues du processeur sont stockées dans la base de données (cf. figure 5.1). Les figures 5.5 et 5.6 montrent les versions réduites des structures de données produites par le processeur.

La table d'objets Ada (cf. figure 5.5) montre les relations de dépendances directes.

Pour élaborer ces relations, nous avons défini un certain nombre d'entités :

- **variable(V)** : signifie que V est une variable de type *entier, réel* ou *énuméré*[13].
- **var-composée(C)** : signifie que C est une variable de type *array* ou *record.*
- **type(T)** : signifie que T est un type de donnée, et il décrit la structure ou le comportement de donnée.

13. Ada enumeration type

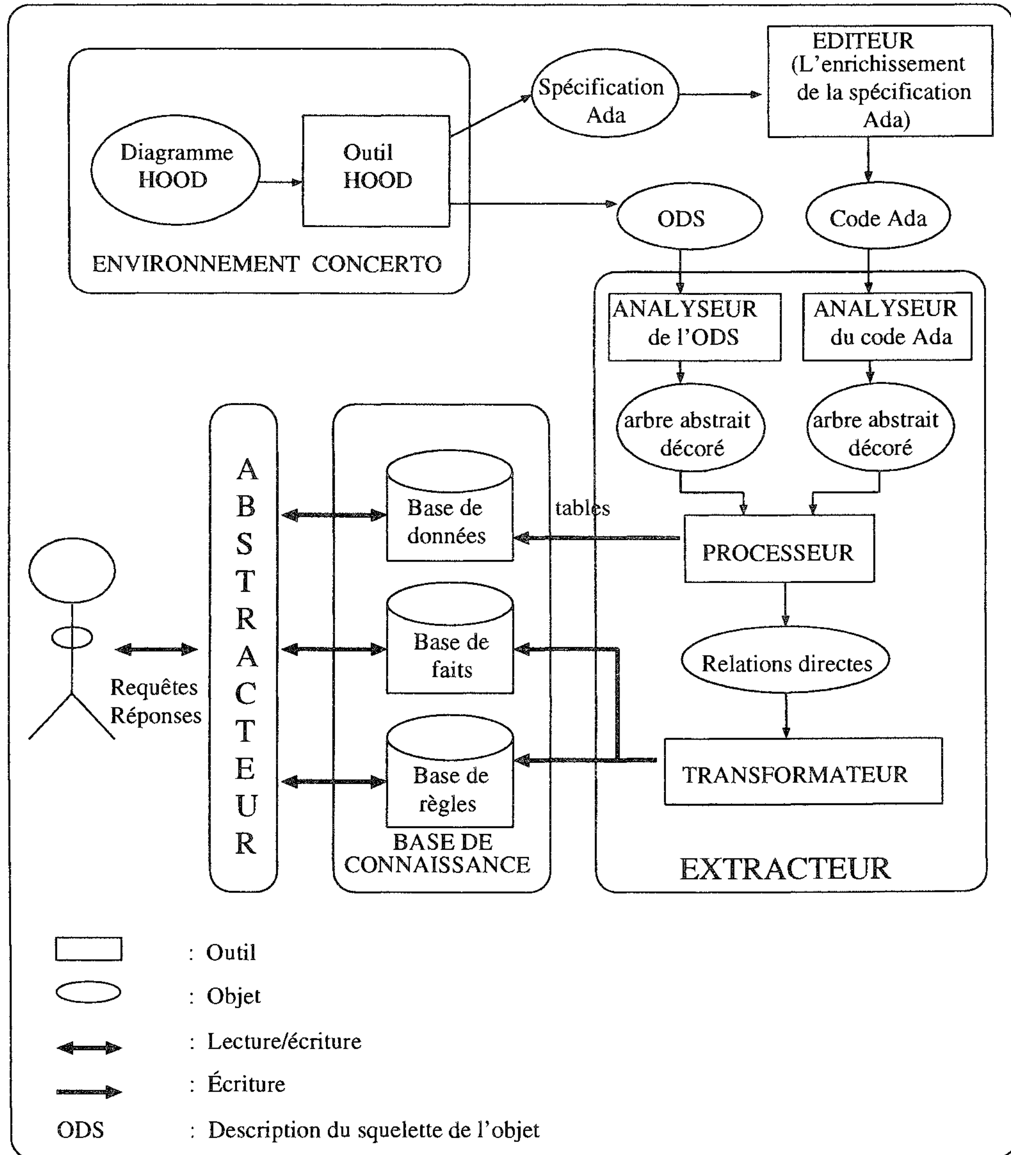


FIG. 5.1 - *L'architecture du prototype WHAT-IF*

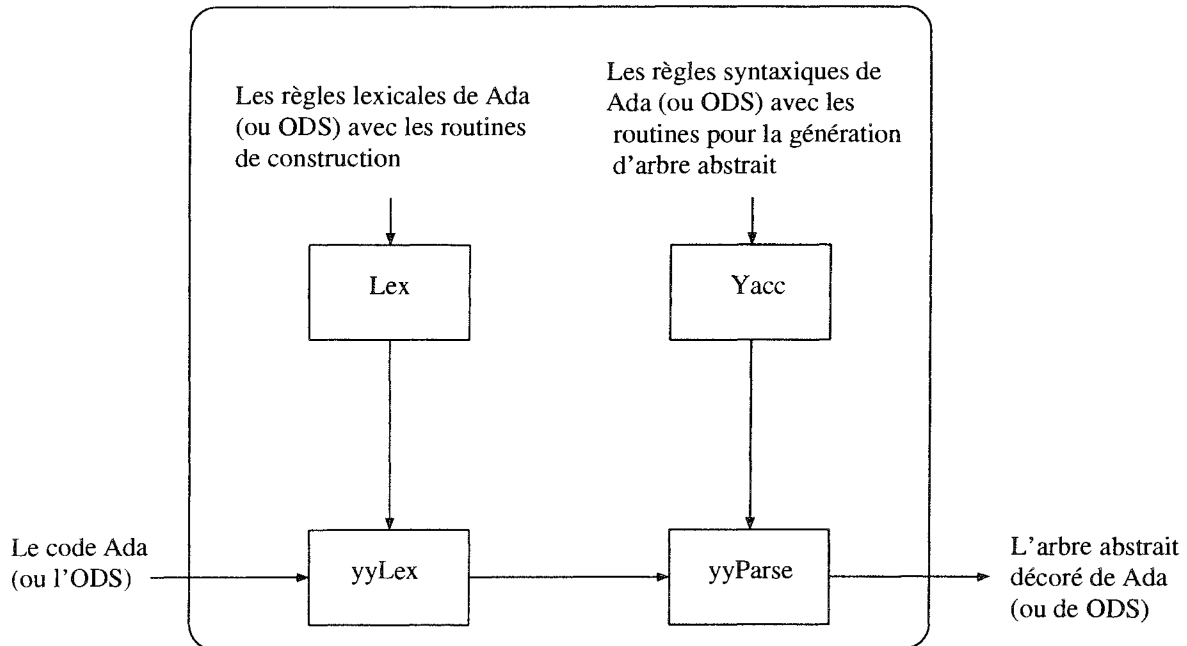


FIG. 5.2 - *L'analyseur*

No	Ident_Name	Ident_type	Ident_kind	Line_no.	Parent
0	monitor_temperatures	procedure_spec	PRS	3	text_io
1	command	END	END	5	monitor_temperatures
2	sensor_name	END	END	6	monitor_temperatures
...	...	...	...	...	...
8	alarm	task_spec	TKS	19	monitor_temperatures
...	...	...	...	...	...

FIG. 5.3 - *Exemple de la table d'identificateur Ada*

Object_Name	Operation_Name	Constraint
h_monitor_temperatures	h_disable_sensors	no_constraint
h_monitor_temperatures	h_enable_sensors	no_constraint
...	...	...
h_collection_of_sensors	h_force_record	aser
...	...	...
h_alarm	h_post_out_of_limits	aser_by_it(limit)

FIG. 5.4 - *Exemple de la table d'identificateur HOOD*

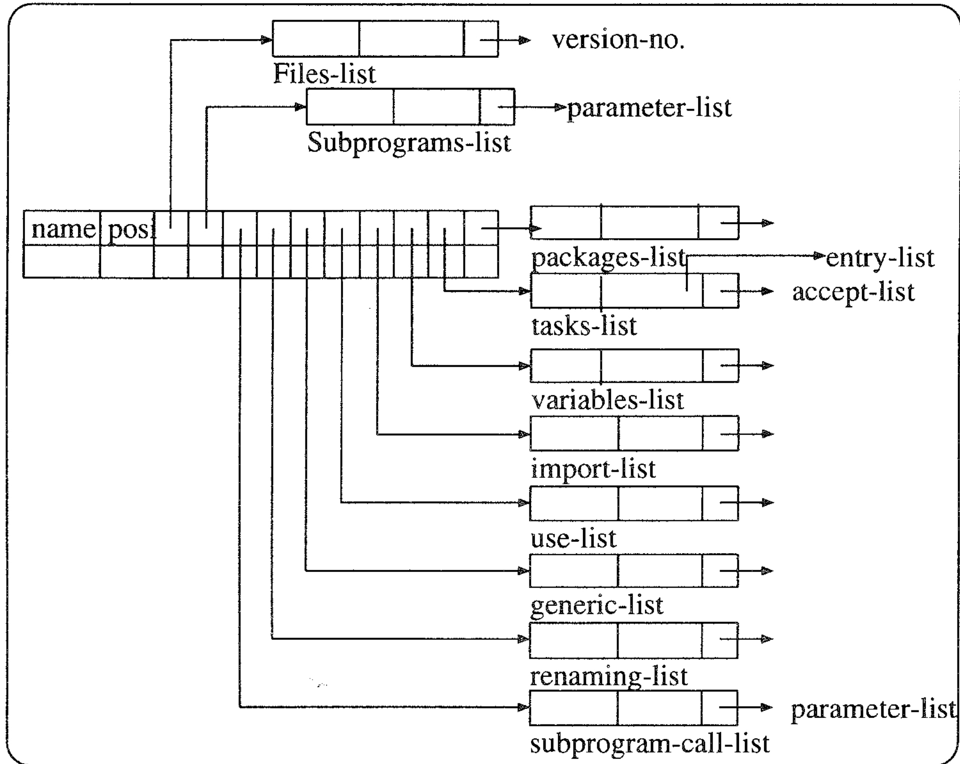


FIG. 5.5 - *La table d'objet Ada*

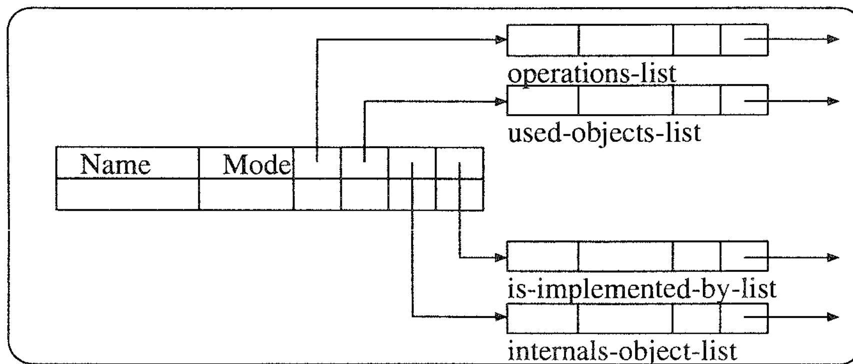


FIG. 5.6 - *La table d'objet Ada*

- **sous-programme(S)** : signifie que S est un sous-programme et, S doit être une fonction ou une procédure Ada.
- **objet-ada(X)** : signifie que X est un objet Ada et X peut être de type *paquetage, tâche, fonction,* ou *procédure.*
- **entier(N)** : signifie que N est un entier.

5.3.3 La spécification des relations de dépendances directes pour le code Ada

Avec la table d'objet Ada et les entités définies, nous avons spécifié les relations de dépendances directes suivantes pour le code Ada :

(R1) **est-paramètre-de :- {S, V, N}**

La relation *est-paramètre-de(obj, var, i)*, $obj \in S$, $var \in V$, $i \in N$ existe si et seulement si la variable *var* est déclarée dans le module *obj* comme un paramètre formel à la position i.

(R2) **appel :- {X, S}**

La relation *appel*(obj_1, obj_2), $obj_1 \in X$, $obj_2 \in S$ existe si et seulement si le module obj_1 appelle directement le sous-programme obj_2.

(R3) **valeur-de-retour :- {S, T}**

La relation *valeur-de-retour(obj, t)*, $obj \in S$, $t \in T$ existe si et seulement si la fonction *obj* renvoie après exécution une valeur de type t.

(R4) **obj-est-déclaré-dans :- {X, X}**

La relation *est-déclaré-dans*(obj_1, obj_2), $obj_1 \in X$, $obj_2 \in X$ existe si et seulement si le module obj_2 est déclaré dans le module obj_1.

(R5) **var-est-déclarée-dans :- {X, V_t}**, $V_t \in (V, C)$

La relation *var-est-déclarée-dans(obj, var)*, $obj \in X$, $var \in V_t$ existe si et seulement si la variable *var* est déclarée dans le module *obj* comme une variable.

(R6) **record-contient-var :- {C, V_t}**, $V_t \in (V, C)$

La relation *record-contient-var*(var_1, var_2), $var_1 \in C$, $var_2 \in V_t$ existe si et seulement si la variable var_2 est définie dan var_1 qui est de type *record.*

(R7) **obj-surnommé :- {X, X}**

La relation *obj-surnommé*(obj_1, obj_2), $obj_1, obj_2 \in X$ existe si et seulement si le module obj_1 est surnommé obj_2 en utilisant la clause *renames* de Ada.

(R8) **var-surnommé :-** $\{V_t,\ V_t\}$, $V_t \in (V, C)$

La relation *var-surnommé*(var_1, var_2), $var_1, var_2 \in V_t$ existe si et seulement si la variable var_1 est surnommé var_2 en utilisant la clause *renames* de Ada.

(R9) **importe :- {X, X}**

La relation *importe*(obj_1, obj_2), $obj_1, obj_2 \in X$ existe si et seulement si le module obj_1 importe obj_2 en utilisant la clause *use* de Ada. Dans ce cas, obj_2 est visible dans obj_1.

(R10) **est-spécifié-par :- {X, X}**

La relation *est-spécifié-par*(obj_1, obj_2), $obj_1, obj_2 \in X$ existe si et seulement si le module obj_1 est la partie spécification de obj_2. obj_2 est alors le corps Ada de obj_1.

(R11) **instancie :- {X, X}**

La relation *instancie*(obj_1, obj_2), $obj_1, obj_2 \in X$ existe si et seulement si le module obj_2 est une unité générique de Ada et obj_1 instancie obj_2 en utilisant la clause *new*.

(R12) **mappe :- {X, S, V, N}**

Nous utilisons le terme "mappë pour désigner la correspondance entre l'appel d'une fonction et les paramètres réels.

La relation *mappe*(obj_1, obj_2, *var*, *i*), $obj_1 \in X$, $obj_2 \in S$, $var \in V$, et $i \in N$ existe si et seulement si

- *appel*(obj_1, obj_2), et
- et que obj_1 défini *var* comme un paramètre réel à la position i.

Cette relation peut être combinée avec la relation (R1) pour définir la correspondance entre les paramètres formels et les paramètre réels.

5.3.4 Les entités et la spécification des relations de dépendances directes pour les objets HOOD

De la même façon que la table d'objet Ada (cf. figure 5.5), la table d'objet HOOD (cf. figure 5.6) contient le nom et le mode (actif ou passif) des objets HOOD. Pour

chaque objet, on trouve les listes suivantes :

- **opération** : Cette liste contient les opérations définies pour cet objet et leurs caractéristiques.
- **objets-utilisés** : Cette liste contient tous les objets utilisés. Un objet utilise un autre objet si le premier requiert une ou plusieurs opérations fournies par le second.
- **est-implanté-par** : Une opération du niveau parent peut être implantée par une ou plusieurs opérations enfants. Cette liste contient les opérations enfants qui implantent une ou plusieurs opérations parents.
- **objets-internes** : Afin de fournir une décomposition d'un système, un objet parent est décomposé en un ensemble d'objets enfants, qui collectivement fournissent les mêmes fonctionnalités que le parent. Donc, cette liste contient les objets enfants d'un objet parent.

En suivant la même démarche que pour les objets Ada, nous avons défini deux types d'entités qui nous permettent d'élaborer les relations directes de dépendances dans la table d'objet HOOD :

- **objet-HOOD(H)** : signifie que H est un objet HOOD. H peut être actif ou passif, et il peut être objet enfant ou parent.
- **opération-HOOD(O)** : signifie que O est une opération HOOD.

D'après la table d'objets HOOD et les deux entités définies, nous avons défini les relations de dépendances suivantes :

(R13) **définie :- {H, O}**

La relation *définie(obj, op)*, $obj \in H$, $op \in O$ existe si et seulement si l'opération *op* est définie dans l'objet *obj*. *obj* peut être actif ou passif.

(R14) **inclus :- {H, H}**

La relation *inclus*(obj_1, obj_2), $obj_1, obj_2 \in H$ existe si et seulement si obj_1 est le père objet de obj_2. Donc, obj_2 est le fils ou l'un des fils de obj_1.

(R15) **implantée-par :- {O, O}**

La relation *implantée-par*(op_1, op_2), $op_1, op_2, \in O$ existe si et seulement si :

il existe deux objets $obj_1, obj_2 \in H$ tel que

- *définie*(obj_1, op_1),

Objets HOOD	Objets Ada correspondants
objets (actif ou passif)	paquetages
opérations	procédures ou fonctions
contraintes sur les opérations	tâches
objets classes	paquetages génériques
structure de contrôle objet(OBCS)	tâches dans le corps de paquetages
structure de contrôle opération(OPCS)	unités séparées dans le corps de procédures

FIG. 5.7 - *La table de correspondances entre les objets Ada et les objets HOOD*

- *définie*(obj_2, op_2), et
- *inclus*(obj_1, obj_2).

(R16) **utilise :- {H, H}**

La relation *utilise*(obj_1, obj_2), $obj_1, obj_2 \in H$ existe si et seulement si :

il existe deux opérations $op_1, op_2 \in O$ tel que

- *implantée-par*(op_1, op_2), et
- *[(soit* obj_1 *et* obj_2 *sont actifs) ou*
  (soit obj_1 *est actif et* obj_2 *est passif) ou*
  (soit obj_1 *et* obj_2 *sont passifs)]*
  (cf. sous-section 5.5.5)

5.3.5 La spécification de la relation de correspondance entre les objets Ada et les objets HOOD

Nous avons également crée une table de correspondance (cf. figure 5.7) entre la table d'objets Ada et la table d'objets HOOD. A partir de cette table, nous avons défini la relation de dépendance suivante :

(R17) **correspond-à :- {H, X}**

La relation *correspond-à*(obj_1, obj_2), $obj_1 \in H$, $obj_2 \in X$ existe si obj_2 est un objet Ada (paquetage, tâche, procédure, ou fonction) et obj_1 est un objet HOOD (classe, objet, opération, opcs, obcs) et que la correspondance entre obj_1 et obj_2 est basée sur la règle de correspondance définie dans la table 5.7. L'inverse de cette relation (i.e $correspond^{-1}(obj_2, obj_1)$) est aussi vraie.

5.4 Le transformateur

Le transformateur transforme les relations de dépendances directes issues du processeur en PROLOG *faits* et *règles*.

Nous décrivons ci-dessous rapidement l'emploi de certains termes dans le contexte PROLOG :

- **base de faits** : on appelle *base de faits* l'ensemble de faits relatifs au problème que l'utilisateur formalise en PROLOG. Formellement, les faits sont des clauses de Horn positives.
- **base de règles** : on appelle *base de règles* l'ensemble de règles relatives au problème que l'utilisateur formalise en PROLOG. Les règles sont des clauses de Horn strictes.
- **questions** : la *question* est une clause de Horn négative. C'est ce que l'utilisateur demande au système.
- **moteur d'inférence de PROLOG** : le *moteur d'inférence de PROLOG* est la partie du système qui réalise les inférences logiques sur les faits et la question posée par l'utilisateur de façon à donner une (ou plusieurs) réponse(s). Il s'agit d'un moteur non-monotone à chaînage arrière et à régime par tentatives. Il opère sur un ensemble de faits et de règles, donc un ensemble de clauses de Horn négatives ou strictes. Sur le plan théorique, le moteur PROLOG fait une preuve d'inconsistance sur la base de clauses de Horn [Haton 91, Alliot 94].

La transformation des relations de dépendances directes en faits et en règles se fait de la façon suivante :

1. Les entités définies pour les objets Ada et les objets HOOD sont transformés en faits;
2. Les relations R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R13, R14, et R17 sont aussi transformées en faits; et
3. Les relations R12, R15, et R16 sont transformées en règles.

La table 5.2 montre une partie de l'ensemble de faits générés à partir d'un exemple réduit d'un programme Ada (table 5.1) qui est une partie du Global-Exemple 2.

```
with text_io, system;
use text_io;
procedure monitor_temperatures is

  type command      is (disable, enable, record_status, set_limits);
  type sensor_name  is (lobby, main_office, warehouse, stock_room,
                        terminal_room, library, computer_room,
                        lounge, clean_room);
  type sensor_state is (disable, enable);
  type sensor_value is delta 0.5 range 0.0 .. 100.0;
```

```
  package command_io       is new enumeration_io(command);
  use     command_io;
  package sensor_name_io   is new enumeration_io(sensor_name);
  use     sensor_name_io;
  package sensor_value_io  is new fixed_io(sensor_value);
  use     sensor_value_io;

  task alarm is
        entry post_fault_in_sensor;
        entry post_out_of_limits(on_sensor : in sensor_name);
  end alarm;

  task collection_of_sensors is
        entry disable          (sensor     : in sensor_name);
        entry enable           (sensor     : in sensor_name);
        entry force_record     (of_sensor  : in sensor_name);
        entry set_the_limits   (for_sensor : in sensor_name;
                                low_limit  : in sensor_value;
                                high_limit : in sensor_value);
  end collection_of_sensors;

  task timer is
        entry interrupt;
        for interrupt use at 16#8E#;
  end timer;

  high_bound    : sensor_value;
  low_bound     : sensor_value;
  name          : sensor_name;
  user_command  : command;
  value         : sensor_value;

  task body alarm                 is separate;
  task body recording_device      is separate;
  task body collection_of_sensors is separate;
  task body timer                 is separate;

  begin

     ....

   end;

end monitor_temperatures;

separate(monitor_temperatures)

....
```

table 5.1

```
array(limit-check).
array(sensor-group).
constant(bits).
entry-for-param(of-sensor).
entry-for-param(on-sensor).
imports(collection-of-sensors, sensor-set).
imports(collection-of-sensors, set-package).
instantiates(sensor-set, set-package).
instantiates(sensor-value-io, fixed-io).
object-declared-in(force-record, collection-of-sensors).
object-declared-in(high-bound, monitor-temperatures).
package-body(sensor-set).
package-body(sensor-value-io).
procedure-spec(monitor-temperatures, 3).
record-contains-var(sensor-record, low-limit).
record-contains-var(sensor-record, value).
```

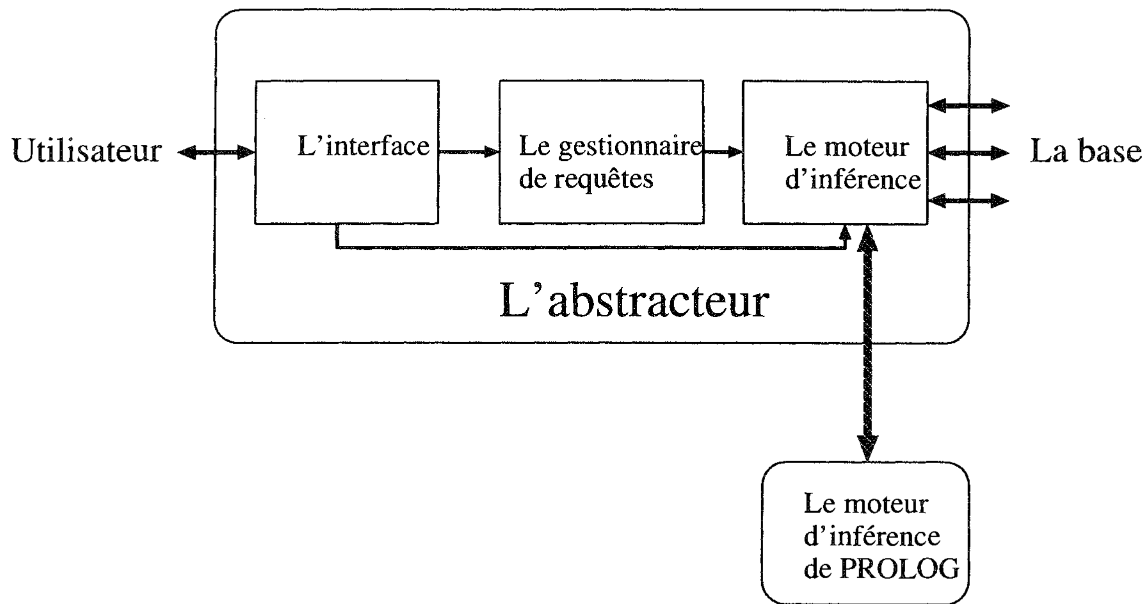


FIG. 5.8 - *L'abstracteur*

```
task-body(alarm).
task-body(collection-of-sensors).
uses(active-sensors, if-statement, 206).
uses(alarm, function-or-library-call-statement, 210).
uses(collection-of-sensors, function-or-library-call-statement, 86).
variable(active-sensors).
variable(fault-light).
```

table 5.2

5.5 L'abstracteur

L'abstracteur comporte trois parties (figures 5.1 et 5.8) :

- l'interface utilisateur,
- le gestionnaire de requêtes,
- le moteur d'inférence.

L'interface est implantée en XF. XF est un environnement intégré de programmation de développement graphique d'interface utilisateur. XF est fondé sur Tk, un ensemble de *widget* de type *Motif*[14]. L'accès au Tk se fait par Tcl[15], un langage de programmation interprété. L'avantage de cette approche est fondé sur le fait qu'elle permet la modification d'un programme d'application pendant que celui-ci est en exécution.

14. Une marque déposée de Open Software Foundation
15. Tool Command Language

Le gestionnaire de requêtes est implanté en PROLOG. Le moteur d'inférence est le moteur d'inférence de PROLOG. Il est complété par un système qui permet d'interroger le moteur à partir de l'interface utilisateur et de déclencher le moteur selon le type de requête.

L'abstracteur tel qu'il est conçu dans ce prototype n'est pas un "produit fini" mais, un noyau sur lequel les utilisateurs peuvent ajouter au système des relations par le moyen de règles au système ou utiliser les règles existantes d'une façon variée. Il est actuellement implanté d'une manière à répondre aux questions prédéfinies dans les requêtes.

5.5.1 Les requêtes et le gestionnaire de requêtes

Le gestionnaire de requêtes est codé en PROLOG. Le choix du PROLOG est basé d'une part sur le fait que la logique mathématique permet de mener une suite d'opérations sur des structures symboliques tout en préservant la véracité et d'autre part, la cohérence des conclusions déduites [Haton 91]. En logique, toute connaissance est représentée par une formule construite selon une syntaxe précise. Une base de connaissances peut être alors constituée exclusivement d'un ensemble de formules décrivant l'univers du discours. Le langage de programmation logique PROLOG fournit au concepteur un mode de représentation et de mécanismes de raisonnement totalement intégrés. Le raisonnement logique présente des avantages importants pour la réalisation d'un système à bases de connaissances. Les règles de raisonnement sont syntaxiquement et sémantiquement définies; elles sont simples et faciles à mettre en oeuvre [Haton 91, Alliot 94]. La clarté et la puissance d'expression de ce formalisme sont également séduisantes.

Néanmoins, la logique possède de gros inconvénients qui posent d'importants problèmes dès que l'on aborde des domaines complexes. Le concepteur d'un système fondé sur la logique est confronté à deux caractères fondamentaux de ce mode de raisonnement [Haton 91]:

- le manque d'efficacité des mécanismes de preuve: il s'agit en quelque sorte du prix à payer pour disposer d'une grande puissance d'expression. En pratique, il est sans cesse nécessaire de rechercher le meilleur compromis entre ces deux aspects;

- le manque de structuration des connaissances exprimées, obstacle important dans la construction et la maintenance de systèmes de grande taille. Des tentatives ont été faites pour remédier en partie à cet inconvénient en associant raisonnement logique et représentation de connaissances par objets structurés.

Le gestionnaire agit sur les requêtes en définissant une sorte d'agenda pour chaque requête. Une requête est basée sur un méta-règle et un méta-règle est un ensemble de règles:

<une requête> → *<un méta-règle>;*

<un méta-règle> $\rightarrow$ *< [règles]$^+$ >;*
<règles> $\rightarrow$ *< règle, [règle]* >;*
<règle> $\rightarrow$ *<une règle PROLOG>.*

Pour répondre aux besoins de l'utilisateur, nous avons défini trois types de requêtes :

- WHAT-IS,
- WHAT-IF,
- WHAT-LIST.

5.5.2 La requête WHAT-IS

La visibilité de chaque objet défini dans un logiciel est très importante pour l'analyse d'impact d'un changement.

La requête WHAT-IS permet d'interroger la base de données pour avoir une idée sur les caractéristiques d'un objet donné.

La syntaxe de WHAT-IS est :

WHAT-IS(identificateur, niveau),

identificateur: identifie le nom d'un objet à un niveau donné, et

niveau: indique une vue. Ici, il s'agit de HOOD ou Ada.

Nous donnons dessous deux exemples tirés du Global-Exemple 2.

<u>Exemple 1</u>
La requête :- WHAT-IS(sort, Ada)
La réponse : -

```
OBJECT VISIBILITY (Ada objects)
===============================

Object name    : sort
Object type    : procedure_body
Object kind    : procedure_body
Line No.       : 63

Starting point of (sort) in the program is line-no :- 63
Ending point of   (sort) in the program is line-no :- 88

(sort) define the following formal parameter(s) :

     Formal parameter            position in the list      mode type
     ================            ====================      =========

(1 )  project_records                    1                   IN
(2 )  records                            2                   IN
(3 )  record_index                       3                   IN
(4 )  size                               4                   IN
(5 )  criteria                           5                   IN
(6 )  list                               6                   OUT

Object name    : sort
Object type    : procedure_body
Object kind    : separate
Line No.       : 204
```

```
Starting point of (sort) in the program is line-no :- 204
Ending point of   (sort) in the program is line-no :- 223

(sort) define the following formal parameter(s) :

     Formal parameter                position in the list      mode type
     ================                ====================      =========

(1 )  project_records                       1                     IN
(2 )  records                               2                     IN
(3 )  record_index                          3                     IN
(4 )  size                                  4                     IN
(5 )  criteria                              5                     IN
(6 )  list                                  6                     OUT
```

Dans cet exemple, il s'agit d'un identificateur (sort) correspondant à 2 objets Ada. Deux procédures (procedure-body) avec paramètres, défini dans deux endroit différents du programme.

Exemple 2
La requête :- WHAT-IS(h_project, HOOD)
La réponse : -

```
OBJECT VISIBILITY (HOOD objects)
================================


Object name          : h_project
Object mode/type     : active

****

Internal object : (1) h_unit_information
Internal object : (2) h_data_base

****

Operation : (1) h_project_unit_name       --> constrained by : no_constraint
Operation : (2) h_project_sort_criteria   --> constrained by : aser_by_it
```

Dans Exemple 2, *h_ project* est un objet actif avec deux objets fils : *h_ unit_ information* et *h_ data_ base*. Il y a aussi deux opérations définies dans *h_ project* : *h_ project_ unit_ name* non contrainte et *h_ project_ sort_ criteria* contrainte par *aser_ by_ it*.

En HOOD toutes les opérations contraintes produites par un objet doivent être apparentes dans la partie visible de son OBCS[16] [Lai 91]. L'étiquette de contrainte attachée à ce type d'opération est représentée par les six possibilités suivantes :

- ASER[17] : Cette étiquette permet de formaliser en phase de conception, les interruptions logicielles ou matérielles (ASER-by-IT)ainsi que les messages entre processus systèmes ou transitant par un réseau;

- ASER-by-IT (interruption matérielle),

16. OBject Control Structure (Structure de contrôle Objet)
17. Asynchronous Execution Request (requêtes asynchrones)

- LSER[18] : Avec ce type d'étiquette, le flot de contrôle de l'objet appelant et de l'objet appelé se déroulent indépendamment en parallèle;
- HSER[19] : Avec ce type d'étiquette, le flot de contrôle de l'objet appelant est suspendu jusqu'à la fin de l'exécution complète de sa requête par l'objet appelé ce qui correspond à un protocole de communication *WAIT_ REPLY*;
- LSER-TOER[20] (durée),
- HSER-TOER (durée).

Chaque alternative peut être suivie d'un texte informel.

5.5.3 La requête WHAT-IF

La requête WHAT-IF est la plus importante parmi les trois types de requêtes définies. C'est elle qui permet l'analyse d'impact et de sa propagation. La syntaxe globale est la suivante :

WHAT-IF(*identificateur, niveau, type-de-changement*) où

identificateur et *niveau* sont définis comme dans WHAT-IS, et

type-de-changement qui dépend d'un niveau donné et qui peut être vide dans certains cas précis.

Cette requête donne l'impact direct de l'*identificateur* selon le *niveau* et le *type-de-changement*. Elle donne aussi la propagation de l'impact.

Cette requête contient quatre variantes :

- **WHAT-IF-Ada**(*identificateur, type-de-changement*),
- **WHAT-IF-HOOD**(*identificateur, type-de-changement*),
- **WHAT-IF-Local**(*identificateur, niveau*), et
- **WHAT-IF-Global**(*identificateur, niveau*).

La requête **WHAT-IF-Ada** s'adresse à tous les objets du niveau Ada et la requête **WHAT-IF-HOOD** les objets du niveau HOOD. Nous avons défini ces deux variantes de **WHAT-IF** d'une part parce que les types de changement ne sont pas identiques pour les deux niveaux. D'autre part, il existe des objets qui n'ont pas de correspondance. En fait, la table de correspondances (cf. figure 5.7) ne s'applique que sur les objets de grosses granularités (opérations, paquetage, tâches, etc.).

La requête **WHAT-IF-Local** s'adresse à tous les objets à un niveau donné. Elle agit sur les objets qui ne dépendent pas d'un type de changement quelconque et qui n'ont pas de correspondance au niveau global.

---

18. Loosely Synchronous Execution Request (Requêtes faiblement synchrones)
19. Highly Synchronous Execution Request (Requêtes fortement synchrones)
20. Time-Out Execution Request (Requêtes avec délais)

La requête **WHAT-IF-Global** quant à elle s'adresse à tous les objets du prototype, c'est-à-dire, les objets du niveau Ada et les objets du niveau HOOD. Elle ne dépend d'aucune contrainte de type de changement. Elle permet de propager l'impact d'un changement d'un niveau à l'autre.

5.5.4 La requête WHAT-IF-Ada et le type de changements au niveau Ada

Nous avons défini trois types de changements au niveau Ada :

- les changements au niveau de paramètres formels PAT[21],
- les changements au niveau de la définition de type TDT[22], et
- les changements au niveau des instructions d'affectations AST[23].

A part les relations de dépendance définies pour les objets Ada (cf. sous-section 5.3.3), nous avons aussi défini les relations suivantes pour la requête **WHAT-IF-Ada** :

(R18) **obj-est-déclaré-dans-globale :- {X, X}**; cette relation est définie comme une *fermeture transitive* de la relation **obj-est-déclaré-dans**, (R4).

La relation *obj-est-déclaré-dans-globale(*obj_1*,* obj_2*)*, $obj_1, obj_1 \in X$ existe si et seulement si obj_1 déclare directement obj_2 ou obj_2 est déclaré dans un module obj_i de tel façon que *obj-est-déclaré-dans-globale(*obj_1*,* obj_i*)* existe.

(R19) **var-paramètre-déclarée :- {X, V_t}**, $V_t \in (V, C)$; cette relation est une relation de composition entre la relation **var-est-déclarée-dans**, (R5) et la relation **est-paramètre-de**, (R1).

La relation *var-paramètre-déclarée(obj, var)*, $obj \in X$, $var \in V_t$ existe si :

- *var-est-déclarée-dans(obj, var)* existe, ou
- $\exists i \in N$*, tel que est-paramètre-de(obj, var, i)* existe.

Les relations R18 et R19 apportent une solution au problème de la portée d'un objet dans un programme. Le problème est de savoir si un objet x déclaré dans un module *m1* comme une variable ou un paramètre formel est visible dans un autre module *m2*.

En effet, selon la règle de visibilité en Ada,

1. un objet x est visible dans un module m (une fonction ou une procédure) si l'objet a été déclaré soit comme une variable ou comme un paramètre formel dans m;

21. PArameter change Type
22. Type Definition change Type
23. Assignment Statements change Type

2. un objet x déclaré dans un module *m2*, soit comme une variable ou soit comme un paramètre formel est visible dans un module *m1* si *m2* est visible dans *m1* (i.e. l'importation de *m2* dans *m1*) et si *m1* ne re-déclare pas x. De plus, s'il n'y a pas des modules déclarés directement ou in-directement dans *m2* qui re-déclarent x.

Pour prendre en compte cette notion de visibilité en Ada, nous avons spécifié ces deux *règles* de la façon suivante :

(R20) **Un objet x déclaré dans obj_1 est visible dans obj_2** si et seulement si

- Soit $(obj_2 = obj_1)$ ou
- Soit *var-paramètre-déclaré*(obj_1, x) et *var-paramètre-déclaré*(obj_i, x), et pour $\forall obj_i$ tel que
  (a) *obj-est-déclaré-dans-globale*(obj_1, obj_i), et
  (b) *obj-est-déclaré-dans-globale*(obj_i, obj_2)

L'exécution de la requête **WHAT-IF-Ada** est basée sur l'activation des relations R12, R18, R19, et R20 et sur les types de changements. Cet ensemble de relations est une sorte d'*agenda* qui agit sur la base de faits définie selon les relations R1 à R11.

Nous donnons ci-dessous deux exemples d'application de la requête **WHAT-IF-Ada**. Exemple 1 est tiré du Global-Exemple 2 et exemple 2 du Global-Exemple 1 (cf. Annexes B et C).

Exemple 1

La requête :- WHAT-IF-Ada(sensor, PAT)

La réponse :-

```
OBJECT VISIBILITY (Ada objects)
==============================

Object name    : sensor
Object type    : sensor_name
Object kind    : entry_formal_parameter
Line No.       : 25


Object name    : sensor
Object type    : sensor_name
Object kind    : entry_formal_parameter
Line No.       : 26

Object name    : sensor
Object type    : sensor_group
Object kind    : object declaration
Line No.       : 161

(sensor) == is defined as a parameter of ==> disable
 == position of parameter in the list is ==> 1
 == parameter mode type ==> IN

**** NOTA ****

To have a global visibility, you can invoke the query (what_if_Global)
```

```
 on the object : disable

(sensor) == is defined as a parameter of ==> enable
 == position of parameter in the list is ==> 1
 == parameter mode type ==> IN

**** NOTA ****

To have a global visibility, you can invoke the query (what_if_Global)
 on the object : enable

IMPACTS and PROPAGATION (Ada objects)
====================================

 == MODULE OR VARIABLE : sensor == is defined / visible in :
collection_of_sensors

 == MODULE OR VARIABLE : collection_of_sensors == is defined / visible in  :
monitor_temperatures


 == MODULE OR VARIABLE : collection_of_sensors == is defined / visible in  :
text_io

 == MODULE OR VARIABLE : monitor_temperatures == is defined / visible in  :
text_io

sensor == IS USED INSIDE/ AT LINE-NO. == assignment_statement = 178

sensor == IS USED INSIDE/ AT LINE-NO. == assignment_statement = 182

sensor == IS USED INSIDE/ AT LINE-NO. == assignment_statement = 200

sensor == IS USED INSIDE/ AT LINE-NO. == assignment_statement = 201

sensor == IS USED INSIDE/ AT LINE-NO. == assignment_statement = 207

sensor == IS USED INSIDE/ AT LINE-NO. == function_or_library_call_statement
 = 188

sensor == IS USED INSIDE/ AT LINE-NO. == function_or_library_call_statement
 = 61

sensor == IS USED INSIDE/ AT LINE-NO. == function_or_library_call_statement
 = 67

sensor == IS USED INSIDE/ AT LINE-NO. == if_statement = 208

sensor == IS USED INSIDE/ AT LINE-NO. == if_statement = 209

sensor == IS USED INSIDE/ AT LINE-NO. == select_statement = 177

sensor == IS USED INSIDE/ AT LINE-NO. == assignment_statement = 178
```

Dans cet exemple, il s'agit d'un objet Ada *sensor* qui est déclaré dans deux sous-programmes *disable* et *enable* comme un paramètre formel. L'impact direct et la propagation sont aussi 1 donnés. Par exemple, l'objet *sensor* est visible dans le module *collection_ of_ sensors* et ce module quant à lui est visible dans *monitor_ temperatures* et dans *text_ io*, etc.. On voit aussi que *sensor* est utilisé dans les instructions d'affectation et dans les autres parties du programme.

Exemple 2

La requête :- WHAT-IF-Ada(project_records, TDT)

La réponse : -

```
OBJECT VISIBILITY (Ada objects)
===============================

Object name    : project_records
Object type    : ATD
Object kind    : array type definition
Line No.       : 27

Object name    : project_records
Object type    : data_base
Object kind    : subprogram_formal_parameter
Line No.       : 63

Object name    : project_records
Object type    : data_base
Object kind    : subprogram_formal_parameter
Line No.       : 204

 (project_records) == is defined as a parameter of ==> sort
 == position of parameter in the list is ==> 1
 == parameter mode type ==> IN

**** NOTA ****

To have a global visibility, you can invoke the query (what_if_Global)
 on the object : (sort)

Since (project_records) is a generic type definition (ATD), it can be
modified by instantiating it to another type definition. BUT the
mapping between (project_records) a (subprogram_formal_parameter) and
an eventual (subprogram_call_actual_parameter) must be taken into
consideration before the modification.

IMPACTS and PROPAGATION (Ada objects)
=====================================

  == MODULE OR VARIABLE : project_records == is defined / visible in  :
 data_base


  == MODULE OR VARIABLE : project_records == is defined / visible in  :
 sort

  == MODULE OR VARIABLE : data_base == is defined / visible in  :  project

  == MODULE OR VARIABLE : sort == is defined / visible in  :  list_data


  == MODULE OR VARIABLE : sort == is defined / visible in  :  project

  == MODULE OR VARIABLE : list_data == is defined / visible in  :
 inquiry_operations


  == MODULE OR VARIABLE : inquiry_operations == is defined / visible in  :
 project
```

Il s'agit dans cet exemple d'un objet *project_records* déclaré comme tableau et utilisé dans le sous-programme *sort* comme un paramètre formel. Il y a 2 impacts directs :

- *project_records* $\rightarrow$ *data_base* et

- *project_ records → sort.*

Les propagations sont les suivantes :

- *data_ base → project,*
- *sort → list_ data,*
- *sort → project,*
- *list_ data → inquiry_ operations,* et
- *inquiry_ operations → project.*

5.5.5 La requête WHAT-IF-HOOD et les types de changement dans HOOD

Nous avons défini deux types de changements :

- **Les changements au niveau de types d'objets OMT**[24].

  HOOD distingue deux classes de types d'objets : les *objets passifs* et les *objets actifs* [Lai 91] (cf. Annexe A). Un objet est dit passif s'il possède uniquement des opérations qui s'exécute en mode séquentiel. Un objet passif n'a aucune répercussion sur le flot de contrôle de l'objet appelant. Un objet comportant au moins une opération contrainte ou dépendant de l'état interne de l'objet est dit actif.

- **les changements au niveau des opérations contraintes produites par un objet OCT**[25].

  Une opération peut être contrainte dans son exécution selon :

  1. *l'état interne* d'un objet qui dépend de l'exécution en cours d'une opération de l'objet. L'exécution de telle ou telle autre opération peut être fonction du contexte exprimé à partir des prédicats sur un ensemble de conditions ou d'événements. De telles opérations sont dites "à contraintes d'activation fonctionnelles" décrites dans l'OBCS.
  2. *le type d'exécution.* L'objet utilisé reçoit une requête ou un stimulus et réagit en générant un nouveau flot de contrôle à l'intérieur de l'objet utilisé. Les types de requêtes suivants sont identifiés : **HSER, LSER, ASER,** et **TOER.**

24. Object Mode Type
25. Object Constraint Type

Toutes les opérations d'un objet passif sont non contraintes. Certaines opérations d'un objet actif peuvent être non contraintes.

A part ces deux types de changements, la requête WHAT-IF-HOOD est basée sur deux type de relations HOOD :

- **La relation *use* entre objets** (flot de contrôle).

  Un objet utilise une ou plusieurs opérations fournies par un autre objet. HOOD autorise les objets actifs à utiliser librement tout autre objet passif ou actif. Il recommande toutefois, que les objets passifs n'utilisent que des objets passifs.

  La relation *use* définit sur les objets (d'un même-niveau de parenté) une hiérarchie senior-junior. Les objets actifs sont sur des niveaux seniors tandis que les objets passifs sont placés sur des niveaux juniors. Un objet passif ne doit pas utiliser une opération contrainte d'un objet actif car l'objet passif pourrait hériter du comportement actif et entraîner une contradiction avec ses propriétés de base.

- **La relation *include* entre objets.**

  Dans le but de produire une décomposition descendante d'un système, un objet parent est décomposé en un ensemble d'objets enfants. Les objets fils (enfants) de cet ensemble produisent la même fonctionnalité que l'objet parent. Chaque objet fils peut à son tour être décomposé en d'autres objets. Cette façon de décomposer définit la relation d'inclusion *include* entre objets où un objet contient d'autres objets (cf. Annexe A).

L'exécution de la requête WHAT-IF-HOOD est basée sur l'activation des relations R15 et R16 (cf. sous-section 5.3.4), et sur les deux types de changement. Ces deux relations agissent sur la base de faits définie selon les relations R13 et R14.

Nous donnons ci-dessous deux exemples de l'activation de la requête WHAT-IF-HOOD. Exemple 1 est tiré du Global-Exemple 2 (cf. Annexe C) et exemple 2 du Global-Exemple 1 (cf. Annexe B).

Exemple 1

La requête :- WHAT-IF-HOOD(h_inquiry_operations, OMT)

La réponse :-

```
OBJECT VISIBILITY (HOOD objects)
================================

Object name          : h_inquiry_operations
Object mode/type     : active

****


****

Operation : (1) h_inquiry_command         --> constrained by : aser
Operation : (2) h_inquiry_request         --> constrained by : aser
```

```
Operation : (3) h_inquiry_collect_statistics --> constrained by : no_constraint
Operation : (4) h_inquiry_list_data       --> constrained by : no_constraint
Operation : (5) h_inquiry_select_unit     --> constrained by : no_constraint
Operation : (6) h_inquiry_select_unit     --> constrained by : no_constraint

IMPACTS and PROPAGATION (HOOD objects)
======================================


h_inquiry_operations == defines == h_inquiry_command

GLOBAL MAPPING ===> h_inquiry_operations :- IS MAPPED TO ===>
inquiry_operations

MICRO-CONCLUSION
================

1
-->  Since (= h_inquiry_operations =) an <active object>
     used (= h_project =) also an <active object>.

     Therefore the mode of (= h_inquiry_operations =) CANNOT be
     changed because :-

     A <passive object> can ONLY use another object that is <passive>
2
-->  Since (= h_inquiry_operations =) an <active object>
     used (= h_request =)  a <passive object>.

     Therefore the mode of (= h_inquiry_operations =) CAN be
     changed to <passive> because :-

     A <passive object> can use another object that is <passive>
3
-->  Since (= h_inquiry_operations =) an <active object>
     used (= h_collect_statistics =)  a <passive object>.

     Therefore the mode of (= h_inquiry_operations =) CAN be
     changed to <passive> because :-

     A <passive object> can use another object that is <passive>
4
-->  Since (= h_inquiry_operations =) an <active object>
     used (= h_list_data =)  a <passive object>.

     Therefore the mode of (= h_inquiry_operations =) CAN be
     changed to <passive> because :-

     A <passive object> can use another object that is <passive>
5
-->  Since (= h_inquiry_operations =) an <active object>
     used (= h_select_unit =)  a <passive object>.

     Therefore the mode of (= h_inquiry_operations =) CAN be
     changed to <passive> because :-

     A <passive object> can use another object that is <passive>
6
-->  Since (= h_inquiry_operations =) an <active object>
     used (= h_select_unit =)  a <passive object>.

     Therefore the mode of (= h_inquiry_operations =) CAN be
     changed to <passive> because :-

     A <passive object> can use another object that is <passive>

**********
```

```
GLOBAL-CONCLUSION
=================

***  The mode type of (= h_inquiry_operations =) can only be changed
     if the mode type of (= h_project =) can be changed to passive.
     If not, the mode type (= h_inquiry_operations =) can not be
     changed.
```

Dans cet exemple, on voit qu'il y a une contradiction au niveau de *micro-conclusion*. La micro-condition 1 affirme que le type d'objet de *h_ inquiry_ operations* ne peut pas être changer tandis que les micro-conclusions 2, 3, 4, 5, et 6 présentent le contraire.

Pour résoudre cette contradiction, nous avons introduit la *global-conclusion*. Cette conclusion globale affirme que

> le type d'objet de *h_ inquiry_ operations* peut être changé si et seulement si le type de *h_ project* peut être aussi changé en passif en tenant compte d'autres dépendances. Sinon, le type de *h_ inquiry_ operations* doit rester actif.

<u>Exemple 2</u>

La requête :- WHAT-IF-HOOD(h_project_name, OCT)

La réponse :-

```
OBJECT VISIBILITY (HOOD objects)
================================

Object name          : h_project_unit_name
Object mode/type     : operation

****

(h_project_unit_name) is defined in : h_project
constrained by          : no_constraint

IMPACTS and PROPAGATION (HOOD objects)
======================================

h_project_unit_name == is implemented by == h_unit_name

GLOBAL MAPPING ===>  h_project_unit_name :- IS MAPPED TO ===> unit_name

CONCLUSION
==========

-->  Since the object (= h_project =) that defines
     (= h_project_unit_name =) is active then the constraint of
     (= h_project_unit_name =) can be changed, because :-
     any operation of an active object can be
     constraint or non_constraint.

**********
```

5.5.6 Les requêtes "WHAT-IF-Local" et "WHAT-IF-Global"

La requête **WHAT-IF-Local** est fondée sur l'activation des relations de même type que les relations de la requête WHAT-IF-Ada quand il s'agit du niveau Ada ou

de la requête WHAT-IF-HOOD quand il s'agit du niveau HOOD. La différence entre l'activation de WHAT-IF-Ada (ou WHAT-IF-HOOD) et l'activation de WHAT-IF-Local est le fait qu'il n'y a pas de contrainte de type de changement sur la requête WHAT-IF-Local. Quelque soit l'objet, l'activation de WHAT-IF-Local donne toutes les dépendances entre l'objet et d'autres objets dans le système.

Nous donnons ci-dessous un exemple d'activation de WHAT-IF-Local. C'est un exemple tiré du Global-Exemple 1.

La requête :- WHAT-IF-Local(sort, Ada)

La réponse :-

```
OBJECT VISIBILITY (Ada objects)
===============================

Object name    : sort
Object type    : procedure_body
Object kind    : procedure_body
Line No.       : 63

Starting point of (sort) in the program is line-no :- 63
Ending point of   (sort) in the program is line-no :- 88

(sort) define the following formal parameter(s) :

     Formal parameter             position in the list      mode type
     ================             ====================      =========

(1 )  project_records                     1                   IN
(2 )  records                             2                   IN
(3 )  record_index                        3                   IN
(4 )  size                                4                   IN
(5 )  criteria                            5                   IN
(6 )  list                                6                   OUT

---->

Object name    : sort
Object type    : procedure_body
Object kind    : separate
Line No.       : 204

Starting point of (sort) in the program is line-no :- 204
Ending point of   (sort) in the program is line-no :- 223

(sort) define the following formal parameter(s) :

     Formal parameter             position in the list      mode type
     ================             ====================      =========

(1 )  project_records                     1                   IN
(2 )  records                             2                   IN
(3 )  record_index                        3                   IN
(4 )  size                                4                   IN
(5 )  criteria                            5                   IN
(6 )  list                                6                   OUT

IMPACTS and PROPAGATION (Ada objects)
=====================================

  == MODULE OR VARIABLE : sort == is defined / visible in  :  list_data

  == MODULE OR VARIABLE : sort == is defined / visible in  :  project
```

```
  == MODULE OR VARIABLE : list_data == is defined / visible in  :
 inquiry_operations

  == MODULE OR VARIABLE : inquiry_operations == is defined / visible in  :
 project


sort == IS USED INSIDE/ AT LINE-NO. == function_or_library_call_statement = 212
```

La requête **WHAT-IF-Global** quant à elle agit sur l'activation de l'ensemble des relations définies :

- pour la requête WHAT-IF-Ada,
- pour la requête WHAT-IF-HOOD,
- la relation de correspondance entre les objets Ada et les objets HOOD. En fait, c'est cette relation qui permet de passer d'un niveau à un autre.

Cette requête ne dépend d'aucun type de changement. Elle agit sur l'ensemble d'objets définis pour les deux niveaux (Ada et HOOD). Si toutefois, la requête est activée sur un objet qui n'a pas de correspondance et quelque soit le niveau (Ada ou HOOD) d'activation, elle est réduite à la requête WHAT-IF-Local.

Nous donnons ci-dessous deux exemples : Exemple 1 tiré du Global-Exemple 2 et exemple 2 tiré du Global-Exemple 1.

Exemple 1

La requête :- WHAT-IF-Global(monitor_ temperatures, Ada)

La réponse :-

```
OBJECT VISIBILITY (Ada objects)
===============================

Object name    : monitor_temperatures
Object type    : procedure_spec
Object kind    : procedure_specification
Line No.       : 3

Starting point of (monitor_temperatures) in the program is line-no :- 3
Ending point of   (monitor_temperatures) in the program is line-no :- 101

IMPACTS and PROPAGATION (Ada objects)
=====================================

  == MODULE OR VARIABLE : monitor_temperatures == is defined / visible in  :
 text_io

monitor_temperatures ==  IMPORTS :   command_io

monitor_temperatures ==  IMPORTS :   sensor_name_io

monitor_temperatures ==  IMPORTS :   sensor_value_io

monitor_temperatures ==  IMPORTS :   system
```

```
monitor_temperatures == IMPORTS :   text_io


 GLOBAL MAPPING ===> monitor_temperatures :- IS MAPPED TO ===>
 h_monitor_temperatures


OBJECT VISIBILITY (HOOD objects)
================================

Object name          : h_monitor_temperatures
Object mode/type     : active

****

Internal object : (1) h_collection_of_sensors
Internal object : (2) h_recording_device
Internal object : (3) h_alarm
Internal object : (4) h_timer
Internal object : (5) h_timer

****

Operation : (1) h_disable_sensors        --> constrained by : no_constraint
Operation : (2) h_enable_sensors         --> constrained by : no_constraint
Operation : (3) h_status                 --> constrained by : aser
Operation : (4) h_limits                 --> constrained by : no_constraint

IMPACTS and PROPAGATION (HOOD objects)
======================================

h_monitor_temperatures  == USES ==  h_collection_of_sensors

h_monitor_temperatures == includes ==  h_recording_device

h_monitor_temperatures == defines ==  h_disable_sensors
```

Exemple 2

La requête :- WHAT-IF-Global(h_project, HOOD)

La réponse :-

```
OBJECT VISIBILITY (HOOD objects)
================================


Object name          : h_project
Object mode/type     : active

****

Internal object : (1) h_unit_information
Internal object : (2) h_data_base

****

Operation : (1) h_project_unit_name       --> constrained by : no_constraint
Operation : (2) h_project_sort_criteria   --> constrained by : aser_by_it

IMPACTS and PROPAGATION (HOOD objects)
======================================

h_project  == USES == h_unit_information
```

```
h_project == includes == h_unit_information

h_project == defines == h_project_unit_name


 GLOBAL MAPPING ===>  h_project :- IS MAPPED TO ===> project


OBJECT VISIBILITY (Ada objects)
===============================

Object name    : project
Object type    : package_spec
Object kind    : package_specification
Line No.       : 1

Starting point of (project) in the program is line-no :- 1
Ending point of   (project) in the program is line-no :- 32


Object name    : project
Object type    : package_body
Object kind    : package_body
Line No.       : 34

Starting point of (project) in the program is line-no :- 34
Ending point of   (project) in the program is line-no :- 50

IMPACTS and PROPAGATION (Ada objects)
=====================================

project ==  IMPORTS : absolute_io

project ==  IMPORTS : criteria_io

project ==  IMPORTS : inquiry_operations

project ==  IMPORTS : relative_io

project ==  IMPORTS : status_io

project ==  IMPORTS : text_io

project == CALLS :    print

project == CALLS :  sort
```

Exemple 1 montre la propagation d'impacts à partir du niveau Ada au niveau HOOD et exemple 2, à partir de HOOD à Ada. Ce passage d'un niveau à d'autres est assuré par la relation de correspondance **GLOBAL MAPPING**. La requête **WHAT-IF-Global** établit un lien entre un niveau avec un autre en utilisant la relation de correspondance **GLOBAL MAPPING** (cf. section 5.3.5), la table de correspondance (cf. figure 5.7) et le type d'objet donné dans la requête.

5.5.7 La requête WHAT-LIST

La requête **WHAT-LIST** quant à elle s'adresse à l'ensemble d'objets à un seul niveau. C'est une requête de service. L'activation de cette requête donne la liste de tous les objets d'un seul niveau. La syntaxe de WHAT-LIST est :

WHAT-LIST(niveau) où
niveau représente le niveau Ada ou le niveau HOOD.
Exemple

La requête :- WHAT-LIST(Ada)

La réponse :-

```
monitor_temperatures
command
sensor_name
sensor_state
sensor_value
command_io
sensor_name_io
sensor_value_io
alarm
post_fault_in_sensor
post_out_of_limits
on_sensor
collection_of_sensors
disable
sensor
enable
sensor
force_record
of_sensor
set_the_limits
for_sensor
low_limit
high_limit
timer
interrupt
light
fault_light
limit_check
...
```

Cette requête est activée automatiquement au niveau de l'interface utilisateur.

5.5.8 L'interface utilisateur

Dans cette sous-section, nous présentons l'interface utilisateur du prototype WHAT-IF. C'est donc pour l'utilisateur le moyen de communication avec le système et le moyen de présentation d'informations par le système.

L'interface utilisateur de WHAT-IF est *contrôlée par menu* sur le *mode fenêtre.*

Ce choix est basé sur le fait que :

1. la *fenêtre* indique aux utilisateurs que toutes les choses qui apparaissent sont liées par un lien commun. Dans le cas de ce prototype, il y a une partie en C, une autre partie en PROLOG, et ces deux parties sont liées par un langage de programmation interprété. Ce langage de programmation interprété TCL[26]est aussi interfacé à un environnement intégré de développement graphique XF.

26. Tool Common Language

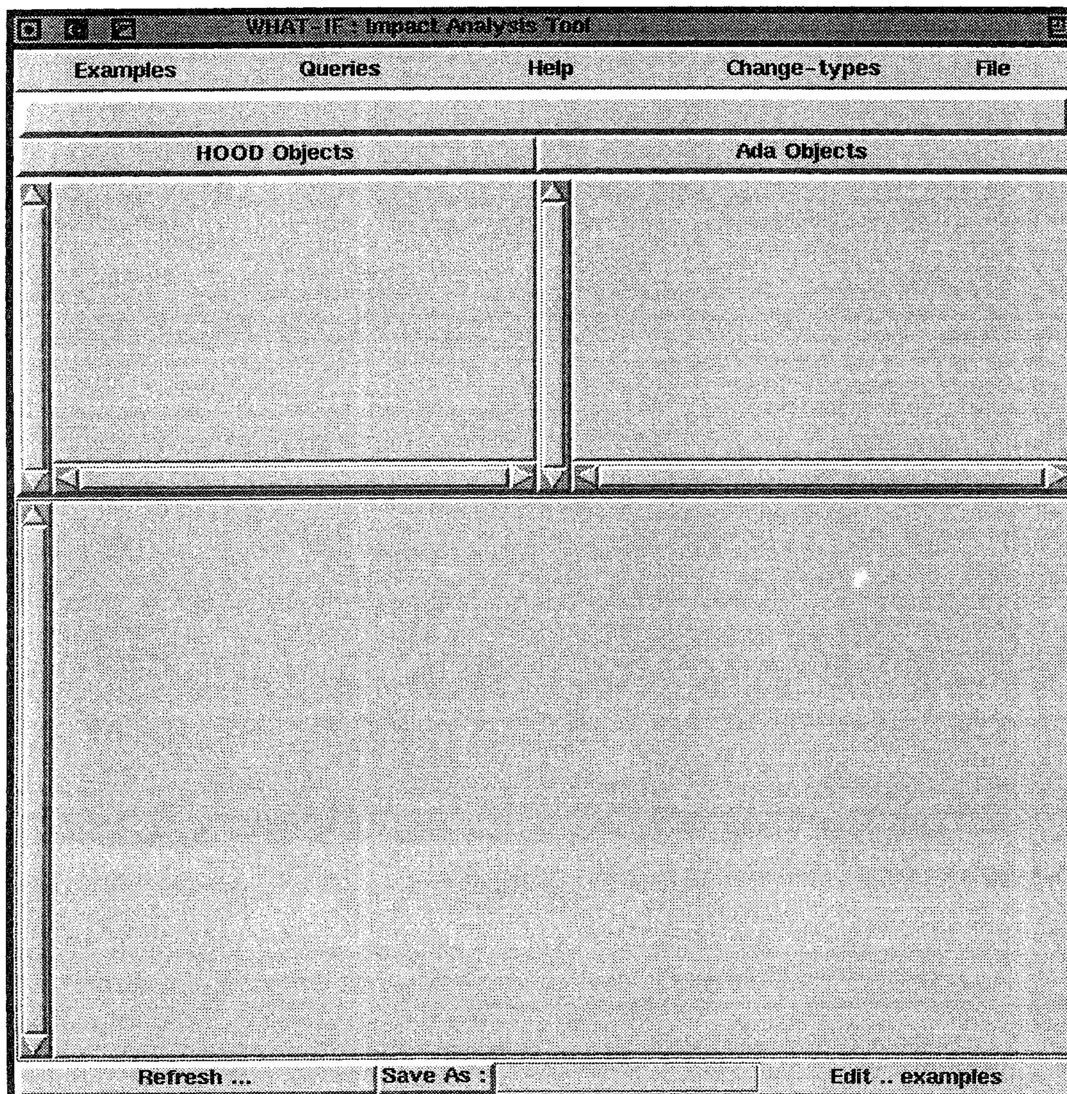


FIG. 5.9 - *L'interface utilisateur du prototype WHAT-IF*

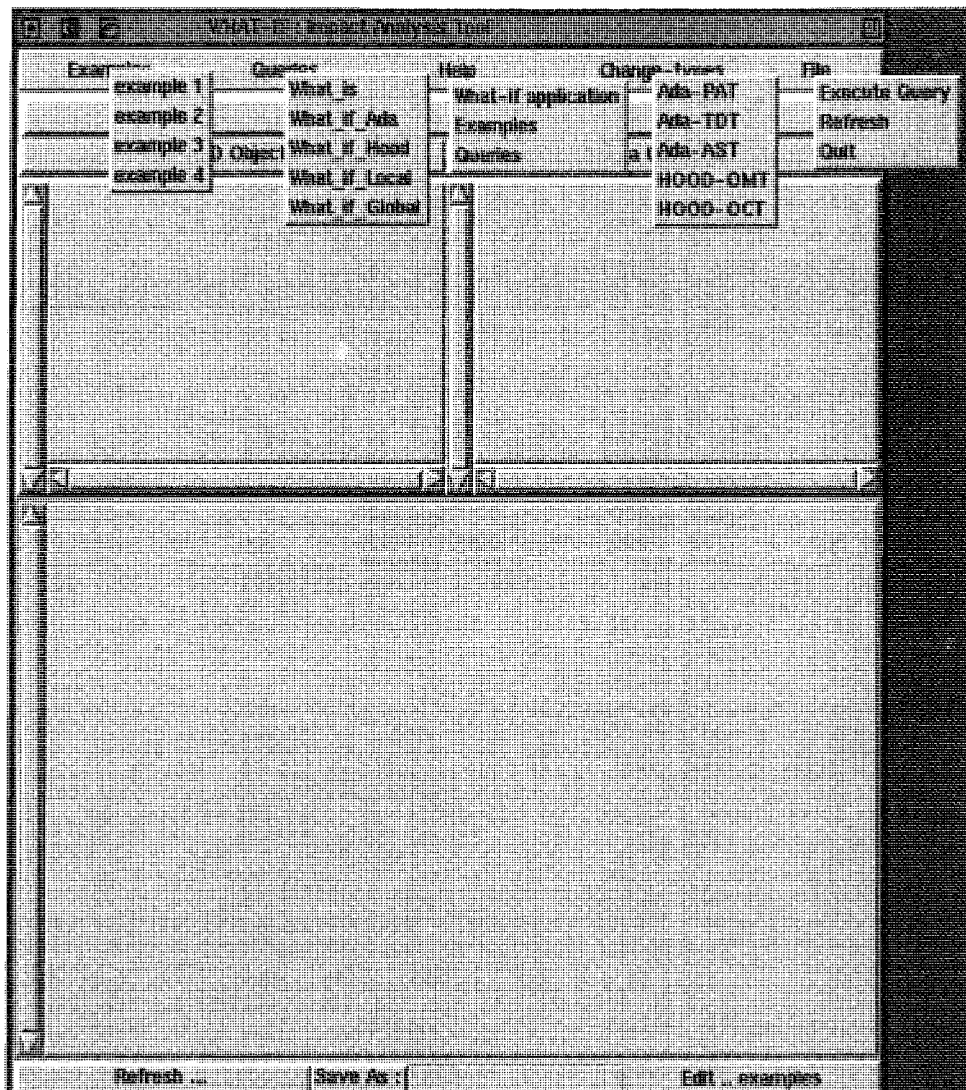


FIG. 5.10 - *Les différents menus*

2. le système de menu est une partie importante de l'interface utilisateur car :
   - l'utilisateur, même débutant peut facilement apprendre à utiliser le menu, et
   - les menus permettent au utilisateur de choisir parmi l'ensemble des actions à exécuter.

L'interface utilisateur comporte six différents type de menus (figure 5.10) :

- **le menu d'exemples** : ce menu permet de sélectionner un exemple à exécuter parmi les différents exemples,
- **le menu de requêtes** : ce menu contient une liste de toutes les requêtes définies pour le prototype,

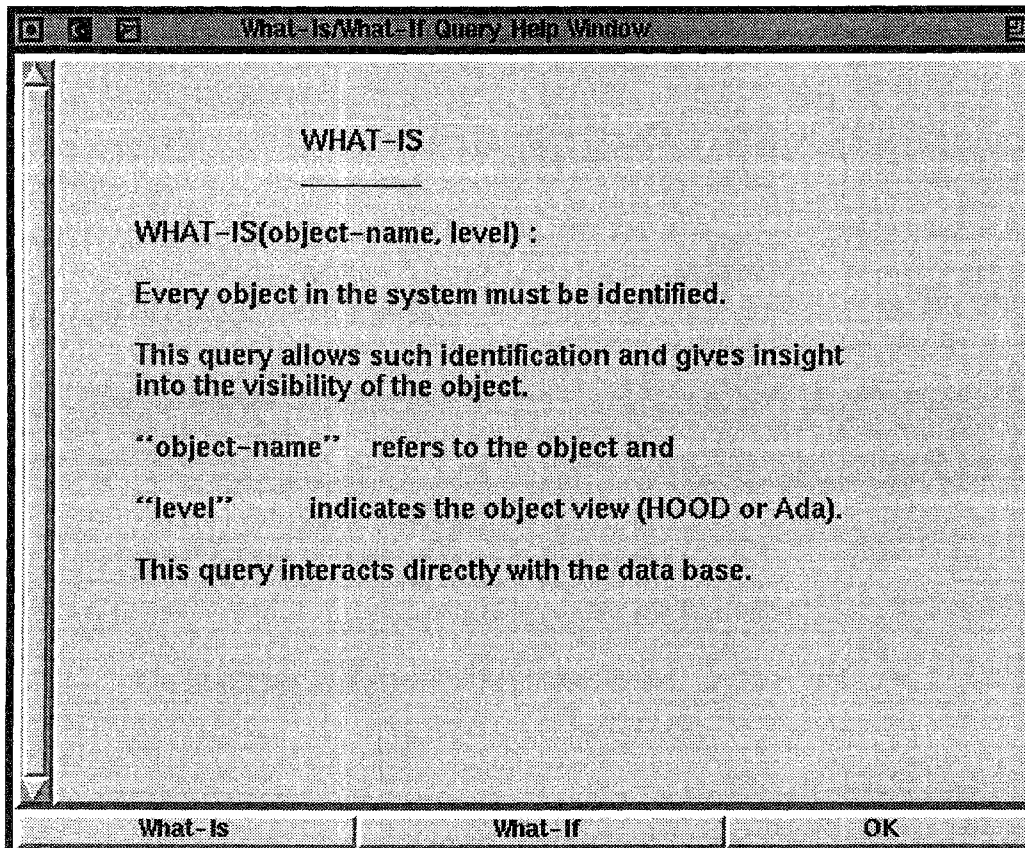


FIG. 5.11 - *L'information d'aide sur la requête WHAT-IS*

- **le menu d'aide aux utilisateurs**: en cliquant sur un choix, ce menu fait apparaître une fenêtre (cf. figure 5.11) qui donne une information sur le choix,
- **le menu de type de changements**: ce menu contient les choix pour les types de changement définis dans le prototype,
- **le menu d'exécution et de sortie du système**: ce menu permet de choisir entre l'exécution d'une requête, l'effacement de la fenêtre, ou la sortie du système, et
- **le menu d'édition des exemples**: ce menu permet d'éditer les exemples; il est basé actuellement sur l'éditeur *Emacs*.

L'implantation de l'interface utilisateur

L'interface utilisateur de WHAT-IF est implanté avec XF, un environnement intégré de développement graphique. XF utilise Tk, un ensemble de *widget* de type *Motif* et l'accès au Tk se fait par le Tcl. Tcl est un langage de programmation interprété [Ousterhout 93, Delmas 93].

Le choix de l'environnement XF est basé sur les points suivants:

- XF permet une construction *rapide* de l'interface utilisateur;
- XF permet un changement ultérieur avec beaucoup de flexibilité;
- XF autorise le développement coopératif;
- XF par le moyen de Tcl autorise l'appel de fonctions externes.

5.6 La conclusion

Le prototype du modèle WHAT-IF est basé sur deux sous-système de vues; à savoir, le sous-système de vue pour la conception et le sous-système de vue pour le programme. Nous avons choisi la méthode HOOD pour la conception et le langage Ada pour le programme (cf. section 5.1).

Le modèle WHAT-IF étant un modèle générique, le prototype WHAT-IF démontre la puissance de ce modèle. Alors que dans les recherches récentes [Gopal 89, Laski 92, Jackson 94], l'analyse d'impact de changement est limitée à un niveau donné voire localisée dans une partie de code, nous avons montré qu'avec la méthode d'analyse de dépendances et une base de connaissances, on peut:

- dépasser la limite des méthodes de simulation (cf. section 2.0),
- donner les informations plus détaillées sur l'effet d'un changement (cf. les exemples),

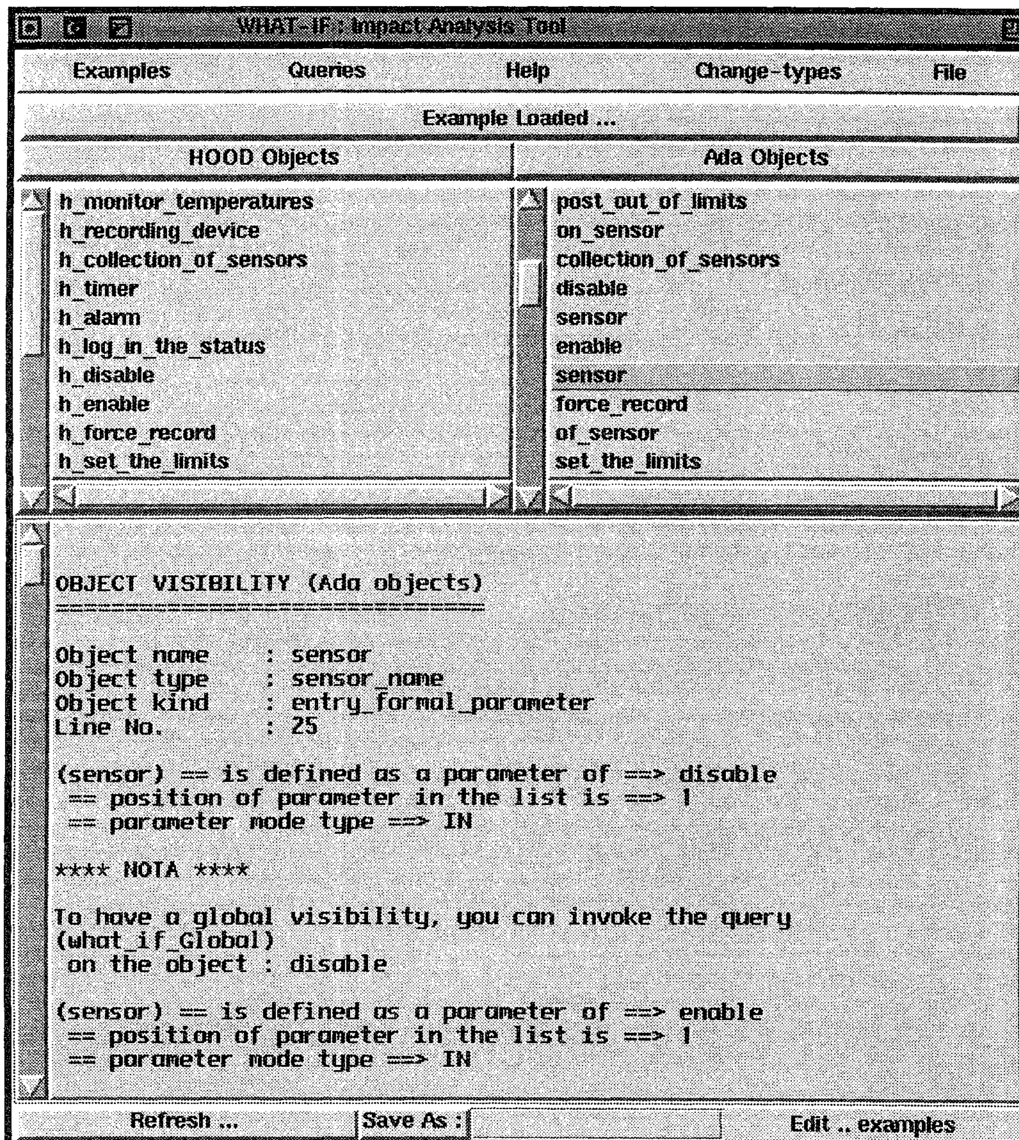


FIG. 5.12 - *Le résultat de la requête* **WHAT-IF-Ada(sensor, PAT)**

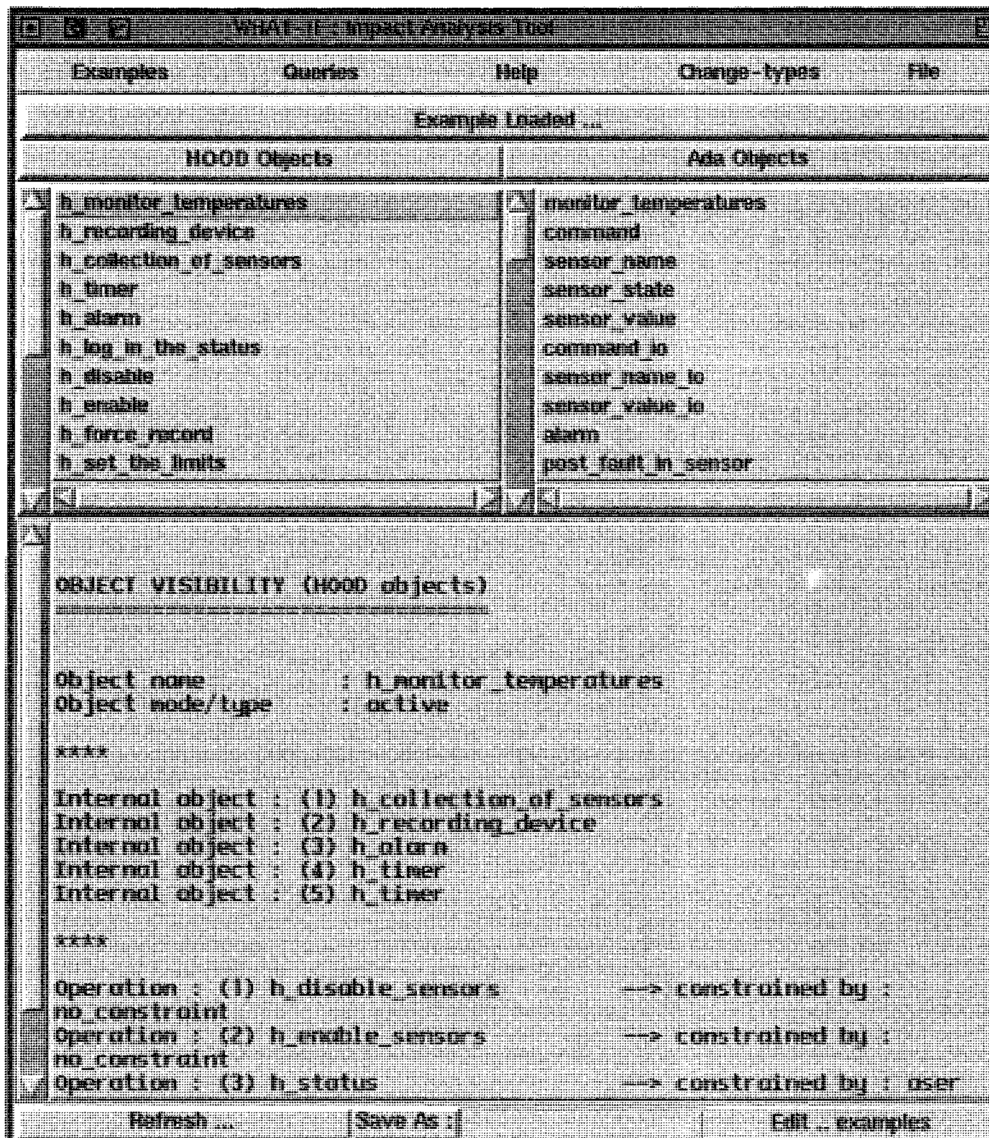


FIG. 5.13 - *Le résultat de la requête* **WHAT-IS(monitor_temperatures, HOOD)**

- propager les impacts de changements d'un niveau à d'autre,
- limiter le temps de codage du changement par la moyenne des requêtes prédéfinies et très facile à utiliser, et
- ajouter des relations par le moyen de règles au système ou utiliser les règles existantes d'une façon variée.

Chapitre 6

Conclusion

6.1 Bilan

Le procédé de changement de logiciel (cf. figure 2.3) consiste en :

- la compréhension du changement,
- la planification du changement,
- l'exécution du changement,
- la validation du logiciel modifié.

Dans le cadre de notre travail, nous nous sommes particulièrement intéressés à la compréhension du changement et de ses conséquences sur le système logiciel, à savoir, l'évaluation *a priori* de l'impact de changements réalisés sur les objets logiciels.

Parmi les types de conséquences possibles (psychologiques, économiques, et techniques) d'un changement, le procédé de changement décrit dans la problématique (cf. section 2.3 et figure 2.3) et repris ici n'adresse que les aspects techniques d'un changement.

Pour évaluer *a priori* l'impact d'un changement dans un logiciel, nous avons montré la nécessité de comprendre la signification du changement et d'avoir des connaissances précises sur les liens entre les objets logiciel. Analyser les impacts du changement suppose que l'on ait un modèle de celui-ci qui soit adapté aux types de facteur que l'on veut évaluer. Pour cette raison, nous avons étudié les caractéristiques souhaitables d'une analyse d'impact qui permettront ultérieurement de comprendre, de comparer, et d'évaluer des approches différentes et nous avons proposé un cadre générique (cf. section 2.5). Ce cadre générique comporte trois parties :

- une approche d'analyse d'impact : cette partie examine comment une approche peut être utilisée pour analyser l'impact d'un changement;
- la structure d'une approche d'analyse d'impact : cette partie présente la fonctionnalité d'une approche;

- la mesure d'efficacité d'une approche d'analyse d'impact : cette partie présente trois hypothèses basées sur un certain nombre de concepts que l'on peut utiliser pour mesurer l'efficacité d'une approche.

Nous avons également proposé une approche d'analyse d'impact (cf. chapitre 4) et pour cela nous avons :

- défini un modèle générique à base de connaissances sur la nature des liens entre les objets logiciels et les règles de propagation de changement.
- validé ce modèle dans un prototype basé sur deux étapes de cycle de vie, à savoir, la conception et le code. Nous avons choisi la méthode HOOD pour la conception et le langage de programmation Ada pour le code.

Une contribution importante de notre travail est la mise en évidence du fait que l'analyse et la propagation de l'impact du changement ne se limitent pas à une étape donnée mais couvre un ensemble d'étapes de cycle de vie de logiciel. Les outils d'analyse d'impact du changement existants sont basés sur les méthodes de simulation (les méthodes dites *algorithmiques*) et la plupart de ces méthodes ne s'adressent que le code et parfois même, seulement une partie du code.

Le système de vues proposé dans le modèle permet (cf. section 4.6) :

- une expression plus détaillé du système logiciel,
- une spécification des relations de dépendances entre les objets de grosse granularité et des objets de granularité fine,
- une manipulation efficace des connaissances des différents composants d'un système logiciel.

Nous pouvons comparer notre travail avec celui de Fillon et al [Fillon 94] à l'université de Kyoto au Japon. Fillon et al propose un modèle[27] pour tracer les dépendances entre l'analyse des besoins et la conception. Le DCFD[28] est utilisé pour l'analyse de besoins et HOOD pour la conception. L'approche proposée dans [Fillon 94] est de tracer et stocker les relations de dépendances en utilisant PCTE[29]. Le ODS de HOOD et le DCFD sont représentés en SDS[30]. Le USM[31] entre HOOD et DCFD est également représenté en SDS.

Ces deux travaux se ressemblent parce que le prototype du modèle WHAT-IF comme le USM adressent deux étapes de cycle de vie de logiciel. Mais, le modèle WHAT-IF propose un système de vues de logiciel plus détaillé que USM. Ce qui marque

27. Unified Model
28. Data Control Flow Diagrams
29. Portable Common Tool Environment
30. Schema Definition Set (Module de modèle de données en PCTE
31. Unified Semantic Model

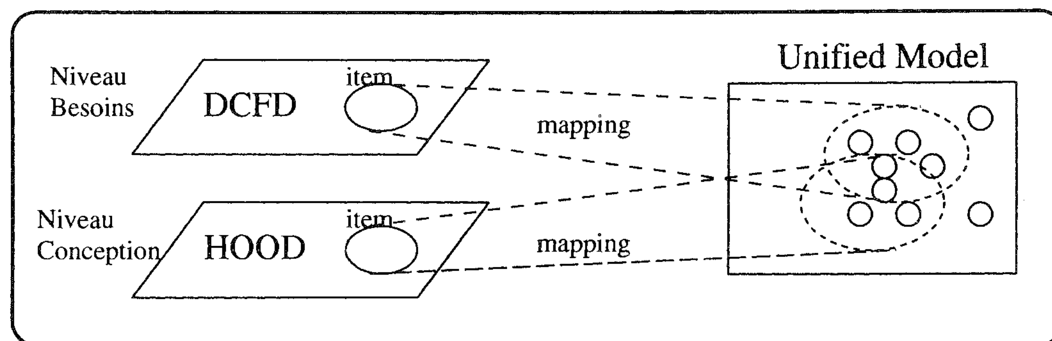


FIG. 6.1 - *USM Unified Model*

la différence est que USM utilise PCTE comme une *plate-forme* qui unifie[32] les deux niveaux d'abstraction (DCFD et HOOD) (cf. figure 6.1) et que WHAT-IF utilise le graphe de dépendances. Une autre différence est que USM est un modèle d'analyse d'impact *a posteriori* (c'est-à-dire, une analyse d'impact après la modification) et que WHAT-IF est un modèle *a priori*.

6.2 Perspectives

Une poursuite possible de ce travail consistera à :

1. implanter dans le prototype du modèle WHAT-IF les deux autres étapes définies dans le système de vues WHAT-IF (cf. section 4.6 et figure 4.3). Dans ce cas, on peut s'inspirer du travail de Fillon et al. [Fillon 94] en utilisant DCFD pour l'analyse de besoins et la méthode VDM[33] (ou le langage Z);

2. implanter dans le prototype l'impact dite "automatique" (i.e la propagation inconditionnelle). Celui-ci est réutilisé (réinjecté) automatiquement comme une nouvelle "modification" permettant de déduire des nouveaux impacts;

3. détailler les relations de dépendances définies pour les objets Ada en ajoutant des relations qui permettent de spécifier les rapports entre une tâche Ada (task) et les instructions *entry* et *accept* de Ada. On peut également faire la même chose pour le cas de *exception handling* en Ada;

4. étudier d'une façon approfondie les types de liens proposés dans le modèle pour ensuite trouver les propriétés intéressantes attachées à ces liens;

5. intégrer le "WHAT-IF Impact Analysis Tool" (cf. figures 5.9, 5.10, et 5.11) dans un environnement de développement de logiciel qui dispose à la fois d'une interface graphique puissante, des outils HOOD, et d'un éditeur de textes (cf. figure 6.2).

32. Unified Model
33. Vienna Development Method

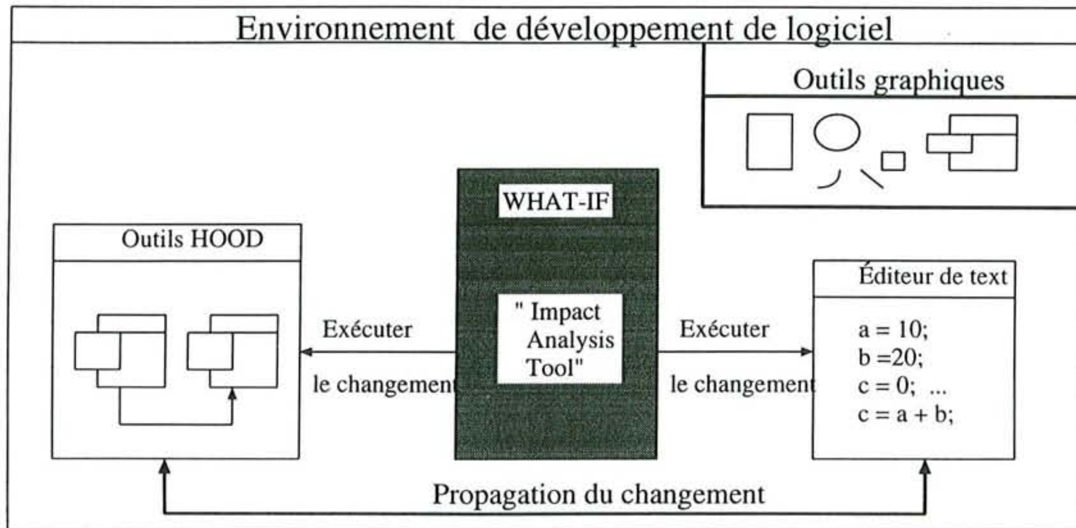


FIG. 6.2 - *L'intégration du modèle WHAT-IF dans un environnement de développement de logiciel*

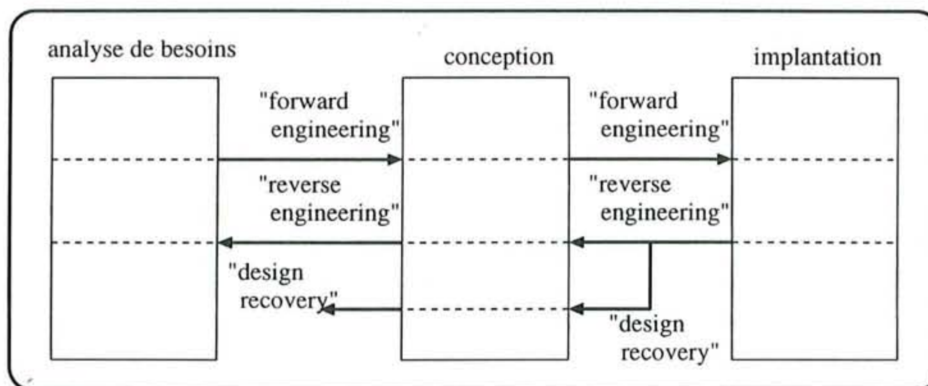


FIG. 6.3 - *Le "reverse engineering" et le procédé de transformation entre les différents niveaux d'abstraction*

Cette intégration permet en effet d'avoir un environnement de travail complet. Après avoir analysé *a priori* l'impact d'un changement, on peut décider d'exécuter le changement et prendre en compte les impacts du changement.

Cette intégration permettra aussi d'appliquer à *Impact Analysis Tool* les trois hypothèses proposée dans la problématique (cf. sections 2.5.4 et 2.5.5) pour mesurer l'efficacité de l'approche;

6. appliquer le modèle WHAT-IF au "reverse engineering" (cf. figure 6.3). Le but des outils de *reverse engineering* est de donner les informations sur la conception d'un programme. Ils fournissent des mécanismes d'abstractions qui permettent de comprendre un code [Chikofsky 90, Rugaber 90].

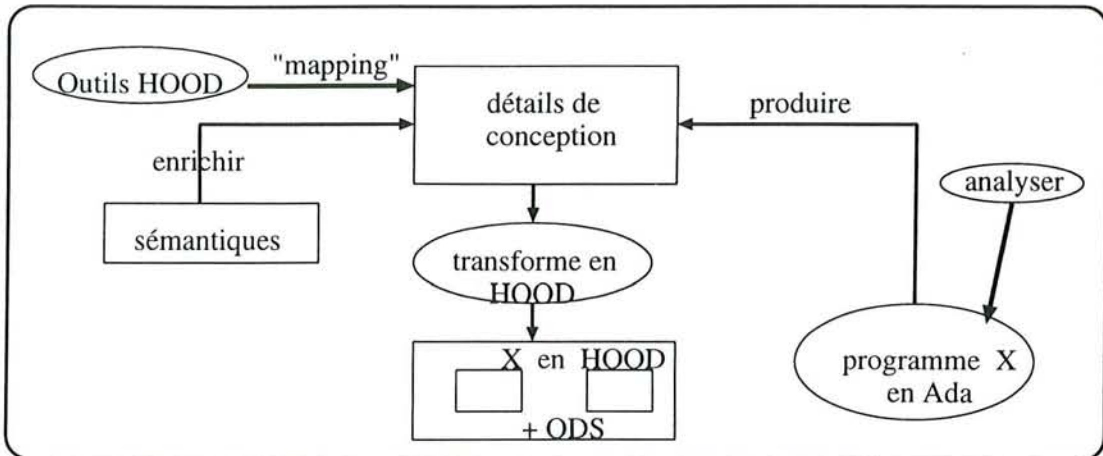


FIG. 6.4 - *Le "reverse engineering" de code Ada*

Il y a deux étapes nécessaires pour obtenir ces abstractions :

(a) le programme source doit être analysé pour obtenir tous les éléments dans le code qui contiennent les détails de la conceptions;

(b) ensuite, l'utilisateur doit ajouter à ces squelettes de la conception, la sémantique que l'on n'arrive pas à déduire automatiquement à partir du code.

Une première application peut être une analyse d'un code Ada. A partir du programme Ada, on peut essayer de récupérer tous les détails de la conception de ce programme et ensuite les enrichir et les représenter en HOOD (cf. figure 6.4).

Nous pensons que le prototype WHAT-IF démontre la puissance du modèle WHAT-IF. Le modèle WHAT-IF étant générique, il peut être appliqué au cas par cas. Nous avons également montré qu'avec ce modèle, on peut dépasser la limite des méthodes de simulation et donner les informations plus détaillées sur l'effet d'un changement.

Bibliographie

[**Ajila 91**] S. Ajila, *La fonction "What If" dans le cycle de vie du logiciel,* Mémoire de DEA, Université de Nancy I, Centre de Recherche en Informatique de Nancy (CRIN-CNRS), 4, Sept. 1991.

[**Ajila 92**] S. Ajila, H. Basson, et N. Boudjlida, *Software Process Assistance: a case study of the impact of object modification during software development,* in 1st African Conference on Research in Computer Science, Yaoundé, Cameroun, pp 73-84, vol. 1,Oct. 14-20, 1992.

[**Ajila 93a**] S. Ajila, J. C. Derniame, et H. Basson, **WHAT-IF** *: A Function to Estimate the Impacts of Potential Changes in a Software without First Putting them into Actual Practice,* Rapport Interne : CRIN 93-R-100, 1993, Nancy, France.

[**Ajila 93b**] S. Ajila, **Software Process Assistance** *: Management of objetcs modifications in an Integrated Software Engineering Environment,* Nigerian Journal of Science, vol. 28, 1993.

[**Ajila 94**] S. Ajila, H. Basson, et J.C. Derniame, *WHAT-IF: A function to estimate the impacts of potential changes in a software,* in Second International Conference on Software Quality Management (SQM 94), 26 - 28 July 1994, Edinburgh, Scotland.

[**Ajila 95**] S. Ajila, J.C. Derniame, et H. Basson, *A Logic-Based Approach to Impact Analysis of Objects Change,* in Third International Conference on the Practical Application of Prolog PAP'95, 3 - 7 April, 1995, Paris, France.

[**Alliot 94**] J.-M. Alliot et T Schiex, *Intelligence artificielle et Informatique théorique,* Collection Intelligence Artificielle, Cépaduès Editions, 1994.

[**Ambras 88**] J. Ambras et V. O'Day, *Microscope: a Knowledge-based programming environment,* IEEE Software, pp 50-58, May 1988.

[**Arango 86**] G. Arango, I.D. Baxter, P. Freeman, et C. Pidgeon, *TMM: Software Maintenance by Transformation,* IEE Software, pp 27-39, May 1986.

[Avellis 91] G. Avellis, A. Iacobbe, D. Palmisano, G. Semeraro et C. Tinelli, *An Analysis of Incremental Assistant Capabilities of a Software Evolution Expert System,* in Proc. of the Conference on Software Maintenance '91, CSM91, Sorrento, Italy, Oct. 1991.

[Avellis 92] G. Avellis, *CASE Support for Software Evolution : A Dependency Approach to Control the Change Process,* 5th Int. Workshop on CASE, Montréal, Québec, Canada, pp 62-73, July 6-10 1992.

[Basson 90] H. Basson et J. C. Derniame, *Towards an evolutive Kernel of Assessments on Ada Objects hosted in integrated Software Engineering Environment,* ACM Proceedings of the 7th Washington Ada Symposium, Washington, June 1990, pp 37-53.

[Basson 92] H. Basson, M.C. Haton, and J.C. Derniame, *Characteristics graph of software quality,* In International symposium on Computer and Information Sciences VII, pp.455-461, Nov. 1992.

[Basu 92] A. Basu et R. W. Blanning, *Enterprise Modeling Using Metagraphs,* DECISION SUPPORT SYSTEMS: Expriences and Expectations Elsevier Science Publishers B.V. (North-Holland) IFIP, 1992.

[Booch 88] G. Booch, *Software Engineering with Ada,* The Benjamin/Cummings Publishing Company, Inc, Menlo Park, CA 94025, 1988.

[Boehm 81] B. W. Boehm, *Software Engineering Economics,* Prentice Hall, 1981.

[Briand 93] L. C. Briand, S. Morasca, et V. R. Basili, *Measuring and Assessing Maintainability at the End of High Level Design,* in Proceedings of Conference on Software Maintenance, Sept. 27 - 30, 1993, Montréal, Quebec, Canada.

[Bustard 94] D. W. Bustard et A. C. Winstanley, *Making Changes to Formal Specifications : Requirements and an Example,* In IEEE Transaction on Software Engineering, Vol. 20. No. 8, Aug. 1994.

[Caldiera 91] G. Caldiera, *Domain Factory and Software Reusability,* in Proc. of Software Engineering Symposium, S.E.SY.91, Milan, May 1991.

[Canfora 92] G. Canfora, A. Cimitile, et Ugo de Carlini, *A Logic-Based Approach to Reverse Engineering Tools Production,* IEEE Transaction on Software Engineering, vol. 18, No. 12, Dec. 1992.

[Cerf 83] S. Cerf et S. Crespi-Reghizzi, *Relational Data Bases in the Design of program construction systems,* SIGPLAN Notices, Vol. 18, No. 11, Nov. 1983.

[Chapin 89] N. Chapin, *Changes in Change Control,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.246-253, Miami FL, 1989.

[Chen 93] T. Y. Chen et Y. Y. Cheung, *Dynamic Program Dicing,* In Proc. of IEEE Conference on Software Maintenance, CSM93, pp. 378 - 385, Montréal, Quebec, Canada, Sept. 27 - 30, 1993.

[Chen 76] P. Chen, *The Entity-Relationship model - Towards a unified view of Data,* ACM Transactions on Databases systems, Vol. 1, March 1976.

[Chen 86] Y. F. Chen et C. V. Ramamorthy, *C Information Abstractor,* Computer Science Division, U. C. Berkeley, 1986.

[Cheng 92] J. Cheng, *The Task Dependence Net in Ada Software Development,* ACM Ada Letters, Volume XII, Number 4, Jul/Aug, 1992.

[Chikofsky 90] E. J. Chikofsky and J. H. Cross II, *Reverse Engineering and Design Recovery : A Taxonomy,* In IEEE Software, Jan. 1990.

[Choi 94] J-D. Choi, R. Cytron, et J. Ferrante, *On the Efficient Engineering of Ambitious Program Analysis,* IEEE Transactions on Software Engineering, vol. 20, No. 2, Feb. 1994.

[Chu 93] W. C. Chu, *A Re-engineering Approach to Program Translation,* In Proc. of IEEE Conference on Software Maintenance, CSM93, pp. 42 - 50, Montréal, Quebec, Canada, Sept. 27 - 30, 1993.

[Clarke 81] L. A. Clarke et D. J. Richardson, *Symbolic Evaluation Methods for Program Analysis,* in Program Flow Analysis, Muchnick, S. S. et al, Editors, Prentice-Hall Inc., 1981, pp. 29-32.

[Cooper 89] S.D. Cooper et M. Munro, *Software Change Information for Maintenance Management,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.279-289, Miami FL, 1989.

[Curtis 92] B. Curtis, M. I. Kellner and J. Over, *Process Modeling,* Communications of the ACM, Vol. 35. No. 9, September 1992.

[Dankel II 89] D. D. Dankel II, *Final Report: The Intelligent Programming Environment,* SERC-TR-30-F, CIS Department,University of Florida, Gainesville, FL 32611,USA, Aug. 1989.

[Das 89] B.K. Das, *A knowledge-Based Approach to the Analysis of Code and Program Design Language (PDL),* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.290-298, Miami FL, 1989.

[Davis 88] C.G. Davis, P.J. Layzell, *Rules to Govern Change in JSD-Based Systems,* in Proc. of IEEE Conference on Software Maintenace, CSM88, Phoenix AZ, pp 34-40, 1988.

[DeBalbine 75] G. DeBalbine, *Better Manpower Utilization Using Automatic Restructuring,* AFIPS Proceedings of the 1975 National Computer Conference, AFIPS Press, 1975.

[Delmas 93] S. Delmas, **XF** : *Designe and Implementation of a Programming Environment for Interactive Construction of Graphical User Interfaces,* Users Manual, Berlin, 19 March, 1993.

[DeRemer 76] F. DeRemer et Hans H. Kron, *Programming-in-the-Large Versus Programming-in-the-small,* IEEE Trans. on Software Engineering, Vol.SE-2,No.2, pp 80-86, June 1976.

[Devanbu 90] P. T. Devanbu et al., *LASSIE-a Knowledge-based Software Information System,* in Proc. of 12th International Conference on Software Engineering, pp 249-261, March 1990.

[Donahoo 80] J. Donahoo et D. Swearingen, *Software Maintenance Technology,* Proceedings IEEE COMPSAC 80, October 1980, pp. 394-400.

[Dowson 87] M. Dowson, *Iteration in the software process : review of the 3rd International Software Process Workshop,* in Proc. 9th Int. Conf. on Software Eng., Monterey, CA pp 36-39, Apr. 1987.

[Escudié 94] A. Escudié, P. Y. Lamboley, J.P. Quille, F. Sedes, et J. F. Voidrot, *A Traceability-based Model for an Integrated Maintenance Environment,* In Proc. of the RIAO'94 Conference, New York, USA, Oct. 1994.

[Fay 85] D. Sandra Fay et G. Denise Holme, *Help! I Have to Update an Undocumented Program,* Proceedings of the conference on Software Maintenance, IEEE Computer Society Press, 1985, pp. 194-202.

[Ferrante 87] J. Ferrante, K. J. Ottenstein, et J. D. Warren, *The Program Dependence Graph and Its Use in Optimization,* ACM Trans. on Programming Languages and Systems, Vol. 9, No. 3, pp 319-349, July 1987.

[Fillon 94] P. Fillon, N. Mitsuda, A. Sawada, T. Ajisaka, et Y. Matsumoto, *A Facility to Trace Dependencies Betweew PCTE Objects for Software Maintenance,* In Proc. of the PCTE '94 Conference, California, Dec. 1994.

[Freedman 82] L. Daniel Freedman et M. Gerald Weinberg, *Maintenance Reviews,* In Techniques of Program and System Maintenance, Winthrop Publishers Inc., Cambridge, Mass., 1982.

[Gopal 89] R. Gopal et S.R. Schach, *Using Automatic Program Decomposition Techniques in Software Maintenance,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.132-141, Miami FL, 1989.

[Gottler 90] H. Gottler, J. Gunther et G. Nieskens, *Use Graph grammars to design CAD-systems,* Graph Grammars and Their Application to Computer Science, 4th International Workshop Proceedings, pp 396-410.

[Gustafson 93] D. A. Gustafson, J. T. Tan, et P. Weaver, *Software Measure Specification,* In ACM SIGSOFT'93, USA, 1993.

[Harandi 90] M. T. Harandi et J. Q. Ning, *Knowledge-Based Program Analysis,* IEEE Software, January 1990.

[Haton 91] J. P. Haton, *Les systèmes à base de connaissances,* Intelligence Naturelle et Intelligence Artificielle, Symposium de l'association de Psychologie Scientifique de Langue Française, Rome , 1991, Presse Universitaires de France, pp 19-42.

[Heitz 92] M. Heitz, *Towards more formal developments through integration of behavior expression notations and methods within HOOD developments,* In 5th Interntional Conference on Software Engineering & Its Applications, Toulouse 92, Dec. 7-11, 1992.

[Hewett 94] R. Hewett et M. Hewett, *A Knowledge-Based Framework for Automated Software Synthesis Control,* in Automated Software Engineering: The International Journal of Automated Reasoning and Artificial Intelligence in Software Engineering, Vol. 1, No. 3/4, September 1994.

[Holbrook 87] H.B. Holbrook et S. M. Thebaut, *A Survey of Software Maintenance Tools That Enhance Program Understanding,* SERC-TR-9-F, CIS Department,University of Florida, Gainesville, FL 32611,USA, Sept. 1987.

[Howden 81] E. William Howden, *A Survey of Dynamic Analysis Methods,* in Tutorial: Software Testing & Validation Techniques, IEEE Computer Society Press, Los Alamitos, CA., 1981.

[Ichbiah 86] J. D. Ichbiah, J. G.P. Barnes, Robert J. Firth, et Mike Woodger, *Rationale for the Design of the* **Ada** *Programming Language,* 1986.

[Jackel 86] M. Jackel, *Graph - Theoretic Concepts in Computer Science,* In Lecture Notes in Computer Science, 246, New York, Springer-Verlag, 1986.

[Jackson 94] D. Jackson et E. J. Rollins, *A New Model of Program Dependences for Reverse Engineering,* In ACM SIGSOFT'94, pp 2 - 10, New Orleans LA USA, Dec. 1994.

[Kaiser 88] G. E. Kaiser et S. S. Popvitch, *Intelligent Assistance for Software Development and Maintenance,* IEEE Software 5, no 3: pp 40-49. 1988.

[Kamklar 93] . Kamkar, P. Fritzson, et N. Shahmehri, *Interprocedural Dynamic Slicing Applied to Interprocedural Data Flow Testing,* In Proc. of IEEE Conference on Software Maintenance, CSM93, pp. 386 - 395, Montréal, Quebec, Canada, Sept. 27 - 30, 1993.

[Karimi 88] J. Karimi et B. R. Konsynsky, *An Automated Software Design Assistant,* IEEE Transaction on Software Engineering Vol. 14, No. 2 Feb. 1988.

[Katalagarianos 95] P. Katalagarianos et Y. Vassiliou, *On the Reuse of Software : A Case-Based Approach Employing a Repository,* in Automated Software Engineering : The International Journal of Automated Reasoning and Artificial Intelligence in Software Engineering, Vol. 2, No. 1, March 1995.

[Kozaczynski 94] W. Kozaczynski et J. Q. Ning, *Automated Program Understanding by Concept Recognition,* in Automated Software Engineering : The International Journal of Automated Reasoning and Artificial Intelligence in Software Engineering, Vol. 1, No.1, March 1994.

[Lai 91] M. Lai, *Conception orienté objet : Pratique de la méthode HOOD,* Paris, DUNOD, 1991.

[Lanubile 93] F. Lanubile et G. Visaggio, *Function Recovery based on Program Slicing,* In Proc. of IEEE Conference on Software Maintenance, CSM93, pp. 396 - 404, Montréal, Quebec, Canada, Sept. 27 - 30, 1993.

[Laski 92] J. Laski et W. Szermer, *Identification of Program Modifications and its Application in Software Maintenance,* in Proceedings of Conference on Software Maintenance, Nov. 9-12, 1992, Orlando, florida.

[Legout 92] C. Legout, *Electre, Un langage cible pour La methode HOOD dans le cadre de l'atelier de génie logiciel CONCERTO,* In 5th Interntional Conference on Software Engineering and its Applications, Toulouse 92, Dec. 7-11, 1992.

[Lehman 85] M. M. Lehman et L. A. Belady, *Program evolution : processes of software change,* APIC Studies in Data Processing No. 27, 1985, Academic Press.

[Lieberherr 89] K.J. Lieberherr et I.M. Holland, *Tools for Preventive Software Maintenance,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.2-13, Miami FL, 1989.

[Lientz 80] B. P. Lientz et E.B. Swanson, *Software maintenance management: a study of the maintenance of computer application software in 487 DP organizations,* Addison-Wesley, 1980.

[Linos 93] P. Linos, Ph. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, et Ph. Tulula, *CARE: An Environment for Understanding and Reengineering C Programs,* In Proc. of IEEE Conference on Software Maintenance, CSM93, pp. 130 - 139, Montréal, Quebec, Canada, Sept. 27 - 30, 1993.

[Linton 83] M. Linton et M. Powell, *OMEGA,* Computer Science Division, U. C. Berkeley, 1983.

[Livadas 92] P. E. Livadas et P. K. Roy, *Program Dependence Analysis,* in Proceedings of Conference on Software Maintenance, Nov. 9-12, 1992, Orlando, florida.

[Lejter 92] M. Lejter, S. Meyers, et S. P. Reiss, *Support for Maintaining Object-Oriented Programs,* In IEEE Transaction on Software Engineering, Vol. 18, No. 12, Dec. 1992.

[Loyall 93] J. P. Loyall et S. A. Mathisen, *Using Dependence Analysis to Support the Software Maintenance Process,* in Proceedings of Conference on Software Maintenance, Sept. 27 - 30, 1993, Montréal, Quebec, Canada.

[Mackworth 77] A. K. Mackworth, *Consistency in Networks of Relations,* Artificial Intelligence, 8 (1977), pp 99-118.

[MacLennan 83] B. J. MacLennan, *Overview of Relational Programming,* SIGPLAN Notices, Vol. 18, No. 3, March, 1983.

[Madhavji 92] N. H. Madhavji, *Environment Evolution : The Prism Model of Changes,* IEEE Transactions on Software Engineering, Vol. 18, No. 5, pp 380-392 May 1992.

[Meyer 85] B. Meyer, *The Software Knowledge Base,* In Proceedings of the Eighth International Conference on Software Engineering, London, UK, Aug. 1985.

[Myers 79] J. Glenford Myers, *The art of Software Testing,* John Wiley and Sons, Inc., New York, 1979.

[Ntafos 84] C. Simeon Ntafos, *On Required Element Testing,* IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, pp. 795-803.

[Ousterhout 93] J. K. Ousterhout, *An Introduction to Tcl and Tk,* Computer science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720, Dec., 1993.

[Ott 92] L. M. Ott et J. M. Bieman, *Effects of Software Changes on Module Cohesion,* In Proc. of IEEE Conference on Software Maintenance, CSM93, pp. 345 - 353, Orlando, Florida, USA, Novv. 9 - 12, 1992.

[Parikh 86] G. Parikh, *Restructuring Your Cobol Programs,* Computerworld Focus, Vol. 20, No. 7A, 39 - 42, Feb. 19, 1986.

[Parnas 79] D. L. Parnas, *Designing Software for Ease of Extension and Contraction,* IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, pp. 128-138, March 1979.

[Penedo 85] M. H. Penedo et E. Stuckle, *A Project Master Database for Software Engineering Environments,* In Proceedings of the Eighth International Conference on Software Engineering, London, UK, Aug. 1985, pp 150-157.

[Penedo 89] M. H. Penedo et E. Stuckle, *Integrated Project Master Database (PMDB),* IR $ D Final Report - Technical report TRW-84-SS-22-Reprinted 1989, Arcadia-TRW-89-008, TRW Defense System Group, CA.

[Plaice 93] J. Plaice et W. W. Wadge, *A Unix Tool for Managing Reusable Software Components,* Software-Practice and Experience, vol. 23(9), 933-948, Sept. 1993.

[Perry 88] D. E. Perry, *Problems of scale and process models,* in Proc. 4th Int. Software Process Workshop, UK, May 1988, ACM SIGSOFT, Vol. 14, pp. 126-128.

[Queille 94] J-P. Queille, J-F. Voidrot, N. Wilde et M. Munro, *The Impact analysis Task in Software Maintenance : A Model and a Case Study,* in International Conference for Software Maintenance ICSM 94, 1994.

[Queille 94] J-P. Queille, A. Richermo, J-F. Voidrot, et C. Chrisment, *A Generic Model for the Exploitation of Traceability Links between Documents of the Software Development Cycle,* In Proc. of IEEE

Conference on Software Maintenance, CSM93, Montréal, Quebec, Canada, Sept. 27 - 30, 1993.

[Ramamoorthy 86] C.V. Ramamoorthy, V. Garg et A. Prakash, *Programming in the Large,* IEEE Trans. on Software Engineering, Vol.SE-12, No. 7, pp 769-783, July 1986.

[Redwine 88] S. T. Redwine, *Constructing enactable models (or process models for process models): session summary,* in Proc. 4th Int. Software Process Workshop, UK, May 1988, ACM SIGSOFT, Vol. 14, pp. 17-22.

[Reynolds 91] R. G. Reynolds, *Automatic, Rapid generation of design prototypes from Logic specifications,* International Jounal of Software Engineering and Knowledge Engineering Vol. 1, No. 4 pp 331-350, 1991.

[Rich 92] C. Rich et R. C. Waters, *Knowledge Intensive Software Engineering Tools,* IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 5, October 1992.

[Roman 86] D. Roman, *ClassifyingMaintenance Tools,* Computers Decisions, June 1986.

[Rothermel 93] G. Rothermel et M. J. Harrold, *A Safe, Efficient Algorithm for Regression Test Selection,* In Proc. of IEEE Conference on Software Maintenance, CSM93, pp. 358 - 367, Montréal, Quebec, Canada, Sept. 27 - 30, 1993.

[Rugaber 90] S. Rugaber, Stephen B. Ornburn, et Richard J. LeBlanc(Jr), *Recognizing Design Decisions in Programs,* In IEEE Software, Jan. 1990.

[Rugaber 93] S. Rugaber et S. Doddapaneni, *The Transition of Application Programs From COBOL to a Fourth Generation Language,* In Proc. of IEEE Conference on Software Maintenance, CSM93, pp. 60 - 70, Montréal, Quebec, Canada, Sept. 27 - 30, 1993.

[Ruven 83] B. Ruven, *Towards a Theory of the Comprenhension of Computer Programs,* International Journal of Man-Machine Studies, Vol. 18, 1983, pp. 543-554.

[Ryder 89] B.G. Ryder, *ISMM: The Incremental Software Maintenance Manager,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.142-164, Miami FL, 1989.

[Selfridge 94] P. G. Selfridge, *Report on the First Working Conference on Reverse Engineerin,* in Automated Software Engineering : The International Journal of Automated Reasoning and Artificial Intelligence in Software Engineering, Vol. 1, No.1, March 1994.

[Software 86] *Software Aids and Tools Survey,* Office of Software Development and Information Technology, U.S. Government Printing Office, 1986 GPO 022-002-00106-2.

[Sok 80] S. SOK (Chak), *Evolutivité du Logiciel,* Doctrat de 3ème cycle Informatique, Université de Nancy I, sept. 1980.

[Stan 85] L. Stan et S. Elliot, *Strategies for Documenting Delocalized Plans,* Proceedings of the Conference on Software Maintenance, IEEE Computer Society Press, 1985, pp. 144-151.

[Stonebraker 92] M. Stonebraker, *The Intergration of Rule Systems and Database Systems,* In IEEE Transaction on Knowledge and Data Engineering, Vol. 4, No. 5, Oct. 1992.

[Stucki 77] G. Leon Stucki, *New Directions in Automated Tools for Improving Software Quality,* in Current Trends in Programming Methodology, Vol. II: Program Validation, Raymond Yeh, Editor, Prentice-Hall Inc., 1977, pp. 80-111.

[Taha 87] A. M. Taha et S. M. Thebaut, *Program Change Analysis Using Incremental Data Flow Techniques,* SERC-TR-14-F, CIS Department,University of Florida, Gainesville, FL 32611,USA, Dec. 1987.

[Thebaut 86] S. Thebaut et N. Wilde, *Program Change Analysis: Improving the Reliability of Modified Programms,* SERC-TR-1-F, CIS Department,University of Florida, Gainesville, FL 32611,USA, July 1986.

[Vanek 89] L.I. Vanek et M.N. Culp, *Static Analysis of Program Source Code using EDSA,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.192-199, Miami FL, 1989.

[Ward 89] M. Ward, F.W. Calliss, et M. Munro, *The Mainteainer's Assistant,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.307-317, Miami FL, 1989.

[Weber 93] M. Weber, *Elaboration Order Issues in Ada 9X,* ACM Ada Letters, Volume XIII, Number 1, Jan/Feb, 1993.

[Westfechtel 92] B. Westfechtel, *A Graph-Based Approach to the Construction of Tools for the Life Cycle Integration between Software Documents,* 5th Int. Workshop on CASE, Montréal, Québec, Canada, July 6-10 1992.

[Wild 89] C. Wild et K. Maly, *Decision-Based Software Development: Design and Maintenance,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.297-306, Miami FL, 1989.

[Wilde 86] N. Wilde, *Software Maintenance Management,* SERC-TR-2-F, CIS Department,University of Florida, Gainesville, FL 32611,USA, Oct. 1986.

[Wilde 92] N. Wilde et A. Chapman, *Describing Object Oriented Software: What Maintainers Need to Know,* SERC Purdue University and University of Florida Technical Report No. SERC-TR-54-F, USA, Feb. 28, 1992.

[Wilde 88] N. Wilde et R. Huitt, *A Data-Base Program Representation for Software Maintenance Tools,* SERC-TR-25-F, CIS Department,University of Florida, Gainesville, FL 32611,USA, Dec. 1988.

[Wilde 89] N. Wilde, R. Huitt, et S. Huitt, *Dependency Analysis Tools: Reusable Components for Software Maintenance,* In Proc. of IEEE Conference on Software Maintenance, CSM89, pp.126-131, Miami FL, 1989.

[Wilde 87] N. W. Wilde et Steve Thebaut, *The Maintenance Assistant: Work in Progress,* SERC-TR-10-F, CIS Department,University of Florida, Gainesville, FL 32611,USA, Sept. 1987.

[Williams 88] L. G. Williams, *Emerging issues : session summary,* in Proc. 4th Int. Software Process Workshop, UK, May 1988, ACM SIGSOFT, Vol. 14, pp. 29-31.

[Yau 80] S. S. Yau et J.S. Collofello, *Some Stability Measures for Software Maintenance,* IEE Trans. on Software Engineering, Vol.SE-6, No. 6, pp 545-552, Nov. 1980.

[Yau 81] S. S. Yau et P. C. Grabow, *A model for representing programs using hierarchical graphs,* IEEE Trans. Software Eng., vol SE 7, pp. 556 - 574, Nov 1981.

[Yau 85] S. S. Yau, J. P. Tsai et R. A. Nicholl, *Knowledge representation of software life-cycle information using first-order logic,* Proc. COMPSAC 85, pp 268-277, Oct. 1985.

[Yau 88a] S. S. Yau, Robin A. Nicholl, J. P. Tsai, et Sying-Syang Liu, *An Integrated Life-Cycle Model for Software Maintenance,* IEEE Trans. on Software Engineering, Vol. 14 No. 8 Aug 1988.

[Yau 88b] S. S. Yau et Sying-Syang Liu, *Some Approaches to Logical Ribble Effect Analysis* SERC-TR-24-F, CIS Department,University of Florida, Gainesville, FL 32611,USA, Sept. 1988.

Annexe A

HOOD : Une méthode de conception hiérarchisée orientée objets

A.1 L'introduction

HOOD (Hierarchical Object Oriented Design) est une méthode de conception descendante orientée objets, utilisant une notation proche de Ada, et particulièrement adaptée aux gros logiciels temps réel, scientifiques et techniques nécessitant un développement réparti. Issue de l'expérience acquise par la pratique industrielle de méthodes de Conception Orientées Objets (COO) et des Machines Abstraites (MA), HOOD intègre les concepts d'objets, de structurations par hiérarchies et d'expression de contrôle, facilitant ainsi le développement d'applications de grandes tailles.

HOOD a été retenue par l'AGENCE SPATIALE EUROPEENNE comme la méthode commune des industriels européens collaborant pour la réalisation du projet de station spatiale COLUMBUS. HOOD, définie en 1986 par CISI-INGENIERIE en collaboration avec MATRA et la société danoise CRI, a été sélectionnée parmi 15 autres propositions concurrentes.

A.2 La conception orientée objet et HOOD

A.2.1 La conception d'objet

Par opposition aux méthodes classiques qui privilégient la décomposition d'un système soit par les fonctions, soit par les données, la conception orientée objets se présente comme une approche de synthèse basée sur les concepts d'objet, de modèle abstrait, et qui regroupe à la fois données et opérations sur ces données. Ce concept permet de satisfaire les principes fondamentaux du génie logiciel, visant à l'obtention

[Lai 92, Legout 92]:

- **d'une forte cohésion logique**
  Un objet doit avoir une forte cohésion interne, c'est-à-dire que les données et opérations qu'il contient doivent être très fortement liées et doivent représenter un sous-ensemble logiquement très significatif dans la solution.

- **d'un faible couplage**
  Les différents objets composant un système doivent être très faiblement liés les un aux autres, c'est-à-dire que les interfaces entre ces objets doivent être les plus simples possible et ne contenu que ce qui est strictement nécessaire.

A.2.2 Le modèle objet HOOD

HOOD étend le concept d'objet de COO en définissant et fournissant des critères de structuration statique et dynamique des systèmes basée sur:

- **des principes d'abstraction, d'information cachée et d'encapsulation** (les objets sont définis par leurs propriétés externes uniquement, leur structure et composition internes étant cachés aux utilisateurs).

- **des principes de hiérarchie**:
  • les objets peuvent être décomposés en d'autres, et un système est représenté par un **objet parent** composé **d'objets enfants.**
  • les objets peuvent utiliser les opérations offertes par d'autres objets, et un système peut être représenté par des **objets seniors utilisant des objets juniors dans une hiérarchie.**

- **des principes d'expression du contrôle**: Les opérations fournies par les objets sont activées par des flots de contrôle. A un instant donné, il peut y avoir plusieurs flots de contrôle opérant simultanément à l'intérieur d'un objet. Ces flots apparaissent dans les systèmes temps réel lors de la prise en compte d'événements asynchrones (interruptions), ou par l'implantation de parallélisme sur une machine séquentielle.

A.3 Caractéristiques de HOOD

HOOD fournit des mécanismes évitant les écueils de COO dans le domaine de gros logiciels: la formalisation du modèle objet a conduit à un processus de conception hiérarchisé facilitant la mise en place de procédures de validation et vérification, et à la définition d'une stratégie de conception compatible à la fois avec les modèles mise en œuvre par les techniques d'analyse et avec ceux utilisés en conception de systèmes et validés par l'expérience acquise.

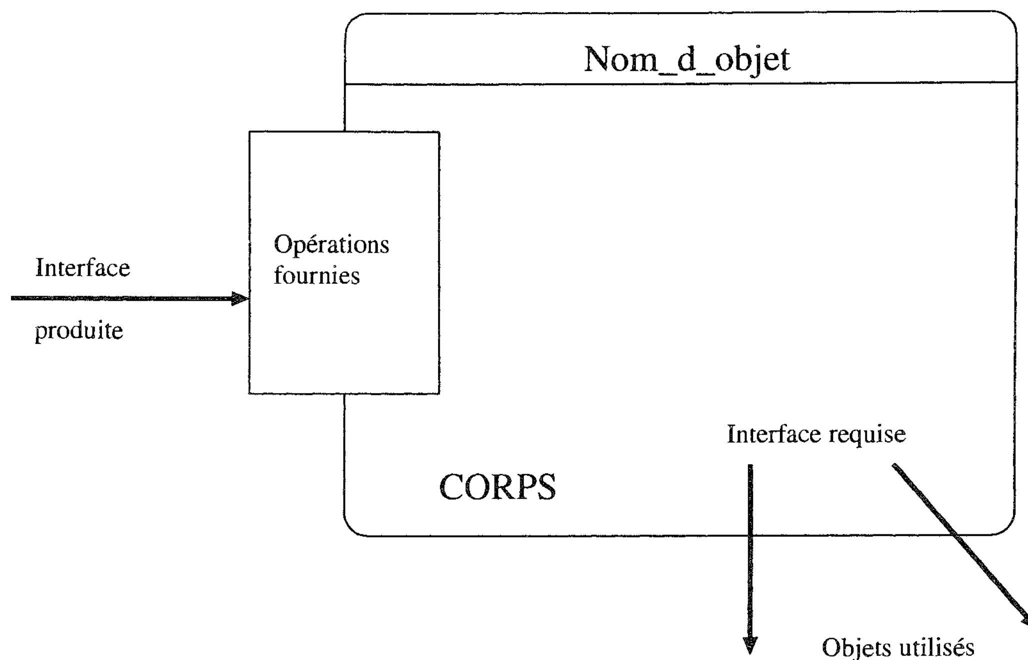


FIG. A.1 - *La représentation d'un objet HOOD*

A.3.1 La formalisation du concept d'objet

Elle s'appuie sur deux notations (graphiques et textuelles) qui s'utilisent de manière complémentaire et permettent de capturer à la fois les propriétés statiques et dynamiques d'un objet :

(a) **propriétés statiques** : un objet a une partie visible, l'interface fournie, et une partie cachée, le corps, qui n'est pas accessible directement par les autres objets. Un objet est connu du monde extérieur par son nom. La représentation graphique d'un objet est la suivante :

- L'interface HOOD définit les opérations et ressources fournies et requises par un objet vis-à-vis des autres objets, ainsi que les données, paramètres formels et types associés;
- Le corps est l'implantation des opérations et ressources spécifiées dans la partie interface. Il utilise des opérations et données internes, ainsi que des objets internes. Ces composants internes du corps implantent l'état interne de l'objet;

(b) **propriétés dynamiques** : l'exécution d'une opération, vue par un utilisateur d'un objet, peut s'effectuer de deux manières :

- séquentiellement : le contrôle est transféré de l'utilisateur vers l'objet utilisé et l'opération est effectuée immédiatement selon la structure de contrôle de

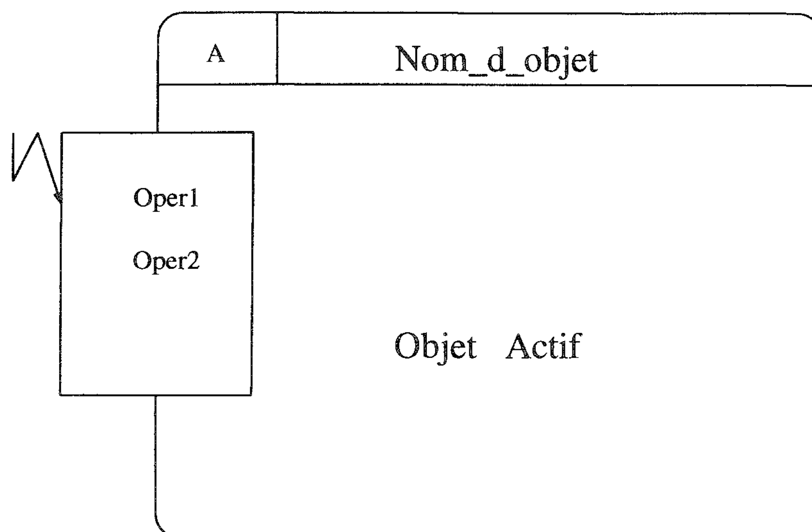


FIG. A.2 - *La représentation d'un objet actif*

l'opération associée (OPCS pour Operation Control Structure). Le contrôle est rendu à l'utilisateur en fin d'exécution de l'opération. Ce comportement définit un objet passif vis-à-vis des flots de contrôle qui le traversent.

- concurremment : le contrôle est transféré selon un protocole dépendant à la fois de l'état interne de l'objet et/ou de l'interaction des flots de contrôle entrants dans l'objet. Ce protocole est décrit dans la structure de contrôle de l'objet (OBCS pour Object Control Structure) en utilisant la notation et la sémantique Ada du rendez-vous. L'association d'un OBCS à un objet définit un objet actif.

Dans les deux cas, si une opération ne peut être effectuée avec succès, une exception peut être levée par l'objet utilisé et transmise à l'utilisateur.

A.3.2 La description des relations inter objets

La relation USE

Un objet utilise un autre objet si le premier requiert une ou plusieurs opérations fournies par le second. La relation "use" définit une hiérarchie senior-junior entre les objets. Afin de contribuer à des conceptions de meilleur qualité, HOOD renforce les principes de faible couplage et forte cohésion en contraignant la relation use :

- de façon à ce que le graphe d'utilisation ne soit pas cyclique, et

- de façon à ce que le graphe ait la plus faible complexité possible : les objets doivent avoir un **faible "fan-out"** (les objets doivent utiliser le moins possible les ressources fournies par les autres objets = faible couplage) et un **fort "fan-in"** (les objets doivent être utilisés autant que possible = forte cohésion).

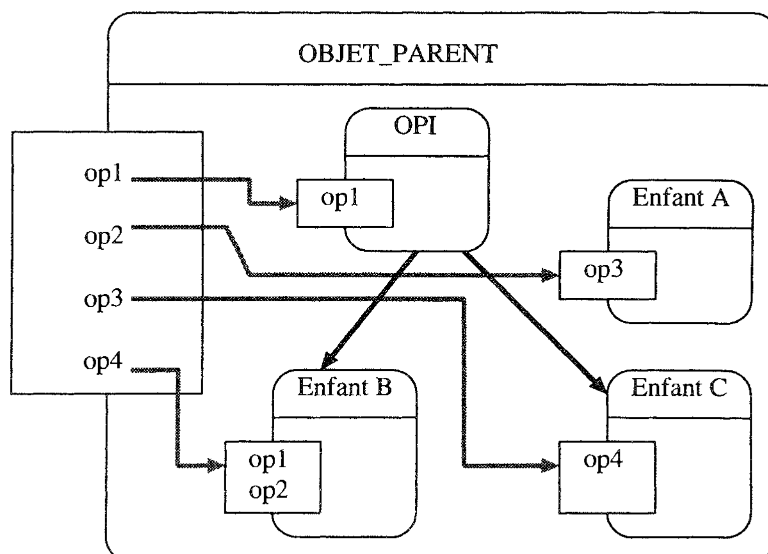


FIG. A.3 - *La décomposition des objets passifs*

HOOD permet aux objets actifs d'utiliser les autres objets sans restriction mais contraint les objets passifs à utiliser uniquement des objets passifs et de manière acyclique. La relation est représentée par une flèche en gras, et matérialise le flot de contrôle de l'utilisateur vers l'utilisé.

La relation INCLUDE ou parent-enfant

Afin de fournir une décomposition d'un système, un objet parent est décomposé en un ensemble d'objets enfants, qui collectivement fournissent les mêmes fonctionnalités que le parent. Chacun des enfants peut à son tour être décomposé. Ce processus est basé sur la relation INCLUDE entre objets. La récursivité est interdite, si bien qu'un système donné se représente par une hiérarchie parents-enfants avec un objet sommet et différents objets à différents niveau juniors. La représentation graphique est la suivante :

- la relation include est représentée en dessinant les enfants à l'intérieur du parent;
- la correspondance entre opérations du niveau parent et opérations du niveau enfant est matérialisée par une flèche en pointillé;
- un objet passif peut contenir des objets actifs, à partir du moment où cela n'induit pas une structure de contrôle d'objet au niveau parent.

Les règles de décomposition des objets passifs

Une opération du niveau parent peut être implantée par une ou plusieurs opérations enfants. Dans ce dernier cas un objet enfant spécifique, du nom de l'opération parent, gère la structure de contrôle (OPCS) vers les autres opérations enfants.

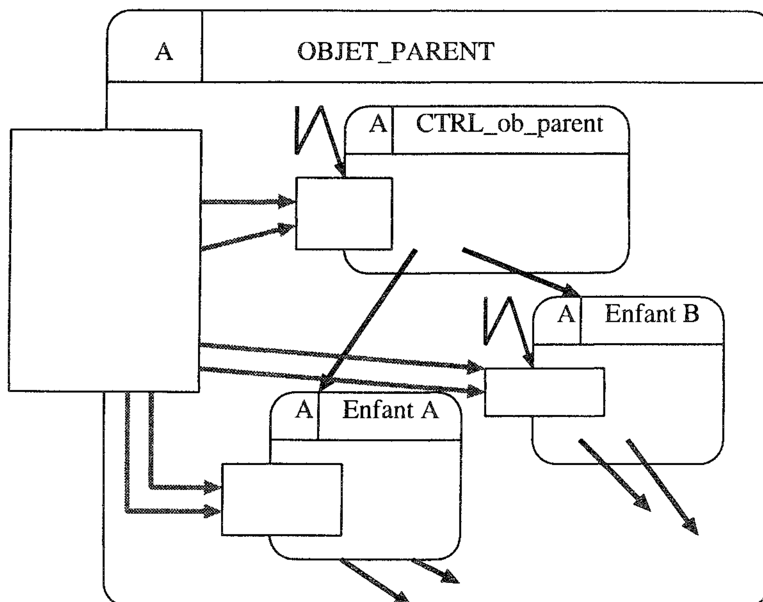


FIG. A.4 - *La décomposition des objets actifs*

Les règles de décomposition des objets actifs

L'interaction des flots de contrôle est décrite au niveau parent dans une structure de contrôle d'objet (OBCS). L'OBCS peut être implanté selon trois stratégies :

- L'OBCS est géré dans un objet enfant dédié. Dans ce cas cet objet devrait être nommé CTRL-Objet-Parent, et comme les flots de contrôle entrant dans cet objet interagissent, il doit être actif.

- L'OBCS est "géré" par plusieurs enfants. (cas d'implantation d'opérations parallèles).

- L'OBCS est géré à la fois par un objet enfant dédié et d'autres objets enfants (cas général).

L'expression des flots de contrôle, de données et des exceptions

La relation "use" entre deux objets est matérialisée par une flèche en gras; généralement cette relation d'utilisation consistera en l'appel d'une ou plusieurs opérations offertes par l'objet utilisé. La flèche "use" matérialise donc le flot de contrôle allant d'un objet vers un autre.

Si une ou plusieurs exceptions peuvent être levées par un objet, on peut l'indiquer par un barre perpendiculaire à la flèche "use" donnant la direction du flot de contrôle. Cette barre sera commentée avec la liste des exceptions pouvant 'remonter' le long du flot indiqué par la flèche.

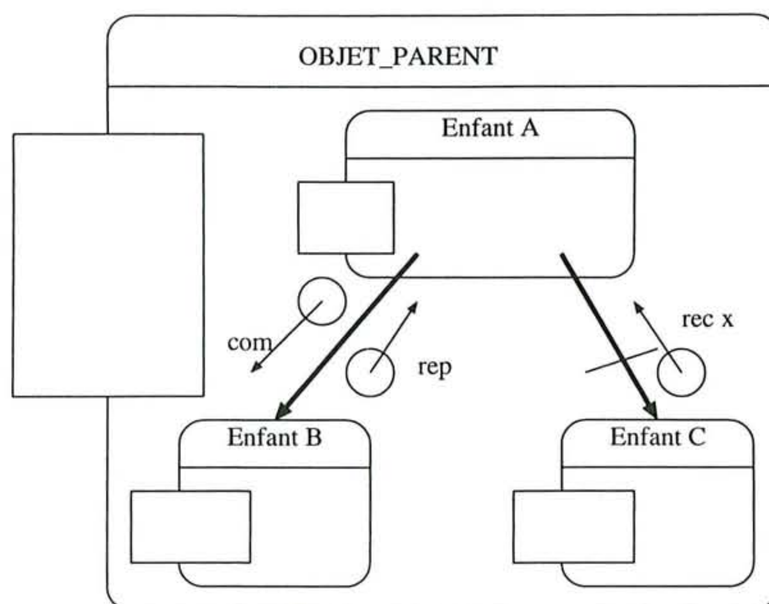


FIG. A.5 - *Les flots de contrôle, de données et des exceptions*

A.4 Description formelle des objets HOOD

La description *textuelle* des objets HOOD est exprimée par un squelette de définition d'objet (ODS = Object Description Skeleton) dont les champs sont mis à jour au fur et à mesure que la conception avance. Chaque champ d'un ODS est décrit par un PDL HOOD qui est défini formellement à partir d'un ensemble de mots réservés. La syntaxe formelle est défini par une forme de grammaire normale de BACCHUS (BNF).

Exemple: ODS pour un objet d'environnement

OBJECT object-Name **is ENVIRONMENT** Object-Type
    /* – Object-Type est limité à **PASSIVE** ou **ACTIVE** */
    VIRTUAL-NODE
    Description
    Implementation-or-synchronisation-constraints
    Provided-Interface
    Required-Interface
    [Object-Control-structure] /*Seulement dans le cas d'un objet actif */
END-OBJECT Object-Name

A.4.1 La translation de HOOD vers Ada

La transition vers le code final consistera en :

- La génération de squelettes d'unités de code à partir des définitions des objets (ODS). Cette génération pourra être automatisée ou semi automatisée.

- La réutilisation et l'affinement jusqu'au code des descriptions des structures de contrôle des objets et des opérations, exprimées en ADA-PDL.

Annexe B

Global-Exemple 1 : Système de base de données

Cette annexe vient en complément de chapitre 5 : Le prototype du moodèle WHAT-IF.

B.1 La définition du problème

Imaginons que nous ayons une importante collection de disques. De temps en temps, il nous arrive d'ajouter de nouveaux albums, de nous débarrasser de ceux dont nous ne voulons plus, ou d'en remplacer des vieux par de plus récents. Lorsque le nombre d'albums s'accroît, il devient de plus en plus laborieux de rechercher manuellement dans notre collection des morceaux particuliers, ou des morceaux d'un style donné ou joués par tel ou tel artiste. Pour cette raison, nous voulons construire un système de base de données simple qui nous permette de contrôler notre collection.

Quel est le cahier des charges de ce problème? Il nous faut principalement une base de données formée d'articles qui contiennent l'information correspondant à tous les albums de notre collection. Pour chaque album, nous voulons conserver le titre, le nom de l'artiste, le style de musique (par exemple classique, jazz ou rock), l'année de sortie, ainsi que le nom et la durée de chaque morceau. Il nous faudra les fonctionnalités habituelles de gestion d'une base de données - la possibilité de créer, d'ouvrir et de fermer la base de données et la possibilité d'ajouter, de supprimer et de modifier les données de tout album. De plus, nous voulons pouvoir produire des états concernant les albums de notre base de données. Plus précisément, nous voulons pouvoir rechercher des albums en fonction de n'importe quel critère. Nous voulons également être capables de tirer les articles résultants, puis d'imprimer un état. Pour offrir la plus grande souplesse de recherche, nous permettrons à l'utilisateur de spécifier n'importe quel nombre de critères de recherche à vérifier pour chaque état. Ainsi, l'utilisateur pourra rechercher parmi ceux-ci les albums parus une année donnée. De cette façon, la recherche s'effectue sur des ensembles d'articles de plus petits, et non sur la base de données tout entière.

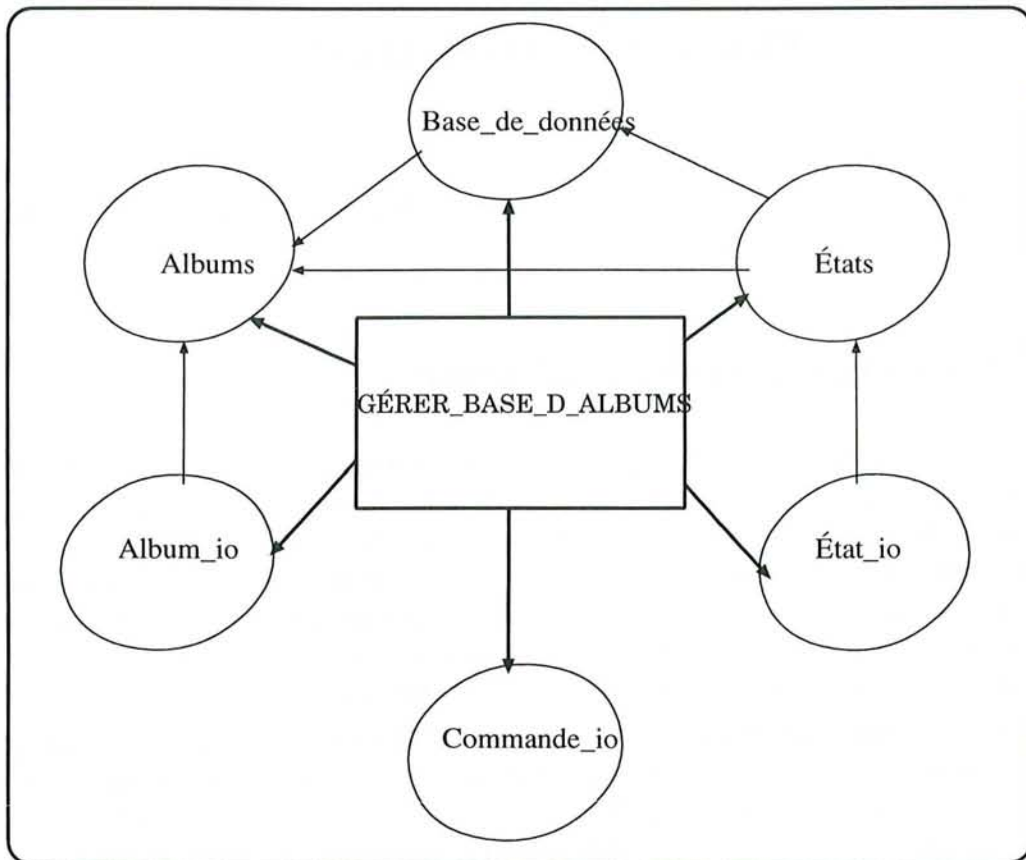


FIG. B.1 - *Le problème du système de base de données*

B.2 Le document de conception

B.2.1 L'introduction

Comment avancer dans la création d'une solution informatique à notre problème? Avec une approche fonctionnelle, nous commencerions par identifier les principales étapes du processus général. Cependant, cette approche conduit souvent à des solutions qui ne suivent pas le domaine de problème et qui sont réfractaires aux changements. Nous allons plutôt commencer par identifier les objets et les classes d'objets existant dans l'espace de problème.

B.2.2 L'identification des objets

Notre description du problème contient un certain nombre de noms qui décrivaient les entités concernées. Nous avons en particulier mentionné les entités suivantes (figure B.2) :

- inquire_about_project_data,
- inquiry_operation,
- project.

L'objet *project* comporte deux objets enfants : unit_information et data_base (figure B.3)

L'objet *inquiry_operations* contient aussi six objets fils : print, print_heading, request, collect_statistics, list_data, et select_unit (figure B.4).

B.2.3 L'identification des opérations

Nous avons identifié les principaux objets intéressants, mais nous n'en sommes pas encore établir les opérations. Il nous faut d'abord considérer le comportement de nos abstractions.

Examinons par exemple l'objet *Inquiry_operations.* D'après la description de notre problème, cet objet contient les opérations suivantes (figure B.4) :

- *command_inquiry* : pour imprimer,
- *request_inquiry* : une requête particulière,
- *collect_statistics_inquiry* : une collection de statistiques,
- *list_data_inquiry* : une liste de tous les albums,
- *select_unit_inquiry* : renvoie la valeur d'un article particulier de la base.

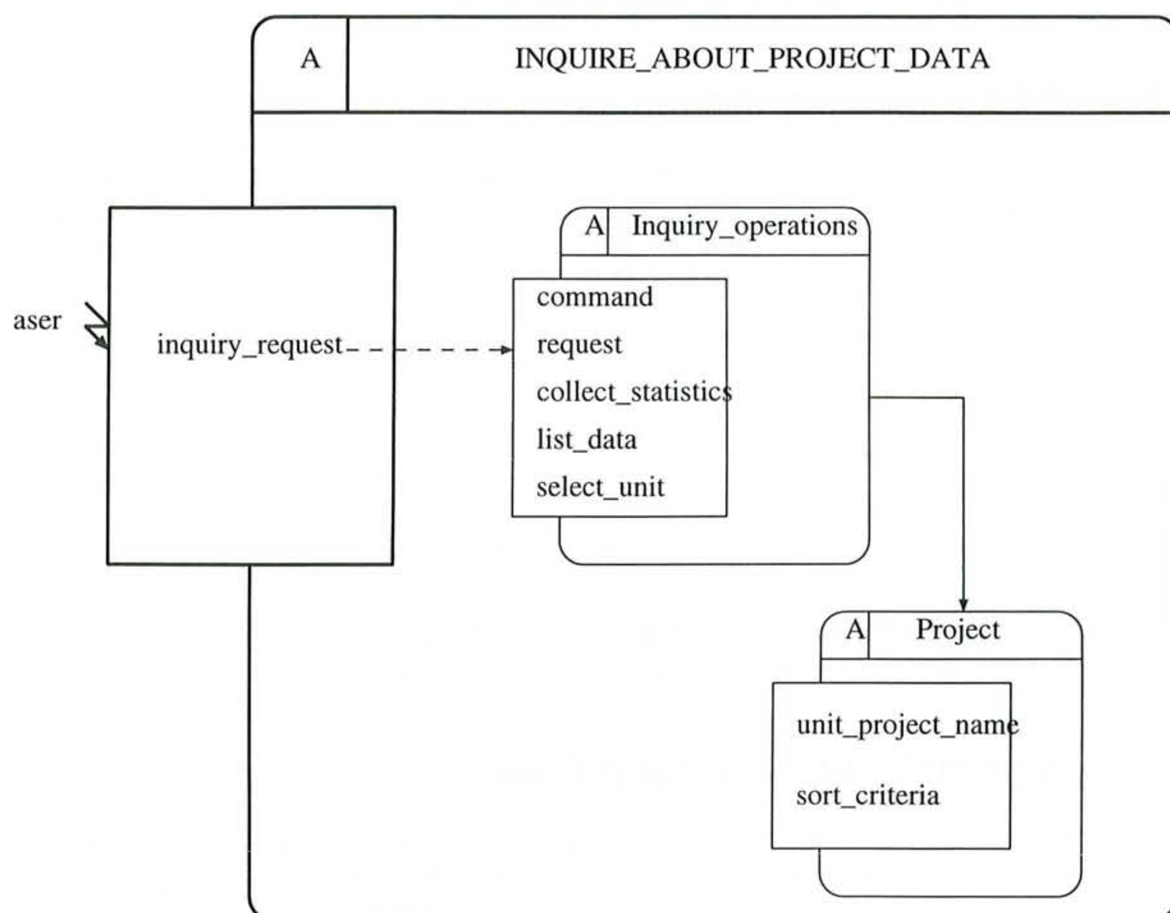


FIG. B.2 - *La description graphique de l'objet inquire_about_project_data (Diagramme HOOD)*

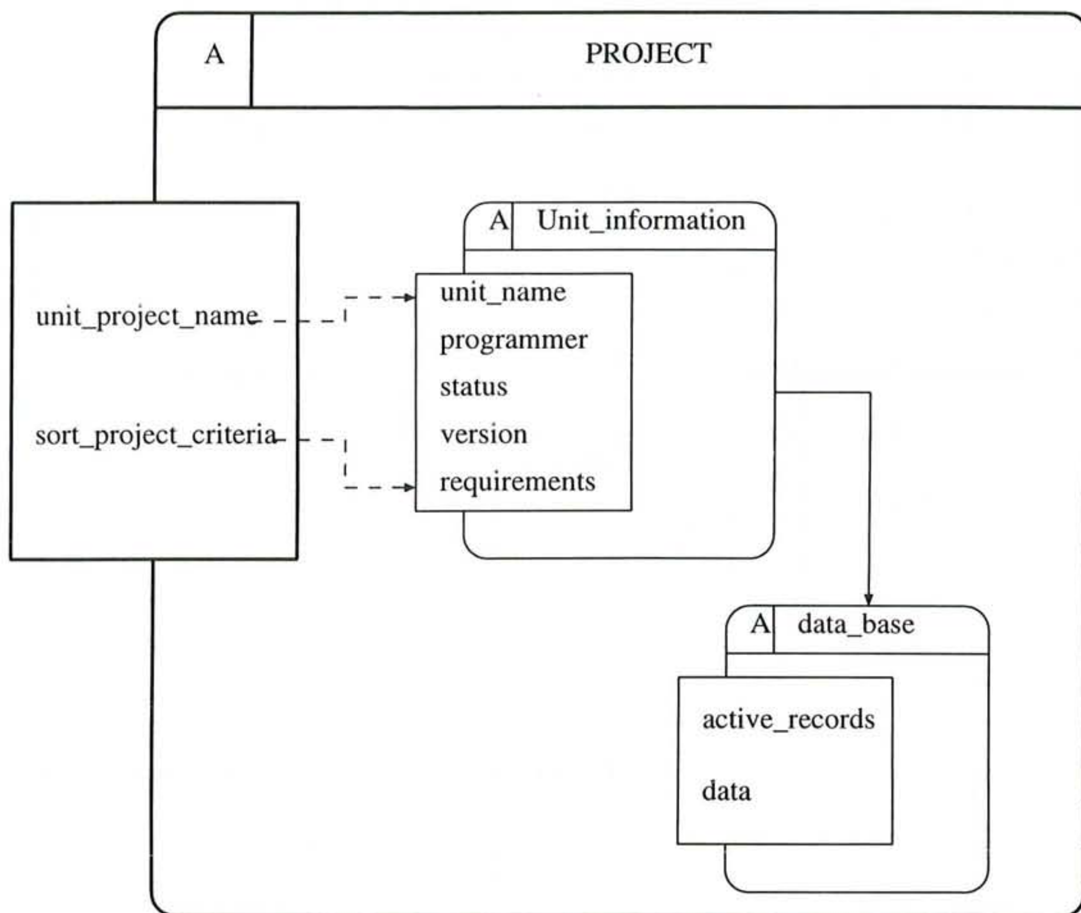


FIG. B.3 - *La description graphique de l'objet project(Diagramme HOOD)*

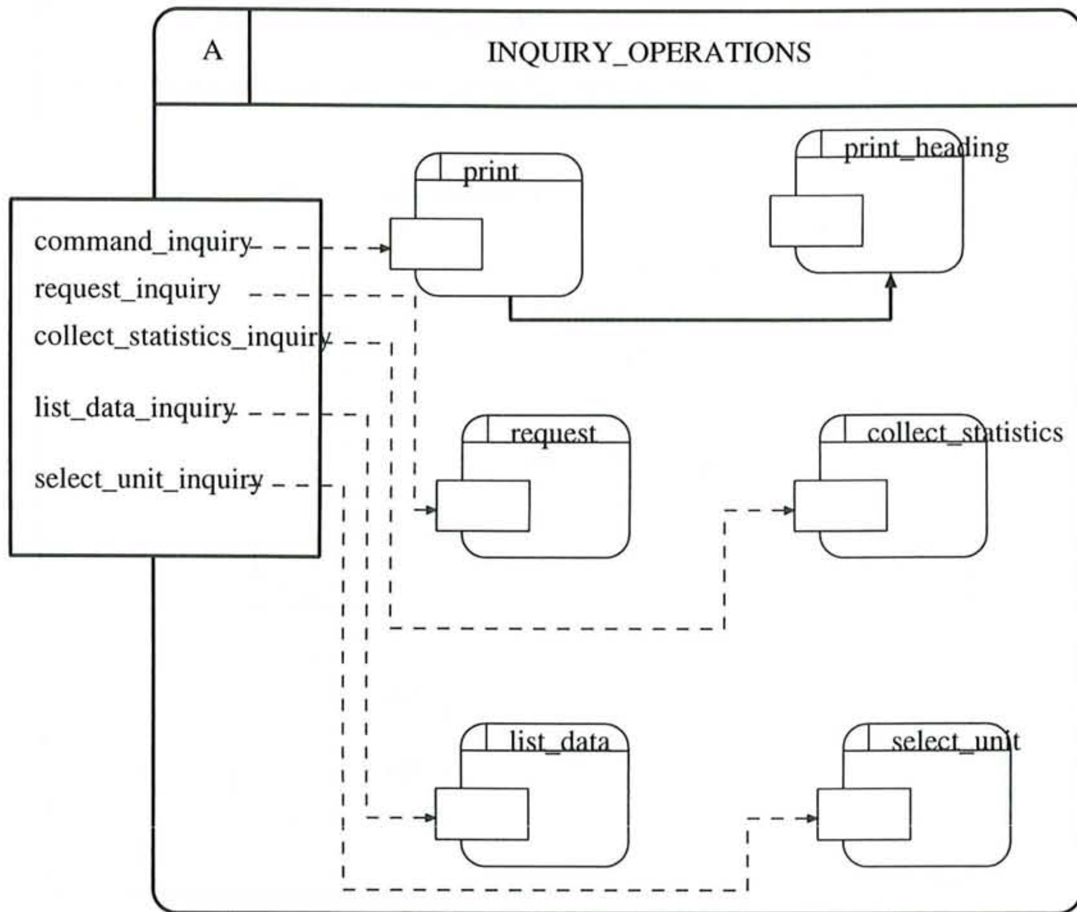


FIG. B.4 - *La description graphique de l'objet inquiry_ operations (Diagramme HOOD)*

B.3 Formalisation de la solution : la description du squelette de l'objet (ODS)

Dans cette section nous allons présenter une partie de l'ODS pour l'objet parent *inquire_about_project_data*.

```
OBJECT inquire_about_project_data IS ACTIVE

DESCRIPTION
$PRAGMA_HCS
-#-

IMPLEMENTATION_OR_SYNCHRONISATION_CONSTRAINTS
-#-
$PRAGMA_MAIN
$PRAGMA_EXECEPTION


PROVIDED_INTERFACE
TYPES
  NONE
CONSTANTS
  NONE
OPERATIONS
  inquiry_request;-#-

OPERATION_SETS
  NONE
EXCEPTIONS
  NONE

...

-#-

...

OBJECT_CONTROL_STRUCTURE
 DESCRIPTION
 -#-
 CONSTRINTED_OPERATIONS
   status CONSTRAINED_BY (ASER -#_); $PRAGMA_SERVER $PRAGMA_FIFO
 $PRAGMA_GROUP
 $OBCS_CODE
```

```
INTERNALS OBJECTS
   inquiry_operations;-#-
   project;-#
DECLARATIONS
  NONE
OPERATIONS
  NONE


OPERATION inquiry_request IS

...

-#-

...

EXCEPTIONS
 NONE
IMPLEMENTED_BY request.inquiry_operation
END)_OPERATION  inquiry_request

END_OBJECT monitor_temperature
```

L'ODS pour les autres objets peuvent être générés de la même façon que pour l'objet *inquire_ about_ project_ data.*

B.4 Le code Ada généré et enrichi

```
package project is

        package unit_information is
                type name_type          is new string(1..20);
                type programmer_type    is new string(1..20);
                type reference;
                type reference_access   is access reference;
                type reference          is record
                                                document : string(1..80);
                                                page     : positive;
                                                next     : reference_access;
                                        end record;
                type status_type        is(design, code, test, operational);
                type version_type       is range 0..99;
                type unit_record        is record
                                                programmer  : programmer_type;
                                                status : status_type;
                                                unit_name : name_type;
                                                version : version_type;
                                        end record;
```

```
        end unit_information;

        package data_base is
                use unit_information;
                maximum_records         : constant := 100;
                type record_index       is range 0..maximum_records;
                type project_records    is array(record_index) of unit_record;
                active_records          : record_index;
                data                    : project_records;
        end data_base;

end project;
with sequential_io;
package body project is
        package body data_base is
        package base_io is new sequential_io(unit_record);
        use base_io;
        data_file : base_io.file_type;
begin
        active_records := 0;
        open(data_file, mode => in_file, name => "project_a/units");
        while not end_of_file(data_file)
                loop
                        active_records := active_records + 1;
                        read(data_file,item => data(index));
                end loop;
        close(data_file);
        end data_base;

end project;

with project;
use project;
package inquiry_operations is
        type command is (collect_statistics, list_data, quit, select_unit);
        function request return command;
        procedure collect_statistics;
        procedure list_data;
        procedure select_unit;
end inquiry_operations;

separate(list_data)
procedure sort(records  : in data_base.project_records;
               size     : in data_base.record_index;
               criteria : in sort_key;
               list     : out sort_list) is
        temp_record : unit_information.unit_record;
        function out_of_order(first    : in unit_information.unit_record;
                              second   : in unit_information.unit_record;
                              criteria : in sort_key) return boolean is

        begin
                case criteria is
                        when unit           => if first.unit_name > second.unit_name then
                                                return true;
                                               else
                                                return false;
                                               end if;
                        when requirements   => if first.requirements.document >
                                                second.requirements.document then
                                                return true;
                                               else
                                                return false;
                                               end if;
                end case;
        end out_of_order;
        begin
                for first_index in 1..size - 1
```

```
                loop
                  for second_index in (first_index + 1)..size
                      loop
                        if out_of_order(records(first_index),
                                        records(second_index),
                                        criteria) then
                           temp_record           := records(first_index);
                           records(first_index)  := records(second_index);
                           records(second_index) := temp_record;
                        end if;
                      end loop;
                end loop;
        for index in 1..size
                loop
                  list(index) := index;
                end loop;
        end sort;

with text_io;
use text_io;
package body inquiry_operations is

   procedure print(data_record : in unit_information.unit_record) is
        package status_io is new
                text_io.enumeration_io(unit_information.status_io);
        package version_io is new
                text_io.integer_io(unit_information.version_io);
        use status_io, version_io;
     begin
        set_col(to => 1);
        put(data_record.unit_name, width => 20);
        set_col(to => 21);
        put(data_record.programmer, width => 20);
        set_col(to => 42);
        put(data_record.status, width => 11);
        set_col(to => 54);
        put(data_record.version, width => 5);
        set_col(to => 62);
        put(data_record.requirements.document, width => 80);
        new_line;
     end print;

   procedure print_heading is
     begin
        set_col(to => 1);
        put("unit_nam\"e);
        set_col(to => 21);
        put("programmer_nam\"e);
        set_col(to => 42);
        put("status");
        set_col(to => 54);
        put("version");
        set_col(to => 62);
        put("requirements");
        new_line;
     end print_heading;

   function request return command is
     user_inquiry : command;
     package command_io is new text_io.enumeration_io(command);
    begin
        command_io.get(user_inquiry);
        text_io.new_line;
        return user_inquiry;
    end request;

procedure collect_statistics is
   package status_io is new
```

```
        text_io.enumeration_io(unit_information.status_type);
    use status_io;
    type absolute is range 0..data_base.maximum_records;
    package absolute_io is new text_io.integer_io(absolute);
    use absolute_io;
    type relative is delta 0.1 range 0.0..100.0;
    package relative_io is new text_io.fixed_io(relative);
    use relative_io;
    type status_record is record
                total   : absolute;
                percent : relative;
         end record;
    type status_data is array(unit_information.status_type)
                        of status_record;
    frequence : status_data := (design..operation =>
                               (total => 0, percent => 0.0));
begin
    put_line("collecting statistics...");
    new_line;
    put("status       absolute frequency    relative frequency");
    new_line;
    for index in 1 .. data_base.active_records
         loop
           frequency(data_base.data(index).status).total :=
                 frequency(data_base.data(index).status).total + 1;
         end loop;
    for index in unit_infromation.status_type
         loop
           frequency(index).percent := relative(frequency(index).total)/
                                       data_base.active_records;
           put(index, width => 11);
           put("       ");
           put(frequency(index).total);
           put("       ");
           put(frequency(index).percent);
           new_line;
    end loop;
end collect_statistics;

procedure list_data is
    type sort_key is (requirement, unit);
    sort_criteria : sort_key;
    package criteria_io is new text_io.enumeration_io(sort_key);
    use criteria_io;
    type sort_list is array (data_base.record_index)
             of data_base.record_index;
    sort_table  : sort_list;
    procedure sort (records  : in data_base.project_records;
                    size     : in data_base.record_index;
                    criteria : in sort_key;
                    list     : out sort_list) is separate;

    begin
         put("what field do you want to sort on?");
         get(sort_criteria);
         sort(records  => data_base.data,
              size     => data_base.active_records,
              criteria => sort_criteria,
              list     => sort_table);
         put("sorting data. . .");
         new_line;
         print_heading;
         for index in 1..data_base.total_records
           loop
                 print(data_base.data(sort_table(index)));
           end loop;
         end list_data;
```

```
   procedure select_unit is
        unit_name : unit_information.name_type;
   begin
        put("what unit do you want information on?");
        get(unit_name);
        new_line;
        put("selecting data by unit name. . .");
        new_line;
        print_heading;
        for index in 1..data_base.active_records
           loop
                if data_base.data(index).unit_name = unit_name then
                        print(data_base.data(index));
                        exit;
                end if;
           end loop;
   end select_unit;

   end inquiry_package;

with inquiry_operations, text_io;
use inquiry_operations;
procedure inquiry_about_project_data is
begin
  loop
        text_io.put("enter inquiry request:");
        case inquiry_operations.request is
           when collect_statistics =>
                text_io.put("statistics command accepted");
                text_io.new_line;
                inquiry_package.collect_statistics;
           when list_data =>
                text_io.put("list command accepted");
                text_io.new_line;
                inquiry_package.list_data;
           when quit  =>
                exit;
           when select_unit =>
                text_io.put("select command accepted");
                text_io.new_line;
                inquiry_operations.select_unit;
        end case;
  end loop;
end inquiry_about_project_data;
```

B.5 Les tables générées par le processeur

B.5.1 La table d'identificateurs

No.	Ident_Name	Ident_type	Ident_kind	Line	Ident_parent
0	project	package_spec	PKS	1	project
1	unit_information	package_spec	PKS	3	project
2	name_type	DTD	TYP/NEW	4	unit_information
3	programmer_type	DTD	TYP/NEW	5	unit_information
4	reference	incomplete_type	ICD	6	unit_information
5	reference_access	ACD	ACD	7	unit_information
6	reference	RTD	RTD	8	unit_information
7	document	string	RTE	9	unit_information
8	page	positive	RTE	10	unit_information
9	next	reference_access	RTE	11	unit_information
10	status_type	END	END	13	unit_information

11	version_type	ITD	ITD	14 unit_information
12	unit_record	RTD	RTD	15 unit_information
13	programmer	programmer_type	RTE	16 unit_information
14	status	status_type	RTE	17 unit_information
15	unit_name	name_type	RTE	18 unit_information
16	version	version_type	RTE	19 unit_information
17	data_base	package_spec	PKS	23 project
18	maximum_records	number	CND	25 data_base
19	record_index	ITD	ITD	26 data_base
20	project_records	ATD	ATD	27 data_base
21	active_records	record_index	OBD	28 data_base
22	data	project_records	OBD	29 data_base
23	project	package_body	PKB	34 project
24	data_base	package_body	PKB	35 project
25	base_io	package_body	NEW	36 data_base
26	file_type	base_io	OBD	38 data_base
27	data_file	base_io	OBD	38 data_base
28	inquiry_operations	package_spec	PKS	54 project
29	command	END	END	55 inquiry_operations
30	request	function_spec	FNS	56 inquiry_operations
31	collect_statistics	procedure_spec	PRS	57 inquiry_operations
32	list_data	procedure_spec	PRS	58 inquiry_operations
33	select_unit	procedure_spec	PRS	59 inquiry_operations
34	sort	procedure_spec	PRS	63 project
35	project_records	data_base	FMP	63 sort
36	records	data_base	FMP	63 sort
37	record_index	data_base	FMP	64 sort
38	size	data_base	FMP	64 sort
39	criteria	sort_key	FMP	65 sort
40	list	sort_list	FMP	66 sort
41	unit_record	unit_information	OBD	67 sort
42	temp_record	unit_information	OBD	67 sort
43	out_of_order	function_spec	FNS	68 sort
44	unit_record	unit_information	FMP	68 out_of_order
45	first	unit_information	FMP	68 out_of_order
46	unit_record	unit_information	FMP	69 out_of_order
47	second	unit_information	FMP	69 out_of_order
48	criteria	sort_key	FMP	70 out_of_order
49	inquiry_operations	package_body	PKB	109 project
50	print	procedure_spec	PRS	111 inquiry_operations
51	unit_record	unit_information	FMP	111 print
52	data_record	unit_information	FMP	111 print
53	status_io	package_body	NEW	112 print
54	version_io	package_body	NEW	114 print
55	print_heading	procedure_spec	PRS	131 inquiry_operations
56	request	function_spec	FNS	146 inquiry_operations
57	user_inquiry	command	OBD	147 request
58	command_io	package_body	NEW	148 request
59	collect_statistics	procedure_spec	PRS	155 inquiry_operations
60	status_io	package_body	NEW	156 collect_statistics
61	absolute	ITD	ITD	159 collect_statistics
62	absolute_io	package_body	NEW	160 collect_statistics
63	relative	RED	RED	162 collect_statistics
64	relative_io	package_body	NEW	163 collect_statistics
65	status_record	RTD	RTD	165 collect_statistics
66	total	absolute	RTE	166 collect_statistics
67	percent	relative	RTE	167 collect_statistics
68	status_data	ATD	ATD	169 collect_statistics
69	frequence	status_data	OBD	171 collect_statistics
70	list_data	procedure_spec	PRS	196 inquiry_operations
71	sort_key	END	END	197 list_data
72	sort_criteria	sort_key	OBD	198 list_data
73	criteria_io	package_body	NEW	199 list_data
74	sort_list	ATD	ATD	201 list_data
75	sort_table	sort_list	OBD	203 list_data
76	sort	procedure_spec	PRS	204 list_data
77	project_records	data_base	FMP	204 sort
78	records	data_base	FMP	204 sort

79	record_index	data_base	FMP	205	sort
80	size	data_base	FMP	205	sort
81	criteria	sort_key	FMP	206	sort
82	list	sort_list	FMP	207	sort
83	select_unit	procedure_spec	PRS	225	list_data
84	name_type	unit_information	OBD	226	select_unit
85	unit_name	unit_information	OBD	226	select_unit
86	inquiry_about_project_data	procedure_spec	PRS	247	inquiry_operations

B.5.2 La table de sous-programmes et la table des paramètres de sous-programmes

No.	Sub_program_Name	Sub_prog_type	F_line_No	L_line_No	Presence_of_parameter
0	request	function_spec	56	105	N
1	collect_statistics	procedure_spec	57	58	N
2	list_data	procedure_spec	58	59	N
3	select_unit	procedure_spec	59	60	N
4	sort	procedure_spec	63	88	Y
5	out_of_order	function_spec	68	86	Y
6	print	procedure_spec	111	129	Y
7	print_heading	procedure_spec	131	243	N
8	request	function_spec	146	153	N
9	collect_statistics	procedure_spec	155	194	N
10	list_data	procedure_spec	196	243	N
11	sort	procedure_spec	204	223	Y
12	select_unit	procedure_spec	225	241	N
13	inquiry_about_project_data	procedure_spec	247	0	N

No.	Sub_program_Name	Param_seq_No.	Parameter	Parameter_mode_type
0	sort	1	project_records	IN
1	sort	2	records	IN
2	sort	3	record_index	IN
3	sort	4	size	IN
4	sort	5	criteria	IN
5	sort	6	list	OUT
6	out_of_order	1	unit_record	IN
7	out_of_order	2	first	IN
8	out_of_order	3	unit_record	IN
9	out_of_order	4	second	IN
10	out_of_order	5	criteria	IN
11	print	1	unit_record	IN
12	print	2	data_record	IN

B.5.3 La table d'importation

No.	Imported_objects	Importing_Objects	Line_No.
0	unit_information	data_base	24
1	sequential_io	data_base	33
2	base_io	data_base	37
3	project	project	52
4	project	project	53
5	text_io	inquiry_operations	107
6	text_io	inquiry_operations	108
7	status_io	inquiry_operations	116

8	status_io	project	158
9	absolute_io	project	161
10	relative_io	project	164
11	criteria_io	project	200
12	inquiry_operations	project	245
13	text_io	project	245
14	inquiry_operations	project	246

B.5.4 La table des objets génériques

No.	Generic_Ident	New_Ident	Line_No.
===	=============	=========	========
0	base_io	sequential_io	36
1	status_io	text_io	112
2	version_io	text_io	114
3	command_io	text_io	148
4	status_io	text_io	156
5	absolute_io	text_io	160
6	relative_io	text_io	163
7	criteria_io	text_io	199

B.6 La base de faits

```
actual_param([]).
array([]).
array(project_records).
array(sort_list).
array(status_data).
calls([], []).
calls(project, print).
calls(project, print).
calls(project, sort).
constant([]).
constant(maximum_records).
entry([]).
entry_act_param([]).
entry_for_param([]).
formal_param([]).
formal_param(criteria).
formal_param(criteria).
formal_param(criteria).
formal_param(data_record).
formal_param(first).
formal_param(list).
formal_param(list).
formal_param(project_records).
formal_param(project_records).
formal_param(record_index).
formal_param(record_index).
imports(data_base, base_io).
imports(data_base, sequential_io).
imports(data_base, unit_information).
imports(inquiry_operations, status_io).
imports(inquiry_operations, text_io).
imports(inquiry_operations, text_io).
imports(project, absolute_io).
imports(project, criteria_io).
imports(project, inquiry_operations).
```

```
imports(project, inquiry_operations).
imports(project, project).
imports(project, project).
imports(project, relative_io).
imports(project, status_io).
imports(project, text_io).
instantiates([], []).
instantiates(absolute_io, text_io).
instantiates(base_io, sequential_io).
instantiates(command_io, text_io).
instantiates(criteria_io, text_io).
instantiates(relative_io, text_io).
instantiates(status_io, text_io).
instantiates(status_io, text_io).
instantiates(version_io, text_io).
is_parameter_of([], [], []).
is_parameter_of(out_of_order, 1, unit_record).
is_parameter_of(out_of_order, 2, first).
is_parameter_of(out_of_order, 3, unit_record).
is_parameter_of(out_of_order, 4, second).
is_parameter_of(out_of_order, 5, criteria).
is_parameter_of(print, 1, unit_record).
is_parameter_of(print, 2, data_record).
is_parameter_of(sort, 1, project_records).
is_parameter_of(sort, 2, records).
is_parameter_of(sort, 3, record_index).
is_parameter_of(sort, 4, size).
is_parameter_of(sort, 5, criteria).
is_parameter_of(sort, 6, list).
object_declared_in([], []).
object_declared_in(absolute, collect_statistics).
object_declared_in(absolute_io, collect_statistics).
object_declared_in(active_records, data_base).
object_declared_in(base_io, data_base).
object_declared_in(collect_statistics, inquiry_operations).
object_declared_in(collect_statistics, inquiry_operations).
object_declared_in(command, inquiry_operations).
object_declared_in(command_io, request).
object_declared_in(version, unit_information).
object_declared_in(version_io, print).
object_declared_in(version_type, unit_information).
package_body([]).
package_body(absolute_io).
package_body(base_io).
package_body(command_io).
package_body(criteria_io).
package_body(data_base).
package_body(inquiry_operations).
package_body(project).
package_body(relative_io).
package_body(status_io).
package_body(status_io).
package_body(version_io).
package_spec([], []).
package_spec(data_base, 23).
package_spec(inquiry_operations, 54).
package_spec(project, 1).
package_spec(unit_information, 3).
procedure_body([]).
procedure_spec([], []).
procedure_spec(collect_statistics, 155).
procedure_spec(collect_statistics, 57).
procedure_spec(inquiry_about_project_data, 247).
procedure_spec(list_data, 196).
procedure_spec(list_data, 58).
procedure_spec(print, 111).
procedure_spec(sort, 63).
rec_type_var([]).
```

```
rec_type_var(document).
rec_type_var(next).
rec_type_var(page).
rec_type_var(percent).
rec_type_var(programmer).
rec_type_var(status).
rec_type_var(total).
rec_type_var(unit_name).
rec_type_var(version).
record([]).
record(reference).
record(status_record).
record(unit_record).
record_contains_var([], []).
record_contains_var(reference, document).
record_contains_var(reference, next).
record_contains_var(reference, page).
record_contains_var(status_record, percent).
record_contains_var(status_record, total).
record_contains_var(unit_record, programmer).
record_contains_var(unit_record, status).
record_contains_var(unit_record, unit_name).
record_contains_var(unit_record, version).
rename_as([], []).
renaming([]).
renaming(name_type).
renaming(programmer_type).
task_body([]).
task_spec([], []).
type([]).
type_return_value([], []).
uses([], [], []).
uses(active_records, assignment_statement, 186).
uses(active_records, assignment_statement, 40).
uses(active_records, assignment_statement, 44).
uses(active_records, assignment_statement, 44).
uses(active_records, function_or_library_call_statement, 213).
uses(active_records, loop_statement, 178).
uses(active_records, loop_statement, 234).
uses(close, function_or_library_call_statement, 47).
uses(collect_statistics, case_statement, 252).
uses(collect_statistics, function_or_library_call_statement, 255).
uses(command_io, function_or_library_call_statement, 150).
uses(criteria, case_statement, 73).
uses(unit_name, function_or_library_call_statement, 229).
uses(unit_name, if_statement, 236).
uses(unit_name, if_statement, 236).
uses(unit_name, if_statement, 74).
uses(unit_name, if_statement, 74).
uses(user_inquiry, function_or_library_call_statement, 150).
uses(user_inquiry, return_statement, 152).
uses(version, function_or_library_call_statement, 125).
uses(width, function_or_library_call_statement, 119).
uses(width, function_or_library_call_statement, 121).
uses(width, function_or_library_call_statement, 123).
uses(width, function_or_library_call_statement, 125).
uses(width, function_or_library_call_statement, 127).
uses(width, function_or_library_call_statement, 187).
variable([]).
variable(absolute).
variable(active_records).
variable(command).
variable(data).
variable(data_file).
variable(file_type).
variable(frequence).
variable(name_type).
variable(record_index).
```

```
variable(reference).
variable(reference_access).
variable(relative).
variable(sort_criteria).
variable(sort_key).
variable(sort_table).
variable(status_type).
variable(temp_record).
variable(unit_name).
variable(unit_record).
variable(user_inquiry).
variable(version_type).
```

Annexe C

Global-Exemple 2 : Surveillance d'environnement

De la même façon que l'annexe B, cette annexe aussi vient en complément du chapitre 5.

C.1 La définition du problème

Notre problème comporte plusieurs capteurs de température dans différentes pièces d'un bâtiment. Les capteurs échantillonnent en permanence la température ambiante; si un capteur donné détecte une température supérieure à une limite donnée, notre système déclenche une alarme. Un capteur affiche également la température, si elle est à l'intérieur des limites. L'utilisateur peut interagir avec le système en donnant la limite d'alarme pour chaque capteur et en lisant l'état de tous les capteurs. Lorsqu'une alarme sonne, l'utilisateur peut rapidement localiser la température anormale. Périodiquement, notre système doit imprimer la valeur courante de tous les capteurs sur un journal permanent. Notre système doit être également capable de détecter une faute de périphérique et déclencher une alarme.

C.2 Le document de conception

C.2.1 L'introduction

En examinant les éléments de l'espace de notre problème, nous pouvons extraire immédiatement les objets et les classes d'objets qui nous intéressent pour notre analyse. Nous avons mentionné en particulier (figure C.1):

- Un ensemble de capteurs;
- Une imprimante;
- Une alarme pour les conditions de dépassement et les erreurs d'imprimante;

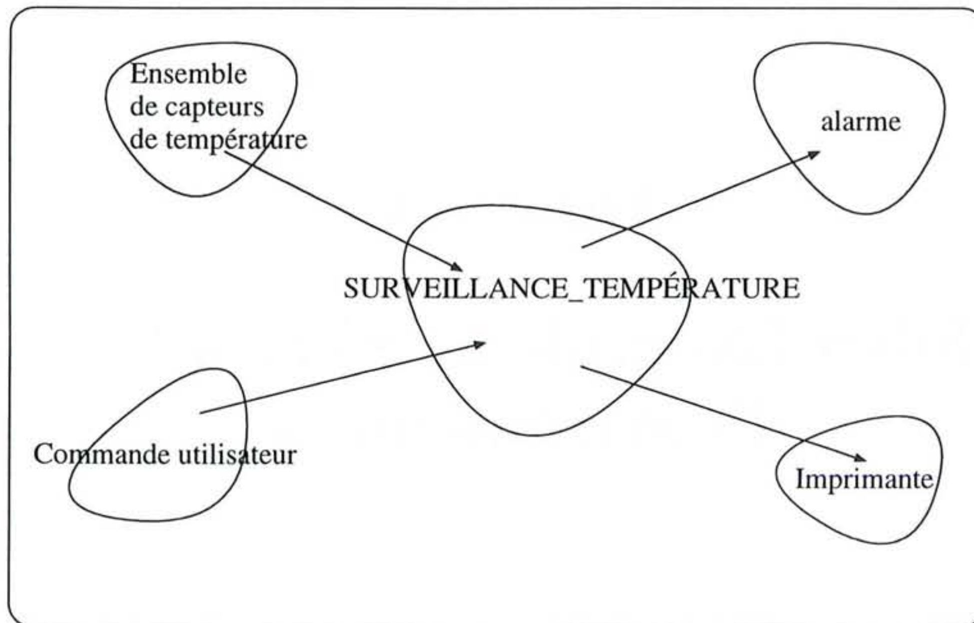


FIG. C.1 - *Le problèlem de surveillance d'environnement*

- Des ports d'entrées-sorties en mémoire;
- Un utilisateur.

L'ensemble des capteurs définit en fait une classe d'objets alors que notre système ne comporte qu'une imprimante et une alarme. Par conséquent, nous pouvons abstraire les capteurs par des types de données abstraits et considérer l'imprimante et l'alarme comme des machines à état abstraits.

C.2.2 L'identification des objets

Nous identifions les objets suivants (figure C.2) :

- recording_device,
- collection_of_sensors,
- timer,
- alarm.

C.2.3 L'identification des opérations

Dans cette étape, nous devons envisager le comportement de tous les objets du point de vue externe. Non seulement nous spécifierons ici les opérations subies par

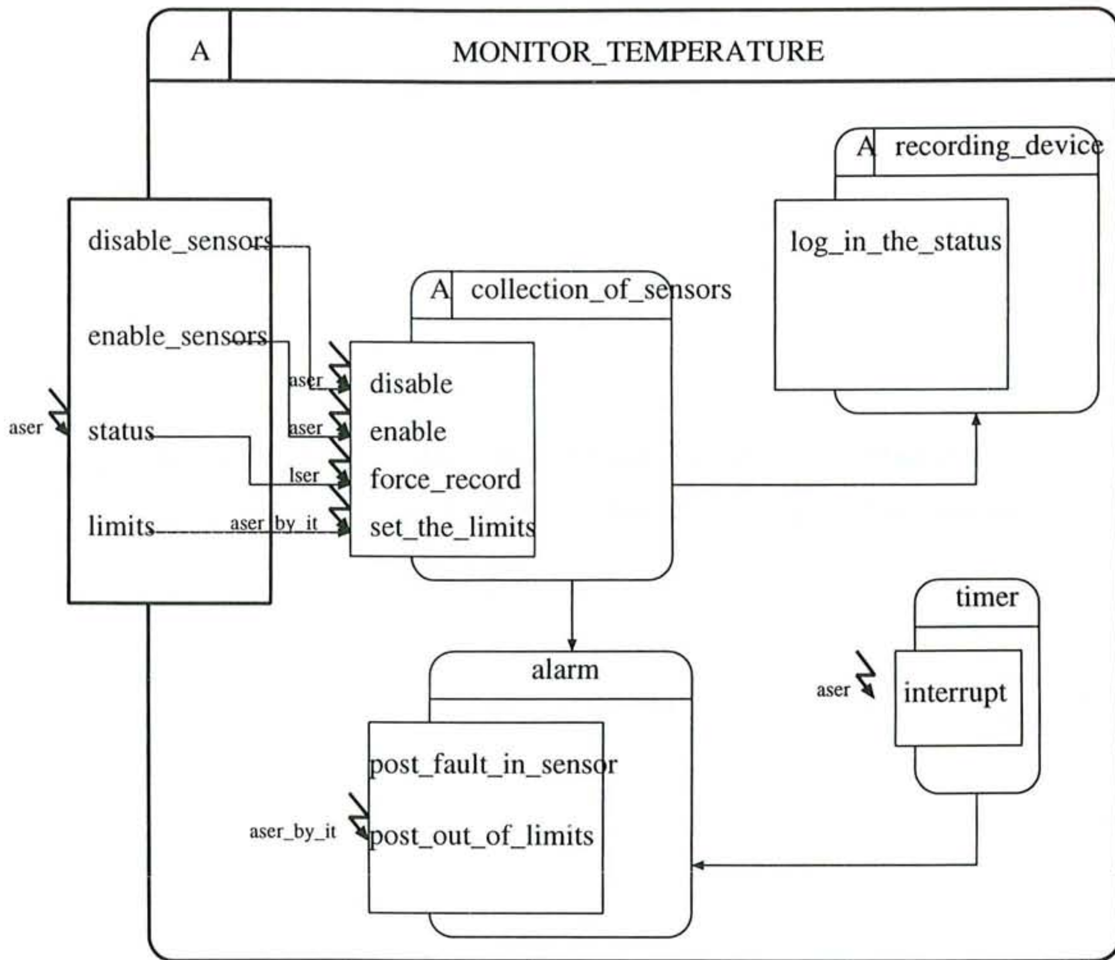


FIG. C.2 - *La description graphique (Diagramme HOOD)*

chacun des objets, mais nous devons également identifier le parallélisme de chacun. Par exemple, nous allons abstraire un capteur comme une entité parallèle. Son rôle principal est de surveiller la température d'une pièce en permanence. Donc, Nous identifions les opérations suivantes (figure C.2) :

- log_in_the_status,
- disable,
- enable,
- force_record,
- set_the_limits,
- interrupt,
- post_fault_in_sensor,
- post_out_of_limits.

C.3 Formalisation de la solution : la description du squelette de l'objet (ODS)

Dans cette section nous allons présenter une partie de l'ODS pour l'objet parent *monitor_temperature*.

```
OBJECT monitor_temperature IS ACTIVE

DESCRIPTION
$PRAGMA_HCS
-#-

IMPLEMENTATION_OR_SYNCHRONISATION_CONSTRAINTS
-#-
$PRAGMA_MAIN
$PRAGMA_EXECEPTION


PROVIDED_INTERFACE
TYPES
  NONE
CONSTANTS
  NONE
OPERATIONS
  disable_sensors;-#-
```

```
  enable_sensors;-#-
  status;-#-
  limits;-#-
OPERATION_SETS
  NONE
EXCEPTIONS
  NONE

...

-#-

...

OBJECT_CONTROL_STRUCTURE
 DESCRIPTION
 -#-
 CONSTRINTED_OPERATIONS
   status CONSTRAINED_BY (ASER -#_); $PRAGMA_SERVER $PRAGMA_FIFO
 $PRAGMA_GROUP
 $OBCS_CODE


INTERNALS OBJECTS
   recording_device;-#-
   collection_of_sensors;-#-
   timer;-#-
   alarm;-#
DECLARATIONS
  NONE
OPERATIONS
  NONE


OPERATION disable_sensors IS

...

-#-

...

EXCEPTIONS
 NONE
IMPLEMENTED_BY collection_of_sensors.disable
```

```
END)_OPERATION  disable_sensors


OPERATION enable_sensors IS

...

-#-

...

EXCEPTIONS
 NONE
IMPLEMENTED_BY collection_of_sensors.enable
END)_OPERATION  enable_sensors


OPERATION status IS

...

-#-

...

EXCEPTIONS
 NONE
IMPLEMENTED_BY collection_of_sensors.force_record
END)_OPERATION  status


OPERATION limits IS

...

-#-

...

EXCEPTIONS
 NONE
IMPLEMENTED_BY collection_of_sensors.limits
END)_OPERATION  limits
```

```
END_OBJECT monitor_temperature
```

L'ODS pour les autres objets peuvent être générés de la même façon que pour l'objet parent.

C.4 Le code Ada généré et enrichi

```
with text_io, system;
use text_io;
procedure monitor_temperatures is

  type command      is (disable, enable, record_status, set_limits);
  type sensor_name  is (lobby, main_office, warehouse, stock_room,
                        terminal_room, library, computer_room,
                        lounge, clean_room);
  type sensor_state is (disable, enable);
  type sensor_value is delta 0.5 range 0.0 .. 100.0;

  package command_io       is new enumeration_io(command);
  use     command_io;
  package sensor_name_io   is new enumeration_io(sensor_name);
  use     sensor_name_io;
  package sensor_value_io  is new fixed_io(sensor_value);
  use     sensor_value_io;

  task alarm is
        entry post_fault_in_sensor;
        entry post_out_of_limits(on_sensor : in sensor_name);
  end alarm;

  task collection_of_sensors is
        entry disable          (sensor     : in sensor_name);
        entry enable           (sensor     : in sensor_name);
        entry force_record     (of_sensor  : in sensor_name);
        entry set_the_limits   (for_sensor : in sensor_name;
                                low_limit  : in sensor_value;
                                high_limit : in sensor_value);
  end collection_of_sensors;

  task timer is
        entry interrupt;
        for interrupt use at 16#8E#;
  end timer;

  high_bound    : sensor_value;
  low_bound     : sensor_value;
  name          : sensor_name;
  user_command  : command;
  value         : sensor_value;

  task body alarm                 is separate;
  task body recording_device      is separate;
  task body collection_of_sensors is separate;
  task body timer                 is separate;

  begin
    loop
      begin -- start of a local block with exception handler
        put("enter your command:");
        get(name);
        new_line;
        put_line("command accepted");
```

```
        case user_command is
                when disable =>
                        put("enter sensor name:");
                        get(name);
                        new_line;
                        collectiont_of_sensors.enable(sensor => name);
                        put_line("sensor enabled");
                when enable =>
                        put("enter sensor name:");
                        get(name);
                        new_line;
                        collectiont_of_sensors.enable(sensor => name);
                        put_line("sensor enabled");
                when record_status =>
                        put("enter sensor name:");
                        get(name);
                        new_line;
                        collectiont_of_sensors.force_record(of_sensor => name);
                        put_line("sensor status set");
                when set_limits =>
                        put("enter sensor name:");
                        get(name);
                        new_line;
                        put("enter lower limit:");
                        get(low_bound);
                        new_line;
                        put_line("lower limit accepted");
                        get(high_bound);
                        new_line;
                        put_line("upper limit accepted");
                        collection_of_sensors.set_the_limits
                                (for_sensor => name,
                                 low_limit  => low_bound,
                                 high_limit => high_bound);
                        put_line("limits set");
           end case;

           exception
                when data_error =>
                        put_line("illegal entry...try again");
           end;
    end loop;
end monitor_temperatures;


with system;
separate (monitor_temperatures)
task body alarm is

        bits   : constant := 1;
        words  : constant := 16*bits;

        type light      is (off, on);
        for  light'size use 1 * words;
        for  light      use (off => 16#0000#, ON => 16#FFFF#);
        fault_light     :light := off;
        for fault_light use at 16#0010#;

  type limit_check      is array (sensor_name) of light;
  for  limit_check'size use (sensor_nam\'epos(sensor_nam\'elast) + 1) * words;
  out_of_limits_light   :limit_check := limit_check'(others => off);
  for out_of_limits_light use at 16#0011#;

  begin
    loop
      select
        accept post_fault_in_sensor do
           fault_light := on;
```

```
        end post_fault_in_sensor;
      or
        accept post_out_of_limits(on_sensor : in sensor_name) do
           out_of_limits_light(on_sensor) := on;
        end post_out_of_limits;
      end select;
    end loop;
  end alarm;

with device_io;
separate (monitor_temperatures)
task body recording_device is
  begin
    loop
        accept log_the_status(of_sensor  : in sensor_name;
                              with_value : in sensor_value;
                              with_state : in sensor_state) do
               device_io.put(of_sensor);
               device_io.put(with_value);
               device_io.put(with_state);
        end log_the_status;
    end loop;
  end recording_device;

with set_package, system;
separate (monitor_temperatures)
task body collection_of_sensors is

  bits  : constant := 1;
  words : constant := 16*bits;

type sensor_record is record
  high_limit :sensor_value := sensor_valu\'elast;
  low_limit  :sensor_value := sensor_valu\'efirst;
  value      :sensor_value := sensor_valu\'efirst;
                 end record;
type sensor_group  is array (sensor_name) of sensor_recoed;
     sensor        : sensor_group;


package sensor_set is new set_package(universe => sensor_name);
use sensor_set;
active_sensors     :set := null_set;

type sensor_port      is range 0..(2**words - 1);
for sensor_port'size  use 1 * words;
type sensor_list  is array (sensor_name) of sensor_port;
for sensor_list'size  use (sensor_nam\'epos(sensor_nam\'elast) +1) * words;
    sensor_map        : sensor_list;
for sensor_map        use at 16#0100#;

begin
  loop
    select
        accept disable(sensor : in sensor_name) do
          active_sensors := active_sensors - sensor;
        end disable;
    or
        accept enable(sensor : in sensor_name) do
          active_sensors := active_sensors + sensor;
        end enable;
    or
        accept force_record(of_sensor : in sensor_name) do
          if is_a_member(of_sensor, of_set => active_sensors) then
              recording_device.log_the_status(of_sensor,
                                              sensor(of_sensor).value,
                                              with_state => enabled);
          else
```

```
          recording_device.log_the_status(of_sensor,
                                          with_value => sensor_valu\'efirst,
                                          with_state => disabled);
        end if;
      end force_record;
      or
      accept set_the_limits(for_sensor : in sensor_name;
                            low_limit  : in sensor_value;
                            high_limit : in sensor_value)do
        sensor(for_sensor).low_limit  := low_limit;
        sensor(for_sensor).high_limit := high_limit;
      end set_the_limits;
        else
      for i in sensor_name
        loop
          if is_a_member(i, of_set => active_sensors) then
            sensor(i).value := (sensor_map(i) * sensor_value(0.5));
            if (sensor(i).value < sensor(i).low_limit) or
                 (sensor(i).value > sensor(i).high_limit) then
               alarm.post_out_of_limits(i);
            end if;
          end if;
        end loop;
      end select;
    end loop;
end collection_of_sensors;

separate (monitor_temperatures)
task body timer is
  minutes       : constant := 1;
  type interval is range 0 .. 15;
  ticks         : interval := 0;

begin
  loop
    accept interrupt do
        ticks := ticks + 1;
        if ticks = 15 * minutes then
           for i in sensor_name
             loop
               select
                collection_of_sensors.force_record(of_sensor => i);
               or
                delay 5.0;
                alarm.post_fault_in_sensor;
               end select;
             end loop;
           ticks := 0;
        end if;
    end interrupt;
  end loop;
end timer;
```

C.5 Les tables générées par le processeur

C.5.1 La table d'identificateurs

```
No.  Ident_Name             Ident_type        Ident_kind   Line   Ident_parent
===  ==========             ==========        ==========   ====   ===========

0    monitor_temperatures    procedure_spec       PRS        3      text_io
1    command                END                   END        5      monitor_temperatures
2    sensor_name            END                   END        6      monitor_temperatures
```

3	sensor_state	END	END	9	monitor_temperatures
4	sensor_value	RED	RED	10	monitor_temperatures
5	command_io	package_body	NEW	12	monitor_temperatures
6	sensor_name_io	package_body	NEW	14	monitor_temperatures
7	sensor_value_io	package_body	NEW	16	monitor_temperatures
8	alarm	task_spec	TKS	19	monitor_temperatures
9	post_fault_in_sensor	entry_defn	ENT	20	alarm
10	post_out_of_limits	entry_defn	ENT	21	alarm
11	on_sensor	sensor_name	EFP	21	alarm
12	collection_of_sensors	task_spec	TKS	24	monitor_temperatures
13	disable	entry_defn	ENT	25	collection_of_sensors
14	sensor	sensor_name	EFP	25	collection_of_sensors
15	enable	entry_defn	ENT	26	collection_of_sensors
16	sensor	sensor_name	EFP	26	collection_of_sensors
17	force_record	entry_defn	ENT	27	collection_of_sensors
18	of_sensor	sensor_name	EFP	27	collection_of_sensors
19	set_the_limits	entry_defn	ENT	28	collection_of_sensors
20	for_sensor	sensor_name	EFP	28	collection_of_sensors
21	low_limit	sensor_value	EFP	29	collection_of_sensors
22	high_limit	sensor_value	EFP	30	collection_of_sensors
23	timer	task_spec	TKS	33	monitor_temperatures
24	interrupt	entry_defn	ENT	34	timer
25	high_bound	sensor_value	OBD	38	monitor_temperatures
26	low_bound	sensor_value	OBD	39	monitor_temperatures
27	name	sensor_name	OBD	40	monitor_temperatures
28	user_command	command	OBD	41	monitor_temperatures
29	value	sensor_value	OBD	42	monitor_temperatures
30	alarm	task_body	TKB	103	text_io
31	bits	number	CND	105	alarm
32	words	number	CND	106	alarm
33	light	END	END	108	alarm
34	fault_light	light	OBD	111	alarm
35	limit_check	ATD	ATD	114	alarm
36	out_of_limits_light	limit_check	OBD	116	alarm
37	recording_device	task_body	TKB	135	text_io
38	collection_of_sensors	task_body	TKB	150	text_io
39	bits	number	CND	152	collection_of_sensors
40	words	number	CND	153	collection_of_sensors
41	sensor_record	RTD	RTD	155	collection_of_sensors
42	high_limit	sensor_value	RTE	156	collection_of_sensors
43	low_limit	sensor_value	RTE	157	collection_of_sensors
44	value	sensor_value	RTE	158	collection_of_sensors
45	sensor_group	ATD	ATD	160	collection_of_sensors
46	sensor	sensor_group	OBD	161	collection_of_sensors
47	sensor_set	package_body	NEW	163	collection_of_sensors
48	active_sensors	set	OBD	165	collection_of_sensors
49	sensor_port	ITD	ITD	167	collection_of_sensors
50	sensor_list	ATD	ATD	169	collection_of_sensors
51	sensor_map	sensor_list	OBD	171	collection_of_sensors
52	timer	task_body	TKB	219	text_io
53	minutes	number	CND	220	timer
54	interval	ITD	ITD	221	timer
55	ticks	interval	OBD	222	timer

C.5.2 La table d'identificateurs *entry* et la table des paramètres *entry*

No.	Entry_Name	Line_No	Presence_of_parameter
===	==========	=======	====================
0	post_fault_in_sensor	20	N
1	post_out_of_limits	21	Y
2	disable	25	Y
3	enable	26	Y

4	force_record	27	Y
5	set_the_limits	28	Y
6	interrupt	34	N

No.	Entry_Name	Param_seq_No.	Parameter	Parameter_mode_type
0	post_out_of_limits	1	on_sensor	IN
1	disable	1	sensor	IN
2	enable	1	sensor	IN
3	force_record	1	of_sensor	IN
4	set_the_limits	1	for_sensor	IN
5	set_the_limits	2	low_limit	IN
6	set_the_limits	3	high_limit	IN

C.5.3 La table d'importation

No.	Imported_objects	Importing_Objects	Line_No.
0	text_io	monitor_temperatures	1
1	system	monitor_temperatures	1
2	text_io	monitor_temperatures	2
3	command_io	monitor_temperatures	13
4	sensor_name_io	monitor_temperatures	15
5	sensor_value_io	monitor_temperatures	17
6	system	alarm	101
7	device_io	recording_device	133
8	set_package	collection_of_sensors	148
9	system	collection_of_sensors	148
10	sensor_set	collection_of_sensors	164

C.5.4 La table des objets génériques

No.	Generic_Ident	New_Ident	Line_No.
0	command_io	enumeration_io	12
1	sensor_name_io	enumeration_io	14
2	sensor_value_io	fixed_io	16
3	sensor_set	set_package	163

C.6 La base de faits

```
actual_param([]).
array([]).
array(limit_check).
array(sensor_group).
array(sensor_list).
calls([], []).
constant([]).
constant(bits).
constant(bits).
constant(minutes).
constant(words).
constant(words).
entry([]).
```

```
entry(disable).
entry(enable).
entry(force_record).
entry(interrupt).
entry(post_fault_in_sensor).
entry(post_out_of_limits).
entry(set_the_limits).
entry_act_param([]).
entry_for_param([]).
entry_for_param(for_sensor).
entry_for_param(high_limit).
entry_for_param(low_limit).
entry_for_param(of_sensor).
entry_for_param(on_sensor).
entry_for_param(sensor).
entry_for_param(sensor).
formal_param([]).
function_body([]).
function_spec([], []).
generic([]).
imports([], []).
imports(alarm, system).
imports(collection_of_sensors, sensor_set).
imports(collection_of_sensors, set_package).
imports(collection_of_sensors, system).
imports(monitor_temperatures, command_io).
imports(monitor_temperatures, sensor_name_io).
imports(monitor_temperatures, sensor_value_io).
imports(monitor_temperatures, system).
imports(monitor_temperatures, text_io).
imports(monitor_temperatures, text_io).
imports(recording_device, device_io).
instantiates([], []).
instantiates(command_io, enumeration_io).
instantiates(sensor_name_io, enumeration_io).
instantiates(sensor_set, set_package).
instantiates(sensor_value_io, fixed_io).
is_parameter_of([], [], []).
object_declared_in([], []).
object_declared_in(active_sensors, collection_of_sensors).
object_declared_in(alarm, monitor_temperatures).
object_declared_in(alarm, text_io).
object_declared_in(bits, alarm).
object_declared_in(bits, collection_of_sensors).
object_declared_in(collection_of_sensors, monitor_temperatures).
object_declared_in(collection_of_sensors, text_io).
object_declared_in(command, monitor_temperatures).
object_declared_in(command_io, monitor_temperatures).
object_declared_in(disable, collection_of_sensors).
object_declared_in(enable, collection_of_sensors).
object_declared_in(fault_light, alarm).
object_declared_in(for_sensor, collection_of_sensors).
object_declared_in(force_record, collection_of_sensors).
object_declared_in(high_bound, monitor_temperatures).
object_declared_in(high_limit, collection_of_sensors).
object_declared_in(high_limit, collection_of_sensors).
object_declared_in(interrupt, timer).
object_declared_in(interval, timer).
object_declared_in(light, alarm).
object_declared_in(limit_check, alarm).
object_declared_in(low_bound, monitor_temperatures).
object_declared_in(low_limit, collection_of_sensors).
object_declared_in(low_limit, collection_of_sensors).
object_declared_in(minutes, timer).
object_declared_in(monitor_temperatures, text_io).
object_declared_in(name, monitor_temperatures).
object_declared_in(of_sensor, collection_of_sensors).
object_declared_in(on_sensor, alarm).
```

```
object_declared_in(out_of_limits_light, alarm).
object_declared_in(post_fault_in_sensor, alarm).
object_declared_in(post_out_of_limits, alarm).
object_declared_in(recording_device, text_io).
object_declared_in(sensor, collection_of_sensors).
object_declared_in(sensor, collection_of_sensors).
object_declared_in(sensor, collection_of_sensors).
object_declared_in(sensor_group, collection_of_sensors).
object_declared_in(sensor_list, collection_of_sensors).
object_declared_in(sensor_map, collection_of_sensors).
object_declared_in(sensor_name, monitor_temperatures).
object_declared_in(sensor_name_io, monitor_temperatures).
object_declared_in(sensor_port, collection_of_sensors).
object_declared_in(sensor_record, collection_of_sensors).
object_declared_in(sensor_set, collection_of_sensors).
object_declared_in(sensor_state, monitor_temperatures).
object_declared_in(sensor_value, monitor_temperatures).
object_declared_in(sensor_value_io, monitor_temperatures).
object_declared_in(set_the_limits, collection_of_sensors).
object_declared_in(ticks, timer).
object_declared_in(timer, monitor_temperatures).
object_declared_in(timer, text_io).
object_declared_in(user_command, monitor_temperatures).
object_declared_in(value, collection_of_sensors).
object_declared_in(value, monitor_temperatures).
object_declared_in(words, alarm).
object_declared_in(words, collection_of_sensors).
package_body([]).
package_body(command_io).
package_body(sensor_name_io).
package_body(sensor_set).
package_body(sensor_value_io).
package_spec([], []).
procedure_body([]).
procedure_spec([], []).
procedure_spec(monitor_temperatures, 3).
rec_type_var([]).
rec_type_var(high_limit).
rec_type_var(low_limit).
rec_type_var(value).
record([]).
record(sensor_record).
record_contains_var([], []).
record_contains_var(sensor_record, high_limit).
record_contains_var(sensor_record, low_limit).
record_contains_var(sensor_record, value).
rename_as([], []).
renaming([]).
task_body([]).
task_body(alarm).
task_body(collection_of_sensors).
task_body(recording_device).
task_body(timer).
task_spec([], []).
task_spec(alarm, 19).
task_spec(collection_of_sensors, 24).
task_spec(timer, 33).
type([]).
type_return_value([], []).
uses([], [], []).
uses(active_sensors, assignment_statement, 178).
uses(active_sensors, assignment_statement, 178).
uses(active_sensors, assignment_statement, 182).
uses(active_sensors, assignment_statement, 182).
uses(active_sensors, if_statement, 186).
uses(active_sensors, if_statement, 206).
uses(alarm, function_or_library_call_statement, 210).
uses(alarm, function_or_library_call_statement, 235).
```

```
uses(collection_of_sensors, function_or_library_call_statement, 86).
uses(collection_of_sensors, select_statement, 232).
uses(collectiont_of_sensors, function_or_library_call_statement, 61).
uses(collectiont_of_sensors, function_or_library_call_statement, 67).
uses(collectiont_of_sensors, function_or_library_call_statement, 73).
uses(device_io, function_or_library_call_statement, 141).
uses(device_io, function_or_library_call_statement, 142).
uses(device_io, function_or_library_call_statement, 143).
uses(disable, case_statement, 57).
uses(disable, select_statement, 177).
uses(disabled, function_or_library_call_statement, 193).
uses(enable, function_or_library_call_statement, 61).
uses(enable, function_or_library_call_statement, 67).
uses(enabled, function_or_library_call_statement, 189).
uses(fault_light, assignment_statement, 123).
uses(for_sensor, assignment_statement, 200).
uses(for_sensor, assignment_statement, 201).
uses(for_sensor, function_or_library_call_statement, 87).
uses(force_record, function_or_library_call_statement, 73).
uses(force_record, select_statement, 232).
uses(get, function_or_library_call_statement, 53).
uses(get, function_or_library_call_statement, 59).
uses(get, function_or_library_call_statement, 65).
uses(get, function_or_library_call_statement, 71).
uses(get, function_or_library_call_statement, 77).
uses(get, function_or_library_call_statement, 80).
uses(get, function_or_library_call_statement, 83).
uses(high_bound, function_or_library_call_statement, 83).
uses(high_bound, function_or_library_call_statement, 89).
uses(high_limit, assignment_statement, 201).
uses(high_limit, assignment_statement, 201).
uses(high_limit, function_or_library_call_statement, 89).
uses(high_limit, if_statement, 209).
uses(i, assignment_statement, 207).
uses(i, assignment_statement, 207).
uses(i, function_or_library_call_statement, 210).
uses(i, if_statement, 206).
uses(i, if_statement, 208).
uses(i, if_statement, 208).
uses(i, if_statement, 209).
uses(i, if_statement, 209).
uses(i, loop_statement, 204).
uses(i, loop_statement, 229).
uses(i, select_statement, 232).
uses(interrupt, accept_statement, 226).
uses(is_a_member, if_statement, 186).
uses(is_a_member, if_statement, 206).
uses(log_the_status, accept_statement, 138).
uses(log_the_status, function_or_library_call_statement, 187).
uses(log_the_status, function_or_library_call_statement, 191).
uses(low_bound, function_or_library_call_statement, 80).
uses(low_bound, function_or_library_call_statement, 88).
uses(low_limit, assignment_statement, 200).
uses(low_limit, assignment_statement, 200).
uses(low_limit, function_or_library_call_statement, 88).
uses(low_limit, if_statement, 208).
uses(minutes, if_statement, 228).
uses(name, function_or_library_call_statement, 53).
uses(name, function_or_library_call_statement, 59).
uses(name, function_or_library_call_statement, 61).
uses(name, function_or_library_call_statement, 65).
uses(name, function_or_library_call_statement, 67).
uses(name, function_or_library_call_statement, 71).
uses(name, function_or_library_call_statement, 73).
uses(name, function_or_library_call_statement, 77).
variable([]).
variable(active_sensors).
variable(command).
```

```
variable(fault_light).
variable(high_bound).
variable(interval).
variable(light).
variable(low_bound).
variable(name).
variable(out_of_limits_light).
variable(sensor).
variable(sensor_map).
variable(sensor_name).
variable(sensor_port).
variable(sensor_state).
variable(sensor_value).
variable(ticks).
variable(user_command).
variable(value).
```

Nom : **AJILA**

Prénom : **Samuel Adesoye**

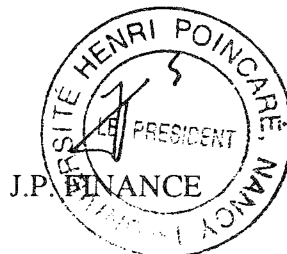
DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 28 JUIN 1995 - n°252

Le Président de l'Université



J.P. FINANCE

3C-F9

UNIVERSITÉ HENRI POINCARÉ · NANCY 1
24-30, rue Lionnois B.P.3069 54013 Nancy cedex
Tél : 83 32 81 81 Fax : 83 32 95 90

// Résumé

Notre travail concerne l'évaluation *a priori* de l'impact des changements réalisés sur les objets logiciels.

Il est parfois difficile de comprendre toutes les conséquences d'un changement et de garantir qu'il n'y aura pas d'effets de bord gênants au niveau de l'ensemble du système logiciel. Il est donc utile d'apporter une aide à la décision de changement permettant de comprendre et d'analyser un changement avant de le faire. En particulier on doit prendre en compte la complexité des liens qui existent entre les différents composants du logiciel.

Pour tenir compte des conséquences d'un changement donné, il faut avoir une connaissance précise sur les liens entre les objets logiciels et analyser l'impact du changement.

Pour cette raison, nous avons étudié les caractéristiques souhaitables d'une analyse d'impact qui permettront de comprendre, de comparer, et d'évaluer des approches différentes et nous proposons un cadre générique d'analyse d'impact. Ce cadre générique comporte trois parties: une approche d'analyse d'impact, la structure de cette approche, et la mesure d'efficacité de l'approche.

Nous proposons ensuite une approche d'analyse d'impact d'un changement et pour cela nous avons :

- défini un modèle générique à base de connaissances sur la nature des liens entre les objets logiciels et sur les règles de propagation du changement;
- validé ce modèle avec un prototype limité à deux étapes de cycle de vie, la conception et le code. Pour cette validation, nous avons choisi la méthode HOOD pour la conception et le langage de programmation Ada pour le code.

Les outils existants d'analyse de l'impact du changement sont basés sur les méthodes de simulation (les méthodes dites *algorithmiques*) et ils n'adressent souvent que le code et parfois même qu'une partie du code. Nous avons montré dans ce travail qu'avec le modèle proposé, on peut dépasser la limite des méthodes de simulation et donner des informations plus détaillées sur l'effet d'un changement.

Mots-clés: Maintenance logicielle, Analyse d'impact, Propagation d'un changement, Objets logiciels, Système de vues, Relations de dépendances, Base de connaissances